Computer Science and Engineering
Ciclo XXXIV

Settore Concorsuale: 01/B1
Settore Scientifico Disciplinare: INF/01

Big Code Applications and Approaches

Presentata da:
Francesca Del Bonifro

Supervisore:
Maurizio Gabbrielli

Coordinatore Dottorato:
Davide Sangiorgi

Esame finale anno 2022

To my family,
Maurice and Q

Abstract

The availability of a huge amount of source code from code archives and open-source projects opens up the possibility to merge machine learning, programming languages, and software engineering research fields. This area is often referred to as Big Code where programming languages are treated instead of natural languages while different features and patterns of code can be exploited to perform many useful tasks and build supportive tools.

Among all the possible applications which can be developed within the area of Big Code, the work presented in this research thesis mainly focuses on two particular tasks: the Programming Language Identification (PLI) and the Software Defect Prediction (SDP) for source codes. Programming language identification is commonly needed in program comprehension and it is usually performed directly by developers. However, when it comes at big scales, such as in widely used archives (GitHub, Software Heritage), automation of this task is desirable. To accomplish this aim, the problem is analyzed from different points of view (text and image-based learning approaches) and different models are created paying particular attention to their scalability.

Software defect prediction is a fundamental step in software development for improving quality and assuring the reliability of software products. In the past, defects were searched by manual inspection or using automatic static and dynamic analyzers. Now, the automation of this task can be tackled using learning approaches that can speed up and improve related procedures. Here, two models have been built and analyzed to detect some of the commonest bugs and errors at different code granularity levels (file and method levels).

Exploited data and models' architectures are analyzed and described in detail. Quantitative and qualitative results are reported for both PLI and SDP tasks while differences and similarities concerning other related works are discussed.

Acknowledgements

I would like to thank my research supervisor Prof. Maurizio Gabbrielli who always supported and guided me during my Ph.D. experience and the production of this thesis.

In addition, I'd like to thank the two reviewers of this thesis Prof. Fernando Martínez-Plumed and Prof. Jacopo Mauro who dedicated their time providing me with very helpful improvement hints.

I express my deep gratitude to the Software Heritage and INRIA organizations, especially Prof. Stefano Zacchiroli, that hosted me as a research visitor and helped in the development of some parts of this work.

Finally, I'd like to thank DISI and ENEA to provide me with the computation infrastructure that the work presented in this thesis required.

Contents

1. Introduction **19**

2. Related Works **29**

2.1. *Big Code* applications . . . 29

2.1.1. Models evaluation . . . 36

2.2. Programming Language Identification . . . 38

2.2.1. File-type Identification . . . 41

2.2.2. Image-based Programming Language Identification . . . 42

2.3. Software Defect Prediction . . . 43

2.3.1. Learning Approaches . . . 45

2.3.2. Defect Prediction aspects . . . 50

3. Programming Language Identification **65**

3.1. File Extension Identification . . . 68

3.1.1. Preprocessing . . . 69

3.1.2. The model . . . 73

3.1.3. Results . . . 77

3.1.4. Threats to Validity . . . 81

3.1.5. Discussion and future work . . . 82

3.2. Image-based Programming Language Identification . . . 85

3.2.1. Data preparation . . . 85

3.2.2. The model . . . 86

3.2.3. Scrambling . . . 89

3.2.4. Results . . . 90

3.2.5. Classification Results . . . 91

3.2.6. Threats to Validity . . . 93

3.2.7. Discussion and future works . . . 94

3.3. Conclusions about the Programming Language Identification task 98

4. Software Defect Prediction **101**

4.1. Dataset . . . 103

4.2. Representations and Models . . . 108

4.2.1. Code2Vec . . . 109

4.2.2. Infercode 117
4.3. Conclusions about the Software Defect Prediction task 123

5. Conclusions **127**

A. Results for File Extension Identification **131**

B. Results for Image-based Programming Language Identification **135**

C. Issues managed by Infer **141**

List of Figures

3.1. Extensions distribution within the dataset corpus. 72
3.2. File type classification model. 73
3.3. Word cloud for token frequency distribution. 75
3.4. Source lines of code extraction from source code bundles and their rendering. 87
3.5. AlexNet architecture, the simplest pre-trained CNN among those we specialized for visual code recognition 87
3.6. Java code snippet in original form v. several scrambled variants 90
3.7. Class Activation Map (CAM) heat-maps for selected code snippets in various languages, for the ResNet-based classifier. . . . 96

4.1. Data distribution. 106
4.2. Functions distribution. 108
4.3. Portions of *Astminer* output files generated for the `server.c` source file. 111

List of Tables

2.1. Papers about the PLI task summarization. In the Dataset column, GH and SO stand for custom extractions from GitHub and StackOverflow, respectively while RC stands for Rosetta Code. 39
2.2. Papers about the SDP task treated with deep learning approaches. 55

3.1. Average performance of the encoder architecture without and with trigrams. 78
3.2. Extension confusion groups 80
3.3. Average performance of the bigram model before and after the re-training phase on the 2019 testset 81
3.4. Average training times per epoch (minutes and seconds), number of epochs and total training times (hours, minutes and seconds) for AlexNet (A), MobileNet (M) and ResNet (R) -based models for the 2 considered steps of training. 89
3.5. Average performance of the models for image-based PLI 91
3.6. PLI performances with and without scrambling. From left to right: no scrambling (Orig), scrambling of alphabetic characters (A), digits (N), symbols (S), combinations of them (AN, AS, NS, ANS) and substitution of all non-blank characters with `x` (X). 92

4.1. Issues' popularity within the dataset 104
4.2. Files distribution among classes. 105
4.3. Functions distribution among classes. 107
4.4. Results for code2vec based models with 30-dimensional code vectors. 114
4.5. Results for code2vec based models with 50-dimensional code vectors. 115
4.6. Results for each class of the two best models. 115
4.7. Confusion matrix at best epoch for model [M1] 117
4.8. Confusion matrix at best epoch for model [M2] 118
4.9. Confusion matrix for binary [M1] 118
4.10. Confusion matrix for binary [M2] 119

4.11. Results for Neural Networks. 121
4.12. Results for Encoder-like models. 121
4.13. Results for Random Forests. 122
4.14. Results for Support Vector Classifier. 123
4.15. Results for each class of the selected SVC model. 123
4.16. Results for each class of the selected Neural Network model. . 124
4.17. Results for each class of the selected Encoder model. 124
4.18. Results for each class of the selected Random Forest model. . 125

A.1. Performance of the encoder architecture without and with tri-grams. 134

C.1. Issues detectable using *Infer* 143

Acronyms

ACG Aggregated Call Graph

AI Artificial Intelligence

ANN Artificial Neural Network

ASCII American Standard Code for Information Interchange

AST Abstract Syntax Tree

AUC Area Under Curve

BERT Bidirectional Encoder Representations from Transformers

bi-LSTM bidirectional Long Short Term Memory

BLEU Bilingual Evaluation Understudy

BOW Bag Of Words

CAM Class Activation Map

CBOW Continuous Bag Of Words

CC Clean Code

CFG Control-Flow Graph

CNN Convolutional Neural Network

CWE Common Weakness Enumeration

DBN Deep Belief Network

DFD Data Flow Diagram

DL Deep Learning

DNN	Deep Neural Network
DP	Defect Prediction
DS	Dead Store
F	F-measure
FCL	Fully Connected Layers
FN	False Negative
FP	False Positive
FPR	False Positive Rate
FTI	File-Type Identification
GAN	Generative Adversarial Network
GAT	Graph Attention Network
GCN	Graph Convolution Network
GH	GitHub
GIN	Graph Isomorphism Network
GNN	Graph Neural Network
groum	Graph-based Object Usage Model
GRU	Gated Recurrent Unit
HAN	Hierarchical Attention Network
IDE	Integrated Development Environment
JIT	Just-In-Time
KNN	K-Nearest Neighbors
LIME	Local Interpretable Model-Agnostic Explanations
LOC	Line Of Code

LR	Learning Rate
LSTM	Long Short Term Memory
ML	Machine Learning
MNB	Multinomial Naive Bayes
ND	Null Dereference
NLP	Natural Language Processing
NTM	Neural Translation Model
OBOE	Off-By-One Error
OCR	Optical Character Recognition
OOV	Out Of Vocabulary
P	Precision
PHOG	Probabilistic Higher-Order Grammar
PLD	Programming Language Detection
PLI	Programming Language Identification
PN	Pointer Network
PRNG	Pseudo-Random Number Generator
R	Recall
RC	Rosetta Code
ReLU	Rectified Linear Unit
RF	Random Forest
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic

SDP	Software Defect Prediction
SGC	Simplified Graph Convolution
SH	Software Heritage
SHA	Secure Hash Algorithm
SHAP	SHapely Addictive exPlanation
SLOC	Source Line Of Code
SMOTE	Synthetic Minority Oversampling Technique
SO	StackOverflow
SVM	Support Vector Machine
TBCNN	Tree-Based Convolutional Neural Network
TN	True Negative
TP	True Positive
TPR	True Positive Rate
UV	Uninitialized Value
VCS	Version Control System

Chapter 1. Introduction

Artificial Intelligence (AI) [156] is a huge and growing field that refers to many heterogeneous techniques and tools that are developed to support and ease human activities in a variety of fields such as industry, education, health, entertainment, and so on.

The AI field includes many approaches which could be very different from each other. The two main sub-fields of AI can be identified in *symbolic* and *subsymbolic* (or *connectionist*) classes of methods.

Symbolic AI is based on logic and rigorous formalism and attempts to reproduce human reasoning focusing on knowledge representation, logic, and search. This is the case of *knowledge based* agents and *expert systems* [156] which embody knowledge in rules and facts and exploit them for various purposes. Interpretability, reasoning, and inference are the most important features of such systems, but they lack the capability of dealing with information encoded into noisy data, managing uncertainty and learning. On the other hand, the field of *subsymbolic* AI contains various *data-driven* techniques that exploit statistical properties and methods to analyze and learn from collected data about a certain phenomenon. These approaches extract knowledge directly from data even without any prior knowledge but often they lack interpretability of the extracted knowledge and their general behavior.

Sometimes, when one refers to data-driven learning approaches applied to relatively large datasets the term *Big Data* [140] is used, which emphasizes the sizable volume of the managed data.

One of the various areas in which data-driven techniques can be applied is represented by the so-called *Big Code* field. Similar to *Big Data*, *Big Code* proposes to apply learning and mining techniques to a big amount of a particular kind of data which is represented by source code.

The opportunity to work with this kind of data has been opened thanks to the huge availability of Open Source projects and the existence of many public source code archives and repository hosting platforms, e.g. *GitHub* [68], *Software Heritage* [79, 50], etc.., which are growing and affirming their importance among developers' communities and institutions. The presence of these archives has made it possible to use the code that is hosted by them as code data examples permitting the building of huge datasets that can be mined and analyzed similarly to what is done for other kinds of data. In this way, it is possible to apply techniques and concepts developed within the *Big Data* field to source code data in order to learn from existing software and use the extracted knowledge for different purposes, i.e., research analysis, study, or building various kind of supporting tools which can be exploited in software production processes to ease and speed up the development and maintainability of new projects [8]. For example, common applications of *Big Code* are bug detection, code clone identification, code summarization and captioning, and also code generation and completion tools [8, 171, 6, 86]. A more precise description of the applications of *Big Code* will be discussed in Chapter 2.

Most works performed within the *Big Code* field exploit methods and concepts developed in the *Natural Language Processing* (NLP) area. In fact, from a certain perspective source code can be viewed as a textual production written in a particular language, i.e., a programming language. The use of NLP techniques is suggested and supported by the so-called *Naturalness Hypothesis* [8] which states that software corpora is a particular form of human communication and the properties which are statistically found within it are similar to the ones found in natural language corpora. This idea has been extensively applied and many satisfying results have been achieved in this way. Beyond that, programming languages that are used to produce code have the peculiarity of being composed of rigid rules and syntax which compose the grammar of these languages, this causes the structure of the code to be less flexible than texts produced using natural language. In this way, also other kinds of features can be exploited when this kind of data is analyzed in order to optimize the information content. For example, beyond textual representation, source code can also be represented in the form of an *Abstract Syntax Tree* (AST) which is built by means of the rules which define the grammar of the used programming language. This enables the use of grammar-driven approaches which are different from the NLP techniques tailored to manage plain texts.

Many other representations of code exist and can be used in this field, the choice can be done based on the aimed task, the model requirements, and so on, in fact, different code representations deserve to be analyzed in order to find the most suitable one for a given task or to appropriately integrate them in a unique model.

The work presented in this thesis has been inspired and made possible thanks to the collaboration with *Software Heritage*[1] [50] which is the biggest available Open Source code archive and aims to collect any software artifact produced in the human history. This archive guarantees a rich availability of code data and permits to deal with different code styles and languages both because the stored code has been created in very different environments and because of the natural evolution of languages, techniques, and habits during the years as the archive stores software produced during the last 70 years. This permits a wider range of possible analysis than in other platforms whose content is mostly about recently emerging trends and projects.

In the following, we focus on two main tasks among the many possibilities that can be found within the *Big Code* field, i.e., Programming Language Identification (PLI) and Software Defect Prediction (SDP).

The importance of the first task, the Programming Language Detection, is due to the central role played by programming languages in software artifacts and there could be peculiar situations in which the used language is not known and it is desirable to identify it. In software engineering research, "programming language" is a common variable to correlate against—researchers have measured language productivity [119, 118, 152], trends [146, 181], usability [123], and code quality [97, 149, 15], to name just a few. Developers would know by heart the language their code is written in, and would also easily identify the main programming language used in a given codebase. However, when it comes to *large* codebases, which often mix and match multiple programming languages; or when it comes to very large-scale analyses [53, 116], encompassing entire collaborative development forges if not the full body of publicly available source code [2], manual detection is not an option. In these cases, researchers tend to rely on either off-the-shelf language detection tools (e.g., *GitHub*'s *Linguist*[2] [68, 69]), or on metadata exposed by the code hosting platforms [92], which were in turn extracted using some of the just cited available tools.

In this work, different approaches have been attempted in this direction in order to study the problem from different points of view.

The first attempt is described in Section 3.1 and in [47], it focuses on the prediction of files extensions for textual files commonly found in software version control systems repositories based solely on file contents. Here, the word "solely" is interpreted in a strict sense, depriving us of the use of any *a priori* heuristic or information on the content of the analyzed files. The detection model does *not* know keywords in the grammar of any given programming language, nor shebang (`#!/usr/bin/perl`) or editor mode lines

[1] `https://www.softwareheritage.org/`
[2] `https://github.com/github/linguist`

(`%% mode: latex`) which are usually written in code files, i.e., this attempt focuses on the intrinsic recognizability of code file types, and programming languages. This information is not used by our model as it could dramatically bias the model and bring confusion when it is not available (which could often happen in large-scale code archives) as the trained model could learn to mostly focus on this piece of information rather than the whole content (see the *Linguist* case [64]). Also, we chose to focus on the file content as intuitively the programming language used in a code file both influences and emerges from the content structure and organization.

File *extensions* are highly predictive of programming languages and, more generally, file types. Most of the available language detection tools tend to heavily rely on them to determine file types [64]. While very effective in the general case, doing so is problematic when file extensions are either not available (e.g., code snippets posted on the Web or embedded in the document, executable scripts, etc.) or wrongly assigned (either on purpose or by mistake) because the identification tool highly relies on the information brought by the extension and their performance usually drops [64].

As it is becoming increasingly more possible [50, 164] to analyze historically relevant software source code dating back several decades, heuristics built today are prone to failures when applied to “old” code, and heuristics spanning several generations of file formats will be fragile and hard to maintain in the long run.

After the implementation of some data preparation techniques, a simple neural model is proposed to make predictions about the file extensions. The proposed model is based on tokens and 2-grams frequencies which are encoded into vector representations and have an encoder-like structure as the number of neural units contained in the layers progressively shrinks. This approach also focuses on the number of recognized languages which makes predictions harder as it increases, and we want to build a model characterized by scalability as the usage of such a tool should be applied to very large codebases which potentially contain programs written in a high number of programming languages. Arguably, recognizing a handful of programming languages or file types could be approached with simple heuristics without incurring the maintenance overhead of (re-)training machine learning models. However, hundreds of programming languages exist in the wild and sometimes they exhibit only subtle syntactic differences from each other. Also, they evolve over time, slowly but regularly [159]. It is at such a scale of diversity that PLI approaches based on machine learning would be most useful

The second approach that has been attempted is described in Section 3.2 and in [54]. This time the prediction is done directly at the programming language level instead of file extensions and the ground truth information about languages is assumed to be the one predicted by the *Linguist* tool [69].

This method uses a completely different representation of the input data with respect to the previously introduced work, i.e., programming language identification is treated from an artificial vision point of view. Image-based Programming Language Identification is the other of the two major classes of ML approaches that have been used [95].

Image-based PLI models are currently capable of recognizing a limited amount of different programming languages, with a maximum "diversity" of 10 languages found in the literature [95, 82] while, as we already pointed out we also want to focus on the number of programming languages among the predictions that can be done.

This approach studies if it is possible to *visually* recognize that many programming languages (in the order of hundreds) from source code snippets images without any *a priori* knowledge about the languages.

Another contribution of this work is represented by the assessment of *what* actually allows image-based ML models to visually recognize programming languages, especially at this scale of language diversity as, to our knowledge, this aspect has never been investigated in the literature. Such knowledge would allow the future to specialize in recognition networks and improve performances. This aspect is analyzed by randomly scrambling different character classes (alphabetic characters, decimal digits, symbols) and comparing PLI efficacy. We show that symbols are the characters that contribute the most to visual recognition of code snippets (halving precision when scrambled), followed by far by alphabetic characters, with decimal digits and indentation having a negligible impact on the visual detection of programming languages.

The proposed model relies on Convolutional Neural Networks (CNNs) which have been pre-trained on generic images and subsequently adapted to PLI using the transfer learning technique. We validate the approach on the same real-world dataset that has been used in the first presented approach and consider 149 different programming languages. The prediction capabilities are evaluated using three different classifiers, each based on a CNN which are pre-trained for image recognition: a Residual Network (ResNet) [78], MobileNetv2 [83], and AlexNet [101].

The second main point of interest in this thesis is the Software Defect Prediction (SDP) task. Software defects are responsible for many kinds of programs' unwanted behavior and decrease the software products' quality and reliability. To mitigate the impact of potential bugs, a fundamental phase in software production is represented by code testing, and review. However, these activities are highly time-consuming and very expensive. This naturally brought to the development of automatic supporting tools such as static and dynamic analyzers which helps in the detection of bugs and errors within code. Static and dynamic analyzers are often slow, and their predictions are not always accurate, for these reasons as in other fields, the availability of many machine learning models which exploit statistical properties of data to learn and approximate tasks can be adapted to this particular problem. In fact, many works have been done in this direction bringing promising results.

The first attempts of learning-based defect prediction models relied on traditional machine learning exploiting traditional code metrics usually evaluated when defect proneness has to be assessed. For example, some common metrics used in code analysis are complexity metrics such as McCabe [120] and Halstead [75] features. However, these kinds of approaches do not capture syntactic, semantic, and context information that can be extracted from source code with other more advanced methods. This is the case of deep learning models, which are able to process data and automatically extract relevant information from them (feature learning) without the explicit usage of field-related knowledge. Many of these models have been analyzed with respect to the defect prediction task and brought interesting and promising results.

During literature study about this task a particular aspect emerges: almost every work for defect prediction is represented as a binary classification in which a given piece of code is evaluated and predicted to be defective or clean, i.e., no other kind of information about the defect found in the code are mentioned. For example, static code analyzers usually provide the user with a detailed list of potential bugs, containing the kind and location of the predicted bugs. For this reason, the work reported here is developed in the direction of predicting defects and, if they are detected, the kind of bugs that they represent.

The approach presented in this thesis models this problem as a multi-class classification problem in which the classes are a limited number of bug types. The dataset analyzed here is presented in [65] and is composed of C and C++ open-source projects which are processed and labeled by a static analyzer. Of course, using this dataset limits the capabilities that can be reached by our model as it can at most imitate the static analyzer behavior. However, we aim to build a more efficient tool while keeping reasonable bug detection capabilities.

Two code representations are studied in this work: *code2vec* [10] and *Infercode* [25]. Both approaches are based on the exploitation of the Abstract Syntax Tree (AST) structure that can be used to represent codes when the language is known.

Code2vec [10] describe a valuable technique to generate code vector representations that are originally used in the method naming task, i.e., to predict the names of methods from the contents of their bodies. As described in [10], this representation exploits paths extracted from code ASTs which are sequences of nodes encountered while traversing the tree structure in the *up* and *down* directions going from one starting token to another token. A collection (bag) of these paths is collected and processed by Attention mechanisms to build the code vectors. The representation obtained in this way can also be used in different tasks and, as *code2vec* showed state-of-the-art capabilities also in tasks different from the method naming one we want to assess if it can be suitable for the defect identification within code files. As *code2vec* is designed for Java projects another tool is used to generate the *code2vec* input constituting a suitable tool-chain. The tool is known as *Astminer* [100] and has been built to extend the *code2vec* applicability to many other languages, including C and C++.

The other representation strategy is given by *Infercode* described in [25]. This approach defines a code representation based on ASTs which is task agnostic, in fact, the tool just provides the user with the code vectors and does not require the final task knowledge to work properly. The code representation is computed by training the model in predicting sub-AST which are used as labels in the learning phase. This task agnostic feature is interesting, and it is worth investigating if it is suitable for the defect identification task. A problem that has been found with the usage of this tool is represented by its limited scaling capability. In fact, its application at file-level granularity resulted to be mostly infeasible. To overcome this limit we change here the granularity level at which the predictions are performed and the previously used dataset needs to be modified in such a way that we end up with a collection of functions labeled with the eventual bugs found in each of them.

Research Objectives

The main purpose behind the *Big Code* research is to exploit the huge source code data availability enabled by the massive usage of open repository archives to build statistical and learning-based models that can serve as support in several large-scale tasks. In fact, many code management aspects become intractable by means of manual human action when it comes to *huge* amount of projects, repositories, and so on. In this view, automatic tools can perform the desired tasks completely replacing the human work or just support, enhance and facilitate it.

The aims of the works presented in this thesis share this purpose and focus on the particular tasks of automatic programming language identification and software defect prediction. Our works go in the direction of increasing models' scalability and classes coverage without neglecting the aspect of models' simplicity in view of actual future deployment in real-world situations also looking at models' maintainability in the long run.

Thesis Structure The content of this thesis is structured as follows: Chapter 2 contains the description of some works developed within the *Big Code* field mainly focusing on the two tasks that we analyzed in more details, i.e., PLI (Section 2.2) and SDP (Section 2.3), also reviewing some useful code representation models.

Chapter 3 describes our two approaches to the PLI task starting with Section 3.1 which illustrates in detail the related problem of predicting file extensions for textual files exploiting only their content, the data processing, the model choices, and design that we propose to approach this task. The recognized classes are 133 and results for each of them are provided in this Section.Section 3.2 investigates the PLI problem from the image representation point of view describing the data treatment and image generation processes together with the implementation, training, and evaluation details of the prediction model. Here the recognition task focuses on 149 different classes and the results for each of them are reported. Moreover, this Section also reports the analysis of the most relevant visual features and character symbols with respect to our model.

Chapter 4 focuses on the SDP topic and describes the approaches attempted by us. Particularly, Section 4.1 contains a detailed description of the dataset and data pre-processing together with the bug classes distribution analysis and selection for our prediction model, i.e., `Null Pointer Dereference`, `Dead Store` and `Uninitialized Value`. Section 4.2.1 contains the presentation of the approach based on the customized version of the *code2vec* code representation adapted to the file-level bug identification task. Results for each of the three selected bugs are discussed in this Section showing an encouraging scenario for future research in this direction. Section 4.2.2 describes the models based on the *Infercode* code representation for the function-level bugs identification illustrating design, training and evaluation strategies. Similar to the previously introduced sections, we report here the obtained results for each bug class. This approach showed poor results, however, most of the problems in the evaluation step arise because of the extreme imbalance between defective and clean code classes' popularity, i.e., the number of examples for the considered classes.

General conclusions and discussion are treated in Chapter 5 where the main problems and gains encountered performing the illustrated works are depicted together with some ideas for future works.

Some parts of this thesis are extracted from the published papers [47, 54].

Chapter 2.

Related Works

The purpose of this Chapter is to present some of the existing techniques and the state-of-the-art for the *Big Code* area and the main approaches (representations, embeddings, model architectures, etc...) used in literature and developed during the years to build and improve supporting tools, especially in the software engineering context for developers while producing code.

The following of this Chapter is organized into three main parts: in Section 2.1 we present several tasks, approaches, and applications that emerged within the *Big Code* area from a general point of view to give an idea of the possibilities offered by this field; Section 2.2 focuses on the presentation of the literature about the Programming Language Identification task while Section 2.3 focuses on the literature of the Software Defect Prediction problem.

2.1. Big Code applications

As extensively explained in [8], code can be viewed as a form of linguistic communication and many tasks can be performed on code data treating them similarly to natural language texts and simply applying well-known Natural Language Processing (NLP) techniques to them. Besides, this is not the only possible representation for source code, in fact, there are other features that codes have and that they do not share with natural language, for instance, a given piece of code can be represented by its *Abstract Syntax Tree* (AST) which comes directly from the formal nature of the grammar rules that define the programming language used to write it.

One of the first and most common applications developed within the Big Code community is represented by the code completion task, where a model is created in order to be able to suggest possible code tokens during the code is being written. Many Integrated Development Environments (IDEs) already perform a similar function, but they often have several limitations, for example dynamically typed languages are less supported than statically typed ones, token suggestions are based on a high number of handwritten rules and predictions are often context-independent which is an additional limitation on the achievable accuracy of the proposed suggestions.

Integrating classical IDEs' suggestion systems with intelligent supports has been widely studied and different representations of code have been proposed. For example, the first works exploited simple n-grams models, while more recent works started using more complex learning algorithms, grammar-based models, or some combination of these approaches, in order to improve the overall model performance. Tung Thanh et al. [179] propose a code representation dependent on semantics, in which each token in the code is described by its own ID coupled with additional information about both the token itself (context) and the global code functionality (topic). Bielik et al. [17] present the code completion task for the JavaScript language case. The work describes a new approach, the use of a Probabilistic Higher-Order Grammar (PHOG) language model using a data-driven learning approach, in fact, the final model is able to automatically learn the grammar production rules of the language which are designed as context-dependent by means of context parameters and rules weights which represent the probability (score) of a certain rule, model parameters are learned during the training phase as usual. For the same dataset, comparable results are obtained in [109] where, given the AST representation of the code, a sequence of grammar terminal/non-terminal (T, N) pairs is obtained. Each pair (T, N) has its own embedding resulting in a meaningful vector representation for that pair. The sequence of the embedding vectors is processed by an LSTM architecture which is usually exploited to treat sequences of variable length. An LSTM model is also exploited in [171] for code completion in the case of the Python language, with particular care about decreasing learning computational expenses by splitting the training tasks into parallel sub-tasks. Instead, Li et al. [106] performs the code completion task for both JavaScript and Python codes, it has been built using both a classical attention algorithm and a less common architecture which is known as Pointer Networks introduced in [185]. This is basically a modified version of the Attention technique which is particularly suitable when dealing with problems in which the vocabulary of the output sequence cannot be fixed *a priori* but it depends on the input sequence itself. Li et al. [106] use an AST code representation which is then passed to the Attention and Pointer Network architectures. The Pointer Network (PN) is also able to compute longer term-context dependencies than in the case of classic Attention architectures. This happens because the internal memory vectors used by Attention mechanisms are highly related to the context definition, i.e., they have fixed length, while PN is more flexible. A sparse PN model on AST Python code representation is also used in [16] where a comparison with n-grams, classical Attention, and neural language model is developed showing how Attention and PN are the best approaches for this task, especially for what concerns the identifiers prediction. In fact, identifiers are usually the tokens that are often excluded from the *a priori* defined vocabularies. All these works made for the code completion task are designed for Java, JavaScript, and Python projects which are three of the most popular languages, for this reason, there is a high availability of projects for these languages which permits extensively training and testing of these models.

Another application commonly developed in Big Code is naming or code summarization. The aim of the models here is to link code texts to related natural language descriptions, captions, or names. Naming models are designed to suggest suitable names for functions or code snippets based on their semantics. This is helpful while developing software in order to make the produced code easily readable and maintainable. Allamanis et al. [6] develop a model that assigns embeddings, i.e., vectors, in a high dimensional space to methods and classes names defined in a given piece of code. The embedding should be built in order to capture the semantic meaning, e.g., names that refer to a semantically similar concept are closer than others that represent different concepts given a distance in the vector space. This work is presented a log-bilinear neural language model which exploits local and non-local contexts for the name suggestion. In the bilinear context models, every token has its own two representations in high-dimensional space: one refers to the token as a target and the other refers to the token as a context component. Embeddings are learned from training data and the target suggestion is computed using the cosine products between the target's representations and the computed context representations (if the target is present in multiple contexts the context representation is an average of all of them). The more the cosine product approximates the value of 1, the more the considered target is appropriate for that token. Allamanis et al. [6] also present a variant of the bilinear model based on the use of subtokens which make it possible to handle neologisms[1]. This is one of the first works which exploits long-distance relations for the method/class naming task.

Liu et al. [110] describe another approach for the naming problem. In this work, for each method, two kinds of embeddings are computed: one for the tokenized method's name and another for the method body. The first embedding is based on the Paragraph Vector algorithm [102] while the bodies embeddings are developed by using the *Word2Vec* algorithm [125] on the method body and passing the vectorized output to a CNN architecture

[1]Neologisms are also important in mining software in a long-term perspective.

which is used to create the embedding vector[2]. Given a piece of code, it is possible to compute the embeddings and, studying the neighborhoods of the two embeddings, inconsistency in naming can be eventually detected and better suggestions can be provided.

Allamanis et al. [7] and Iyer et al. [86] use CNN and LSTM on top of Attention architectures respectively to suggest an appropriate summary to a given input code snippet. The model in [7] has been considered the state-of-the-art for this task for years and it is used to compare other methods' effectiveness for this task.

An interesting approach is the one developed in [10] which is based on the so-called Path Attention network. This kind of network exploits the AST structure of the code and analyzes the various possible paths in them. A given AST path has also an associated path context which is exploited in the present model. The architecture presented in this work is a particularly effective choice to combine multiple context vector representations into a single embedding vector. This final vector is obtained by means of an attention algorithm and will represent the piece of code that produced the AST in the following. In fact, the embedded representation of the code is used to predict the suitable name for the input code snippet. Words and names are represented by embedded tags and the probability distribution for the whole name suggestion are obtained by means of the normalized dot product of the code vector and these tags vectors.

Xu et al. [197] develop a model which focuses on the code structure looking at the code as a tokens sequence and not using the AST structure. It uses a hierarchical attention mechanism [201] to encode the variable-length token sequence inputs into a fixed-length representation exploiting the hierarchical structure of code snippets in which sets of tokens constitute blocks and sets of blocks form methods. The encoded sequence is processed by a GRU architecture [35] which permits to predict the sequence of tokens that should constitute the predicted method name. The training of the two parts is performed jointly. The model correctness is checked by comparing results to the [7] model which is outperformed. Also, the model in [204] resulted in performing better than [7] in the most complex situations. Here, a novel kind of approach is presented. The methods embeddings are computed based on a simplified version of the Call Graphs which is preferred for this situation because of the enhanced efficiency in the case of graphs with many nodes, the Aggregated Call Graph (ACG). This structure represents the flow for methods calls (callees) in the body of a given method (caller). The callee-caller relationships will determine

[2]Both Paragraph Vector and *Word2Vec* algorithms are widely used in NLP applications.

the methods' embeddings. After the training phase on a train code corpus, the model is used to suggest the name for a method whose body is provided as an input by the developer at query time.

The linkage of the semantic meaning of code to its natural language description is not only applied in naming models but also in the more general code summarization task. In this field, reinforcement learning has also been attempted and it seems to gain good results in code descriptions generation. For example, Wan et al. [188] encode a given code snippet in two ways, one based on token sequence and the other via AST structures using separate LSTMs. An additional attention layer is used to merge the two encoding results and pass the representation result to a deep reinforcement learning architecture. This approach brings quite good results and it is one of the few attempts to treat code using a reinforcement learning framework. This joint analysis of code as a tokens sequence and a graph structure for summarization task is also exploited in [62] in which a sequence-based embedding is used together with a graph neural network (GNN) architecture in order to capture code information from both the analyzed structures. Also, LeClair et al. [103] use two separate attention architectures which treat code as text and AST respectively, the resulting representations are used to find the most suitable description for the provided code content.

Another application of Big Code is in improving translations of a code written in a programming language to another one or from a certain version of a language/library to a different one permitting automatic migration. Usually, these migration support tools were designed by handwritten rules which map expressions of a language to the correspondent ones in the other language. Statistics and learning algorithms applied in big code permit to build new tools decreasing the need for human efforts in writing matching rules. For instance, Nguyen et al. [130] develop a model based on statistical alignment[3] [115] for APIs usage sequences in two languages (Java and C#) in order to mutually map them. This is done without the need for human intervention in the mapping procedure but it leverages a statistical algorithm for sequence-to-sequence alignment (phrase-based model [99, 98]) to provide the results. The sequences that are treated here are extracted from a certain graph which reflects actions and control points flow within code, the graph structure is called *groum* (graph-based object usage model) and its building strategy is defined in [131].

S. et al. [157] present a study of the application of various phrase-based

[3]Statistical alignment is a task which is usually exploited in genomics but which is possible to adapt to different situations for aligning pairs of related sequences.

models which have been successfully exploited in natural language translations to the case of code translations (from C# to Java as well) in order to monitor their value in this field. The analysis starts using a base data-driven model which then is modified by integrating some rules which permit the incorporation of explicit grammar constraints to improve the model performance. The authors start building a phrase table, i.e., a table in which pairs of phrases (c_1, c_2), one written in the input language and the other in the target language, are stored together with the correspondent probability $p_{1,2}$ that the phrase c_2 in Java is the actual translation of the phrase c_1 in C#. The phrase table is used together with a language model of the target language, i.e., a model which assigns high probability to sequences of words that are considered correct sequences in that language and low probability to wrong sentences. The translation is found using the information contained both in the language model and the phrase table as features and weighting them in a sum. Once the model is trained and its parameter is learned, given a phrase in the input language c_1, the target sequence c_2 which maximize the weighted sum results to be the correct translation of c_1. To produce the sequence c_2 there is an algorithm that gradually generates pieces of the sequence until all tokens from c_1 are translated. To improve this process, language formal rules are added to the model. This guides the sequence generation and reduces the search space in which the algorithm searches for the prefixes. In the end, they also included in the model some custom rules about pairing syntax trees from the source to the target language and mapping the non-terminals of these trees. During the evaluation, the best model results are the ones that incorporate both grammar and custom rules. Also, some important remark emerges, for example, the fact that switching from the natural language to the programming language area some scores measures, e.g., BLEU score, do not represent a meaningful performance measure anymore even if it is widely used in the NLP field. In fact, they also take into account parse and compile rate[4] as quality indicators.

Machine Translation Models (NTM) are also exploited for other purposes, for example, they can be used to learn code changes in software development projects. For example, this particular kind of application has been performed in [178] and [29] where classic NLP neural machine translation models are used to capture code templates and changes. In the first work, a NTM is trained using pairs of methods that changed after a pull request as training examples. In this way, the model learns to translate a method from the version which is antecedent to a pull request to the updated version which should be obtained after it. A RNN encoder-decoder architecture is used and the next

[4]The percentage of sequences which parse and compile in the target language respectively.

token suggestion for the decoding step is done by using the beamer search method already used in [182]. In this work, the author highlights how it is possible for a NTM to learn code changes even when the pull requests provided in the dataset are highly heterogeneous. This suggests the possibility to use such a model even in very different projects.

Chakraborty et al. [29] also use the tree structure of the code in order to predict the code changes. More precisely, the code prediction in the form of tokens sequence depends on the probability distribution for the tree structures and another distribution which guides the token generation. The first distribution represents the tree translation model, a tree is built applying a sequence of grammar rules of the given language, the sequences of the rules used to build the tree are passed to an LSTM encoder-decoder architecture in order to learn which tree structure (corresponding to new code version) is likely to be produced starting from another one (corresponding to the old version). The other probability distribution represents the token generation model which supports the generation of the new code as a tokens sequence starting from the old sequence and the new tree structure predicted from the previously described tree translation model. This model is built by using an LSTM architecture similar to the one used in the previous tree translation model. The evaluation given in the paper seems to suggest that tree-based models can have higher performances than sequence-to-sequence models.

Beyond these applications, Big Code has also been useful in the development of tools for bug and vulnerability detection in code. Some examples can be found in [155] where there is an exploration of the capabilities of Machine Learning algorithms for the vulnerability detection task. The authors used a fixed-length embedding for tokens in codes and then they used both CNN and RNN architectures on the embedded representation of codes to automatically extract meaningful feature vectors. These vectors are then passed to a dense layer and used for the actual classification. All the model versions tested in the paper showed the effectiveness of applying these methods which are usually exploited in NLP for the vulnerability detection task on codes. A different architecture is developed in [77], in which a model based on a 1-dimensional CNN using a Wasserstein GAN loss is used to detect and correct vulnerabilities in C and C++ codes. The model presented in the paper can also be applied in cases in which there are no labels for the training phase which represents the main advantage of the model.

Code clone and plagiarism detection is another popular application that took advantage of the power offered by Big Code. Büch and Andrzejak [27] present one of the most recent works for this application, here the AST code representation is used and its nodes are mapped into suited vector representations (embeddings) which are provided to a particular LSTM model developed to process tree structures instead of tokens sequences. The training procedure is performed by processing pairs of ASTs exploiting a Siamese Network [22], i.e., two identical networks with shared weights, in order to compute the similarity of the two code components of each pair trying to maximize the cosine similarity when the two components are similar to each other and minimize it when this is not the case.

One of the most challenging issues in this field is the Out Of Vocabulary management. This is a well-known problem defined in the NLP area which is even more severe here. In fact, a code can contain almost any character sequence as any of them can be used as identifiers (except for reserved language keywords). For this reason, some works are also going in the direction of finding smart ways to manage this situation for example by presenting methods to define *good* vocabularies [13], models on unbounded vocabularies [41] and so on.

Beyond the actual applications, a quite general trend in Big Code is about not only treating code like a particular kind of textual communication which enables the use of common NLP techniques on source codes but exploiting the AST and/or graph representation of code as well as the grammar rules which define a certain programming language. In fact, using the statistical approaches together with precise language rules can bring disambiguation in many cases and an improvement in the models' performance and efficiency.

As the two tasks on which we focus in this research thesis are Programming Language Identification and Software Defect Prediction the following of this chapter focuses on the description of the state-of-the-art of these two particular applications describing other works found in the literature which focus on these topics. Before the presentation of these works we briefly illustrate here some of the most useful metrics which will be used to evaluate the models that we are going to describe.

2.1.1. Models evaluation

Unbalance among classes is a very common situation in both PLI and SDP tasks. Even if it is often done in literature, we decided to not focus on the accuracy measure to evaluate the goodness of our models as accuracy values could be misleading in the case of highly unbalanced datasets. In fact, being the probability estimation of correct classification outcomes of the model, it gives little importance to the minority class which often contains the most interesting cases. The more the data are imbalanced, the less the accuracy should be taken into account to measure the performance of the predictive models. From its mathematical definition Eq. 2.1 (where C_{ij} are the elements of the confusion matrix and N_c is the number of considered classes) one can see that high values of this metric can also be achieved by a model that only classifies instances as belonging to the majority class.

$$Acc = \frac{\sum_{i=0}^{N_c-1} C_{ii}}{\sum_{i,j} C_{ij}} \tag{2.1}$$

In such a case, the model could seem to perform well but in practice it does not work at all as everything is predicted as belonging to the same class.

Instead of accuracy, precision (P) in Eq. 2.2 and recall (R) in Eq. 2.3 values can be used to evaluate the model.

$$P_i = \frac{C_{ii}}{\sum_{j=0}^{N_c-1} C_{ji}} = \frac{TP_i}{TP_i + FP_i} \tag{2.2}$$

$$R_i = \frac{C_{ii}}{\sum_{j=0}^{N_c-1} C_{ij}} = \frac{TP_i}{TP_i + FN_i} \tag{2.3}$$

These values are separately defined for each class and a model which classifies every instance as belonging to the majority class would bring valid precision and non-zero recall values only for the majority class showing the poor performance of such a model. The Precision of the i-th class measures the probability that an example classified by the prediction model as belonging to the i-th class is actually an instance of the i-th class, so a high value for this metric (a value close to 1) means a few amounts of false positives with respect to the i-th class (FP_i). Recall of the i-th class measures the probability that examples of the i-th class are actually predicted as belonging to it, an high value of this metric (a value close to 1) means a little number of false negatives with respect to the class i (FN_i).

Another important measure which summarizes the precision and recall values as it is defined as their harmonic mean is the F1-score or F-measure (F) in Eq. 2.4 which gives an idea of a trade-off between the precision and recall values.

$$F_i = \frac{2R_iP_i}{R_i + P_i} \tag{2.4}$$

Especially in the defect prediction literature, the models are usually compared using the F-measure (F) and the Area Under Curve of the Receiver Operating Characteristic (AUC-ROC) curve. The ROC curve is obtained by plotting the True Positive Rate (TPR), i.e., Recall, against the False Positive Rate (FPR) Eq. 2.5 at various classification thresholds.

$$FPR = \frac{FP}{FP + TN} \tag{2.5}$$

The Area Under Curve of the ROC curve, i.e., the integral of the ROC curve from 0.0 to 1.0, gives a measure of the probability that an actual positive example has and higher probability than an actual negative example of being classified as positive according to the model under evaluation.

As many of these metrics refer to every single class it can be useful to compute the average values to have an idea of the whole model's performance. For multi-class unbalanced problems, there are different ways to compute averages, here we use two of the mainly used averages, i.e., micro and macro-average. The former is useful in order to take into account the number of instances per class: classes with a higher number of examples will have a heavier influence on the average value than the less popular ones. In fact, micro-average for precision is defined as in Eq. 2.6 where TP_i are the true positives (correct predictions for the i-th class) and FP_i are the false positives (instances predicted as belonging to the i-th class but that belong to another one). Macro-average is defined without taking into account the number of instances per class, i.e., every class will equally influence the average. For example, in the case of precision macro-average is defined as in Eq. 2.7 where P_i is the precision value for the i-th class as in Eq. 2.2. Analog definitions hold for the other performance measures.

$$P_m = \frac{\sum_i TP_i}{\sum_i (TP_i + FP_i)} \tag{2.6}$$

$$P_M = \frac{\sum_i P_i}{N_c} \tag{2.7}$$

2.2. Programming Language Identification

Traditionally, PLI has been implemented in effective tools [191, 44, 69] by relying on heuristics such as file name extensions, shebang lines in executable scripts (e.g., `#!/bin/bash`), editor mode lines (e.g., `-*- mode: python -*-`), and *a priori* knowledge about programming language grammars (e.g., their keywords or comment delimiters). More recently PLI methods based on supervised machine learning (ML) have emerged, replacing the need of maintaining complicated heuristics as languages evolve with neural network training.

Several approaches have been explored for programming language detection, the most relevant of them have been also reported in Table 2.1.

[5]Well-known Open Source projects

[6]http://domex.nps.edu/corp/files/govdocs1/

Paper	**Year**	**Dataset**	**Task**	**Classes**	**Method**	**Evaluation**
van Dam and Zaytsev [183]	2016	GH	text-based PLI	20	various NLP models	F_1=0.97
Klein et al. [96]	2011	GH	text-based PLI	25	handcrafted features	acc=0.5
Ugurel et al. [180]	2002	Projects[5]	text-based PLI	10	SVM	acc=0.89
Reyes et al. [150]	2016	RC, GH, custom	text-based PLI	391, 338, 10	LSTM	acc=0.80, 0.29, 1.00
Gilda [67]	2017	GH	text-based PLI	60	word-level CNN	acc=0.97
Alreshedy et al. [12]	2018	SO	text-based PLI	21	BoW + MNB	acc=0.75
Alrashedy et al. [11]	2020	SO	text-based PLI	21	BoW +XGBoost	acc=0.89 F_1=0.89
Fitzgerald et al. [63]	2012	benchmark[6]	FTI	24	Byte-level 1/2-grams +SVM	acc=0.47
Gopal et al. [70]	2011	benchmark[6]	FTI	316	Byte-level N-grams + KNN	F_{1m}=0.90, F_{1M}=0.60
Kiyak et al. [95]	2020	GH	text vs. img -based PLI	8	CNN	acc=0.99
Hong et al. [82]	2019	SO + GH	img-based PLI	10	pretrained ResNet	acc=0.90
Ott et al. [138]	2018	video frames	img-based code detection	Java vs non code	CNN	acc=0.86
Ott et al. [139]	2018	video frames	img-based PLI	Java, Python,non code	CNN	acc=0.98

Table 2.1.: Papers about the PLI task summarization. In the Dataset column, GH and SO stand for custom extractions from GitHub and StackOverflow, respectively while RC stands for Rosetta Code.

As very relevant to our work and often used in the labeling step by many other related works we start presenting *Linguist*. *Linguist* [69] is an open-source language detection tool developed and used by GitHub to predict the language used in the files hosted by the system. The model works by implementing several strategies which account for Vim or Emacs modeline, commonly used filename, shell shebang, file extension, XML header, man page section, several heuristics, and Naïve Bayesian classification. Contrarily to our content-based

PLI model it uses much extra information about files. Its own accuracy is reported by GitHub [64] as being around 85%. The accuracy of studies that have used Linguist as ground truth should then be diminished accordingly. Additionally, Linguist relies on file extensions as a feature and its accuracy drops significantly when they are missing [64], as in the case in our problem statement. van Dam and Zaytsev [183] tested various programming language classifiers on source files extracted from GitHub for 19 language classes and labeled them using Linguist as a source of truth. They obtained a value of 0.97 for F_1-score, precision, and recall. Klein et al. [96] performed language recognition among 25 language classes using source code from GitHub and files are labeled using both file extensions and *Linguist*. Only files for which file extension and *Linguist*-detected language match have been included in their trainset. Then, a feature vector is extracted from source code using features, such as parentheses used, comments, keywords, etc., and used for training. The obtained accuracy is 50%. Ugurel et al. [180] propose various mechanisms for the classification of source code, programming language, and topics, based on support vector machines (SVM). On the language front, they were able to discriminate among 10 different programming languages with 89% accuracy. Their data were retrieved from various source code archives available at the time (it was 2002, pre-GitHub). With respect to the aforementioned studies, the approach presented in this paper is simpler, performs better in terms of accuracy, and handles significantly more (5–10x) file type classes. Reyes et al. [150] used a long short-term memory (LSTM) algorithm to train and test a programming language classifier on 3 different datasets, one from Rosetta Code [129] (391 classes), GitHub archive (338), and a custom dataset (10). The obtained accuracies were, respectively, 80%, 29%, and 100%. They also compared their results to *Linguist*, finding *Linguist* scored worse except for the second dataset in which it reached 66% accuracy. The comparison between [150] and the approach presented in the present paper is interesting. The custom dataset confirms what was already apparent from previous comparisons: one can do much better in terms of accuracy by reducing the number of language classes (and as few as 10 classes is not enough for our stated purpose). The other two datasets exhibit larger diversity than ours ($\approx$300 classes v.$\approx$100), but perform very differently. We score better than the (more controlled) Rosetta Code dataset and *much* better than the GitHub dataset. Our approach is simpler in terms of architecture than theirs and we expect it to perform better in terms of training and recognition time (as LSTM tends to be slow to train)—but we have not benchmarked the two approaches in comparable settings, so this remains a qualitative assessment at this stage. Gilda [67] used file extensions as ground truth for source code extracted from GitHub and a word-level convolutional neural network (CNN) is exploited as a classifier. The model reached 97% accuracy and is able to classify 60 different languages.

Alreshedy et al. [12] tackled the language identification problem focusing on code snippets enhancing the effectiveness of the implemented models for a relatively low number of lines of code. The dataset used here is extracted from Stack Overflow posts where code snippets are retrieved and labeled using the provided tags. The number of analyzed languages is 21 and the snippets are represented as feature vectors computed on a *Bag-of-Word* (BoW) model, while the implemented classifier is a *Multinomial Naive Bayes* (MNB) algorithm which permits the achievement of 75% accuracy. Also, the authors built an improved model based on Random Forest and XG-Boost including textual information from Stack Overflow questions besides the code snippet contents improving the accuracy score as explained in [11].

2.2.1. File-type Identification

In approaching the PLI task, the labeling strategies are almost never completely correct in identifying the language as the labeling tools are not 100% accurate, and heuristics, for their nature, represent approximations. In a first attempt, we decided to focus on the related but far from equivalent task of File-Types Identification (FTI) predicting file extensions using as labels the extensions found in the file names of repositories' instances which most probably represent source code or at least textual files.

Other works on file-type identification have been done on the more general problem of classifying file types that might also be binary formats. In the field of digital forensic Fitzgerald et al. [63] performed classification for 24 file classes using byte-level 1-grams and bigrams to build feature vectors fed to a SVM algorithm, reaching 47% accuracy on average. Gopal et al. [70] used and compared similar approaches for the same task, but for 316 classes including 213 binary formats and 103 textual ones. They reached 90% micro-average and 60% macro-average F_1-score. This relatively big gap between the two average measures hints at significant differences in the classes (e.g., frequencies in the test set). Binary file type detection is a significantly different problem than ours. There one can rely on file signatures (also known as "magic numbers"), as popular Unix libraries like *libmagic* and the accompanying *file* utility do. Such approaches are viable and could be very effective for binary files, but they are less maintainable in the long run (as the database of heuristics should be maintained lest it becomes stale) and less effective on textual file formats, where magic numbers are either missing or easily altered.

2.2.2. Image-based Programming Language Identification

One of the models presented in this work focuses on representing code as images and performing classification with respect to languages on these images.

Image-based PLI models are currently capable of recognizing a limited amount of different programming languages, with a maximum “diversity” of 10 languages found in the literature [95, 82]. Arguably, recognizing a handful of programming languages could be approached with simple heuristics without incurring the maintenance overhead of (re-)training machine learning models. Hundreds of programming languages exist in the wild, sometimes exhibiting only subtle syntactic differences and they evolve over time, slowly but regularly [159]. It is at such a scale of diversity that PLI approaches based on machine learning would be most useful, but it remains to be seen if it is possible to visually recognize many programming languages with high accuracy.

Kiyak et al. [95] compared several image and text-based approaches to Programming Language Identification (PLI). At a glance, Table 1 in their work reports that the maximum diversity supported by image-based PLI among surveyed works was 8 languages, reached by the same authors in [95] with an accuracy of 93.5% on a dataset of 40 K files. We achieve comparable performances (92% precision and recall) with much higher language diversity (149 languages) and on a larger dataset (300 K snippets). Both approaches use Convolutional Neural Networks (CNNs), the main difference being our usage of transfer learning to adapt pre-trained image CNNs. Considering the obtained performances, the saving in training effort enabled by transfer learning appears to validate our choice.

Image-based PLI has been attempted by others too. Ott et al. [138] have shown how to use CNNs to identify video frames that contain Java code within video programming tutorials (versus frames not showing code at all) and to distinguish frames containing Java from frames containing Python [139]. In the present work, we consider a much larger set of languages. They use real images from screencasts and labeling performed manually by students, whereas we use synthetic images and rely on Linguist [69] as the source of truth.

Hong et al. [82] (not considered in [95]) performed image-based PLI over 10 languages with 90% accuracy, using snippets from StackOverflow and GitHub, rendering them to bitmaps like we do, but using GuessLang [168] as the source of truth. They use ResNet as a pre-trained CNN, which we also considered in this work. In comparison, we achieve a slightly better accuracy at much higher language diversity, and we provide a more complete overview of the possible approaches by comparing the results of several CNNs.

Other visual artifacts have also been analyzed for uses cases other than PLI. CodeTube [144] uses Optical Character Recognition (OCR) techniques to index the parts of video programming tutorials that contain code fragments and allows to query them as text. Yadid and Yahav [198] used similar techniques to extract code from video tutorials, joining together snippets that spawn multiple frames with OCR error correction. Zhao et al. [207] used CNNs to automatically identify common development workflow actions in programming screencasts. The images in our dataset are not from screencasts but given the high quality of screencast frames, we expect the proposed classifiers to be applicable in that context as well.

2.3. Software Defect Prediction

Software defects can be related to different aspects of a software artifact. Defects can occur as code errors, inefficiency concerns, unwanted behaviors, or because the software does not match the aimed expectations. Software defects are often introduced at the design and implementation level, because of non-optimal choices or coding errors, and so on. Also, for compiled languages, they can be introduced as compiler-induced anomalies or defects in the assembly-produced code. In this work, we focus on defects related to source code issues that are not related to the production expectation, i.e., we mostly look at code errors, bugs, and vulnerabilities.

Detecting such defects represents a crucial task in the software production pipeline and, besides manual checking and reviewing usually made by humans, various automatic techniques exist to spot different categories of defects.

Traditional automatic methods for defect detection in software are mainly based on code analysis which can be static or dynamic and is briefly described below.

Static code analyzers Program static analysis (white-box testing) permits the detection of certain kinds of defects before executing the program, i.e., at compile time. This kind of analysis finds errors and/or code flaws that could become an error in the future, some of them are similar to compiler warnings. The static analysis includes simple syntactic checking together with more complex reasoning by implementing rules about code semantics which are not usually considered by compilers.

The actual analyzer's behavior depends on its implementation and features, in fact, depending on the actually used tool also formatting suggestions can be highlighted to respect some standards, software metrics can be computed while performing the analysis, etc.

This kind of analysis is based on the implementation of some code rules and is based on logical reasoning to check if the analyzed code respects the prefixed rules. To do so, such tools usually exploit formal methods, i.e., rigorously applying mathematical reasoning.

A common problem for static analyzers is the high number of false positives, which means that sometimes the analyzer could raise an alarm for a detected defect that is not an actual defect for several reasons. An example of static analysis is given by symbolic execution in which the algorithm is not executed on actual input data but on a symbolic version of them on which the algorithm perform logic operations. The output computation can bring different results because of the presence of conditional statements and the result is an execution tree whose leaves correspond to different execution paths (and outputs). There can be a situation in which some of these paths are never crossed in practice because of the sets of inputs that would cause these paths to be empty. Generally, establishing if a given path is walkable or not is an *undecidable* problem, i.e., finding all the possible run-time errors in a program is an undecidable task. These constraints to make approximations on the code analysis could bring the previously mentioned false positives.

Static analysis is usually used to spot defects at the early stages of the project implementation to drastically decrease the required effort and costs due to the review processes. The kinds of issues that a static analyzer can find could be the detection of dead code, anti-patterns, bugs, code guidelines violations, type inconsistency, injection, buffer overflows, etc.

Dynamic code analyzers Dynamic program analysis (black-box testing) is a kind of code testing based on the actual code execution and the program should be ready to be executed to be eligible for dynamic analysis. It can be based on known vulnerabilities and potentially malicious or unexpected inputs which are used to feed the algorithm that is going under testing and record the consequent behavior. There are no actual rules about how to implement such analysis, attention should be focused on the inputs used to test the algorithm to guarantee enough testing coverage which is an indication of the percentage of the code which is executed during tests.

This analysis can spot vulnerabilities that are hard or impossible to be found employing static analysis. For example, dynamic analysis is more suitable to test memory management issues.

As dynamic program analysis operates on program execution it does not study source code but focuses on compiled and executable programs. For this reason, this approach is more linked to binary representation than to the actual source code content.

2.3.1. Learning Approaches

Traditional automatic tools are way faster than humans in code checking and testing, this is one of the main reasons behind their success. However, depending on the project size, they may not always be exploited because of efficiency considerations which could make the application of these kinds of code analyzers unfeasible. Moreover, code projects and repositories are growing fast nowadays and the risk of making the traditional tools inapplicable is growing as well.

We already emphasized the fact that these kinds of methods (especially static models) often bring a high rate of false alarms, which means that if the analyzer tool detects a certain defect it could also not be an issue but only detected as such due to analysis approximations required by the undecidable nature of these problems. These two issues affect the efficiency, scalability, and accuracy of defect detectors, for these reasons, researchers are investing resources to explore new ways to perform the same (or similar) tasks while mitigating the limitations proper of the traditional tools.

All the new learning models developed within the software defect prediction area aim to decrease the time and costs that are usually spent in the quality assurance, testing, and fixing stages, and they are mainly oriented to the statistical approach. In fact, like in many other fields, even in software defect prediction the need of improving or adapting to new scenarios traditional tools finds the answer in data-driven learning approaches that can be exploited thanks to the availability of many open software projects stored in repositories on several archives.

In the following, we are going to introduce these new trends in research from different points of view such as data, data representations, detection models, and performance evaluation strategies.

Datasets

Source codes used in open source projects are often stored in repositories systems and can be accessed and reused accordingly to licenses specifications. These are often used to build code datasets that can serve different purposes and, in this precise case, in the defect prediction task. Depending on the model that is chosen to tackle the problem raw code data are appropriately cleaned and prepared to be used as inputs for the chosen statistical learning algorithm.

Contrarily to the easy accessibility of data, the information needed to label them is not always straightforward to obtain. Data labels are fundamental for the implementation of supervised learning algorithms and the performances of the produced models. In the defect prediction case, labels could concern the presence, nature, location, and/or other characteristics of software defects within the considered code depending on how the dataset is built and the purposes that it should satisfy. In the datasets found in the literature, labels are mostly characterized by binary nature as the analyzed code is labeled as buggy or non-buggy, without other details about the defects. However, few existing datasets also show the number of bugs within the considered code, the location, and the kind of the bugs.

The main problem for labels availability is the cost of the labeling task to create an accurate dataset data should be also manually reviewed by human experts to be reasonably confident about their truthfulness. Only a few datasets have been manually reviewed by experts as this is a highly time-consuming task but it also contributes to create more valuable models as data has been checked for defect presence. In other cases, the dataset is labeled exploiting external information as commit messages or bugs reports.

There are also some works that use datasets whose labels are represented by traditional tools' outputs, in these cases the models trained on such datasets attempt to imitate the behavior of the traditional tools used in the labeling phase. This can be useful if we aim to create a model that is faster and/or lighter than the traditional ones or just for research reasons as they cannot improve the prediction from the accuracy point of view. An example of this kind of dataset can be found in [66] where bugs information is represented by the output files created by the Infer static analyzer [55] which shows the type and location of the bugs. This dataset is used in our work about defect prediction as it will be presented in the dedicated Chapter 4. However, as we already noticed in the previous sections this labeling choice can be a source of many false positives that would impact the models trained using this dataset.

An important issue shared among almost every dataset analyzed in this literature study step is represented by the high unbalance of the classes. It is way more common to find non-defective pieces of code than defective ones in actually used projects and, for this reason, when bug prediction datasets are created exploiting them the buggy class is much less populated than the non-buggy class.

Another feature that characterizes the analyzed works and the exploited datasets are the prediction levels at which the defect detection is performed, e.g., code granularity. Part of the literature is referred to detect the presence of defects within the entire file, in these cases, the prediction is done at file-level.

Instead, other works are focused at class-level, function-level, or, only in a few recent works, at line-level.

Open manually labeled datasets for software defect prediction are very rare, an example can be found in [210, 37] where the dataset used by the authors has been retrieved from some important C libraries by collecting vulnerability-related commits (based on some relevant keywords) and extracting buggy or non-buggy functions, then manually reviewed and labeled (a task which required around 600 man-hours). This dataset is named *Devign* and it is part of the *CodeXGLUE* dataset described in [113]. Also, in [173, 145] the used dataset has been manually reviewed from the labeling point of view.

Sometimes, labels are generated with some other strategies. Wang et al. [190] obtained labels from bug reports and commit history for various Java projects. Similarly, [4] codes changed by bug fixing commits in a version control system are labeled as defective. Li et al. [104] labeled a file as buggy if it contains at least one post-release bug and non-buggy otherwise.

There are several benchmark datasets for the defect prediction task that have been exploited in many works in the defect prediction research area. Some works used datasets that are not or only partially publicly released, and, for other researchers, it is difficult to compare performance on reasonably similar data samples. For this reason, some datasets have been shared to permit fair comparisons between models, for example, the *Unified Bug Dataset* [59] has been built for this purpose. To do so, the authors unified multiple preexisting datasets which share some important features such as the use of the Java programming language, the availability of bug information referred to at file or class-level, and the availability of unambiguous code elements names. The data features are numerical as they are represented by software metrics computed on code, but the actual code can be retrieved thanks to the mentioned availability of the code elements' names. The code features included in the Unified Bug Dataset are both extracted from the original datasets and extended with new ones.

The *Unified Bug Dataset* unify 5 datasets: *PROMISE* [161], *Eclipse Bug Dataset* [212], *Bug Prediction Dataset* [43], *Bugcatchers Bug Dataset* [74] and *GitHub Bug Dataset* [176]. In all these datasets the labels about the bugs are represented by the number of defects that have been found in the considered code elements.

The *PROMISE* dataset [161, 20] is one of the most important and used datasets in the defect prediction research community. It is a collection of different important projects written in Java (such as the *NASA* dataset) aimed to promote advances in software engineering research. These projects are kept separated in the structure of the *PROMISE* repository as they can be not homogeneous in their features and detection levels. Code inputs are represented by various kinds of software metrics computed by McCabe [120] and Halstead [75] features[7] extractors on the actual projects' source code. This repository is not composed of code-like data but of the software measures' values which are numerical input data. The actual code can be retrieved using the information provided about the considered projects and code elements' names.

The other datasets included in the *Unified Bug Dataset* are similar in structure and data retrieval strategy even if the code features which represent the codes could be different.

There exist many other datasets for the defect prediction task. The dataset presented in [126] is designed for the defect localization task and it contains programs in Java, C++, and Python programming languages. It has been created to support the research in the area of bug localization among different languages as the already existing dataset for this task only focuses on the Java language.

Another useful and widely used tool is the bug database *Defects4J* presented in [90] which collects bugs information and can be used to create datasets. Just et al. [90] describe an approach to isolate 357 real-world bugs in Java projects and use it to create a bug database that can easily be extended with new bugs. Bugs isolation means that by detecting versions that involve bug fixing changes, only fixing related changes are considered in the retrieving step. Also, bugs are presented with reproducible tests which demonstrate bugs and patches. As bugs contained in this database refer to real-world projects, Defects4J is suitable for working with large-scale software. *Defects4J* also inspired the construction of a similar dataset, *BugsInPy*, which is about bugs in codes written in the Python programming language [192].

Saha et al. [158] criticize *Defect4J* dataset of being too restrictive in the Java bugs variety and try to improve the dataset from this point of view by creating the *Bugs.jar* large-scale Java bugs dataset built thanks to the high level of building automation. This work aims to create a bug dataset that maximizes bug diversity representing real-world situations. It also focuses on bug reproducibility by presenting valid test cases and sharing the others *Defects4J* features. In this dataset, each instance contains both the buggy and fixed code versions together with a bug description and a set of test cases that can be used to demonstrate the bug presence and the effectiveness of the patch implementation. The projects exploited in this dataset have been extracted from the Apache ecosystem on GitHub based on their tagging to increase the aimed diversity.

[7]These are software complexity measures.

An interesting dataset is the *Draper VDISC Dataset*, it has been built in [153] and has been made available by the authors. This dataset has been created because the existing datasets result to be limited in size and variety concerning the authors' aims. They focus on function-level vulnerability detection considering C and C++ codes from the SATE IV Juliet Test Suite, Debian Linux distribution, and public Git repositories on GitHub. Instances from the SATE IV Juliet Test Suite are already labeled while the ones from the other projects are not and three static analyzers are used to label functions as vulnerable or not.

Fei et al. [57] construct the *GHPR* dataset by extracting defective codes from GitHub repositories querying for defect-related pull requests instead of commits. Class-level detection is the main purpose of this dataset and it is characterized by the fact of being balanced, a very rare feature in bug datasets. The dataset is balanced because of the data extraction methodology, in fact, the authors found defect-related pull requests and they used the code element before fixing as a defective example and the correspondent version after fixing as a non-defective example. In this way, they obtain the same number of examples for the two examined classes.

Many other datasets can be found with their strengths and weaknesses. The work presented in [61] is an attempt to create a new dataset, called *BugHunter* dataset, which can benefit from the different positive aspects of previously existing datasets and a novel strategy in the dataset building phase. The bugs are isolated, only the defective and fixed code portions are used and the entire version history is considered. Bugs are associated at different levels, in fact, there are some instances for file-level detection and others for class-level and method-level detection. Each instance of the dataset is represented by the actual Java code element and several software metrics and bug information. The construction of the dataset has been designed to be automated to create a large and easily expandable dataset.

A dataset that covers projects written in several languages rather than focuses on one language only is the *Software Assurance Reference Dataset* (SARD) [19]. The code data come from both synthetic and real-world programs and the covered languages are C, C++, Java, PHP, and C#. Around 150 kinds of defects are represented within this dataset, and they are precisely located in the program's body. This is a very different feature with respect to all the other described datasets which only have the number of defects or a binary buggy/non-buggy class feature.

2.3.2. Defect Prediction aspects

The models found in the literature about defect prediction can focus on slightly different applications and have different features. The first distinction that can be highlighted is between within-project and cross-projects defect prediction. In Within-project defect prediction, there is the attempt of building a predictive model which is trained and tested on data that belongs to the same project. In this case, the model is trained on the code of a project and then tested and exploited on the same projects to make predictions in parts that have not been used in the training phase such as all the code that will be written in the future within that project. To use these techniques, each software project should have its predictive model and to effectively train it, there should be enough code data from that project, which is not always a condition that can be satisfied. Cross-project defect prediction, on the other hand, can solve the problem of insufficient amount of data from the same project by creating models that can be trained on data from some projects and tested and exploited on completely different projects. The cross-projects defect prediction approach seems to be more challenging than the Within-project defect prediction one, probably because of the presence of more differences between data used for train and test as the projects would share fewer characteristics such as project structures, programming styles, identifier names, function names, etc.

Another important feature that is used to distinguish among the models presented in the literature is the *level* or *granularity* at which the considered model operates. In fact, in some works the defect detection task is performed within a whole file so the predictive models operate at file-level, other works focus on detecting defect presence in code methods and the corresponding models are detectors at method-level or function-level. Similarly, models can be found to operate at class-level and line-level. On the other hand, change-level approaches are a bit different as they do not act at the level of code elements but refer to the changes which are performed by programmers on the considered code. Usually, data for these models are retrieved as commits changes from open Version Control Systems (VCS) available over the internet, i.e., GitHub [68]. Commits are usually represented as couples of added and removed lines of code and are also used to build also line-level models. Change-level defect prediction is also known as Just in time defect prediction, an approach that tries to detect the possibility of the introduction of a defect when the code is produced or modified.

Machine Learning models and Software metrics

The first works which tried to improve traditional methods for software defect predictions by exploiting the huge code data availability focused on traditional Machine Learning (ML) techniques which use handcrafted features designed by code experts and computed on source codes. Mainly, features used in these works are metrics used in software quality and complexity estimations. It is a common understanding among the software security community that complexity negatively impacts software projects from a security point of view as increasing the complexity level of software the probability of having vulnerabilities increases too. Cyclomatic complexity is evaluated on the program's control-flow graph (CFG), a particular representation of program execution, by computing how many linearly independent paths can be found in it, i.e., none of these paths can be rebuilt by combining other of these paths. Many complexity measures have been proposed, some of them can refer to the size of the considered code, number and/or frequency of operators, the number of loops and conditional statements, pointers usage, and so on. Very common features in metrics-based defect prediction models are the McCabe [120] and Halstead [75] complexity measures together with CK metrics designed in [33, 34] within the scope of the Object-Oriented paradigm to inform about the complexity and quality of software. As pointed out in [205, 135, 134], also Michura's standard deviation metrics, Bansiya and Davis' metrics, and Etzkorn's average complexity measures are useful to create defect prediction models.

Radjenović et al. [147] show how to process metrics are effective in post-release defect prediction models found in the literature. Process metrics involve code delta, code churn, the net increase in Lines Of Code (LOC) for each module, the number of developers, the number of past faults, the number of changes, the age of a module, and so on.

Dependency metrics are also considered in some works like [127, 128] and are a measure of the relationship between different code elements, such as data and call dependencies.

These kinds of features are computed on code and used to build the code representation that will be passed as input to some Machine Learning algorithms which learn to discriminate defective codes from non-defective ones by looking at these features and their appropriate combinations.

The works presented in [186, 108, 56] are systematic literature reviews that analyze several papers published in software engineering journals and conferences about software defect prediction and filtered by setting some desirable features and covering different time slots such as 2000-2013, 2014-

2017 and 2016-2019, respectively.

Wahono [186] shows how most of the used datasets are private and the models built on them are hard to be compared with other research works. The most used traditional machine learning methods are Logistic Regression, Naive Bayes, KNN, Neural Network, Decision Tree, SVM, and Random Forest but, even if many works focused on comparative analysis among different classifiers, they only show results for a given set of data. From here, it follows that there are no models which perform generally better than others for all datasets and comprehensive analysis of different models remains a tricky task. Despite this, some strategies are suggested to improve models' performance such as parameters optimization, using ensemble and boosting algorithms, and implementing some feature selection methods to reduce data dimensionality.

Li et al. [108] also focus on feature selection implementation together with other useful data manipulation strategies found in data prediction works. They also show other important aspects of common problems found in defect prediction data such as the need of handling the widespread problems of class imbalance and the presence of noise. A common and promising strategy to handle class imbalance in defect prediction seems to be the SMOTE technique [30] which aims to synthesize new instances for the minority class. Other interesting cases look at working with heterogeneous data which can help improve models when the lack of historical data is an issue. Some new techniques which emerged in machine learning fields are cited such as dictionary learning [89, 187], transfer learning [31, 194], kernel ensemble learning [200] and deep learning. The presence of many works which are hard to be replicated is highlighted, together with the problem of the difficulty in a fair evaluation of different models and generalization to unavailable closed software projects.

Faseeha et al. [56] describe several recent attempts to improve existing works on defect prediction. Many of these works focus on improving data quality by using various data preprocessing techniques such as feature extraction and selection, standardization and normalization of numerical features, and data sampling for class imbalance. Another kind of attempt in improving predictions is given by the usage of meta-learning frameworks. Meta learners are used to select the best learning model among a set of selected ones performing on some datasets. This kind of technique is used in [45, 52, 51] implementing it in different ways.

Ponnala and Reddy [143] present one of the most recent surveys about defect prediction describing the current state-of-the-art in this field. In this work, the literature about traditional machine learning models mainly focuses on hyperparameters optimization [1], features reduction via Principal Component Analysis (PCA) [73] or utilizing hybrid wrapper-filter heuristics [84], the use of hybrid features (static and dynamic) [32] and hybrid models [175].

An attempt to design defect detection as an anomaly detection task is described in [4]. As the defect prediction datasets are usually highly unbalanced and being the minority class the one which refers to defective codes, these defects can be treated as anomalies in non-defective codes. The dataset used in this work is the *NASA* function-level dataset which belongs to the *PROMISE Repository* on which they extracted 21 features among the Halstead and McCabe ones and performed deduplication of repeated instances. The approach used here exploits autoencoders and trains them on a reconstruction task for the examples labeled as correct ones by minimizing the reconstruction error calculated as the Root Mean Square Error (RMSE). After this phase, the reconstruction error for defective and non-defective instances are used in statistical tests to determine the distributions of errors in the two cases which are then used to decide the defectiveness of examples. The model reached a $0,2 \leq F_1 \leq 0,6$ depending on the dataset (as the *NASA* dataset is composed of different datasets too) but 4 out of 5 cases perform better than other tested methods, i.e., Gaussian Naive Bayes, Logistic Regression, k-Nearest-Neighbors, Decision Tree, and SVM.

As one of the main problems for learning-based defect prediction approaches is the lack of labeled data some works based on semi-supervised and unsupervised models have been analyzed. Li et al. [107] collected and reviewed several primary studies which implement unsupervised algorithms that bring results comparable to the ones obtained using supervised algorithms. Catal [28] compared several semi-supervised algorithms to find the best choice that would be useful when not enough labeled data are available.

Within these surveys, the most evident trend in defect prediction seems to be the implementation of deep and representation learning algorithms. As we will see in the following section, representation learning seems to better extract features from source code as it is also capable of detecting semantic aspects aside from syntactic and static features. The extraction and exploitation of useful semantic features will help in improving the performance of the defect prediction task.

Deep Learning models and Feature Learning

Methods presented in the previous section are mainly based on hand-crafted features which are used to build feature vectors to represent the codes and traditional machine learning algorithms that process these vectors to distinguish between potential defective and defect-free software. These models are not able to capture syntax and semantic aspects of programs that are more relevant in the defect prediction task than the traditionally used code complexity metrics.

More recently, due to the advances in the Deep Learning (DL) field, new powerful models are developed and one of their main characteristics is the capability to automatically learn the features that will be used to classify the piece of code directly from the code itself instead of selecting and computing metrics on it, permitting to enrich the feature pool that is available. This feature is known as Representation Learning and working with models based on it within the defect prediction task brings consistent improvements to the previously described works based on handcrafted features.

Deep learning algorithms are based on Artificial Neural Networks (ANN) with more than one layer of neurons. The presence of multiple layers is the characteristic that permits to automatically learn, and extract features relevant to the task without the need to manually define them. Moreover, the features extracted by deep models are inherently different from the traditional ones as they are capable to capture and encode semantic and context aspects. The usage of these techniques in the defect prediction field permits improving detection performances and to produce the most advanced models in this field; some of them are discussed here and reported in Table 2.2.

There are some works that use the predictive power of deep learning models to detect software defects by analyzing bytecode or assembly programs representation [23, 142] as defects do not depend only on source code but, for example, they could be introduced during the compilation phase. However, most of the works found in the literature focus on predicting defects starting from information found within the source code, and we will focus on this trend in the following.

Different representations can be obtained from the source code content. Source code can be viewed as plain text and treated with techniques similar to the ones developed within the Natural Language Processing (NLP) field, the source code text can be analyzed at character level or tokenized accordingly to some tokenization rules. Also, from the source code texts, it is possible to extract the Abstract Syntax Tree (AST) code representations by using the parsing rules of the programming language grammar. Control Flow Graph (CFG) is another common code representation that can be built from source

[8]Note that the evaluation values provide just an indication about the models' goodness as as evaluation strategies, particular applications and settings could differ among different works and also within the same paper.

[9]https://zenodo.org/record/3733794

[10]https://www.codechef.com/problems/

[11]https://sites.google.com/view/devign

[12]https://github.com/serg-ml4se-2019/group5-deep-bugs

Paper	Year	Dataset	Task	Granularity	Method	Evaluation[8]
Bryksin et al. [23]	2020	GH[9]	anomalies detection	function-level	metrics, AST/bytecode N-grams	qualitative
Phan and Le Nguyen [142]	2017	benchmark[10]	binary semantic DP	-	assembly instructions + CNN	F_1=0.73, acc=0.73
Wang et al. [189]	2016	PROMISE	Java binary DP	file-level	AST node sequence + DBN	F_1=0.64
Russell et al. [154]	2018	SATE IV Juliet Test Suite	C/C++ binary DP	function-level	token sequences + CNN/RNN	F_1=0.84
Li et al. [105]	2017	Tera-PROMISE	Java binary DP	file-level	AST node sequence + CNN + hand-crafted +logistic regression	F_1=0.61
Dam et al. [42]	2019	PROMISE	C/C++ binary DP	file-level	AST + Tree-LSTM + logistic regression/RF	F_1=0.90
Zhou et al. [209]	2019	custom humanly labeled[11]	C binary DP	function-level	AST, data/control flow graphs + Gated Graph Recurrent layer	F_1 =0.85
Ferenc et al. [60]	2020	Unified Bug Dataset	Java binary DP	class-level	metrics + DNN	F_1=0.55
Xu et al. [195]	2020	GHPR dataset: bug/clean pairs	binary DP	file-level	AST + GNN	F_1=0.75
Hoang et al. [80]	2019	QT + OPEN-STACK	binary JIT-DP	code change-level	commit msg, added/deleted SLOCs + CNN	AUC=0.10, 0.14
Shi et al. [163]	2021	PROMISE	binary DP	file-level	AST nodes sequence + CBOW + CNN	F_1=0.63
Hoang et al. [81]	2020	QT + OPEN-STACK	code change representation + binary JIT-DP	code change-level	added/deleted SLOCs + GRU +HAN + DeepJIT [80]	AUC=0.82, 0.81
Shi et al. [162]	2020	PROMISE	binary DP	file/function-level	AST path pairs + bi-LSTM	F_1=0.74
Briem et al. [21]	2019	GH[12]	Java binary OBOE prediction	function-level	*code2vec* + sigmoid	F_1=0.76
Humphreys and Dam [85]	2019	PROMISE	Explainable Java binary DP	file-level	tokens sequence +Self-Attention Transformer Encoder	F_1=0.67

Table 2.2.: Papers about the SDP task treated with deep learning approaches.

code and represents the entire program execution flow and variables states, a Data Flow Diagram (DFD) is useful to study the flow of data for a given program. Other program representations exist and focus on different aspects of the program they represent. Depending on the aspect they focus on and the task that we aim to perform they can be chosen to be used as inputs for deep learning models. All these representations should be translated into a numerical vector form to be processed by the learning algorithms.

According to [5], within the literature about deep learning applied to defect prediction the most used techniques are Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), and Long Short Term Memory (LSTM), and Transformer architecture.

One of the most relevant works about deep learning in defect prediction is presented in [189] and most of the other papers that will be discussed in the following refer to it to compare the described models. Wang et al. [189] present an important step in the defect prediction literature as it is the first work that uses automatic feature learning for this task, allowing learning about semantic and syntax aspects of code. In this work, Java codes are parsed into their ASTs and among all the nodes only those about method invocations, class instantiations, declarations, and control-flows are kept into account to represent the codes. These nodes are extracted and used to form a sequence which is then encoded in the form of integer vectors that represent the code snippet. These vectors are used as inputs for a Deep Belief Network (DBN) [14] after being normalized. The DBN architecture is capable to learn features during training and classify code w.r.t. its defectiveness. The described model performance improves the results for defect prediction of previous works which only leverage traditional manually defined features in both within-project and cross-project cases.

Russell et al. [154] perform vulnerability detection on C/C++ source codes as a binary classification task at function-level granularity. The model developed here exploits data from various sources, in fact, data belongs to the *SATE IV Juliet Test Suite* dataset [132] which contains synthetic code vulnerability examples but, to cover natural code situations, the dataset has been enriched with data from *Debian* [46] and *GitHub* [68] projects labeled utilizing static analyzer tools. The dataset obtained from the authors is highly unbalanced and a weighted loss is used during the training phase to mitigate the problems which could arise from this unbalance. During tokenization authors used a customized lexer to unify tokens according to their types, e.g each identifier for an integer variable is represented as a `INT_TOKEN` which is a type-specific placeholder, so they reduce the problems due to the Out Of Vocabulary (OOV) tokens and to create a vocabulary of only 156 tokens. Tokens are then represented as

13-dimensional embedding vectors whose components are randomly initialized and then learned during training, while the code is represented as a sequence of these vectors. Tokens' sequence is fed into both CNN and RNN models which are used to automatically extract relevant features which are then used in a binary classifier. Features extraction and classification are trained and performed separately. The best classifier results are shown by the Random Forest model reaching $F_1 = 0.840$ and ROC-AUC$= 0.954$, but also a Fully Connected Layers Neural Network has been tested.

Li et al. [105] describe a within-project defect prediction model *DP-CNN* for Java code files on projects extracted from the *tera-PROMISE* Repository and the data preparation strategy is similar to the one described in the previous work [189]. As the authors focus on within-project DP, the trainset is composed of older versions of a project while tests are performed on newer versions of the same project. Each entity of the dataset is labeled as buggy if it contains post-release bugs and non-buggy otherwise. The trainset portion is modified by duplicating instances of the minority class until the balance w.r.t the majority class is reached. As in [14], tokens are extracted from ASTs selecting the three kinds of AST nodes cited before: method invocations, class instantiations, declarations, and control-flows. Sequences of tokens are obtained, and they need to be transformed into a numerical vector. Each token has an integer ID that represents it within sequences which are then passed to an embedding layer, which learns tokens embeddings during training and then processed by a CNN which can automatically extract features and capture structural information from the sequence. In this model, learned features are then used together with traditional hand-engineered features (code metrics or process metrics) as input of a logistic regression classifier. During model performance evaluation the model performs better than the state-of-the-art DBN model [189] on some datasets. Comparisons have been done also against a model version that does not include the previously mentioned hand-crafted features whose usage seems to slightly improve performance in almost every tested project. The average F_1 score of this model among all the tested datasets is $F_1 = 0.608$.

An LSTM version that has been adapted to work on tree-structured data is used in [42] where the AST structure of code is provided as input of the model used to predict software defects in both within-project and cross-projects defect prediction cases. Tree-LSTMs [172] have the property of being able to capture both syntactic and structural information while automatically learning nodes representations that reflects semantic aspects. Each AST node is represented by a numerical vector obtained from an embedding matrix which is randomly initialized and then it learns the nodes embeddings during training. The tree-LSTM model takes a tree node as input and it recursively processes its children nodes producing for each node a hidden state vector and a context vector. To perform defect prediction, the embedding of the root node of a given file AST is used as input for the model, and the hidden state vector obtained from the tree-LSTM is passed to the final binary classifier (Logistic Regression and Random Forest). The model is developed on the *PROMISE* Repository and real projects provided by the Samsung company, on the Samsung dataset the performance is above F_1=0.9 but on the *PROMISE* Repository the F_1 measure is less than the one found in [189].

Zhou et al. [209] treat defect prediction for C functions as a binary classification, but this time the dataset is manually labeled and accurately humanly reviewed (requiring 600 hours of human work, here is one example of the high cost of the labeling phase). The test set is sampled until the real-world distribution of vulnerabilities is reached (around 10% is vulnerable). Functions are represented in different ways, i.e., AST, data and control flow, and multi-edged graph, and they are used in a composite unique representation. Graph nodes are represented as vectors and initialized using *word2vec* [124] and passed to a graph embedding layer, a gated graph Recurrent layer, and a convolution module. This model reaches $F_1 = 0.85$ and seems to be suitable to catch new kinds of vulnerabilities.

Ferenc et al. [60] treat a binary classification defect prediction for Java at class-level built on the *Unified Bug Dataset* [59]. 18% of the dataset is composed of vulnerable classes and a parametric sampling is used to balance the trainset used in the 10-fold cross-validation. Despite this work uses a Deep Learning architecture, the input is represented by source code static metrics passed to a neural network of 5 layers used to classify the vulnerability proneness. In this work, all the numerical features are standardized and normalized. The best model reaches $\mathrm{F}_1 = 0.550$ and AUC-ROC= 0.84.

Xu et al. [195] develop an attempt to detect defect patterns at file-level focusing on an important aspect of software defects, in fact, the authors highlight the fact that in some cases pieces of code could be considered defective or not depending on the environment in which they are placed. For this reason, defects do not only depend on semantics and/or syntax but also on dependencies, location, and requirements. In this work, the *GHPR* dataset is used and it is built and described in [57], in particular, the authors obtained the dataset from *GitHub* [68] repositories and extracted classes of Java codes based on Pull Requests selected employing regular expression rules considering the classes both before and after the defect is fixed. In this way, there is the same number of defective and non-defective examples remaining with a balanced dataset. The code representation used in [195] are subtrees obtained by pruning

ASTs trying to retain information about defects through community detection algorithms. In addition, names found within codes are split into subtokens according to camel case syntax, and semantic and context information is captured by means of *word2vec* [124] and Bag of Words (BoW) techniques. The classification is performed by means of various algorithms such as Graph Convolution Network (GCN) [94] and its simplified version (SGC) [193], Graph Isomorphism Network (GIN) [196], Graph Attention Network (GAT) [184] and GraphSAGE [76] with 50-cross validation and an hyperparameters selection strategy reaching values for F_1 above $0,75$ for all the implemented algorithms.

Defect prediction at change-level is also known as just-in-time defect prediction. Hoang et al. [80] use representation learning to build an end-to-end deep learning model for just-in-time (JIT) defect prediction *DeepJIT*. Commit messages, removed and added lines are parsed and represented as sequences of vectors (the vectors are the words' representations). The encoded commit messages and code changes are then used as inputs for a CNN which can extract relevant features for the FCL network which is used to do the final binary classification. To face data imbalance the authors use a weighted loss during training exploiting as datasets the *QT* [40] and *OPENSTACK* [136] projects as they were prepared in [122]. The model has been compared with [199] and [122] showing relevant improvements.

An unsupervised Java code representation is described and applied to defect prediction in [163]. The *PROMISE* Repository is used to build the dataset and random duplication is implemented to balance the trainset. The authors start from AST code representation and extract trees which are isomorphic to them, retaining only information which is relevant to a given perspective to reduce ASTs complexity. Four different relations are defined among tree nodes to establish the relatedness of the nodes. Following each relation definition, it is possible to build tokens sequences for the specific relation and, after splitting tokens into subtokens to promote vocabulary and model flexibility, these sequences of subtokens are used to predict a node presence employing close nodes (context) using methods similar to *CBOW* or *SkipGram* in *word2vec* [124] and consequently learning node embeddings. These embeddings are then passed to a CNN architecture which extracts features at a higher abstraction level and uses them to classify codes as defective or non-defective.

In our work on defect prediction, we want to treat the problem as a multi-class task to be able to identify the *kind* of the defect (among three selected kinds of bugs) instead of just predicting the presence or absence of defect within a code fragment. To our knowledge, the work in [190] is the most similar to our aim. They focus on predicting the presence of null pointer dereference, array index out of bound, and class casting bugs. However, their approach is to focus on a prediction model structure definition in general and then separately train it on different datasets. Each dataset has binary labels but referred to different kinds of bugs respectively, in this way the problem is still treated as a binary classification task and only the joined usage of the trained models permits multiple bugs identification. This approach has the advantage of being easily extensible to other kinds of bugs but can suffer from scalability issues in terms of the number of predicted bugs types. Another similarity to our approach stays in the data labeling which is based on the outcomes of a static analyzer which the authors try to imitate. The code is represented as a control flow graph which is then fed into a Graph-Neural Network (GNN) and the model results to be effective in bug prediction for each of the analyzed bugs kinds.

Code Representations Even if not strictly related to defect prediction here we describe some of the available models which can serve to extract vector code representations that can be used for several downstream tasks.

An important model for language representation is the Bidirectional Encoder Representations from Transformers, also known as *BERT*, and it is described in [49]. Even if it was not originally developed within the field of *Big Code* it has been adapted to this field and exploited to generate code representations as it happens in *CuBERT* [91] where the advantage of using a pre-trained model fine-tuned on the desired tasks is highlighted. Similarly, *CodeBERT* in [58] is a pre-trained general-purpose model based on *BERT* model which is adapted to manage tasks that involve textual data written in both programming and natural languages.

Hoang et al. [81] describe a programming language agnostic technique to learn a vector representation for code changes *CC2Vec* from unlabeled data that can be used for various purposes and supervised learning tasks, for example in Just-In-Time (JIT) defect prediction, allowing semi-supervised learning which is suitable when few labeled data are available. Code changes are extracted from files, removed and added lines are tokenized and used to build a token vocabulary and two 3-dimensional matrices which are the representations of added and removed code. These two representations are used as inputs to a Hierarchical Attention Network (HAN) which encodes the matrix by means of a bidirectional Gated Recurrent Unit (GRU) and then processes it via an attention mechanism at word-level, the result of these processes is then subject to the same mechanisms at line level and hunk level. This combined hierarchical mechanism is capable of extracting the embedding vectors from the matrices that are used as inputs, i.e., the representations of the removed and added code, which are then used to represent the whole code change. The model is trained to learn a probability distribution over words which are referred to as log messages related to the code change. The authors apply this representation strategy to the Just-In-Time defect prediction task on *QT* and *OPENSTACK* datasets and using the state-of-the-art model, *DeepJIT* [80] as a classifier and verify that this approach improves the previous results reaching AUC-ROC = 82.2 and AUC-ROC = 80.9 on the two datasets.

In [10, 133] the authors developed *code2vec* a model to generate semantic labeling for code, such as suggesting methods names based on the methods bodies. The most interesting part of this work is the strategy that has been implemented to generate the code representation which is based on the extraction and elaboration of paths within the AST of the analyzed code and, as the authors claim, the obtained representation is suitable for being used in other tasks employing the transfer learning technique. Once the code is parsed and the corresponding AST is obtained it is possible to obtain a sequence of internal tree nodes ($path_{if}$) which connects two terminal nodes ($token_i$ and $token_f$). Some of these paths are extracted for each analysed piece of code and are represented as a so-called *context* ($token_i$, $path_{if}$, $token_f$) whose three components are represented as vectors and concatenated to form the *context* vector representation. The obtained representations for each piece of code are processed by an attention-like model which selects relevant paths and generates a vector that represents the whole piece of code and can be used to perform the actual prediction. This representation results to be the state-of-the-art in method naming and the approach is suitable to be adapted in many other situations. Inspired by the work in [10], a code representation named *PathPair2Vec* based on AST path pairs is described and adopted in the defect prediction task at both file and method-level in [162]. Data used in this work come from the *PROMISE* Repository and the portion of the dataset used in the training step has been subjected to a random replication of the instances belonging to the minority class until the balance between classes is reached. The path pair (a pair of *short paths*) is defined as a sequence of internal nodes of the program's AST which connects two terminal nodes. Among all the path pairs that can be found in an AST, the authors defined the concepts of span and length which help to select the most relevant paths to include in the pairs. Terminal nodes are usually identifiers and subtoken splitting is used to capture names semantics and reduce the OOV effects, the subtokens information is encoded in a vector representation together with type information which is then processed by a bidirectional-LSTM (bi-LSTM) model. Internal nodes are represented by embedding vectors and the two paths which compose a pair are processed separately through a bi-LSTM architecture. An attention model is used to create the final vector representation of the path pairs starting from the previously obtained four embedding representations, this vector is then passed to a *softmax* layer to perform the actual classification. Experimental results show that in both within-project and cross-projects defect prediction the models built on method-level ASTs are stronger than the ones built on file-level ASTs, this means that logical meaning in code is mostly a local property, and it is concentrated within a method while it seems to be weak between different methods. This model reached $F_1 = 0.74$ which represents a considerable improvement with respect to the *DP-CNN* model [105] they are comparing to and it also perform better than *code2vec* [10] and *code2seq* [9] models which share the ATS path-pair based code representation.

Another work which exploits the code representation obtained by using the *code2vec* model [10] for defect prediction is described in [21]. As in [10] the authors only tested the representation for the method naming task, in [21] the same representation strategy is used to build a model which can detect Off-By-One kind of errors (OBOE). These errors are logic errors that occur when wrong comparators are used in some statement, for example, using $<$ instead of $\leq$ or vice versa when iterating over an array bringing to fewer or more iterations than the needed ones. In this work, the authors also compared the results to the outcomes of several static analyzers and pointed out that these kinds of errors are not usually detected by these tools. The dataset has been built starting from the one used in [9] and modified to the proposed aim. Methods containing comparators are kept for learning and testing of the model, some of these have been corrupted in a controlled way to generate off-by-one errors examples (e.g., by switching from $<$ to $\leq$) while the original versions are interpreted as correct instances. Depending on the type of statement (`if`, `for`, `while`, etc..) in which the comparators are the authors defined different context classes and study the detection capability in various situations. The *code2vec* trained model provided by the authors is exploited by extracting the embedding layers and replacing the last layer with a binary classifier layer, i.e., a sigmoid layer, implementing the transfer learning strategy. The model is then used both by freezing the embedding weights while learning only the modified layers and by using the trained layer as initialization to exploit the fine-tuning technique. The latter results to be the best approach and depending on the context and the comparator type the obtained F_1 values are very different ranging from 0.52 to 0.88.

InferCode is another AST-based code representation and is described in [25]. This approach works on unlabeled source code data and the final aim is to build a task-agnostic vector representation for the input code. Differently from other code representations such as *code2vec* which can both be trained on a final task or exploited as a pre-trained model and fine-tuned on other downstream tasks, *InferCode* generates code vectors without any knowledge about the final task, for this reason, the vector representations can be computed before any kind of task consideration as a separate phase. After that these vectors could be used for any other purpose such as method naming, defect prediction, etc., which can benefit from the information stored within the AST code representation structure. Self-supervision is exploited by using sub-trees extracted from ASTs as labels to train the model without the need for labeling effort which as we already pointed out is one of the limitations in working within *Big Code* projects. The extracted sub-trees are accumulated within a vocabulary and used as targets for a modified version of the Tree-Based-CNN (TBCNN) model which takes the entire ASTs as inputs. The original architecture of the TBCNN is modified by replacing max-pooling with an attention mechanism permitting node embeddings combination and using textual information to initialize node embeddings. After the training phase, the TBCNN encoder can be used in a pre-trained form to generate the code vectors to serve other tasks. Another advantage of this technique is that it applies to different code granularity levels, and it is not constrained only to some kinds of code units. The *InferCode* model comes as a pre-trained model trained on Java codes and results to perform well in many tasks even considering different languages as it is built on ASTs based on a combined vocabulary of node types for multiple languages [38]. In [25] the model is evaluated on various downstream tasks and compared to other pre-existing models resulting to be an effective method for code representation.

Explainable models

Finally, in the last few years, few works emerges as attempts to introduce in this research field the feature of *explainability* otherwise the prediction models would only signal the presence of the defects without any other information to support the understanding and fixing of the bugs. A well-known drawback of many (deep) learning models is that they are mostly used as *black boxes*, which means that they do predictions after the training phase but the interpretability of the reasons behind their outcomes is usually hard to be investigated.

Several works have been attempted to deal with this issue, some of them are presented in [72, 24] together with their strengths and flaws.

Not all these works focus on explainable deep learning models which are the hardest ones to be investigated from the interpretability and explainability point of view. So, as in many other fields, this still represents an open challenge to be addressed. The reasons behind this research direction can be found both in the study of the models' dynamics and to find the actual source of the defects within the code to speed up the error localization and code fixing tasks even more and prevent similar defects occurrence in the future.

Tantithamthavorn et al. [174] focus on predicting defects representing code with vectors computed on bag of tokens, traditional software metrics, and features and using them with Random Forests. After the predictive model creation, they implement different techniques to explain the predictions of the DP models. They use the *LIME* [151] model-agnostic technique which starting from a file-level prediction model helps to identify which tokens and lines are the problematic ones within the whole files. Also, another approach is attempted in the predictions explanation, i.e., a rule-based model-agnostic technique *LoRMikA* [148] which is able to generate a rule-based explanation for the predictions.

Jiarpakdee et al. [88] perform a survey on the preferred visual explanation techniques used in defect prediction from the practitioner's point of view, in this study the preference of the *LIME* model over other attempts emerges.

The relevance of the *LIME* approach is also supported in [87] in terms of effectiveness in predictive power and explanation support. Here the *LIME* technique is also compared to another model-agnostic approach used for ease interpretability which is known as *BreakDown* [71, 170] and an improved *LIME* version (*LIME-HPO)* which implements hyper parameters optimization is proposed by the authors. However, [87] reports how modifying learning settings for the models under analysis both *LIME*-based and *BreakDown* model-agnostic explanation techniques bring unstable and inconsistent results, suggesting that these models could not be reliable in practice.

Santos et al. [160] pointed out the importance of explainability in DP too. The authors use traditional software metrics to numerically encode the pieces of code and trained an XG-Boost model. To explain the reasons behind the model predictions they also computed the SHAP (SHapley Addictive exPlanation) values [114] for each feature with respect to the used model to highlight which feature has the highest impact according to the chosen classifier.

Humphreys and Dam [85] report an attempt to build an explainable model for DP based on Deep Learning. A state-of-the-art Self Attention Transformer Encoder is implemented to process the tokens sequences inputs and the capability to locate semantic information to regions within these sequences should help in model explainability. However, results about how to inspect and evaluate the explanation capability of the proposed model are not presented.

Chapter 3.

Programming Language Identification

In any software project, the programming language(s) used to create it represent an important variable that depends on several factors such as quality, trends, usability, maintainability, etc. Programming Language Identification (PLI) is a task that can be easily performed by developers or people who are curing the software project but there exist many situations in which this is not possible. In fact, as many languages do exist and they also evolve and change in time (new language versions releases, usage habits, and trends), even when the code base scale is not prohibitively large to be manually analyzed by humans there can be situations which would require too much effort to be solved. Another situation in which it is not possible to manually perform PLI is when it comes to look at a huge amount of source code as for the case of software hosting service platforms and code archives such as GitHub [68], Software Heritage (SH) [79, 2], StackOverflow [169], etc. which host a huge size of code content (1B+ contributions for GitHub, 169M+ projects for Software Heritage and 21M+ questions on StackOverflow) and are also going to grow more. Of course, at this scale, it is impossible to manually perform PLI where this could also represent a crucial task. To name just a few of the possible applications it can support archive organization, code search, indexing and classification, program comprehension, programming trend analysis, and so on.

As the codebases scale can represent an obstacle for this and other similar tasks, it also can be viewed as the means to achieve the desired solution. These archives can be viewed as rich sources of source code data that can be treated to serve as datasets to train some learning algorithms. In this way, the chosen automatic model should learn to imitate the desired behavior to automatically perform a task, in this case, should learn how to identify a programming language starting from the source code content. The granularity level at which the task can be performed can vary based on the specific situations as it can be done at the project, file, or just a few lines of code levels.

As introduced in Chapters 1 and 2, there is a specific research field known as *Big Code* in which automatic PLI seems to fit, and this is the idea that has been followed in this work.

Depending on the use case, two major classes of ML approaches to PLI have been used [95]: text-based and image-based programming language identification. In text-based approaches source code is viewed as a characters' sequence, such as files stored in version control system (VCS) repositories. Image-based approaches can classify raster images showing code, such as screenshots of development environments or individual frames extracted from video programming tutorials. The problem here is about the possibility to identify the programming language used in code images, among *many languages*[1], without any *a priori* knowledge about the languages. In fact, in literature, only a few languages are considered in similar works. Also, it is not yet established in the literature *what* allows image-based ML models to visually recognize programming languages, especially at this scale of language diversity. Such knowledge would allow the future to specialize in recognition networks and improve performances.

Both text-based and image-based strategies are exploited here.

In order to follow these leads, the first step consists in building the dataset that will be exploited to train the model. For the text-based model, we focused on entire files containing source code texts while for the image-based approach the focus is on snippets composed of 32 lines of code each. To build these datasets we used snapshots of GitHub dated (D_1) and (D_2) which have been retrieved and made available by the Software Heritage project.

The datasets D_1[2] and D_2[3] contain files from all commits of GitHub projects. The number of *stars* has been used as a filter to avoid the processing of many unreliable data as the stars are used to evaluate projects' quality and relevance, in this way the number of *junk files* found within the dataset should be limited. D_1 contains projects ranked with 1000 or more *stars* while D_2 contains only the 1000 top GitHub repositories as the filter based on the number of stars would include too many repositories at the D_2 extraction date.

Each file within the dataset is identified by a SHA1 cryptographic checksum which differs from the filename provided by GitHub and can be used to uniquely map files. In fact, the same file could be found on GitHub several times and consequently, it can be extracted multiple times. For this reason, it can happen that the very same file is associated with different names, and to overcome the

[1]An amount comparable to the language diversity supported by practical state-of-the-art PLI tools (machine learning-based or otherwise), in the order of hundreds.

[2]`https://annex.softwareheritage.org/public/dataset/content-samples/2017-01-27-github-1000+stars/`

[3]`https://annex.softwareheritage.org/public/dataset/content-samples/2019-10-08-github-top1k/`

difficulty in files identification the file deduplication has been performed on the files extracted from GitHub and the SHA1 identifier is used as data ID. After data deduplication, data compression is applied to the extracted snapshots in order to easily store the datasets and make them downloadable `tar` archives. D_1 is a 141 GB archive containing $\approx 15M$ unique files while D_2's size is 252 GB with $\approx 25M$ unique files.

SHA1 values are also used to compare the two datasets D_1 and D_2 to check if there are some overlapping portions. This analysis shows that the datasets result to be two disjoint sets and, as we will see, D_2 can be fairly used to test model performance after a few years.

Together with the files, the dataset contains a `txt` file which represents the map between the original file names retrieved from GitHub and the corresponding SHA1 currently associated with the file after the deduplication, so a SHA1 ID can be associated with different file names.

At this point, two main issues need to be solved. First, the datasets are not composed of source code files only, but they also include any kind of file that can be found in a repository, for example, images, pdfs, binary files, simple texts, and so on. As the main aim of this work is to create a model which is able to recognize the programming language in which a source code file is written in non-code files that need to be discarded. So, the first problem arises because there is not a direct and precise way to perform this first crucial cleaning step. Secondly, we are going to use supervised learning models to classify source code into programming language classes. The problem is that our files are not associated with any ground truth about languages without which the supervised approach would be impossible. As we have seen in Section 2.2 many works in literature approach this problem by using existing tools that can predict the language with a certain accuracy, for example, one of the most popular tools is *Linguist* which is the official language detection tool used by GitHub. We follow this lead to label data used in the image-based approach.

However, adopting this method we are limited in imitating the *Linguist* predictions and even if the model would reach the perfect match between labels and predictions it would replicate the *Linguist* behavior and, for this reason, it will inherit the *Linguist* accuracy, precision, recall, etc. [64]. Except for this exploitation of already existing tools, there is only the manual annotation solution which would be impossible for our amount of data. For the text-based approach, we choose another option to label the files. The only available information about our files that could be somehow informative with respect to the programming language is the file extension with which the files have been stored on the GitHub platform. In fact, in the case of a source code file, the extension used when such a file is stored depends on the programming language used to write the code contained in it. For example, a Python source code file should be saved with a filename ending with `py` or similar file extensions. The file extensions can be found in the file containing the mapping between file names and SHA1 checksums. In fact, a file extension is defined as a substring that can be extracted from the filename starting after the last occurrence of the dot punctuation symbol (.) until the end of the filename if the filename contains the dot symbol, otherwise we say that the file extension is not available for that filename. Of course, there is no certainty of the correctness of the file extension association as the authors could have committed errors in the file storage processes. Despite that, we assume that these wrong associations occur with a statistically irrelevant frequency within the dataset, and they should not affect the training and evaluation of the model too much. From now on the extracted file extensions are considered the ground truth about the files belonging to our datasets and, for this reason, the task that we are going to perform in the text-based approach is slightly different from the aim that we planned to achieve at the beginning which was to predict the programming language of a source code file.

In the following of this Chapter, we are going to illustrate the two models used to approach the PLI task. We are going to illustrate the textual-based model for file type identification in the next Section while the image-based PLI is described after this.

3.1. File Extension Identification

Using file extensions as labels the classification that we are going to perform will be different from the programming language identification as the classes are defined by these extensions which are not mapped to languages. An issue that can arise analyzing this task is intrinsic in the definition of the classes and it refers to the ambiguity related to the file extensions. First, there is not only one extension for every file type, for example C++ source code files can use `cc`, `cpp`, `c++` and other extensions. For this reason, some extension classes could be seen as if they were equivalent from the programming language identification point of view and for their content. The fact that several classes could contain the same kind of files can create confusion for the model that has to learn to classify files in different classes while they actually belong to the same file type. Secondly, a single file extension can be shared between more than a single kind of file as in the case of `txt` files which usually contain general textual content. So, even if one can try to disambiguate these classes' definitions by means of domain knowledge, i.e., joining extensions that refer to the same kind of files in a unique class, this process is not simple as it may seem, in fact, some of these extensions can be shared with other kinds of files. Moreover, one has to consider the complexity of this task, for example, if there are a lot of different extensions and/or some of them correspond to unpopular file types the required effort to recognize and effectively cluster all the involved extensions could be unfeasible for a *human expert* and the task itself can represent a source of errors. From all these considerations has been decided to accept the intrinsic ambiguity concerning the file extensions and file types definition proceeding to work with the extensions as they have been extracted from the file names without any modification to avoid the introduction of further confusion and possible errors.

3.1.1. Preprocessing

After the extensions extraction, i.e., the labeling step, there is the need to perform some analysis on the extensions that have been found.

Labels frequency At this stage, the number of different extensions on the D_1 is $\approx 546K$, despite this high number if we study the distribution of the files among all these classes it is immediately clear that a huge portion of them only contains a few examples while only a few hundred have a relevant number of instances. Generally, some problems arise in supervised learning when dealing with classes with too few examples because they are not enough to teach them generalizations about those classes, so we need to deal with underrepresented classes. Most of these classes seem to be typos or very uncommon sequences of characters that do not match any particular kind of file. This fact, together with the very low frequency with which these classes are found within the dataset, suggests simply ignoring the problematic classes and proceeding with the analysis of the remaining ones. In order to do so, a frequency threshold of 10^{-4} has been chosen and all the classes which show a lower frequency and the files belonging to them are excluded from the new dataset version. After this filtering step we end up with 220 extensions while the number of files is still $\approx 15M$, so the frequency-based filtering step drastically dropped the number of classes but did not impact the number of examples too much.

Non-textual files The other problem is about the potential presence of any kind of file and some of them could not be of our interest if we consider the aim from which we started, i.e., PLI. To find these files and clean as much as possible the dataset from them a manual analysis of the 220 remaining extensions has been attempted but it resulted in tricky in many cases due to their ambiguity. Moreover, some extensions were very hard to be identified with respect to the actual content of the files they referred to, even after extensive research. As many sources of confusion were aroused in this treatment of the dataset, we decided to automate this preprocessing step to avoid unnecessary human intervention that can bias the model and which does not represent a viable direction to build scalable models. So, instead of relying on *expert* knowledge or specific research about each extension, we implemented an automatic method based on a heuristic. In this way, we limit our influence on the files choices to the heuristic design only. As source code files have text-like content, at least we should restrict the dataset on these kinds of data, and discriminate between these and others we assumed that the content of files of our interest should have a percentage of *non-printable* characters which are not higher than a certain threshold, i.e., our assumption is that source code files and more generally textual files should not be *dense* in non-printable characters and we set the non-printable frequency threshold at 20%. This step permits us to split the files of the dataset into two groups based on their non-printable characters percentage and to perform a distribution analysis of these two groups among the extensions classes that we were considering. Some of the extensions classes are indeed mostly populated by files with a percentage of non-printable characters higher than our threshold and we considered these classes as not relevant to our task. Discarding the classes with high non-printable characters rate the number of remaining extensions for the D_1 dataset drops to 133, while the number of files results to be $\approx 13M$.

Most of the retained extensions are commonly used as extensions for programming or markup languages, in fact the extensions associated with the highest frequencies are `py`, `rb`, `html`, `po`, `php`, `h`, `java`, `js`, `c`. Despite this, the dataset also contains extensions that do not refer to programming languages, for examples some of them are typically associated with textual files of other nature like in the case of `txt` files but we continue to consider these cases too as we made the decision to automatize the extensions selection phase in the previous steps.

Multi-label files As we mentioned before, there is the possibility that a unique file (identified by a precise SHA1) can be associated with multiple filenames, and, consequently, there is also the possibility that a single file can be linked to more than one extension, i.e., with multiple labels. Every SHA1 checksum has been associated with the corresponding extensions extracted from every file name the given checksum is linked to.

For each file, a given extension can occur multiple times, for this reason, every extension associated with that file has been coupled with a frequency that indicates how often the extension has been found in the filenames associated with that file. For example, if the SHA `file_SHA1` appears three times in the mapping between SHAs and filenames and it is associated with `filename1.cpp`, `filename2.c` and `filename3.cpp`, the file has a label of the form ($\frac{1}{3}$`c`, $\frac{2}{3}$`cpp`). However, within the whole dataset, only a small percentage of files has been found coupled with multiple extensions (less than 0.4%), in the following we simplified the task by ignoring these multi-label occurrences and treating the problem as a multi-class single-label classification case with 133 classes of extensions. Besides the low number of multi-labeled files, this simplification is justified by the fact that we conducted some tests even considering the multi-label task but this resulted in low performance of the model with respect to the model for the single-label task.

Adaptation of the D_2 dataset In this work, we use the D_2 dataset to test the model prediction capability after a few years to check which measure of the model still represents a valid solution to the PLI task. The results obtained from the previous cleaning procedure have been applied to the D_2 dataset without repeating the whole cleaning procedure in order to inherit all the outcomes from the dataset of a past date. Particularly, we retained only the files belonging to the 133 previously selected extensions. However, some of these extensions do not have any example in the second dataset and only 121 of the 133 extensions are represented there.

Handling unbalance Even though we ignored rare and uncommon extensions, the frequency range remains wide, with some classes containing many examples while others just a few, as shown in 3.1. The most frequent extension results to be `c` and it has a frequency of 10^{-1} while the least frequent one is `tcl` and has a frequency of 10^{-4}.

This frequency imbalance represents a serious issue with respect to the training phase for most of the supervised learning approaches, since models which learn from unbalanced datasets have the tendency to overfit and do not generalize well [18]. After splitting the dataset with an 80/20 train/test ratio the train set part needs to be treated in order to create a balanced set.

Several techniques can be used for dataset balancing, and they are usually divided into two major classes: over-sampling and under-sampling [117]. In the first case, new instances for the minority classes are generated until they reach a numerosity similar to that of the majority class(es). The new instances

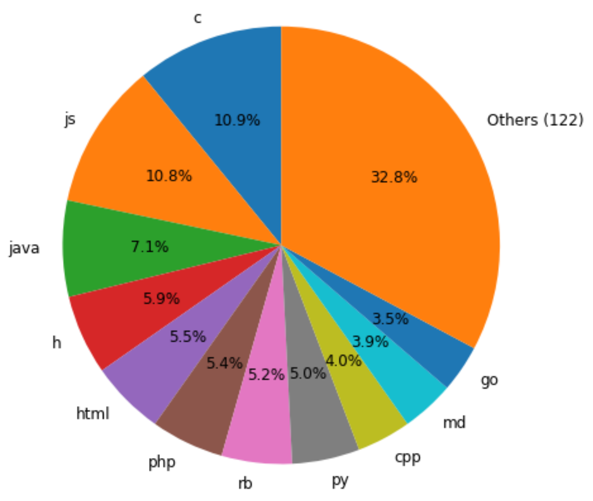


Figure 3.1.: Extensions distribution within the dataset corpus.

are either copies of the existing examples (e.g., in random over-sampling) or synthesized by using statistical properties computed on the minority classes (e.g., in the SMOTE technique [30]) and there are several techniques to do so.

Since in our case the classes are highly unbalanced, we could be forced to use the same examples (or the same information extracted from a few examples) too many times, increasing the risk of overfitting. We did not consider the option of synthesizing artificial samples to avoid introducing biases, given that understanding how intrinsically recognizable are *real* textual files found in Version Control Systems (VCSs) is part of our goal. Rather, we have applied a random under-sampling technique consisting of (sub)sampling the various classes randomly to obtain the same number of elements for each class. Of course, with this approach we are limited by the number of instances of the less populated class and, as a result, we ended up with a train set containing ≈13M total examples.

Conversely, the portion of the dataset used to evaluate the model preserves the original dataset classes distribution in order to fairly represent a real situation and to evaluate the results as in an actual PLI model exploitation.

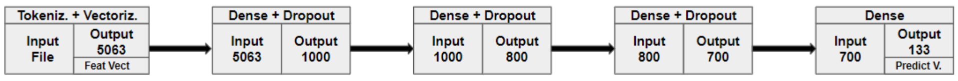


Figure 3.2.: File type classification model.

3.1.2. The model

In this section, we describe the architecture of the model that has been proposed to predict file types automatically which has been developed in several steps. First, the content of the files is divided into tokens (using a language-agnostic tokenization strategy) which are then used to define a reference vocabulary V based on the frequencies of the tokens in corpora. The file content is then represented in a “tokenized” form, i.e., a sequence of tokens according to the defined vocabulary V. If the tokens sequence file form contains some tokens which do not belong to the V vocabulary, these tokens are represented as a placeholder for unknown tokens that we indicate here as `UNK`. Then, as common practice in many NLP applications, we also construct a vocabulary V_2 containing 2-grams which are sub-sequences of 2 consecutive tokens extracted from the “tokenized” files. At this stage, we can extract the actual feature vectors from the files by considering the frequencies of both tokens and of 2-grams in the “tokenized” files[4]. Finally, the classifier which makes predictions on the basis of the feature vectors is defined, by using a simple neural network whose structure is depicted in Figure 3.2; these steps are detailed in the following of this section.

Data representation

Since our files are available as sequences of bytes, to treat them as texts we first need to decode them using a suitable character encoding. We used the ASCII encoding while all non-ASCII characters are mapped to a unique special value. After this conversion, we can consider files as simple text files, i.e., sequences of ASCII characters, on top of which we can define the notion of tokens.

Different from what is usually done in Natural Language Processing (NLP), case sensitivity is relevant in our setting, hence is important to preserve character case-ness. Also, while in NLP punctuation symbols are usually discarded, they are crucial in source code and we need to consider them. [5]

[4]Not all possible tokens and 2-grams are considered but only those appearing in the vocabularies V and V_2, respectively.

[5]With the exception of _ which is considered an alphanumeric character, as it is often part

Tokenization and vocabulary definition For the tokenization step we exploit the following definition:

Definition 3.1.1. Given a sequence of characters S, a *token* (or equivalently a *word*) in S is defined as follows:

- Any character representing a punctuation symbol is a token
- Any sub-sequence of S which is delimited by (characters representing) punctuation symbols and/or white spaces, and which does not contain punctuation symbols and/or white spaces is a token.

For example, according to this definition the string '`a=b`' is interpreted as a sequence of the three tokens '`a`', '`= `' and '`b`'.

For model manageability, we cannot consider all the possible tokens that occur in any file. A common technique used to address this issue is looking at the frequencies of the tokens that occur within the train set and assembling a vocabulary consisting of all the tokens whose frequency is higher than a given threshold. In our case, several thresholds have been tested but in the end, the 10^{-2} one has been selected.

To mitigate overfitting risk due to the usage of the same dataset in vocabulary definition and model training we have defined the vocabulary V using only a portion of the train set (still a balanced set), which was then excluded from the set used to train the network [165].

Many files in code bases include at their beginning and/or end explicit information about the file content, in the form of shebang lines (e.g., `#!/usr/bin/perl`) or editor mode lines ((`%% mode: latex`)). This information can be really helpful for the classification task, but it can also compromise the performance of the model since this information could gain too much relevance with respect to other features, inducing poor results on files that lack it. For this reason, when collecting tokens to build the vocabulary and during training, we excluded a few lines from both the beginning and end of the files to ensure that no explicit information about the language is used by the model.

The resulting tokens vocabulary V contains 465 tokens, which are represented with their own identity while the special token `UNK` represents any unknown, Out-Of-Vocabulary (OOV) token. Figure 3.3 contains a word cloud representation of the tokens (where word size is proportional to the frequency of the token in the dataset) of the V vocabulary except for punctuation symbols.

of identifiers in source code.

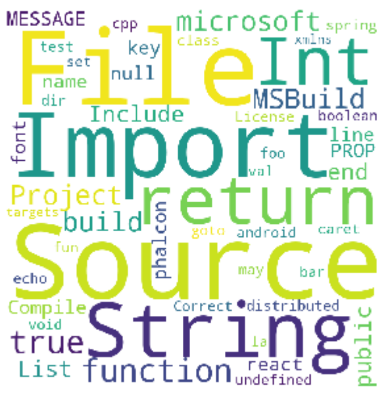


Figure 3.3.: Word cloud for token frequency distribution.

n-grams Information about the relative position of tokens in a text can be richer than information about isolated tokens. There exist various algorithms and techniques that can capture different kinds (e.g., short or long) of the relation among tokens. Some of them like CNN and LSTM [203, 202], can capture meaningful relations, automatically but they are also computational quite expensive. Simpler approaches are based on n-grams, i.e., sub-sequences of n tokens extracted from a sequence of tokens defined according to a given vocabulary V.

Taking into account n-grams, instead of individual tokens, it is possible to identify *co*-occurrences of tokens, extracting more information about the actual text structure. By increasing the length of the n-grams (the value of n) it becomes possible to capture longer and more complex relations among tokens, at the cost of increased computational costs and increased overfitting risk—since longer n-grams tend to become tightly bound to the text they are derived from. For this work, we have used bigrams, i.e., $n = 2$, which turned out to be a good choice from the model performance point of view. We have also experimented with models with trigrams which have worse performance.

To define the bigram vocabulary, we relied on the same approach used for building the token vocabulary V. We define V_2 as the set of all bigrams whose frequency is higher than 10^{-3}) in each class, by considering the same dataset subset used to define V.

Bigrams that do not belong to V_2 are mapped to the placeholder for unknown bigrams $\texttt{UNK}_2$.

Vectorization For each input file, F one can now build the feature vector v_F, which will be the representation of F in our model. To build this vector we first decode F from a byte sequence to a characters sequence using the ASCII encoding and then tokenize it according to the vocabulary V. We then compute, for each token in V, the frequency with which it is found in the F tokenized form. Then we do the same for bigrams: for each bigram in V_2, its frequency among F's bigrams is determined. Finally, we enumerate the elements of the set $D = V \cup V_2 \cup \{\texttt{UNK}\} \cup \{\texttt{UNK}_2\}$ whose cardinality is $|D| = 5063$ and determining a fixed order for them.

The feature vector v_F is built by assigning the computed frequency for of i-th element of the ordered version of D to the i-th component of the vector itself. v_F will represent the file F in the following.

This process is applied to each file in the dataset; the resulting vectors will be used as inputs for the classification algorithm. The same has been performed taking into account trigrams too but the best results have been achieved using just tokens and bigrams.

Classifier

The model implementation has been done using the *Keras* framework [36] for Python.

As a classifier we use a Deep Fully Connected Layers (FCL) Neural Network which has 5063 input units, 133 output units (which correspond to the possible extensions that we are considering), and 3 hidden layers with 1000, 800, and 700 units, respectively, defining an encoder-like structure for the network. We set a dropout rate of 0.5 for each layer. The model structure is shown in Figure 3.2.

The problem is represented as a multi-class, single label classification, hence we use in the output layer the *softmax* activation function

$$\sigma\left(\vec{x}\right)_j = \frac{e^{x_j}}{\sum_{k=0}^{132} e^{x_k}} \tag{3.1}$$

which normalizes the values obtained from the previous layer with respect to the number of the available classes. The output values computed in this way represent the probabilities that a given file belongs to each class.

As we have multiple classes, we use the *categorical cross entropy* loss function. For each instance passed to the model, i.e., the i-th one, the loss function takes the form

$$L\left(y_i, \hat{y}_i\right) = -\sum_{k=0}^{132} y_{i;k} ln\left(\hat{y}_{i;k}\right) \tag{3.2}$$

where y_i represents the actual ground truth label (in the form of a one-hot encoded vector) and $\hat{y}_i$ represents the predicted probability output vector.

During the training phase we use the *Adam* optimizer [93] with learning rate $lr = 0.0001$, which converge to the results shown in 3.1.3 after 8 epochs of training. During the training of the model, parameters are progressively modified in order to improve the similarity of the predicted $\hat{y}_i$ vectors to the correspondent y_i ground truth vectors.

3.1.3. Results

The architecture underlying the proposed model is quite simple in comparison to other machine learning approaches used to treat text. Contrarily to automatic feature learning algorithms, such as those used in Convolutional, Recursive, or Attention Neural Networks, computations in our model are faster, both for training and prediction with the drawback of the needing for manual design for features.

The learning task can be completed in a relatively short time: it took around 10 hours of training for 8 epochs to train the parameters of the model and to reach $\approx 85\%$ of accuracy 2.1 on the validation and test sets.

$$Acc = \frac{\sum_{i=0}^{N_c-1} C_{ii}}{\sum_{i,j} C_{ij}}$$

Predictions are made by transforming the input feature vectors by means of simple operations such as vector multiplications, sums, and activation functions applications based on the parameters learned during the training phase.

Testing on D_1 dataset The test set consists of $\approx 2M$ elements that we extracted from the original dataset. It is subjected to the same pre-processing steps described at the beginning of this Section except for the balancing step as this could bring unfair evaluations and the resulting test set would not represent the real-world statistical distribution of classes.

Various performance measures are used here and they are all based on the confusion matrix computed on the test set whose generic element C_{ij} contains the number of files of class (extension) i which are classified as j.

As we mentioned, in addition to bigrams also trigrams have been tested in this study and the results for each class can be found in Table A.1 in Appendix A where the values for the precision P_i (Eq. 2.2), recall R_i (Eq. 2.3) and F_1-score (Eq. 2.4) for each of the considered classes are reported ($N_c = 133$ is the total number of considered classes) referred to the test set extracted from the D_1 dataset.

$$P_i = \frac{C_{ii}}{\sum_{j=0}^{N_c-1} C_{ji}} = \frac{TP_i}{TP_i + FP_i}$$

$$R_i = \frac{C_{ii}}{\sum_{j=0}^{N_c-1} C_{ij}} = \frac{TP_i}{TP_i + FN_i}$$

$$F_i = \frac{2R_iP_i}{R_i + P_i}$$

	bigrams			**trigrams**			**Δ**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
micro avg.	0.85	0.85	0.85	0.81	0.81	0.81	0.04	0.04	0.04
macro avg.	0.64	0.91	0.71	0.59	0.89	0.66	0.05	0.02	0.05

Table 3.1.: Average performance of the encoder architecture without and with trigrams.

In Table 3.1 we only report micro and macro-average values for the selected scores.

The accuracy value obtained on all the classes by the model is 0.85 in the bigram case and 0.81 if also trigrams are considered and the results for the latter model are almost always worst than the ones obtained from the bigram one. Moreover, including trigrams requires handling a higher-dimensional input (as we need to add the trigram vocabulary V_3 to the D set) and a heavier model which could be less efficient than the version which considers only bigrams so we do not consider the trigrams in the rest of the study.

From the results, it appears that the main problem faced by the model is the presence of very similar classes that could actually be treated as the same class, for example, both `.yaml` and `.yml` are used for files written in YAML, which can introduce errors in the predictions. Also, there are classes whose text contents can be very general and therefore could not be easily recognized, such as files with the `.txt` extension. This suggests that including explicit knowledge about the files' nature could improve the performance, however, it was our design choice not to include such information.

Classes confusion We tried to keep track of the classes which are most likely confused in an automatic way without introducing any prior knowledge about classes. To do so we keep track of classification errors on the validation set in two different ways.

Definition 3.1.2. In the following we will use the following assumptions and notations:

- C is the set of classes
- $Vset$ denotes our validation set
- for $x \in Vset$, $GroundT(x) = i$ iff i is the ground truth label of x
- for $x \in Vset$, $Predict(x) = i$ iff i is the label assigned by the classifier to x
- for $x \in Vset$, $p_x(i)$ is the predicted probability value that the input x belongs to each possible class $i \in C$

For $x \in Vset$, the predicted class is the one which correspond the highest predicted probability value, that is, $Predict(x) = m$ iff $p_x(m) \geq p_x(j)$ for each $j \neq m$, $j \in C$.

First, we measure how many of the examples of a given class are classified in the wrong way. This is done by introducing, for each pair of classes i and j, a quantity T_{ij} indicating how many times a file whose label (i.e., ground truth) is i is classified as belonging to the class j.

More precisely we define:

$$T_{ij} = \frac{\sum_{x \in Vset} 1 \mid GroundT(x) = i \wedge Predict(x) = j}{\sum_{x \in Vset} 1 \mid GroundT(x) = i} \tag{3.3}$$

A second way to keep track of possible classes confusion consists in considering also the second, third, fourth, and fifth-best choices for classifying an input according to the predicted probabilities $p_x(i)$ with respect to the classes to see whether some of them significantly co-occur with the predicted class. More precisely, given $x \in Vset$, assume that $Predict(x) = m$ and that $p_x(m) \geq p_x(h) \geq p_x(k) \geq p_x(l) \geq p_x(r) \geq p_x(i)$, for $h, k, l, r \in C$ and for any other $i \in C$. In this case we say that m, h, k, l, r are the top five classifications for x, written $Top(x) = \{m, h, k, l, r\}$, for short. Then we normalise these five values by defining:

$$pn_x(j) = \frac{pr_x(j)}{p_x(m) + p_x(h) + p_x(k) + p_x(l) + p_x(r)} \tag{3.4}$$

for $j \in \{m, h, k, l, r\}$.

For each pair of classes $i, j \in C$, we define:

$$S'_{ij} = \sum_{x \in Vset} pn_x(j) \mid Predict(x) = i \wedge j \in Top(x) \tag{3.5}$$

and we normalize this value by considering the total number of times in which the i-th class has been predicted, as follows:

$$S_{ij} = \frac{S'_{ij}}{\sum_{x \in Vset} 1 \mid Predict(x) = i} \tag{3.6}$$

Given the quantities T_{ij} and S_{ij} we set two thresholds for their values, τ_T, and τ_S, respectively. For a pair of classes i and j, if $T_{ij} > \tau_T \vee S_{ij} > \tau_S$, the classes i and j are considered "related" in the predictions and we say that they belong to the same "*confusion group*". By setting the thresholds $\tau_T = 0.05$ and $\tau_S = 0.02$ we obtain the confusion groups shown in Table 3.2. Note how some groups contain labels that commonly refer to the same or similar file types, justifying the origin of the ambiguity. The labels which do not appear in the table are those that do not pass the thresholds, meaning that the classifier does not incur relevant ambiguity for them according to our defined measures.

Table 3.2.: Extension confusion groups

Group ID	Extensions
0	`.bash` , `.sh` , `.ps1` , `.after` , `.jet` , `.kt` , `.template`
1	`.markdown` , `.md`
2	`.cmake` , `.cmd` , `.yaml` , `.yml` , `.rst` , `.txt` , `.baseline` , `.bat`
3	`.dts` , `.dtsi`
4	`.ctp` , `.php`
5	`.ml` , `.mli`
6	`.csproj` , `.ilproj`
7	`.jl` , `.j`
8	`.rb` , `.cr` , `.exs`
9	`.h` , `.ino` , `.hpp`
10	`.m4` , `.ac`
11	`.cpp` , `.cc`
12	`.clj` , `.cljs`
13	`.swift` , `.sil` , `.gyb`
14	`.tsx` , `.jsx` , `.js` , `.ts` , `.htm` , `.html`
15	`.cjsx` , `.coffee`
16	`.css` , `.scss`
17	`.after` , `.kt`

Considerations about the D_2 dataset As previously mentioned, we also considered a more recent dataset D_2 which after pre-processing contains 121 of the classes considered in our model.

The D_2 dataset has been split into two halves maintaining the classes' frequencies distributions. One of these two subsets is used to test the model and the other one has been joined to the previously used trainset and balanced. The second portion is then used to re-train the model for 3 epochs which are then tested again on the first D_2 portion.

The average results obtained on the new test set before and after the updating step are shown in Table 3.3. Analyzing these results, it does not seem that

	Before			**After**			**Δ**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
micro avg.	0.88	0.88	0.88	0.91	0.91	0.91	0.03	0.03	0.03
macro avg.	0.69	0.89	0.77	0.71	0.92	0.80	0.02	0.03	0.03

Table 3.3.: Average performance of the bigram model before and after the re-training phase on the 2019 testset

after two years the performance drops, however, the updating step brings some improvements suggesting that monitoring this aspect is important and careful periodical updates of the model (both in the vocabulary definition and in the training) can be needed to maintain good performance, trying to avoid the so-called, catastrophic forgetting phenomenon [121]. This phenomenon often shows up when one re-trains a network on new data which are different from those used in the first training session. The model tends to forget what it learned during the first session, thus producing relevant worsening in the performance. Probably, as in our case, the difference between old and new data is not that relevant, we obtained good results by simply mixing the new train set extracted from D_2 with the one extracted from D_1. By using this updating approach, the updated model reached an accuracy score of 91% on the (mixed) test set (obtained joining the test sets from D_1 and D_2), 91% on the test set extracted from D_2 and 92% on the test set extracted from D_1.

3.1.4. Threats to Validity

External validity. Even if the proposed classifier has been designed and tested on a fairly large set of files, the used dataset falls short of the entire corpus of source code distributed via publicly accessible version control systems. Datasets that approximate that corpus [2, 50, 116] do exist and could potentially be more challenging to tackle because: (1) they will include more extension/classes that did not occur within this paper's datasets, (2) they can include noisier data as repository popularity is likely a proxy of project quality in terms of coding practices, and (3) they can include files from more varied chronological epochs—also programming languages always evolve and new ones emerge increasing the variety of classes to be considered. Nevertheless, our design choices were oriented to address this kind of problem, as we discussed before, and we aim to use the proposed approach on these larger datasets in future work.

Internal validity. Various (non-domain specific) heuristics have been applied. First, filtering of non-textual files has been performed based on the percentage of non-printable characters within a random sample of the datasets; different samples or thresholds might affect stated results. On the same front, the choice of using the ASCII encoding can bring the lack of some encoding information. An alternative approach would have been guessing the used encoding (using libraries such as `chardet`); it is not clear which biases either approach would introduce if any.

The vocabularies V and V_2 are based on tokens and bigrams frequency distributions defined separately on different classes. This can introduce model biases and could be mitigated by using separate datasets to define the vocabularies and to train the neural network. Hyperparameters tuning has been performed as an iterative process, certainly not exhaustive. In spite of the achieved good results, it is possible that different choices for hyperparameters and/or neural network topology would score even better. At the end of the previous section we mentioned some possible threats due to extensions classification, in particular, some of the extension classes that we treated as different might actually refer to the same abstract file type, which can cause confusion for the model and makes the training task harder as a class shares its instances with other classes. It is difficult to mitigate this last threat without relying on domain-specific knowledge.

3.1.5. Discussion and future work

In this work we considered the problem of predicting textual file types for files commonly found in version control systems (VCS), relying solely on file content without any a priori domain knowledge or predetermined heuristic. The problem is relatively novel (as most existing language/file type detectors rely heavily on extensions as input features) and relevant in contexts where extensions are missing or cannot be trusted, and shed light on the intrinsic recognizability of textual files used in software development repositories.

We propose a simple model to solve the problem based on a universal word tokenizer, word-level n-grams of length 1 to 2, feature vectors based on n-gram frequency, and a Fully Connected Layer neural network as a classifier. We applied the model to a large dataset extracted from GitHub spanning 133 well-represented file type classes. Despite its simplicity, the model performed very well, nearing 85% average accuracy, and outperforming previous work in either accuracy, number of supported classes, or both. We expect that model simplicity will make it more maintainable in the future and less computationally expensive to train and run than alternatives based on learning algorithms such as CNN, LSTM, or Attention.

Our biggest achievements are due to the effectiveness of the model in presence of many classes with a relatively simple model. Hopefully, this remains true also if we increase this number but more inspection about this needs to be done. Also, we used the actual label that we find for the files without any additive manipulation, i.e., employing outcomes of other predictors or transformation.

In addition, we performed a test on the validity of the results after a few years and evaluated an updating strategy for the model to inspect its long-term quality.

Many more kinds of extensions have been found in our dataset, however, only 133 of them have been selected in our study because of their nature (we tried to exclude non-textual file types) or their popularity (we excluded infrequent classes). A possible model modification could focus on including these neglected classes as belonging to an extra class, i.e., identified as *other*, to which all the excluded file types belong. However, such a class would be highly heterogeneous as it could be populated by both binary and textual file types of very different nature. This extra class would be useful to classify all the files which do not fit the other classes according to what the model learned and could be helpful in a real-world application in which we do not know if the actual class belongs to our selected extensions set. This aspect can be implemented and investigated in future works.

Concerning future work, a straightforward next step is scaling up experimentation of the proposed model, moving from the high-starred GitHub dataset we used for this work to larger and more diverse datasets such as Software Heritage [2]. In that context, we will have significantly more starting classes and hopefully enough samples in each of them for enlarging the set of labels actually used in training. As we observed, extensions alone are ambiguous in many cases and this poses challenges in training and evaluation. To mitigate this issue, it is worth exploring the possibility of inserting narrow domain knowledge about file extensions that often go together. It is not clear whether doing so, partly backtracking the "no domain knowledge" assumption of this work, would be worth the effort in terms of increased accuracy; hence, it is worth exploring. An alternative approach for improving the current handling of model confusion is adding a second tier of classifiers, one for each class of ambiguous extensions, a popular technique in NLP. There are various methods to combine the predictions from the first and second-level classifiers which should be explored. We have briefly explored the topic of accuracy degradation in the lack of retraining at a 2-year distance. A more general characterization would be interesting to have and is feasible to obtain by exploiting historical software archives and/or VCS timestamp information. Such a characterization will allow devising data-driven approaches for when and how to retrain file type and language detection tools in a world in which programming languages constantly evolve.

3.2. Image-based Programming Language Identification

In this Section, another approach to Programming Language Identification is described. The dataset used here is the D_2 dataset introduced in the previous section, but the pre-processing phase is defined in a different way as the final data representation that we aim to use here is completely different from the other one. Image representation for source code is used and an appropriate model needs to be selected to treat such kind of data. Convolutional Neural Networks (CNNs) [137] represent the state-of-the-art and the most used neural network architectures for image recognition. CNNs are commonly used for image-based PLI and our approach relies on them as well, with two notable differences from related works in the literature: a large number of classes to be recognized and the use of transfer learning.

Transfer learning [211] is a well-known machine learning approach that, rather than training models from scratch for a specific classification task, starts from a model that has been pre-trained on a related domain and then adapts it to the target domain. The key advantage of transfer learning is that it allows obtaining good classification performances while using a reduced training set and therefore reduced training costs. Whereas we did have enough data in our dataset to start from scratch, training cost remains an important concern in PLI because, due to the rapid evolution of source code artifacts in the target domain, one has to add to the *initial* training cost that of *periodic* retraining. This domain-specific aspect of the problem led us to the decision to use transfer learning.

Another key point of this approach stays in the data labeling strategy which differs from the previously used one and that counts on external tool prediction outcomes. The different labeling approach permits to end up with programming language labels instead of using the ambiguous file extension.

3.2.1. Data preparation

Labeling

After filtering out rare file extensions we run Linguist [69] (a popular choice for source of truth of PLI works in the literature) on all source code files having one of the remaining extensions and excluding all the files for which Linguist was unable to predict the language. The extensions are important as Linguist uses them as a feature to predict the language and without this information, its reliability drastically drops. At the end of this step, 212 languages remained in the dataset, together with the associated source code files. In this way, the classification labels are now the programming languages detected by Linguist and are no longer the file extensions.

Bundles and snippets extraction

At this stage, the amount of both files and Source Lines Of Code (SLOCs) in the dataset was highly unbalanced across languages. For example, popular programming languages such as Python or JavaScript occur in thousands of source code file examples, whereas other languages only had 100 examples or less. As we already pointed out, unbalanced datasets are a well-known issue for supervised machine learning we wanted to mitigate this issue, while at the same time avoiding both high file repetition rate (oversampling) and the exclusion of too many languages (downsampling). Avoiding downsampling is particularly relevant here since our goal is to assess the feasibility of high-accuracy image-based PLI with *many* languages.

We exploited the fact that we want to focus on code *snippets* images rather than entire source code *files* like in the previous work. As a first step, we created one source code *bundle* for each programming language concatenating together up to 1000 files randomly selected among all the files written in that language. Then we used a sliding window of 32 SLOCs (see Fig. 3.4) which randomly moves along the vertical axis of each bundle, extracting 2000 code snippets for each language. We took care of ensuring that the snippets belonging to the test set do not have overlapping SLOCs with the train set.

Bundles that did not contain enough SLOCs to allow the extraction of non-overlapping snippets for the test set have been excluded, thus obtaining a total of 149 recognizable languages and $149 * 2000 = 298000$ labeled snippets.

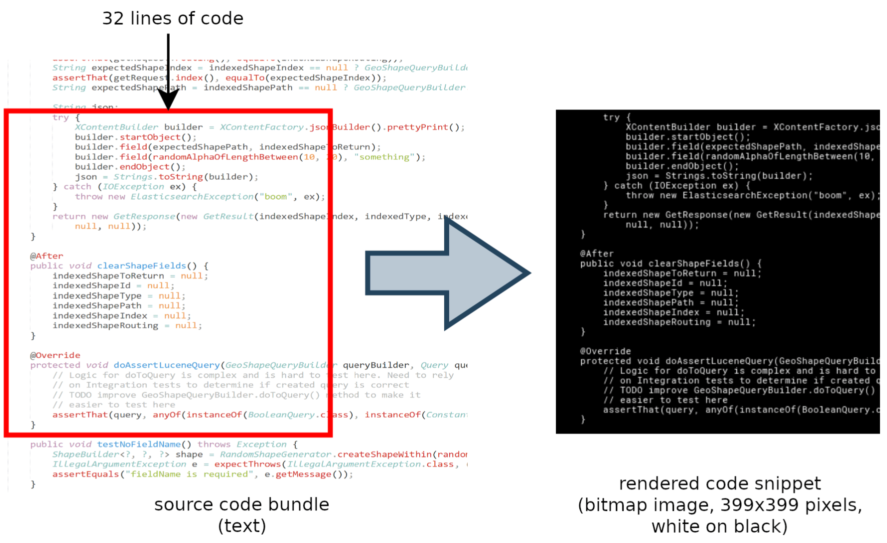
Images rendering

Each snippet has been rendered to a raster PNG image of size 399×399 pixels by using white on black text typeset in the Roboto Mono monospace font[6] with a font size of 11 points. Note that all obtained images are squared and have the same size, so trimming of long lines could happen as shown in Fig. 3.4.

3.2.2. The model

We compared the performances of three classifiers based on three different CNN architectures pre-trained for images recognition: ResNet34 [78], MobileNetv2 [83] and AlexNet [101].

[6]https://fonts.google.com/specimen/Roboto+Mono



```
    String expectedShapeIndex = indexedShapeIndex == null ? GeoShapeQueryBuild
    assertThat(getRequest.index(), equalTo(expectedShapeIndex));
    String expectedShapePath = indexedShapePath == null ? GeoShapeQueryBuilder

    String json;
    try {
        XContentBuilder builder = XContentFactory.jsonBuilder().prettyPrint();
        builder.startObject();
        builder.field(expectedShapePath, indexedShapeToReturn);
        builder.field(randomAlphaOfLengthBetween(10, 20), "something");
        builder.endObject();
        json = Strings.toString(builder);
    } catch (IOException ex) {
        throw new ElasticsearchException("boom", ex);
    }
    return new GetResponse(new GetResult(indexedShapeIndex, indexedType, indexe
        null, null));
}

@After
public void clearShapeFields() {
    indexedShapeToReturn = null;
    indexedShapeId = null;
    indexedShapeType = null;
    indexedShapePath = null;
    indexedShapeIndex = null;
    indexedShapeRouting = null;
}

@Override
protected void doAssertLuceneQuery(GeoShapeQueryBuilder queryBuilder, Query que
    // Logic for doToQuery is complex and is hard to test here. Need to rely
    // on Integration tests to determine if created query is correct
    // TODO improve GeoShapeQueryBuilder.doToQuery() method to make it
    // easier to test here
    assertThat(query, anyOf(instanceOf(BooleanQuery.class), instanceOf(Constan
}
public void testNoFieldName() throws Exception {
    ShapeBuilder<?, ?, ?> shape = RandomShapeGenerator.createShapeWithin(rando
    IllegalArgumentException e = expectThrows(IllegalArgumentException.class,
    assertEquals("fieldName is required", e.getMessage());
}
```

```
    try {
        XContentBuilder builder = XContentFactory.jsc
        builder.startObject();
        builder.field(expectedShapePath, indexedShape
        builder.field(randomAlphaOfLengthBetween(10,
        builder.endObject();
        json = Strings.toString(builder);
    } catch (IOException ex) {
        throw new ElasticsearchException("boom", ex);
    }
    return new GetResponse(new GetResult(indexedShape
        null, null));
}

@After
public void clearShapeFields() {
    indexedShapeToReturn = null;
    indexedShapeId = null;
    indexedShapeType = null;
    indexedShapePath = null;
    indexedShapeIndex = null;
    indexedShapeRouting = null;
}

@Override
protected void doAssertLuceneQuery(GeoShapeQueryBuild
    // Logic for doToQuery is complex and is hard to
    // on Integration tests to determine if created q
    // TODO improve GeoShapeQueryBuilder.doToQuery()
    // easier to test here
    assertThat(query, anyOf(instanceOf(BooleanQuery.c
}
```

Figure 3.4.: Source lines of code extraction from source code bundles and their rendering.

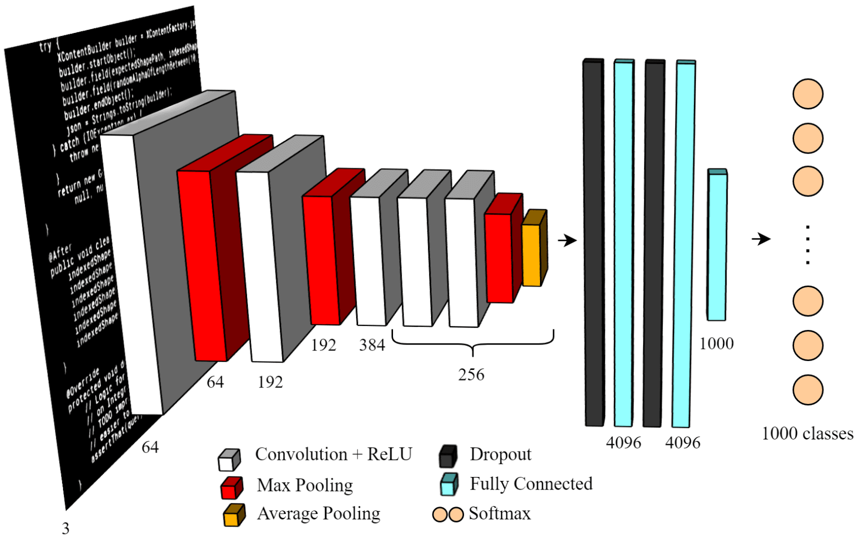


Figure 3.5.: AlexNet architecture, the simplest pre-trained CNN among those we specialized for visual code recognition

The first two have about 30 layers each, while the latter has 8 layers. We show in Fig. 3.5 the architecture of AlexNet, as it is the simplest to fully depict; the other two architectures are similar but significantly deeper. The three CNNs we used were all pre-trained on ImageNet [48], a generic image dataset composed of more than 14 million images classified into 20000 classes. This allowed us to benefit from the features and invariants learned on ImageNet in order to perform image-based PLI.

As a preliminary step in the model training, we replaced the classification layer (or *head*) of each CNN—initially composed of 1000 output neurons, as required by the ImageNet classification task—with a head composed of 149 output neurons, corresponding to the cardinality of our set of programming languages to recognize.

We applied a (usual) 80/20% split to our dataset twice to obtain the train, validation, and test sets. First, we kept aside 20% of the obtained code bundles for testing and then further split the rest to obtain the train and validation sets. This resulted in an overall partition of all code bundles into three sets: 64% for training, 16% for validation, and 20% for testing.

We then applied a two-step training procedure to all three CNN architectures. As a first step, the weights of the CNN (the *body*) have been frozen so that training could only affect the substituted head. This way the features previously learned by the convolutional layers during ImageNet training are exploited to make predictions about the new classification task.

After a few epochs of training, we moved to the second training step, where all the weights are unfrozen, thus allowing training updates all over the architecture. A slightly lower learning rate is used in the second training step with respect to the first so that the network can adapt to the task of image-based PLI without completely forgetting what the network has learned about images in general.

Table 3.4 shows the total training times for the three architectures, as well as a breakdown per training step, the number of training epochs, and the average per-epoch training time in each case. Training has been performed on a Linux machine equipped with an Intel Xeon 8 core 2.1 GHz CPU, Nvidia Titan XP GPU, and 96 GiB of RAM. The slowest architecture is MobileNet, which took around 7 hours to complete both training steps; the fastest architecture is AlexNet, requiring less than 2 hours of total training time.

Improvements in the training process, including finding a good balance between underfitting and overfitting phenomena, can be obtained by setting suitable values for several network hyperparameters. We mainly focused on tuning the learning rate (LR) while the other hyperparameters, such as epochs and batch size, have been kept at fixed values. For LR tuning we used the

	Training time/Epoch			**Epochs**			**Total training time**		
	A	M	R	A	M	R	A	M	R
step 1	6m33s	22m20s	20m04s	8	8	8	52m23s	2h58m39s	2h40m29s
step 2	6m49s	30m32s	26m46s	8	7	7	54m29s	3h44m41s	3h07m20s
total	13m22s	52m52s	46m50s	-	-	-	1h46m52s	6h43m20s	5h47m49s

Table 3.4.: Average training times per epoch (minutes and seconds), number of epochs and total training times (hours, minutes and seconds) for AlexNet (A), MobileNet (M) and ResNet (R) -based models for the 2 considered steps of training.

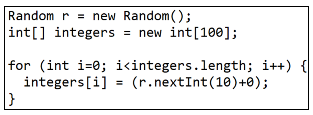
One Cycle Policy [166], where the LR cyclically varies within a certain range allowing improvements both in classification accuracy and training time. The LR range's upper bound has been determined during a pre-training phase according to a method recently proposed by Smith [167], which represents an efficient alternative to the common random search technique. We run one epoch of training starting with a small LR and gradually increasing it at each training iteration while recording the validation loss values. At the beginning the loss decreases, then reaches its minimum, and then starts to increase: such a minimum indicates the LR value that we have retained. The lower bound of the LR range was set to be $\frac{1}{25}$ of the upper bound.

3.2.3. Scrambling

When using CNNs for image-based PLI, learned features are automatically extracted by the network during training. In order to better understand what are the domain features (indentation, particular character classes, text placement, etc.) that allow the networks to visually recognize programming languages we selectively added noise to the code snippet images of the test set. This allowed us to determine which characters impact the most language identification capabilities.

We defined three classes of characters that are commonly used to define lexemes in the syntax of programming languages: *alphabetic* characters (denoted by A), *numeric* decimal digits (N), and *symbols* (S) for the remaining non-blank characters (mostly punctuation characters, mathematical symbols, and parentheses). Scrambling consists in replacing each character of a class being scrambled by another character, randomly selected within the same class while preserving string lengths. Fig. 3.6 shows some examples of scrambling results.

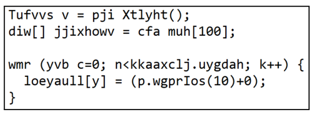
We first applied the scrambling to one character class at a time, without



```
Random r = new Random();
int[] integers = new int[100];

for (int i=0; i<integers.length; i++) {
  integers[i] = (r.nextInt(10)+0);
}
```

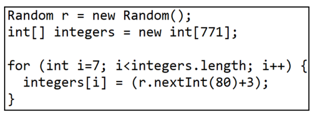
(a) original snippet



```
Tufvvs v = pji Xtlyht();
diw[] jjixhowv = cfa muh[100];

wmr (yvb c=0; n<kkaaxclj.uygdah; k++) {
  loeyaull[y] = (p.wgprIos(10)+0);
}
```

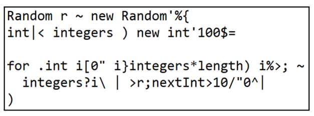
(b) scrambling alphabetic characters (A)



```
Random r = new Random();
int[] integers = new int[771];

for (int i=7; i<integers.length; i++) {
  integers[i] = (r.nextInt(80)+3);
}
```

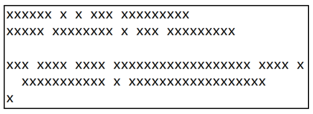
(c) scrambling decimal digits (N)



```
Random r ~ new Random'%{
int|< integers ) new int'100$=

for .int i[0" i}integers*length) i%>; ~
  integers?i\ | >r;nextInt>10/"0^|
)
```

(d) scrambling symbols (S)



```
xxxxxx x x xxx xxxxxxxxx
xxxxx xxxxxxxx x xxx xxxxxxxxx

xxx xxxx xxxx xxxxxxxxxxxxxxxxxx xxxx x
  xxxxxxxxxxx x xxxxxxxxxxxxxxxxxx
x
```

(e) replacing all non blank characters with lowercase `"x"` (X)

Figure 3.6.: Java code snippet in original form v. several scrambled variants

changing characters belonging to other classes, obtaining 3 scrambled test sets denoted A, N, and S. We then scrambled pairs of character classes together, obtaining 3 additional scrambled test sets denoted AN, AS, and NS; then we scrambled all the three character classes together, obtaining the scrambled test set denoted ANS. Finally, we considered the extreme case in which every character except blanks has been replaced by a (lowercase) `x` character, preserving only the overall code "layout", as dictated by code indentation, obtaining the scrambled test set denoted by (uppercase) X.

3.2.4. Results

For each architecture, the training phase has been performed on the original non-scrambled dataset and then repeated for each classifier and for each scrambled version of the dataset. In order to be able to compare results among the different architectures, we used a fixed PRNG (Pseudo-Random Number Generator) seed to make sure that images were processed in the same order during both training and evaluation.

3.2.5. Classification Results

On the non-scrambled dataset, after 8 epochs of the first training phase—in which only the weights of the classifier's head were able to be updated—the ResNet- and MobileNet-based classifiers reached $\approx$ 90% accuracy on the validation set, while the AlexNet-based model reached only $\approx$ 60%. Performances improved for all models after the fine-tuning phase—when all weights could be updated, although at a lower learning rate. After 7 epochs of fine-tuning ResNet accuracy reached $\approx$ 92% (+2%), MobileNet $\approx$ 93% (+3%), and AlexNet $\approx$ 84% (a significant +24%, but still the worst performing classifier overall).

The considered performance measures are the same considered in the previous work, i.e., precision P, recall R and F_1, the results obtained for each class are shown in Table B in Appendix B while in Table 3.5 we only report average results.

ResNet34			**MobileNetv2**			**AlexNet**		
P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
0.92	0.92	0.92	0.92	0.92	0.92	0.83	0.83	0.83

Table 3.5.: Average performance of the models for image-based PLI

Two aspects are worth noticing: first, performances range from acceptable to very good for all classifiers, with precision in the 83–93% range (depending on the base CNN) and recall in the 83–92% range. Second, the ResNet- and MobileNet-based classifiers significantly outperform the AlexNet-based one, probably due to the higher number of layers that allow the first two networks to better generalize to the PLI dataset within a limited number of training epochs. Performances of the ResNet and MobileNet classifiers are almost as good (−1.5%) as the state-of-the-art in image-based PLI, in spite of a $\times$15-time increase in language diversity and reduced training costs.

We notice from Table B that most languages perform very well, close to the overview given by the aggregate performance metrics. Most of the languages that perform poorly still perform well above 80% precision with the best performing classifiers. The languages that perform the worst tend to have common syntactic characteristics either among them or with other languages included in the dataset. This is the case when a language is a subset of another one, as it is for example for Objective-C and Objective-C++; and yet the two languages are recognizable with 76–82% precision by the MobileNet-based classifier. Other low-precision cases are related to languages that can embed other languages, such as HTML, JavaScript, JSX, Less, and XSLT. All

Table 3.6.: PLI performances with and without scrambling. From left to right: no scrambling (Orig), scrambling of alphabetic characters (A), digits (N), symbols (S), combinations of them (AN, AS, NS, ANS) and substitution of all non-blank characters with `x` (X).

	ResNet34								
	Orig	**A**	**N**	**S**	**AN**	**AS**	**NS**	**ANS**	**X**
Precision	0.92	0.87	0.92	0.47	0.87	0.34	0.48	0.35	0.01
Recall	0.92	0.85	0.92	0.33	0.85	0.20	0.33	0.19	0.17
F1	0.92	0.86	0.92	0.39	0.86	0.25	0.39	0.25	0.01

	MobileNetv2								
	Orig	**A**	**N**	**S**	**AN**	**AS**	**NS**	**ANS**	**X**
Precision	0.92	0.89	0.93	0.42	0.89	0.31	0.44	0.29	0.04
Recall	0.92	0.88	0.92	0.25	0.88	0.16	0.25	0.16	0.05
F1	0.92	0.88	0.93	0.31	0.89	0.21	0.32	0.21	0.04

	AlexNet								
	Orig	**A**	**N**	**S**	**AN**	**AS**	**NS**	**ANS**	**X**
Precision	0.83	0.75	0.83	0.57	0.75	0.44	0.58	0.44	0.09
Recall	0.83	0.72	0.83	0.49	0.72	0.31	0.50	0.31	0.05
F1	0.83	0.73	0.83	0.53	0.73	0.37	0.53	0.36	0.06

classifiers exhibit weaknesses in visually recognizing these languages.

Scrambling results

The three classifiers trained on the original trainset have been then tested on the scrambled versions too.

Performance results are presented in Table 3.6 for each architecture on the various dataset versions. The results provide some insights on what makes a code snippet visually recognizable, as we discuss below.

We can see that randomly scrambling decimal digits alone (dataset “N”) induces no degradation in PLI performances for the three classifiers. Scrambling alphabetic characters alone (dataset “A”) has a mild performance impact, degrading precision and recall by 3–11%, depending on the CNN, with AlexNet being the most affected one. Scrambling symbol characters alone (punctuation, operators, parentheses, etc.; dataset “S”) on the other hand is enough to have a dramatic effect on the performances of every considered architecture, inducing an impressive drop in both precision and recall in the 2–4× range. This degradation is likely due to the syntactic (and hence visual) importance that punctuation characters play in programming languages and their highly different usage of them across different languages.

Scrambling several character classes at once (datasets: "AN", "AS", "NS", and "ANS") appears to simply combine the effects of scrambling individual character classes. AN still performs relatively well (because symbols are *not* scrambled), all the datasets which *also* involve symbol scrambling perform badly, and scrambling all three character classes at once exhibits the worst performances.

Performances for the "X" dataset, where all non-blank characters have been replaced by `x`, are below 10% for both precision and recall in most cases, reaching as low as 1% for the precision of the ResNet-based classifier. It appears that the code "layout" alone, as captured by indentation, is nowhere near enough to make programming languages visually recognizable.

3.2.6. Threats to Validity

We rely on Linguist [69] as the source of truth for what is the "real" language of a code snippet. Whereas this is a common choice in the PLI literature due to the efficacy, efficiency, and broad language support of Linguist, it still means that our precision and recall results should be diminished by Linguist's performances, which is reported by GitHub as having 85% accuracy [64]. As we are comparing with other works in the literature that also used Linguist as the source of truth, this fact does not impact the improvement in language diversity that we introduce with this paper.

We used code snippets of fixed size (32 SLOCs), this could generate snippets that come from a code written in a given language X that seems written in some similar language, for example, snippets including only lines about syntactical constructs allowed in a similar language. These situations can generate confusion for the model from both training and evaluation points of view but they are very difficult to be detected.

We used synthetic images rendered from textual code snippets instead of real-world images. Both approaches can be found in the PLI literature and we compare well with previous works that also used rendered images. Moreover, works that use "real" images rely for the most part on video frames extracted from programming tutorials. Those tutorials are generally produced as screencasts, in which programming editors and IDEs are recorded directly from the desktop environment (rather than, say, from a physical camera pointed at the screen), resulting in very high-quality video frame images. Therefore we do not expect our performances to be significantly impacted by the switch from synthetic images to screen-casts video frames. Visual recognition of actual real-world images—e.g., pictures of billboards showing code or movie frames of screens showing code—would be a different matter, but it is a challenge we share with most works in the image-based PLI literature.

We relied on Software Heritage as the data source (instead of *ad hoc* crawling) and used a dataset corresponding to code retrieved from the most popular GitHub repositories. We further cleaned up all unrecognizable and unpopular languages according to the data pipeline discussed before. There is certainly a bias in this process, tilting in favor of "good" snippets rather than considering a wide spectrum of good and bad ones. We believe that our choice in this respect leads to comparable results to other approaches (e.g., retrieving snippets from StackOverflow or extracting video frames), as those solutions are also characterized by a significant selection bias for quality (the barrier for posting a snippet on StackOverflow or publishing a video tutorial being much higher than that of pushing code to GitHub).

In terms of dataset size, and according to a recent literature review [95], our experiments have been conducted over 12–15$\times$ more snippets than the largest empirical studies in *image*-based PLI. Rather, our dataset size is in line with the largest datasets used for *text*-based PLI. It is worth noting that we have arguably amplified the number of snippets that form our experimental dataset, in the sense that we have extracted several snippets from each source code file. At the same time, we have been careful in not extracting overlapping snippets for the training set, mitigating (if not fully neutralizing) this threat.

3.2.7. Discussion and future works

Based on the obtained results we can say that it is possible to automatically recognize up to 149 different programming languages in brief code snippet images, with high accuracy. This is a significant step forward in the state-of-the-art of image-based PLI, which was limited to recognize up to 10 languages. This result paves the way to use image-based PLI in real-world settings, where much more than a handful of programming languages need to be handled. It is worth noting that we stopped at 149 languages only to avoid dataset imbalance, not due to intrinsic limitations in the proposed approach. Using larger code bases [3, 116] as training datasets it should be possible to achieve even higher language diversity without significant reductions in identification accuracy.

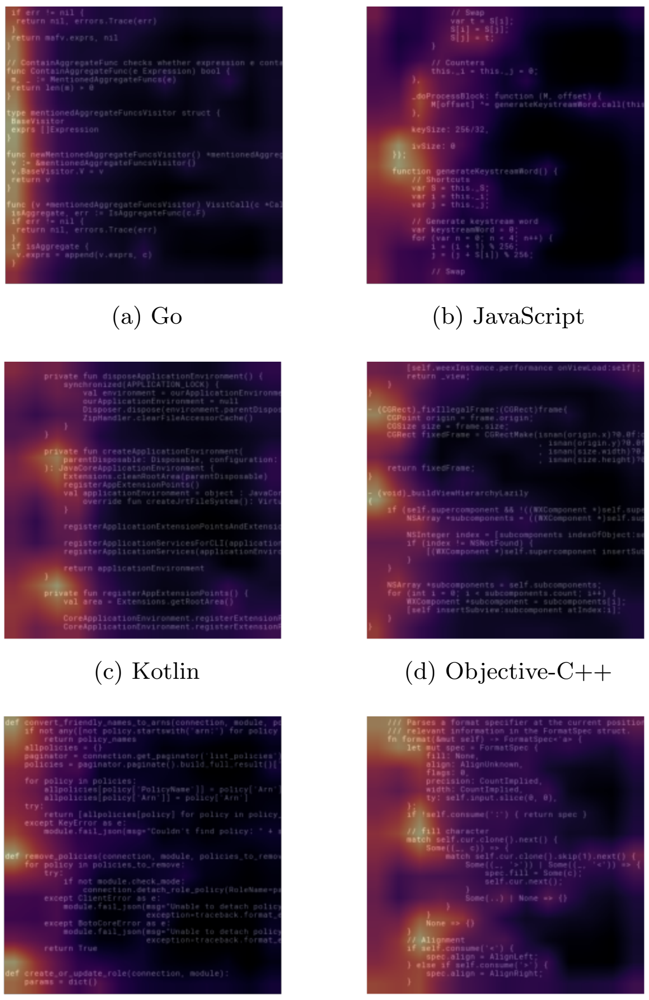
As highlighted by the results obtained from the experiments about scrambling we have gathered evidence that symbols—punctuation, arithmetic operators, parentheses, etc.—are the characters that impact the ability to visually recognize programming languages the most, making precision halve and recall diminish by $\frac{2}{3}$ when scrambled and with the best-performing classifier. Alphabetic characters have a minor impact (a few percentage points drop), whereas decimal digits have no measurable impact. The difference among classes makes intuitive sense, but the impact of symbol scrambling remains remarkable, especially considering how symbols tend to be reused for similar needs across different languages (e.g., many languages share the use of arithmetic operators or, to a lesser extent, of `";"`). A more fine-grained analysis of which *individual* symbols impact recognition the most is needed for future work.

We intuitively expected the "layout" of the code alone, as captured by indentation, to perform better than what we have observed with the "all `x`" scrambled dataset, which performed terribly with all classifiers in our experiments. In particular, we expected indentation to be a tell for at least the languages where indentation is syntactically meaningful, such as Python. That factor is probably contrasted by the fact that proper indentation is a coding best practice for all languages, and certainly so in a dataset assembled from *popular* open source projects. This could make good indentation a common trait of all languages and snippets, from which nothing can be learned by a trained classifier.

To get a *qualitative* feeling of where the classifiers gather the most relevant information for image-based PLI, we show in Fig. 3.7 the class activation map (CAM) [208] heat-maps for selected snippets. We have generated CAMs from the ResNet-based classifier, using PyTorch hooks just after the last convolutional layer. Colors in CAMs highlight which image parts contributed the most to the final classifier decision, helping with the understandability of machine learning models[7].

A few observations about these CAMs are in order. First, CAMs highlight the fact that the *beginning* of code lines is very relevant for classifier decisions. This relates to the importance of symbol characters—which are often found at the beginning of each line, like parentheses, and are also highlighted by CAMs elsewhere in code snippets—but it appears to go deeper than that. For instance, it seems that for several languages the CNN has learned to recognize full language keywords, such as `def` for the Python programming language and other keywords for Dockerfiles and Visual Basic. Second, CAMs confirm that indentation is not useful for PLI: spaces at the beginning of code lines remain

[7]These maps can be seen as heat-maps, with black/blue colors indicating low temperatures and the scale going up to yellow for high temperature in the usual way. High temperature in our case means a high contribution to the final classifier decision.



(a) Go

(b) JavaScript

(c) Kotlin

(d) Objective-C++

(e) Python

(f) Rust

Figure 3.7.: Class Activation Map (CAM) heat-maps for selected code snippets in various languages, for the ResNet-based classifier.

almost invariably in the dark. These are just some preliminary considerations based on CAMs, whose exhaustive analysis in the context of image-based PLI was out of scope for this paper but constitutes a promising lead for future work.

In terms of machine learning architectures, we have shown that transfer learning starting from pre-trained image CNNs is a viable option for image-based PLI, an approach that had received little attention in the literature for this domain thus far. With respect to starting from scratch, our approach offers the benefits of cheaper (re)training, reducing maintenance costs. Considering the very marginal reduction in precision in comparison to previous work ($\leq 1.5\%$ with respect to [95]), which is probably in large part imputable to the much higher language diversity in our experiments, the pros/cons balance seems to tilt towards pre-trained CNNs and transfer learning.

In this respect, it seems worth exploring side-tuning [206], a recent technique for transfer learning which consists in adapting a pre-trained network by training a lightweight "side" network that is then fused with the (unchanged) pre-trained network via summation. Side-tuning works very well in several scenarios [206] and it has recently been shown [213] to be applicable to multi-modal document classification, where diverse data sources such as text and images are considered, improving document classification accuracy with respect to the state of the art. Such a multi-modal approach could be naturally applied to PLI, by interpreting code snippets as both images and text. The empirical validation of the applicability of side-tuning to PLI is left as future work.

3.3. Conclusions about the Programming Language Identification task

Programming Language Identification is a fundamental task during software development and its maintenance. When the software project under analysis has a reasonable size PLI can be easily manually performed by developers. However, there exist many situations in which the scale of the software content makes the manual approach infeasible. Examples of this situation can be found when dealing with hosting repository systems (e.g. GitHub), developers' forums (e.g. StackOverflow), or large-scale software archives (e.g. Software Heritage). These codebases are becoming more and more popular among the developers' community and the size of stored software is growing. When dealing with these cases the manual strategy for PLI is no longer a reasonable choice and task automation is required.

The work presented in this chapter described different approaches to deal with the automatic PLI task developed by exploiting learning algorithms. The huge amount of freely available code data is not accompanied by the same support for labels, i.e., the ground truth about the programming language. This constitutes an important impediment to the implementation of supervised learning models whose functioning and effectiveness strictly depend on labels' availability and quality. However, some foresight and assumption can serve to partially solve the labels' problem.

The text-based approach is used as labels the files extensions found at the end of the filenames as they are usually related to the language used to create the source code file. This strategy does not rely on external tools or other kinds of knowledge so does not introduce any dependency to the model but it has the drawbacks of having intrinsically ambiguous labels and no clear unique relation between extensions and languages. The model makes predictions at the file level, it showed good performance despite its simple architecture which makes it easy to maintain in the long term. However, some weaknesses can be due to the chosen features that have been manually defined and depend on the vocabularies definitions which can represent a limit in the information extraction. Also, this model has to deal with Out-Of-Vocabulary (OOV) tokens and bigrams occurrences which sometimes can represent a non-negligible problem.

The image-based approach used an external tool to label the code instances, so they are labeled directly with the language but considering that the labels' quality directly depends on the performance of the employed tool. The model makes predictions for snippets of 32 lines of code which are transformed into an image representation and processed by a usual image recognition neural architecture. This model reaches very good results even if it deals with few lines of code only, however it has a more complex architecture that the text-based model which in some cases could represent an impediment. Features are not manually defined but are automatically extracted by the model and relevant information seems to be carried by punctuation symbols as shown in the analysis of the scrambled files. Moreover, the model does not suffer from the OOV problem.

Both models reached reasonably good results proving their effectiveness, also they can deal with a relatively high number of classes suggesting their scalability for this aspect. Depending on the final application features, i.e., the environment in which the model should actually be employed, one model can be preferable with respect to the other one.

Chapter 4.

Software Defect Prediction

Code analysis, testing, and review are very important in assuring the quality and reliability of the final software product and they should be performed accurately even if they are usually highly time expensive tasks in software artifacts production.

As we already pointed out in Chapter 2, these tasks have been traditionally performed by static and dynamic code analyzers or directly by humans. Recently, many works have been developed to treat bugs and vulnerabilities detection as machine learning problems. In particular, Deep Learning models result to be the best-performing ones and bring relevant improvements to the field of defect prediction as they can exploit syntactic, context, and semantic information at the same time especially because of the possibility to perform automatic feature learning. In our work, we want to follow this direction as it seems to bring promising results.

First of all, while analyzing the related literature we noticed that almost every work developed within this field treats the defect prediction problem as a binary classification task in which the two classes simply represent the presence or absence of defects within the analyzed code fragment. All the kinds of defects are treated in the same way and are grouped together in a unique class to which defective codes belong. Conversely, in our work, we want to be able to retrieve some additive information about defects by also assigning an identity to them. To do so the problem is treated here as a multi-class problem in which the classes are represented by the kind of bugs and the class of non-defective code.

The dataset used in this work has been presented in [65][1] consisting of GitHub C/C++ projects and the outcomes of a static analyzer applied to the code. From the analyzer output, it is possible to extract the exact locations and names of the detected bugs. To begin our analysis, we decided to focus on the three most common bugs found within the dataset which are the `Dead Store` (DS), `Null Pointer Dereference` (ND), and `Uninitialized Value`

[1] `https://zenodo.org/record/3472048`

(UV) bugs. The following brief descriptions of these bugs are provided together with the associated reference in the Common Weakness Enumeration (CWE) list which is an important categorization reference for vulnerabilities and weaknesses [39].

`DEAD_STORE` (CWE-563[2]) refers to the *dead store* bug which is an issue that happens when a value assigned to a variable is never used in the rest of the code. This issue is not strictly an error but it causes a waste of resources in time and memory terms.

`NULL_DEREFERENCE` (CWE-476[3]) refers to the *null pointer dereference* which occurs when the program causes dereference of a pointer which is expected to have a valid value but it turns to be `NULL`. This kind of issue can actually cause crashes and undefined behavior of the program.

`UNINITIALIZED_VALUE` (CWE-457[4]) is caused when a program tries to read or access a value before it has been initialized. This can cause crashes and unwanted program behavior.

As a static analyzer is used to label the data, we can at most imitate the static analyzer performances but, to our knowledge, this is the only freely available dataset that contains information about bugs identities and, due to the temporary lack of another method of labeling (e.g., labeled by humans), the static analyzer output is considered reasonably acceptable. Such work can serve to study how close to the static analyzer outcomes we can go by performing bugs detection by implementing statistical methods which learn from data. Moreover, the designed models can be used as heuristic methods to spot bugs instead of performing a complete static analysis when the source code size would require a big amount of time to perform the complete static analysis.

Based on the results obtained performing this study we can deduce which are the most promising models, among the attempted ones, that can be exploited in defect prediction. In the future, we aim to apply the models selected in this work to an improved dataset in order to surpass the constraint given by the labeling strategy, i.e., using the static analyzer output. We aim to build this dataset using multiple labeling techniques and select the data for which the labeling techniques share their results. The labeling tools should be several static *and* dynamic analyzers used on the same input data whose final results are compared. A label is considered reliable only when all the employed tools agree on the outcome.

[2]`https://cwe.mitre.org/data/definitions/563.html`
[3]`https://cwe.mitre.org/data/definitions/476.html`
[4]`https://cwe.mitre.org/data/definitions/457.html`

4.1. Dataset

As pointed out in [65], the static analyzer used to build the dataset is *Infer* [55], an open source tool developed by Facebook[5]. This tool is able to process other programming languages too, i.e., Java and Objective-C, for now, we only focus on the content of the original dataset but the work could be easily extended to these other languages without too much effort.

Infer can handle 92 different potential bugs and 15 of them do not concern C and C++ programs. A complete list of the issues detectable by means of the *Infer* static analyzer together with their availability for C and C++ programs is shown in Table C.1 in Appendix C.

Despite the rich variety of bugs types we only concentrate on three of them as they are the only ones that are relevantly represented within the dataset. In fact, the null pointer dereference has 9482 examples the dead store has 10722 examples, while uninitialized value is present 8050 times, other kinds of bugs are too underrepresented with respect to these others and for the moment we focus on these three only.

Initially, the dataset contains 3169 projects cloned from GitHub and its size is around 33 GB. The dataset is presented in a raw form, in which the whole project is coupled with the whole static analyzer output and log files, for this reason, it requires a lot of effort to parse it in a suitable form, i.e., the code fragment coupled with the related bug(s) found in it. The data directory contains one directory for each project downloaded from GitHub. These project directories are named with the *GraphQL-ID* from GitHub's *GraphQL API*. In each of these *GraphQL-ID* labeled directories, there is a `license.txt`, a `url.txt`, a `source` directory, and a `derivatives` directory. The `license.txt` contains the license for the original project, the `url.txt` contains a link to the original project on GitHub, the `source` directory contains the original code, and the `derivatives` directory contains the output of *Infer*. The `derivatives/infer-out` directory contains the files `.infer_runstate.json`, `bugs.txt`, `costs-report.json`, `logs`, `proc_stats.json`, `report.json`, `results.db` and the subdirectories `events` and `specs`.

The files `bugs.txt` and `report.json` contain the most relevant results about bug detection and we exploit them to extract the bug's information.

For each project, the associated `bugs.txt` file is analyzed and the reported bugs are extracted and associated with the file and code line in which they are detected by the static analyzer.

As a first step, we decided to focus on the analysis at file-level granularity.

[5]https://github.com/facebook/infer

To do so we started creating a file `files-list.txt` which lists every `.c` and `.cpp` file found within the projects stored together with the precise file location and name in order to easily retrieve the files from the dataset.

For each project, we parsed the `bugs.txt` file extracting from it the name and position (code line within the file) of each detected bug together with the name and location of the file in which that bug is detected. This permits to have for each of the files in `files-list.txt` a list of all bugs (if present) and bugs positions that have been detected in it by the static analyzer.

From this analysis emerges the bugs *popularity* among projects which is represented in Table 4.1 (note that each file could contain the same bug multiple times at different positions, the situation depicted in Table 4.1 counts these multiple occurrences). We choose to restrict on the ND, DS, and UV bugs, however also `MEMORY_LEAK` and `RESOURCE_LEAK` have thousands of examples but we decided to not focus on them because after the preprocessing and vectorization phases their number dropped dramatically (because of several issues, for example in code parsing, etc.) bringing to an even more pronounced imbalance in data.

Issue	**Occurrences**
NULL_DEREFERENCE	9482
DEAD_STORE	10722
UNINITIALIZED_VALUE	8050
RESOURCE_LEAK	1329
MEMORY_LEAK	4543
USE_AFTER_FREE	100
POINTER_TO_INTEGRAL_IMPLICIT_CAST	15
LOCK_CONSISTENCY_VIOLATION	32
USE_AFTER_LIFETIME	90
STATIC_INITIALIZATION_ORDER_FIASCO	644
DEALLOCATE_STATIC_MEMORY	8
DEALLOCATION_MISMATCH	16
DEALLOCATE_STACK_VARIABLE	7
PREMATURE_NIL_TERMINATION_ARGUMENT	1

Table 4.1.: Issues' popularity within the dataset

File-level dataset Focusing on the three commonest bugs that we previously introduced, i.e., `Dead Store` (DS), `Null Pointer Dereference` (ND) and `Uninitialized Value` (UV), the `file-list.txt` can be enriched with bugs information. In particular, for each file of the list, we can assign a list of 3 binary values which represent the presence (1) or absence (0) of the selected bugs. If none of the 3 values is set to 1 then the file is considered clean.

Due to the possibility of having multiple bugs within the same file, this problem would be originally a multi-class multi-label classification task but the analyzed dataset only contains a few examples in which different bugs occur within the same file, i.e., multi-labeled files.

To treat multiple labels we tried to represent them using the Label Powerset strategy [177], in which each combination of bugs (co-)occurrences is considered as a class, for this reason, we end up with $2^n = 8$ classes, where $n = 3$ is the number of considered bugs.

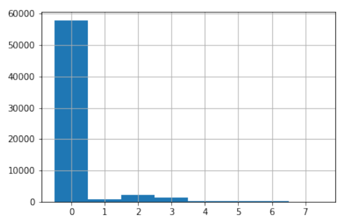
After the preprocessing and data preparation step which is described in the next Section the files that successfully complete the preparation phase and that can be used in our model are 63439 and the dataset situation after the cleaning and preparation steps is shown in Table 4.2 and can also be visualized using the Label Powerset strategy to have an immediate representation of how many and which classes co-occurrences can be found within the dataset under analysis.

DS	ND	UV	lab. pow.set	n of examples
0	0	0	0	57885
0	0	1	1	743
0	1	0	2	2222
1	0	0	3	1292
0	1	1	4	153
1	0	1	5	200
1	1	0	6	296
1	1	1	7	40

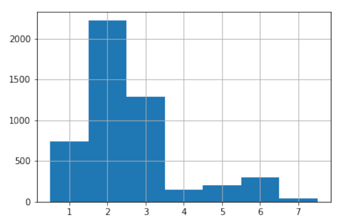
Table 4.2.: Files distribution among classes.

In the present case, the situation is represented in Figure 4.1 in which the second histogram 4.1b represents the same situation as the first one 4.1a but it does not report the clean code cases, this is done to better visualize the defective classes distributions as compared to the clean code class it can be difficult to visualize them.

In the performed experiments we considered only the classes 0, 1, 2, and 3 where there is only one bug or no bugs while the other cases are discarded reducing the total number of files to 62139. This is assumed as the bugs' co-occurrences rarely happen, however, we aim to include them in future work



(a) Data distribution after preprocessing.



(b) Defective classes distribution after preprocessing.

Figure 4.1.: Data distribution.

and analysis of this problem.

At this point, the dataset can be split into train and validation sets and this is done by selecting 80% of the available instances for the first one and the remaining examples for the second one. The splitting procedure has been performed in a stratified way in order to reproduce the same class distribution in the two obtained sets of data.

As Table 4.2 shows, the dataset is highly unbalanced and this is true for the just created train set too. In particular, the class correspondent to clean files resulted to have more examples than the other ones. This always happens in works like this as we pointed out in Section 2 and some strategy to face this problem should be implemented. In this work, random oversampling technique is used, in this way the balanced train set is composed of 185210 examples. On the other hand, the validation set is kept unbalanced to fairly reproduce the class distribution of the original dataset with 11578 examples for class 0, 149 for class 1, 444 for class 2, and 258 for class 3.

Function-level dataset As we will see in the next Sections, one of the representations on which we based one of our models did not work well at file-level. This problem is due to the nature of the model used to extract code vectors which working recursively on AST structures has memory management limitations when it comes to large-scale ASTs, i.e., ASTs with a high number of nodes. At file-level it is more common to find such highly sized structures and to avoid the memory management problem we decided to work at a finer granularity level for this representation, i.e., at function-level.

Starting from the original dataset we implemented a function extractor using the python package version of *Clang* which is a C/C++ parser[111, 112]. In particular, each project found within the dataset is processed by the function extractor which searches for all the function declarations and creates a file containing information about them. In fact, for each function declaration, we store the function's name, the path and name of the file in which the declaration has been found, and the numbers of the starting and ending lines of the declaration in order to easily obtain the actual function location. Separately, the whole function definitions are stored in `txt` files in order to easily retrieve their contents in successive phases.

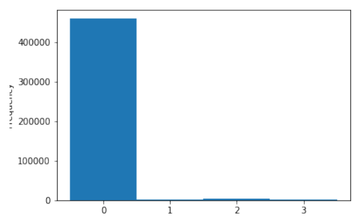
Previously we stored bugs information, i.e., bug's type, file location, and bug's line position within the file. Retrieving this information we can compare the bugs' positions with the locations of the extracted functions. More precisely, if a bug has been detected at a certain line of a certain file and this line is located between the starting and ending lines of a function declaration, this function is labeled as affected by that defect. Again, multiple labeling is possible but this time no multiple-labels cases have been detected, so the single class labeling choice does not represent an approximation here.

The function declaration contents are then passed to the representation model which, when successfully applied, generates a vector for each function. The dataset after the application of the representation model is composed of 471033 instances representing the functions whose distribution among classes is shown in Table 4.3 and Figure 4.2 in which, as in the file-level case, the second histogram 4.2b represents the same situation of the first one 4.2a but it does not report the clean code cases for better visualization.

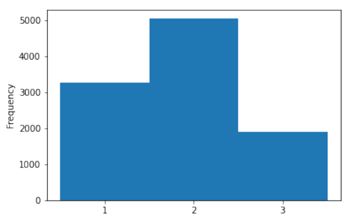
DS	ND	UV	lab. pow.set	n of examples
0	0	0	0	460813
0	0	1	1	3273
0	1	0	2	5040
1	0	0	3	1907

Table 4.3.: Functions distribution among classes.

As expected also in this case we end up with a highly unbalanced dataset. After splitting the total dataset with an 80/20 ratio into train and validation sets in a stratified way, we apply random oversampling to the portion dedicated to the training procedure. In this way, the train set results to have 1474600 examples, while the validation set contains 92163 clean functions, 655 functions belonging to class 1, 1008 functions with label 2, and 381 for class 3.



(a) Functions distribution after preprocessing.



(b) Defective classes distribution after preprocessing.

Figure 4.2.: Functions distribution.

4.2. Representations and Models

Source code can be represented with several methodologies each of which emphasizes different code aspects. Depending on the specific situation some code features can be more relevant than others and the representation which highlights the important characteristics the most should be preferred among the other possibilities.

An Abstract Syntax Tree (AST) is a tree representation of code that is obtained from the source code, as a written sequence of characters, after it has been processed by a certain language *parser* which builds the tree data structure according to the language grammar rules while checking the code's syntax correctness.

First of all, the code stream is subject to *lexical analysis* which transforms it into a sequence of tokens, usually tokenizing according to language-specific rules. The tokenized code is then passed to the actual *parser* which establishes if the sequence is composed by *legal* expressions and organizes them in a tree structure. AST's internal nodes represent language operations while leaf nodes are identifiers, values, constants, etc. (the operands of the operations).

Many aspects which emerge at compile-time are handled by the *parser* and encoded into the AST structure, in fact, it usually serves as input for code analyzers, especially static analyzers.

As in this work, the aim is to learn and replicate a static analyzer behavior we focused on the AST code representation, in this way the starting point of our prediction models would be the same as the one of the static analyzer that we aim to learn.

Learning algorithms take as inputs data encoded into numerical vectors and we need a strategy to represent AST structures. Two different code vectorizer models based on ASTs are applied in the following and adapted in slightly different situations for defect prediction and identification. The next Sections focus on describing the vectorizer models and the steps needed to practically apply them to our situation.

4.2.1. Code2Vec

Among various strategies to represent codes and extract information from their AST structures *code2vec* results to be the state-of-the-art in several tasks. Here we try to adapt the vectorization strategy proposed in [10] to our purpose to investigate its capability to detect and identify the selected kind of bugs in C and C++ code fragments modeling the task as a multi-class problem.

Code2vec is an AST-path based representation which means that the code representation is built on a collection of paths that are extracted from the AST.

An AST path p is a path between nodes in the tree structure which starts from one terminal token t_s (a leaf-node) and ends in another terminal token t_e. The path is composed by internal nodes which are common ancestors of the start and end tokens together with the directions in which the nodes are traversed as the nodes can be reached following the two directions *up* and down on the tree structure. AST paths are characterized by their length which is the number of nodes that connect the two considered leaves.

Another important feature of paths is the path *width* which is the maximum difference between the child indexes for child nodes of the same intermediate node.

Finally, given the path p which connects the starting and ending tokens t_s and t_e, a *path-context* is defined as the triple (t_s, p, t_e), where p is encoded as a sequence of internal nodes (with directions).

Given a piece of code, many *path-contexts* can be extracted. Due to efficiency considerations a maximum number of considered *path-context* needs to be set as a model hyperparameter, together with maximum length and width for the paths.

The extracted *path-context* are then fed into the actual model, using embeddings for both tokens and paths combining them employing a Fully Connected Layer to build the context vectors. An attention architecture is used on the computed context vectors to generate the final code vector representation which is used by the final classifier to make predictions about a certain task.

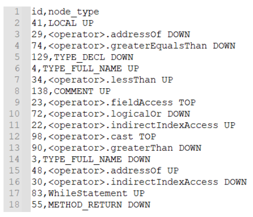
As *code2vec* has been initially designed to treat Java code, some adaptation steps are required to use this model to process C and C++ files. Particularly, our files which contain C/C++ code need to be parsed to switch from a source code textual representation to the AST representation according to the C/C++ grammar rules. Once these AST representations are created the same strategy used in [10] to extract paths and to build the code vector representations with them can be applied.

A tool which can be exploited to do this kind of work is already available on GitHub and it is known as *Pathminer* or *Astminer*[6] and it is described in [100]. This software has been developed by the *JetBrains* research department and it aims to extend the applicability of the *code2vec* strategy to other languages than Java by using different parsers to generate ASTs. *Astminer* includes C and C++ support by means of the integrated *Fuzzy* parser[7] and we can use this tool jointly to the *code2vec* model building a 2-parts tool-chain. In particular, the *Astminer* tool provides the list of path contexts extracted for each code fragment whose granularity can be set at folder, file or function levels. For each input passed to *Astminer* the produced output is a folder structured as follows.

```
/
├── c
│   ├── data
│   │   └── path_contexts.c2s
│   ├── node_types.csv
│   ├── paths.csv
│   └── tokens.csv
└── cpp
    ├── data
    │   └── path_contexts.c2s
    ├── node_types.csv
    ├── paths.csv
    └── tokens.csv
```

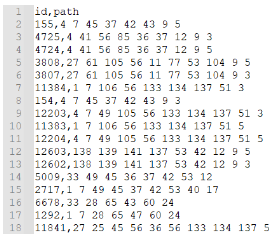
`node_types.csv`, `paths.csv` and `tokens.csv` contains a mapping between integer identifiers (IDs) and nodes (with directions up/down), paths and tokens found within the provided input, respectively. In `tokens.csv` retrieved tokens are eventually treated by splitting the original camelCase notations into subtokens separated by the pipe symbol (vertical line). In the `path_contexts.c2s` file each line starts with the name of the folders, files or functions (accordingly to the chosen level of code granularity) followed by a sequence of space-separated path contexts. Each of these contexts are represented as triples

[6] https://github.com/JetBrains-Research/astminer
[7] https://github.com/ShiftLeftSecurity/fuzzyc2cpg
https://github.com/ShiftLeftSecurity/codepropertygraph/



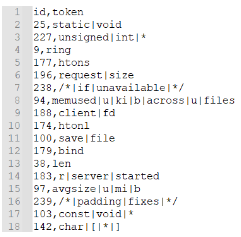
```
id,node_type
41,LOCAL UP
29,<operator>.addressOf DOWN
74,<operator>.greaterEqualsThan DOWN
129,TYPE_DECL DOWN
4,TYPE_FULL_NAME UP
34,<operator>.lessThan UP
138,COMMENT UP
23,<operator>.fieldAccess TOP
72,<operator>.logicalOr DOWN
22,<operator>.indirectIndexAccess UP
98,<operator>.cast TOP
90,<operator>.greaterThan DOWN
3,TYPE_FULL_NAME DOWN
48,<operator>.addressOf UP
30,<operator>.indirectIndexAccess DOWN
83,WhileStatement UP
55,METHOD_RETURN DOWN
```

(a) node_types.csv



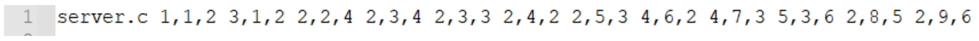
```
id,path
155,4 7 45 37 42 43 9 5
4725,4 41 56 85 36 37 12 9 3
4724,4 41 56 85 36 37 12 9 5
3808,27 61 105 56 11 77 53 104 9 5
3807,27 61 105 56 11 77 53 104 9 3
11384,1 7 106 56 133 134 137 51 3
154,4 7 45 37 42 43 9 3
12203,4 7 49 105 56 133 134 137 51 3
11383,1 7 106 56 133 134 137 51 5
12204,4 7 49 105 56 133 134 137 51 5
12603,138 139 141 137 53 42 12 9 5
12602,138 139 141 137 53 42 12 9 3
5009,33 49 45 36 37 42 53 12
2717,1 7 49 45 37 42 53 40 17
6678,33 28 65 43 60 24
1292,1 7 28 65 47 60 24
11841,27 25 45 56 36 56 133 134 137 5
```

(b) paths.csv



```
id,token
25,static|void
227,unsigned|int|*
9,ring
177,htons
196,request|size
238,/*|if|unavailable|*/
94,memused|u|ki|b|across|u|files
188,client|fd
174,htonl
100,save|file
179,bind
38,len
183,r|server|started
97,avgsize|u|mi|b
239,/*|padding|fixes|*/
103,const|void|*
142,char|[|*|]
```

(c) tokens.csv



```
server.c 1,1,2 3,1,2 2,2,4 2,3,4 2,3,3 2,4,2 2,5,3 4,6,2 4,7,3 5,3,6 2,8,5 2,9,6
```

(d) path_contexts.c2s

Figure 4.3.: Portions of *Astminer* output files generated for the `server.c` source file.

of IDs separated by commas which represent the start token, path, and end token respectively, i.e., ID_{t_s}, ID_p, ID_{t_e}, according to the mapping defined in the other three files. An example of the *Astminer* output is given in Figure 4.3 where we show a portion of each generated file for the case of a C source code file named `server.c` which has been found in one of the analysed projects.

As *code2vec* has been originally built to generate an automatic naming model, once the granularity is selected the names of folders, files or functions are meant to be used as labels in the output of *Astminer*, so we need to modify this feature to label our examples with the error classes labels.

The input of *Astminer* can be the whole project repository but when it is used in this way only the file name is used to identify the file and not the complete path to it, this brings to ambiguity and non-unique references to files as there could be different files with the same name but characterized by different paths within a repository. These files can have completely different contents and error reports but would be named in the same way in the *Astminer* output and would be impossible to distinguish the two files.

These characteristics could cause unnecessary data loss, so the *Astminer* tool is used separately on each file of our file-level dataset described in Section 4.1.

At the beginning, the dataset contained 86358 files but this number decreases after the preprocessing and representation extraction steps. The *Astminer* is not always able to process the files or, for some files, the representation is not computed as filters on AST size in terms of number of nodes and tokens are used. This filtering strategy has been required as some files could generate huge AST structures which could be hard to handle and could generate an unmanageable number of paths. In our case we set to 5000 the maximum number of nodes in the tree and to 300 the maximum number of treated tokens within a file, while the maximum length of the considered paths is set to 10 while the maximum path width is set to 2.

Another modification that has to be done before feeding the *code2vec* model with the extracted path contexts is due to the usage of the *Astminer* tool separately on single files. This kind of usage causes the encoding of nodes, tokens and paths to not be unique, i.e., the same node ID can refer to different nodes when we refer to different files. This is an unwanted behavior as data used in this way would not be useful to build a meaningful representation of code and we need an uniform encoding for our data. To do so, all tokens and nodes found within the dataset are collected. Nodes are only available in a limited number, i.e., there are only 191 different kind of nodes, and an integer identifier is assigned to these nodes and substituted to the one assigned by the *Astminer* tool. Conversely, tokens are defined by programmers and are potentially infinite, so we need to define a vocabulary of tokens that we want to consider as known, i.e., based on the number of occurrences within the dataset, while any other encountered token is represented as an unknown token UNK_T. Only tokens which have been counted to be in more than 200 files are kept to build the tokens vocabulary and they result to be 2851. These selected tokens are used to generate the unique token IDs that are substituted to the ones assigned by *Astminer*.

Similarly, identifiers are assigned to paths and, after assigning the right ID to the nodes found within the paths, a vocabulary for paths is defined based on the paths' occurrences within the dataset. Paths found in more than 400 files are selected to build the vocabulary and they result to be 47842. Any other path is encoded as a unique unknown path UNK_P. After substituting all these IDs to the extracted path contexts for our files, they are ready to be used as inputs for the *code2vec* model.

The *code2vec* has been adapted to our use case and, instead of predicting tokens which would constitute the predicted name for the code fragment, it needs to be able to predict which of the four bugs classes is the appropriate one for the given input. To this end, once the code vectors are computed by the Attention architecture of the model, these are passed to a simple neural classifier which is suitable for this particular classification problem whose output layer is composed by four classes. Our model uses a *categorical cross entropy* loss 3.2 and exploits the *Adam* optimizer [93] with learning rate 0.0001 for training.

The whole model is trained in an end-to-end fashion and the network weights are randomly initialized, here we do not use the pre-trained parameters as the input is numerically encoded using *Astminer* and our token and path ID unification which do not correspond to the original *code2vec* path-context encoding.

Different configurations for the model are evaluated by modifying some of the hyperparameters, they are described and evaluated in the following Section.

Results

In this Section we discuss the results obtained from the implementation of the described model based on *code2vec* representation.

As we already pointed out this is a 4-classes classification problem and the classes are: Clean Code (CC, 0), Uninitialized Value (UV, 1), Null Dereference (ND, 2) and Dead Store (DS, 3).

The input of the *code2vec* model is a set of a fixed number of accepted path-contexts which in the following has been set at the value of 200. The maximum number of training epochs is set to 20 and the batch size to 64. After each training epoch the model performance is evaluated on the unbalanced validation set.

In the *code2vec* neural architecture we set at different values the embeddings sizes, one is for the tokens embedding and the other one is for the paths embedding. The embeddings of the start and the end tokens is the same, as the embedding layer is shared between the two tokens.

In this experiment we set the same size (column Emb. in Table 4.4 and Table 4.5) for both tokens and paths embedding layers and the tested sizes are 70 and 150. Also, we evaluated different configurations for the final classifier which has been set as a simple neural network with different layer sizes and depth. The tested values for the layer size are 10 and 30 while for the number of layers we tested for 1 and 3.

Table 4.4 reports the values for categorical accuracy (Acc, Eq. 2.1) and weighted average values for AUC-ROC, precision (P, Eq.2.2), recall (R, Eq. 2.3) and F-measure (F, Eq.2.4) on the validation set at the *best* epoch for each configuration. In this evaluation we suppose that it is preferred to have clean code examples classified as defective ones, i.e., False Negatives for class 0 (FN_0), than having defects which are wrongly classified, i.e., False Negatives for class i (FN_i) for $i = 1, 2, 3$. For this reason the *best* epoch is chosen with respect to the macro average value (Eq.2.7) of recall as higher values for R_i mean lower number for FN_i with respect to TP_i (True Positives for class i). For defective classes ($i = 1, 2, 3$) optimizing recall values means optimizing with respect to false negatives, i.e., aiming to keep low the number of unrecognized defects in these classes, which can be examples of i-th defect classified either as non-defective or as examples of the j-th defect ($i \neq j$). Other kinds of averages, i.e., micro (Eq.2.6) and weighted (Eq.4.1, where n_i is the number of examples belonging to the i-th class in validation set and $N = \sum_{i=0}^{3} n_i$ is the total number of examples of the validation set), would give to the Clean Code (CC) class a higher weight (due to the higher number of examples belonging to this class in the validation set) causing epoch choice mainly focused on keeping low false negatives of this class, i.e., clean code predicted as defective, which should not be the main driver of our choice.

$$R_w = \frac{\sum_{i=0}^{3} n_i R_i}{N} \tag{4.1}$$

After the best epoch selection process, weighted average values are shown in the results as they are more representative of the unbalance which characterize the validation set. Table 4.4 refers to experiments in which the code vectors dimension is set to 30.

Emb.	**Depth**	**Layer S.**	**Acc**	**AUC**	**R**	**P**	**F**
70	1	10	0.786	**0.921**	0.786	0.939	0.844
70	3	10	0.744	0.908	0.744	0.939	0.815
70	1	30	0.779	0.908	0.779	0.941	0.84
70	3	30	0.742	0.891	0.742	0.941	0.816
150	1	10	**0.801**	0.919	**0.8**	0.937	**0.852**
150	3	10	0.719	0.899	0.719	0.939	0.798
150	1	30	0.783	0.916	0.783	0.941	0.842
150	3	30	0.744	0.89	0.743	0.942	0.817

Table 4.4.: Results for code2vec based models with 30-dimensional code vectors.

The same configurations have been tested with code vectors of an higher size, i.e., setting code vector dimension to 50. The results are reported in Table 4.5.

Increasing the depth and the layer size of the final classifier slightly worsen the model performance in all cases except for the precision values. It is not clear how increasing the embedding size and code vector dimension affects the results as it sometimes makes the metric values worse while in some other cases it improves them.

However, the best results seem to be concentrated in the model which uses 30-dimensional code vectors and the preferable configurations could be identified in the models with depth 1 and layer size 10 with both 70 ([M1])

Emb.	**Depth**	**Layer S.**	**Acc**	**AUC**	**R**	**P**	**F**
70	1	10	0.787	0.915	0.787	0.938	0.844
70	3	10	0.738	0.907	0.737	0.94	0.811
70	1	30	0.772	0.912	0.772	0.941	0.835
70	3	30	0.736	0.889	0.736	**0.943**	0.813
150	1	10	0.79	**0.921**	0.79	0.939	0.846
150	3	10	0.723	0.897	0.723	0.939	0.801
150	1	30	0.784	0.919	0.784	**0.943**	0.844
150	3	30	0.755	0.888	0.755	0.942	0.824

Table 4.5.: Results for code2vec based models with 50-dimensional code vectors.

and 150 ([M2]) embedding sizes. These two models have similar results for almost all the shown metric values. In Table 4.6 are shown the precision, recall and F-measure for each of the considered classes for these two models.

Class	**Emb.**	**P**	**R**	**F**
CC	70	0.992	0.795	0.882
UV	70	0.135	0.698	0.227
ND	70	0.306	0.712	0.428
DS	70	0.112	0.589	0.188
CC	150	0.990	0.811	0.891
UV	150	0.138	0.651	0.228
ND	150	0.295	0.707	0.416
DS	150	0.129	0.589	0.211

Table 4.6.: Results for each class of the two best models.

Looking at these results we can see that Precision values are very low for each of defective classes ($i = 1, 2, 3$), while better results can be found for the recall values. Given the definition of the i-th class Precision $P_i = \frac{TP_i}{TP_i+FP_i}$ we can see that it is an indicator of the False Positives for the i-th class. In our cases P_0 has a high value which means the FP_0 is low compared to the number of instances predicted as belonging to the 0 class, i.e., relatively few defective instances are classified as non-defective. On the other hand, for $i \neq 0$ P_i has very low values which means that for these classes FP_i has a high value (considering the total number of predictions for the i-th class). For these defective classes we can distinguish between two main cases between False Positives as they can be due to either Clean Code or code with defect j ($j \neq i$ and $J \neq 0$) predicted as having the i-th defect. Having low values of P_i

$(i = 1, 2, 3)$ means that *often* our models could predict errors where they are not present or predict the wrong kind of error when it is present.

Similar considerations can be done analysing the Recall as its definition is $R_i = \frac{TP_i}{TP_i+FN_i}$ so it represents an indicator of False Negatives for the i-th class. For class 0 FN_0 represent clean code examples which are predicted as having some kind of defect, while for $i = 1, 2, 3$ FN_i could be due to code with the i-th defect classified as either clean code or code with the j-th defect ($j \neq i$ and $j \neq 0$). In our case there are lower gaps between the recall values of the four classes than the ones found for precision.

To better visualize the kind of mistakes done by the models and improve the results interpretation we show the confusion matrices obtained at the selected epochs for the two best models [M1] in Table 4.7 and [M2] in Table 4.8. Summing along the i-th column all the elements except for the i-th one (which represents TP_i) we obtain the value of FP_i, while summing along the i-th row all the elements except for the i-th one (still representing TP_i) we obtain the value of FN_i. The i-th element of the last row contains the total number of examples predicted as belonging to the i-th class, i.e., $TP_i + FP_i$, which represents the denominator of P_i. Comparing these values with the respective value of TP_i we can understand why P_i values are so low for $i = 1, 2, 3$, most of these FP_i are due to Clean Code classified as having the i-th defect, so the low P_i values are also due to the high degree of unbalance between the class 0 and the others, i.e., 78:1, 26:1, 45:1 for class 1, 2 and 3, respectively. Conversely, the i-th element of last column contains the total number of examples which belong to the i-th class, i.e., $TP_i + FN_i$, which represents the denominator of R_i. Again, comparing these values with the respective TP_i we can see that the gaps between them are smaller than in the precision cases and this justify the better results obtained for the recall metric.

The defective class which is better recognized by these models is class 2 which represents *Null Pointer Dereference* errors. Class 3, i.e., *Dead Store* bug, results to be the most problematic error in terms of models performance for the recall value too. This different behavior in error catching could be due to the errors nature and their capability of being captured within path contexts. Also, it has to be noticed that the model input is constrained to only 200 randomly selected path contexts and it can happen that relevant paths have been discarded at the beginning.

We can compare these results with [190] which also treat individual kinds of bugs. Particularly, *Null Pointer Exception* is treated there as well and we can see that for this class they have P=0.351 and R=0.507 while we obtained P=0.306, R=0.712 in [M1] and P=0.295, R=0.707 in [M2] so, while P is slightly worse in our cases R results to have consistent improvements in both [M1] and

		Predicted				
		CC	UV	ND	DS	Tot.True
True	CC	**9200**	579	664	1135	11578
	UV	5	**104**	16	24	149
	ND	29	54	**316**	45	444
	DS	40	31	35	**152**	258
	Tot.Pred.	9274	768	1031	1356	

Table 4.7.: Confusion matrix at best epoch for model [M1]

[M2]. However, fair comparison is not possible as the dataset used in [190] is different from ours and also their models work at different granularity levels.

If we group together the defects in a unique class (without retrain the model as a binary classifier) the confusion matrix for [M1] becomes as depicted in Table 4.9 obtaining P=0.246, R=0.91, F=0.388 and Acc=0.803, while for [M2] in Table 4.9 we have P=0.256, R=0.888, F=0.397 and Acc=0.816.

Other works we presented in Section 2.3 have higher F values, however to better compare this approach against these models we should retrain our models as binary classifiers.

4.2.2. Infercode

Infercode in another useful model for code vectorization which has been recently developed and described in [25].

The model's purpose is to generate representative code vectors using unsupervised learning, i.e. without using labels, whose availability is usually limited. This makes code vectorization an independent step and it can be performed separately from the actual classification concerning the final task, i.e. Software Defect Prediction. In this way, the overall model is not trained as an end2end architecture but the two steps, i.e. vectorization and classification, are performed separately and chained together.

The *Infercode* model generates AST structures from the input codes and, for each of them, several subtrees are extracted and, employing a selection

		Predicted				
		CC	**UV**	**ND**	**DS**	**Tot.True**
True	**CC**	**9385**	533	702	958	11578
	UV	9	**97**	13	30	149
	ND	44	45	**314**	41	444
	DS	42	27	37	**152**	258
	Tot.Pred.	9480	702	1066	1181	

Table 4.8.: Confusion matrix at best epoch for model [M2]

TN	**FP**
9200	2378
74	777
FN	**TP**

Table 4.9.: Confusion matrix for binary [M1]

technique some subtrees are chosen to form a *vocabulary* of trees that are used in the unsupervised training phase.

The main neural architecture is a Tree-Based CNN which is a neural network based on the convolution approach adapted to process tree-like data structures. The model is trained by optimizing it with respect to the task of predicting for a given AST the probabilities of having each of the vocabulary's sub-trees among its sub-trees. In this way, the model uses AST sub-trees as labels, i.e. a piece of information that is already part of the starting data structure without the need for manual labeling or external dependencies. Any parsable source code has its own AST representation and can be used as input for the model. These features permit the generation of code vector representations for any kind of task as the sub-trees labels are not restricted to certain topics but naturally arise from general AST code representation.

The model is pre-trained on unlabeled data and then directly exploitable to create code vectors for our downstream task.

As we already discussed the *Infercode* model has serious scalability limits

TN	**FP**
9385	2193
96	756
FN	**TP**

Table 4.10.: Confusion matrix for binary [M2]

and it has not been successful on most of the files of the dataset. This happens because of the size of the ASTs that could be generated at file-level granularity. For this reason, we suppose that ASTs produced to represent functions have smaller sizes so to test *Infercode* model we focused on the function-level.

The function-level dataset described in Section 4.1 is processed by the *Infercode* encoder which is provided within the homonym *Python* package (PyPI[8]) using all the default parameters. For each function, we obtain a 100-dimensional numeric vector representation that will be the actual input of the final classifier. In vectorizing the functions of our dataset with the provided encoder 1635 input functions still represent a memory problem and have been discarded, however at this granularity level we are able to end up with a dataset composed of 471033 100-dimensional vectors which represent the same number of functions whose distribution among classes has been shown and described in Table 4.3 and Figure 4.2 in Section 4.1.

Model

In the following the tested classifiers are described, many of them have been evaluated in different configurations to find the best performing one using a simple grid search for hyper-parameters tuning.

Not only neural models are used as classifiers, but in fact, also traditional Machine Learning algorithms are implemented to check their effectiveness. However, as the input vector is generated using *Infercode* which is a deep learning model which performs representation learning on source code, the work still lies among the deep learning approaches for defect prediction.

Fully Connected Layers We implemented different configurations for Fully Connected Layers (FCL) Neural Networks using the *Keras* framework [36], in particular, different combinations of layer sizes and network depths are evaluated. The tested sizes are 8, 16, and 32 units per layer and the tested depths are 1, 2, and 3. The dropout strategy is implemented at each hidden

[8]https://pypi.org/

layer with a rate of 0.2.

The activation function for each layer except for the output one is the hyperbolic tangent activation function (*tanh*) as almost any test with the *ReLU* activation brought to a divergent behavior.

Encoder-like model An encoder-like model progressively shrinks the dimensionality layer by layer, i.e. decreasing the representation space dimension, without remaining linked to the first layer dimension. So, starting from a 100-dimensional input vector different sizes for shrinkage are evaluated.

The tested shapes of the encoder (in terms of neural units per layer) are [80, 50, 30] (1), [64, 32, 16] (2), [32, 16, 8] (3) and [64, 32, 16, 8] (4)[9]. Like in the previous FCL case, dropout is used at each hidden layer with a rate of 0.2 and the implementation has been performed by means of the *Keras* [36] *Python* library.

Random Forests We also implemented Machine Learning algorithms different from neural networks. Random Forest is a class of ensemble algorithms based on Decision Trees in which multiple estimators are trained and the classification is obtained by the collective decisions of these multiple classifiers. The hyperparameters which characterize models belonging to this class of algorithms can have different natures, in this study we attempted various configurations for the number of estimators (N) parameter and the minimum number of examples (m) needed to split an internal node. The values for N are 50, 100, 150, and 200 while the ones for m are 2, 4, and 8. The implementation of the Random Forest models has been performed by using the *Scikit-learn Python* library [141, 26].

SVM Between the possible traditional Machine Learning algorithms, we also selected the Support Vector Machine class of models implemented using the *Scikit-learn Python* library [141, 26]. SVM can be built with different kernels and some of them are tested here, in particular we tested the *radial basis function* (*rbf*), *sigmoid* (*s*) and *linear* (*l*) kernels. The other parameter with respect we study the model performance is the regularization parameter C with values 0.001, 0.01, 0.1, 1, and 3.

[9]Numbers in parentheses represent model indexes which will be used to identify the encoder configuration in the following discussions.

Results

In this section, we are going to show the results obtained for the previously described models. Each neural model has been trained on 20 epochs and the evaluation of the goodness of the model is performed by looking at the epoch which shows the best macro average value for recall as in the *code2vec* based model evaluation.

In Table 4.11, 4.12, 4.13 and 4.14 are reported the results for the various configurations of the neural network classifier, Encoder-like models, Random Forests and Support Vector Classifier, respectively.

Depth	**Layer S.**	**Acc**	**AUC**	**R**	**P**	**F**
1	8	0.891	0.748	0.891	0.962	0.924
2	8	0.949	0.501	0.949	0.957	0.953
3	8	0.975	0.608	0.975	0.957	0.966
1	16	0.894	0.753	0.894	0.962	0.926
2	16	0.889	0.751	0.889	0.962	0.923
3	16	0.896	0.76	0.896	0.963	0.927
1	32	0.883	0.768	0.883	0.963	0.919
2	32	0.864	0.77	0.864	0.964	0.909
3	32	0.853	0.772	0.853	0.964	0.903

Table 4.11.: Results for Neural Networks.

Model idx	**Acc**	**AUC**	**R**	**P**	**F**
1	0.842	0.782	0.842	0.965	0.897
2	0.838	0.78	0.838	0.965	0.895
3	0.974	0.591	0.974	0.957	0.965
4	0.871	0.769	0.871	0.964	0.914

Table 4.12.: Results for Encoder-like models.

There is not a clear trend between hyperparameters setting and model performance, however, the best values for the shown metrics seem to be concentrated within the Support Vector classifiers. Even if many SVM-based models share similar results one of the best ones seems to be the one that uses the *sigmoid* kernel with the C parameter set to 1. However, computing the performance metrics values for each class we can see very poor results having R_i=0.0 and P_i=0.0 for every $i = 1, 2, 3$. By inspecting the confusion matrix we discover that this model classifies instances only as belonging to class 0, i.e.

N	m	Acc	AUC	R	P	F
50	2	0.919	0.878	0.919	0.971	0.942
50	4	0.917	0.877	0.917	0.971	0.941
50	8	0.92	0.874	0.92	0.97	0.942
100	2	0.925	0.885	0.925	0.971	0.945
100	4	0.925	0.884	0.925	0.971	0.945
100	8	0.927	0.882	0.927	0.971	0.946
150	2	0.927	0.888	0.927	0.971	0.947
150	4	0.927	0.886	0.927	0.971	0.946
150	8	0.929	0.885	0.929	0.971	0.947
200	2	0.928	**0.889**	0.928	**0.972**	0.947
200	4	0.928	0.888	0.928	0.971	0.947
200	8	0.931	0.886	0.931	0.971	0.948

Table 4.13.: Results for Random Forests.

the model is completely useless. If we use the selection strategy that has been used in epoch selection in the *code2vec* model, i.e., based on the macro-average recall value, the selected model results to be the one with *rbf* kernel and C=3. For this model, the values reported in Table 4.14 are slightly worse than the ones for the model just analyzed but results obtained class by class make more sense now and are reported in Table 4.15.

Again, considering results class by class, the configuration of the neural network which shows the best results can be identified in the one with depth 2 and layer size of 32, and the metrics values for this model are shown in Table 4.16.

Among the encoders, we select the model with index 2, i.e. with structure [64-32-16] whose class by class performance is reported in Table 4.17.

For Random Forests different configurations have similar results, however, one of the best ones is obtained when the N parameter is set to 200 and m is set to 2 whose results are detailed in Table 4.18.

Precision values are high only for the class 0 while for the defective classes they are very low, even lower than in the *code2vec*-based model. Values for recall are more acceptable but even in this case, the results for the *code2vec* model are the best ones.

C	kernel	Acc	AUC	R	P	F
0.001	*rbf*	**0.978**	0.311	**0.978**	0.957	**0.968**
0.01	*rbf*	**0.978**	0.662	**0.978**	0.957	**0.968**
0.1	*rbf*	0.896	0.737	0.896	0.961	0.927
1	*rbf*	0.861	0.786	0.861	0.965	0.908
3	*rbf*	0.849	0.807	0.849	0.966	0.901
0.001	*sigmoid*	**0.978**	0.325	**0.978**	0.957	**0.968**
0.01	*sigmoid*	**0.978**	0.645	**0.978**	0.958	**0.968**
0.1	*sigmoid*	0.948	0.692	0.948	0.958	0.953
1	*sigmoid*	**0.978**	0.694	**0.978**	0.957	**0.968**
3	*sigmoid*	**0.978**	0.581	**0.978**	0.957	**0.968**
0.001	*linear*	0.9	0.743	0.9	0.961	0.929
0.01	*linear*	0.889	0.779	0.889	0.963	0.923
0.1	*linear*	0.881	0.807	0.881	0.964	0.919
1	*linear*	0.89	0.823	0.89	0.965	0.924
3	*linear*	0.895	0.826	0.895	0.965	0.927

Table 4.14.: Results for Support Vector Classifier.

Class	P	R	F
CC	0.986	0.86	0.919
UV	0.042	0.244	0.071
ND	0.056	0.468	0.101
DS	0.028	0.121	0.045

Table 4.15.: Results for each class of the selected SVC model.

4.3. Conclusions about the Software Defect Prediction task

In this Chapter, we approached the problem of detecting three kinds of defects within code fragments by means of learning models. Several attempts to discriminate between clean and defective code have been found in the literature while few works which consider errors identities have been detected.

In this work, we focused on `Uninitialized Value`, `Null Pointer Dereference` and `Dead Store` defects and we tried to detect and identify them within code fragments at two different granularity levels, i.e., file and function levels.

The dataset we used to train the proposed learning methods has been built using the output of a static analyzer and for this reason, the capability of

Class	**P**	**R**	**F**
CC	0.985	0.867	0.922
UV	0.039	0.229	0.066
ND	0.043	0.379	0.078
DS	0.013	0.013	0.013

Table 4.16.: Results for each class of the selected Neural Network model.

Class	**P**	**R**	**F**
CC	0.986	0.85	0.913
UV	0.041	0.241	0.069
ND	0.042	0.423	0.077
DS	0.018	0.037	0.024

Table 4.17.: Results for each class of the selected Encoder model.

our models is limited by the features of the employed analyzer, i.e., we are studying if it is possible to imitate the analyzer behavior exploiting statistical code properties and learning strategies.

Two different approaches for code vectorization have been used in the evaluated architectures, one is based on the *code2vec* [10] model and the other one is based on *Infercode* [25]. Both these vectorization strategies are based on AST code representation. Particularly, the first one is developed to analyze a bag of AST path contexts while the second one just uses ASTs to feed Tree-Based CNN to extract vectors directly from these structures.

The vectors obtained from the different models are then used for the actual code classification with respect to the defective classes. This is performed considering different learning techniques, i.e., using a neural network classifier or traditional machine learning algorithms such as Random Forests and Support Vector Machines.

The *code2vec* based model is trained in an end2end fashion and it focuses on file-level defect identification, it reaches very good average results while when inspecting the performance class by class these can vary a lot depending on the kind of software defect we are considering. Particularly, it is very good with respect to the clean code class identification which also represents the majority class, `Null Pointer Dereference` error result to be the one which is better recognized by this model which is then followed by `Uninitialized Value` and `Dead Store`.

The models showed acceptable values for recall but not a good evaluation with respect to the precision metric, however when inspecting the confusion

Class	**P**	**R**	**F**
CC	0.99	0.938	0.963
UV	0.184	0.456	0.262
ND	0.121	0.549	0.199
DS	0.22	0.394	0.282

Table 4.18.: Results for each class of the selected Random Forest model.

matrices the main reason for this behavior can be found in the relatively high number of clean code examples classified as defective with respect to the number of actual defective code which can be due to the high unbalance which characterize our validation set.

Even if precision values are not satisfying in this work we decided that the recall metric is the most important one as it measures false negatives which for classes $i = 1, 2, 3$ are actual defective codes that are wrongly classified as non-defective or as having a different defect.

The *Infercode* based model focuses on function-level defect identification and it is not an end2end model as the *Infercode* vectorization is treated as a separate step with respect to the actual code classification. Several classifiers have been inspected in this second approach and even in these cases, the average results are very good. However, when it comes to the class by class analysis the model shows difficulties in the error identity discrimination and the overall good performance seems to be only due to the recognition of the clean code. Even in this case recall values are better than the precision ones, however, the *code2vec*-based model results to be the best one.

The difference in the models' capabilities could be due to the different features extraction from codes. In fact, the usage of the direct AST structure for a task-agnostic vectorization seems to not be effective in the characterization of the considered errors while bags of AST path contexts are used in a task-specific way (due to the end2end model training) seem a better strategy for our aims.

A fair comparison to other works is difficult to perform as almost any work in literature manages defect prediction as a binary classification task. However, a recent work presented in [190] treats defects in an individual way and we can compare the results with respect to the *Null Pointer Dereference* defect which is the only one that is also managed by us. In this view, we considerably improved recall performance while slightly worse precision values are found in our study, even if an exact comparison is not possible due to the different datasets used for evaluation.

This pioneering analysis of the defect identification task brought promising results and showed that it is possible to approach this problem using statistical learning strategies and it suggests that improvements are possible together with the opportunity to treat and include other kinds of errors in the proposed settings. As pointed out, we aim to extend this study to an improved dataset built using several analyzers (both static and dynamic) to generate labels for code defects. In this view, our study suggests that working with an end2end task-specific model, like for the case of the *code2vec*-based one, is a preferable strategy.

Many other vectorization and classification strategies can be studied within this topic also considering the different errors nature as some of them can be easily captured in a certain way while others could need a different treatment.

Chapter 5. Conclusions

The work presented in this thesis belongs to a vast research area known as *Big Code* which proposes to apply several learning techniques initially developed within the more generic field of *Big Data* to code data that can be found in repositories and software projects archives. Among the numerous possibilities, two main tasks have been chosen, analyzed, and discussed with respect to several aspects, i.e., Programming Language Identification (PLI) and Software Defect Prediction (SDP).

Two different learning-based approaches have been presented and discussed to deal with the Programming Language Identification problem. The main differences between these approaches stay in the input code representation and the labeling strategies.

The first work represents input code as text and its functioning is based on features extracted from text content, in particular, it exploits feature vectors built from tokens and 2-grams frequencies within texts obtained with a language-agnostic tokenization strategy and defining vocabularies which make the model suffering of the well-known Out-Of-Vocabulary (OOV) problem. These vectors are used as inputs for a neural encoder-like architecture which serves as a classifier with respect to the defined classes. In this approach, external dependencies are avoided and labeling is performed by using the file extensions which are available in the starting dataset and are informative about the programming language. The model reaches $\approx$ 85% average accuracy, answering positively the research question about the possibility to recognize the extension of textual files commonly found in software version control systems repositories, based solely on file contents. In addition, the model's simplicity suggests it can be easily maintained in the future.

The second work represents code as images and uses pre-trained architectures which are usually employed in image recognition tasks fine-tuned to perform the PLI task. Representing code as images also permits to reduce the severity of OOV-related issues, being no vocabulary definition here. The labels for the dataset are generated using the tool *Linguist*, this permits the direct classification with respect to the programming languages at the expense of adding dependency from the external tool. MobileNet resulted to be the best model reaching $\approx$ 93% accuracy improving the state-of-the-art even with respect to the number of considered languages. This permits to answer positively to the research question about the model's capability to identify the programming language used in code snippet images without any a priori knowledge. Treating input code as images we also investigated which classes of characters contribute the most to the language identification by performing scrambling of characters and discovering that punctuation symbols are the most relevant characters in PLI.

Both models brought satisfying results, especially because we focused on the model's capability of handling a relatively high number of languages. The main difficulty in developing these works has been due to the lack of reliable labels for data. This fact deeply affects the goodness of the models as well as the evaluation itself which depends on labels too. The lack of labels for the PLI task is a serious issue and no dataset which solves this problem has been found, also it could be a very expensive task to reliably label such a dataset. From here we can see how the lack of a curated benchmark dataset limits models analysis, evaluation, and fair comparison. Building such a dataset could be a viable path for future works on this topic.

The second task we focused on is the Software Defect Prediction performed using statistical code properties and learning algorithms. Three different errors have been selected and our aim is to approximate the static analyzer behavior in catching and recognizing these errors, instead of the most common approaches in literature which treat defective code as a unique class. Two approaches for code vectorization have been explored and both of them are based on the AST representation of code.

The *Infercode* model generates vectors by means of a pre-trained tree-based CNN in a task-agnostic way. We used this vectorization approach for function-level defect prediction and used the obtained vectors in various learning architectures.

The *code2vec* model works in an end2end way and learns features from bags of path-contexts extracted from code AST. We used this model for file-level defect prediction and resulted in being the best model. For this task the overall performance is promising as it reaches $\approx$ 80% accuracy but when inspecting them class by class some metrics values (especially precision) considerably drop mostly because of clean code (the majority class) classified as defective. The best recognized error is the Null Pointer Dereference improving recall values for this class with respect to another work treating this error. The different behavior with respect to different errors suggests that they could need different treatment as their nature can vary among them.

Given these results, we aim to build a new dataset that would not be labeled by the static analyzer but which exploits several techniques together as well as commit messages contents and so on, in order to make bug labeling more reliable and overcome the limitations given by the static analyzer labeling strategy.

Both the analyzed tasks showed that it is possible, to a certain extent, to use machine learning techniques using statistical properties of code to approximate the stated goals and possibly build some supporting tools that could be used as heuristics. However, in both cases, a huge limitation in building, training, and evaluating such models comes from the lack of precisely labeled datasets as labels are usually provided as the outputs of other external tools making the models limited in imitating these tools' behavior which almost never represent the *truth* but which are used as such. Manual labeling and joint usage of several external tools could be the way to address this issue even if it can result in a very expensive but highly valuable task. In the future, we aim to overcome such difficulties and apply the preliminary results obtained during this analysis together with other possible approaches to more curated datasets.

Appendix A.

Results for File Extension Identification

In the following Table are presented the values for each class of the selected metrics for the File Extension Identification model presented in Section 3.1.

Ext	**bigrams**			**trigrams**			**Δ**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`.js`	0.93	0.69	0.79	0.93	0.61	0.74	0.0	0.08	0.05
`.c`	0.96	0.94	0.95	0.96	0.94	0.95	0.0	0.0	0.0
`.html`	0.98	0.87	0.92	0.98	0.87	0.92	0.0	0.0	0.0
`.java`	0.99	0.97	0.98	0.98	0.97	0.97	0.01	0.0	0.01
`.h`	0.93	0.71	0.81	0.9	0.66	0.76	0.03	0.05	0.05
`.py`	0.99	0.93	0.96	0.99	0.86	0.92	0.0	0.07	0.04
`.go`	0.99	0.96	0.97	0.99	0.96	0.97	0.0	0.0	0.0
`.md`	0.97	0.72	0.83	0.96	0.7	0.81	0.01	0.02	0.02
`.rb`	0.98	0.85	0.91	0.98	0.69	0.81	0.0	0.16	0.1
`.json`	0.95	0.95	0.95	0.95	0.93	0.94	0.0	0.02	0.01
`.cpp`	0.74	0.59	0.66	0.65	0.19	0.29	0.09	0.4	0.37
`.ts`	0.65	0.82	0.73	0.62	0.78	0.69	0.03	0.04	0.04
`.php`	0.96	0.89	0.92	0.95	0.88	0.91	0.01	0.01	0.01
`.cs`	0.98	0.96	0.97	0.91	0.95	0.93	0.07	0.01	0.04
`.rs`	0.97	0.97	0.97	0.98	0.95	0.96	-0.01	0.02	0.01
`.cc`	0.58	0.77	0.66	0.42	0.87	0.57	0.16	-0.1	0.09
`.xml`	0.96	0.93	0.94	0.94	0.94	0.94	0.02	-0.01	0.0
`.glif`	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0
`.txt`	0.82	0.52	0.64	0.84	0.42	0.56	-0.02	0.1	0.08
`.kt`	0.96	0.72	0.82	0.94	0.71	0.81	0.02	0.01	0.01
`.scala`	0.96	0.95	0.95	0.94	0.93	0.93	0.02	0.02	0.02
`.swift`	0.92	0.85	0.88	0.89	0.8	0.84	0.03	0.05	0.04
`.yml`	0.77	0.81	0.79	0.79	0.68	0.73	-0.02	0.13	0.06
`.css`	0.88	0.9	0.89	0.88	0.93	0.9	0.0	-0.03	-0.01
`.rst`	0.59	0.89	0.71	0.52	0.9	0.66	0.07	-0.01	0.05

Ext	**bigrams**			**trigrams**			**Δ**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`.sh`	0.84	0.83	0.83	0.86	0.7	0.77	-0.02	0.13	0.06
`.csproj`	0.96	0.9	0.93	0.99	0.53	0.69	-0.03	0.37	0.24
`.coffee`	0.82	0.92	0.87	0.76	0.89	0.82	0.06	0.03	0.05
`.scss`	0.85	0.93	0.89	0.89	0.9	0.89	-0.04	0.03	0.0
`.phpt`	0.68	0.94	0.79	0.73	0.89	0.8	-0.05	0.05	-0.01
`.jsx`	0.26	0.79	0.39	0.23	0.79	0.36	0.03	0.0	0.03
`.hpp`	0.2	0.83	0.32	0.16	0.83	0.27	0.04	0.0	0.05
`.xht`	0.79	0.97	0.87	0.79	0.94	0.86	0.0	0.03	0.01
`.tsx`	0.34	0.81	0.48	0.25	0.68	0.37	0.09	0.13	0.11
`.jl`	0.9	0.79	0.84	0.75	0.84	0.79	0.15	-0.05	0.05
`.htm`	0.41	0.93	0.57	0.42	0.92	0.58	-0.01	0.01	-0.01
`.dart`	0.74	0.98	0.84	0.65	0.97	0.78	0.09	0.01	0.06
`.ml`	0.98	0.95	0.96	0.94	0.94	0.94	0.04	0.01	0.02
`.m`	0.72	0.95	0.82	0.73	0.94	0.82	-0.01	0.01	0.0
`.haml`	0.92	0.98	0.95	0.91	0.96	0.93	0.01	0.02	0.02
`.yaml`	0.46	0.65	0.54	0.37	0.7	0.48	0.09	-0.05	0.06
`.vb`	0.95	0.99	0.97	0.95	0.99	0.97	0.0	0.0	0.0
`.asciidoc`	0.64	0.95	0.76	0.48	0.92	0.63	0.16	0.03	0.13
`.gradle`	0.75	0.95	0.84	0.48	0.93	0.63	0.27	0.02	0.21
`.cr`	0.34	0.92	0.5	0.14	0.93	0.24	0.2	-0.01	0.26
`.lua`	0.7	0.95	0.81	0.7	0.93	0.8	0.0	0.02	0.01
`.ex`	0.83	0.96	0.89	0.69	0.95	0.8	0.14	0.01	0.09
`.ilproj`	0.79	0.95	0.86	0.39	1.0	0.56	0.4	-0.05	0.3
`.dtsi`	0.64	0.88	0.74	0.7	0.78	0.74	-0.06	0.1	0.0
`.props`	0.79	0.97	0.87	0.73	0.96	0.83	0.06	0.01	0.04
`.vcxproj`	0.97	0.99	0.98	0.95	0.99	0.97	0.02	0.0	0.01
`.clj`	0.96	0.99	0.97	0.93	0.99	0.96	0.03	0.0	0.01
`.markdown`	0.13	0.73	0.22	0.13	0.7	0.22	0.0	0.03	0.0
`.symbols`	0.88	0.99	0.93	0.91	0.99	0.95	-0.03	0.0	-0.02
`.hs`	0.78	0.99	0.87	0.63	0.99	0.77	0.15	0.0	0.1
`.dts`	0.75	0.69	0.72	0.69	0.82	0.75	0.06	-0.13	-0.03
`.el`	0.95	0.99	0.97	0.97	0.99	0.98	-0.02	0.0	-0.01
`.proto`	0.53	0.99	0.69	0.47	0.98	0.64	0.06	0.01	0.05
`.toml`	0.6	0.98	0.74	0.7	0.97	0.81	-0.1	0.01	-0.07
`.pbxproj`	0.98	1.0	0.99	0.94	1.0	0.97	0.04	0.0	0.02
`.exs`	0.56	0.95	0.7	0.44	0.94	0.6	0.12	0.01	0.1
`.mk`	0.79	0.91	0.85	0.7	0.92	0.8	0.09	-0.01	0.05
`.sil`	0.71	0.97	0.82	0.57	0.98	0.72	0.14	-0.01	0.1

Ext	bigrams			trigrams			Δ		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`.after`	0.19	0.74	0.3	0.18	0.78	0.29	0.01	-0.04	0.01
`.erb`	0.19	0.77	0.3	0.14	0.73	0.23	0.05	0.04	0.07
`.jade`	0.27	0.9	0.42	0.26	0.89	0.4	0.01	0.01	0.02
`.gyb`	0.33	0.97	0.49	0.26	0.94	0.41	0.07	0.03	0.08
`.log`	0.57	0.91	0.7	0.62	0.91	0.74	-0.05	0.0	-0.04
`.ipynb`	0.43	0.98	0.6	0.27	0.99	0.42	0.16	-0.01	0.18
`.cmake`	0.29	0.95	0.44	0.27	0.94	0.42	0.02	0.01	0.02
`.ps1`	0.78	0.96	0.86	0.79	0.95	0.86	-0.01	0.01	0.0
`.pyx`	0.43	0.95	0.59	0.14	0.96	0.24	0.29	-0.01	0.35
`.tmpl`	0.24	0.67	0.35	0.23	0.65	0.34	0.01	0.02	0.01
`.m4`	0.74	0.95	0.83	0.5	0.9	0.64	0.24	0.05	0.19
`.check`	0.24	0.81	0.37	0.2	0.78	0.32	0.04	0.03	0.05
`.il`	0.77	1.0	0.87	0.8	1.0	0.89	-0.03	0.0	-0.02
`.am`	0.54	0.96	0.69	0.71	0.93	0.81	-0.17	0.03	-0.12
`.adoc`	0.58	0.95	0.72	0.39	0.95	0.55	0.19	0.0	0.17
`.mli`	0.68	1.0	0.81	0.61	0.99	0.75	0.07	0.01	0.06
`.sln`	0.99	1.0	0.99	0.98	1.0	0.99	0.01	0.0	0.0
`.sass`	0.69	0.96	0.8	0.67	0.96	0.79	0.02	0.0	0.01
`.gyp`	0.32	1.0	0.48	0.34	0.98	0.5	-0.02	0.02	-0.02
`.bat`	0.42	0.72	0.53	0.36	0.72	0.48	0.06	0.0	0.05
`.erl`	0.6	1.0	0.75	0.31	1.0	0.47	0.29	0.0	0.28
`.gemspec`	0.78	1.0	0.88	0.84	1.0	0.91	-0.06	0.0	-0.03
`.fish`	0.58	0.95	0.72	0.65	0.94	0.77	-0.07	0.01	-0.05
`.i`	0.09	0.89	0.16	0.11	0.81	0.19	-0.02	0.08	-0.03
`.texi`	0.93	1.0	0.96	0.88	1.0	0.94	0.05	0.0	0.02
`.template`	0.09	0.48	0.15	0.03	0.37	0.06	0.06	0.11	0.09
`.pl`	0.62	0.95	0.75	0.41	0.95	0.57	0.21	0.0	0.18
`.ac`	0.68	0.99	0.81	0.6	0.99	0.75	0.08	0.0	0.06
`.groovy`	0.23	0.91	0.37	0.15	0.86	0.26	0.08	0.05	0.11
`.mak`	0.86	0.96	0.91	0.67	0.95	0.79	0.19	0.01	0.12
`.vbproj`	0.54	0.98	0.7	0.36	1.0	0.53	0.18	-0.02	0.17
`.pkgproj`	0.85	0.98	0.91	0.56	0.99	0.72	0.29	-0.01	0.19
`.sql`	0.16	0.93	0.27	0.14	0.92	0.24	0.02	0.01	0.03
`.j`	0.22	0.98	0.36	0.27	0.94	0.42	-0.05	0.04	-0.06
`.tpl`	0.12	0.77	0.21	0.1	0.74	0.18	0.02	0.03	0.03
`.rake`	0.1	0.95	0.18	0.09	0.92	0.16	0.01	0.03	0.02
`.textile`	0.2	0.99	0.33	0.15	0.97	0.26	0.05	0.02	0.07
`.webidl`	0.59	0.99	0.74	0.3	0.98	0.46	0.29	0.01	0.28

Ext	**bigrams**			**trigrams**			**Δ**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`.bash`	0.22	0.77	0.34	0.13	0.75	0.22	0.09	0.02	0.12
`.cjsx`	0.35	0.97	0.51	0.32	0.93	0.48	0.03	0.04	0.03
`.pb`	0.39	1.0	0.56	0.54	0.99	0.7	-0.15	0.01	-0.14
`.builds`	0.93	1.0	0.96	0.96	0.99	0.97	-0.03	0.01	-0.01
`.vcproj`	0.92	1.0	0.96	0.94	0.99	0.96	-0.02	0.01	0.0
`.xcscheme`	1.0	1.0	1.0	0.98	1.0	0.99	0.02	0.0	0.01
`.ngdoc`	0.1	0.96	0.18	0.07	0.97	0.13	0.03	-0.01	0.05
`.perl`	0.66	0.98	0.79	0.64	0.98	0.77	0.02	0.0	0.02
`.eslintrc`	0.13	0.98	0.23	0.16	0.98	0.28	-0.03	0.0	-0.05
`.sbt`	0.61	0.97	0.75	0.41	0.98	0.58	0.2	-0.01	0.17
`.handlebars`	0.1	0.97	0.18	0.12	0.98	0.21	-0.02	-0.01	-0.03
`.iml`	0.62	1.0	0.77	0.75	1.0	0.86	-0.13	0.0	-0.09
`.rml`	0.79	1.0	0.88	0.87	0.99	0.93	-0.08	0.01	-0.05
`.cmd`	0.28	0.81	0.42	0.28	0.85	0.42	0.0	-0.04	0.0
`.zsh`	0.2	0.94	0.33	0.17	0.91	0.29	0.03	0.03	0.04
`.tcl`	0.65	0.98	0.78	0.52	0.97	0.68	0.13	0.01	0.1
`.xib`	0.55	1.0	0.71	0.63	1.0	0.77	-0.08	0.0	-0.06
`.jet`	0.05	0.91	0.09	0.05	0.81	0.09	0.0	0.1	0.0
`.dsp`	0.94	1.0	0.97	0.93	1.0	0.96	0.01	0.0	0.01
`.w32`	0.31	0.99	0.47	0.14	0.99	0.25	0.17	0.0	0.22
micro avg.	0.85	0.85	0.85	0.81	0.81	0.81	0.04	0.04	0.04
macro avg.	0.64	0.91	0.71	0.59	0.89	0.66	0.05	0.02	0.05

Table A.1.: Performance of the encoder architecture without and with trigrams.

Appendix B.

Results for Image-based Programming Language Identification

In the following Table are presented the values for each class of the selected metrics for the Image-based Programming Language Identification model presented in Section 3.2.

	ResNet34			**MobileNetv2**			**AlexNet**		
Language	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
ANTLR	0.95	0.99	0.97	0.97	0.99	0.98	0.94	0.97	0.96
ActionScript	0.78	0.91	0.84	0.76	0.93	0.83	0.65	0.72	0.68
AGC	0.99	1.00	1.00	0.99	1.00	0.99	0.97	0.99	0.98
AsciiDoc	0.98	0.94	0.96	0.98	0.95	0.97	0.78	0.86	0.81
Assembly	0.94	0.87	0.90	0.97	0.88	0.92	0.86	0.83	0.84
Batchfile	0.98	0.99	0.98	1.00	0.99	0.99	0.91	0.94	0.93
C	0.84	0.92	0.88	0.83	0.94	0.88	0.73	0.75	0.74
C#	0.83	0.91	0.87	0.88	0.91	0.89	0.74	0.83	0.78
C++	0.64	0.66	0.65	0.69	0.70	0.70	0.47	0.39	0.43
CMake	0.97	0.98	0.98	0.97	0.99	0.98	0.81	0.89	0.84
CSON	1.00	0.99	1.00	1.00	0.99	1.00	0.96	0.99	0.98
CSS	0.81	0.87	0.84	0.78	0.91	0.84	0.68	0.72	0.70
CSV	0.97	0.58	0.73	0.98	0.57	0.72	0.93	0.40	0.56
Cabal Config	1.00	0.99	0.99	0.99	1.00	0.99	0.98	0.99	0.99
Cap'n Proto	1.00	1.00	1.00	1.00	1.00	1.00	0.96	0.99	0.98
Clojure	0.98	0.96	0.97	0.96	0.97	0.97	0.84	0.90	0.87
CoffeeScript	0.97	0.88	0.92	0.96	0.89	0.92	0.83	0.85	0.84
Crystal	0.83	0.89	0.86	0.84	0.86	0.85	0.73	0.76	0.74
Cuda	0.78	0.88	0.83	0.72	0.91	0.80	0.56	0.81	0.67
Cython	0.84	0.86	0.85	0.83	0.90	0.87	0.66	0.76	0.70
DIGITAL CL	1.00	0.99	1.00	1.00	0.99	1.00	0.96	0.99	0.98
Dart	0.94	0.95	0.94	0.93	0.96	0.95	0.80	0.84	0.82

	ResNet34			**MobileNetv2**			**AlexNet**		
Language	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
Diff	0.97	0.97	0.97	0.99	0.96	0.97	0.93	0.93	0.93
Dockerfile	0.99	1.00	1.00	1.00	1.00	1.00	0.98	0.99	0.99
eC	0.96	0.97	0.97	0.97	0.98	0.98	0.88	0.95	0.91
EJS	0.96	0.88	0.92	0.96	0.92	0.94	0.83	0.81	0.82
EML	0.92	0.97	0.94	0.92	0.96	0.94	0.89	0.85	0.87
Elixir	0.98	0.95	0.96	0.93	0.92	0.93	0.92	0.84	0.88
Emacs Lisp	0.98	0.96	0.97	0.98	0.98	0.98	0.82	0.83	0.82
Erlang	0.98	0.98	0.98	0.98	0.99	0.98	0.94	0.95	0.94
fish	0.98	0.99	0.99	0.99	0.99	0.99	0.93	0.91	0.92
FreeMarker	0.90	0.95	0.93	0.91	0.95	0.93	0.86	0.92	0.89
GAP	0.96	0.89	0.93	0.95	0.91	0.93	0.86	0.77	0.81
GDB	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.99	0.99
GLSL	0.96	0.94	0.95	0.96	0.92	0.94	0.84	0.81	0.83
GN	0.99	1.00	0.99	1.00	1.00	1.00	0.91	0.99	0.95
Gettext C.	0.98	1.00	0.99	0.98	1.00	0.99	0.99	0.99	0.99
Gherkin	0.98	0.99	0.99	1.00	0.99	1.00	1.00	0.95	0.97
Go	0.92	0.81	0.86	0.93	0.80	0.86	0.70	0.70	0.70
Gradle	0.91	0.93	0.92	0.94	0.93	0.93	0.70	0.74	0.72
GraphQL	0.99	0.98	0.98	0.99	1.00	0.99	0.96	0.90	0.93
Graphviz	0.99	0.96	0.97	1.00	0.97	0.98	0.97	0.92	0.94
Groovy	0.89	0.91	0.90	0.86	0.92	0.89	0.71	0.79	0.75
HAProxy	1.00	0.99	1.00	0.99	1.00	1.00	0.98	0.99	0.98
HTML	0.79	0.54	0.64	0.81	0.60	0.69	0.69	0.53	0.60
HTML+Django	0.89	0.95	0.92	0.75	0.95	0.84	0.83	0.92	0.87
HTML+ERB	0.85	0.89	0.87	0.90	0.90	0.90	0.74	0.80	0.77
HTML+Razor	0.94	0.94	0.94	0.97	0.93	0.95	0.83	0.86	0.85
Hack	0.93	0.95	0.94	0.96	0.96	0.96	0.90	0.92	0.91
Haml	1.00	0.99	1.00	1.00	0.99	1.00	0.97	0.97	0.97
Handlebars	0.97	0.93	0.95	0.96	0.94	0.95	0.91	0.85	0.88
Haskell	0.95	0.70	0.81	0.96	0.72	0.82	0.81	0.64	0.71
INI	0.92	0.99	0.95	0.93	0.99	0.96	0.84	0.92	0.88
Ignore List	1.00	0.99	1.00	0.99	1.00	1.00	0.96	0.99	0.97
Inno Setup	1.00	0.97	0.99	0.99	0.97	0.98	0.98	0.96	0.97
JSON	0.98	0.94	0.96	0.97	0.95	0.96	0.79	0.40	0.53
JSON5	1.00	0.99	1.00	0.99	1.00	0.99	1.00	0.99	0.99
JSX	0.72	0.56	0.63	0.76	0.68	0.72	0.52	0.49	0.51

	ResNet34			**MobileNetv2**			**AlexNet**		
Language	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
Java	0.81	0.81	0.81	0.84	0.81	0.83	0.69	0.59	0.64
Java Prop.	0.97	0.99	0.98	0.99	0.98	0.98	0.83	0.93	0.88
Java SP	0.96	0.93	0.95	0.98	0.93	0.96	0.92	0.84	0.88
JavaScript	0.59	0.62	0.60	0.59	0.58	0.59	0.41	0.36	0.38
Jison	0.97	0.99	0.98	0.98	0.99	0.99	0.94	0.99	0.97
Julia	0.91	0.85	0.88	0.88	0.84	0.86	0.76	0.78	0.77
Jupyter Not.	0.97	0.99	0.98	0.97	0.99	0.98	0.90	0.94	0.92
Kotlin	0.91	0.95	0.93	0.92	0.97	0.94	0.69	0.84	0.76
LLVM	1.00	0.84	0.92	0.99	0.88	0.93	0.98	0.68	0.80
Less	0.67	0.68	0.67	0.76	0.67	0.71	0.52	0.46	0.49
Lex	0.95	0.91	0.93	0.98	0.92	0.95	0.88	0.87	0.88
Linux KM	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Lua	0.83	0.94	0.88	0.96	0.94	0.95	0.73	0.82	0.77
M4Sugar	0.98	0.94	0.96	1.00	0.93	0.96	0.91	0.83	0.87
MLIR	0.98	0.99	0.98	0.98	0.99	0.99	0.96	0.98	0.97
Makefile	0.88	0.97	0.92	0.88	0.97	0.92	0.77	0.93	0.84
Mako	0.99	0.97	0.98	0.99	0.96	0.97	0.96	0.90	0.93
Markdown	0.85	0.93	0.89	0.91	0.90	0.90	0.72	0.58	0.64
Micr. DSP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
MMS	0.99	0.99	0.99	0.99	1.00	1.00	0.97	0.99	0.98
Assembly	0.69	0.71	0.70	0.72	0.72	0.72	0.44	0.53	0.48
NASL	0.93	0.81	0.87	0.91	0.81	0.85	0.89	0.78	0.83
NSIS	0.99	1.00	0.99	0.96	1.00	0.98	0.92	0.96	0.94
OCaml	0.95	0.91	0.93	0.98	0.94	0.96	0.88	0.75	0.81
Object.-C	0.71	0.81	0.75	0.76	0.81	0.78	0.60	0.66	0.63
Object.-C++	0.73	0.62	0.67	0.82	0.71	0.76	0.60	0.49	0.54
Objective-J	0.88	0.93	0.90	0.86	0.93	0.90	0.80	0.88	0.84
OpenCL	0.91	0.96	0.93	0.92	0.97	0.94	0.76	0.90	0.83
OpenType FF	1.00	0.96	0.98	0.99	0.93	0.96	0.97	0.82	0.89
Org	1.00	0.99	1.00	0.99	0.99	0.99	0.93	0.94	0.94
PHP	0.90	0.89	0.89	0.93	0.89	0.91	0.80	0.78	0.79
PLpgSQL	0.90	0.96	0.93	0.90	0.96	0.93	0.86	0.86	0.86
Perl	0.96	0.96	0.96	0.98	0.95	0.96	0.89	0.80	0.84
Pod	0.97	0.98	0.98	0.97	0.98	0.97	0.75	0.88	0.81
PowerShell	0.96	0.97	0.97	0.99	0.96	0.98	0.87	0.95	0.91
Proguard	0.99	0.97	0.98	0.99	0.98	0.99	0.99	0.80	0.88

	ResNet34			**MobileNetv2**			**AlexNet**		
Language	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
Prot. Buffer	0.99	0.98	0.99	0.97	0.97	0.97	0.89	0.93	0.91
Pug	0.96	0.94	0.95	0.98	0.96	0.97	0.85	0.77	0.81
Python	0.80	0.78	0.79	0.82	0.74	0.78	0.60	0.54	0.56
QML	0.96	0.98	0.97	0.97	0.98	0.97	0.78	0.90	0.83
QMake	0.98	0.98	0.98	0.99	0.99	0.99	0.96	0.96	0.96
R	0.98	0.99	0.99	0.98	0.99	0.99	0.95	0.97	0.96
RDoc	0.98	0.97	0.97	0.97	0.97	0.97	0.76	0.82	0.79
RMarkdown	0.99	0.98	0.99	0.97	1.00	0.98	0.93	0.97	0.95
RPM Spec	0.99	0.99	0.99	0.98	1.00	0.99	0.97	0.94	0.96
Ragel	0.94	0.94	0.94	0.92	0.96	0.94	0.81	0.83	0.82
Rascal	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	0.99
ReST	0.94	0.92	0.93	0.93	0.94	0.94	0.76	0.83	0.79
RTF	0.97	0.98	0.98	0.98	0.99	0.99	0.93	0.94	0.94
Roff	0.90	0.66	0.76	0.89	0.65	0.75	0.85	0.64	0.73
Roff Manp.	0.73	0.90	0.81	0.73	0.92	0.81	0.71	0.86	0.78
Ruby	0.81	0.81	0.81	0.77	0.83	0.80	0.66	0.72	0.69
Rust	0.88	0.91	0.89	0.92	0.93	0.92	0.66	0.83	0.74
SCSS	0.75	0.78	0.77	0.77	0.82	0.79	0.60	0.70	0.64
SQL	0.88	0.69	0.77	0.69	0.67	0.68	0.75	0.53	0.62
SVG	0.73	0.80	0.77	0.82	0.85	0.83	0.87	0.66	0.75
SWIG	0.84	0.91	0.87	0.84	0.89	0.86	0.69	0.71	0.70
SaltStack	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00
Sass	0.97	0.94	0.96	0.98	0.96	0.97	0.93	0.91	0.92
Scala	0.96	0.93	0.94	0.95	0.94	0.94	0.77	0.88	0.82
Scheme	0.94	0.99	0.97	0.97	0.99	0.98	0.90	0.97	0.94
Shell	0.88	0.91	0.89	0.89	0.93	0.91	0.62	0.64	0.63
Slash	0.99	1.00	0.99	0.99	1.00	1.00	0.90	0.99	0.94
Smarty	0.95	1.00	0.97	0.96	1.00	0.98	0.90	1.00	0.95
Starlark	0.96	0.94	0.95	0.95	0.95	0.95	0.91	0.86	0.88
Stylus	0.90	0.94	0.92	0.94	0.94	0.94	0.86	0.89	0.88
Svelte	0.88	0.98	0.93	0.89	0.98	0.93	0.78	0.92	0.84
Swift	0.89	0.90	0.89	0.90	0.84	0.87	0.57	0.48	0.52
TOML	0.98	0.97	0.98	0.98	0.98	0.98	0.93	0.95	0.94
TSQL	0.76	0.88	0.82	0.72	0.84	0.78	0.69	0.77	0.73
TSX	0.64	0.73	0.68	0.65	0.63	0.64	0.57	0.43	0.49
Tcl	0.92	0.99	0.96	0.90	0.99	0.95	0.81	0.94	0.87

	ResNet34			**MobileNetv2**			**AlexNet**		
Language	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
TeX	0.99	0.96	0.97	0.98	0.95	0.97	0.93	0.88	0.91
Texinfo	0.99	0.99	0.99	1.00	1.00	1.00	0.94	0.92	0.93
Text	0.66	0.90	0.76	0.64	0.88	0.74	0.82	0.71	0.76
Textile	0.97	0.95	0.96	0.98	0.95	0.97	0.94	0.89	0.92
Twig	0.95	0.96	0.96	0.94	0.95	0.95	0.86	0.88	0.87
TypeScript	0.78	0.78	0.78	0.77	0.79	0.78	0.45	0.35	0.40
UnixAssembly	0.81	0.89	0.84	0.81	0.89	0.85	0.60	0.79	0.68
Vim Snippet	0.98	0.91	0.94	0.99	0.94	0.97	0.96	0.85	0.90
Vim script	0.96	0.99	0.98	0.92	0.99	0.96	0.87	0.91	0.89
VB .NET	0.99	0.94	0.96	0.96	0.96	0.96	0.95	0.90	0.93
Vue	0.91	0.84	0.87	0.94	0.86	0.90	0.76	0.74	0.75
Wavefront	1.00	0.91	0.95	1.00	0.89	0.94	0.90	0.67	0.77
WebIDL	0.99	1.00	1.00	0.98	1.00	0.99	0.94	0.99	0.97
Windows RE	0.99	1.00	1.00	0.99	1.00	1.00	0.98	0.99	0.99
XML	0.79	0.97	0.87	0.89	0.85	0.87	0.63	0.81	0.71
XML Pr. List	1.00	1.00	1.00	0.99	1.00	1.00	0.99	0.99	0.99
XSLT	0.95	0.57	0.71	0.98	0.73	0.84	0.79	0.54	0.64
YAML	0.94	0.99	0.96	0.96	0.99	0.97	0.88	0.92	0.90
Yacc	0.96	0.97	0.97	0.95	0.94	0.95	0.85	0.85	0.85
Micro avg.	0.92	0.92	0.92	0.92	0.92	0.92	0.83	0.83	0.83
Macro avg.	0.92	0.92	0.92	0.93	0.92	0.92	0.83	0.83	0.83

Appendix C.

Issues managed by Infer

Complete list of issues detectable by means of the *Infer* static analyzer.

Issue	
arbitrary code execution under lock	C/C++
assign pointer warning	C/C++
autoreleasepool size complexity increase	C/C++
autoreleasepool size complexity increase ui thread	C/C++
autoreleasepool size unreachable at exit	C/C++
bad pointer comparison	C/C++
buffer overrun	C/C++
captured strong self	C/C++
checkers allocates memory	C/C++
checkers annotation reachability error	C/C++
checkers calls expensive method	C/C++
checkers expensive overrides unannotated	C/C++
checkers fragment retains view	C/C++
checkers immutable cast	C/C++
checkers printf args	C/C++
component initializer with side effects	C/C++
component with multiple factory methods	C/C++
constant address dereference	C/C++
cxx reference captured in objc block	C/C++
deadlock	C/C++
dead store	C/C++
direct atomic property access	C/C++
discouraged weak property custom setter	C/C++
empty vector access	C/C++
eradicate condition redundant	C/C++
eradicate field not initialized	C/C++
eradicate field not nullable	C/C++

eradicate inconsistent subclass parameter annotation	C/C++
eradicate inconsistent subclass return annotation	C/C++
eradicate meta class needs improvement	C/C++
eradicate parameter not nullable	C/C++
eradicate return not nullable	C/C++
eradicate return over annotated	C/C++
execution time complexity increase	C/C++
execution time complexity increase ui thread	C/C++
execution time unreachable at exit	C/C++
expensive autoreleasepool size	C/C++
expensive execution time	C/C++
expensive loop invariant call	C/C++
global variable initialized with function or method call	C/C++
guardedby violation	C/C++
impure function	C/C++
inefficient keyset iterator	C/C++
infinite autoreleasepool size	C/C++
infinite execution time	C/C++
integer overflow	C/C++
interface not thread safe	C/C++
invariant call	C/C++
ivar not null checked	C/C++
lockless violation	C/C++
lock consistency violation	C/C++
memory leak	C/C++
mixed self weakself	C/C++
modifies immutable	C/C++
multiple weakself	C/C++
mutable local variable in component file	C/C++
nullptr dereference	C/C++
optional empty access	C/C++
parameter not null checked	C/C++
pointer to const objc class	C/C++
premature nil termination argument	C/C++
pure function	C/C++
resource leak	C/C++
retain cycle	C/C++
stack variable address escape	C/C++
starvation	C/C++
static initialization order fiasco	C/C++

strict mode violation	C/C++
strong delegate warning	C/C++
strong self not checked	C/C++
thread safety violation	C/C++
uninitialized value	C/C++
use after delete	C/C++
use after free	C/C++
use after lifetime	C/C++
vector invalidation	C/C++
weak self in no escape block	C/C++
assign pointer warning solo objc	other
autoreleasepool size complexity increase sol objc	other
autoreleasepool size complexity increase ui thread solo objc	other
bad pointer comparison solo objc	other
checkers immutable cast	other
expensive autoreleasepool size solo objc	other
infinite autoreleasepool size solo objc	other
ivar not null checked solo objc	other
mixed self weakself solo objc	other
multiple weakself solo objc	other
parameter not null checked solo objc	other
pointer to const objc class solo objc	other
starvation	other
strict mode violation	other
thread safety violation	other

Table C.1.: Issues detectable using *Infer*

Bibliography

[1] Zakrani abdelali, Hain Mustapha, and Namir Abdelwahed. Investigating the use of random forest in software effort estimation. *Procedia Computer Science*, 148:343–352, 2019. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2019.01.042. URL https://www.sciencedirect.com/science/article/pii/S1877050919300420. The Second International Conference on Intelligent Computing in Data Sciences, ICDS2018.

[2] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, October 2018. ISSN 0001-0782. doi: 10.1145/3183558.

[3] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, September 2018. ISSN 0001-0782. doi: 10.1145/3183558. URL http://doi.acm.org/10.1145/3183558.

[4] Petar Afric, Lucija Sikic, Adrian Satja Kurdija, Goran Delac, and Marin Silic. Repd: Source code defect prediction as anomaly detection. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 227–234, 2019. doi: 10.1109/QRS-C.2019.00052.

[5] Elena Akimova, Alexander Bersenev, Artem Deikov, Konstantin Kobylkin, Anton Konygin, Ilya Mezentsev, and Vladimir Misilov. A survey on software defect prediction using deep learning. *Mathematics*, 9:1180, 05 2021. doi: 10.3390/math9111180.

[6] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849.

[7] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, 2016, New York City, NY, USA, June 19-24, 2016*, pages 2091–2100, 2016. URL http://proceedings.mlr.press/v48/allamanis16.html.

[8] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[9] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *CoRR*, abs/1808.01400, 2018. URL http://arxiv.org/abs/1808.01400.

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290353. URL https://doi.org/10.1145/3290353.

[11] Kamel Alrashedy, Dhanush Dharmaretnam, Daniel M. German, Venkatesh Srinivasan, and T. Aaron Gulliver. Scc++: Predicting the programming language of questions and snippets of stack overflow. *Journal of Systems and Software*, 162:110505, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110505. URL https://www.sciencedirect.com/science/article/pii/S0164121219302791.

[12] Kamel Alreshedy, Dhanush Dharmaretnam, Daniel M. German, Venkatesh Srinivasan, and T. Aaron Gulliver. Scc: Automatic classification of code snippets. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 203–208, 2018. doi: 10.1109/SCAM.2018.00031.

[13] Hlib Babii, Andrea Janes, and Romain Robbes. Modeling vocabulary for big code machine learning. *CoRR*, abs/1904.01873, 2019. URL http://arxiv.org/abs/1904.01873.

[14] Y. Bengio. Learning deep architectures for ai. *Foundations*, 2:1–55, 01 2009. doi: 10.1561/2200000006.

[15] Emery D Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: a reproduction study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(4):1–24, 2019.

[16] Avishkar Bhoopchand, Tim Rocktaschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *Arxiv*, 2016.

[17] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942, 2016.

[18] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. ISBN 9780387310732. URL http://www.worldcat.org/oclc/71008143.

[19] Paul Black. A software assurance reference dataset: Thousands of programs with known bugs. *Journal of Research of the National Institute of Standards and Technology*, 123, 04 2018. doi: 10.6028/jres.123.005.

[20] Gary Boetticher, Tim Menzies, and Thomas Ostrand. {PROMISE} repository of empirical software engineering data, 01 2007.

[21] Jón Arnar Briem, Jordi Smit, Hendrig Sellik, and Pavel Rapoport. Using distributed representation of code for bug detection. *CoRR*, abs/1911.12863, 2019. URL http://arxiv.org/abs/1911.12863.

[22] Jane Bromley, James Bentz, Leon Bottou, Isabelle Guyon, Yann Lecun, Cliff Moore, Eduard Sackinger, and Rookpak Shah. Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, 7:25, 08 1993. doi: 10.1142/S0218001493000339.

[23] Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, and Nikita Povarov. Using large-scale anomaly detection on code to improve kotlin compiler. *CoRR*, abs/2004.01618, 2020. URL https://arxiv.org/abs/2004.01618.

[24] Vanessa Buhrmester, David Münch, and Michael Arens. Analysis of explainers of black box deep neural networks for computer vision: A survey, 2019.

[25] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees, 2020.

[26] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud

Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[27] L. Büch and A. Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, Feb 2019. doi: 10.1109/SANER.2019.8668039.

[28] Cagatay Catal. A comparison of semi-supervised classification approaches for software defect prediction. *Journal of Intelligent Systems*, 23, 01 2014. doi: 10.1515/jisys-2013-0030.

[29] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, page 1–1, 2020. ISSN 2326-3881. doi: 10.1109/tse.2020.3020502. URL http://dx.doi.org/10.1109/TSE.2020.3020502.

[30] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, jun 2002. ISSN 1076-9757.

[31] Lin Chen, Bin Fang, Zhaowei Shang, and Yuanyan Tang. Negative samples reduction in cross-company software defects prediction. *Inf. Softw. Technol.*, 62:67–77, 2015.

[32] Jitender Chhabra and Varun Gupta. A survey of dynamic software metrics. *Journal of Computer Science and Technology*, 25:1016–1029, 09 2010. doi: 10.1007/s11390-010-9384-3.

[33] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 197–211, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0201554178. doi: 10.1145/117954.117970. URL https://doi-org.ezproxy.unibo.it/10.1145/117954.117970.

[34] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi: 10.1109/32.295895.

[35] Kyunghyun Cho, Bart van Merrienboer, Caglar Gucehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 2014, October 25-29, 2014, Doha, Qatar,A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734, 2014.

[36] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[37] David Coimbra, Sofia Reis, Rui Abreu, Corina S. Pasareanu, and Hakan Erdogmus. On using distributed representations of source code for the detection of C security vulnerabilities. *CoRR*, abs/2106.01367, 2021. URL https://arxiv.org/abs/2106.01367.

[38] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519, 2013. doi: 10.1109/ICSM.2013.85.

[39] CWE Community. CWE. https://cwe.mitre.org/.

[40] Qt Company. Qt. https://www.qt.io/.

[41] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. Deep learning on code with an unbounded vocabulary. EasyChair Preprint no. 466, EasyChair, 2018. doi: 10.29007/bc6w.

[42] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 46–57, 2019. doi: 10.1109/MSR.2019.00017.

[43] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, 2010. doi: 10.1109/MSR.2010.5463279.

[44] Al Danial. cloc. https://github.com/AlDanial/cloc, 2006. Retrieved 2021-01-13.

[45] Silvia N. das Dôres, Luciano Alves, Duncan D. Ruiz, and Rodrigo C. Barros. A meta-learning framework for algorithm recommendation in software fault prediction. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1486–1491, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337397. doi: 10.1145/2851613.2851788. URL https://doi.org/10.1145/2851613.2851788.

[46] Debian. Debian. https://www.debian.org/.

[47] Francesca Del Bonifro, Maurizio Gabbrielli, and Stefano Zacchiroli. Content-based textual file type detection at scale. In *ICMLC 2021: The 13th International Conference on Machine Learning and Computing*. ACM, 2021.

[48] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[50] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, 2017.

[51] Dario Di Nucci and Andrea De Lucia. The role of meta-learners in the adaptive selection of classifiers. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 7–12, 2018. doi: 10.1109/MALTESQUE.2018.8368452.

[52] Dario Di Nucci, Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. Dynamic selection of classifiers in bug prediction: An adaptive method. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1 (3):202–212, 2017. doi: 10.1109/TETCI.2017.2699224.

[53] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[54] Del Bonifro F, Gabbrielli M, Lategano A, and Zacchiroli S. Image-based many-language programming language identification. *PeerJ Computer Science*, 2021.

[55] Facebook. Infer, a tool to detect bugs in java and c/c++/objective-c code before it ships. https://fbinfer.com.

[56] Matloob Faseeha, Aftab Shabib, Ahmad Munir, Adnan Khan Muhammad, Fatima Areej, Iqbal Muhammad, Mohsen Alruwaili Wesam, and Sabri Elmitwally Nouh. Software defect prediction using supervised machine learning techniques: A systematic literature review. *Intelligent Automation & Soft Computing*, 29(2):403–421, 2021. ISSN 2326-005X. doi: 10.32604/iasc.2021.017562. URL http://www.techscience.com/iasc/v29n2/42941.

[57] Wang Fei, Ai Jun, and Xu Jiaxi. Software defect prediction based on graph representation learning. *IEEE Transactions on Software Engineering*, 2020.

[58] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[59] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, page 12–21, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365932. doi: 10.1145/3273934.3273936. URL https://doi.org/10.1145/3273934.3273936.

[60] Rudolf Ferenc, Dénes Bán, Tamás Grósz, and Tibor Gyimóthy. Deep learning in static, metric-based bug prediction. *Array*, 6:100021, 2020. ISSN 2590-0056. doi: https://doi.org/10.1016/j.array.2020.100021. URL https://www.sciencedirect.com/science/article/pii/S2590005620300060.

[61] Rudolf Ferenc, Péter Gyimesi, Gábor Gyimesi, Zoltán Tóth, and Tibor Gyimóthy. An automatically created novel bug dataset and its validation in bug prediction. *CoRR*, abs/2006.10158, 2020. URL https://arxiv.org/abs/2006.10158.

[62] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. *CoRR*, abs/1811.01824, 2018. URL http://arxiv.org/abs/1811.01824.

[63] Simran Fitzgerald, George Mathews, Colin Morris, and Oles Zhulyn. Using nlp techniques for file fragment classification. *Digital Investigation*, 9:S44–S49, 08 2012. doi: 10.1016/j.diin.2012.05.008.

[64] Kavita Ganesan and Romano Foti. C# or java? typescript or javascript? machine learning based classification of programming languages. GitHub, Inc. blog post: https://github.blog/2019-07-02-c-or-java-typescript-or-javascript-machine-learning-based-classification-of-programming-languages/, 2019. Retrieved 2020-01-06.

[65] Ben Gelman, Banjo Obayomi, Jessica Moore, and David Slater. Source code analysis dataset. *Data in Brief*, 27:104712, 2019. ISSN 2352-3409. doi: https://doi.org/10.1016/j.dib.2019.104712. URL https://www.sciencedirect.com/science/article/pii/S2352340919310674.

[66] Ben Gelman, Banjo Obayomi, Jessica Moore, and David Slater. Source code analysis dataset. *Data in Brief*, 27:104712, 10 2019. doi: 10.1016/j.dib.2019.104712.

[67] Shlok Gilda. Source code classification using neural networks. In *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6. IEEE, 2017.

[68] GitHub. Github. https://github.com/.

[69] GitHub, Inc. Linguist: Language savant. https://github.com/github/linguist, 2011. Retrieved 2020-01-06.

[70] Siddharth Gopal, Yiming Yang, Konstantin Salomatin, and Jaime G. Carbonell. Statistical learning for file-type identification. *2011 10th International Conference on Machine Learning and Applications and Workshops*, 1:68–73, 2011.

[71] Alicja Gosiewska and Przemyslaw Biecek. ibreakdown: Uncertainty of model explanations for non-additive predictive models, 03 2019.

[72] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Dino Pedreschi, and Fosca Giannotti. A survey of methods for explaining black box models, 2018.

[73] Novi Trisman Hadi and Siti Rochimah. Enhancing software defect prediction using principle component analysis and self-organizing map. In *2018 Electrical Power, Electronics, Communications, Controls and Informatics Seminar (EECCIS)*, pages 320–325, 2018. doi: 10.1109/EECCIS.2018.8692889.

[74] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol.*, 23(4), September 2014. ISSN 1049-331X. doi: 10.1145/2629648. URL https://doi-org.ezproxy.unibo.it/10.1145/2629648.

[75] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977. ISBN 0444002057.

[76] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017. URL http://arxiv.org/abs/1706.02216.

[77] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Sang Peter Chin. Learning to repair software vulnerabilities with generative adversarial networks. *CoRR*, abs/1805.07475, 2018. URL http://arxiv.org/abs/1805.07475.

[78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

[79] Software Heritage. Software heritage. https://www.softwareheritage.org/.

[80] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45, 2019. doi: 10.1109/MSR.2019.00016.

[81] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. Cc2vec: Distributed representations of code changes. *CoRR*, abs/2003.05620, 2020. URL https://arxiv.org/abs/2003.05620.

[82] Juntong Hong, Osamu Mizuno, and Masanari Kondo. An empirical study of source code detection using image classification. In *10th International Workshop on Empirical Software Engineering in Practice, IWESEP 2019, Tokyo, Japan, December 13-14, 2019*, pages 1–6. IEEE, 2019. doi: 10.1109/IWESEP49350.2019.00009. URL https://doi.org/10.1109/IWESEP49350.2019.00009.

[83] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL http://arxiv.org/abs/1704.04861.

[84] Shamsul Huda, Sultan Alyahya, Md Mohsin Ali, Shafiq Ahmad, Jemal Abawajy, Hmood Al-Dossari, and John Yearwood. A framework for software defect prediction and metric selection. *IEEE Access*, 6:2844–2858, 2018. doi: 10.1109/ACCESS.2017.2785445.

[85] Jack Humphreys and Hoa Khanh Dam. An explainable deep model for defect prediction. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 49–55, 2019. doi: 10.1109/RAISE.2019.00016.

[86] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2016.

[87] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. An empirical study of model-agnostic techniques for defect prediction models. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi: 10.1109/TSE.2020.2982385.

[88] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models, 2021.

[89] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 414–423, New York, NY, USA, 2014. Association for Computing

Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568320. URL https://doi.org/10.1145/2568225.2568320.

[90] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL https://doi-org.ezproxy.unibo.it/10.1145/2610384.2628055.

[91] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code, 2020.

[92] Jeremy Katz. Libraries.io open source repository and dependency metadata, December 2018. URL https://doi.org/10.5281/zenodo.2536573.

[93] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. Technical Report 1412.6980, arXiv, 2014.

[94] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL http://arxiv.org/abs/1609.02907.

[95] Elife Ozturk Kiyak, Ayse Betul Cengiz, Kökten Ulas Birant, and Derya Birant. Comparison of image-based and text-based source code classification using deep learning. *SN Comput. Sci.*, 1(5):266, 2020. doi: 10.1007/s42979-020-00281-1. URL https://doi.org/10.1007/s42979-020-00281-1.

[96] David Klein, Kyle Murray, and Simon Weber. Algorithmic programming language identification. Technical Report 1106.4064, arXiv, 2011. URL https://arxiv.org/abs/1106.4064.

[97] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573. IEEE, 2016.

[98] Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521874157, 9780521874151.

[99] Philipp Koehn, Franz J. Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 127–133, 2003. URL https://www.aclweb.org/anthology/N03-1017.

[100] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[101] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017. doi: 10.1145/3065386. URL http://doi.acm.org/10.1145/3065386.

[102] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1188–II–1196. JMLR.org, 2014. URL http://dl.acm.org/citation.cfm?id=3044805.3045025.

[103] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. *CoRR*, abs/1902.01954, 2019. URL http://arxiv.org/abs/1902.01954.

[104] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328, 2017. doi: 10.1109/QRS.2017.42.

[105] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328, 2017. doi: 10.1109/QRS.2017.42.

[106] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *IJCAI'18: 27th International Joint Conference on Artificial Intelligence*, 2018.

[107] Ning Li, Martin Shepperd, and Yuchen Guo. A systematic review of unsupervised learning techniques for software defect prediction. *Information and Software Technology*, 122:106287, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106287. URL https://www.sciencedirect.com/science/article/pii/S0950584920300379.

[108] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. Progress on approaches to software defect prediction. *IET Software*, 12, 02 2018. doi: 10.1049/iet-sen.2017.0148.

[109] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez, and Dawn Song. Neural code completion, 2017.

[110] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1–12, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00019. URL https://doi.org/10.1109/ICSE.2019.00019.

[111] LLVM. Clang. https://clang.llvm.org/, .

[112] LLVM. Python bindings for clang. https://github.com/llvm-mirror/clang/tree/master/bindings/python, .

[113] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. URL https://arxiv.org/abs/2102.04664.

[114] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions, 2017.

[115] Gerton Lunter, Alexei J. Drummond, István Miklós, and Jotun Hein. *Statistical Alignment: Recent Progress, New Applications, and Challenges*, pages 375–405. Springer New York, New York, NY, 2005. ISBN 978-0-387-27733-2. doi: 10.1007/0-387-27733-1_14. URL https://doi.org/10.1007/0-387-27733-1_14.

[116] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: an infrastructure for mining the universe of open source vcs data. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 143–154. IEEE Press, 2019.

[117] Oded Maimon and Lior Rokach, editors. *Data Mining and Knowledge Discovery Handbook, 2nd ed.* Springer, 2010. ISBN 978-0-387-09822-7. URL http://www.springerlink.com/content/978-0-387-09822-7.

[118] Katrina D Maxwell and Pekka Forselius. Benchmarking software development productivity. *IEEE Software*, 17(1):80–88, 2000.

[119] Katrina D Maxwell, Luk Van Wassenhove, and Soumitra Dutta. Software development productivity of european space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, 1996.

[120] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

[121] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 24(C):109–165, January 1989. ISSN 0079-7421. doi: 10.1016/S0079-7421(08)60536-8.

[122] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2018. doi: 10.1109/TSE.2017.2693980.

[123] Linda McIver. The effect of programming language on error rates of novice programmers. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*, page 15, 2000.

[124] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[125] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL http://arxiv.org/abs/1301.3781.

[126] Sandeep Muvva, A. Eashaan Rao, and Sridhar Chimalakonda. Bugl - A cross-language dataset for bug localization. *CoRR*, abs/2004.08846, 2020. URL https://arxiv.org/abs/2004.08846.

[127] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005. doi: 10.1109/ICSE.2005.1553571.

[128] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 364–373, 2007. doi: 10.1109/ESEM.2007.13.

[129] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE, 2015.

[130] A.T. Nguyen, Nguyen H.A., Nguyen T.T., and Nguyen T.N. Statistical learning approach for mining api usage mappings for code migration. In *ASE '14: 29th International Conference on Automated Software Engineering*, 2014.

[131] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595767. URL http://doi.acm.org/10.1145/1595696.1595767.

[132] NIST. Juliet test suite. https://samate.nist.gov/SRD/testsuite.php.

[133] Technion Israel Institute of Technology. code2vec. https://github.com/tech-srl/code2vec.

[134] Hector Olague, Letha Etzkorn, Sherri Messimer, and Harry Delugach. An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile,

software: A case study. *Journal of Software Maintenance*, 20:171–197, 05 2008. doi: 10.1002/smr.366.

[135] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, 33(6):402–419, 2007. doi: 10.1109/TSE.2007.1015.

[136] OpenStack. Openstack. https://www.openstack.org/.

[137] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015. URL http://arxiv.org/abs/1511.08458.

[138] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. A deep learning approach to identifying source code in images and video. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 376–386. ACM, 2018. doi: 10.1145/3196398.3196402. URL https://doi.org/10.1145/3196398.3196402.

[139] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. Learning lexical features of programming languages from imagery using convolutional neural networks. In Foutse Khomh, Chanchal K. Roy, and Janet Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 336–339. ACM, 2018. doi: 10.1145/3196321.3196359. URL https://doi.org/10.1145/3196321.3196359.

[140] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Big data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences*, 30(4):431–448, 2018. ISSN 1319-1578. doi: https://doi.org/10.1016/j.jksuci.2017.06.001. URL https://www.sciencedirect.com/science/article/pii/S1319157817300034.

[141] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duch-

esnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[142] Anh Viet Phan and Minh Le Nguyen. Convolutional neural networks on assembly code for predicting software defects. In *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*, pages 37–42, 2017. doi: 10.1109/IESYS.2017.8233558.

[143] Ramesh Ponnala and Dr Reddy. Software defect prediction using machine learning algorithms: Current state of the art. *Solid State Technology*, 64: 6541–6556, 05 2021.

[144] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Anamul Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. Too long; didn't watch!: extracting relevant fragments from software development video tutorials. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 261–272. ACM, 2016. doi: 10.1145/2884781.2884824. URL https://doi.org/10.1145/2884781.2884824.

[145] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction, 2021.

[146] Latifa Ben Arfa Rabai, Yan Zhi Bai, and Ali Mili. A quantitative model for software engineering trends. *Information Sciences*, 181(22):4993–5009, 2011.

[147] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2013.02.009. URL https://www.sciencedirect.com/science/article/pii/S0950584913000426.

[148] Dilini Rajapaksha, Christoph Bergmeir, and Wray Buntine. Lormika: Local rule-based model interpretability with k-optimal associations. *Information Sciences*, 540:221–241, 2020. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2020.05.126. URL https://www.sciencedirect.com/science/article/pii/S0020025520305521.

[149] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Communications of the ACM*, 60(10):91–100, 2017.

[150] Julio Reyes, Diego Ramìrez, and Julio Paciello. Automatic classification of source code archives by programming language: A deep learning approach. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 514–519, 2016. doi: 10.1109/CSCI.2016.0103.

[151] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-agnostic interpretability of machine learning, 2016.

[152] Daniel Rodríguez, MA Sicilia, E García, and Rachel Harrison. Empirical findings on team size and productivity in software development. *Journal of Systems and Software*, 85(3):562–570, 2012.

[153] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018. URL `http://arxiv.org/abs/1807.04320`.

[154] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018. URL `http://arxiv.org/abs/1807.04320`.

[155] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018. URL `http://arxiv.org/abs/1807.04320`.

[156] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[157] Karaivanov S., Raychev V., and Onward M. Vechev. Phrase-based statistical translation of programming languages. In *Onward! 2014: International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014.

[158] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 10–13, 2018.

[159] Jean E Sammet. Programming languages: history and future. *Communications of the ACM*, 15(7):601–610, 1972.

[160] Geanderson Santos, Eduardo Figueiredo, Adriano Veloso, Markos Viggiato, and Nivio Ziviani. Predicting software defects with explainable machine learning. In *19th Brazilian Symposium on Software Quality*, SBQS'20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450389235. doi: 10.1145/3439961.3439979. URL https://doi.org/10.1145/3439961.3439979.

[161] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005. URL http://promise.site.uottawa.ca/SERepository.

[162] Ke Shi, Yang Lu, Jingfei Chang, and Zhen Wei. Pathpair2vec: An ast path pair-based code representation method for defect prediction. *Journal of Computer Languages*, 59:100979, 2020. ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2020.100979. URL https://www.sciencedirect.com/science/article/pii/S2590118420300393.

[163] Ke Shi, Yang Lu, Guangliang Liu, Zhenchun Wei, and Jingfei Chang. Mpt-embedding: An unsupervised representation learning of code for software defect prediction. *Journal of Software: Evolution and Process*, 33(4):e2330, 2021. doi: https://doi.org/10.1002/smr.2330. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2330. e2330 smr.2330.

[164] Leonard J. Shustek. What should we collect to preserve the history of software? *IEEE Annals of the History of Computing*, 28(4):110–112, 2006. doi: 10.1109/MAHC.2006.78. URL https://doi.org/10.1109/MAHC.2006.78.

[165] Surendra K Singhi and Huan Liu. Feature subset selection bias for classification learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 849–856. ACM, 2006.

[166] Leslie N. Smith. Cyclical learning rates for training neural networks. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, 2017.

[167] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay, 2018.

[168] Y. Somda. GuessLang. https://guesslang.readthedocs.io/, 2017. Retrieved 2021-01-14.

[169] StackOverflow. StackOverflow. https://stackoverflow.com/.

[170] Mateusz Staniak and Przemysław Biecek. Explanations of model predictions with live and breakdown packages. *The R Journal*, 10 (2):395, 2019. ISSN 2073-4859. doi: 10.32614/rj-2018-072. URL http://dx.doi.org/10.32614/RJ-2018-072.

[171] Alexey Svyatkovskiy, Shengyu Fu, Ying Zhao, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *KDD '19: 25th International Conference on Knowledge Discovery & Data Mining*, 2019.

[172] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015. URL http://arxiv.org/abs/1503.00075.

[173] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 321–332, 2016. doi: 10.1145/2884781.2884857.

[174] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. Explainable ai for software engineering, 2020.

[175] Myo Wai Thant and Nyein Thwet Thwet Aung. Software defect prediction using hybrid approach. In *2019 International Conference on Advanced Information Technologies (ICAIT)*, pages 262–267, 2019. doi: 10.1109/AITC.2019.8921374.

[176] Z. Tóth, Péter Gyimesi, and R. Ferenc. A public bug database of github projects and its application in bug prediction. In *ICCSA*, 2016.

[177] Grigorios Tsoumakas, Ioannis Katakis, and I. Vlahavas. *Mining Multi-label Data*, pages 667–685. Springer, Boston, MA, 07 2010. doi: 10.1007/978-0-387-09823-4_34.

[178] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 25–36, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00021. URL `https://doi.org/10.1109/ICSE.2019.00021`.

[179] Nguyen Tung Thanh, Nguyen Anh Tuan, Nguyen Hoan Anh, and Nguyen Tien N. A statistical semantic language model for source code. In *ESEC/FSE '13: 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[180] Secil Ugurel, Robert Krovetz, and C Lee Giles. 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*, pages 632–638. ACM, 2002.

[181] Raoul-Gabriel Urma, Dominic A. Orchard, and Alan Mycroft, editors. *Proceedings of the 1st Workshop on Programming Language Evolution, PLE@ECOOP 2014, Uppsala, Sweden, July 28, 2014*, 2014. ACM.

[182] Raychev V., Vechev M., and Yahav E. Code completion with statistical language models. In *PLDI '14: 35th Conference on Programming Language Design and Implementation*, 2014.

[183] Juriaan Kennedy van Dam and Vadim Zaytsev. Software language identification with natural language classifiers. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 624–628. IEEE, 2016.

[184] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[185] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5866-pointer-networks.pdf`.

[186] Romi Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1, 05 2015.

[187] Hongyan Wan, Guoqing Wu, Ming Cheng, Qing Huang, Rui Wang, and Mengting Yuan. Software defect prediction using dictionary learning. In *SEKE*, 2017.

[188] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. *CoRR*, abs/1811.07234, 2018. URL http://arxiv.org/abs/1811.07234.

[189] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 297–308, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884804. URL https://doi.org/10.1145/2884781.2884804.

[190] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. Learning a static bug finder from data. *CoRR*, abs/1907.05579, 2019. URL http://arxiv.org/abs/1907.05579.

[191] David A. Wheeler. SLOCCount. https://dwheeler.com/sloccount/, 2001. Retrieved 2021-01-13.

[192] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1556–1560, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417943. URL https://doi.org/10.1145/3368089.3417943.

[193] Felix Wu, Tianyi Zhang, Amauri H. Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks. *CoRR*, abs/1902.07153, 2019. URL http://arxiv.org/abs/1902.07153.

[194] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect

prediction. *IEEE Transactions on Software Engineering*, 42(10):977–998, 2016. doi: 10.1109/TSE.2016.2543218.

[195] Jiaxi Xu, Fei Wang, and Jun Ai. Defect prediction with semantics and context features of codes based on graph representation learning. *IEEE Transactions on Reliability*, 70(2):613–625, 2021. doi: 10.1109/TR.2020.3040191.

[196] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018. URL http://arxiv.org/abs/1810.00826.

[197] Sihan Xu, Sen Zhang, Weijing Wang, Xinya Cao, Chenkai Guo, and Jing Xu. Method name suggestion with hierarchical attention networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, pages 10–21, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6226-9. doi: 10.1145/3294032.3294079. URL http://doi.acm.org/10.1145/3294032.3294079.

[198] Shir Yadid and Eran Yahav. Extracting code from programming tutorial videos. In Eelco Visser, Emerson R. Murphy-Hill, and Crista Lopes, editors, *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, pages 98–111. ACM, 2016. doi: 10.1145/2986012.2986021. URL https://doi.org/10.1145/2986012.2986021.

[199] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, 2015. doi: 10.1109/QRS.2015.14.

[200] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.03.007. URL https://www.sciencedirect.com/science/article/pii/S0950584917302501.

[201] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016.

[202] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

[203] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *ArXiv*, abs/1702.01923, 2017.

[204] Hiroshi Yonai, Yasuhiro Hayase, and Hiroyuki Kitagawa. Mercem: Method name recommendation based on call graph embedding. *CoRR*, abs/1907.05690, 2019. URL `http://arxiv.org/abs/1907.05690`.

[205] Liguo Yu and Alok Mishra. Experience in predicting fault-prone software modules using complexity metrics. *Quality Technology & Quantitative Management*, 9(4):421–434, 2012. doi: 10.1080/16843703.2012.11673302. URL `https://doi.org/10.1080/16843703.2012.11673302`.

[206] Jeffrey O. Zhang, Alexander Sax, Amir Roshan Zamir, Leonidas J. Guibas, and Jitendra Malik. Side-tuning: A baseline for network adaptation via additive side networks. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part III*, volume 12348 of *Lecture Notes in Computer Science*, pages 698–714. Springer, 2020. doi: 10.1007/978-3-030-58580-8_41. URL `https://doi.org/10.1007/978-3-030-58580-8_41`.

[207] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. Actionnet: vision-based workflow action recognition from programming screencasts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 350–361. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00049. URL `https://doi.org/10.1109/ICSE.2019.00049`.

[208] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2921–2929, 2016.

[209] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive

program semantics via graph neural networks. *CoRR*, abs/1909.03496, 2019. URL http://arxiv.org/abs/1909.03496.

[210] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019.

[211] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proc. IEEE*, 109(1):43–76, 2021. doi: 10.1109/JPROC.2020.3004555. URL https://doi.org/10.1109/JPROC.2020.3004555.

[212] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9, 2007. doi: 10.1109/PROMISE.2007.10.

[213] S.P Zingaro, G. Lisanti, and M. Gabbrielli. Multimodal side-tuning for document classification. In *Proceedings of the 25th International Conference on Pattern Recognition (ICPR)*, pages 5206–5213. IEEE, 2021.