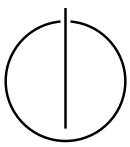
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Vectorizing Software for Machine Learning

Christian Feiler



DEPARTMENT OF INFORMATICS

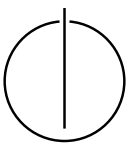
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Vectorizing Software for Machine Learning

Vektorisieren von Software für Maschinelles Lernen

Author:	Christian Feiler
Supervisor:	Prof. Dr. Alexander Pretschner
Advisors:	Markus Schnappinger, Dr. Arnaud Fietzke
Submission Date:	08.05.2019



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 08.05.2019 Christian Feiler

Acknowledgments

This thesis was created in cooperation with itestra GmbH. Insights into software health checks and hardware provided by itestra GmbH were highly appreciated and enabled the realization of this work. I would also like to thank the staff of the Chair of Software and Systems Engineering at the Technical University of Munich and especially Prof. Dr. Alexander Pretschner for their feedback and supply of additional hardware.

I am particularly grateful for the assistance given by my advisors Markus Schnappinger and Dr. Arnaud Fietzke. Their professional guidance ensured the outcome of this thesis. They constantly provided valuable advice and also data for the evaluation.

Abstract

Machine learning evolves in many areas as valuable data processing assistance. Recent work translates the application of machine learning to the field of software analysis in order to imitate the human comprehension of source code. A major challenge when learning from software is posed by the hierarchical nature of software that goes along with the variable size of source code. This hinders the application of arbitrary machine learning algorithms and causes the need for specifically designed and complex algorithms that lack flexibility.

In order to allow the flexible application of machine learning algorithms across application areas, this thesis presents a methodology for generating latent vectors for software elements. The vectors can then serve as input to arbitrary algorithms. The proposed methodology produces the vectors by transforming source code to graphs and gently aligning the graphs with software characteristics. Since supervised learning is employed for the alignment, the characteristics are provided by exemplary data in form of a labeled dataset that is created according to the application area. The resulting vector representations will embed the characteristics in a distributed manner.

The applicability of the proposed methodology was tested for the field of software health checks. By providing a dataset that captures the expertise of professional code reviewers, several Java classes which are part of the JSweet project and lack maintainability were successfully detected by the use of the generated, latent vectors.

Contents

Acknowledgments iii

Abstract iv

1. Introduction 1

2. Related Work 3

3. Research Question 5

4. Overview of the Approach 7
4.1. Source Code Levels . . . 7
4.2. Filtering of Source Code Levels . . . 8
4.3. Supervised Learning Setting . . . 9
4.4. Source Code as Machine Learning Input . . . 10
4.5. Model for Vectorization . . . 10
4.6. Procedure . . . 12

5. Transforming Source Code to Graphs 13
5.1. Method Level . . . 14
5.2. Class Level . . . 17
5.3. Package Level . . . 20
5.4. Project Level . . . 20

6. Machine Learning Model 22
6.1. Encoder – Graph Attention Network . . . 22
6.1.1. Initial Vertex Vectors . . . 24
6.1.2. Final Vertex Vectors . . . 25
6.1.3. Graph Vector . . . 30
6.2. Decoder . . . 31
6.2.1. Single Label Classification . . . 33
6.2.2. Multilabel Classification . . . 34
6.3. Hyperparameters . . . 34
6.3.1. Vocabulary Limitation . . . 35

6.3.2. Graph Size Limitation 36
6.3.3. Trainable Matrices Limitation 37

7. Implementation **39**
7.1. Graph Creation Program 39
7.1.1. Parsing Library 39
7.1.2. Structure 40
7.2. Vectorizing Model Program 41
7.2.1. Structure 42
7.2.2. Execution Modes 43

8. Application on Method Level **45**
8.1. Experiment – Semantic Vectorization 45
8.1.1. Label Creation 45
8.1.2. Setup 47
8.1.3. Results 50
8.1.4. Interpretation 51
8.2. Discussion 52
8.2.1. Practicality of the Experiment 52
8.2.2. Usage Example 53

9. Application on Class Level **57**
9.1. Experiment – Maintainability Forecasting 57
9.1.1. Label Creation 57
9.1.2. Setup 58
9.1.3. Results 60
9.2. Discussion 60

10. Future Work **63**

11. Conclusion **65**

A. List of Abbreviations **66**

B. OpenJDK 8 AST Nodes **67**

C. Method Level Model Configuration **69**

D. Method Similarity in Guava **70**

E. Class Level Model Configuration **75**

F. Findings in JSweet **76**

List of Figures **79**

List of Tables **81**

Bibliography **82**

1. Introduction

At itestra GmbH, software health checks are offered that "comprehensively check existing software systems".[1] They also assess the maintainability and overall software quality in order to "reveal cost drivers".[1] Consequently, they normally rely on heavy source code inspection as software systems are the basis of the checks. However, manual inspections are labor-intensive and hence protracted so that reliable assistance in form of tools could accelerate this process.

Existing tooling like ConQAT allows amongst other functionalities to compute software quality metrics [Dei+10] and is a basic help during health checks due to its inclusion of quality measurements: the attention of an IT consultant at itestra GmbH is drawn to weak spots in an existing software system. However, tooling like ConQAT does not exceed beyond the description as a basic help since measurements like quality metrics often produce false positives or false negatives when trying to draw the attention to relevant parts in a software system. This comes due to the fact that the experience and proficiency of an IT consultant can not be matched with tools that are unable to grasp the semantics in source code but rather rely on metrics.

As a result, tooling which combines the predictive approach of ConQAT and the knowledge of code reviewers can enrich the health check process by smartly highlighting weak or interesting spots of code bases that have not been seen by a consultant yet. Interesting spots during a software health check can be for example snippets that lack maintainability. Supervised machine learning addresses the issue of capturing experience since it tries to align observed properties to sample data [Bis06]. The result of this machine learning (ML) process will be an adapted ML model which implicitly represents the judgment and allows the prediction of properties for new data. Adopting this methodology to source code would allow the prediction of software characteristics: instead of any exemplary data, the supervised machine learning will receive source code as input. The source code will then be aligned to observed characteristics, e.g., the lack of maintainability. Consequently, the adapted model could be used for smartly highlighting spots in code bases.

However, traditional machine learning algorithms like support vector machines (SVMs) expect data of fixed length from which they then try to construct inferences [SS01]. In contrast, source code is normally of variable length and hence can not

[1] `https://itestra.com/en/leistungen/software-healthcheck/`

serve as input for any arbitrary supervised ML algorithm. Recent work by Allamanis et al. [ABK17] or Alon et al. [ALY18] tackles this limitation with specific ML models which work on source code. Their approaches are restricted to specific application areas though as they attempt to compute natural language (NL) sequences from code. These sequences can be descriptions of the code for instance. With their approaches, the flexibility of employing different supervised ML algorithms is neglected: apart from the prediction of natural language sequences, supervised ML algorithms could be deployed for other tasks as well, e.g., for the classification of source code. According to the applied algorithm, different facets of a consultant's experience might be captured. Consequently, employing diverse ML algorithms might result in a powerful tool when their results are combined.

In order to apply arbitrary machine learning algorithms and models on software, the software has to be transformed to a fixed length representation first. The resulting fixed length representation would be a vector that embeds information in a distributed manner: the meaning of the vector evolves through the combination of its dimensions so that infinitely many instances can be stored in a finite vector space [RM87]. The contribution of this thesis is the creation of such a representation for software. The vector representation will be solely based on source code so that other artifacts like binaries do not have to be provided. Since the vectors will be computed with supervised machine learning, the embedded information is gathered according to representative source code and depends on the intuition and experience of the code reviewer who delivered the data. Thus, the vectors will embed knowledge as it is acquired from previous software health checks and can overcome the limitations of current tooling like ConQAT: tools like ConQAT only rely on metrics and do not incorporate the experience of code reviewers in their analysis. The broad variety of ML algorithms allows further processing of the vectors so that the application in the software health check can be tuned gently. Since the exemplary data for the supervised learning does not necessarily have to be related to health checks, the outcome of this thesis can be applied for diverse source code analyses.

Chapter 2 will give an insight in recent approaches of how machine learning can be applied on source code. Followed by Chapter 3, the differing application area of our thesis, the contribution of this work and its novelty are described. Chapter 4 outlines the general approach in this thesis. The representation of source code for the machine learning approach is discussed in Chapter 5. The explanation of the approach is concluded with a detailed introduction of our machine learning model in Chapter 6. Implementational details of the approach can be found in Chapter 7. In order to assess the practicality of our methodology, Chapter 8 and Chapter 9 present experiments and discuss their results. The work finishes with motivations for future research in Chapter 10 and a conclusion in Chapter 11.

2. Related Work

There exists several recent work about machine learning on software where the respective publications differ in their application areas, their machine learning setting, the employed ML algorithms and their representation of source code for the algorithms.

For building a language model from source code, Dam et al. [DTP16] implement a deep learning model that reads source code as a sequence of words similar to natural language processing models. It is based on a long short-term memory (LSTM) proposed by Hochreiter et al. [HS97]. The deep learning model is evaluated on the ability to predict the next word for a preceding sequence of words within a code snippet. The application area differs from our motivation to that point that the methodology of Dam et al. does not attempt to summarize the information in a code snippet but rather tries to assist during the programming by incrementally suggesting the next token. Interpreting source code as a form of natural language is also implemented by Allamanis et al. [APS16]. They built a convolutional network which is enriched with an attention mechanism in order to actually summarize the knowledge within code snippets. The convolutional network is optimized towards sequence prediction and is evaluated on the prediction of Java method names.

Since source code is normally not directly interpreted as natural language but as a structured entity, other related work transforms source code to synthetic, structured objects. For generally learning from source code, Alon et al. [Alo+18a] propose a representation that is purely syntactic and extracted from the abstract syntax tree (AST). The result are paths in the AST. Even though this representation captures the hierarchical nature of source code, other information might have to be learnt by a model that uses the path-based mapping of the code. For instance, the cross-referencing of variables is not obvious in the AST and has to be identified by the model. An application of the path-based representation is given in a further publication by Alon et al. [Alo+18b] where the synthetic paths in the AST are processed by a neural network and weighted by an attention mechanism. The model is called *code2vec* and the evaluation is based on the ability to predict Java method names. The *code2seq* of Alon et al. [ALY18] proceeds and optimizes the idea of *code2vec*: it uses an LSTM instead of a simple neural network and aims for a sequential prediction of software characteristics.

In contrast to the path-based mapping, Henkel et al. [Hen+18] make use of traces collected from symbolic executions. These traces facilitate the extraction of the locality

and usual contexts of declarations. In contrast to most of the related work, this approach is employed in an unsupervised ML setting by applying the skip-gram model [Mik+13]. It is important to notice that the transformation of software to traces causes the machine learning input to be not close to the source code: the actual structuring of the code is not apparent anymore. Depending on the application area, this might be disadvantageous.

Similar to Henkel et al., Ben-Hun et al. [BJH18] apply the skip-gram model. They chose to use an intermediate representation (IR) of source code as input to the skip-gram model. With this approach, the usual contexts of instructions are identified. Thereby, instructions in source code can be assigned to semantic categories. The representation of software with an IR is not close to the original code either.

In order to automatically detect misused variables in Java, Allamanis et al. [ABK17] create graphs which serve as machine learning input. Such a graph is built on top of the AST: it is essentially an extension of the AST which contains additional edges for referencing where a variable was last used and for indicating the execution order of the statements. A Java type representation is incorporated as well. The final graph is then processed by a gated graph neural network (GG-NN) [Li+15].

3. Research Question

The goal of thesis is to embed latent information in source code in a vector of fixed length. This information can be any characteristics that a code reviewer can identify when inspecting source code, e.g., the semantics, the degree of maintainability or a combination of both. The resulting vectors have to allow further analyses with diverse machine learning algorithms. The intended application during software health checks makes this work unique. In order to achieve the above mentioned goal, the research questions (RQs) below arise. The approach of this thesis attempts to solve these questions in an optimal way in order to achieve the overall goal.

RQ1: Which algorithm is suited best for the computation of vectors from source code?

The developed algorithm has to cope with source code and output vectors for source code elements. Additionally, the algorithm has to allow the tuning of the vectors according to characteristics which might be relevant during a software health check. Due to the differing application areas, the algorithms in related work might not be able to capture the experience of a code reviewer. Consequently, we investigate to which extent the tuned vectors of our algorithm are meaningful for diverse application areas. The algorithm itself can be composed of several other algorithms. For instance, we analyze whether a preliminary transformation of source code – which is given as a set of NL sequences – to another representation enhances the later computation of health check related vectors.

RQ2: Which source code elements are suitable for an automated software analysis?

Different granularities of elements can be distinguished in source code. When analyzing code in an object oriented language, vectors could be created for methods, classes or other granularities. In contrast to most examples in related work, the outcome of this thesis has to be generally applicable and is not designed for solving a specific task like method name suggestion. Consequently, we investigate different granularities. According to the aforementioned motivation, the vectorization has to be done for source code elements which optimally benefit an analysis of software projects: a trade-off between level of detail and availability of contextual information has to be found.

RQ3: Which characteristics can be vectorized?

The aim of the thesis is to embed knowledge about source code in a latent vector. In order to ensure the general applicability, we analyze if and which software characteristics are obvious in the vectors and can be successfully identified by machine learning methods that are applied to the vectors.

4. Overview of the Approach

The goal of this thesis is to compute vectors from source code. These vectors are considered to be distributed representations of fixed length that capture the characteristics of the software. As highlighted in Figure 4.1, further machine learning models can be applied to the resulting vectors, e.g., unsupervised clustering algorithms. The intermediate model which establishes the distributed representation solves the problem that many models like decision trees can not cope with variable-length input but need a fixed-length input. With the help of the vectors, these models can be indirectly applied to software. We will refer to the intermediate model as *vectorizing model*. The following sections explain the scope of and the approach for the vectorizing model.

4.1. Source Code Levels

In order to create a distributed representation for source code in the form of vectors, the first step is to decide which kind of source code is considered. Since Java is one of most used and most popular programming languages[1] [2], our approach is based on software that is written in Java. Please note that only versions up to Java 8 are taken into account. The methodology of this thesis can however be easily extended in order to support other object-oriented programming languages.

For the purpose of vectorizing source code, one has to specify for which parts of the code vectors have to be created. With Java being the inspected programming language, different granularities of the source code are considered:

- Methods
- Classes
- Packages
- Projects

With the term *project* we refer to a set of Java packages where each of them either depends on another package or is needed by another one. We will refer to the above

[1] `https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages`
[2] `https://www.tiobe.com/tiobe-index/`

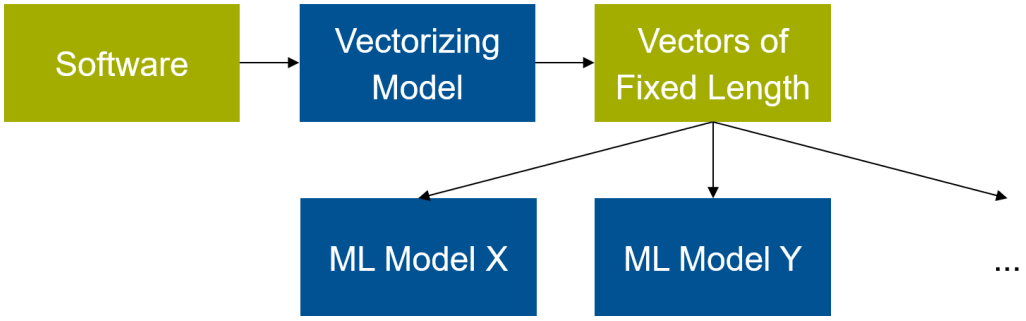


Figure 4.1.: The vectorizing model is the focus of this thesis. Machine learning models that are applied to the vectors are indirectly applied to the software.

mentioned granularities as *source code level*, too. Each of these levels can contain crucial information, e.g., about the maintainability of software. Therefore, vectors will be created for each of the above mentioned granularities. Other machine learning models can then be applied to the resulting vectors, e.g., clustering can be executed on the vectors of Java classes in order to group them by readability. The computed vectors are essentially a representation for entities in the source code levels. Even though variables that are declared in a class or a method body might yield additional information in some cases, this information might be not relevant without the class or method context in most cases so that we neglect the option to specifically learn vector representations for variables.

The vectors will be computed with the vectorizing model which is a machine learning model, i.e., an adaptive model that can be optimized [Bis06]. However, the information that can be retrieved from the source code levels differs for each level, e.g., the maintainability of a Java method can be heavily impacted by the statement nesting depth of the method whereas the maintainability of a Java package will be impacted more by coupling within the package than by the nesting depth. As a result, a single model will not be powerful enough to distinctively embed information for each of the source code levels. In order to overcome this issue, an individual machine learning model will be trained for methods, classes, packages and projects respectively. Therefore, each of the levels is investigated independently.

4.2. Filtering of Source Code Levels

For the method and class level, we only consider methods and classes that fulfill the conditions that are explained below. Some instances are filtered out since they do not

contain sufficient information in order to supply additional knowledge, e.g., during a health check.

For methods, this means first of all that they must not be abstract. The reason for this is that abstract methods do not contain a method body. Hence, they normally do not embed valuable latent information. Even though constructors do have a similar structure to methods, we introduce the condition that methods must not be constructors either. This is due to the fact that constructors are tightly coupled to their classes and appear out of context if they are inspected individually. Additionally, constructors usually deviate in their semantics and nature highly from other methods. Finally, method declarations that occur within another method declaration are not considered: these methods are already captured by their surrounding method. An example where such nested methods can occur in Java are anonymous classes which are defined in a method and contain other methods in turn.

Not all classes are taken into consideration either. For instance, interfaces are filtered out even though their structure resembles the structure of classes. This makes sense as interfaces normally do not contain an implementation similarly to abstract methods. In contrast, enums are regarded as classes and are part of the class level as they can contain method implementations in Java. Another constraint is that only top level classes are considered, i.e., nested classes are not processed: nested classes are captured by their surrounding instance which is normally another class.

4.3. Supervised Learning Setting

The goal of the vectorization is to store characteristics of the source code in a distributed representation. Characteristics can be for example the semantics or the comprehensibility of the source code. Since characteristics like quality attributes are often ambiguous and established by experience, the vectorizing model has to be provided with a labeled dataset, i.e., the vectors will be created with supervised learning. The labeled data contains examples for the properties which are supposed to be embedded in the distributed representation: if the vectors should be created according to the readability of methods, the dataset will contain methods for which a person assessed the readability. Consequently, the distributed representation will be biased towards the intuition and experience of the human that provided the data. With a set of examples provided, the vectorizing model will then be trained so that the resulting vectors reveal the characteristics in the examples in an optimal way.

4.4. Source Code as Machine Learning Input

In contrast to natural language, source code is not interpreted as a linear sequence of tokens. It is rather comprehended as a tree or a forest that is mapped from a sequence of tokens according to the syntax of the programming language. Additionally, the definitions in the code interact with and depend on each other. Consequently, programmers who attempt to identify properties in source code are aware of the Java syntax and other specific rules that make source code valid. Without the knowledge regarding the rules and the nature of the programming language, it is difficult to identify characteristics of the code. Therefore, it is advantageous to transfer the knowledge about specific programming rules to the machine learning model. Since it is not possible to store a set of rules in the machine learning model before it is trained, the rules have to be supplied differently. The best way to do so is to provide the data from which the model learns on a higher level of abstraction: methods, classes, packages and projects respectively are not given to the machine learning model as a token sequence like natural language but as structured objects. The structure then contains information that is obvious to programmers, e.g., the hierarchical nature of Java. In this thesis, we decided to use directed graphs as the representation of source code. The details of how the graphs are created from methods, classes, packages and projects respectively are explained in Chapter 5.

4.5. Model for Vectorization

As previously mentioned, individual machine learning models will be trained for the source code levels. This does not mean that for each level a new model is introduced, but it means that the same model will be trained separately for methods, classes, packages and projects respectively. The vectorizing model itself is loosely based on the encoder-decoder framework which enjoys great popularity in the field of neural machine translation [BCB14]. The encoder-decoder framework joins the encoder and the decoder by a hidden representation, i.e., a fixed-length vector [Cho+14]. The encoder maps the input sample to the hidden representation whereas the decoder attempts to predict the properties of the sample from the hidden representation. The encoder and decoder respectively can be individual neural networks.

We utilize the fixed-length vector in order to be the distributed representation for the input of the model as shown in Figure 4.2. This means that the whole model is trained by applying gradient based optimization as proposed by Kingma et al. [KB14] and that the decoding is left out for the prediction of the distributed representation. In our case, the sample that is forwarded to the model is a graph that represents an

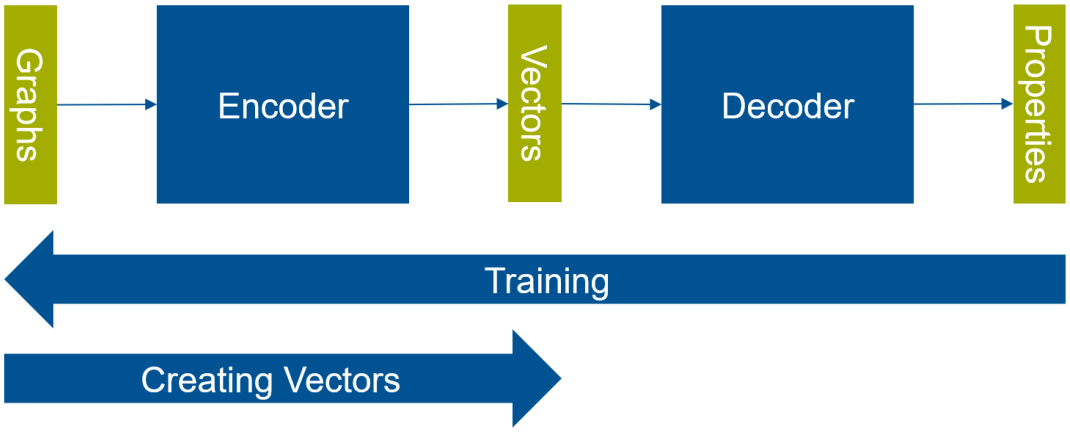


Figure 4.2.: The vectorizing model is based on the encoder-decoder framework that connects the encoder and the decoder by a fixed-length vector. During the training phase of the model, the encoder and decoder are optimized according to the given combinations of graphs and properties. For the vectorization of a graph, the decoder is omitted.

entity within a source code level. Traditionally, the encoder and decoder are recurrent neural networks (RNNs) that are neural networks which cope with sequences, but are typically unable to handle graphs [Jai+16]. As a result, the encoder will not be an RNN but a neural network that is able to handle graphs. The used neural network is inspired by the encoder in the Graph2Seq model [Xu+18] which employs attention mechanisms in order to distinguish crucial and less important information in the input graph. A RNN was neither employed as the decoder in this thesis: recurrent neural networks as decoder are used for the prediction of sequences. In our setting however, the software properties which should be encoded are normally available as an unordered set of categories or as a single category per sample. Additionally, the hidden representation that is optimized towards a RNN decoder does not allow a simple inference about the exact meaning of the representation since the RNN transforms the representation in a non-linear way for the prediction. An alternative to the RNN that we will employ is to use a single transformation matrix as the decoder. This allows deductions about the vector in between the encoder and decoder. As this vector is the focus of this work, the deductions might come in handy during subsequent analyses of the vector. Using a transformation matrix is inspired by latent factors models [BKV09]. More information about our machine learning model can be found in Chapter 6.

4.6. Procedure

The whole methodology of vectorizing parts of a software can be summarized as follows:

1. A source code level has to be chosen for which a distributed representation is needed. Further machine learning algorithms can be applied on the resulting vectors. In the subsequent enumeration we will assume that methods are the chosen source code level. For other levels, the following steps are valid as well.

2. A training dataset has to be created. The dataset contains methods that have properties assigned to them. These properties might consist of a single label or a set of labels, i.e., multiclass and multilabel classification are supported.[3]

3. The methods of the training set are transformed to directed graphs. The directed graphs serve as structured objects that implicitly store information about the characteristics of the source code. Details about this are available in Chapter 5.

4. The vectorizing model is trained with a set of tuples where each tuple consists of a graph and the corresponding method properties. Chapter 6 depicts the specifics of the model.

5. As shown in Figure 4.2, vectors for new method graphs can be computed with the optimized model.

[3] `https://scikit-learn.org/stable/modules/multiclass.html`

5. Transforming Source Code to Graphs

The nature of the input for a machine learning model can heavily influence its performance [KKP07]. If correlations between the input and the expected labels are hard to detect, the model will perform badly. Therefore, the transformation of Java source code before the model is applied has to be chosen well so that meaningful vectors will be produced.

Previous work by Allamanis et al. [ABK17] takes the AST as basis for a machine learning model: the AST of code snippets is transformed to a directed graph which is extended with additional edges. These edges reference where a variable was last used and indicate the execution order of the statement in the code. A Java type representation is incorporated in the graphs as well. These extensions of the AST can enhance the machine learning performance, e.g., with respect to the capturing of software maintainability. Since this graph representation is based on the AST, it is close to the actual source code which is also important when gathering knowledge that should be helpful during a software health check. Therefore, our representation is inspired by this approach. Due to the differing areas of application, the methodology of Allamanis et al. [ABK17] also embeds knowledge that is not needed in our setting. This results in graphs of bigger size which in return would affect the scalability of the vectorizing model. Consequently, we will not employ the graph representation proposed by Allamanis et al. but create graphs based on their approach.

As a summary of the above, a good choice for the source code representation should be close to the original code and contain information that is obvious to programmers but not to a person that has no experience with the particular programming language. This means that as much knowledge as possible about the meaning of the source code should be included in the input of the machine learning model. Despite that, the representation should be of reasonable size so that the scalability of the machine learning algorithm is not jeopardized.

In this chapter, we propose a transformation of the source code that fulfills the aforesaid. The resulting representation will be a directed graph. The following sections discuss the transformation for each of the source code levels.

```
public String setAndGetProperty(String newValue) {
  classProperty = newValue;
  return classProperty;
}
```

Figure 5.1.: An exemplary method.

5.1. Method Level

Methods form the actual behavior of programs, e.g., handling communications or data manipulation. Therefore, they build an essential foundation for analyzing software. Due to their nature, a purely hierarchical representation of methods might produce poor results when machine learning is applied to it: the control and data flow should be encoded as well. The basis for our representation of methods is the AST which is a purely hierarchical structure. It will be extended with nodes and edges in order to capture the additional aspects. Each node in the resulting graph can have multiple labels – also referred to as attributes – and each edge will have exactly one label. We will distinguish the two steps *transformation step* and *extension step*.

The transformation step converts the AST to a directed graph which is done by directing the edges in the tree towards the parent node. These edges are labeled as *PARENT*. Figure 5.1 gives a simple example of a Java method. The result of the transformation step for it is shown in Figure 5.2. The graph after the transformation step has some peculiarities:

- Since the syntax tree is extracted from OpenJDK 8 internals, the structure of the graph corresponds to the internal representation of source code in the Java development kit. Other parsers like JavaParser[1] would produce a different tree. The extension step can be adapted for them though.
- The labels of the nodes are also created from the internals of the java development kit (JDK). The AST in the JDK is represented by a nested structure of Java classes. As a result, each node in our graph contains exactly one label which is the name of the Java class that captures the subtree attached to it.[2] An exception is the labeling of nodes that contain *JCModifiers*: these nodes can contain additional attributes as the actual modifiers are appended. An example is the blue node in Figure 5.2 that has two labels: *JCModifiers* and *PUBLIC*.

[1] `http://javaparser.org/`
[2] The prefix *JC* in the labels refers to *javac*.

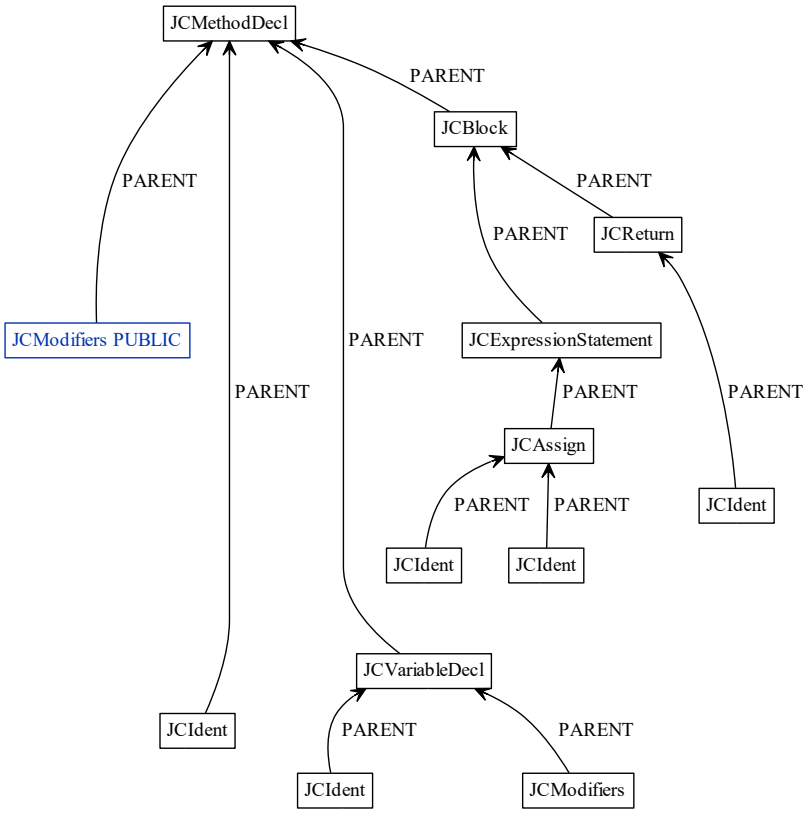


Figure 5.2.: The AST is the basis for the method graph. In this figure, the AST of the method in Figure 5.1 is illustrated as a directed graph.

All node types that can occur in the directed graph after the transformation step are listed in Appendix B.

After generating a directed graph, additional nodes and edges are inserted in the extension step. The result for the method in Figure 5.1 is illustrated in Figure 5.3 where the additional nodes and edges are highlighted in color. First of all, the execution order of statements is indicated by adding links labeled as *NEXT*. These links interconnect pairs of consecutive children of a *JCBlock*. These children always represent Java statements and are ordered according to their occurrence in the source code. In Figure 5.3, only one such link exists and is colored orange: it indicates that *JCExpressionStatement* is executed before *JCReturn*. The *NEXT* edges are necessary due the fact that there is no order for adjacent vertices in a purely mathematical graph so that the succession of Java statements is not obvious anymore.

For comprehending source code, a lot of information is contained in the naming

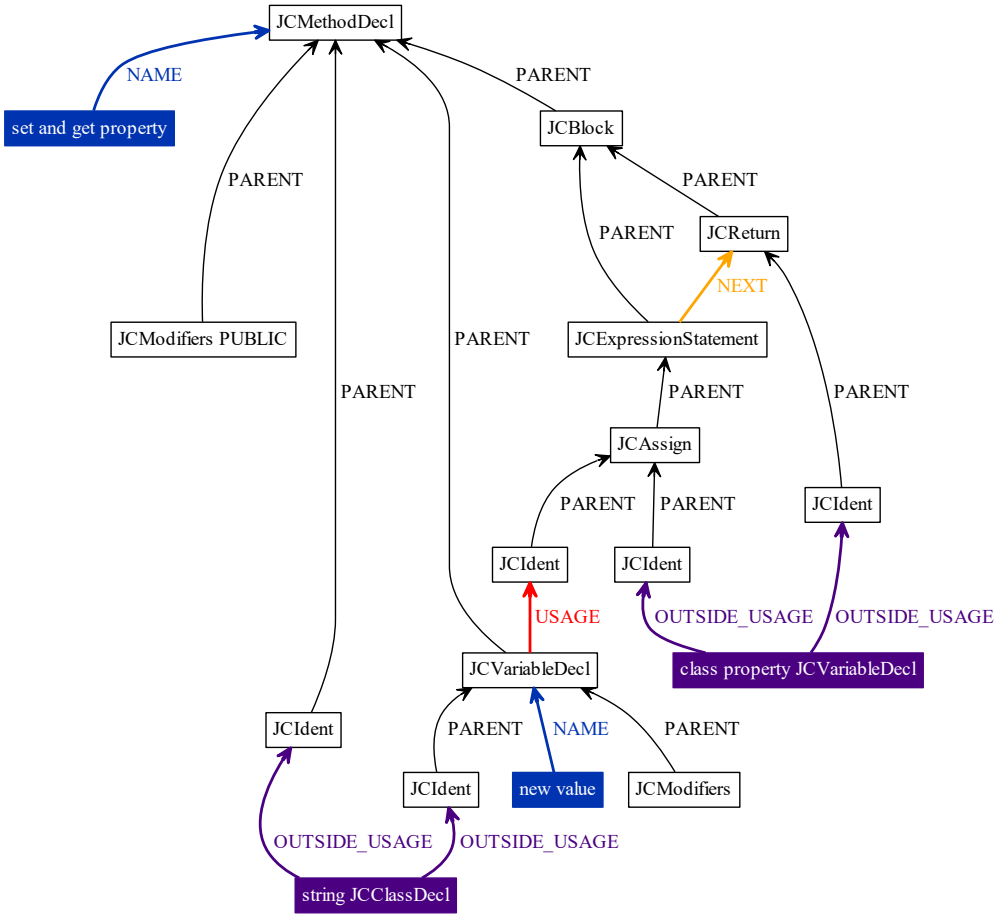


Figure 5.3.: The final graph for the method in Figure 5.1. Nodes and edges that were added to the graph in Figure 5.2 are highlighted in color.

of identifiers [DP05]. In the AST however, there are no names included as visualized in Figure 5.2. Hence, the naming of declarations is added to the graphs in form of a separate node that is attached to the declaration with a *NAME* edge. These amendments are colored blue in Figure 5.3. The new nodes can have multiple attributes which are the words that arise from splitting the identifier names according to camel case and snake case. The resulting words are generally transformed to lower case. As an example, *setAndGetProperty* in Figure 5.1 is split into *set*, *and*, *get* and *property* in Figure 5.3. Please note that there are only three different types of declarations in the AST: *JCClassDecl*, *JCMethodDecl* and *JCVariableDecl*. Therefore, only these three nodes can have names attached to them.

The data flow within a program reveals a lot of information about its semantics. The flow can be traced by inspecting the reuse of declarations in the source code, more specifically in the methods. The following node types indicate a reuse:

- *JCIdent*
- *JCFieldAccess*
- *JCMemberReference*
- *JCNewClass*

We will refer to these vertices as referencing nodes. In order to help the machine learning model understanding the semantics of software, we add links from declarations towards referencing nodes so that their nature and origin become apparent and the data manipulation in a program gets obvious. Since not every declaration is made in the AST of the method as the variable *classProperty* in Figure 5.1 proves, we differentiate between two different edges towards referencing nodes: *USAGE* and *OUTSIDE_USAGE*. If the declaration from which the edge originates happens in a method, we will add the *USAGE* link towards the referencing node as the declaration is already available in the AST. An example is the reuse of the method parameter in Figure 5.3 which is highlighted in red. Otherwise, we will use *OUTSIDE_USAGE*. However, the linked declaration has to be made available first. This is done by adding a new vertex to the graph for the declaration. The attributes of the vertex consist of the declaration type and the tokens that result from splitting the name of the declaration according to camel and snake case. The declaration type is either *JCClassDecl* or *JCMethodDecl* or *JCVariableDecl*. The *OUTSIDE_USAGE* edges will then be injected with the newly created vertex as source and each of the corresponding, referencing nodes as a destination. The thereby injected vertices and edges are colored purple in Figure 5.3.

No further additions are made in the extension step. Advanced modifications could be realized, of course. For instance, the final graph in Figure 5.3 does not allow inferences about which identifier is assigned to which one at the *JCAssign* node – even though the code in Figure 5.1 makes clear that *newValue* is assigned to *classProperty*. As a result, our graphs are not isomorphic to the interpreted source code of a method. Nevertheless, the aforesaid alternations of the AST build a reasonable trade-off between simplicity and availability of information.

5.2. Class Level

For building graphs from Java classes, we distinguish two approaches: the non-truncated and the truncated graph.

```
public class ExampleClass {
  private String someString;

  ExampleClass() {
    super();
    someString = "asdf";
  }

  public String getSomeString() {
    return someString;
  }
}
```

Figure 5.4.: An exemplary class.

The non-truncated graph results from applying the transformation and extension step of Section 5.1 onto the AST of a Java class:

1. At first, a directed graph is established from the AST of the class as mentioned in Section 5.1. The root node is then *JCClassDecl* instead of *JCMethodDecl*.

2. The *NEXT* edges will be inserted afterwards to highlight the execution order of statements in methods that are defined in the class.

3. The names of all declarations in the class are attached to the graph as additional vertices. The *NAME* links are created towards the corresponding declarations.

4. The data flow is indicated by *USAGE* and *OUTSIDE_USAGE*. If the declaration for a referencing node is made within the class, the *USAGE* linkage is employed. For declarations that happen outside of the class, a new vertex will be created in the same way as mentioned in Section 5.1. *OUTSIDE_USAGE* will then interconnect the new vertices and the referencing nodes.

The disadvantage of this approach is the size of the resulting graph: it can consist of multiple thousands of nodes. Graphs of this dimension will heavily slow down the machine learning algorithm. For instance, the non-truncated graph of the class in Figure 5.4 contains 27 nodes – even though the class is minimalist. Furthermore, the outcome of the machine learning model can be worse for huge inputs since it is harder for the model to identify important parts in the graph. Nevertheless, the non-truncated

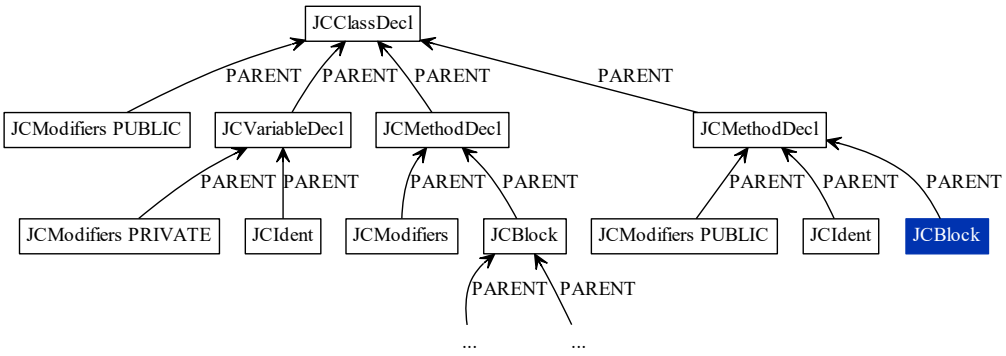


Figure 5.5.: The figure shows the truncated graph after the transformation step for the class in Figure 5.4. The children of the blue *JCBlock* were cut away. Some other subgraphs were omitted due to their irrelevance. The omittance is indicated by ellipses. As constructors are also referred to as methods in the OpenJDK, the leftmost *JCMethodDecl* node is the constructor of the class.

graphs make sense if the method level is not analyzed separately so that all available information in the class should influence the vectors that are computed by the model.

As the denomination suggests, the truncated graphs are smaller than non-truncated graphs. However, they are also based on the transformation and extension step of Section 5.1. The main difference compared to the non-truncated graph is that the truncated graph is cropped after the transformation step: the subgraphs that represent method bodies are cut away. Please note that constructor bodies are not cut off but remain part of the graph. An example Java class is shown in Figure 5.4. For this class, Figure 5.5 highlights the node that is affected by the truncation. The extension step for the residual graph is conducted as follows:

1. The *NEXT* links will be placed in the graph in order to connect statements. Since all method bodies were cut off, only the constructor bodies have to be investigated therefor.

2. The names of the remaining declarations in the graph will be attached as described in Section 5.1.

3. The data manipulation in the leftover graph is also marked by *USAGE* and *OUTSIDE_USAGE* edges. If a declaration that was not made in a method body is reused in a method body, the residual *JCBlock* node will be the target of the *USAGE* and *OUTSIDE_USAGE* edges since the content of the block is not available

anymore. Hence, the data usage in method bodies is captured even though the bodies do not exist.

The cropping of method bodies can reduce the size of the class graphs tremendously. The training time of the machine learning model decreases accordingly. Admittedly, the nature of the methods and the procedures described by a class are no longer captured. Thus, the truncated graphs should only be employed if the data passing on the class level is important and the exact process of data mutation can be neglected. This is normally the case when the method level is examined by a separate vectorizing model.

5.3. Package Level

Due to the fact that we did not have a dataset available for evaluating the package level we did not implement the transformation of Java packages to graphs. Hence, we will illustrate some factors that should be considered for this transformation. However, we will not present a definite approach.

The above mentioned trade-off between size and information content of the graphs amplifies for Java packages. Embedding explicit information about method behavior in the package graph goes beyond the limits of magnitude which the machine learning model can handle. Therefore, it is not possible to encode every piece of knowledge within a Java package. Instead, the graph has to be focused on the data that is relevant for the task which the machine learning model is used for.

Generally, a package in Java is a set of classes, enums, interfaces and nested packages. There can be several motivations for analyzing a structure like this – especially during a software health check. For instance, the structural organization and the dependencies within the package might reveal misplaced classes so that the package should be inspected manually during a review. These two aspects could be easily encapsulated in a directed graph. A more complicated aspect is the semantics of the package as it requires a deeper inspection of the contained classes. The semantics allows indeed a more sophisticated evaluation of misplaced contents in the package. An approach to embed semantics in the graph could be to integrate distributed representations of methods and classes. These vectors can be computed in advance, e.g., by deploying the methodology for vectorizing methods and classes as it is proposed in this thesis.

5.4. Project Level

Similarly to the package level, we did not implement the project graphs for the same reason. We will again explain some thoughts about these graphs.

As mentioned in Chapter 4, we refer to a project as a set of Java packages where each of these packages either depends on or is needed by at least one other package. Knowledge that can be helpful for software health checks at this level includes the architecture within the project where each package or subpackage depicts a component in the software system. Therefore, the relation between the packages and subpackages should be extractable from the project graphs. As a result, the coupling in the system is contained in the graphs as well.

Similarly to the package graphs, semantics could be included by attaching precomputed vectors of classes to the graph. A possible realization is to average the calculated vectors of classes for each package or subpackage and attach the average as a special node to the graph. Semantics of methods can be included in a likewise manner. This would allow the machine learning model to gain deeper knowledge about actual data transfer and manipulation in the project.

6. Machine Learning Model

Our vectorizing model which is a machine learning model transforms the graphs from Chapter 5 to a distributed representation according to characteristics of the source code. Its trainable variables are learnt in multiple epochs with the optimizer proposed by Kingma et al. [KB14]. For the optimization, we assume that a training dataset, which consists of graphs and their corresponding labels, is given.

The model itself is based on the encoder-decoder framework where the encoder and the decoder are only coupled by a hidden vector. Since both of them are optimized together for an ideal prediction of the properties, the vector contains in the end all the knowledge that is needed for the decoder in order to predict the attributes. Hence, the hidden representation of the graphs will encode their properties in an optimal way. Due to the fact that the goal of this thesis is to vectorize source code, the hidden representation is the actual focus as shown in Figure 4.2. The decoder is only utilized in order to impart meaning in the distributed representation.

The encoder and decoder which together build the vectorizing model will be discussed in the following sections.

6.1. Encoder – Graph Attention Network

The encoder is a neural network that is employed in order to grasp the information which is stored in a graph and embed it in a vector. This neural network is loosely based on the Graph2Seq model of Xu et al. [Xu+18]. The Graph2Seq model is essentially an adoption of GG-NNs as introduced by Li et al. [Li+15]. The adoption is also extended by an attention mechanism for better managing sequence prediction. The purpose of such an attention mechanism is to search for important variables by aligning them with a reference [BCB14]. The idea behind it can be compared with a human intuitively attending over items: a human will pay more attention to items which seem special or relevant for solving a specific task. Since our encoder copes with graphs and also applies attention mechanisms, we will refer to the encoder as *graph attention network*, too. Similarities to the Graph2Seq model will be mentioned later.

Before the specifics of the vectorizing model and its graph attention network are illustrated, we create a custom formalization for the graphs of Chapter 5 as the graph attention network retrieves them as input. At first, we define the vocabularies X and Y:

- $X = \{x_0, x_1, \dots\}$ is the set of all node labels. X will also be referred to as *vertex vocabulary* or *node vocabulary*. The vertex vocabulary comprises all labels that can possibly be attached to nodes in any graph. Theoretically, $|X| = \infty$ can hold true. In practice, the size of the vocabulary will be limited. More information is given later in Section 6.3.
- $Y = \{y_0, y_1, \dots\}$ contains all edge labels. Y is called *edge vocabulary*, too. In contrast to the vertex vocabulary, the size of Y is limited since we are only working with method or class graphs. Therefore, the following will hold true:

$$Y \subseteq \{PARENT, NEXT, NAME, USAGE, OUTSIDE\_USAGE\}$$

A method or class graph G is defined as a tuple (V, E):

- V is the set of vertices in the graph.
- E consists of the directed edges that interconnect the nodes in V.

For capturing the interrelations in the graph, we define the functions

$$s : E \to V \text{ and } d : E \to V$$

where $s(e)$ specifies the source node of an edge $e \in E$ and $d(e)$ its destination node respectively. We also introduce the terms *forward connectivity* and *backward connectivity*. The forward connectivity of a vertex $v \in V$ is the set of links $F_v \subseteq E$ that contains all edges that originate at vertex v:

$$F_v = \{e \in E | v = s(e)\}$$

We define the backward connectivity $B_v \subseteq E$ of a vertex $v \in V$ correspondingly:

$$B_v = \{e \in E | v = d(e)\}$$

The labeling of nodes is captured by the function

$$\mathcal{L} : V \to \mathcal{P}(X)$$

which returns the set of attributes for a given vertex. In contrast, edges have exactly one attribute:

$$l : E \to Y$$

The graph attention network will then transform the formal graph G to a hidden representation $h_G \in \mathbb{R}^{n_{graph}}$ which serves as input for the decoder. The dimension n_{graph} is a hyperparameter and therefore fixed in advance. The methodology of the graph

attention network consists of three major steps which are inspired by the steps in the Graph2Seq model by Xu et al. [Xu+18]. However, the formulas and mathematical processes in the graph attention network differ highly from those in the Graph2Seq model: the computations were adapted to the graphs of Chapter 5. The three major steps will be explained in detail in the subsequent subsections and can be summarized as follows:

1. For each node $v \in V$, we learn a hidden representation $h_v^{(0)} \in \mathbb{R}^{n_{graph}}$ that captures the information about the node itself, i.e., $h_v^{(0)}$ will be a latent vector that encodes the vertex attributes.

2. The vector $h_v^{(0)}$ will be updated in an iterative manner by incorporating information about the neighborhood. The result after a fixed amount of k iterations is $h_v^{(k)}$. Each $h_v^{(k)}$ contains knowledge about the node v and its locality. An attention mechanism will be employed in order to filter seemingly more relevant parts of the neighborhood.

3. In the final step, a weighted sum will be computed from the node vectors:

$$h_G = \sum_{v \in V} w_v \times h_v^{(k)}$$

   The weights $w_v \in \mathbb{R}$ are determined by another attention mechanism: this attention mechanism compares the node vectors and decides which nodes should have the most influence on the graph vector h_G.

Consequently, the distributed representation h_G of a graph G embeds information about the vertices that seem most important – i.e., they allow a good inference of the graph labels – and their neighborhood. By translating this methodology to Java method or class graphs, it becomes obvious that the graph attention network encodes the context of the most relevant, atomic source code components for determining properties in the software. Each of the three aforementioned actions as well as the attention mechanisms will be illustrated in detail in the following subsections.

6.1.1. Initial Vertex Vectors

The first step is to assign a vector to each node that is based on the labels of the node. This is done with the help of the embeddings matrix $U_X \in \mathbb{R}^{n_{graph} \times |X|}$. For each word $x_i \in X$, the embeddings matrix contains the column vector $U_{X,i} \in \mathbb{R}$ that represents the word with a meaningful vector. U_X can be interpreted as a lookup table for the vertex vocabulary. The embeddings matrix will be optimized during the training process of

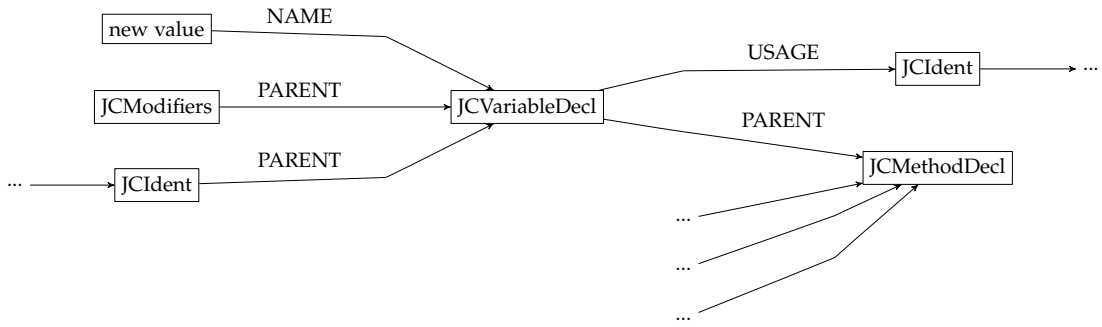


Figure 6.1.: A part of the method graph of Figure 5.3 is shown. Omitted subgraphs are indicated with ellipses.

the vectorizing model so that the vocabulary embeddings do not have to be known beforehand. The initial vertex vector $h_v^{(0)}$ for a node $v \in V$ is then the sum over its label embeddings:

$$h_v^{(0)} = \sum_{x_i \in \mathcal{L}(v)} U_{X,i}$$

With the graph of Figure 6.1 as basis, Figure 6.2 illustrates the assignment of the initial vectors for the nodes after their labels have been looked up.

6.1.2. Final Vertex Vectors

A major information in a graph is the connectivity between the different types of nodes. Hence, the initial vertex vectors have to be updated with knowledge in their neighborhood so that the encoder can understand the graph. This is done with an iterative approach.

Prior to that, edges have to be assigned a latent representation similarly to the initial vectors of nodes. Therefore, we define an embeddings matrix $U_Y \in \mathbb{R}^{n_{edge} \times |Y|}$ for the edges: each edge label $y_i \in Y$ is mapped to a latent vector with its corresponding column $U_{Y,i}$ in the embeddings matrix. U_Y is also learnt during the model optimization. The dimension n_{edge} is a hyperparameter like n_{graph}. However, n_{edge} and n_{graph} can differ: the latent representation of edges can have a different dimension than the representation of a graph or a vertex. Since there are way fewer different edge types than node types in a method or class graph, a lower value for n_{edge} is sufficient for embedding the information of an edge. The hidden vector h_e for a link $e \in E$ is computed by the following equation:

$$h_e = U_{Y,i} \quad \text{with } y_i = l(e)$$

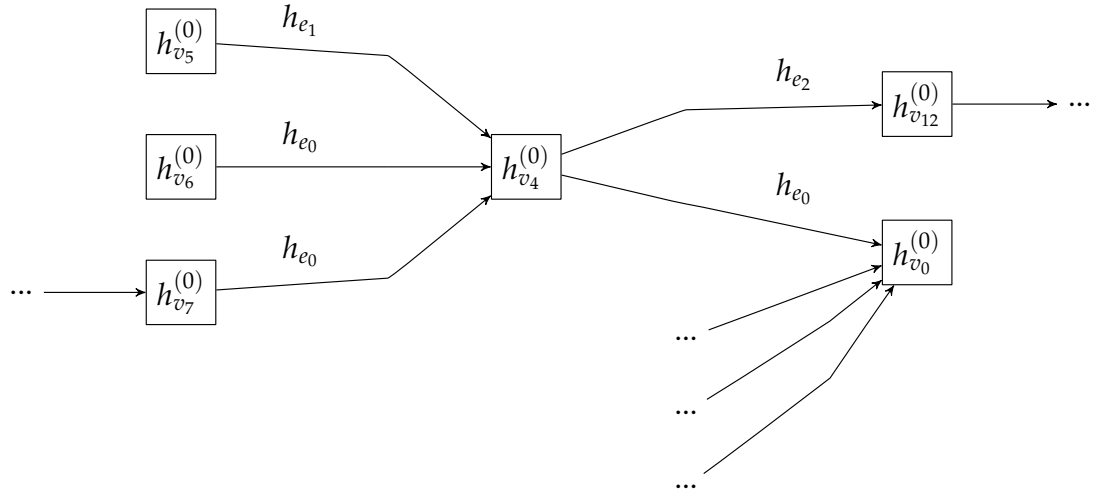


Figure 6.2.: The initial hidden representations were assigned to the nodes of Figure 6.1. The edges have a latent vector as well. The indices enhance the distinction of different values but have no deeper meaning.

Figure 6.2 shows the latent vectors of the edges, too.

Each iteration of the node representation update attempts to incorporate the information of adjacent neighbors in the current vertex. The updating is repeated k times with k being a hyperparameter. Although it is done for all vertices simultaneously, we will illustrate the update by means of a single vertex $v \in V$.

Forward Neighborhood

At first, the forward neighborhood of v is inspected in an iteration $j \in \{1, \ldots, k\}$. Each neighbor and the link towards it respectively contain a hidden representation. These vectors will be joined to a single vector with the help of a fully-connected layer which is unbiased, i.e., the layer consists of a multiplication only and no constant vector is added:

$$comb_{for}^{(j)}(e) = \sigma(W_{for,j} \times [h_e, h_{d(e)}^{(j-1)}]) \tag{6.1}$$

The function $comb_{for}^{(j)}$ represents the fully-connected layer. It returns the joined vector for an edge and the destination vertex of the edge. There are new definitions and notations introduced in Equation 6.1:

- $[a, b, \ldots]$ indicates the concatenation of vectors.

- σ is an activation function. We will use a leaky rectified linear unit (ReLU) with $\alpha = 0.2$ for this purpose due to its enhanced convergence behavior [Xu+15]. The leaky ReLU is defined as follows:

$$\sigma(x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}$$

- $W_{for,j} \in \mathbb{R}^{n_{graph} \times (n_{graph} + n_{edge})}$ is a weight matrix[1] that is optimized during the training of the vectorizing model. The index j suggests that there can be an individual weight matrix for each iteration j. More information about the flexible amount of weight matrices will be given in Section 6.3.

- The index for refers to *forward* and is used in order to distinguish forward neighborhood variables from the ones of the backward neighborhood aggregation which is highlighted later.

Since not each edge in the forward connectivity might deliver equally relevant knowledge to v, we assign weights to the forward edges with the function $weight_{for}^{(j)}$:

$$weight_{for}^{(j)}(v, e) = \left\langle h_v^{(j-1)}, A_{for,j} \times comb_{for}^{(j)}(e) \right\rangle \tag{6.2}$$

The function takes the result of the layer in Equation 6.1, transforms it linearly by multiplying it with $A_{for,j}$ and compares it to the latest hidden representation of a vertex by calculating the dot product. The matrix $A_{for,j} \in \mathbb{R}^{n_{graph} \times n_{graph}}$ is also a weight matrix which is optimized by the model. The index j indicates yet again that there can be an individual matrix for each iteration j.

Since the dot product computes the cosine similarity, the information $comb_{for}^{(j)}(e)$ gathered from a forward connection $e \in E$ retrieves a greater weight in Equation 6.2 if it is more similar to the latest representation $h_v^{(j-1)}$ of the current vertex $v \in V$. Informally expressed, forward connections which are more similar to the current vertex are considered to be more relevant. In order to allow the vectorizing model more flexibility during the similarity measurement, the linear transformation with $A_{for,j}$ is introduced which can alter the result of Equation 6.1 before the similarity is calculated.

Equation 6.2 implements the general attention mechanism which was proposed by Luong et al. [LPM15]. As mentioned above, the purpose of an attention mechanism is to search for important variables by aligning them with a reference [BCB14]. In this

[1] The terms *weights* and *weight matrix* have different meanings in this thesis. A *weight* is a scalar that indicates the relevancy of a variable, e.g., in a weighted sum whereas a *weight matrix* is trainable and used as a transformation matrix, e.g., in a layer of a neural network.

case, the variables are the forward connections of v whereas the reference is the latest vector representation of v. Please note that the attention mechanism in Equation 6.2 is employed independently of the one that is explained later in Subsection 6.1.3 and that calculates the importance of the nodes in order to create a graph vector.

The weights returned by $weight_{for}^{(j)}$ are then scaled to values in $[0, 1]$ with the softmax function:

$$att_{for}^{(j)}(v, e) = \frac{exp(weight_{for}^{(j)}(v, e))}{\sum_{e_i \in F_v} exp(weight_{for}^{(j)}(v, e_i))}$$

The result of the function $att_{for}^{(j)}$ indicates the importance of a node as it is determined by the general attention mechanism of Luong et al. [LPM15]. Consequently, the forward neighborhood of the vertex v is gathered in a single vector $h_{for,v}^{(j)}$ which is a weighted sum over the connections:

$$h_{for,v}^{(j)} = \sum_{e \in F_v} att_{for}^{(j)}(v, e) \times comb_{for}^{(j)}(e)$$

The idea behind the attention weights returned by $att_{for}^{(j)}$ is that more similar connections appear more relevant and should be dominant in the forward neighborhood as this might enhance a convergence of the vertex representations in the graph. The above mentioned attention mechanism achieves that by gently aligning each forward connection with the current vertex $v \in V$.

Backward Neighborhood

The backward neighborhood of v is aggregated with the exact same methodology so that the vector $h_{bac,v}^{(j)} \in \mathbb{R}^{n_{graph}}$ emerges for an iteration j. In order to distinguish the variables and functions from the forward neighborhood, the index for is replaced with bac in the backward neighborhood aggregation. Figure 6.3 highlights the forward and backward neighborhood vectors for an exemplary vertex. The scope of the aggregated information is darkened. The following equations summarize the computations for the backward neighborhood:

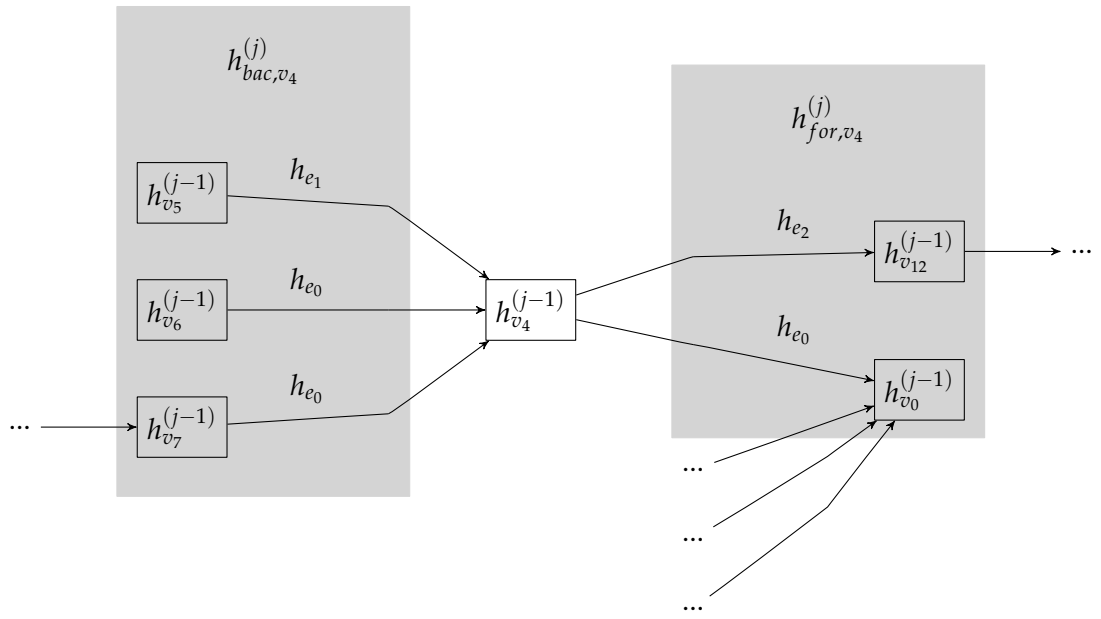


Figure 6.3.: The figure illustrates the aggregation of the forward and backward neighborhood of a vertex. The graph is the one shown in Figure 6.2. Vertex v_4 was chosen exemplarily.

$$h^{(j)}_{bac,v} = \sum_{e \in B_v} att^{(j)}_{bac}(v,e) \times comb^{(j)}_{bac}(e)$$

$$att^{(j)}_{bac}(v,e) = \frac{exp(weight^{(j)}_{bac}(v,e))}{\sum_{e_i \in B_v} exp(weight^{(j)}_{bac}(v,e_i))}$$

$$weight^{(j)}_{bac}(v,e) = \left\langle h^{(j-1)}_{v}, A_{bac,j} \times comb^{(j)}_{bac}(e) \right\rangle$$

$$comb^{(j)}_{bac}(e) = \sigma(W_{bac,j} \times [h_e, h^{(j-1)}_{s(e)}])$$

Incorporation of the Neighborhood

The last action in the iteration j is to integrate the neighborhood information in the latest vertex representation of v. This is done with an unbiased, fully-connected layer. The inputs to this layer are three vectors: the backward neighborhood vector of the

current iteration j, the vertex vector of the previous iteration $j-1$ and the forward neighborhood vector also of the current iteration j. The new node vector is reckoned as follows:

$$h_v^{(j)} = \sigma(W_{inc,j} \times [h_{bac,v}^{(j)}, h_v^{(j-1)}, h_{for,v}^{(j)}])$$

Similar to Equation 6.1, σ is a leaky ReLU. $W_{inc,j} \in \mathbb{R}^{n_{graph} \times 3n_{graph}}$ is a weight matrix and will be adapted during the optimization.[2] Similarly to the matrices $W_{for,j}$, $W_{bac,j}$, $A_{for,j}$ and $A_{bac,j}$, a different weight matrix can be learnt for each iteration.

Even though only adjacent neighbors are taken into account during an iteration, knowledge of distant vertices can be forwarded to v through its adjacent nodes as they might have retrieved information during previous iterations. As the updating is done for all nodes in the graph simultaneously, the vertex vectors can contain information about nodes that are up to k hops away. After the final iteration, all nodes $v \in V$ consist of a hidden representation $h_v^{(k)}$ that has evolved from their attributes, locality and connectivity.

Differences to Graph2Seq

The Graph2Seq model uses iterations as well in order to identify vertex vectors. A major difference with our graph attention network is that Graph2Seq does not consider edge labels in the iterations. Furthermore, the Graph2Seq model does not make use of an attention mechanism when aggregating the neighborhoods of vertices. There are other disparities as well, e.g., the specific formulas differ and Graph2Seq maintains two separate states for each vertex during the iterations instead of a single vector [Xu+18].

6.1.3. Graph Vector

The last step in the graph attention network is to combine the vertex representations in the graph to a single vector $h_G \in \mathbb{R}^{n_{graph}}$ after the iterations have finished. As indicated beforehand, this is done with the following sum:

$$h_G = \sum_{v \in V} w_v \times h_v^{(k)} \tag{6.3}$$

The vectors $h_v^{(k)}$ are the outcome of the iterations as explained in Subsection 6.1.2. Not every node in the graph might be equally important. For instance, the *JCModifiers* node in Figure 6.1 might not yield additional information if one tries to capture the semantics of the code that is represented by that graph. As a result, the sum in Equation 6.3 introduces the weights $w_v \in [0,1]$ with $\sum_{v \in V} w_v = 1$. These weights are figured out

[2] The index *inc* refers to *incorporation*.

by an attention mechanism, more specifically by the attention mechanism proposed by Bahdanau et al. [BCB14]. Please note that this attention mechanism is deployed independently of the one which is specified in Subsection 6.1.2 and which is based on the work of Luong et al. [LPM15]. It differs from the previously used mechanism to that point that it does not use the scalar product as a basis but a feedforward neural network. This feedforward neural network tries to align each instance to a reference vector.

$$a(r, h_{inst}) = c^T \times tanh(W_{ref} \times r + W_{inst} \times h_{inst} + b) \tag{6.4}$$

Equation 6.4 describes the feedforward neural network that takes $r \in \mathbb{R}^{n_{graph}}$ as the reference vector and $h_{inst} \in \mathbb{R}^{n_{graph}}$ as the hidden representation of the instance. W_{ref} and W_{inst} are weight matrices that are trained during the vectorizing model optimization. The vector $c \in \mathbb{R}^{n_{graph}}$ and the bias $b \in \mathbb{R}^{n_{graph}}$ are trained as well. The hyperbolic tangent function serves as the activation function of the network. The purpose of the vector c is to transform the result of the network to a single scalar.

In our case, the reference vector r is the mean m of all vertex vectors:

$$m = \frac{1}{|V|} \times \sum_{v_i \in V} h_{v_i}^{(k)} \tag{6.5}$$

In contrast, the instances h_{inst} are the particular node vectors $h_v^{(k)}$. Each vertex $v \in V$ is then assigned the weight $w_v \in \mathbb{R}$:

$$w_v = \frac{exp(a(m, h_v^{(k)}))}{\sum_{v_i \in V} exp(a(m, h_{v_i}^{(k)}))} \tag{6.6}$$

The weights w_v are computed by using the softmax function over the result of the feedforward neural network – which represents the attention mechanism proposed by Bahdanau et al. [BCB14]. With w_v, the graph attention network can determine the relevancy of vertices by distinguishing how much the vertices vary in their meaning from the rest of the graph: the encoder can decide if and to which extent deviating nodes are less or more important. The weights w_v resulting from the attention mechanism can be again compared to the attention of a human who attempts to understand a method or class graph: a human would probably focus on a few vertices that seem most relevant due to their labels or connectivity.

6.2. Decoder

After a formal graph G is transformed to a hidden vector h_G as explained in Section 6.1, the vector h_G is used to predict properties or labels of G. The decoder is inspired by

latent factor models that commonly use matrix factorization in order to align an input vector space to a different output vector space by introducing latent factors [BKV09]. For our use case, we assume that the encoder of Section 6.1 transformed each graph G to its latent factors represented by the vector h_G. Therefore, we expect that each dimension in h_G specifies the extent of a factor. Even though we do not know the factors in advance, the vectorizing model will learn them during the training phase. For the prediction of the graph labels, a linear alignment between the latent factors and the properties is employed.

The goal of this thesis is the vectorization of source code so that the calculation of the representation h_G is the actual goal since each method or class graph G embeds a source code element. However, the meaning of this representation is imparted by the decoder, i.e., by the fact how the decoder transforms the hidden representation to a set of properties. By processing a vector h_G as latent factors, each dimension of the vector is independent. This eases the interpretation of the hidden vectors: for instance, the cosine distance between vectors is guaranteed to be an effective similarity measurement if the vectorizing model computes meaningful distributed representations for the source code elements.

In order to realize the decoder, we first define the *property vocabulary* $Z = \{z_0, z_1, \dots\}$ which contains all possible labels that a graph can have. In theory, the size of Z can be unlimited similarly to the vertex vocabulary X. In practice, the size will be limited as explained below in Section 6.3. The predicted properties of a graph G will then be given by the variable $prop_G \in \mathcal{P}(Z)$. During the optimization phase of the model, the correct prediction of $prop_G$ is enforced by adapting the trainable variables of the encoder as well as the decoder. For this purpose, the predicted properties $prop_G$ will be compared to the ground truth properties in the training set. The optimizer proposed by Kingma et al. [KB14] is used to adapt the trainable variables if the predicted and the ground truth properties differ.

In order to compute $prop_G$ for G, each word $z_i \in Z$ is assigned a weight $k_{G,i} \in \mathbb{R}$:

$$k_{G,i} = K_i^T \times h_G$$

The vectors $K_i \in \mathbb{R}^{n_{graph}}$ are learnt by the vectorizing model and capture the relevancy of the latent factors in h_G for each label z_i. This means that each K_i identifies the importance of a label z_i by computing a linear combination of the latent factors. As a result, the latent factors – i.e., the dimensions in the graph vector h_G – directly impact the label prediction in a traceable manner so that the meaning of the vector h_G emerges. This is a great advantage over a neural network as decoder which would determine the properties of a graph G in a non-linear way so that the interpretation of h_G can not be identified easily. The interpretation of h_G should be obvious though as the focus of this thesis relies on h_G as mentioned above. Since the computation of $h_{G,i}$ is done for every

label simultaneously, the computation can be written in matrix notation:

$$k_G = K^T \times h_G$$

Each vector K_i is then a column in the matrix $K \in \mathbb{R}^{n_{graph} \times |Z|}$.

We distinguish two different scenarios in which our machine learning model can be employed: multilabel and single label classification. The multilabel classification assumes that elements in Z are not exclusive. Hence, $prop_G$ can be of arbitrary size. In contrast, words in Z are exclusive in the single label classification and every graph has one label: for each graph G, $|prop_G| = 1$ holds true. According to this distinction, the weights k_G are processed differently. The following subsections illustrate the discrepancies.

6.2.1. Single Label Classification

If only a single label has to be predicted for the graphs, the property with the highest probability will be chosen. The probability $P(z_i|Z, G)$ of a label $z_i \in Z$ for a graph G is computed with the softmax function over the weights $k_{G,i}$:

$$P(z_i|Z, G) = \frac{exp(k_{G,i})}{\sum_{z_j \in Z} exp(k_{G,j})} \tag{6.7}$$

The predicted property set is then given by a set with a single label z_i which has the maximum probability.

$$prop_G = \{argmax_{z_i \in Z}(P(z_i|Z, G))\} \tag{6.8}$$

Equation 6.8 finishes the calculation of predicted property set $prop_G$ by taking the label z_i that maximizes the probability in Equation 6.7.

If the prediction $prop_G$ differs from the ground truth properties $truth_G \in \mathcal{P}(Z)$ of G during the optimization of the model, the optimizer of Kingma et al. [KB14] adapts the trainable variables of the encoder as well as the decoder. Since this optimizer is based on the computation of gradients, a loss function has to be supplied which specifies to which extent the prediction deviates from the ground truth. The cross entropy between the probabilities of Equation 6.7 and the ground truth $truth_G$ will be used as the loss function $loss_{single}$ for the single label classification:

$$loss_{single}(G) = -\sum_{z_i \in Z} \begin{cases} log(P(z_i|Z, G)) & z_i \in truth_G \\ log(1 - P(z_i|Z, G)) & z_i \notin truth_G \end{cases} \tag{6.9}$$

This loss function specifies the accumulated difference between the predicted and the actual probabilities of the labels. The log function in the cross entropy makes sure that

a higher deviation from the ground truth is heavily penalized. To that effect, the cross entropy enables a fast convergence of a correct label prediction since the loss will be minimized by the optimizer as the trainable variables are adapted according to their gradients.

6.2.2. Multilabel Classification

In multilabel classification, a variable amount of properties can be predicted. Therefore, the probability of a property $z_i \in Z$ is independent of the probability of other labels: it is given by $P(z_i|G)$. This is a major difference to Equation 6.7 in the single label classification. Hence, the probability will not be calculated with the softmax but with the logistic function:

$$P(z_i|G) = \frac{1}{1 + exp(-k_{G,i})}$$

All labels with a probability greater than a certain confidence *con* are then added to the prediction:

$$prop_G = \{z_i | z_i \in Z \wedge P(z_i|G) > con\}$$

In our implementation of the vectorizing model, we take a confidence threshold of $con = 50\%$. The threshold of 50% could be adapted to higher values as well in order to ensure an increased certainty of predicted labels.

Similarly to $loss_{single}$, we specify a loss function $loss_{multi}$ for the multiclass classification that is used by the optimizer during the training phase in order to adapt the trainable variables of the vectorizing model:

$$loss_{multi}(G) = -\sum_{z_i \in Z} \begin{cases} log(P(z_i|G)) & z_i \in truth_G \\ log(1 - P(z_i|G)) & z_i \notin truth_G \end{cases}$$

The definition of $loss_{multi}$ equals Equation 6.9 except that the probability for a property $z_i \in Z$ is not given by $P(z_i|Z, G)$ but by $P(z_i|G)$.

6.3. Hyperparameters

The previous sections elucidated the specifics of the vectorizing model. These specifics include some hyperparameters that are necessary for the model in order to be operable. Examples for these parameters are n_{graph}, n_{edge} and k. In this section, we will introduce additional hyperparamters that can be set in order to contain overfitting [Haw04]. Additionally, the parameters can have positive effect on the resource consumption and should be adapted according to the available machine on which the vectorizing model is trained.

Label	Count	Percentage	Percentile
get	160	80%	80%
set	30	15%	95%
hello	6	3%	98%
world	4	2%	100%

Table 6.1.: Exemplary distribution of vertex labels for a training set. The labels are sorted according to their frequency. The percentiles sum up the percentage of the current label and all labels that are more or equally frequent.

6.3.1. Vocabulary Limitation

The vectorizing model specifics also include many trainable values, e.g., the vocabulary matrices $U_X \in \mathbb{R}^{n_{graph} \times |X|}$, $U_Y \in \mathbb{R}^{n_{edge} \times |Y|}$ or $K \in \mathbb{R}^{n_{graph} \times |Z|}$. The size of the mentioned matrices depends on the number of attributes in each vocabulary, i.e., for each label in the vocabulary, the degrees of freedom increase. Normally, the vertex, edge and property vocabulary respectively are created from all attributes in the training set that are attached to the nodes, edges and graphs respectively. Our machine learning model would then try to optimize the embedding of each label even if the label occurs rarely and should have little to no impact on the resulting properties of the graphs. The embeddings of rare attributes do not only consume additional resources but might tamper with the predictions: they might lead to correlations that are purely coincidental in the training data but are learnt by the model nevertheless. These correlations are more probable for infrequent labels than for frequent ones. As a result, the model might apply the correlations to unseen graphs misleadingly which is referred to as overfitting [Haw04].

In order to contain overfitting due to infrequent attributes, we do not take all labels in the training set into account for the creation of the vocabularies. Instead we preprocess the graphs in the training data and only use attributes that account for a certain percentage of label occurrences. All other attributes are replaced with the label *UNKNOWN*. The threshold for the percentage is a hyperparameter that is specified for each vocabulary separately. The particular parameters are:

- t_{vertex} for the vertex vocabulary X
- t_{edge} for the edge vocabulary Y
- $t_{property}$ for the property vocabulary Z

With the vocabulary X as an example, the exact methodology is illustrated in following enumeration:

1. Count the occurrences for each label and sort the words according to their frequency. Table 6.1 illustrates an exemplary distribution of vertex labels.
2. Compute the percentile for the node labels. We define the percentile of a vertex label as a sum that is computed from the percentage of the label and the percentage of all other labels that are at least equally frequent.
3. The minimum set of words that accounts for a percentile that is greater than t_{vertex} builds the vocabulary X.
4. The label *UNKNOWN* is added to X. Vertex labels that are not part of X are then replaced with *UNKNOWN*.

For instance, if we assume $t_{vertex} = 0.90$ and a training set with the appearances of Table 6.1, the vertex vocabulary will consist of the labels *get*, *set* and *UNKNOWN* since *get* and *set* build the minimum set that accounts for at least 90% of label occurrences. *hello* and *world* will be replaced with *UNKNOWN*.

6.3.2. Graph Size Limitation

The resource consumption of the vectorizing model is an important factor since it will determine which machines are sufficient for executing the model optimization. Hence, higher consumption can result in longer execution time and higher expenses. The graph attention network in Section 6.1 inspects every node and its connections so that graphs with more vertices result in a higher memory consumption. Additionally, it might be difficult for the network to identify important parts in a huge graph even though the attention mechanism of Subsection 6.1.3 is employed. To overcome these difficulties, we limit the size of the input graphs to a fixed number of $t_{\#nodes}$ nodes where $t_{\#nodes}$ is a hyperparamter. Before each epoch in the optimization process, graphs that contain more than $t_{\#nodes}$ vertices are then truncated randomly. However, not every node is equally likely to be cut away, i.e., the truncation process does not work with a uniform distribution when cutting away vertices. Instead, we decided that nodes with many connected neighbors are more likely to remain part of the graph. Introducing different probabilities for the vertices in the truncation process was found to perform better than a uniform probability distribution. This comes due to the fact that the overall connectivity of the graphs as they are produced in Chapter 5 should not be jeopardized: nodes with many neighbors are likely to play an important role in the method and class graphs because they either build a junction for the structural representation of the source code or represent declarations that are reused at many spots. As a result, graphs with more than $t_{\#nodes}$ nodes will be reduced in their size but the chances remain high that they still are close to the source code elements which they represent. The

probability distribution which specifies the likelihood of a vertex to reside in a graph G is given by a value p_v for each vertex $v \in V$:

$$p_v = \frac{|F_v| + |B_v|}{\sum_{v_i \in V} |F_{v_i}| + |B_{v_i}|}$$

Therefore, the likelihood of a vertex v to remain part of the graph for an optimization epoch is proportional to the count of its outgoing and incoming edges. The limitation of the graph size to $t_{\#nodes}$ nodes saves memory and eases the attention towards important vertices as long as they are not cut off coincidentally before the respective epoch.

The value of $t_{\#nodes}$ should be chosen carefully and definitely rely above the average number of nodes per graph. Otherwise, most graphs will be truncated during the training phase which destroys the purpose of the careful graph creation in Chapter 5. For instance, $t_{\#nodes} = 200$ can be a good value for learning from method graphs whereas the size of non-truncated class graphs is usually bigger so that the limitation should be adapted to a higher value, e.g., $t_{\#nodes} = 1500$.

Another preprocessing precaution that saves memory is the limitation of labels per node and properties per graph respectively. This is done prior to the training phase. By setting the limits to a high value, the overall performance of the model will not be impacted.

6.3.3. Trainable Matrices Limitation

The number of matrices that have to be learnt influences the resource consumption in the same way as the count of their dimensions. There are five weight matrices that occur in each iteration $j \in \{1, \ldots, k\}$ of Subsection 6.1.2: $W_{for,j}$, $W_{bac,j}$, $A_{for,j}$, $A_{bac,j}$ and $W_{inc,j}$. Training separate matrices for each iteration not only raises the resource consumption but also increases the degrees of freedom. Depending on the application area and the training dataset, this can lead to overfitting. Hence, we introduce the hyperparameters $t_{\#aggr}$, $t_{\#att}$ and $t_{\#inc}$ which can be set in order to limit the number of weight matrices that have to be learnt. The resulting limitations can be expressed with the following logical statements:

$$\begin{aligned}
\forall j \geq t_{\#aggr}: \quad & W_{for,t_{\#aggr}} = W_{for,j} \quad \wedge \quad W_{bac,t_{\#aggr}} = W_{bac,j} \\
\forall j \geq t_{\#att}: \quad & A_{for,t_{\#att}} = A_{for,j} \quad \wedge \quad A_{bac,t_{\#att}} = A_{bac,j} \\
\forall j \geq t_{\#inc}: \quad & W_{inc,t_{\#inc}} = W_{inc,j}
\end{aligned}$$

Please note that the forward and backward neighborhood variables are restricted symmetrically. The limitations can be interpreted as follows: since the first iterations of Subsection 6.1.2 mainly aggregate information of the close neighborhood, these

iterations are not affected by the restriction. However, later iterations do not make use of distinctive weights as the vertex vectors should already be converged to a certain degree. As a result, information that comes from further neighbors in later iterations is incorporated less adapted. Therefore, the three limiting hyperparameters should not influence the overall ML performance but reduce the resource consumption during the model optimization.

7. Implementation

The approach for vectorizing software as it is proposed in this thesis is implemented by two different sub-programs. The first of them deals with the transformation of Java code bases to graphs. Therefore, it implements the concept of Chapter 5. The second program handles the machine learning process that encodes the graphs to vectors as highlighted in Chapter 6. The design decisions for both sub-programs, their configuration options and the particular outcome are discussed in the following sections.

7.1. Graph Creation Program

The first sub-program creates the graphs according to Chapter 5. Therefore, it has to parse Java code bases which it then transforms to method and class graphs. The methods and classes are filtered beforehand according to Section 4.2. The parsing library decision and the program structure are explained below.

7.1.1. Parsing Library

For the parsing of code bases, there exist several Java code parsers that capture source code as an AST and could be suitable as basis for the graph creation. An example of such a library is JavaParser.[1] A major disadvantage of JavaParser is that it is only designed to parse but not to compile Java code. As a result, its symbol resolving is immature. Since the resolving of symbols is necessary for specifying the reuse of declarations and hence for creating the method and class graphs, JavaParser will not be utilized in this work.

A mature compiler for Java is the one that is included in the OpenJDK which itself is implemented in Java. Up until version 8, the OpenJDK exposes compiler internals which include not only the AST but also the symbols of syntax tree elements. These exposed classes of the JDK implementation allow an easy and reliable creation of method and class graphs. As a result, our program for the graph creation uses OpenJDK 8 as dependency.[2] This means that versions beyond Java 8 are not supported by our

[1] `http://javaparser.org/`
[2] `http://openjdk.java.net/projects/jdk8/`

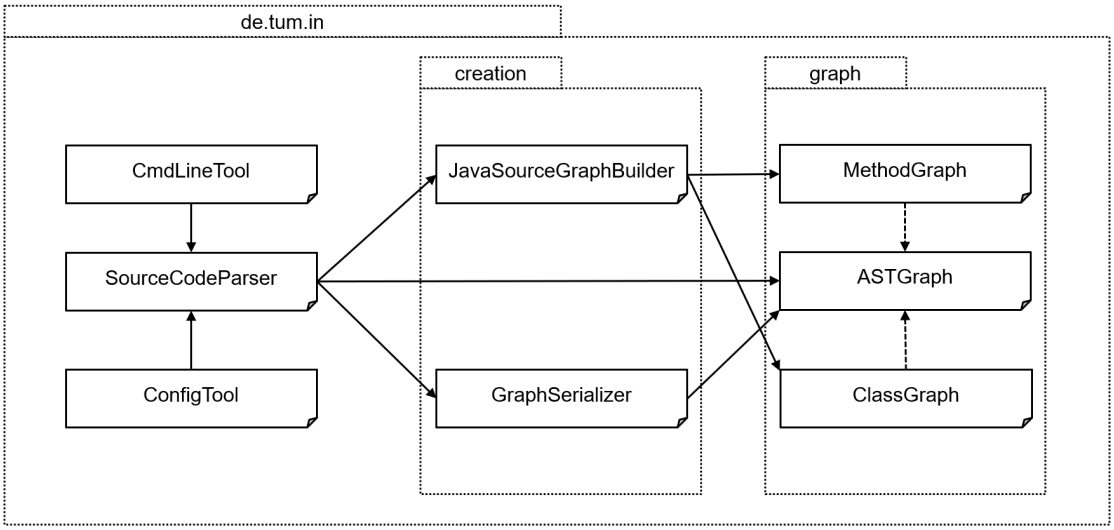


Figure 7.1.: The figure illustrates the most relevant classes and their dependencies in the graph creation program. Packages are illustrated with dotted rectangles whereas all other rectangles represent a class. The dashed arrows indicate inheritance. All other arrows indicate that a method from the destination class is directly called in the source class.

program since OpenJDK 9 and later versions of OpenJDK do not allow access to the internals anymore. Because this thesis builds a proof of concept, the support for the latest Java versions is not necessary at this stage and can be added in future work. Please note that the Java code bases have to be compilable in order to be eligible for the parsing. Otherwise, OpenJDK might not be able to construct the AST.

7.1.2. Structure

The software for the graph creation is written in Java like the OpenJDK. The overall structure of the this software is illustrated in Figure 7.1. Please note that only the most relevant classes and packages are highlighted in this figure in order to preserve a clear view. The program can be called via *CmdLineTool* with a configuration file which is given in JSON format. Alternatively, command line options can be passed to *ConfigTool* with the same effect. The following inputs have to be provided:

- At least one code base: the code bases are given in form of directories. The program will traverse the directories in order to find files ending with *.java*. These files will be parsed with OpenJDK 8 for the method and class graph creation.

- Output directory: the resulting graphs will be stored in this directory at the end of the execution. Each graph is saved in separate JSON file as this format is flexible and allows a simple data exchange across programming languages.
- Classpath: the dependencies of the code bases have to be given in form of a classpath. If the classpath is not provided, the JDK might not be able to discover and resolve all symbols correctly. In case of missing dependencies, the program will terminate depending on the severity of the dependency absence.
- Additional input options specify whether method graphs, class graphs or both should be produced. If class graphs should be outputted, a further option distinguishes the truncated or non-truncated graph generation.

There are other options as well. As they have little impact and are too detailed, they are not listed here.

The general process of the graph creation is then managed by *SourceCodeParser* according to the input options. It thereby delegates the execution to the responsible classes:

1. The code bases will be parsed by *JavaSourceGraphBuilder* first. It also filters the methods and classes according to Section 4.2. It then passes the corresponding subtrees of the AST to *MethodGraph* and *ClassGraph* respectively – depending on whether method graphs, class graphs or both have to be generated. *MethodGraph* and *ClassGraph* implement the methodology of Chapter 5 and store a single method or class respectively. Both classes inherit from *ASTGraph* which in return depends on JGraphT.[3] JGraphT is a Java library for graphs.
2. After the graphs have been created and are stored internally in the program, they have to be transformed to JSON files. This is done by *GraphSerializer*. The serialization is realized with help of the Jackson project.[4]

7.2. Vectorizing Model Program

The second sub-program implements the vectorizing model which is explained in Chapter 6. The implementation uses Python 3 in combination with Tensorflow as basis.[5] Tensorflow allows the implementation of neural networks as dataflow graphs [Aba+16] and enjoys great popularity in the field of deep learning. The following subsection will highlight the overall structure of the program and its execution modes.

[3] `https://jgrapht.org/`
[4] `https://github.com/FasterXML/jackson`
[5] `https://www.tensorflow.org/`

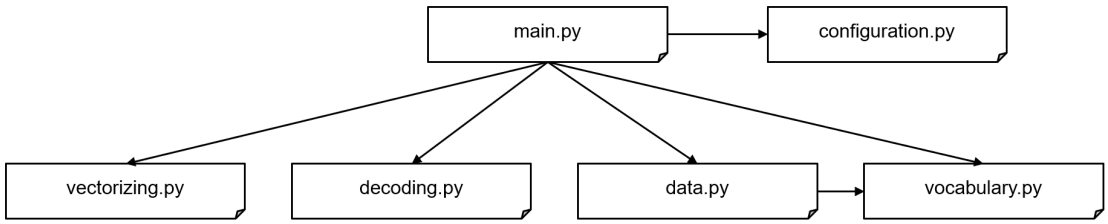


Figure 7.2.: Each Python source file is illustrated as a separate rectangle. The arrows indicate whether contents of a file are imported in another file.

7.2.1. Structure

Figure 7.2 illustrates the most important source files in the program and their interconnections. Please note that two files were omitted due to their negligible impact. The following list explains the source files and their responsibilities.

- *main.py*: this file is the entry point for the program. It adjusts the control and data flow. According to the chosen execution mode, a different control flow within *main.py* is executed. The different modes are explained in Subsection 7.2.2.
- *configuration.py*: the input options are stored in this file. Each hyperparameter of Chapter 6 has to be assigned here. The execution mode is also selected in *configuration.py*. Additional input options include the directories that contain the location of the graphs which have to be processed. Most of the additional input options depend on the execution mode.
- *vocabulary.py*: this file contains amongst others the class which represents the vertex, edge or property vocabulary.
- *data.py*: the graphs are parsed and preprocessed in classes of this file. This means that several hyperparameters of Section 6.3 are realized here. This also includes the vocabulary creation.
- *vectorizing.py*: it contains only a single class which represents the encoder that was introduced in Section 6.1. This class defines all variables and calculations of the encoder. It is only instantiated once during the program execution.
- *decoder.py*: the decoder that is explained in Section 6.2 is maintained here. The file contains two classes: one for the single label classification and another one for the multilabel classification. Similarly to the class for the encoder, either of the two classes is instantiated only once.

7.2.2. Execution Modes

In *configuration.py* of the model implementation, all hyperparameters explained in Chapter 6 have to be specified. The model distinguishes three major execution modes: *training*, *predicting* and *vector export*. Each of these modes has further, individual parameters.

Training Mode

In training mode, the model implementation expects a training dataset as input. The dataset contains graph JSON files that are created with the program of Section 7.1. However, the JSON files have to be extended with the properties towards which the vectorizing model should be optimized. According to the attached properties in the JSON files, the configuration has to specify whether the decoder for single label or multilabel classification has to be used. Afterwards, the program optimizes all trainable values of the vectorizing model in several consecutive epochs by using the Adam algorithm [KB14] so that correlations between the graphs and their attached properties are captured. After each epoch, a checkpoint is created which contains the trained values. In addition to the training dataset, a validation dataset can be specified. It also contains graphs that have properties attached to them. If a validation set is specified, metrics about the prediction performance on the validation set will be printed after epoch: for each model checkpoint, the prediction of properties can be evaluated for graphs which the vectorizing model did not learn on. Hence, the generality of the specific checkpoint is tested with the validation set.

Predicting Mode

In the predicting mode, the python program expects a set of graph JSON files as they are produced by the graph creation implementation. According to a given checkpoint, the program predicts the attributes for the graphs by employing the encoder and the decoder together. The particular decoder has to be specified corresponding to the type of classification: single label or multilabel. Even though the prediction of attributes for method or class graphs is not the focus of this thesis, this mode can be used to determine how a trained model performs when applied to unseen graphs – similar to the validation set during the training mode. Hence, this mode is mainly helpful for validation purposes.

Vector Export Mode

This mode is employed for the vectorization of graphs after the vectorizing model has been trained. The trained model is handed over as a checkpoint. The model implementation employs the optimized encoder in order to transform user-defined graphs to vectors. These graphs are again delivered in form of JSON files. The resulting vectors are stored in a single JSON file that can then be used as input for other machine learning algorithms like outlier analyses or clustering.

8. Application on Method Level

The hypothesis of this thesis is that a neural network is able to locate the correlations between source code and its hidden properties for a set of given Java projects. As a result, the neural network would be able to encode these properties for any code base according to the identified correlations. In order to evaluate the quality of the approach, this chapter presents an experiment with the vectorizing model on method level and discusses the outcome.

8.1. Experiment – Semantic Vectorization

The goal of the experiment is to create vectors that describe the semantics and therefore the meaning of Java methods. Since the vectorizing model is a supervised ML algorithm, a training dataset with predefined labels has to be specified. The quality of the learnt vectors will then be measured by evaluating whether the vectorizing model is able to predict the labels of methods which were not part of the training dataset. Subsection 8.1.1 will therefor explain the decisions for the label creation. Afterwards, the experiment and its outcome are explained in detail.

8.1.1. Label Creation

In order to use our vectorizing model, one has to specify a training dataset first. This means that a set of methods has to be gathered where descriptive properties are attached to each method. These properties build the ground truth and the model will try to infer correlations between them and the respective method. For semantic vectors, these properties would have to characterize the meaning of the method.

The most labor-intensive way to create the characteristics would be to manually assign properties in natural language. The manual labeling approach will result in a small dataset though which in return might lead to a trained model that does not generalize well: the model can only make inferences that exist in the training data. It is improbable that a small dataset contains enough variety or a concise discrimination of method semantics. As a result, a model based on few training examples will not reach a sufficient capability for capturing the meaning of diverse methods that occur during software health checks: the practicality of the resulting ML model is questionable.

Alternatives for the manual property assignment are the automated label generation or the utilization of existing properties. To the best of our knowledge, there are no ready-to-use tools for the automated label generation of source code so that we will neglect this possibility: creating an own tool set involves a heavy exploration and comparison of technologies and exceeds the scope of this thesis.

Therefore, the question about existing properties remains. A solution might be the utilization of online forums: StackOverflow[1] is a website where programming related questions are posted in natural language and answers or questions can contain source code. Previous work about machine learning on source code – e.g., by Allamanis et al. [All+15] – uses data from this website in order to retrieve NL sequences that correlate to source code snippets. Since the code in the questions or their answers are normally well explained by the NL phrases, the data from StackOverflow could build a reliable basis. However, the included code is usually simplified in order to make it better understandable for the audiences. To that effect, a resulting training set does not represent source code of real world software projects as they occur during health checks: the trained model does not generalize well yet again.

Another source for method attributes are the method names: regardless of the method's nature, it always has a name. Hence, it can always be extracted. Additionally, there exist methodologies for the meticulous nomenclature as suggested by Deißenböck et al. [DP05]: they propose a methodology of concise and consistent naming by introducing so called *concepts*: "a concept is a unit with an associated meaning in terms of properties or behavior"[DP05]. With this approach in mind, the method names within a certain scope – e.g., a project – describe the semantics consistently and precisely.

Admittedly, there are some pitfalls with the identifier naming. Obviously, the naming depends on the experience and carefulness of the programmer, i.e., lack of care can result in suboptimal method nomenclature. As a consequence, the names would not be a proper source for identifying the semantics of source code. Furthermore, even if the approach of Deißenböck et al. [DP05] is followed, the naming might only be consistent within a certain scope so that the consistency does not necessarily translate across scopes or projects. However, this effect can occur for any properties built by humans. As a result, method naming will be our best chance of creating a training dataset that captures the semantics and is reliable, general and of sufficient size at the same time. The downsides of this labeling have to be kept in mind when investigating the machine learning performance. Related work which also applies machine learning on source code uses the method naming as evaluation basis, too [APS16][Alo+18a][Alo+18b][ALY18].

[1] `https://stackoverflow.com/`

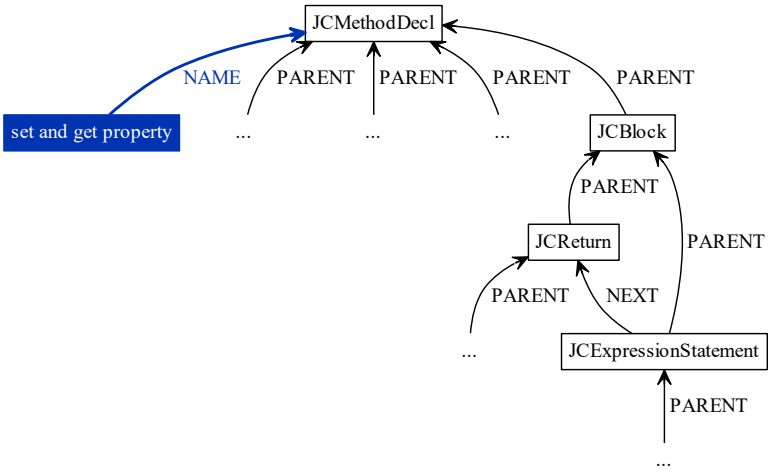


Figure 8.1.: The figure shows the graph of Figure 5.3 with omitted subgraphs. For the evaluation on method names, the vertex highlighted in blue will be cut off.

8.1.2. Setup

The ability of learning semantic properties will be tested in this experiment. The properties will be the method names for the aforementioned reasons. Even though the predicting of properties is not the intended purpose of the vectorizing model, the experiment will allow an assessment whether the model is able to draw correlations between methods and their properties: if it was not able predict the properties, it would be improbable that it is able to correlate the inputs and their attributes. As a result, the computed vectors would not store meaningful information. This subsection will highlight the specifics of the experiment.

Procedure

The methodology of vectorizing methods is illustrated in Section 4.6. For the experiment of predicting method names, the methodology explained in Section 4.6 is adapted and results in the following procedure:

1. A training, validation and test dataset have to be identified first. The three sets have to be distinct. The training dataset will be used for the actual optimization of the vectorizing model whereas the validation dataset will be used in order to determine which combination of hyperparameters and amount of training epochs yields the best model. In contrast, the test set serves as basis for the evaluation of the model performance.

2. Each method in the three datasets has to be assigned properties. In this experiment, the method names serve as properties. Thereby, the method name of each method is split into tokens according to snake case and camel case. The resulting tokens are transformed to lower case and build the properties of the respective method. Since the tokens in a method name are not exclusive, the decoder for multilabel classification will be used later. Due to the nature of this decoder, multiple occurrences of the same token in a method name are ignored. Additionally, the order of the tokens as it is given by the method name is neglected: we assume that the existence of a property is more relevant than the context in which it occurs.

3. The methods in the datasets will be transformed to graphs. As illustrated in Chapter 5, method graphs have the method name attached to them. Since this might prevent the model from learning the names from the hole graph, we remove the method name from the graphs. As highlighted in Figure 8.1, the node that contains the method name is cut off. Apart from that, the method graph creation is not altered.

4. The vectorizing model will be optimized on the training dataset next. This means that the encoder and the decoder are trained as a unity as proposed in Section 6. The outcome of this step is the model that performs best on the validation dataset.

5. The trained model will be evaluated. In contrast to the intended motivation, the trained encoder and decoder are employed as unity in order to predict the method names of the test set. After defining metrics, the predicted and actual names of methods in the test set are compared so that a quantitative evaluation results.

Datasets

As mentioned above, a training, validation and test set have to be gathered. For this purpose, we parsed common, open-source Java projects and libraries from Github[2] and the Apache Software Foundation (ASF) Git repository[3]. Altogether, eleven projects were parsed which resulted in a total of 212,392 method graphs. Table 8.1 depicts additional information about the datasets. Please note that the projects were not necessarily parsed completely. The reason for only partly inspecting projects was the effort of resolving dependencies: the graph creation implementation needs the classpath for a project as input as mentioned in Chapter 7.

[2] `https://github.com/`
[3] `http://git-wip-us.apache.org/`

Dataset	Project Name	#Methods	Git Repository	Hash of Commit
Training		**161,372**		
	Elasticsearch	22,819	Github	485915bbe780949a03a1bff0dcde6a81a39de3bb
	Guava	9,720	Github	3dfee64294921a4d8ec634961991210f6b188e88
	Hadoop-Common	49,042	Github	3f80dbb358190b07f04a0dbf27e049b404bb9303
	Hibernate-ORM	27,511	Github	9ff14a33c78b11f03351748dfe4a5610517325f3
	Maven	1,862	Github	e4e33f73b8f089e443a3bcc678f509bf0daffb15
	Presto	31,820	Github	65edf4963f50798ff9d449f76caa6cbd6e69ed45
	Wildfly	18,598	Github	57204824f93939793c9efc05d8b3d0b131699a9c
Validation		**19,332**		
	Cassandra	19,332	ASF	4dd7faa75210f635af36c0852e9b0d2e8bdbb95c
Test		**31,688**		
	Spring-Boot	8,812	Github	38f112b9e1d1d0e0802bb66f802037959551ed6a
	Spring-Security	4,934	Github	566bc6a6e1064422dd9582abded2481de76f6833
	Tomcat	17,942	Github	ffc4b76e42fd39d88c9417d0ba2b3d697c16f5b5

Table 8.1.: The table enumerates information about the respective datasets. The collected Java projects stem from similar application areas in order to allow a fair evaluation.

Metrics

For the quantitative evaluation in Step 5 of the above procedure, metrics have to be defined. Thus, we will use the same metrics as proposed by Ulon et al. [ALY18]. As a result, precision, recall and F1 scores will be measured for the test dataset. These metrics are based on the count of *true positives*, *false positives* and *false negatives*. For each of the three, a respective counter will be used. The counters are updated be comparing the prediction and the ground truth for each method name. This is done on a token basis:

- If a token that is contained in the method name is not predicted for that method, the *false negative* counter is incremented.
- If a token that is not contained in the method name is predicted for the method, the *false positive* counter is incremented.
- If a token is correctly predicted for a method name, the *true positive* counter is incremented.

According to the three counters, precision, recall and F1 are then calculated. Please note that even if a predicted method name is only partly correct the metrics can improve.

Configuration and Baseline Models

For the prediction with the vectorizing model, the hyperparameters in Appendix C performed best. With these parameters, the vectorizing model was optimized on the training dataset of Table 8.1 for 40 epochs. Later epochs did not show any performance gain on the validation set. The checkpoint that performed best on the validation dataset within the 40 epochs was chosen as final model for the property prediction.

In order to put the scores of our model into context, we will compare it with other machine learning models. We chose the zero rule classifier as trivial baseline. This classifier will always predict the same property for any method in the test set. This property is the most frequent token of the training set. For our training set of Table 8.1, the token *get* occurs most often.

A more sophisticated ML model is the convolutional attention network proposed by Allamanis et al. [APS16] which is specialized on the prediction of sequences for source code snippets, i.e., it predicts an ordered set of properties for source code. Since the aforementioned metrics do not take the order of predictions into account, it will be ignored during the assessment of the scores: the predicted sequences of the neural network are interpreted as a set for our evaluation. This neural network by Allamanis et al. analyzes the code snippets by interpreting source code as a sequence of words like natural language, i.e., Allamanis et al. do not transform software to structured objects before the machine learning algorithm is applied. As the name suggests, the convolutional attention network is based on a convolutional neural network (CNN). It also uses a gated recurrent unit (GRU) which is a variant of an RNN for the sequence prediction [Chu+14]. An attention mechanism is employed as well. Since the convolutional attention network offers several hyperparameters, we went for the configuration that is suggested in the paper by Allamanis et al. [APS16]. The network retrieved the same training and validation set like our model for the optimization phase. Additionally, it was trained for 100 epochs: no performance gain was apparent for later epochs. Corresponding to our model, the checkpoint that performed best on the validation set was chosen as second baseline model.

8.1.3. Results

After training the two baseline models and our vectorizing model, the scores as shown in Table 8.2 result. Surprisingly, the zero rule classifier achieved a precision of over 30%. Since this classifier always predicts the token *get*, nearly a third of the test set methods are getters or methods that include *get*. Due to the low recall, the F1 score of the zero rule classifier settles down at less than 17%. As expected, this trivial classifier was clearly outperformed by both our vectorizing model and the convolutional attention

Model	F1	Precision	Recall
Zero Rule Classifier	0.168	0.301	0.116
Vectorizing Model	0.505	0.582	0.445
Convolutional Attention Network	0.520	0.678	0.422

Table 8.2.: The table highlights the prediction scores for the dataset of Table 8.1.

network by Allamanis et al. [APS16].

The convolutional attention network outperformed not only the zero rule classifier but also our vectorizing model by a gain of more than one percentage point with respect to the F1 score: the precision of the convolutional attention network exceeded the score of our model whereas the vectorizing achieved the better recall. For both models the precision is significantly higher than the recall. This means that many of the predicted tokens were actually part of the method name.

8.1.4. Interpretation

Even though the vectorizing model and the convolutional attention network outperformed the zero rule classifier by far, the F1 scores are still not close to a perfect prediction and have values slightly above 50%. This comes due to the fact that for the method name prediction one has to choose the correct words from a large vocabulary which can theoretically be of infinite size. Additionally, method names can not always be concluded from their method body either because of inconsistent naming or because of lacking the context of the method.

The reasons for the fact that the convolutional attention network performed better than the vectorizing model might lie in the combination of this specific task and the differing approaches of the models. As highlighted by Deißenböck et al. [DP05], identifiers account for a high percentage of tokens in source code. Deißenböck et al. also introduce *concepts* as mentioned in Subsection 8.1.1. By assuming a concise and consistent identifier naming, similar concepts should be reflected by the method name and the identifier names that occur within the respective method. As a result, the method name prediction can be eased by not only inspecting the source code structure and the cross-referencing of declarations but by simply aggregating the concepts which are represented by the identifier names.

The convolutional attention network does not employ a transformation of source code elements before the CNN is applied [APS16]. Hence, the high percentage of identifier tokens is represented in the model input as well. In contrast, the graph creation in Chapter 5 for the vectorizing model causes that the ratio of identifier tokens is not reproduced anymore: each identifier name only occurs once in the final graph

– no matter how many times the identifier appears in the method body. For large methods, the names might even be coincidentally cut away as explained in Section 6.3. Consequently, our vectorizing model can not take advantage of obvious correlations between method and identifier names: it mainly relies on the source code structure and the data usage.

Another reason for the better performance of the convolutional attention network might be that this neural network is specialized for sequence prediction. This means that for each token which is predicted for a method a new vector is computed. As a result, a predicted token results from individually analyzing the method body. In contrast, our model attempts to predict all tokens of a method name from a single vector due to the original motivation. As a result, all relevant information in a method has to be aggregated in this unique vector. Hence, the usage of individual vectors is at an advantage since relevant parts of a method can be aligned gently for each label so that a clear distinction between source code parts and their corresponding label is possible.

8.2. Discussion

The purpose of the previous experiment is to assess whether our approach is able to capture latent information on method level in order to predict semantic properties. The prediction of properties is of course not the intended goal of our methodology. On a higher level of abstraction, it allows conclusions though about the quality of the computed vectors that are the result of the encoder. With a satisfying F1 score of 50.5%, the vectorizing model is able to embed lots of relevant information. This score is achieved even though the specific experiment does not favor the vectorizing model but rather the convolutional attention network as explained in Subsection 8.1.4. Thus, our approach can be assumed to smartly embed the method structure and data usage in order to predict semantic properties. Therefore, the method level allows the embedding of semantic characteristics which provides an answer to RQ3 in Chapter 3. Although the experiment demonstrated abilities of the vectorizing model, the practicality of this experiment will be discussed in the following subsections. Hence, the applicability for our motivation and exemplary usages will be explained.

8.2.1. Practicality of the Experiment

The motivation of our work is based on health checks at itestra GmbH. Thereby, IT consultants are faced with an unknown code base which they have to review. Applying the model of the above experiment, names could be suggested for methods in the respective code base. This does not directly yield additional assistance since the

method names are already given in the source code that is subject to a health check. Furthermore, the score of predicting the nomenclature in the experiment is not sufficient enough in order to evaluate whether the actual names in the software are concise: the predicted method names themselves are not always precise so that comparing the predictions with the actual names does not yield additional assistance either. Even if strong deviations of the prediction from the actual name might indicate a potential misnaming, the above experiment is hardly applicable during health checks.

8.2.2. Usage Example

The experiment did however show that our model is able to gently capture relevant information as it clearly outperformed the trivial baseline model. Keeping the ubiquitous presence of method names in a health check in mind, this fact can be utilized. A benefit for analyzing unknown projects can be achieved by using the model with its intended purpose: the vectorizing of code elements. Assigning methods to semantic categories builds a great assistance: it provides a rough idea about which parts accomplish similar tasks and therefore about the modularity of the project. For instance, if we assume a project in an object oriented programming language for which no clear emergence of semantic clusters can be observed, it might be a sign that the structuring of the project shows poor quality: no functional decomposition is given so that unnecessary complexity might be inflicted to the code base [ST04]. This subsection illustrates the semantic vectorization by means of an example Java project: the Guava library. It is part of the dataset in Table 8.1 and offers implementations of commonly used datatypes and other utils.[4] Since this library contains datatypes like collections and graphs, its methods offer diverse functionalities so that the semantic vectors which are created with the help of the method names yield an interesting insight in the project. Thus, this subsection provides an exemplary automated software analysis that demonstrates the applicability on the method level. An answer for RQ2 in Chapter 3 will therefore be provided.

Setup

For the experiment in Section 8.1, three distinct datasets were employed: the training, validation and test set. This is due to the fact that the experiment of Section 8.1 was conducted in order to demonstrate that the vectorizing model is able to generalize and predict properties for methods that it was not trained on. For the discussion in this subsection, the method vectors and not the predicted names are relevant however. Therefore, the distinction of separate datasets is not necessary. To this effect,

[4] `https://github.com/google/guava`

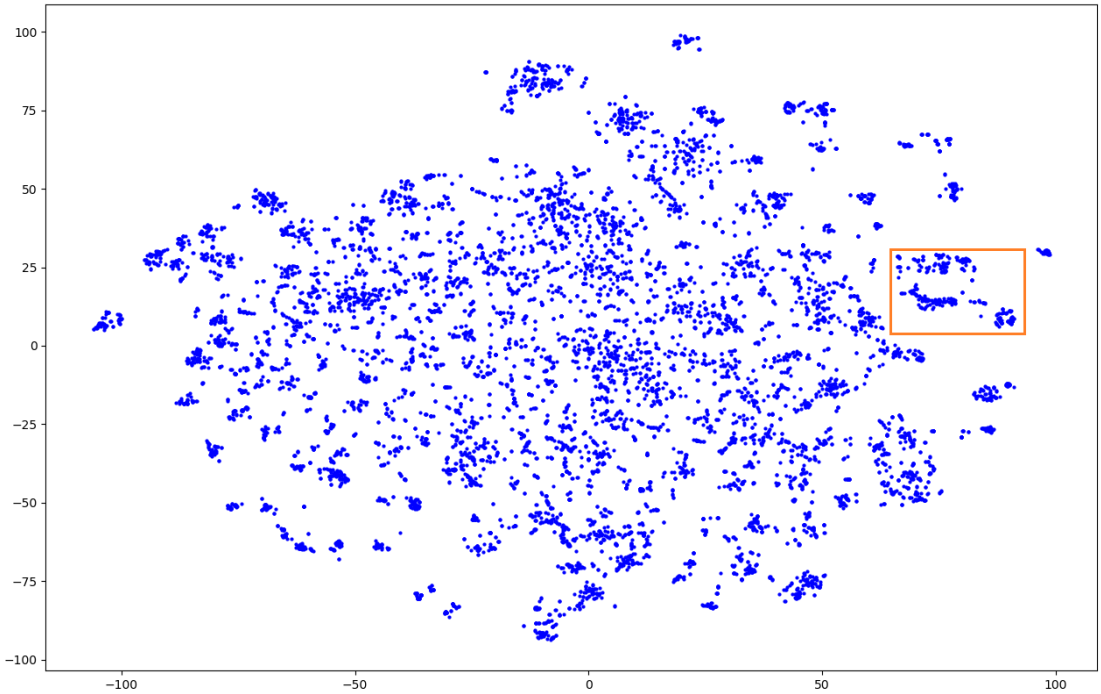


Figure 8.2.: Each point in the figure represents a method of the Guava library. The position of the points is determined by t-SNE which was applied on the method vectors. The orange rectangle highlights some of many point accumulations. Please note that the axes and their scales are figured out by t-SNE and have no deeper meaning.

the vectorizing model will be trained on the method names of the Guava library. Afterwards, the trained model will be used in order to vectorize the methods it was trained on. This approach can also be applied for projects that are subject to a health check since the method names are always available for their methods.

For the usage example, the vectorizing model was trained for 100 epochs on the method graphs of the Guava library with the hyperparameter settings which were used in Subsection 8.1.2 and are listed in Appendix C. The 100 epochs of training were chosen as no improvement was apparent in later epochs. By means of the metrics explained in Subsection 8.1.2, the model checkpoint that reached the highest F1 score for the name prediction in Guava is selected for the following analyses.

Visualization

After the vectorizing model was optimized, the produced method vectors have to be investigated and processed. A first step is to visualize them. However, their dimensionality is given by the hyperparameter n_{graph} which is set to 96 as specified in Appendix C. In order to visualize a dataset of vectors with high dimensionality, the T-distributed Stochastic Neighborhood Embedding (t-SNE) can be used. By attempting to preserve the locality of the datapoints, t-SNE reduces their dimensionality in a non-linear way [MH08]. In our case, this means that the vectors are mapped to points in two-dimensional space. Since the locality is likely to be preserved by t-SNE, vectors that are similar according to the cosine similarity are mapped to points that maintain a close distance in the euclidean space, i.e., their points are visualized close to each other.

With the help of the t-SNE implementation in the scikit-learn library [Ped+11], the visualization in Figure 8.2 emerges. Each point in the figure represents a method vector and therefore a method. Most of the plotted points can not be related to obvious clusters even though denser areas are visible. This is especially the case in the mid of the plot. On the borders of the plot, clear accumulations of points or clusters can be observed however. This means that each method in such an accumulation is encoded to a similar vector. Since the method vectors are created to optimally compute their names, methods in an accumulation should have the same or a similar name. For instance, this is demonstrated by the set of clusters which resides in the right part of Figure 8.2 and is surrounded by an orange rectangle: most of the methods in this accumulation have the name *toString*.

These clusters are first of all a sign for a consistent nomenclature: the model is able to assign methods to clusters by correlating the method bodies and names. On a higher level of abstraction, the accumulations also illustrate semantic aspects as similar method names should refer to similar semantics. Consequently, it can be assumed that a functional decomposition is given for the Guava library since accumulations are obvious in Figure 8.2.

The visualization of the semantic vectors is applicable during health checks and gives a first impression about the consistency of the nomenclature and the functional decomposition.

Similarity Search

Since the method graphs of Chapter 5 mainly embed the structure and the data usage of a method, the machine learning model of Chapter 6 stores exactly this information in the method vectors in a smart manner so that the method names can be predicted. To that effect, the method vectors are likely to embed the structure of a method but

correlate to the method name at the same time. Assuming a complex method, the identification of the correlation with the method name might be hindered but the structure should still be embedded in the resulting vector.

The aforementioned visualization highlighted that several accumulations of method vectors emerge for the inspected project. Since the plot in Figure 8.2 is based on t-SNE with the cosine similarity, the visualization illustrates furthermore that the cosine similarity between the method vectors indeed has a meaning. Keeping in mind that the structure is represented to a certain extent by the method vectors, the cosine similarity between method vectors can be used to find similarly structured methods even if the names were not aligned correctly for these methods. Thus, the computed vectors can be processed with nearest neighbor algorithms.

During a software health check, this feature might be of additional assistance: having identified a complex method that lacks comprehensibility, similar methods in the code base can be found with the computed vectors. Appendix D gives an example for the Guava library.

9. Application on Class Level

The application on method level in Chapter 8 introduced an experiment that demonstrated the quality of our approach and discussed the applicability of the results for health checks. Similarly, the class level will be evaluated in this chapter: an experiment will be presented in Section 9.1 whereas the results and their practicality will be discussed in Section 9.2.

9.1. Experiment – Maintainability Forecasting

Our original motivation arises from software health checks at itestra GmbH. A major goal during these checks is to "reveal cost drivers" for a software system.[1] As highlighted by Boehm et al. [BP88], the software productivity and therefore the costs can be indeed impacted by the software quality which the maintainability is part of. Consequently, this experiment attempts to capture the maintainability for Java classes with the vectorizing model.

9.1.1. Label Creation

Since the vectorizing model is employed in a supervised ML setting, a training dataset has to be supplied for the experiment. Hence, labels which describe the maintainability have to be attached to the Java classes that are part of the training set. However, the estimation of the maintainability is normally subject to personal impression and experience of programmers. In order to capture the judgment of professionals, we adopt the labeling proposed by Schnappinger et al. [Sch+19]. Schnappinger et al. introduce three separate labels for their maintainability forecasting which is based on the result of static analysis tools. The following listing enumerates verbatim the labels as they are defined for classes [Sch+19]:

- *Label A* indicates the absence of indicators for maintainability problems with respect to the ease of change.
- *Label B* covers classes with some room for improvement.

[1] `https://itestra.com/en/leistungen/software-healthcheck/`

- *Label C* is assigned to code that is clearly hard to maintain and requires high effort to be changed.

With their descriptions in mind, the labels have then to be manually assigned to the Java classes by inspecting the source code. Schnappinger et al. realize the assignment with the help of experts so that the labeling is representative for code reviews of industrial software projects. A rule-based assigning of the labels is inadvisable since the ML model might not learn the expert opinion but the rules [Sch+19]. Even though the manual assignment of these labels above might still be subject to personal opinions, they offer a reliable frame for the manual label creation.

9.1.2. Setup

With the above labeling, this experiment evaluates whether the vectorizing model is able to predict the labels for classes it was not trained on. Yet again, not the vectorizing but the predicting of characteristics will be measured. Since the predictions are made from the vectors, their quality is still crucial in order to perform well in the experiment. Therefore, the meaningful vectorization will be tested indirectly.

Procedure

The procedure for conducting the experiment can be summarized as follows:

1. A training, validation and test dataset of Java classes have to be gathered. As usual, the training set is used for the optimization phase whereas the validation set helps to identify the optimal hyperparameter setting and number of training epochs. The test set serves as basis for a quantitative evaluation.

2. After the datasets were identified, each class in the datasets has to be labeled. As explained above, the labels will be created by manual code inspection. Afterwards, each class is assigned to a category: *A*, *B* or *C*. Since each class has exactly one label, the decoder for single label classification is employed in the vectorizing model.

3. The classes are transformed to graphs next. Since the method level is not investigated in a separate model and the methods of classes might play an important role for determining the class maintainability, the Java classes will be transformed to non-truncated graphs. Their creation is explained in Chapter 5.

4. The variables of the vectorizing model will be optimized. This means that the encoder and decoder are trained as a unity on the training dataset as usual. The

	#A	#B	#C	Total
Training Set	65	14	14	93
Validation Set	39	8	9	56
Test Set	26	5	5	36
Total	130	27	28	185

Table 9.1.: The table lists the amount of labels in each dataset. The classes were split 50-30-20 into training, validation and test set. The percentage of labels is preserved in each dataset.

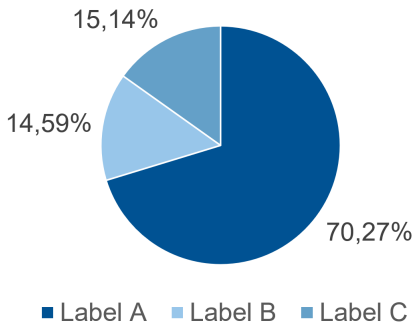


Figure 9.1.: The distribution of the labels *A*, *B* and *C* in the parsed classes. This distribution is preserved in the particular datasets. The datasets are shown in Table 9.1.

model checkpoint that performs best on the validation set serves as basis for the next step.

5. The encoder and decoder of the best model checkpoint are employed together in order to predict the maintainability for the classes in the test set. By inspecting the discrepancy between the predictions and the actual labels, a quantitative assessment results.

Datasets

For the datasets, several classes of JUnit 4[2] and a full stack insurance software were investigated and categorized according to their maintainability. These two projects correspond to two of the three projects that were analyzed by Schnappinger et al. [Sch+19]. Schnappinger et al. also provided the labeling of the extracted classes for our evaluation.

[2] `https://github.com/junit-team/junit4`

	Predictions *A*	Predictions *B*	Predictions *C*	Total
Ground Truth *A*	26	0	0	26
Ground Truth *B*	3	0	2	5
Ground Truth *C*	2	0	3	5
Total	31	0	5	36

Table 9.2.: The table shows the distribution of predictions with absolute numbers for the test set of Table 9.1. It highlights which labels were predicted for each category.

Since their labeling emerged with professional assistance, the labels can be assumed to be precise and consistent. Due to their manual investigation, only 185 examples result. However, the chosen classes sum up to 51k lines of code [Sch+19]. As Table 9.1 illustrates, these examples are randomly split into a training (50%), validation (30%) and test set (20%). The distribution of the labels is visualized in Figure 9.1. It is preserved in each dataset in order to allow a fair evaluation.

Configuration

With the help of the validation set, the hyperparameters as listed in Appendix E were found to perform best. Using these parameters, the model checkpoint that had the best outcome for the validation set within 50 epochs of training was chosen as the result of Step 4 in the above enumeration. The training was stopped after 50 epochs since no performance improvements occurred in later epochs.

9.1.3. Results

The best model checkpoint predicted the degree of maintainability correctly for over 80% of the samples: more specifically, 29 out of 36 test samples in Table 9.1 were properly forecasted. Table 9.2 depicts the results on the test set in detail. It illustrates that the vectorizing model did not once predict the category *B* but only predicts *A* or *C*. On the other hand, it predicted all samples with label *A* properly.

9.2. Discussion

Due to the tiny test dataset and the inhomogeneous label distribution, the high percentage of correct predictions in the experiment is not significant. The detailed listing in

Table 9.2 provides however detailed information about the experiment and its practicality. Informally expressed, the table reveals that the model is able to identify source code of good maintainability whereas bad classes are only partly detected.

Since classes with maintainability flaws were detected, the vectorizing model can be indeed integrated in the health check. Even though it is not guaranteed that all bad classes will be detected, it can be assumed that only classes with flaws will be labeled with category *C*. This means that there might be false negatives with respect to the prediction of *C* whereas the chance of false positives is low: in the experiment, the label *C* was never assigned to a Java class for which "indicators for maintainability problems with respect to the ease of change" [Sch+19] are absent.

In order to investigate the applicability of the vectorizing model, we applied the trained model of the experiment explained in Section 9.1 to parts of the JSweet project.[3] JSweet is an open-source transpiler that converts Java to TypeScript or JavaScript.[4] In total, 97 classes were parsed. Of the 97 classes, the model of Subsection 9.1.3 labeled none as *B* and 18 classes as *C*. For a proper assessment, we inspected the classes that were categorized as *C*: according to our own intuition, we manually assigned the labels by keeping their definition by Schnappinger et al. [Sch+19] in mind. Our manual categorizing for these 18 classes produced the following result:

- Eight of the classes were correctly predicted to be part of the category *C*. Their size or complex methods indeed impact the maintainability.
- Eight other classes should be categorized as *B* rather than as *C*. For them, there is room for improvement but the effort for change remains moderate.
- For two classes, no flaws were obvious.

More details about the manual inspection and the specific findings in the 18 classes can be found in Appendix F.

As the application on JSweet showed, a manual, labor-intensive source code inspection can be accelerated: most of the classes that were highlighted as bad by the vectorizing model indeed contained flaws even if they did not impact the maintainability devastatingly. Thus, the false positive rate in terms of highlighting weak spots in source code is found to be low. The two classes that were actually found to be highlighted totally wrong are similar to the extent that they both contain *switch* statements. Apparently, these statements only occurred for bad classes in the training dataset of Subsection 9.1.2. This might have led to overfitting so that all classes with *switch* statements are predicted to be bad.

[3] `https://github.com/cincheo/jsweet`; hash of commit: 2e8800dedb4c8e9f7dd1b64e8889e5ab208c6b15
[4] `http://www.jsweet.org/`

Even though overfitting apparently occurred due to the tiny dataset for the model optimization, the trained model of the experiment can be applied for health checks. This application illustrates the applicability of the vectorizing model on class level and demonstrated the successful vectorization of maintainability related information. Thus, RQ2 and RQ3 of Chapter 3 were answered for the class level. Additionally, the successful maintainability forecasting in the field of health checks not only realizes our motivation but also demonstrates that our approach is able to capture characteristics of source code as a human would do.

10. Future Work

Even though Chapter 8 and Chapter 9 show some promising application areas for the approach of this thesis, the application of the vectorizing model is limited. By extending and generalizing the approach, additional use cases can be made possible. First of all, the current implementation of this approach is limited to Java versions up to Java 8. As a proof of concept, this limitation is acceptable. However, future work can explore alternative parsers to OpenJDK 8 in order to support all Java versions. The graph creation of Chapter 5 should then be adapted.

Not all health checks at itestra GmbH involve Java as programming language. To ensure a universal applicability of our methodology, the graph creation has to support other programming languages like *C++* or *C#*. Nevertheless, not only common object-oriented programming languages should be supported but also imperative languages like *C*, functional languages like *Haskell* or legacy languages like *COBOL*. The graph creation of Chapter 5 might highly vary for each of the aforementioned since each language and paradigm has its own characteristics.

Due to the time constraints for creating this thesis, an intense evaluation of graph and model variations was not realized. This means that our evaluation – which assessed the graph creation and the vectorizing model in combination – can be complemented with studies that replace or adapt the methodology of Chapter 5 or Chapter 6. For instance, the vectorizing model could be compared to other ML algorithms that cope with graphs. The results answer then the question whether the vectorizing model – which evolved from continuous adaptions for increased performance – is the best choice in order to learn from the graphs of Chapter 5. On the other hand, the graph creation could also be altered by means of an ablation study so that the performance influences due to graph extensions or reductions become obvious. This evaluation of adaptions in future work would provide quantitative results for determining the best methodology in the context of a specific application.

As explained in Section 4.3, our approach assumes a supervised setting. Consequently, a training dataset has always to be supplied in order to adapt the vectorizing model. This can lead to small datasets due to the manual creation or datasets with inconsistent labels. Both scenarios occurred in Chapter 8 and Chapter 9 respectively. Generating training data or employing unsupervised algorithms can help to overcome these issues. For instance, approaches that are proposed for image data augmentation

can possibly adjusted for our ML setting. Several data augmentation solutions in image classification are evaluated by Perez et al. [PW17]. With these solutions, small datasets can be automatically extended. On the other hand, unsupervised algorithms can help to embed distinctive characteristics in vectors. As an example, Kipf et al. [KW16] propose to auto encode graphs with unsupervised learning. This or similar attempts of auto encoding graphs and the meaningfulness of the resulting software vectors could be explored in future work as well.

11. Conclusion

The goal of this thesis is the creation of fixed-length vectors from software. The vectors have to be meaningful so that they allow further processing with diverse machine learning algorithms. The goal is realized by proposing a methodology to learn a vector representation for Java source code elements. The elements comprise instances of different *source code levels*: methods, classes, packages and projects. The learning process is realized by transforming the instances to graphs first and applying a supervised machine learning algorithm on the graphs afterwards. The result of this process is a trained machine learning model.

The application of our approach demonstrated that the ML model is able to capture great parts of the method nomenclature in form of vectors. The resulting fixed-length representations of the methods were then successfully processed, e.g., with nearest neighbor algorithms. Further investigations included the training of the model with the expertise of code reviewers. To that effect, Java classes that impact the maintainability were successfully detected during the analysis of the JSweet project. Hence, our methodology is not only applicable for vectorizing source code but also for predicting software characteristics from the vectors. Thus, the application showed that the proposed methodology is indeed capable of learning meaningful vector representations for source code elements.

Whether the method or class level are better suited for the software analysis, could not be answered since the particular experiments involved single label and multilabel classification respectively which are incomparable. However, both granularities revealed information that is helpful for an automated software analysis. Thus, the method as well as the class level are both suitable for analyses based on the vectorized source code elements.

To that effect, the flexible computation of vectors for different granularities in the source code allows the tuning towards a broad variety of application areas. This also includes health checks at itestra GmbH. This means that our work not only successfully realizes the initial goal but also ensures the applicability for our original motivation.

A. List of Abbreviations

AST	Abstract Syntax Tree
CNN	Convolutional Neural Network
GG-NN	Gated Graph Neural Network
GRU	Gated Recurrent Unit
IR	Intermediate Representation
JDK	Java Development Kit
LOC	Lines of Code
LSTM	Long Short-Term Memory
ML	Machine Learning
NL	Natural Language
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SVM	support vector machine
t-SNE	T-distributed Stochastic Neighborhood Embedding

B. OpenJDK 8 AST Nodes

Information about the Java language and its structure can be found in the language specification[1]. The AST nodes were extracted from *com.sun.tools.javac.tree.JCTree* in the official OpenJDK 8 repository[2]: all nested classes that inherit from *com.sun.tools.javac.tree.JCTree* and are not marked as abstract are considered as possible nodes. The following table is the result[3]:

Node Name	*Description*	*Comment*
JCAnnotatedType		
JCAnnotation		
JCArrayAccess	An array selection	
JCArrayTypeTree	An array type, A[]	
JCAssert	An assert statement.	
JCAssign	A assignment with "=".	
JCAssignOp	An assignment with "+=", "|=" ...	
JCBinary	A binary operation.	
JCBlock	A statement block.	
JCBreak	A break from a loop or switch.	
JCCase	A "case :" of a switch.	
JCCatch	A catch block.	
JCClassDecl	A class definition.	Enums, interfaces and annotations are also considered to be classes.
JCCompilationUnit	Everything in one source file is kept in a JCCompilationUnit structure.	JCCompilationUnit is the root node of a source file.
JCConditional	A ( ) ? ( ) : ( ) conditional expression	
JCContinue	A continue of a loop.	
JCDoWhileLoop	A do loop	
JCEnhancedForLoop	The enhanced for loop.	
JCErroneous		
JCExpressionStatement	an expression statement	
JCFieldAccess	Selects through packages and classes	

[1] `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`

[2] `https://hg.openjdk.java.net/jdk8/jdk8/langtools`

[3] The descriptions are extracted from the documentation of *com.sun.tools.javac.tree.JCTree* whereas the comments were added by ourselves for additional information.

JCForLoop	A for loop.	
JCIdent	An identifier	
JCIf	An "if ( ) else " block	
JCImport	An import clause.	
JCInstanceOf	A type test.	
JCLabeledStatement	A labelled expression or statement.	
JCLambda	A lambda expression.	
JCLiteral	A constant value given literally.	
JCMemberReference	Selects a member expression.	
JCMethodDecl	A method definition.	Constructors are also considered to be methods.
JCMethodInvocation	A method invocation	
JCModifiers		
JCNewArray	A new[...] operation.	
JCNewClass	A new(...) operation.	
JCParens	A parenthesized subexpression (...	
JCPrimitiveTypeTree	Identifies a basic type.	
JCReturn	A return statement.	
JCSkip	A no-op statement ";".	
JCSwitch	A "switch ( ) " construction.	
JCSynchronized	A synchronized block.	
JCThrow	A throw statement.	
JCTry	A "try catch ( ) finally " block.	
JCTypeApply	A parameterized type, T<...>	
JCTypeCast	A type cast.	
JCTypeIntersection	An intersection type, T1 & T2 & ...	
JCTypeParameter	A formal class parameter.	
JCTypeUnion	A union type, T1 | T2 | ...	
JCUnary	A unary operation.	
JCVariableDecl	A variable definition.	
JCWhileLoop	A while loop	
JCWildcard		
LetExpr	(let int x = 3; in x+2)	This is a node created by OpenJDK for handling autoboxing.
TypeBoundKind		

C. Method Level Model Configuration

Several hyperparameters are mentioned in Chapter 6. For the experiment in Section 8.1, the hyperparameters that emerged for an optimal property prediction are listed here[1].

- The hidden dimensions are set as follows:

$$n_{graph} = 96$$

$$n_{edge} = 16$$

- The number of vertex vector update iterations is set to $k = 4$.
- The vocabularies were limited to the following percent thresholds:

$$t_{vertex} = 0.98$$

$$t_{edge} = 1.00$$

$$t_{property} = 0.90$$

- The number of nodes in a graph is restricted by $t_{\#nodes} = 200$.
- The count of labels per node is limited to 10, whereas the count of graph properties is restricted to 15.
- The maximum number of weight matrices is set by the following parameters:

$$t_{\#aggr} = 2$$

$$t_{\#att} = 2$$

$$t_{\#inc} = 1$$

[1]For the optimization of the ML model, a machine with 32GB RAM was used.

D. Method Similarity in Guava

For the method similarity search, the method vectors were calculated with the trained model of Subsection 8.2.2. By choosing a base method, similar methods in Guava can be identified by comparing the method vectors. The metric for the similarity is the cosine distance.

The method *put(K,V)* from the class *com.google.common.collect.CompactHashMap* was chosen as starting point. The following two methods were the closest ones to *put(K,V)*:

- *replace(K,int,V,V)* in *com.google.common.cache.LocalCache.Segment*. It has a cosine distance of 0.085 to *put(K,V)*.
- *storeLoadedValue(...)* in *com.google.common.cache.LocalCache.Segment*. It has a cosine distance of 0.088 to *put(K,V)*.

The source code of the respective methods is printed below.

put(K,V)

```
public @Nullable V put(@Nullable K key, @Nullable V value) {
  long[] entries = this.entries;
  Object[] keys = this.keys;
  Object[] values = this.values;
  int hash = smearedHash(key);
  int tableIndex = hash & hashTableMask();
  int newEntryIndex = this.size; // current size, and pointer to the entry to be
      appended
  int next = table[tableIndex];
  if (next == UNSET) {
    table[tableIndex] = newEntryIndex;
  } else {
    int last;
    long entry;
    do {
      last = next;
      entry = entries[next];
      if (getHash(entry) == hash && Objects.equal(key, keys[next])) {
        @SuppressWarnings("unchecked") // known to be a V
        @Nullable
        V oldValue = (V) values[next];
        values[next] = value;
        accessEntry(next);
        return oldValue;
      }
      next = getNext(entry);
    } while (next != UNSET);
    entries[last] = swapNext(entry, newEntryIndex);
  }
  if (newEntryIndex == Integer.MAX_VALUE) {
    throw new IllegalStateException("Cannot␣contain␣more␣than␣Integer.MAX_VALUE␣elements!
        ");
  }
  int newSize = newEntryIndex + 1;
  resizeMeMaybe(newSize);
  insertEntry(newEntryIndex, key, value, hash);
  this.size = newSize;
  if (newEntryIndex >= threshold) {
    resizeTable(2 * table.length);
  }
  modCount++;
  return null;
}
```

replace(K,int,V,V)

```
boolean replace(K key, int hash, V oldValue, V newValue) {
  lock();
  try {
    long now = map.ticker.read();
    preWriteCleanup(now);
    AtomicReferenceArray<ReferenceEntry<K, V>> table = this.table;
    int index = hash & (table.length() - 1);
    ReferenceEntry<K, V> first = table.get(index);
    for (ReferenceEntry<K, V> e = first; e != null; e = e.getNext()) {
      K entryKey = e.getKey();
      if (e.getHash() == hash
          && entryKey != null
          && map.keyEquivalence.equivalent(key, entryKey)) {
        ValueReference<K, V> valueReference = e.getValueReference();
        V entryValue = valueReference.get();
        if (entryValue == null) {
          if (valueReference.isActive()) {
            // If the value disappeared, this entry is partially collected.
            int newCount = this.count - 1;
            ++modCount;
            ReferenceEntry<K, V> newFirst =
                removeValueFromChain(
                    first,
                    e,
                    entryKey,
                    hash,
                    entryValue,
                    valueReference,
                    RemovalCause.COLLECTED);
            newCount = this.count - 1;
            table.set(index, newFirst);
            this.count = newCount; // write-volatile
          }
          return false;
        }
        if (map.valueEquivalence.equivalent(oldValue, entryValue)) {
          ++modCount;
          enqueueNotification(
              key, hash, entryValue, valueReference.getWeight(), RemovalCause.REPLACED);
          setValue(e, key, newValue, now);
          evictEntries(e);
          return true;
        } else {
          // Mimic
          // "if (map.containsKey(key) && map.get(key).equals(oldValue))..."
          recordLockedRead(e, now);
```

```
          return false;
        }
      }
    }
    return false;
  } finally {
    unlock();
    postWriteCleanup();
  }
}
```

storeLoadedValue(K,int,LoadingValueReference<K,V>,V)

```
boolean storeLoadedValue(K key, int hash, LoadingValueReference<K, V> oldValueReference,
      V newValue) {
  lock();
  try {
    long now = map.ticker.read();
    preWriteCleanup(now);
    int newCount = this.count + 1;
    if (newCount > this.threshold) { // ensure capacity
      expand();
      newCount = this.count + 1;
    }
    AtomicReferenceArray<ReferenceEntry<K, V>> table = this.table;
    int index = hash & (table.length() - 1);
    ReferenceEntry<K, V> first = table.get(index);
    for (ReferenceEntry<K, V> e = first; e != null; e = e.getNext()) {
      K entryKey = e.getKey();
      if (e.getHash() == hash
          && entryKey != null
          && map.keyEquivalence.equivalent(key, entryKey)) {
        ValueReference<K, V> valueReference = e.getValueReference();
        V entryValue = valueReference.get();
        // replace the old LoadingValueReference if it's live, otherwise
        // perform a putIfAbsent
        if (oldValueReference == valueReference
            || (entryValue == null && valueReference != UNSET)) {
          ++modCount;
          if (oldValueReference.isActive()) {
            RemovalCause cause =
                (entryValue == null) ? RemovalCause.COLLECTED : RemovalCause.REPLACED;
            enqueueNotification(key, hash, entryValue, oldValueReference.getWeight(),
                cause);
            newCount--;
          }
          setValue(e, key, newValue, now);
```

```
          this.count = newCount; // write-volatile
          evictEntries(e);
          return true;
        }
        // the loaded value was already clobbered
        enqueueNotification(key, hash, newValue, 0, RemovalCause.REPLACED);
        return false;
      }
    }
    ++modCount;
    ReferenceEntry<K, V> newEntry = newEntry(key, hash, first);
    setValue(newEntry, key, newValue, now);
    table.set(index, newEntry);
    this.count = newCount; // write-volatile
    evictEntries(newEntry);
    return true;
  } finally {
    unlock();
    postWriteCleanup();
  }
}
```

E. Class Level Model Configuration

Several hyperparameters are mentioned in Chapter 6. For the experiment in Section 9.1, the hyperparameters that emerged for an optimal maintainability prediction are listed here[1].

- The hidden dimensions are set as follows:

$$n_{graph} = 40$$

$$n_{edge} = 8$$

- The number of vertex vector update iterations is set to $k = 4$.
- The vocabularies were limited to the following percent thresholds:

$$t_{vertex} = 0.90$$

$$t_{edge} = 1.00$$

$$t_{property} = 1.00$$

- The number of nodes in a graph is restricted by $t_{\#nodes} = 1250$.
- The count of labels per node is limited to 10. The count of graph properties is always 1 since the decoder for single label classification is employed.
- The maximum number of weight matrices is set by the following parameters:

$$t_{\#aggr} = 1$$

$$t_{\#att} = 1$$

$$t_{\#inc} = 1$$

[1]For the optimization of the ML model, a machine with 32GB RAM was used.

F. Findings in JSweet

During the evaluation in Section 9.2, the vectorizing model highlighted 18 classes of JSweet as *C*. This means that these classes should be hard to maintain [Sch+19]. The 18 classes and their maintainability flaws are listed here. The flaws were identified with a manual inspection by ourselves. Thus, the following judgment is subject to our intuition.

org.jsweet.transpiler.extension.Java2TypeScriptAdapter

This class was found to be indeed part of the category *C*. The maintainability is heavily impacted by the length of the class (> 1500 LOC) and the complexity of its methods. On method contains even more than 900 LOC. The statement nesting depth in the methods additionally reduces the comprehensibility.

org.jsweet.transpiler.Java2TypeScriptTranslator

This class was also found to be correctly labeled as *C*. It contains over 5800 LOC. This causes that a high effort is needed to change the class without unwanted side-effects.

org.jsweet.transpiler.JSweetTranspiler

Again, this class was correctly categorized as *C*. The size of 1800 LOC can cause severe maintainability issues. Even though the size of the contained methods is moderate, the amount of methods and their statement nesting depth reduces the comprehensibility as well as the maintainability.

org.jsweet.transpiler.OverloadScanner

The class should be part of the category *C*. Even though the size of the class is acceptable (< 500 LOC), the high statement nesting depth severely impacts the maintainability.

org.jsweet.transpiler.extension.RemoveJavaDependenciesAdapter

The label *C* was predicted correctly. The size of nearly 1800 LOC encourages this prediction. Furthermore, the categorization is strengthened by the heavy use of *case* or

else if statements in the methods.

org.jsweet.transpiler.util.Util

The class should be labeled as *C* again. Reasons are the size of the class (> 1500 LOC) and the amount of methods (close to 80 methods).

org.jsweet.transpiler.util.JSDoc

The class is indeed found to be part of category *C*. The complex methods impact the maintainability.

org.jsweet.transpiler.JSweetContext

Due to its length (> 1800 LOC) and its high number of methods, the class is correctly labeled as *C*.

org.jsweet.JSweetCommandLineLauncher

The class does not severely impact the maintainability, it is considered to be labeled as *B*. The definition and processing of many command line arguments is confusing and could be improved.

org.jsweet.transpiler.extension.PrinterAdapter

The class should be rather part of category *B* than *C*. Even though the class size (> 1000 LOC) and the high amount of methods reduces the maintainability, the contents are fairly easy to comprehend and change.

org.jsweet.transpiler.util.AbstractTreeScanner

The label *B* is appropriate for this class. The moderately high statement nesting depth of many methods makes the class unnecessarily complicated.

org.jsweet.transpiler.eval.JavaEval

The class consists of one method only. The content of the method is rather complex but does not heavily impact the maintainability. Since there is room for improvement, the label *B* is appropriate.

org.jsweet.transpiler.eval.JavaScriptEval

The statement nesting depth decreases the comprehensibility of this class slightly. Thus, it is considered to be part of category *B*.

org.jsweet.transpiler.extension.MapAdapter

The use of a *switch* with many *case* statements slightly decreases the maintainability. Since no other flaws are apparent, this class is found to be labeled as *B*.

org.jsweet.JSweetFileWatcher

This class should also be labeled as *B*: nested *try-catch* bodies and *for{::}* loops impact the maintainability.

org.jsweet.transpiler.extension.BigDecimalAdapter

The use of multiple *case* statements with separate *return* statements causes this class to be labeled as *B*.

org.jsweet.transpiler.JSweetDiagnosticHandler

This class should be labeled as *A*. No flaws are apparent.

org.jsweet.transpiler.model.support.UtilSupport

Since no flaws are obvious, this class should be labeled as *A* as well.

List of Figures

4.1. The vectorizing model is the focus of this thesis. Machine learning models that are applied to the vectors are indirectly applied to the software. 8
4.2. The vectorizing model is based on the encoder-decoder framework that connects the encoder and the decoder by a fixed-length vector. During the training phase of the model, the encoder and decoder are optimized according to the given combinations of graphs and properties. For the vectorization of a graph, the decoder is omitted. 11

5.1. An exemplary method. 14
5.2. The AST is the basis for the method graph. In this figure, the AST of the method in Figure 5.1 is illustrated as a directed graph. 15
5.3. The final graph for the method in Figure 5.1. Nodes and edges that were added to the graph in Figure 5.2 are highlighted in color. 16
5.4. An exemplary class. 18
5.5. The figure shows the truncated graph after the transformation step for the class in Figure 5.4. The children of the blue *JCBlock* were cut away. Some other subgraphs were omitted due to their irrelevance. The omittance is indicated by ellipses. As constructors are also referred to as methods in the OpenJDK, the leftmost *JCMethodDecl* node is the constructor of the class. 19

6.1. A part of the method graph of Figure 5.3 is shown. Omitted subgraphs are indicated with ellipses. 25
6.2. The initial hidden representations were assigned to the nodes of Figure 6.1. The edges have a latent vector as well. The indices enhance the distinction of different values but have no deeper meaning. 26
6.3. The figure illustrates the aggregation of the forward and backward neighborhood of a vertex. The graph is the one shown in Figure 6.2. Vertex v_4 was chosen exemplarily. 29

7.1. The figure illustrates the most relevant classes and their dependencies in the graph creation program. Packages are illustrated with dotted rectangles whereas all other rectangles represent a class. The dashed arrows indicate inheritance. All other arrows indicate that a method from the destination class is directly called in the source class. 40
7.2. Each Python source file is illustrated as a separate rectangle. The arrows indicate whether contents of a file are imported in another file. 42

8.1. The figure shows the graph of Figure 5.3 with omitted subgraphs. For the evaluation on method names, the vertex highlighted in blue will be cut off. 47
8.2. Each point in the figure represents a method of the Guava library. The position of the points is determined by t-SNE which was applied on the method vectors. The orange rectangle highlights some of many point accumulations. Please note that the axes and their scales are figured out by t-SNE and have no deeper meaning. 54

9.1. The distribution of the labels *A*, *B* and *C* in the parsed classes. This distribution is preserved in the particular datasets. The datasets are shown in Table 9.1. 59

List of Tables

6.1. Exemplary distribution of vertex labels for a training set. The labels are sorted according to their frequency. The percentiles sum up the percentage of the current label and all labels that are more or equally frequent. 35

8.1. The table enumerates information about the respective datasets. The collected Java projects stem from similar application areas in order to allow a fair evaluation. 49
8.2. The table highlights the prediction scores for the dataset of Table 8.1. . 51

9.1. The table lists the amount of labels in each dataset. The classes were split 50-30-20 into training, validation and test set. The percentage of labels is preserved in each dataset. 59
9.2. The table shows the distribution of predictions with absolute numbers for the test set of Table 9.1. It highlights which labels were predicted for each category. 60

Bibliography

[Aba+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A System for Large-Scale Machine Learning." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 265–283. ISBN: 978-1-931971-33-1.

[ABK17] M. Allamanis, M. Brockschmidt, and M. Khademi. "Learning to Represent Programs with Graphs." In: *CoRR* abs/1711.00740 (2017). arXiv: 1711.00740.

[All+15] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei. "Bimodal Modelling of Source Code and Natural Language." In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 2123–2132.

[Alo+18a] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. "A General Path-Based Representation for Predicting Program Properties." In: *CoRR* abs/1803.09544 (2018). arXiv: 1803.09544.

[Alo+18b] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. "code2vec: Learning Distributed Representations of Code." In: *CoRR* abs/1803.09473 (2018). arXiv: 1803.09473.

[ALY18] U. Alon, O. Levy, and E. Yahav. "code2seq: Generating Sequences from Structured Representations of Code." In: *CoRR* abs/1808.01400 (2018). arXiv: 1808.01400.

[APS16] M. Allamanis, H. Peng, and C. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code." In: *International Conference on Machine Learning (ICML)*. 2016.

[BCB14] D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." In: *CoRR* abs/1409.0473 (2014). arXiv: 1409.0473.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[BJH18] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics." In: *CoRR* abs/1806.07336 (2018). arXiv: 1806.07336.

[BKV09] R. Bell, Y. Koren, and C. Volinsky. "Matrix Factorization Techniques for Recommender Systems." In: *Computer* 42.08 (Aug. 2009), pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263.

[BP88] B. W. Boehm and P. N. Papaccio. "Understanding and controlling software costs." In: *IEEE Transactions on Software Engineering* 14.10 (Oct. 1988), pp. 1462–1477. ISSN: 0098-5589. DOI: 10.1109/32.6191.

[Cho+14] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078.

[Chu+14] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555.

[Dei+10] F. Deißenböck, L. Heinemann, B. Hummel, and E. Juergens. "Flexible Architecture Conformance Assessment with ConQAT." In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 247–250. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810343.

[DP05] F. Deißenböck and M. Pizka. "Concise and consistent naming [software system identifier naming]." In: *13th International Workshop on Program Comprehension (IWPC'05)*. May 2005, pp. 97–106. DOI: 10.1109/WPC.2005.14.

[DTP16] H. K. Dam, T. Tran, and T. Pham. "A deep language model for software code." In: *CoRR* abs/1608.02715 (2016). arXiv: 1608.02715.

[Haw04] D. Hawkins. "The Problem of Overfitting." English (US). In: *Journal of Chemical Information and Modeling* 44.1 (Jan. 2004), pp. 1–12. ISSN: 1549-9596. DOI: 10.1021/ci0342472.

[Hen+18] J. Henkel, S. Lahiri, B. Liblit, and T. W. Reps. "Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces." In: *CoRR* abs/1803.06686 (2018). arXiv: 1803.06686.

[HS97] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory." In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[Jai+16] A. Jain, A. R. Zamir, S. Savarese, and A. Saxena. "Structural-RNN: Deep Learning on Spatio-Temporal Graphs." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 5308–5317. DOI: 10.1109/CVPR.2016.573.

[KB14] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization." In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980.

[KKP07] S. B. Kotsiantis, D. Kanellopoulos, and P. E. Pintelas. "Data Preprocessing for Supervised Leaning." In: *International Journal of Computer, Electrical, Automation, Control and Information Engineering* 1.12 (2007), pp. 4104–4109. ISSN: eISSN:1307-6892.

[KW16] T. N. Kipf and M. Welling. "Variational Graph Auto-Encoders." In: *CoRR* abs/1611.07308 (2016). arXiv: 1611.07308.

[Li+15] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. "Gated Graph Sequence Neural Networks." In: *CoRR* abs/1511.05493 (2015). arXiv: 1511.05493.

[LPM15] M. Luong, H. Pham, and C. D. Manning. "Effective Approaches to Attention-based Neural Machine Translation." In: *CoRR* abs/1508.04025 (2015). arXiv: 1508.04025.

[MH08] L. van der Maaten and G. Hinton. *Visualizing data using t-SNE*. 2008.

[Mik+13] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. "Distributed Representations of Words and Phrases and their Compositionality." In: *CoRR* abs/1310.4546 (2013). arXiv: 1310.4546.

[Ped+11] 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. Duchesnay. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PW17] L. Perez and J. Wang. "The Effectiveness of Data Augmentation in Image Classification using Deep Learning." In: *CoRR* abs/1712.04621 (2017). arXiv: 1712.04621.

[RM87] D. E. Rumelhart and J. L. McClelland. "Distributed Representations." In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. MITP, 1987. ISBN: 9780262291408.

[Sch+19] M. Schnappinger, M. H. Osman, A. Pretschner, and A. Fietzke. *Learning a Classifier for Prediction of Maintainability based on Static Analysis Tools*. 2019.

[SS01] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001. ISBN: 0262194759.

[ST04] A. Shalloway and J. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design (2Nd Edition) (Software Patterns Series)*. Addison-Wesley Professional, 2004. ISBN: 0321247140.

[Xu+15] B. Xu, N. Wang, T. Chen, and M. Li. "Empirical Evaluation of Rectified Activations in Convolutional Network." In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853.

[Xu+18] K. Xu, L. Wu, Z. Wang, Y. Feng, and V. Sheinin. "Graph2Seq: Graph to Sequence Learning with Attention-based Neural Networks." In: *CoRR* abs/1804.00823 (2018). arXiv: 1804.00823.