Improving Software Fault Prediction Using a Data-Driven Approach

André Sobral Gonçalves
andre.sobral@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

Setembro, 2019

Abstract

In software development, debugging is one of the most expensive tasks in the life cycle of a project. Repetitive, manual identification of faults in software, which are the source of errors and inefficiencies, is inherently time-consuming and error-prone. The solution to this problem is automated software diagnosis. This approach solves the issues mentioned previously as long as the accuracy of the diagnosis is credible. There are different tools and techniques which ensure this diagnosis, but they neglect valuable data. One example of such data are the revisions of source files stored in a revision control systems. Such data is replicated in repositories that are tagged and organized so that they can be mined and analyzed. Examples of this include the Bugs.jar dataset. This data can potentially improve the diagnostic process. This thesis aims to describe the implementation of a solution in this field that leverages innovative deep learning techniques to learn semantic structures of source files, in order to effectively learn to classify them. We develop an approach based on convolutional neural networks, a class of deep neural networks, that has been applied successfully on image processing and can be interesting to apply to this domain. This solution follows a data-driven approach, i.e. data as the principal factor to guide strategic decisions, with the aim to optimize a fault prediction algorithm that is able to learn effectively from source code.
Keywords: debugging; software faults; software diagnosis; data; deep learning.

1. Introduction

Locating software components which are responsible for faults, i.e. statements, methods or classes that are erroneous, is the most expensive, error-prone phase in the software development life cycle. This process is called debugging. In other words, it is the process of detecting and removing existing and potential errors in a software's code. As mentioned before, this is a complex procedure. When software modules become tightly coupled, any change in one module may cause bugs to appear in another. The higher the complexity of a project, the harder it becomes to debug.
The amount of source code underlying the systems that we use on a day-to-day basis is continuously growing. Combined with a practically constant rate of faults per line of code, this implies that system dependability is decreasing [1] . The automation of the diagnosis of these faulty components drastically improves the efficiency of the debugging process. This approach is key for the development of dependable software both in academia and business because it can reduce the effort spent on manual debugging, which shortens the test-diagnose-repair cycle and leads to a shorter time-to-market. Although this topic is of fundamental importance nowadays, there is still room for significant improvements to this approach.

1.1. Goals of the Developed Work

The main motivation behind this work is to innovate in the area of bug prediction and software component reliability estimation by taking advantage of previously unused data and machine learning techniques. As of today, a great deal of valuable information is currently not being exploited by most automatic testing and debugging techniques. This is where the expression data-driven comes into play. Our bug prediction model will be directly influenced by the dataset chosen. This model will make strategic decisions based on the analysis and interpretation of this unused data. For instance, there may be interesting information in commits to a revision control system, such as github or svn, or to an issue tracking system. One example of this are the revisions of every source file stored in a revision control system. It is common that a bug occurs in a source file that was recently revised/altered.
This yet unexploited data can be used to train

a machine learning model to learn how to better identify faulty components within IT projects. This model may be able to discover errors that are not yet being detected by current techniques, and we will test this hypothesis by utilizing a deep learning architecture that has yet to be applied to the bug prediction domain. This model will learn from the collected data how to predict which components are expected to be faulty. It learns which source code less are likely to fail from past observed faults and previous source code revisions.

2. Related Work

2.1. Software Fault Diagnosis

Software diagnosis can be defined as the process which identifies the software components that originate faults in the system, isolates them in the project's architecture and produces a diagnosis, i.e. an explanation as to why the fault occurred in the first place. But first, it is crucial to define the notions of failure and fault in order to better understand what a diagnosis is. A failure is a discrepancy between expected and observed behavior, and a fault is a property of the system that causes such a discrepancy. The purpose of a diagnosis is to identify the fault which is the root cause of a failure. Applying this terminology to computer programs, we can define faults as bugs in the program code and failures as an output that deviates from the expected output. There are mainly two approaches that are considered in this field of study, which we will now detail.

2.1.1 Model-Based Diagnosis

This is an approach to automated diagnosis that uses a model of the diagnosed system to infer possible diagnoses, i.e. possible explanations of the observed system failure. The essence of model-based diagnosis, MBD for short, is to associate every component of the system to a variable that captures its health state, labeling it either healthy or broken. The MBD-based methods then search for an assignment of values to these variables that logically explain the observed behavior. Together with a set of observations, this model can directly serve as input to a debugging framework that automatically derives a sequence of assignments to the health variables which are consistent with the observations, thus generating a diagnosis of a fault.
A formal way to define an MBD method, as suggested by Elmishali, A., et al [2], is by defining a tuple ¡System Description, Components, Observations¿:

- System Description: a formal description of the diagnosed system behavior;
- Components: the set of software components that may be faulty;
- Observations: the output of executing a set of tests.

Although this approach has been applied successfully to digital circuits and complex mechanical systems [1], that rely on truth tables and well-defined paths of execution, this same development has not happened in software projects. It is harder to label a component as being entirely healthy or broken, since the degree of complexity of the tasks performed in such projects is much higher.

2.1.2 Spectrum Fault Localization

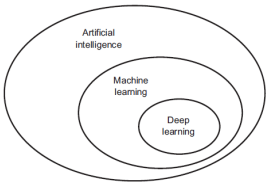
In this approach, there is no need for a logical model of every software component in the system. Spectrum fault localization, SFL for short, is well-suited for diagnosing software systems, and can easily be integrated with existing testing schemes. It only requires the software specifications of the project under test. These specifications typically cover the expected behavior without revealing the internal composition of the project. This naturally requires much less effort than devising a thorough logical model, and since it is not required to specify the internal composition, it is suited for testing proprietary software from third parties. This technique considers traces of executions and finds diagnoses by considering the correlation between the traces that are being executed and prior executions that have failed [2]. To accomplish this, it requires the capture of the run-time information of the software. Executing a test suite of a software package usually identifies several test cases for which the system behaves according to the specification, i.e. passed runs, as well as a number of test cases for which this is not the case, i.e. failed runs. Usually, the testing framework will tell us which parts of the system were involved in the execution of these test cases. This information can be obtained via standard profiling techniques and constitutes a so-called program spectrum per test case [1].
A more detailed definition would be that the program spectrum is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consists of a number of counters or flags for the different parts of a program [1]. It is also important to clearly define what a test trace is. A trace of a test is the full or partial sequence of components involved in running said test.

2.2. Artificial Intelligence

The domain of artificial intelligence is very broad and diverse (see Fig. 1). This discipline is defined

by the study of intelligent agents. These are considered to be any device that perceives its environment and takes actions that maximize its ability to perform a given task. Thus, it applies to a machine that is able to learn and solve a specified task. AI can be summed up by the quote: *the effort to automate intellectual tasks normally performed by humans*[4]. As such, it is a general field that encompasses machine learning and deep learning, but that also includes many more approaches.

Figure 1: IA overview)



2.3. Machine Learning

This field of study focuses on the study of algorithms and mathematical models that computer systems use to progressively improve their performance on a specific task. It stemmed from the intuition: could a computer go beyond the data-processing rules crafted by programmers and learn on its own how to perform a specified task? Could a computer automatically learn these rules by looking at data, by building a mathematical model of it in order to make predictions?

This field approaches problems and challenges differently. In classical programming, the user inputs a set of rules, i.e. a program, and sources data that is to be processed according to these rules and an output is produced. With machine learning, the user inputs data as well as the expected output of the data and the output is the set of rules (see Fig. 2 for an overview on this). These rules can then be applied to new data to produce new knowledge. Thus, a machine-learning system is trained rather than being explicitly programmed. It's presented data relevant to a task, defined as the training set, and it finds statistical structure in these examples that allow the system to make decisions [4].

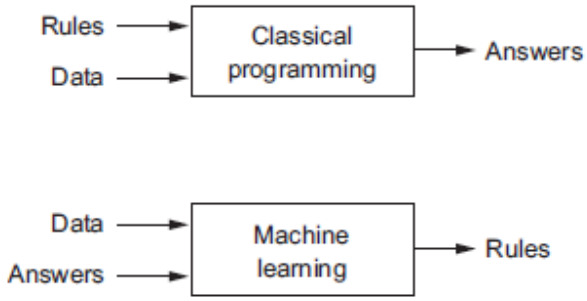
To do machine learning, there are three essential concepts:

The input data: For instance, if the task is speech recognition, these data points could be sound files of people speaking;

The expected output: In a speech-recognition task, these could be human generated transcripts of sound files. In an image task, expected outputs could be tags such as "dog", "cat" and so on;

Having an Evaluation Model: A way to measure whether the algorithm is performing well.

Figure 2: Machine Learning Paradigm



The evaluation of a machine learning model is crucial. This feedback is what enables the algorithm to assert how well it's doing by calculating a distance between its output and the expected output, i.e. an error estimation. According to this value, the algorithm will adjust its parameters in order to shorten this distance or minimize the error. This adjustment step is called learning [4].

A machine-learning model transforms its input data into meaningful outputs, a process that is learned from exposure to known examples of inputs and outputs. Therefore, the central problem in machine learning and deep learning is to meaningfully transform data [4].

In this thesis, we leverage machine learning techniques capable of greater knowledge representation, ability to learn, natural language processing and classication, to extract invaluable information from software artifacts generated during software development, including code revision history, bug reports, history bugs, patches and source code, to improve existing software reliability practices [5].

2.4. Deep Learning

These algorithms are part of a family of machine learning methods specialized in feature learning. Feature learning is a set of techniques that automatically discover the data representations needed for feature detection or classification algorithms. This technique can be seen as a new take on learning representations from data focused on learning successive layers of increasingly meaningful representations [3].

The "deep" in deep learning stands for this idea of successive layers of representations [3]. The number of layers defines the depth of a model. Modern deep learning often involves tens or even hundreds of successive layers of representations and they're all learned automatically from exposure to training data. Meanwhile, other machine learning approaches tend to focus on learning only one or two layers. As in most machine learning models, learning can be supervised (e.g. Classification

problems) or unsupervised (e.g. Pattern analysis). There are various examples of Deep Learning architectures, such as: Deep Neural Networks; Deep Belief Networks; Stacked Auto-Encoders; Convolutional Neural Networks.
Each of these methods have common characteristics that group them as a family:

1. The usage of a cascade of multiple layers of non-linear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as input [4].

2. The learning of data representations, i.e. the ability to learn multiple levels of representations that correspond to the different levels of abstraction [4].

2.4.1 Motivation

Deep learning techniques were applied to improve existing statistical defect prediction models. This approach also aims to consequently reduce software development costs by improving bug detection thus minimizing the time of enhancing source code quality [5].
Previously, studies relied in manually designed features, that encode the characteristics of programs. However, these features often failed to capture the semantic differences between programs, and such a capability is key in building capable bug prediction models. To minimize the gap between code semantics and defect prediction features, this work uses deep learning techniques to learn a semantic representation of programs automatically and build and train defect prediction models based on these newly extracted features.
The efforts of previous studies relied mainly on manually designing new features or new combinations of features to represent defects more effectively [5]. Researchers in this field have designed many quality features to distinguish defective les from non-defective les, such as:

- Halstead features, based on operator and operand counts;
- McCabe features, based on dependencies;
- CK features, based on function and inheritance counts;
- MOOD features, based on polymorphism factor, coupling factor, etc.;
- Process features including number of lines of code added/removed, meta features, etc.;
- Object-oriented features.

Traditional features mainly focus on the statistical characteristics of programs and assume that buggy and clean programs have distinguishable statistical characteristics. However, our observations on real-world programs show that existing traditional features often cannot distinguish programs with different semantics. Specically, program les with different semantics can have traditional features with similar or even the same values [5].

Figure 3: Code sample that illustrates why semantic features are important

```
1  public void copy(Directory to, String
       src, String dest) throws IOException
       {
2    IndexOutput os = to.createOutput(dest);
3    IndexInput is = openInput(src);
4    IOException priorException = null;
5
6    try {
7       is.copyBytes(os, is.length());
8    } catch (IOException ioe) {
9       priorException = ioe;
10   }
11   finally {
12      IOUtils.closeSafely(priorException
            , os, is);
13   }
14 }
```

(a) Original buggy code snippet.

```
1  public void copy(Directory to, String
       src, String dest) throws IOException
       {
2    IndexOutput os = null;
3    IndexInput is = null;
4    IOException priorException = null;
5    try {
6       os = to.createOutput(dest);
7       is = openInput(src);
8       is.copyBytes(os, is.length());
9    } catch (IOException ioe) {
10      priorException = ioe;
11   } finally {
12      IOUtils.closeSafely(priorException
            , os, is);
13   }
14 }
```

(b) Code snippet after fixing the bug.

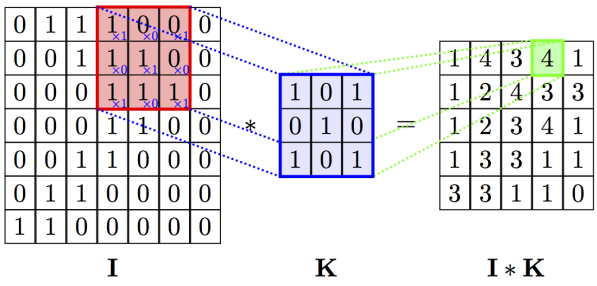
Fig.3 shows an original buggy version, Fig. 3 (a), and a xed version, Fig. 3 (b), of the source code.
In the buggy version, at line 2, there is an *IOException* that is thrown when the program tries to initialize the variables *os* and *is* outside of the try block. This error can lead to a memory leak or even a computation crash. This bug was fixed in the correct version by moving the initializing statements to the inside of the try block. This can be seen in Fig. 3(b), lines 6 and 7.
Using traditional features to represent these two code snippets, their feature vectors are identical [5]. This is due to these two code snippets having the same source code characteristics in terms of complexity, function calls and programming tokens. However, the semantic information of these two code snippets is signicantly different. Specically, of the case previously mentioned of the two variables, *os* and *is*. Features that can distinguish such semantic dierences are valuable because they hold information that is not being exploited. Such features can be used to build prediction models that are more accurate.

2.4.2 Convolutional Neural Networks

In deep learning, a convolutional neural network, CNN for short, is a class of deep neural networks that specializes in processing data that has a grid-like topology, such as image data in a 2D grid. This model is used for a multitude of tasks that include Image recognition, Image Analysis, Image Classification, Recommendation Systems and Natural Language Processing.
In this work, we leverage CNNs for effective feature generation from source. We use natural language

processing approaches and reasoning adapted to process source code.
A convolution could be thought of as a sliding window function applied to a matrix, see Fig.4 for integrated view on this concept. The function K is applied to each submatrix of the input I and each iteration results in an entry of the resulting matrix I*K. This resulting matrix is the new shape taken by I after applying the function K. In summary, a convolution is a mathematical operation on two functions to produce a resulting third function that expresses how the shape of the first is modified by the second.

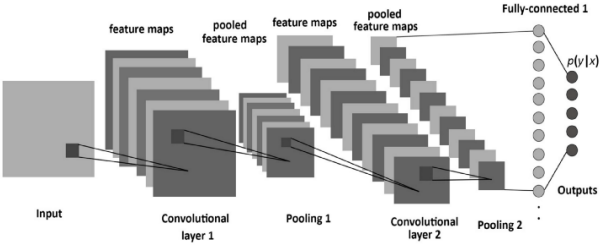
Figure 4: Convolution 'k' applied to matrix 'I'



A CNN is composed of several layers of convolutions with non-linear activation functions, such as the rectifier linear unit function, ReLU for short, or the hyperbolic tangent function, tanh for short. Rather than following the approach of a traditional neural network of using fully connected layers, CNNs boast sparse connectivity [6], meaning a node is not connected to every node of the following layer. The traditional "fully-connectedness" makes networks prone to overfitting data. Instead, CNNs take advantage of hierarchical patterns in data. They learn to assemble complex patterns by leveraging smaller, simpler patterns. This is done by using the process described earlier, convolutions.
CNNs compute convolutions over the input layer to compute the output. This results in local connections, where regions of the input connect to a neuron output. This local connectivity pattern between neurons of adjacent layers generates spatially local correlation of the inputs [6].
Fig. 5 depicts how a CNN is structured and how each layer communicates with the next, commonly referred to as the CNN pipeline.

The CNN pipeline is composed by the input data, convolutional layers, pooling layers and a final fully-connected layer. The objective of the Convolution Neural Network is to extract the high-level features. To do this, it utilizes several convolutional layers. Conventionally, the first convolutional layer is responsible for capturing simpler features, such as edges of objects or differences in color in image processing. When utilizing several layers, the model adapts to the high-level features as well, resulting in a network that perceives the data as a whole by learning each of its constituent parts.

Figure 5: Convolutional Neural Network Pipeline



Each layer applies several filters to the input data and combines the results. These filters are the non-linear activation functions, such as ReLU or tanh.
Shared weights between layers mean each lter shares the same parameterization, i.e. weight vector and bias. Replicating lters in this way enables us to detect features regardless of their position in the input vector. Moreover, weight sharing can greatly increase learning efciency by reducing the number of free parameters. CNNs sparse connectivity and shared weights, benet defect prediction by capturing local structural information of programs.
A Pooling layer is responsible for reducing the spatial size of the convolved feature. This is to decrease the computational effort required to process the data. There are two types of pooling, max pooling and average pooling. Max pooling returns the maximum value from the portion of the image, whereas average pooling returns the average of all the values from the portion of the image.
Max pooling has the distinct advantage that it deals with noisy data. It discards the noisy activations altogether along with dimensionality reduction. Hence, we max pooling is usually chosen for this pooling step.
Finally, adding a fully-connected layer is the usual approach to be able to learn non-linear combinations of the high-level features outputted by the previous sets of convolution and pooling layers. This final layer objective is to learn a non-linear function in order to be able to do classification. Now that we have converted our input data into a suitable form for our multi-level perceptron it is able to iteratively learn. Over a series of epochs, the model can, hopefully, distinguish different kinds of features and classify them using the softmax classification technique.
CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that were

traditionally hand-engineered. This independence from prior knowledge and human intervention is a major advantage. Ultimately, this makes CNN a prime candidate to learn features from source code, and its low connectivity makes its scalable and faster than other alternatives.
A major plus for CNNs is that they are fast and scalable. Convolutions are a central part of computer graphics and implemented on a hardware level on GPUs. Compared to something like other typical NLP approaches such as n-grams, CNNs are also efficient in terms of representation. Convolutional Filters learn good representations automatically, without needing to represent the whole vocabulary. The effectiveness of CNN's largely depends on its parameter's settings. These include parameters such as Iter length or batch size. This aspect of the algorithm completely changes how the algorithm performs. Under a completely wrong parameter setting the model might not even converge to a solution.

2.5. Bugs.jar

Bugs.jar is a large-scale dataset for research in automated debugging, patching, and testing of Java programs. It is comprised of 1,158 bugs and patches, sourced from 8 popular open-source Java projects, of 8 different application categories. It is an order of magnitude larger than another popular debugging dataset Defects4J [7].
The data elected to be part of this dataset was chosen from active projects, with a rich development history. This allows for evaluating software engineering techniques at scale. It also provides diverse database of bugs.
The projects were all maintained using a bug tracking system and have descriptive commit logs to facilitate identification of bug-fixing commits [7]. This limits errors and subjective bias introduced by manual examination in order to identify and document bugs.
In this task, it is being used to research software fault diagnosis techniques. For each bug in the dataset we have [7]: The buggy version of the source code; A bug report describing the nature of the bug; A test-suite, serving as a correctness specification, comprising at least one failing (fault-revealing) test case and one passing test case (to guard against regression); The developer's patch to fix the bug, which passes all the test cases.

2.6. JavaParser

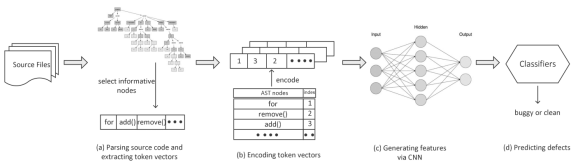
In its simplest form, the JavaParser library [8] allows you to interact with Java source code as a Java object representation in a Java environment. In a formal manner, we refer to this object representation as an Abstract Syntax Tree, AST for short.
This library provides a mechanism to navigate the tree and select relevant nodes to our program. Everything in a java source file is modelled as a node, the responsibility is on the programmer to choose which data is relevant and which isn't.

3. The Solution

3.1. Solution Description

In order to try to improve the current defect prediction approaches, we will take advantage of deep learning models to automatically extract and learn features from source code and obtain an effective defect prediction model. This solution will be able to predict, on a file-level basis, the probability of a given file being faulty.
The advantage of such a model is that it can be used to predict a-priori the probability of the faults of each of the components being analyzed. This could be interesting if applied to initialize the Barinel algorithm. Being a Bayesian approach, Barinel requires some assumption about the prior fault probability of each component. This is an improvement from the original approach of initializing the priors by utilizing a uniform distribution, i.e. considering every file has the same probability of being faulty. This could tune the algorithm to identify buggy files with a higher probability of a correct diagnostic. We structured this solution following the logic depicted in Fig 6.

Figure 6: Solution Workflow



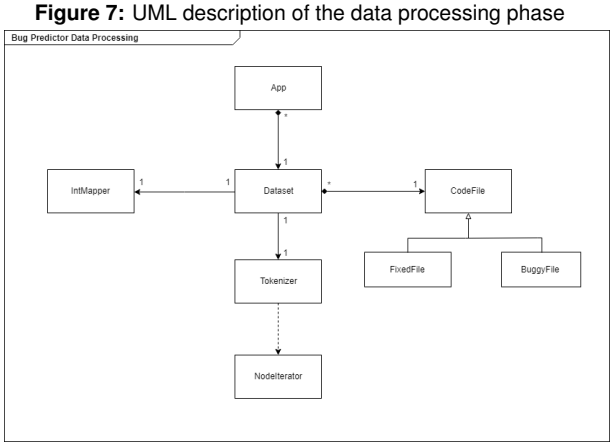
This workflow is separated into three phases: The Data Processing phase, the Data Mapping phase and Deep Learning model development phase.
In the first phase, processing the data, the goal is to extract meaningful data i.e. information from the source files present in the dataset. This means it will be necessary to parse the data. This is done by utilizing the Java Parser library. This phase is the basis for the whole program. If the information extracted is not relevant, meaning it only adds noise to the data, the predictions will have limited quality. It will not be possible to predict with high degree of confidence. In section 3.2.1, all details regarding our implementation are specified.
In the second phase, mapping the data, we map the output tokens parsed during the previous stage to vectors of integer values. This step is required

in order to translate the information to an encoding that can be fed to the machine learning model. In section 3.2.2 the implementation details are specified.
In the third and last phase, the development of the deep learning model, we will train a CNN deep learning architecture on the encoded information and examine its results. In section 3.3, the implementation details are specified whereas the results of the model are discussed in section 4.

3.2. Data Processing

The first step to data processing was choosing a reliable dataset. The decision was made to utilize the Bugs.jar dataset, as mentioned previously, due to a few reasons: It's a well-managed repository that is regularly updated; It has a variety of datasets, each one containing a project developed with a different objective in mind, thus the model is not trained for just one specific type of dataset; The organization of the data itself makes the analysis and mining of data easier. For example, all projects considered are written in Java which simplified the parsing phase.
Having obtained a relevant dataset, now comes the time to establish which procedures are needed in order to extract information. Firstly, we need to extract tokens by parsing the source code. We cannot consider the entire source code file as relevant because it has too much unwanted information. This can be regarded as a data cleaning technique. In actuality, our perspective was to reduce the raw code to simpler tokens that translate the semantic information of the code. After this, we map this information into another representation, one that can be processed by an artificial intelligence model. To achieve the desired outcome, we implemented a program in the Java language that would be able to read the data from the dataset and parse it to extract information. This application is modelled by the UML represented in Fig. 7.

Figure 7: UML description of the data processing phase



The challenge in developing this solution was encapsulating the concept of a dataset and parsing the correct information. The first one because we had to specify exactly which information a dataset should store and how to store its results in permanent storage. The second one came from reading the JavaParser library manual and becoming familiar with its methods in order to correctly utilize it as well as examining the results to see if the extracted data is correct.
The goal of the data processing phase is to translate the data into a meaningful representation that can then be processed by the machine learning model. This is done in 2 steps as seen in Fig. 6. The details to these steps are in the following sections.

3.2.1 Code Parsing

Defect prediction programs generally belong to two different categories, separated by granularity or specialization of the algorithm: Cross-project defect prediction and Within-project defect prediction. The first refers to an application which is regarded as used for general purpose defect prediction whereas the second is only applied with a high degree of confidence on the dataset it was trained. This difference in scope dictates which nodes are interesting, how they should be represented as tokens and consequently how they are mapped to integers. From here on, we will focus on cross-project defect prediction since this is the paradigm utilized by our program. The first obstacle was to define node relevance. Which nodes of the abstract syntax tree are interesting and which ones aren't. The choice to label a node interesting is based on their importance to the semantic meaning of the source file. The goal is to extract all such nodes. There is no benefit on extracting every node. Source code lines of basic type declarations or of arithmetical operations are not considered interesting since they are assumed to pertain to local class or method calculations. If the model considered every specific detail within a source code file, it could not identify generic structures within a file. To extract this overall meaning is our goal, so that the model based on this data predicts effectively. To accomplish this, we developed a list of interesting nodes. This list is based on ones used in other research in this field such as in [5] and [6]. In Fig. 8 is the complete list of AST nodes that were extracted.

Another issue to tackle is how to represent the extracted nodes. Different representations yield different mappings. The scope of the application dictates the choice of node representation. It must offer a degree of abstraction so that each token

Figure 8: List of AST nodes that were parsed

Relevant AST Nodes	
PackageDeclaration	ExpressionStmt
ImportDeclaration	MethodCallExpr
ClassOrInterfaceDeclaration	SuperExpr
ConstructorDeclaration	MethodReferenceExpr
MethodDeclaration	VariableDeclarationExpr
EnumDeclaration	AssertStmt
isFieldDeclaration	ThrowStmt
IfStmt	BreakStmt
WhileStmt	ContinueStmt
DoStmt	ReturnStmt
SwitchStmt	SynchronizedStmt
ForStmt	TryStmt
ForEachStmt	CatchClause

Figure 9: Sample source code of a method that serves as a parsing example

```
static public InetSocketAddress parseAddress(String address, int defaultPort) {
  String[] parts = address.split(":", 2);
  if (parts.length == 2) {
   if (parts[1].isEmpty())
    return new InetSocketAddress(parts[0], defaultPort);
  }
  return new InetSocketAddress(address, defaultPort);
 }
}
```

vector is mostly independent of a specific dataset syntax and types. Hence, there are a set of tokens that remain constant across all project.
The tokens extracted mostly translate Java keywords such as *if* or *while*, but also include several Java declarations. We decided to store these declarations not by name, but instead with a constant token depending on the declaration type because the actual name of the class or method is not important, but rather which statements they are composed by. This enables the extracted information to capture the general structure of the source file instead of overfitting on a particular dataset. Examples of these declarations include package, class or method declarations. This decision comes with a tradeoff nonetheless. This strategy allows for less specificity in detected changes but learns a more abstract semantic structure of the file which may leads to a model that predicts more efficiently over a set of datasets. Additionally, two types of two nodes extracted during parsing remain project specific. These are: Java method calls and variable declarations corresponding to *isMethodReferenceExpr* and *isVariableDeclarationExpr* respectively. Method Calls are stored according to the name of the method and user-defined type declarations are stored using, not the variable name, but the type of variable declared. In Fig. 9, we can see a sample source code that we will use to explain the derived set of tokens.

The tokens derived from this source code are, in order: *¡method-declaration¿, String[], split, ¡if¿, ¡if¿, ¡return¿, InetSocketAddress, ¡return¿, InetSocketAddress*.
By analyzing this result one can see the rules set for the parsing are being enforced. Firstly, each java keyword is recorded by its name, and secondly each variable declaration or method call is recorded by type or name respectively. When processing *IfStatement* nodes, their parameters are not considered since they mostly depend on local variables. Although bugs are bound to be overlooked by using this approach, we feel this improves the abstract nature of the parsing and does not diminish the semantic meaning of the source file.

3.2.2 Data Mapping

The data mapping is the responsibility of the *IntMapper* class. At this stage, it is done using what's commonly called one-hot encoding, meaning that each unique token gets assigned a new integer identifier. If the token being mapped is not unique, i.e. it had already been parsed in the same or another file, that token gets assigned the integer identifier corresponding to that type of token.
After mapping each source file, which was represented has a token vector, they are now translated into an integer vector. This integer vector is written to a text file in order to be permanently stored in the hard drive of the PC.
This structure enables the mapped data to be organized by dataset. All the mappings pertaining to a singular dataset, for example "accumulo", are under the accumulo folder under the root folder "output". We can then process all the issues identified in that dataset with each one having 1 or multiple pairs of fixed and buggy versions of a file, depending on the extent of the bug report.

3.3. Deep Learning Model

We use the convolutional neural network (CNN) to automatically learn features from token vectors extracted from source code, and then we utilize these features to build and train defect prediction models. In the following subsection we will detail how the model was built, i.e. decision on layers and parameters, and how what measures were chosen to evaluate it.

3.3.1 Deep learning evaluation

We examine our CNN approach to generating semantic features on le-level defect prediction tasks ,i.e. predict which les in a release are buggy. Focusing on this approach enables us to compare our proposed technique to prediction features and techniques. For le-level defect prediction, we generate semantic features by using the complete Abstract Syntax Trees of the source files with the help of the JavaParser library. In addition, most defect prediction studies have been conducted under two settings: Within-project defect prediction and Cross-project defect prediction.

We evaluate our approach only in a within-project defect prediction aspect. The metrics utilized to evaluate the models are the common standards used to evaluate machine learning models. Thus, we will consider: the Precision, the Recall and the F-Measure.

The precision measure is the ratio of faulty files among all files identified by the evaluated model as faulty. The Recall is the number of faulty files identified as such by the evaluated model divided by the total number of faulty files. A higher precision makes the manual inspection on a certain amount of predicted defective files to find more defects. An increase in recall reveals more defects in a given project. F-measure is a known combination of precision and recall. We will comment on the obtained results based on these metrics and consequently draw conclusions about the developed solution.

4. Experimental Results

4.1. Test Environment

The testing system is composed by an AMD Ryzen 7 1080x CPU with 8 cores with hyper-thread and a base clock of 4.0GHz, 16GB of RAM and a GTX 760Ti GPU with 4GB of VRAM.

The deep learning model implementation is done using the Keras Library using a TensorFlow backend. The datasets are obtained from the previously mentioned repository Bugs.jar. To guarantee the validity of the results each experiment was ran three times.

4.2. Parameter Tuning

In this section we present the choices made to obtain the optimal configuration of our machine learning model. In order to do this, we had to set the configuration of the mode. The parameters that are relevant include: The number of epochs, the size of the embedding, the number of filters and the filter length.

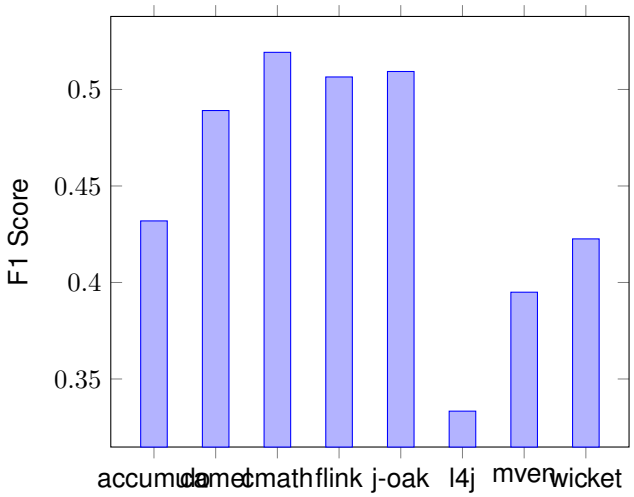
Due to brevity and conciseness, we decided not present this analysis as it has many figures. Please refer to the thesis for the full reasoning on these decisions. In summary, the optimal configuration used for testing corresponds to table 1.

Parameter	Optimal Value
Number of Epochs	10
Word Embedding size	32
Filter Length	3
Number of Filters	32

Table 1: Optimal Configuration of the CNN

4.3. CNN Model Results

The traditional techniques analyzed by Wang, S., et al in [5], on the PROMISE dataset had an average F1 score of 48%, with the maximum score being 65% for the synapse project and the lowest being 33% for jEdit. These results were obtained using a cross-project defect prediction approach and it is against this data we are going to evaluate our model.



The solution proposed by this thesis has an average F1 score of 45%, with the maximum score being 52% for the commons-math dataset and the lowest being 33% for the log4j. It can be generally concluded that the model produced did not perform up to expectations as both its average and maximum F1 scores are lower than those of the traditional methods applied to the PROMISE repository. By doing an analysis on the chart we can say that the model had an average performance on jackrabbit-oak, flink, commons-math and camel datasets since they line up on the average of the traditional techniques or slightly above it. On the contrary, maven, wicket, accumulo and logginlog4j2 datasets performed below the expected results. Thus, this solution does not improve on the statistical approach. This could be due to a few reasons: An overly abstract parsing process. This abstraction is seen on the decision to not tracking local variables as tokens or not distinguish between scopes of code. One such example is not distinguishing between cascading and sequen-

tial if statements. These changes would lead to more software components being parsed and better translation of code statements; A simplistic approach to the mapping of the parsed tokens. Using graph encoding for the tokens, in which its edges represent syntactic relationships between tokens, yields a richer input then a list of integers with no clear pattern.

5. Conclusions

This thesis defines the task at hand, the motivation behind studying this subject, it details the broader scope in which this subject belongs to, it references the related work that is relevant to the developed solution and finally, the solution and its results.

The main advantages of the strategy presented in this document are its ease of application to different datasets. In some cases, this approach improves upon traditional techniques. Another plus it that, development changes to the source application are easy since the code is not very complicated and is easily adaptable to other types of datasets. Finally, this approach generates a black-box model to the programmer. It can be used improve or extend existing diagnostic techniques.

The main drawbacks of this approach are: Firstly, by having a high degree of abstraction on its parsing phase it is, by definition, not specialized in one given dataset, this augments its application but prevents the achievement high accuracy on bug detection, as per our experiments. Secondly, it depends on tagged repositories. These datasets must have a specific structure and be specifically documented in order to be able to extract useful information. This model predicts more accurately if there is more data. This is a double edge sword if the data is hard to come by.

In summary, the development and study of this subject allowed me to research multiple areas in information technology that I find extremely interesting and in which I only had superficial knowledge. I believe this experience was enriching to my journey as person and a student of IT as I had to utilize many different tools and techniques.

5.1. Future Work

In my opinion, there are some different approaches that should also be explored and have the potential to improve the current state of affairs.

Firstly, the development of a specialized model using a different machine learning technique that is not as reliant on vast amounts of data and prioritizes the capture of the semantic structure of a text file. One method that is of interest is a Deep Belief Network. This model is capable of learning a representation from the training data that can reconstruct its semantic relationships and content with high probability [5]. Secondly, an integration of this data-driven approach into a software fault diagnosis algorithm, such as the Barinel algorithm. By integrating this approach into Barinel it may enhance its diagnosis likelihood estimates.

Lastly, having a different strategy towards the parsing phase, where more data is extracted for each source file.

References

[1] Peter Zoeteweij, Rui Abreu, Arjan J.C. van Gemund *Software Fault Diagnosis*. (English) Embedded Software Lab, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2600 GA Delft, The Netherlands.

[2] Amir Elmishali, Roni Stern, Meir Kalech, *Data-Augmented Software Diagnosis*. (English) in Proceedings of the Twenty-Eighth AAAI Conference on Innovative Applications, IAAI, Ben, Gurion University of the Negev, Be'er Sheva, Israel, 2016.

[3] Rui Abreu, Peter Zoeteweij, Arjan J.C. van Gemund *Spectrum-based multiple fault localization*. (English) ASE, vol. 24th IEEE/ACM International Conference, pp. 88-99, 2009.

[4] François Chollet *Deep learning with Python*. (English) [*On the electrodynamics of moving bodies*]. Shelter Island, NY: Manning Publications Co, 2018.

[5] Song Wang *Leveraging Machine Learning to Improve Software Reliability"*. (English) [*On the electrodynamics of moving bodies*]. University of Waterloo, Waterloo, Ontario, Canada, 2018.

[6] Jian Li, Pinjia He, Jieming Zhu, Michael R. Lyu, *Software Defect Prediction via Convolutional Neural Network*. (English) Shenzhen Research Institute, Hong Kong, China.

[7] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, Mukul R. Prasad *Bugs.jar: A Large-scale, Diverse Dataset of Real-world Java Bugs*. (English) ACM/IEEE 15th International Conference on Mining Software Repositories, 2018.

[8] Nicholas Smith, Danny van Bruggen, Federico Tomassetti *JavaParser: Visited. Analyse, transform and generate your java code base*. (English) Lean Publishing, 2018.

[9] François Chollet *Keras: The Python Deep Learning library*. [*Online*]. Available: https://keras.io/.

[10] P. S. Foundation *Python Language Reference, version 3.6*. (English) [*Online*]. http://www.python.org.