Code Smell Detection by Deep Direct-learning and Transfer-learning

Tushar Sharma[1], Vasiliki Efstathiou[2], Panos Louridas[2], and Diomidis Spinellis[2]

[1]Siemens Technology, Charlotte, USA
[2]Athens University of Economics and Business, Athens, Greece

***Abstract*—Context: An excessive number of code smells make a software system hard to evolve and maintain. Machine learning methods, in addition to metric-based and heuristic-based methods, have been recently applied to detect code smells; however, current methods are considered far from mature.**
Objective: First, explore the feasibility of applying deep learning models to detect smells without extensive feature engineering. Second, investigate the possibility of applying transfer-learning in the context of detecting code smells.
Method: We train smell detection models based on Convolution Neural Networks and Recurrent Neural Networks as their principal hidden layers along with autoencoder models. For the first objective, we perform training and evaluation on C# samples, whereas for the second objective, we train the models from C# code and evaluate the models over Java code samples and vice-versa.
Results: We find it feasible to detect smells using deep learning methods though the models' performance is smell-specific. Our experiments show that transfer-learning is definitely feasible for implementation smells with performance comparable to that of direct-learning. This work opens up a new paradigm to detect code smells by transfer-learning especially for the programming languages where the comprehensive code smell detection tools are not available.

Keywords: Code smells, Smell detection tools, Deep learning, transfer-learning.

I. Introduction

The metaphor of code smells is used to indicate the presence of quality issues in source code [1], [2]. A large number of smells in a software system is associated with a high level of technical debt [3] hampering the system's evolution. Given the practical significance of code smells, software engineering researchers have studied the concept in detail and explored various aspects associated with it including causes, impacts, and detection methods [2].

A large body of work has been carried out to detect smells in source code. Traditionally, metric-based [4], [5] and rule/heuristic-based [6], [7] smell detection techniques are commonly used [2], [8]. In recent years, smell detection techniques based on machine-learning [9], [10] have emerged as a potent alternative as they not only have the potential to bring human judgment in the smell detection but also provide the grounds for transferring results from one problem to another. Researchers have used Bayesian belief networks [11], [12], support vector machines [13], and binary logistic regression [14] to identify smells.

The resilience of machine learning models renders them appropriate for reuse beyond the bounds of tasks they may have been trained on. Transfer-learning refers to the technique where a learning algorithm exploits the commonalities between different learning tasks to enable knowledge transfer across tasks [15]. In this context, it would be plausible to explore the possibility of leveraging the availability of tools and data related to code smell detection in a programming language in order to train machine learning models that address the same problem on another language. The cross-application of a machine learning model could provide opportunities for detecting smells without actually developing a language-specific smell detection tool from scratch.

Despite the potential prospects, existing approaches for applying machine learning techniques for smell detection offer significant room for improvement. In a recent study, Di Nucci *et al.* [16] note, after observing practices such as improper data handling in existing software engineering literature, that the problem of detecting smells still requires extensive research to attain a maturity.

Furthermore, machine learning techniques (such as Bayesian networks, support vector machines, and logistic regression) that have been applied so far require considerable pre-processing to generate features for the source code, a substantial effort that hinders their adoption in practice. Traditionally, researchers use machine-learning methods that require extracting feature-sets from source code. Typically, code metrics are used as the feature set for smell detection purposes. We perceive two shortcomings in such usage of machine-learning methods for detecting smells. First, availability of an external tool to compute metrics for the target programming language on which we would like to apply the machine learning model becomes the prerequisite. Second and more importantly, we are limiting the machine learning algorithm to use only the metrics that we are computing and feeding as feature-set. Therefore, the machine learning algorithm cannot observe any pattern that is not captured by the provided set of metrics.

In this context, deep learning models, specifically neural

networks, offer a compelling alternative. The Convolution Neural Network (CNN) and the Recurrent Neural Network (RNN) are state-of-the-art supervised learning methods currently employed in many practical applications, including image recognition [17], [18], speech recognition [19], and natural language processing [20]. These advanced models are capable of inferring features during training and can learn to classify samples based on these inferred features.

In this paper, we present experiments with deep learning models with two specific goals:

- To investigate whether deep learning methods can effectively detect code smells. In particular, to employ architectures that include layers of CNNs, RNNs as well as autoencoders, inspect how different models perform on detecting diverse code smells and how model performance is affected by tweaking the learning hyper-parameters.
- To investigate whether results on smell detection through deep learning are transferable; specifically, to explore whether models trained for detecting smells on a programming language can be re-used to detect smells on another language.

Keeping these goals in mind, we define research questions and prepare an experimental setup to detect four smells *viz. complex method*, *complex conditional*, *feature envy*, and *multifaceted abstraction* using deep learning models in different configurations. We develop a set of tools and scripts to automate the experiment and collate the results. Based on the results, we derive conclusions to our addressed research questions.

The contributions of this paper are summarized below.

- An extensive study that applies deep learning models in detecting code smells without carrying out extensive feature engineering and compares the performance of different methods; to the best of our knowledge this is the first study of this kind and scale.
- An exploration that not only shows the feasibility of applying transfer-learning for identifying code smells but also compares the performance of deep learning models in the transfer-learning context. This exploration potentially will open a new paradigm to detect smells specifically for programming languages for which mature code smell detection tools are not available.
- Openly available tools, scripts, and data used in this experiment[1] to promote replication as well as incremental studies.
- The study identifies and documents challenges that we perceived and opportunities that the exploration offered.

The rest of the paper is organized as follows. Section II sets up the stage by presenting background and related work. We define our research objective in Section III and research method in Section IV. Section V presents our findings, discussion, and further research opportunities. We present threats to validity of this work in Section VI and conclude in Section VII.

[1]https://github.com/tushartushar/DeepLearningSmells

II. Background and Related Work

In this section, we present the background about the topic of code smells as well as machine learning and elaborate on the related literature.

A. Code Smells

Kent Beck coined the term "code smell" [1] and defined it as *"certain structures in the code that suggest (or sometimes scream) for refactoring."* Code smells indicate the presence of quality problems impacting many facets of quality [2] of a software system [1], [21]. The presence of an excessive number of smells in a software system makes it hard to maintain and evolve.

Smells are categorized as implementation [1], design [21], and architecture smells [22] based on their scope, granularity, and impact. Implementation smells are typically confined to a limited scope and impact (*e.g.,* a method). Examples of implementation smells are *long method*, *complex method*, *long parameter list*, and *complex conditional* [1]. Design smells occur at higher granularity, *i.e.,* abstractions, and hence are confined to a class or a set of classes. *God class*, *multifaceted abstraction*, *cyclic-dependency modularization*, and *rebellious hierarchy* are examples of design smells [21]. Along similar lines, architecture smells span across multiple components and have a system-wide impact. Some examples of architecture smells are *god component* [23], *feature concentration* [24], and *scattered functionality* [25].

A plethora of work related to code smell detection exists in the software engineering literature. Researchers have proposed methods for detecting smells that can be largely divided into five categories [2]. *Metric-based smell detection methods* [4], [5], [26] take source code as input, prepare a source code representation, such as an Abstract Syntax Tree (AST), compute a set of source code metrics, and detect smells by applying appropriate thresholds [4]. *Rule/Heuristic-based smell detection* methods [6], [7], [27], [28] typically take source code representations and sometimes additional software metrics as input. They detect a set of smells when the defined rules/heuristics get satisfied. *History-based smell detection* methods use source code evolution information [29], [30]. Such methods extract structural information of the code and how it has changed over a period of time. This information is used by a detection model to infer smells in the code. *Optimization-based smell detection* approaches [31]–[33] apply an optimization algorithm on computed software metrics and, in some cases, existing examples of smells to detect new smells in the source code. Further studies compare different detection methods [34], [35].

B. Deep Learning

Deep learning is a subfield of machine learning that allows computational models composed of multiple processing layers to learn representations of data with multiple levels of abstraction [36], [37]. Even though the idea of layered neural networks with internal "hidden" units was already introduced in the 80s [38], a breakthrough in the field came in 2006 by Hinton *et al.* [39] who introduced the idea of learning a hierarchy of features one level at a time. Ever since, and particularly during the course of the last decade, the field has taken off due to the advances in hardware, the release of benchmark datasets [40]–[42], and a growing research focus on optimization methods [43], [44]. Although deep learning architectures often consist of tens or hundreds of successive layers, much shallower architectures may also fall under the category of deep learning, as long as at least one hidden layer exists between the input and the output layer.

Deep learning architectures are being used extensively for addressing a multitude of detection, classification, and prediction problems. Architectures involving layers of CNNs are inspired by the hierarchical organization of the visual cortex in animals, which consists of alternating layers of simple and complex cells [45], [46]. CNNs have been proven particularly effective for problems of optical recognition and are widely used for image classification and detection [17], [18], [47], segmentation of regions of interest in biological images [48], and face recognition [49], [50]. Besides recognition of directly interpretable visual features of an image, CNNs have also been used for pattern recognition in signal spectograms, with applications in speech recognition [19]. In these applications the input data are given in the form of matrices (2D arrays) for representing the 2D grid layout of pixels in an image. 1D representations of data have been used for applying 1D convolutions in sequential data such as textual patterns [20] or temporal event patterns [51], [52]. However, when it comes to sequential data, RNNs [38] have been proven superior due to their capability to dynamically "memorize" information provided in previous states and incorporate it to a current state. Long Short Term Memory (LSTM) networks are a special kind of RNN that can connect information spanning long-term intervals, thus capturing long-term dependencies. LSTMs have been found to perform reasonably well within the context of representative applications that exhibit sequential patterns, such as speech recognition and music modeling [53], [54]. In addition, they have been established as state-of-the-art networks for a variety of natural language processing tasks; indicative applications include natural language generation [55], sentiment classification [56], [57] and neural machine translation [58], among others. Finally, approaches for addressing problems of both visual and sequential nature, rely on the use of autoencoders [59]. Autoencoders have been used in the past for performing dimensionality reduction and data compression [60], [61]. The basic idea of an autoencoder is that the input data is encoded into a compressed bottleneck-like representation which is in turn reconstructed back to an approximation of the input; the encoding-decoding process takes place in an unsupervised manner. Over the last decade, variants of autoencoders have been widely used as part of deep architectures for addressing problems of visual recognition [62], [63] and natural language processing [64], [65]. One of the advantages of autoencoders is that they have been proven robust to cross-domain generalisations [66], thus providing solutions for domains where training data is imbalanced or scarce. In a similar vein, autoencoders have been used for discovering patterns that do not conform to some—otherwise homogeneous—data sample. Following the same rationale as in using linear dimensionality reduction methods such as Principal Components Analysis (PCA) for outlier detection, autoencoders have been used as a non-linear alternative for discovering anomalies in extremely imbalanced data. Examples of problems where relevant methods have been used include image processing [67], and the identification of anomalies in spacecraft telemetry data [68].

C. Machine Learning Techniques on Source Code

The emergence of online open-source repository hosting platforms such as GitHub in recent years has led to an explosion on the volumes of openly available source code along with metadata related to software development activities; this bulk of data is often referred to as "Big Code" [74]. As an effect, software maintenance activities have started leveraging the wealth of openly available data, the availability of computational resources, and the recent advances in machine learning research. In this context, statistical regularities observed in source code have revealed the repetitive and predictable nature of programming languages, which has been compared to that of natural languages [75], [76]. To this end, problems of automation in natural language processing, such as identification of semantic similarity between texts, translation, text summarization, word prediction and language generation have been examined in the context of automating software development tasks. Relevant problems in software development include clone detection [77], [78], de-obfuscation [79], language migration [80], source code summarization [81], auto-correction [82], [83], auto-completion [84], generation [85]–[87], and comprehension [88].

On a par with equivalent problems in natural language processing, the methods employed for automating several software engineering tasks are switching from traditional rule-based and probabilistic n-gram models to deep learning methods. The majority of the proposed deep learning solutions rely on the use of RNNs which provide sophisticated mechanisms for capturing long term dependencies in sequential data, and specifically LSTMs [89] that have been proved particularly effective for modeling natural

TABLE I: Comparison of code smell detection techniques using machine learning

Study	Machine learning method	Detected smells	Feature-set
Foutse *et al.* [11]	Bayesian belief networks	Blob	Code metrics
Foutse *et al.* [12]	Bayesian belief networks	Blob, functional decomposition, spaghetti code	Code metrics
Abdou *et al.* [9], [13]	Support vector machine	Blog, functional decomposition, spaghetti code, swiss army knife, Code metrics	
Sérgio *et al.* [14]	Binary logistic regression	Long method	Code metrics
Bardez *et al.* [69]	CNN-based architecture	God class, feature envy	Code metrics
Fontana *et al.* [70]	16 machine learning algorithms	Data class, god class, feature envy, long method	Code metrics
Kim *et al.* [71]	Neural network	Large class, lazy class, data class, parallel inheritance hierarchy, god class, feature envy	Code metrics
Liu *et al.* [72]	CNN-based, neural, LSTM-based network	Feature envy, long method, large class, misplaced class	Code metrics and textual information
Hadj *et al.* [73]	Neural network and autoencoder	God class, data class, feature envy, long method	Code metrics
This study	CNN, RNN, and autoencoder-based network	Complex method, complex conditional, feature envy, multifaceted abstraction	Tokenized source code

language. Relevant methods have been applied on source code, aiming either to produce improved representations for encoding semantics of snippets [90], or as part of solutions to downstream tasks. Tasks that involve code edits have attracted particular interest, due to the practical implications induced by learning to automate pertinent maintenance activities. Employed towards this quest have been methods inspired from research on neural machine translation, such as the multi-layered LSTMs sequence-to-sequence model [91]. Results produced by leveraging equivalent architectures, with representations of code snippets before and after applying some change, show the potential that the seq-to-seq model has towards learning meaningful repairs and refactorings [92], [93]. Simpler types of networks that have produced promising results for semantic representations of code are based on the use of autoencoders [77], [94].

Alternative approaches to mining source code have employed CNNs in order to learn features from various representations of code. Li *et al.* [95] have used single-dimension CNNs to learn semantic and structural features of programs by working at the AST level of granularity and combining the learned features with traditional hand-crafted features to predict software defects. This method however incorporates hand-crafted features in the learning process and is not proven to yield transferable results. Similarly, a one-dimensional CNN-based architecture has been used by Allamanis *et al.* [96] in order to detect patterns in source code and identify "interesting" locations where attention should be focused. The objective of the study is to predict short and descriptive names of source code snippets (*e.g.,* a method body) given solely its tokens. CNNs have also been used by Huo *et al.* [97] in order to address the problem of bug localization. This approach leverages both the lexical information expressed in the natural language of a bug report and the structural information of source code in order to learn unified features. A more coarse-grain approach that also employs CNNs has been proposed in the context of program comprehension [98] where the authors use imagery rather than script in order to discriminate between scripts written in two programming languages, namely Java and Python. Similarly, Ren *et al.* [99] use a CNN-based neural network to identify self-admitted technical debt. Rantala *et al.* [100] use NLP techniques to identify technical debt from comments. CNN-based models along with NLP techniques are used by Zampetti *et al.* [101] to identify code patterns to help pay-back technical debt.

D. Machine Learning on Smell Detection

In recent times, *machine learning-based smell detection* methods have attracted software engineering researchers. Machine learning is a subfield of artificial intelligence that *trains* solutions to problems rather than modeling them through hard-coded rules. In this approach, the rules that solve a problem are not set a-priori; rather, they are inferred in a data-driven manner. In supervised learning, a model is trained by being exposed to examples of instances of the problem along with their expected answers and statistical regularities are drawn. The representations that are learned from the data can in turn be applied and generalized to new, unseen data in a similar context.

Table I presents a comparison of existing attempts to detect smells using machine learning techniques. A typical machine learning smell detection method starts with a mathematical model representing the smell detection problem. Existing examples and source code models are used to train the model. The trained model is used to classify or predict the code fragments into smelly or non-smelly instances. Foutse *et al.* [11], [12] use a Bayesian approach for the detection of three design smells. Their study forms a Bayesian graph using a set of metrics and determines the probability of a class being positive to a smell. Similarly, Abdou *et al.* [9], [13] employ support vector machine-based classifiers, trained using a set of 60 object-oriented metrics for each class to detect design smells (*blob*, *feature concentration*, *spaghetti code*, and *swiss army knife*). Furthermore, Sérgio *et al.* [14] detect

long method smell instances by employing binary logistic regression. They use commonly used method metrics, such as Method Lines of Code (MLOC) and cyclomatic complexity as regressors. Bardez *et al.* [69] present an ensemble method that combine outcomes of multiple tools to detect *god class* and *feature envy* smells. They identify a set of key metrics for each smell and feed them to a CNN-based architecture. Fontana *et al.* [70] compare performance of various machine learning algorithms in detecting *data class, god class, feature envy*, and *long method*. Azadi *et al.* [102] introduce WekaNose, a semi-automated tool that learns rules by identifying correlations between code smell instances and relevant metrics. Fontana and Zanoni [103] use regression-based methods with extensively engineered features in order to classify code smell instances according to their severity. Overall, research on smell detection with machine learning techniques relies mostly on traditional methods with decision trees and support vector machines being the most commonly used algorithms [104]. Recent approaches that adapt deep learning architectures in the context of smell detection are limited. These presume substantial data engineering [71], to the extent of combining metrics relevant to different features exhibited in code (e.g., textual and structural features) [72], or hybrid methods that include a feature learning stage before feeding the data into the neural network [73].

The performance of a machine learning model primarily depends on the choice of suitable data representations that will adequately capture informative features for the task in hand. Another crucial factor for the performance of a model is the amount of available training data and the formation of the evaluation samples. As the proportion of positive and negative samples becomes more imbalanced, the classification task of the models becomes harder. Hence, a model would perform significantly better when classifying data from balanced datasets. Most of the above-mentioned approaches do not explicitly mention the ratio of positive and negative samples used for the evaluation. Fontana *et al.* [105] carry out the evaluation using 140 positive and 280 negative samples for each smell which is considerably balanced compared to a realistic case. We further discuss this issue and demonstrate the effect of class imbalance in Section V.

E. The Need of Applying Deep Learning for Smell Detection

Section II-A describes existing techniques for detecting code smells. Mainstream tools use rule-based and metric-based approaches to detect smells [2]. However, *context* plays an important role in deciding whether a reported smell is actually a quality issue for the development team, and existing tools do not consider the context while detecting the smells. For example, a method with a switch-case statement with ten cases, where each case instance has only a couple of simple statements will be detected as a *complex conditional* smell by the mainstream tools, because the associated rule (cyclomatic complexity greater than a threshold) will be triggered. However, the method's developer might not consider it complex, because all case phrases share the same structure. When it comes to the tools' validation, even manual annotation is often inadequate for ensuring the validity of the source code element rules. The main reason is that validation is typically carried out on open-source projects where the human validators are viewing the code snippets for the first time. Therefore, even though the tools are correct by the defined rules they still lack context-sensitivity. Deep learning, without specific feature set specification (such as metrics), could bring the code's context under consideration when detecting smells.

The above discussion takes us to the next question: how to obtain or generate much needed training data for deep learning? The training data clearly need to be prepared considering the project's context, because the aim is to make smell detection more context-sensitive. However, to the best of our knowledge, a large training dataset for training models of this scale is not available. In this study, we are using training data generated from existing tools as a first step towards assessing the extent to which smell detection is feasible via deep learning techniques. The study is a preliminary evaluation to verify the extent to which deep learning is suitable to detect smells that may involve context-sensitivity. This provides a stepping stone for future studies that will address more sophisticated problems such as custom context-sensitive smell detection. Such studies could replace our training data with manually annotated context-sensitive training data to achieve context-sensitive smell detection, thus overcoming the burden of hard coding custom rules into existing tools.

F. Challenges in Applying Deep Learning on Source Code

Applying deep learning techniques on source code is non-trivial. In this section, we present challenges that we face in the process of applying deep learning techniques on source code.

1) Limits in analogies with other domains: Deep learning is advancing rapidly in domains that address problems of image, video, audio, text, and speech processing [36]. Consequently, these advances drive current trends in deep learning and inspire applications across disciplines. As such, studies that apply deep learning on source code rely heavily on results from these domains, and particularly that of text mining.

Based on prior observations that demonstrate similarity between source code and natural language [75], the research community has largely addressed relevant problems on mining source code by adopting latest state-of-the-art natural language processing methods [79], [81], [87], [96], [106]. However, besides similarities, there also exist major differences that need to be taken into consideration when designing such studies. First of all, source code,

unlike natural language, is semantically fragile; minor syntactic changes can drastically change the meaning of code [74]. As an effect, treating code as text by ignoring the underlying formal semantics carries the risk of not preserving the appropriate meaning. Besides formal semantics, the syntax of source code obviously presents substantial differences compared to the syntax found in text. As a result, methods that perform well on text are likely to under-perform on source code. Architectures involving CNN-1D layers, for instance, have been proven effective for matching subsequences of short lengths [107], which are often found in natural language where the length of sentences is limited. This however does not necessarily apply on self-contained fragments of source code, such as method definitions, which tend to be longer. In order to address these shortcomings, currents research invests on developing appropriate representations for code [90], [108], [109].

Finally, even though good practices dictate naming conventions in coding, unlike natural language, there is no universal vocabulary of source code. This results to a diversity in the artificial vocabulary found in source code that may affect the quality of the models learned. Rare and complex identifiers constantly devised by developers result to limited repetition of terms, as well as patterns of locality, that are not common in natural language [110]. The implications of these peculiarities in the quality of the resulting machine-learned models are acknowledged by the community, whilst latest research advances aim towards addressing these shortcomings [111]–[113] and painstakingly re-examining past results [114].

Approaches that treat code as text mainly focus on the mining of sequential patterns of source code tokens. Other emerging approaches look into structural characteristics of the code with the objective of extracting visual patterns delineated on code [98]. Even though there are features in source code, such as nesting, which demonstrate distinctive visual patterns, treating source code in terms of such patterns and ignoring the rich intertwined semantics carries the risk of oversimplifying the problem.

2) Lack of resources: Research employing deep learning techniques on software engineering data, including source code as well as other relevant artifacts, is still young. Consequently, results against traditional baseline techniques are very limited [110], [115] while the debate on whether deep neural networks are suitable for modeling source code is still open [110], [116]. Especially when it comes to processing solely source code artifacts, relevant studies are scarce and mostly address the problem of drawing out semantics related to the functionality of a piece of code [77], [96], [117]–[119]. To the best of our knowledge, our study is the first to thoroughly investigate the application of deep learning techniques with the objective of examining characteristics of source code quality without making use of derived features. Therefore, a major challenge in studies of this kind is that there is no prior knowledge that would guide this investigation, a challenge reflected on all stages of the inquiry. At the level of designing an experiment, there exist no rules of thumb indicating a set up for a deep learning architecture that adequately models the fine-grained features required for the problem in hand. Furthermore, at the level of training a model, there is no prior baseline for hyper-parameters that would lead to an optimal solution. Finally, at the level of evaluating a trained model, there exist no benchmarks to compare against; there is no prior concrete indication on the expected outcomes in terms of reported metrics. Hence, a result that would appear sub-optimal in another domain such as natural language processing, may actually account for a significant advance in software quality assessment.

Besides challenges that relate to the know-how of applying deep learning techniques on source code, there are technical difficulties that arise due to the paucity of curated data in the field. The need for openly available data that can serve for replicating data-driven studies in software engineering has been long stressed [120]. The release of curated data in the field is encouraged through badging artifact-evaluated papers as well as dedicated data showcase venues for publication. However, the software engineering domain is still far from providing benchmark datasets, whereas the available datasets are limited to curated collections of repositories with associated metadata that lack ground truth annotation that is essential for a multitude of supervised machine learning tasks. Therefore, unlike domains such as image processing and natural language processing where an abundance of annotated data exist [40]–[42], [121], in the field of software engineering the lack of gold standards induces the inherent difficulty of collecting and curating data from scratch. The need for curating datasets of reference in software engineering studies has been recognized in the past [122], however, the progress in this front has not kept pace with the increasing volumes of data in the wild. Current efforts for overcoming this limitation include establishing benchmarks for evaluating results on semantic code search [123] and testing [124], and the release of large-scale pretrained models for programming language vocabularies [125], in an analogy to the natural language paradigm [126].

III. Research Objectives

The goal of this research is to explore the plausibility of applying state-of-the-art deep learning methods to detect smells. Furthermore, within the same context, this work examines the feasibility of applying transfer-learning. Based on the stated goals, we define the following research questions that this work aims to explore.

RQ1 **Is it possible to detect code smells using deep learning methods? If yes, which deep learning method performs superior?**

We use CNN, RNN and autoencoder models in this exploration. For the CNN-based architecture, we provide input samples in 1D and 2D format to observe the difference in

their capabilities due to the added dimension; we refer to them as CNN-1D and CNN-2D respectively. In the context of this research question, we define the following hypotheses.

RQ1.H1: *It is feasible to detect code smells using deep learning methods.*

The considered deep learning models have demonstrated high performance in the domain of image processing and natural language processing [127]. We believe we can leverage these models in the presented context.

RQ1.H2: CNN-2D *performs better than* CNN-1D *in the context of detecting smells.*

The rationale behind this hypothesis is the added dimensionality in CNN-2D. The 2D model might observe inherent patterns when input data is presented in two dimensions that may possibly be hidden in one dimensional format. For instance, a 2-D variant could possibly identify the nesting depth of a method easier than its 1-D counterpart when detecting *complex method* smell.

RQ1.H3: RNN *models perform better than* CNN *models in the context of detecting smells.*

RNNs are considered better for capturing sequential patterns and have the reputation to work well with text. Thus, taking into account the similarities that source code and natural language share, RNN models could prove superior to CNN models.

RQ1.H4: RNN *and* CNN *variants of autoencoder model exhibit comparable performance to those of* RNN *and* CNN-1D *models.*

An autoencoder model could be realized in various ways. In this work, we experiment with three variants of autoencoder models in which the models use fully connected neural network layers, LSTM layers, and convolution layers respectively. We hypothesize that the performance of the autoencoder variants follow a pattern similar to that observed with RNN and CNN-1D models.

RQ2 **Is it possible to detect code smells by applying transfer-learning techniques on similar languages? If yes, which deep learning model exhibits superior performance in detecting smells when applied in transfer-learning setting?**

Transfer-learning is the capability of an algorithm to exploit the similarities between different learning tasks and provide a solution for a task by transferring knowledge acquired while solving another task. We would like to explore whether it is feasible to train a deep learning model from samples of C# and predict the smells using this trained model in samples of Java programming language and vice-versa. The feasibility exploration can be termed positive if the produced results are comparable with the results obtained from direct learning (RQ1). We derive the following hypotheses.

RQ2.H1: *It is feasible to detect code smells by applying transfer-learning techniques on similar languages.*

Given the high similarity in the syntax between the two programming languages considered in this study, we believe that we may train a model on samples of the one language and use it to classify smelly and non-smelly fragments on evaluation samples from the other.

RQ2.H2: *The performance of transfer-learning is inferior compared to that of direct-learning.*

Direct-learning in the context of our study refers to the case where training and evaluation samples belong to the same programming language. We expect that the performance of the models in transfer-learning could be inferior to that of direct-learning, given that in direct-learning both training and evaluation samples come from the same programming language, and hence are expected to exhibit homogeneous features.

IV. Research Method

This section describes the employed research method by first providing an overview and then elaborating on the data curation process. We also discuss the selection protocol of smells and architecture of the deep learning models.

A. Overview of the Method

Figure 1 provides an overview of the experiment. We download 922 C# and 1,721 Java repositories from GitHub. We use the Designite and DesigniteJava smell detection tools to analyze C# and Java code respectively. We use CodeSplit, a set of utilities, to extract each method and class definition into separate files from C# and Java programs. Then the learning data generator uses the detected smells to bifurcate code fragments into positive or negative samples for a specified smell—positive samples contain the smell while the negative samples are free from that smell. Tokenizer takes a method or class definition and generates integer tokens for each token in the source code. As a preprocessing step we remove identical samples on the Tokenizer's output, thus ensuring that the effects of code duplication on the evaluation of the resulting models are mitigated [128]. The processed output of the Tokenizer is ready to feed to the neural networks.

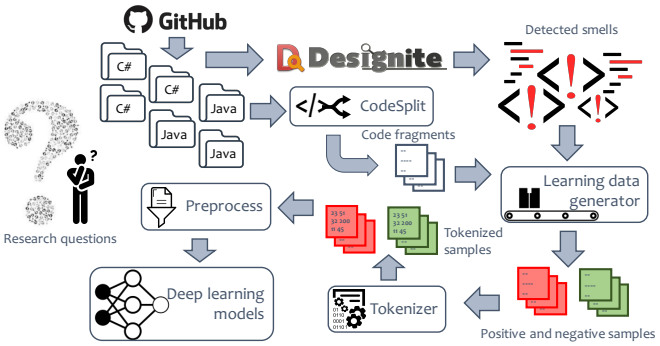


Fig. 1: Overview of the proposed method

B. Data Curation

In this section, we elaborate on the process of generating training and evaluation samples along with the tools used in the process.

1) Downloading repositories: We use the following protocol to identify and download our subject systems.

- We download repositories containing C# and Java code from GitHub. We use RepoReapers [129] to filter out low-quality repositories. RepoReapers analyzes GitHub repositories and provides scores for eight dimensions of their quality. These dimensions are architecture, community, continuous integration, documentation, history, license, issues, and unit tests.
- We select all the repositories where at least seven out of eight RepoReapers' dimensions have suitable scores for both C# and Java repositories. We consider a score suitable if it has a value greater than zero.
- We ensure that RepoReapers results do not include forked repositories [130].
- We discard repositories with fewer than ten stars and less than $1,000$ LOC.
- Following these criteria, we get a filtered list of 922 C# and $1,721$ Java repositories. We select a random subset of 922 Java repositories (by choosing a seed from the system clock for the random number generator) from the filtered Java repository list in order to mitigate the discrepancy between the volume of C# and Java code to be analyzed.
- Finally, we download and analyze the selected 922 C# and 922 Java repositories.

2) Smell detection: We use Designite to detect smells in C# code. Designite [7], [131] is a software design quality assessment tool for code written in C#. It supports detection of 11 implementation, 20 design, and seven architecture smells by analyzing source code properties at different granularities (method, class, and component). It also provides commonly used code metrics and other features such as trend analysis, code clone detection, and dependency structure matrix to help developers assess the software quality.[2]

Similar to the C# version, we have developed DesigniteJava [132], which is an open-source tool for analyzing and detecting smells in a Java codebase. The tool supports detection of 18 design and ten implementation smells. Both the C# and Java versions implement the same rules to detect smells.

We use the console version of Designite (version 3.4.0) and DesigniteJava (version 1.5.0) to analyze C# and Java code respectively and detect the specified design and implementation smells in each of the downloaded repositories.

Manual validation: We conducted a manual validation to establish the accuracy of the used tools. We chose the well-known open-source repository `DotNetOpenAuth`[3] for this purpose. The repository implements OpenAuth and OpenID protocol in C# and has a long development history ($3,500$ commits at the time of writing this paper). It has been used by more than 19.6 thousand repositories and has attracted 742 stars. From this repository, we selected three projects `DotNetOpenAuth.Core`, `DotNetOpenAuth.OpenId.RelyingParty.UI` and `OAuthClient`. The selected projects contain $22,027$ LOC and 166 classes. We sought help from two volunteers to carry out manual validation—one volunteer works in a software development company (three years of industrial experience) and another volunteer is a computer science Ph.D. student with one year of industrial experience. None of the volunteers has worked on the analyzed repository before; however, they both have hands-on experience on complex industrial solutions and have a fair idea of software architecture and code smells.

We enforced the following protocol for the validation.

- Each volunteer carried out the initial manual analysis individually without discussing it with the other volunteer.
- Given their industry experience, they were familiar with the concept of code smell and were aware of commonly known smells. Each volunteer was presented the definition of each of the four considered design and implementation smells and interviewed to verify its correct understanding. We also provided additional material to accelerate their learning.
- Both the individuals went through all source code files one by one and checked the existence of each smell following the corresponding definition.
- While identifying smells, they were allowed to use IDE features such as *go to definition* and *list all references* as well as metrics generated from other tools they wish to use.
- While identifying smells, they were presented with method/class metrics.
- Once both the volunteers completed the analysis, we computed Cohen's Kappa [133] to measure the mutual agreement between the volunteers' findings. We obtained $\kappa = 0.65$ as the value of Cohen's Kappa.
- Once both completed their individual analysis, they discussed their results, sorted out differences, and prepared a consolidated mutually agreed report of results.
- Next, they used Designite and analyzed the considered project and obtained a list of design and implementation smells.
- They compared the results obtained from the tool with their set of smells and tagged them as true-positive, false-positive, and false-negative.

Both volunteers manually scanned 166 classes for the two design smells and 280 methods for the two implemen-

[2]A free academic license of Designite can be requested—http://www.designite-tools.com/acad-lic-request/

[3]https://github.com/DotNetOpenAuth/DotNetOpenAuth

tation smells. They found that the tool's result matched their manual result except two instances of *feature envy* design smell. In both the cases, the tool was not counting the class members (*i.e.,* methods and fields) belonging to another class when more than one member is referenced from the class under analysis. We fixed the problem in the tool before using it in our experiments; hence the tool shows perfect precision and recall for the smells considered in the experiment.

3) Splitting code fragments: CodeSplit is a set of two utility programs, one for each programming language, that split methods or classes written in C# and Java source code into individual files. Hence, given a C# or Java project, the utilities can parse the code correctly (using Roslyn for C# and Eclipse JDT for Java), and emit the individual method or class fragments into separate files following hierarchical structure (*i.e.,* namespaces/packages become folders). CodeSplit for Java is an open-source project that can be found on GitHub [134]. CodeSplit for C# can be downloaded freely online [135].

4) Generating training and evaluation data: The learning data generator requires information from two sources; a list of detected smells for each analyzed repository and a path to the folder where the code fragments corresponding to the repository are stored. The program takes a method (or class in case of design smells) at a time and checks whether the given smell has been detected in the method (or class) by Designite. If the method (or class) suffers from the smell, the program puts the code fragment into a "positive" folder corresponding to the smell otherwise into a "negative" folder.

5) Tokenizing learning data: Machine learning algorithms require the inputs to be given in a representation appropriate for extracting the features of interest, given the problem in hand. For a multitude of machine learning tasks it is a common practice to convert data into numerical representations before feeding them to a machine learning algorithm. In the context of this study, we need to convert source code into vectors of numbers honoring the language keywords and other semantics. Tokenizer [136] is an open-source tool that provides, among others, functionality for tokenizing source code elements into integers where different ranges of integers map to different types of elements in source code. Figure 2 shows a small C# method and corresponding tokens generated by Tokenizer. Currently, it supports six programming languages, including C# and Java.

6) Data preparation: The stored samples are read into *numpy* arrays, preprocessed, and filtered. We first perform bare minimum preprocessing to clean the data—for both 1D and 2D samples—we scan all the samples for each smell and remove duplicates if any exist.

The data preparation steps are explained below.

- We split the samples in the ratio of 70-30 for training; *i.e.,* 70% of the samples are used for training a model while 30% samples are used for evaluation.

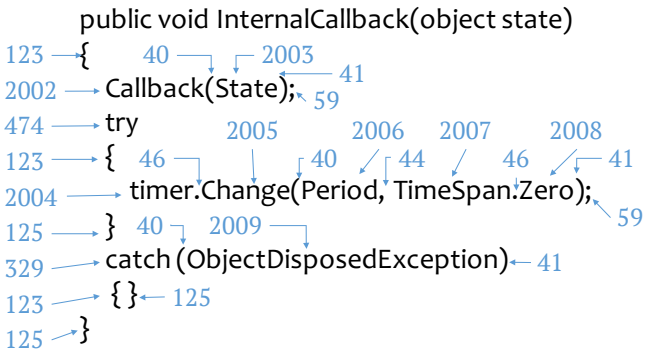


Fig. 2: Tokens generated by Tokenizer for an example

- For the training samples, we perform the following steps.
  - We limit the maximum number of positive/negative samples to 5,000. Therefore, for instance, if the number of negative samples is greater than 5,000, we keep for the experiment exactly 5,000 samples and drop the rest *i.e.,* adopting an under-sampling technique [137].
  - We perform model training using balanced samples, *i.e.,* we balance the number of samples for training by choosing the smaller number between the positive and negative sample count; we discard the remaining training samples from the surplus.
- For the evaluation samples, we perform the following steps.
  - The training and evaluation time depend on the number of samples. We limit the maximum number of positive/negative evaluation samples to 150,000 and 50,000 for implementation and design smells respectively to reduce the processing load. Even with these limits, all the experiments take 298 hours to complete with the best hardware available to us. The upper limit of the samples is set way higher than the typical sample size in studies from the similar domain.
  - In the process of removing excess evaluation samples, we maintain the ratio between positive and negative samples for evaluation.

Table II presents data preparation process by providing number of samples in each step for all smells.

Each individual input instance, either a method in the case of implementation smells, or a class in the case of design smells, is stored in the appropriate data structure depending upon the model that will use it. In 1D representation, each individual input instance is represented by a flat 1D array of sequences of tokens, compatible for use with the RNN, CNN-1D and the autoencoder models. In the 2D representation, each input instance is represented by a 2D array of tokens, preserving the original statement-by-

TABLE II: Number of samples in each step of preparing input data

			Initial samples	70-30 split	Applying max limit	Balancing
CM	Positive	Training	24,963	17,474	5,000	5,000
		Evaluation		7,489	7,489	7,489
	Negative	Training	464,866	325,406	5,000	5,000
		Evaluation		139,460	139,460	139,460
CC	Positive	Training	6,186	4,330	4,330	4,330
		Evaluation		1,856	1,856	1,856
	Negative	Training	484,790	339,353	5,000	4,330
		Evaluation		145,437	145,437	145,437
FE	Positive	Training	1,800	1,260	1,260	1,260
		Evaluation		540	540	528
	Negative	Training	170,439	119,307	5,000	1,260
		Evaluation		51,132	50,000	50,000
MA	Positive	Training	293	205	205	205
		Evaluation		88	88	85
	Negative	Training	172,412	120,688	5,000	205
		Evaluation		51,724	50,000	50,000

statement delineation of source code thus providing the grid-like input format that is required by CNN-2D models. All the individual samples are stored in a few files (where each file size is approximately 50 MB) to optimize the I/O operations due to a large number of files. We read all the samples into a *numpy* array and we filter out the outliers. In particular, we compute the mean input size and discard all the samples with length over one standard deviation away from the mean. This filtering helps us keep the training set in reasonable bounds and avoids waste of memory and processing resources. We pad the input array with zeros to the extent of the longest remaining input in order to create vectors of uniform length and bring the data in the appropriate format for using with the deep learning models. Finally, we shuffle the array of input samples along with its corresponding labels array.

C. Selection of Smells

Over the last two decades, the software engineering community has documented many smells associated with different granularities, scope, and domains [2]. A comprehensive taxonomy of well-established software smells can be found online.[4] For this study, selection of smells is a crucial decision that needs to balance ambition with practicality. On the practicality front, the scope of the higher granularity smells, such as design and architecture smells, is wide, often spanning to multiple classes and components. It is essential to provide all the intertwined source code fragments to the deep learning model to make sure that the model captures the key deciding elements from the provided input source code. Hence, it is naturally difficult to detect them using deep learning approaches, unless extensive feature engineering is performed beforehand in order to attain an appropriate representation of the data. We started with implementation smells because they can be detected typically just by looking at a method. To address ambition, we would like to avoid very simple smells (such as *long method*) which can be easily detected by less sophisticated techniques.

We chose *complex method* (CM—*i.e.,* the method has high cyclomatic complexity) and *complex conditional* (CC—*i.e.,* a condition expression in a conditional statement such as `if statement` is complex). These two smells represent two dissimilar cases where neural networks have to spot specific features. To detect *complex conditional*, the neural networks must spot a specific range of tokens within only conditional blocks. On the other hand, detection of *complex method* requires looking at the entire method and the structural property within it (*i.e.,* nesting depth of the method).

To expand the experiment's ambition, we also select two design smells *feature envy* (FE—*i.e.,* a method seems more interested in an abstraction other than the one it actually is in) and *multifaceted abstraction* (MA—*i.e.,* a class has more than one responsibility assigned to it). The scope of these smells is larger (*i.e.,* the whole class) and detection is not trivial since the neural network has to capture cohesion and coupling aspects. These smells not only allow us to compare the capabilities of neural networks in detecting implementation smells and design smells but also sets the stage for the future work to build on.

D. Architecture of Deep Learning Models

In this section, we present the architecture of the neural network models that we use in this study. The Python implementation of the experiments using the Keras library can be found online [138].

1) CNN *model:* Figure 3 presents the architecture of the CNN model used to detect smells. This architecture is inspired by typical CNN architectures used in image classification [17] and consists of a feature extraction part followed by a classification part. The feature extraction part is composed of an ensemble of layers, specifically, convolution, batch normalization, and max pooling layers. This set of layers forms the architecture's hidden layers. The convolution layer performs convolution operations based on the specified filter and kernel parameters and

[4]http://www.tusharma.in/smells

computes accordingly the network weights to the next layer, whereas the max pooling layer effectuates a reduction on the dimensionality of the feature space. Batch normalization [139] mitigates the effects of varied input distributions for each training mini-batch, thus optimizing training. In order to experiment with different configurations, we use one, two, and three hidden layers.

The output of the last max pooling layer is connected to a dropout layer. Dropout performs another type of regularization by ignoring some randomly selected nodes during training in order to prevent over-fitting [140]. In our experiments we set the dropout rate for the layer to be equal to 0.1 which means that the nodes to be ignored are randomly selected with probability 0.1.

The output of the last dropout layer is fed into a densely connected classifier network that consists of a stack of two dense layers. These classifiers process 1D vectors, whereas the incoming output from the last hidden layer is a 3D tensor. The tensor corresponds to the height and width of an input sample and channel; in this case, the number of channels is one. For this reason, a flatten layer is used first, to transform the data in the appropriate format before feeding them to the first dense layer with 32 units and *relu* activation. This is followed by the second dense layer with one unit and *sigmoid* activation. This last dense layer comprises the output layer and contains a single neuron in order to make predictions on whether a given instance belongs to the positive or negative class in terms of the smell under investigation. The layer uses the sigmoid activation function in order to produce a probability within the range of 0 to 1.

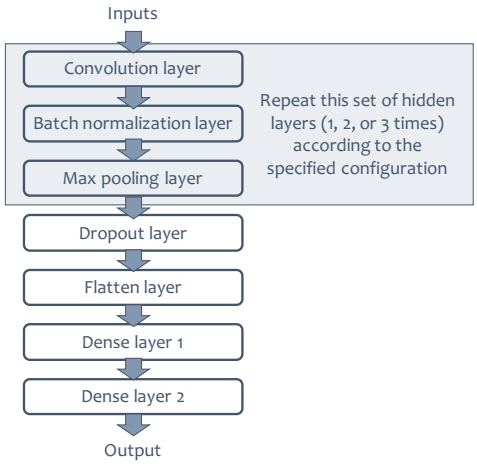


Fig. 3: Architecture of the employed CNN models

We use dynamic batch size depending upon the size of samples to train. We divide the training sample size by 512 and use the result as the index to choose one of the items in the possible batch size array (32, 64, 128, 256). For instance, we use 32 as batch size when the training sample size is 500 and 256 when the training sample size is 2000.

The hyper-parameters are set to different values in order to experiment with different configurations of the model. Table III lists all the different values chosen for the hyper-parameters. *Filters* is the number of convolutional filters employed, *kernel size* controls the size of the convolution window, and *pooling window size* governs the size of the down-sampling window during the pooling operation. We execute CNN models for 144 configurations that result from generating combinations of different values of hyper-parameters and number of repetitions of the set of hidden layers ($4 \times 3 \times 4 \times 3 = 144$). We label each configuration between 1 and 144 where configuration 1 refers to *number of repetitions of the set of hidden layers* = 1, number of *filters* = 8, *kernel size* = 5, and *pooling window size* = 2. Similarly, configuration 144 refers to *number of repetitions of the set of hidden layers* = 3, number of *filters* = 64, *kernel size* = 11, and *pooling window size* = 5. Both the 1D and 2D variants use the same architecture replacing the 2D version of Keras layers for their 1D counterparts.

TABLE III: Chosen values of hyper-parameters for the CNN model

Hyper-parameter	**Values**
Filters in convolution layer	{8, 16, 32, 64}
Kernel size in convolution layer	{5, 7, 11}
Pooling window size in max pooling layer	{2, 3, 4, 5}
Maximum epochs	{50}

We ensure the best attainable performance and avoid over-fitting by using *early stopping*[5] as a regularization method. This implies that even though the model is allowed to reach a predetermined maximum of 50 epochs during training, it may be forced to stop earlier. If there is no improvement in the validation loss of the trained model for five consecutive epochs (since patience, a parameter to early stopping mechanism, is set to five), the training is interrupted. Along with this, we also use *model check point* to restore the best weights of the trained model. We chose a maximum of 50 after carrying out a preliminary experiment which indicated that the majority of models would converge within this threshold. Among 386 total individual experiments for all four smells in RQ1 for CNN-1D, the models reached the maximum epoch only four times. In those cases, we stop the training and evaluate the model based on the weights at the last epoch.

For each experiment, we compute the following performance metrics: precision, recall, F1 score, and average precision score. We also record the actual epoch count where the models stopped training (due to early stopping). After we complete all the experiments with all the chosen hyper-parameters, we choose the best performing configuration and the corresponding number of epochs used by the experiment and retrain the model and record the final and best performance of the model.

[5]https://keras.io/callbacks/

2) RNN model: Figure 4 presents the architecture of the employed RNN model which is inspired by state-of-the-art models in natural language modeling that employ an LSTM network as a recurrent layer [141]. The model consists of an embedding layer followed by the feature learning part — a hidden LSTM layer. It is succeeded by the regularization (realized by a dropout layer) and classification (consisting of a dense layer) part.

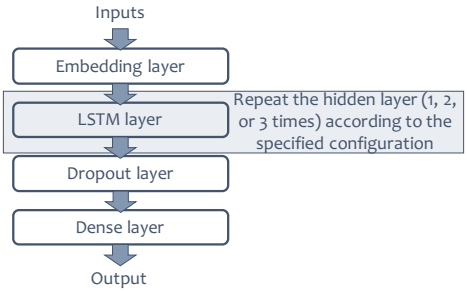


Fig. 4: Architecture of the employed RNN models

The embedding layer maps discrete tokens into compact dense vector representations. One of the advantages of the LSTM networks is that they can effectively handle sequences of varying lengths. To this end, in order to avoid the noise produced by the padded zeros in the input array, we set the *mask_zero* parameter to *True* in the Keras embedding layer. Thus the padding is ignored and only the meaningful part of the input data is taken into account. We set *dropout* and *recurrent_dropout* parameters of LSTM layer to 0.1. The regular dropouts mask (or drop) network units at inputs and/or outputs whereas recurrent dropouts drop the connections between the recurrent units along with dropping units at inputs and/or outputs [142]. The output from the embedding layer is fed into the LSTM layer, which in turn outputs to the dropout layer. As in the case of the CNN model, we experiment for different depths of the RNN model by repeating multiple instances of the hidden layer.

The dropout layer uses a dropout rate equal to 0.2, which we empirically found effective for preventing over-training, yet conservative enough for avoiding under-training. The dense layer, which comprises the classification output layer, is configured with one unit and *sigmoid* activation as in the case of the CNN model. Similarly to the CNN model, we use *early stopping* (with maximum epochs = 50 and patience = 2) and *model check point* callbacks. Also, we use the dynamic batch size selection as explained in the previous section.

We try different values for the model hyper-parameters; Table IV lists the values selected for experimentation. The *dimensionality* of the embedding layer represents the size of each embedding vector; LSTM *units* is the number of units in each LSTM layer. We measure the performance of the RNN model in 18 configurations by forming the combinations produced by the different chosen values of hyper-parameters and the number of repetitions of the hidden LSTM layer ($2 \times 3 \times 3 = 18$).

TABLE IV: Chosen values of hyper-parameters for the RNN model

Hyper-parameter	Values
Dimensionality of embedding layer	{16, 32}
LSTM units	{32, 64, 128}
Maximum epochs	{50}

As described earlier, we pick the best performing hyper-parameters and number of epochs and retrain the model to obtain the final and best performance of the model.

3) Autoencoder model: Autoencoders are neural networks that can learn meaningful representations of the data in an unsupervised way. There exist diverse variants of autoencoders, however, in practice the purpose of all variants is to learn to reconstruct a representative copy of the given input. To this end, a bottleneck-like part between the input and the output layers encodes the input in a compressed representation which is in turn decompressed by a decoding part. The underlying principle is that the encoded representation captures salient features which are reflected in the reconstructed output and discards other, less important, thus providing dimensionality reduction and de-noising capabilities [62].

A typical autoencoder model has essentially two sets of layers—encoding and decoding layers—symmetrically built across the compression pipeline. The model produces an approximate, compressed representation of the input, which then attempts to reconstruct with some loss $\mathcal{L}$. In its simplest architectures an autoencoder consists of dense layers where the input is compressed by limiting the number of units in the intermediate hidden layers. Compression can also be implemented by imposing sparsity constraints on the hidden units that are being activated [143]; this is effectuated by some regularization technique that adds a penalty term to the loss function. Besides autoencoder models implemented with dense layers, more complex architectures involve RNN and CNN hidden layers.

In the context of smell detection we experiment with a variety of autoencoder architectures, ranging from simple models built with dense layers, to more sophisticated models involving RNN and CNN hidden layers. We build the simple sparse autoencoder models with dense layers where we reduce the number of units in the intermediate layers and penalize the loss function through the L1-regularization procedure [144]. We build more complex models by interpolating LSTM or CNN layers with reduced dimensions between the input and the output. We use all variants of the autoencoders as classifiers of anomalies. We train the models to learn to represent patterns of non-smelly samples by using only negative (*i.e.*, non-smelly) examples. We test the trained models on data that include both positive and negative samples. We use the reconstruction loss as a proxy for classifying an instance

as smelly [145]–[147]. If for some instance the output of the model shows high loss, we accept that this example does not follow the pattern learnt by the model, which in turn implies classification of a positive instance of the smell under investigation.

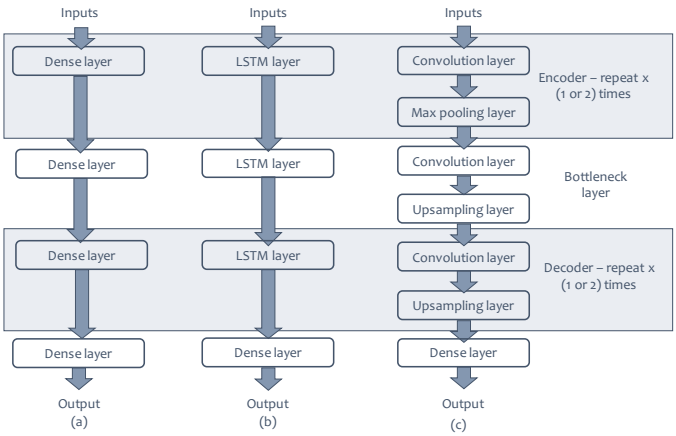


Fig. 5: Architecture of the employed Autoencoder models

As Figure 5 shows, we have employed three variants of autoencoder models; the first variant uses dense, the second uses RNN, and the last variant mainly uses CNN-1D layers as the fundamental component that forms the model's encoder and decoder layers. The convolution variant uses max pooling and upsampling layers with convolution layers in encoder and decoder respectively. Table V lists the hyper-parameters used for the autoencoder model. The *number of units in the dense layer* is the dimension of the output space of the layer, LSTM *units* is the number of units in each LSTM layer, and *filters in the convolution layer* is the number of convolutional filters applied. *Kernel size* controls the size of the convolution window, and *pooling window size* governs the size of down/up-sampling window during the pooling operation. For LSTM layers, we set the values of *dropout* and *recurrent_dropout* to 0.1. The encoder and decoder layers are followed by a fully-connected dense layer. Once the training is complete, we find out the optimal performance of the trained autoencoder model by evaluating the performance at different values of the threshold.

TABLE V: Chosen values of hyper-parameters for the Autoencoder model

Hyper-parameter	**Values**
Number of units (Dense)	{256, 512, 1024}
LSTM units	{8, 16, 32}
Filters in convolution layer	{8, 16, 32, 64}
Kernel size in convolution layer	{5, 7, 11}
Pooling window size in max pooling and upsampling layer	{2, 3, 4, 5}
Epochs	{20}

E. Hardware Specification

We perform all the experiments on the super-computing facility offered by GRNET (Greek Research and Technology Network). The experiments were run on GPU nodes (8x NVidia V100). Each GPU incorporates 5120 CUDA cores. We requested 1 GPU node with 64 GB of memory for most of the experiments while submitting the job to the super computing facility. Some RNN experiments require more memory to perform the training; we requested 128 GB of memory for these.

V. Results and Discussion

As elaborated in this section, we found that it is feasible to detect smells using deep learning models without extensive feature engineering. Our results also indicate that performance of deep learning models is highly smell-specific. Furthermore, we found that it is feasible to apply transfer-learning in the context of code smells detection. In the rest of the section, we discuss the results in detail.

A. Results of RQ1

RQ1 **Is it possible to detect code smells using deep learning methods? If yes, which deep learning method performs superior?**

1) Approach: We prepare the input samples as described in Section IV-B. Table VI presents the number of positive and negative samples used for each smell for training and evaluation; CNN-1D, RNN, and AE use 1D samples and CNN-2D uses 2D samples. As mentioned earlier, we train our models with the same number of positive and negative samples (except in the case of AE where we use only negative samples to train the model). The one-dimensional sample counts are different from their two-dimensional counterparts because we apply additional constraint for outlier exclusion, on permissible height, in addition to the width.

TABLE VI: Number of positive (P) and negative (N) samples used for training and evaluation for RQ1

	CNN-1D, RNN, and AE			CNN-2D		
	Training	Evaluation		Training	Evaluation	
	P and N	P	N	P and N	P	N
CM	5,000	7,489	139,460	5,000	5,822	125,807
CC	4,330	1,856	145,437	3,374	1,446	129,933
FE	1,260	528	50,000	1,194	512	38,963
MA	205	85	50,000	189	82	39,071

2) Results: Figure 6 presents the performance (*i.e.*, F1 score) of the models for the considered smells for all the configurations that we experimented with. The results from each model show that performance of the models varies depending on the smell under analysis. Another observation from the trendlines shown in the plots is that performance of all the models remains more or less stable and unchanged for different configurations except for the RNN model with the *complex method* smell. This implies

that despite the variability in the combinations of hyper-parameters that we experimented with, the effect on the particular models appears to be minor.

Table VII presents the results of Mann-Whitney U test that we perform to ensure that each model performs differently than the other models. We also compute Hedges' `g` [148] to figure out the effect size of the difference between each pair of deep learning models. Hedges' `g` is similar to Cohen's `d` [149] except the Hedges' metric takes into account different sample sizes. The results in the Table show that almost all the model pairs are different and their effect size is significant.

Figure 7 presents the box plots comparing for each smell, the performance of all trained models, under all configurations. For all the analyzed smells, autoencoders outperform all of the other models. The F1 score values produced by all three variants of the autoencoder model are highly concentrated. The figure also shows that the variations in hyper-parameters do not affect the performance of the chosen autoencoder model. We also observe that the performance of individual model architectures vary from smell to smell; for instance, RNN shows small variance for *feature envy* smell but quite large for *complex method* smell.

Equipped with experiment results, we attempt to validate the hypotheses. We present precision, recall, and F1 score to show the performance of the analyzed deep learning models. We attempt to validate each of the addressed hypotheses in the rest of the section.

RQ1.H1: *It is feasible to detect code smells using deep learning methods.*

Table VIII lists performance metrics (precision, recall, F1 score, MCC (Matthews Correlation Coefficient)) for the optimal configuration for each smell, comparing all four deep learning models. We present MCC also along with other accuracy metrics because MCC covers true negative instances as well which is not covered by the F1 score [149]. The table also lists the hyper-parameters associated with the optimal configuration for each smell. Figure 8 presents the performance (F1 score) of the deep learning models corresponding to each smell considered in this exploration. We use fully-connected neural network variant for AE in this experiment.

As regards implementation smells, for the *complex method* smell, even though autoencoders and RNN perform superior than the convolution models, the performance of all models under consideration is comparable. On the other hand, none of the models could identify *complex conditional* smell with a reasonable accuracy. This implies that the models could identify a smell that is exhibited through the structure of a method but could not successfully spot the smell characterized by micro-structure representing the conditional statements.

> The autoencoder models with a simple dense layer (two for *feature envy* smell) perform superior compared to more complex models based on CNN and RNN.

Both of the design smells—*feature envy* and *multifaceted abstraction*—are non-trivial smells. Their detection requires analysis of method interactions to observe respectively coupling of a method with other classes, and incohesiveness of a class. For *feature envy*, autoencoders perform better than the other models; however, for *multifaceted abstraction* none of the employed deep learning models could capture the complex characteristics of the smell, implying that the token–level representation of the data may not be appropriate for capturing higher–level features required for detecting the smell.

> It is evident from the above discussion that the hypothesis exploring the feasibility of detecting smells using deep learning models holds true; however, the performance of the employed models differ significantly depending upon the smells.

RQ1.H2: CNN-2D *performs better than* CNN-1D *in the context of detecting smells.*

Table VIII shows that the performance of CNN-1D and CNN-2D is comparable for *complex method* and *feature envy* smells. For *complex conditional* smell, CNN-2D does better than CNN-1D; probably due to a complex conditional statement contributes to a longer statement and CNN-2D could better identify it compared to CNN-1D using its 1-D form. Neither of the models could detect *multifaceted abstraction* smell instances. In summary, there is not sufficient evidence to conclude that CNN-2D is a superior model compared to CNN-1D.

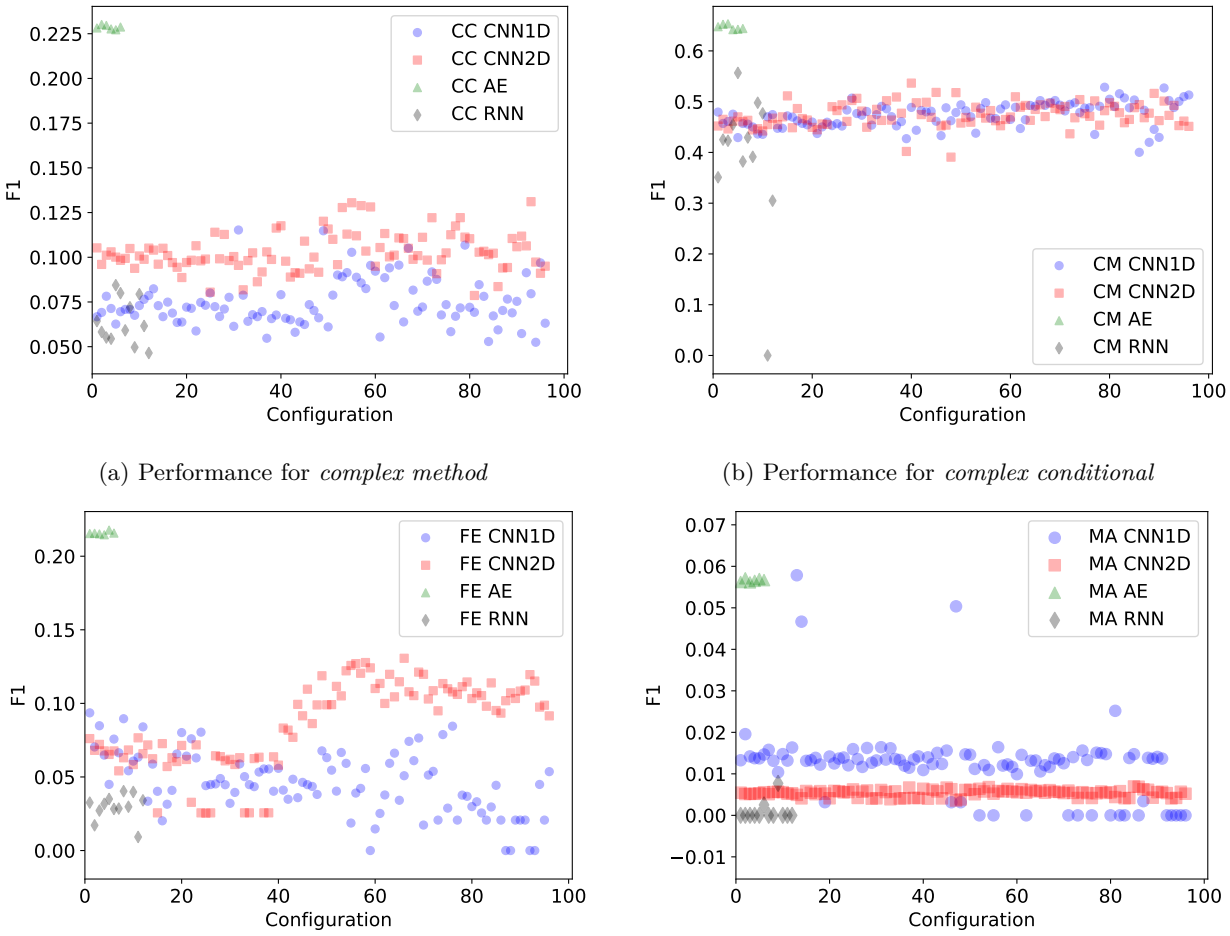
> Therefore, we reject the hypothesis that CNN-2D performs overall better than CNN-1D as none of the models is clearly superior to another in all the cases.

RQ1.H3: RNN *models perform better than* CNN *models in the context of detecting smells.*

Table IX presents the comparison of RNN with CNN-1D and CNN-2D by comparing pairwise F1 measure differences in percentages, where the F1 score values are obtained by the optimal configuration in each case. Here, the performance difference in percentage is calculated by

$$\frac{F1_{RNN} - F1_{CNN}}{F1_{RNN} \times 100}$$

The RNN performs better for *complex method* smell against both convolution models. For *complex conditional* smell, CNN-2D performs better than both CNN-1D and RNN probably due to 2-D input samples could better represent the



(a) Performance for *complex method*

(b) Performance for *complex conditional*

(c) Performance for *feature envy*

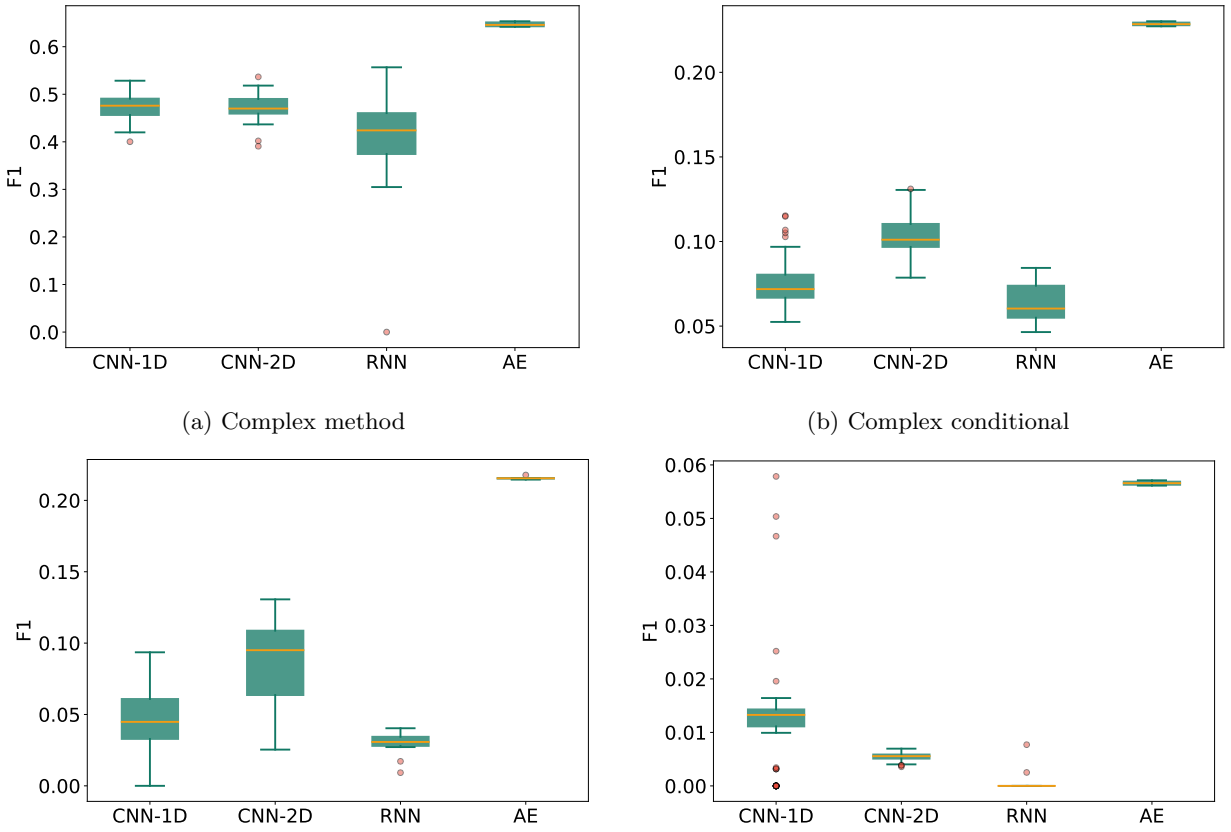
(d) Performance for *multifaceted abstraction*

Fig. 6: Scatter plots of the performance (F1 score) exhibited by the considered deep learning models along with their corresponding trendline

TABLE VII: Results of Mann-Whitney U test and Hedges' g effect size between the F1 values for all configurations of each considered model

		CNN-2D	RNN	AE
CM	CNN-1D	p=0.972, g=-0.015	p=0.001, g=1.60	p= 4.33e-05, g=-7.12
	CNN-2D	–	p=0.001975, g=1.62	p=4.336e-05, g=-7.52
	RNN	–	–	p=7.7e-6, g=-2.09
CC	CNN-1D	p<2.2e-16, g=-2.34	p=0.01, g=0.84	p=4.33e-05, g=-11.96
	CNN-2D	–	p=2.58e-08, g=3.55	p=4.33e-05, g=-11.86
	RNN	–	–	p=7.7e-6, g=-15.13
FE	CNN-1D	p<2.2e-16, g=-1.56	p=0.002, g=0.8	p=4.31e-05, g=-8.03
	CNN-2D	–	p=1.54e-06, g=2.03	p=4.33e-05, g=-4.57
	RNN	–	–	p=7.7e-6, g=-24.08
MA	CNN-1D	p=4.77e-13, g=1.07	p=4.45e-06, g=1.32	p=6.07e-05, g=-4.97
	CNN-2D	–	p=2.74e-06, g=4.23	p= 4.33e-05, g=-63.98
	RNN	–	–	p=2.62e-4, g=-27.94

complex nature of conditional statements compared to its 1-D input form. However, the performance of RNN is lower for *feature envy* compared to both convolution models. Also, for *complex conditional* smell, RNN shows poorer performance compared to CNN-2D. To detect *feature envy* smell, it is required to identify complex relationships among methods and data members which RNN could not grasp.



(a) Complex method

(b) Complex conditional

(c) Feature envy

(d) Multifaceted abstraction

Fig. 7: Box plots of the performance (F1 score) exhibited by the considered deep learning models for all the four smells

TABLE VIII: Performance (Precision, Recall, F1 score, MCC (Matthews Correlation Coefficient)) of all four models with configuration corresponding to the optimal performance. L: deep learning layers; F: number of filters; K: kernel size; MPW: maximum pooling window size; ED: embedding dimension; LSTM: number of LSTM units; E: number of epochs; U: number of units; T: threshold.

		Performance				Configuration								
	Smell	P	R	F1	MCC	L	F	K	MPW	ED	LSTM	E	U	T
CNN-1D	CM	0.46	0.60	0.52	0.54	2	32	5	4	–	–	15	–	–
	CC	0.04	0.68	0.08	0.09	1	32	5	4	–	–	15	–	–
	FE	0.03	0.69	0.06	0.07	1	8	11	2	–	–	31	–	–
	MA	0.01	0.98	0.02	0.02	1	16	11	2	–	–	5	–	–
CNN-2D	CM	0.40	0.81	0.54	0.58	1	64	11	5	–	–	36	–	–
	CC	0.07	0.60	0.13	0.14	2	64	7	2	–	–	22	–	–
	FE	0.05	0.77	0.09	0.10	2	16	5	3	–	–	14	–	–
	MA	0.01	0.92	0.02	0.02	2	64	11	2	–	–	6	–	–
RNN	CM	0.61	0.66	0.63	0.67	1	–	–	–	32	64	24	–	–
	CC	0.04	0.65	0.08	0.10	1	–	–	–	32	64	3	–	–
	FE	0.01	0.85	0.02	0.02	2	–	–	–	16	64	16	–	–
	MA	0.00	0.07	0.01	0.01	2	–	–	–	16	128	11	–	–
AE	CM	0.60	0.68	0.64	0.67	1	–	–	–	–	–	20	32	319,000
	CC	0.20	0.20	0.20	0.21	1	–	–	–	–	–	20	16	328,000
	FE	0.18	0.24	0.21	0.22	2	–	–	–	–	–	20	16	325,000
	MA	0.03	0.14	0.05	0.06	1	–	–	–	–	–	20	16	328,000

The analysis suggests that performance of the deep learning models is smell-specific. Therefore, we reject the hypothesis that RNN models perform better than CNN models for all considered smells.

RQ1.H4: *RNN and CNN variants of autoencoder model exhibit comparable performance to those of* RNN *and* CNN-1D *models.*

For *complex method*, RNN performs better than CNN-1D (refer to Figure 8); however, within autoencoder model,

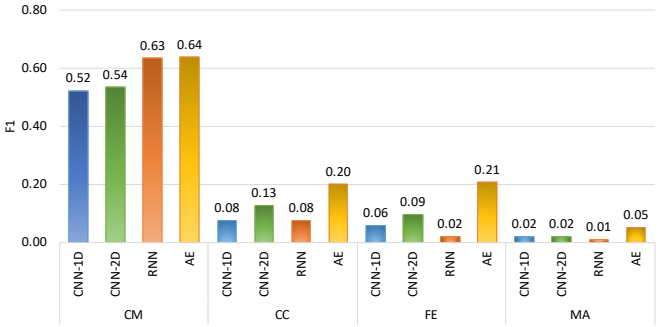


Fig. 8: Comparative performance of the deep learning models for each considered smell

TABLE IX: Performance (F1 score) comparison of RNN with CNN-1D and CNN-2D

Smell	RNN vs CNN-1D	RNN vs CNN-2D
CM	16.71%	16.25%
CC	13.38%	-86.31%
FE	-153.57%	-159.62%
MA	-31.16%	9.65%

CNN-1D is slightly better than RNN variant. Similarly, CNN-1D does better compared to RNN for *feature envy* smell but CNN-1D and RNN variants of autoencoder model show same performance. On the other hand, for *complex conditional* smell, RNN and CNN-1D both show similar performance in both configurations.

With this comparison, it is evident that RNN and CNN-1D variants of autoencoder model do not exhibit similar performance pattern as shown by individual RNN and CNN-1D models.

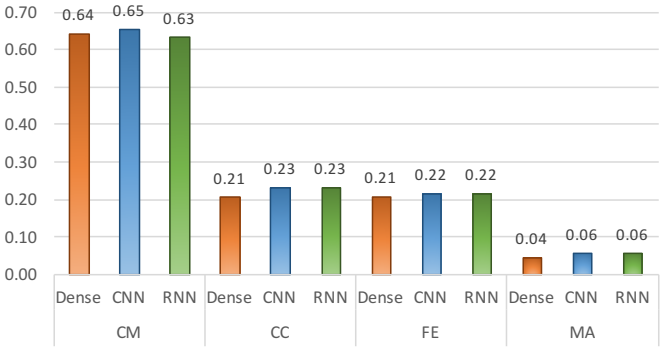


Fig. 9: Comparative performance of variants of autoencoder models for each considered smell

3) Implications: This is the first attempt in the software engineering literature to show the feasibility of detecting smells using deep learning models from the tokenized source code without extensive feature engineering. It may motivate researchers and developers to explore this direction and build over it. For instance, *context* plays an important role in deciding whether a reported smell is actually a quality issue for the development team. One of the future works that the community may explore is to combine the models trained using samples classified by the existing smell detection tools with the developer's feedback to identify more relevant smells considering the context.

Our results show that, even though both convolution methods perform superior for specific smells, their performance is comparable for each smell. This implies that we may use one-dimensional or two-dimensional CNN interchangeably without compromising the performance significantly.

Apart from experimenting with CNN and RNN-based models in various configurations, we also considered autoencoder. The autoencoder model treats a smells as a rare event; a simple autoencoder with one mid dense layer performs equally well with the more complex and deeper autoencoder configurations and better than the RNN and CNN based models. This observation provides grounds for further investigation, encouraging the software engineering community to propose simpler models for smell detection.

The comparative results on applying diverse deep learning models for detecting different types of smells suggest that a model is highly dependent on the kind of smells that it is trying to classify. This result could attract efforts from the software engineering community to develop smell-specific smell detection deep learning models.

B. Results of RQ2

RQ2 **Is it possible to detect code smells by applying transfer-learning techniques on similar languages? If yes, which deep learning model exhibits superior performance in detecting smells when applied in transfer-learning setting?**

1) Approach: In the case of direct-learning, the training and evaluation samples belong to the same programming language whereas in the transfer-learning case, the training and evaluation samples come from two similar but different programming languages. This research question examines the feasibility of applying transfer-learning *i.e.,* train neural networks by using C# samples and employ the trained model to classify code fragments written in Java.

For the transfer-learning experiment (referred to as *TL*) we keep the training samples exactly the same as the ones we used in RQ1. For evaluation, we download repositories containing Java source code and preprocess the samples as described in Section IV-B. Similar to RQ1, evaluation is performed on a realistic scenario, *i.e.,* we use all the positive and negative samples from the selected repositories and when enforcing maximum limit to samples we maintain the original ratio between positive and negative samples. This arrangement ensures that the models would perform as reported if employed in a real-world application. Table X shows the number of samples

used for training and evaluation for this research question.

TABLE X: Positive (P) and negative (N) number of samples used for training and evaluation for RQ2

	CNN-1D, RNN, and AE			CNN-2D		
	Training	Evaluation		Training	Evaluation	
	P and N	P	N	P and N	P	N
CM	5,000	10,244	150,000	5,000	5,818	150,000
CC	4,329	3,440	150,000	3,374	2,724	150,000
FE	1,260	613	50,000	1,194	682	50,000
MA	205	148	50,000	189	158	50,000

2) Results: As an overview, Figure 10 shows the scatter plots for each deep learning model comparing the performance (F1 score) of both the direct-learning and transfer-learning for all the considered smells for all the configurations. These plots outline the performance exhibited by the models in both cases with trend lines distinguishing the compared series. The plots imply that the performance of the models are comparable in the transfer-learning and direct-learning cases. In the rest of the section, we report quantitative results on applying transfer learning between C# and Java.

RQ2.H1: *It is feasible to detect code smells by applying transfer-learning techniques on similar languages.*

Table XI presents the performance of the models for all the considered smells demonstrating strong evidence on the feasibility of applying transfer-learning for smell detection. The performance pattern is in alignment to that in the direct-learning case; Spearman correlation between the performance produced by direct-learning and transfer-learning is 0.88 (with p-value = 6.56×10^{-6}).

> Therefore, we accept the hypothesis that transfer-learning is feasible in the context of code smells detection.

Figure 11 presents a comparison among the performance (*i.e.,* F1 score) exhibited by all the considered deep learning models for each considered smell. Interestingly, CNN-2D performs superior to the rest of the models for all smells except *feature envy*; for *feature envy* smell, AE performs best. As indicated above, in general, the performance of the models follows a similar trend with the one observed in the case of direct-learning in RQ1.

RQ2.H2: *The performance of transfer-learning is inferior compared to that of direct-learning.*

Figure 12 compares the performance of the models at their optimal configurations applied in transfer-learning and in direct-learning. We observe that, in the majority of cases, direct-learning performs better than the corresponding transfer-learning models. The exceptions are convolution models for *complex conditional*, RNN for *feature envy*, CNN-2D and AE for *multifaceted abstraction* smell, where transfer-learning shows better results.

We perform an additional experiment (referred to as $TL_{reverse}$) in which we reverse the direction of transfer-learning *i.e.,* we train the models using Java samples and evaluate the trained models on C# samples. We download Java repositories and perform the data curation operations mentioned in Section IV-B to compile a set of training and evaluation samples. Table XII presents the number of samples that are used for this experiment.

We perform the experiment using the optimal configuration identified earlier and presented in Table XI. We present the obtained results in Table XIII for all the models with all the smells. We observe that the performance of the models in $\text{TL}_{reverse}$ experiment follows the similar pattern as we have seen in TL. We carry out Spearman correlation analysis between the performance of TL and $\text{TL}_{reverse}$ experiments. We found a strong correlation between the two with $\rho = 0.795$ (p-value=0.0002).

The ratio of positive and negative samples plays a significant role in the performance of a deep-learning model [16], [137] and hence it is not easy to compare the performance of models trained with heterogeneous samples. We compute Normalized Performance Difference (NPD) to compare the performance of models from direct-learning to transfer-learning. Normalized performance difference between methods `i` and `j` is given by the following equation.

$$\text{NPD(i, j)} = \frac{F1_i \times R_j - F1_j \times R_i}{R_i + R_j} \tag{1}$$

Here, F1_i and F1_j represent the performance (*i.e.,* F1 score) and R_i and R_j refer to the ratio between negative and positive samples of each method. Table XIV presents the comparison of performance in the terms of both simple difference and normalized difference. Simple performance difference (PD) shows that transfer-learning performs inferior in the majority of instances; however, the normalized performance difference (NPD) that scales the difference between performance proportionally indicates that transfer-learning does not have inferior performance compared to direct-learning.

> Therefore, we reject the hypothesis that transfer-learning performs inferior compared to direct-learning.

The above discussion leads to another interesting question: *which deep learning model's performance is the most or least sensitive to transfer-learning?* We compute the NPD between the performance pairs of direct-learning and transfer-learning for each considered model. Figure 13 depicts the results; CNN-1D shows the highest difference in performance and hence CNN-1D is the most sensitive model to transfer-learning in this experiment. RNN on the other hand, shows the lowest difference in performance which renders the model the least sensitive and, consequently, the

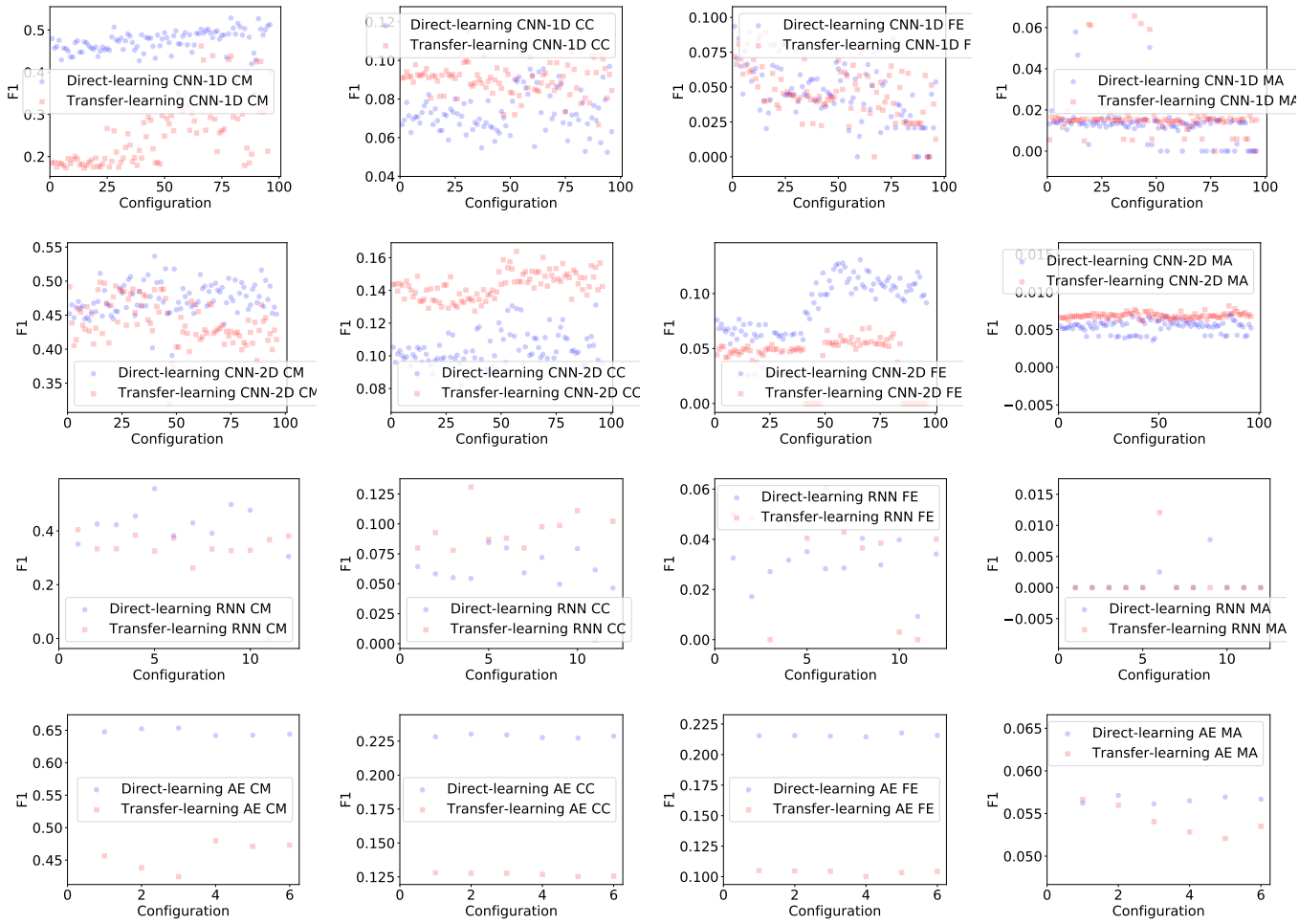


Fig. 10: Scatter plots for each model and each considered smell comparing F1 score of direct-learning and transfer-learning along with corresponding trend-lines

TABLE XI: Performance of all four models with configuration corresponding to the optimal performance. L: deep learning layers; F: number of filters; K: kernel size; MPW: maximum pooling window size; ED: embedding dimension; LSTM: number of LSTM units; E: number of epochs; U: number of units; T: threshold.

		Performance			Configuration								
	Smell	P	R	F1	L	F	K	MPW	ED	LSTM	E	U	T
CNN-1D	CM	0.36	0.59	0.44	2	16	5	3	–	–	17	–	–
	CC	0.08	0.18	0.11	2	32	7	2	–	–	9	–	–
	FE	0.02	0.78	0.04	1	16	11	5	–	–	49	–	–
	MA	0.01	0.78	0.01	1	64	11	5	–	–	5	–	–
CNN-2D	CM	0.36	0.82	0.50	1	16	11	4	–	–	30	–	–
	CC	0.10	0.45	0.16	2	8	7	2	–	–	19	–	–
	FE	0.03	0.37	0.06	2	16	7	3	–	–	24	–	–
	MA	0.04	0.29	0.07	2	64	11	2	–	–	17	–	–
RNN	CM	0.31	0.57	0.40	1	–	–	–	16	32	5	–	–
	CC	0.07	0.55	0.13	1	–	–	–	32	32	4	–	–
	FE	0.03	0.64	0.06	1	–	–	–	32	128	10	–	–
	MA	0.01	0.02	0.01	1	–	–	–	32	128	9	–	–
AE	CM	0.53	0.44	0.48	2	–	–	–	–	–	20	8	328,000
	CC	0.09	0.23	0.13	1	–	–	–	–	–	20	8	328,000
	FE	0.08	0.15	0.10	1	–	–	–	–	–	20	8	328,000
	MA	0.03	0.20	0.06	1	–	–	–	–	–	20	8	328,000

most robust for transfer-learning compared to the other models of this experiment.

3) Implications: Our results demonstrate that it is feasible to apply transfer-learning in the context of smell

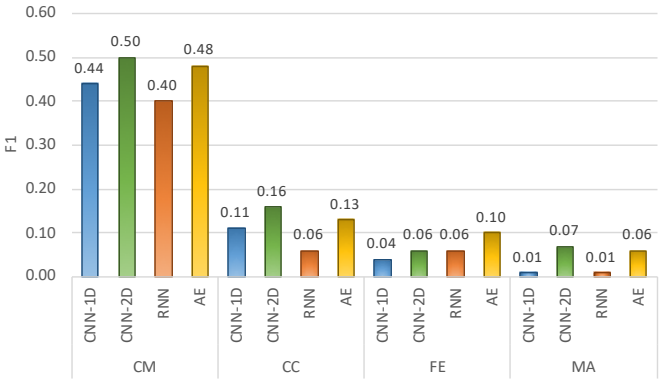


Fig. 11: Comparative performance of the deep learning models for each considered smell in transfer-learning settings

TABLE XII: Number of positive (P) and negative (N) samples used for training (Java samples) and evaluation (C# samples)

	CNN-1D and RNN			CNN-2D		
	Training	Evaluation		Training	Evaluation	
	P and N	P	N	P and N	P	N
CM	5,000	7,760	150,000	5,000	7,117	150,000
CC	5,000	1,843	150,000	5,000	1,669	150,000
FE	2,183	496	50,000	1,987	624	50,000
MA	545	82	50,000	483	105	50,000

detection. Exploiting this approach can lead to a new category of smell detection tools, specifically for the programming languages where no mature smell detection tools are available.

C. Discussion

Although it is possible to detect some code smells using deep learning models, the presented method is by no means universal, and the outcome is sensitive to the training set composition and the training time. In the rest of the section, we elaborate on these observations emerging from the presented results.

1) Is there a silver-bullet?: In practical setting one would want to employ a universal model architecture that performs consistently well for all the considered smells; this would make the implementation of tools simpler.

RNN has the reputation to perform well with textual data and sequential patterns while CNN is considered good for imaging data and visual patterns. Given the similarity of source code and natural language, it is expected to obtain good performance from RNN. Our results show that RNN outperforms both CNN models in the cases of *complex method*; however, it does not live up to its reputation for other smells. AE is considered to be a good mechanism for learning to create copies of a given input where the key features are maintained; we observed that it works considerably well compared to other considered models. We have a uniform architecture for each model and we observed that the performance of the model differs significantly for different smells. It suggests that it is non-trivial, if not impossible, to propose a universal model architecture that works for all smells. Each smell exhibits diverse distinctive features and hence their detection mechanisms differ significantly. Therefore, given the nature of the problem, it is unlikely that one universal model architecture will be the silver-bullet for the detection of a wide range of smells with consistently good performance.

2) Performance comparison with baseline: A comparison with existing methods and tools is expected from a study proposing a new method. However, it is not feasible to compare the results presented in this paper with other attempts that use machine learning for smell detection [9], [11]–[14], [69], [105] due to the following reasons. First, the replication packages of the related attempts are not available. Second, for most of the existing attempts, the ratio of positive and negative evaluation samples is not known; in the absence of this information, we cannot compare them with our results fairly since the ratio plays an important role in the performance of machine learning models. Furthermore, the existing approaches compute metrics and feed them to machine learning models as features. Models that only use metrics as the features can be as good as the metrics themselves. Metrics do not incorporate the context and hence the machine learning models based on the metrics do not exploit the power of machine learning because the models are used merely for selecting a threshold for the input metrics to classify smelly code from non-smelly code. This research attempts to move beyond the use of metrics as the only data source to detect the smells and bring more context sensitivity to the smell analysis. To the best of our knowledge, this is the first attempt to detect smells without using metrics as the features for the employed machine learning models. Due to this reason, it would be unfair to compare any machine learning model that uses metrics with the presented method. Also, it would be unfair to machine-learning based methods if we feed the raw tokenized source code as input because unlike deep learning methods, machine learning algorithms (such as Bayesian belief networks and support vector machines), treat each column of input as a specific feature. This assumption will not hold if tokenized source code is provided as input and the algorithms will perform very poorly.

The above discussion indicates two additional aspects. First, we, as software engineering community, need a manually validated gold-standard smells dataset for a wide range of smells. This would make bench-marking and comparison among different smell detection methods easy. Second, though this study shows that deep learning methods can grasp by themselves latent features that are necessary to identify smells and paves the way to make the smell detection more context sensitive, it is far from the level where it can be compared head-to-head with existing methods and surpass them in performance. In the future, we would like to explore combining source code in

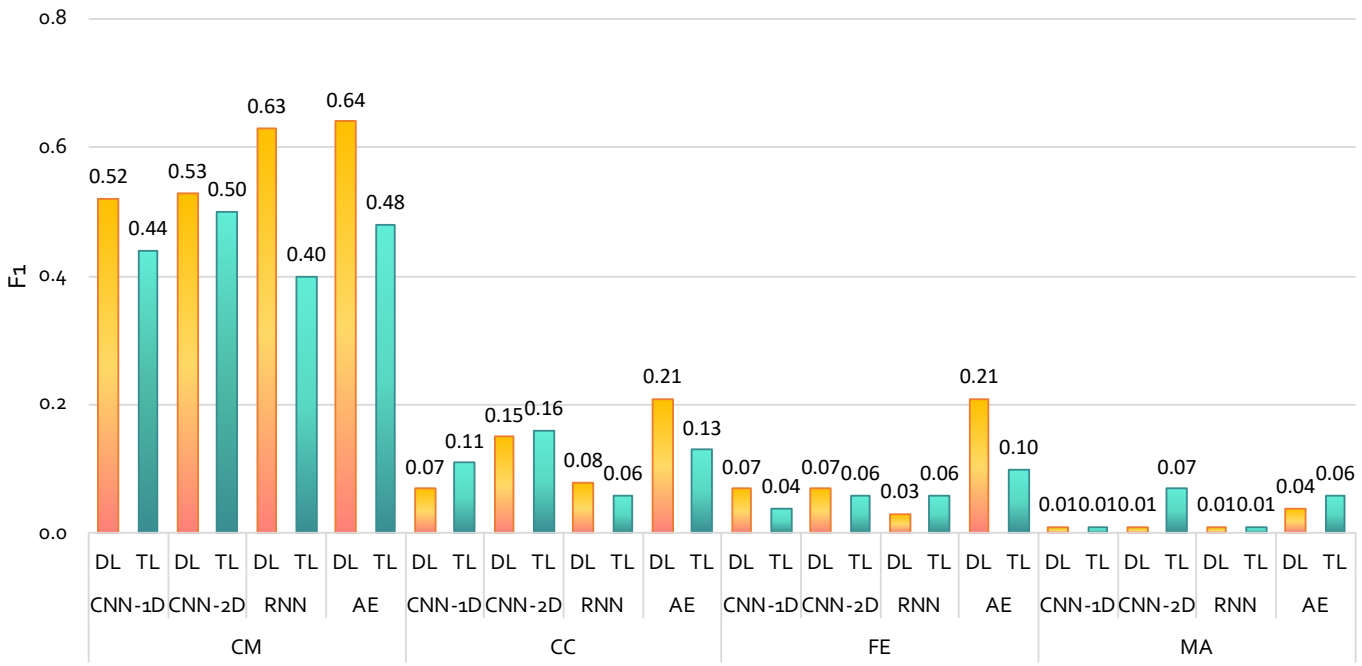


Fig. 12: Comparison of performance of the deep learning models between direct-learning (DL) and transfer-learning (TL) settings

TABLE XIII: Performance of all four models with configuration corresponding to the optimal performance in $\text{TL}_{reverse}$ experiment. L: deep learning layers; F: number of filters; K: kernel size; MPW: maximum pooling window size; ED: embedding dimension; LSTM: number of LSTM units; E: number of epochs; U: number of units; T: threshold

		Performance			Configuration								
	Smell	P	R	F1	L	F	K	MPW	ED	LSTM	E	U	T
CNN-1D	CM	0.06	0.87	0.11	2	16	5	3	–	–	17	–	–
	CC	0.03	0.69	0.05	2	32	7	2	–	–	9	–	–
	FE	0.02	0.83	0.04	1	16	11	5	–	–	49	–	–
	MA	0.00	1.00	0.01	1	64	11	5	–	–	5	–	–
CNN-2D	CM	0.28	0.93	0.43	1	16	11	4	–	–	30	–	–
	CC	0.01	0.93	0.03	2	8	7	2	–	–	19	–	–
	FE	0.03	0.73	0.06	2	16	7	3	–	–	24	–	–
	MA	0.01	0.18	0.01	2	64	11	2	–	–	17	–	–
RNN	CM	0.37	0.40	0.38	1	–	–	–	16	32	5	–	–
	CC	0.05	0.02	0.03	1	–	–	–	32	32	4	–	–
	FE	0.02	0.92	0.04	1	–	–	–	32	128	10	–	–
	MA	0.00	0.11	0.01	1	–	–	–	32	128	9	–	–
AE	CM	0.42	0.43	0.43	2	–	–	–	–	–	20	8	328,000
	CC	0.06	0.26	0.09	1	–	–	–	–	–	20	8	328,000
	FE	0.04	0.35	0.07	1	–	–	–	–	–	20	8	328,000
	MA	0.01	0.22	0.01	1	–	–	–	–	–	20	8	328,000

tokenized form with more refined features to help deep learning methods classify the smelly code with superior performance.

We compare our results with the results obtained from two baseline random classifiers that do not really learn from the data but use only the distribution of smells in the training set to form their predictions. Table XV presents the comparison. The first random classifier generates predictions by following the training set's class distribution: that is, for every sample in the evaluation set it predicts whether it is a smell or not based on the *frequency* of smells in the training data. We did that for both balanced and imbalanced evaluation samples to mimic the learning process of the actual experiment. In the middle three columns, referred to as "RC *(frequency)*", of the table we show the results for the balanced setting, as they were better than the results for the imbalanced setting. The second random classifier predicts that a smell is always present; this gives perfect recall, but low precision, as observed in the columns corresponding to "RC *(all smells)*" of the table. Overall, our models perform far better than a random classifier for all but *multifaceted abstraction* smell for both baseline variants.

TABLE XIV: Comparison of performance of transfer-learning with direct-learning. Here, DL and TL refer to performance of deep learning models in direct-learning and transfer-learning respectively. PD and NPD refer to simple and normalized performance difference between direct-learning and transfer-learning respectively. R_{DL} and R_{TL} refer to the ratio of negative to positive samples for direct-learning and transfer-learning respectively

	Smell	DL	TL	PD	R_{DL}	R_{TL}	NPD
CNN-1D	CM	0.52	0.44	0.08	18.62	14.64	-0.02
	CC	0.07	0.11	-0.04	78.36	43.60	-0.05
	FE	0.07	0.04	0.03	94.70	81.57	0.01
	MA	0.01	0.01	0	588.24	337.84	0.00
CNN-2D	CM	0.53	0.5	0.03	21.61	25.78	0.06
	CC	0.15	0.16	-0.01	89.86	55.07	-0.04
	FE	0.07	0.06	0.01	76.10	73.31	0.00
	MA	0.01	0.07	-0.06	476.48	316.46	-0.04
RNN	CM	0.63	0.4	0.23	18.62	14.64	0.05
	CC	0.08	0.06	0.02	78.36	43.60	-0.01
	FE	0.03	0.06	-0.03	94.70	81.57	-0.02
	MA	0.01	0.01	0	588.24	337.84	0.00
AE	CM	0.64	0.48	0.16	18.62	14.64	0.01
	CC	0.21	0.13	0.08	78.36	43.60	-0.01
	FE	0.21	0.1	0.11	94.70	81.57	0.04
	MA	0.04	0.06	-0.02	588.24	337.84	-0.02

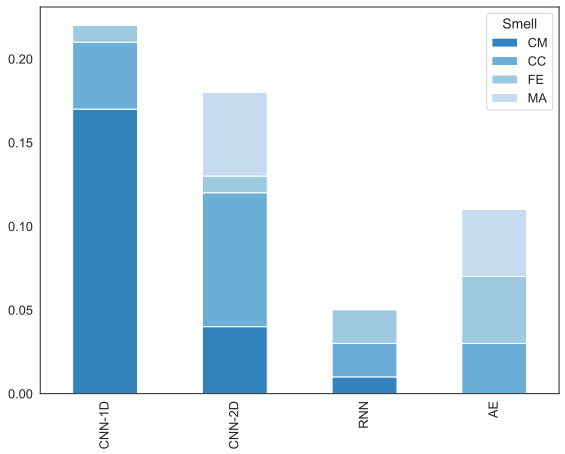


Fig. 13: Difference in performance of the deep learning models and the sample ratio for two transfer-learning tasks

3) Poor performance in detecting design smells: The presented neural networks perform very poorly when it comes to detecting the design smells *feature envy* and *multifaceted abstraction*. We infer the following two reasons for this under-performance. First, design smells such as *feature envy* and *multifaceted abstraction* are inherently difficult to spot unless a deeper semantic analysis is performed. Specifically, in the case of *multifaceted abstraction*, interactions among the methods of a class as well as the member fields have to be modeled in order to observe cohesion among the methods. This is a non-trivial aspect and the neural networks could not spot this aspect with the provided representation of the data. Therefore, we need to provide refined information in the form of engineered features along with the source code to help neural networks identify the inherent patterns. Second, the number of positive training samples were very low, thus significantly restricting our training set. The low number severely impacts the ability of neural networks to infer the responsible aspects that cause the smell. The future research attempts could address this limitation by increasing the number of repositories under analysis or by adopting techniques such as careful formation of artificial samples.

4) Variation in training-time: As observed in the results section, performance of the considered deep-learning models varies depending upon the smells. However, we also note that the models show considerable difference in the time consumed for training. We logged the time taken by each experiment for the comparison. Table XVI presents the average time taken by each model for each smell per epoch. The table shows that the RNN is consuming exorbitant amount of time compared to CNN and autoencoders. In the context of this study, this implies that if the performance of an RNN for a given task is comparable to that of a CNN or an autoencoder model, one should decline the RNN-based solution for significantly faster training time.

5) Exploring other source code representations: In the recent times, we have witnessed a surge in the research towards alternative source code representations. The thriving progress on code mining research and specifically the increasing interest on problems pertinent to semantic code search has recently led to novel code-specific representations that incorporate structural features of a program [90], [108], [109]. Such representations have been proven to perform well on the task they are tailored for, that is, learning semantics of specific source code fragments. Even though the afore-mentioned representations have been proven effective for capturing the semantics of programs, their generalizability to other code mining downstream tasks is questionable [150]. In our study we aimed to address the problem of training models that capture qualitative characteristics in code. These are mostly manifest through syntactic features in the case of implementation smells and through class-specific method interaction in the case of design smells. Consequently, smell detection is agnostic to the semantics of the program under investigation. However, as a reference point we carried out an experiment using the current mainstream method for code representation, that is, *code2vec*.

We used the implementation provided by the authors of *code2vec*[6] and modified it to suit our context. The original *code2vec* implementation is tightly coupled with the problem that its authors address and hence it was a non-trivial challenge to customize it to the needs of our classification problem. We changed the implementation to predict the

[6]https://github.com/tech-srl/code2vec

TABLE XV: Comparison of performance (Precision, Recall, and F1) with a random classifier (RC) following the training set frequencies or responding always indicating a smell

		Performance								
		Our results			RC (frequency)			RC (all smells)		
	Smell	P	R	F1	P	R	F1	P	R	F1
CNN-1D	CM	0.48	0.58	0.52	0.05	0.50	0.09	0.05	1	0.09
	CC	0.04	0.70	0.07	0.01	0.50	0.02	0.01	1	0.02
	FE	0.03	0.69	0.07	0.01	0.50	0.02	0.01	1	0.02
	MA	0.01	0.98	0.01	0.00	0.50	0.00	0.00	1	0.01
CNN-2D	CM	0.38	0.83	0.53	0.04	0.50	0.08	0.04	1	0.08
	CC	0.08	0.60	0.15	0.01	0.50	0.02	0.01	1	0.02
	FE	0.04	0.78	0.07	0.01	0.50	0.02	0.01	1	0.02
	MA	0.0	0.94	0.01	0.00	0.50	0.00	0.00	1	0.00
RNN	CM	0.72	0.55	0.63	0.05	0.50	0.09	0.05	1	0.09
	CC	0.04	0.65	0.08	0.01	0.50	0.02	0.01	1	0.02
	FE	0.01	0.87	0.03	0.01	0.50	0.02	0.01	1	0.02
	MA	0.0	0.06	0.01	0.00	0.50	0.00	0.00	1	0.01
AE	CM	0.61	0.67	0.64	0.05	0.50	0.09	0.05	1	0.09
	CC	0.21	0.20	0.21	0.01	0.50	0.02	0.01	1	0.02
	FE	0.18	0.24	0.21	0.01	0.50	0.02	0.01	1	0.02
	MA	0.03	0.12	0.04	0.00	0.50	0.00	0.00	1	0.01

TABLE XVI: Average training-time taken by the models to train a single epoch in seconds

	CNN-1D	CNN-2D	RNN	AE
CM	0.9	1.2	1,155.5	3.8
CC	1.0	1.4	1,575.9	3.3
FE	1.1	1.7	2,284.6	3.5
MA	1.3	1.5	4,997.7	2.6

presence or absence of smells by customizing the training of the *code2vec* model. During the training, we replaced the method names (as in the original implementation) with either true or false based on whether a smell is present or not. The trained model then predicted true or false indicating the presence or absence. The implementation we used can be found online.[7]

We preprocess and train the *code2vec* model using the same set of samples in the same number that we used for training all models, for the two implementation smells, *i.e.*, *complex method* and *complex conditional*. Given that the *code2vec* model is designed to work at the method level, we did not use it for design smells that require class-level treatment. We run the model with the default parameters as proposed in the original implementation. The model performed mediocre with F1=0.22 (precision=0.16 and recall=0.35) for *complex method* and F1=0.06 (precision=0.03 and recall=0.26) for *complex conditional* smell. This performance is significantly lower than the performance shown by other models using simple source code tokenization. This confirms our speculation on the suitability of the afore-mentioned models for smell detection and agrees with the finding that a state-of-the-art model for semantic representation of code is not necessarily appropriate for downstream tasks.

[7]https://github.com/tushartushar/code2vec

D. Opportunities

This study encourages the research community to explore deep learning as a viable option for addressing the problem of smell detection. We showed that the solution is applicable in two programming languages, namely C# and Java. This result encourages further experimentation with additional programming languages of different paradigms.

We used the detection mechanisms of Designite for obtaining the ground truth to train our models. Relying on a specific tool does not alleviate the fact that smells are indeed detectable using deep learning methods—it rather provides grounds for generalization. A next step towards extending this work could be to investigate variations of smell definitions and diverse tool adaptions by accordingly fine-tuning training. To this end, we release the full pipeline of our deep learning toolkit and invite research in this direction. We are positive that this work will prove robust to extensions, given also the results that we obtained in the transfer-learning experiment.

We have shown that transfer-learning is feasible in the context of code smells. This result additionally introduces new, data-driven directions for automating smell detection which is particularly useful for programming languages for which smell detection tools are either not available or not matured.

Given that we did not consider the context and developers' opinion on smells reported by deterministic tools, it would be interesting to combine these aspects either by considering the developers' opinion (by manually tagging the samples) while segregating positive and negative samples or by designing models that take such opinions as an input to the model.

This work shows the feasibility of detecting implementation smells; however, complex smells such as *multifaceted abstraction* and *feature envy* require further exploration and present many open research challenges. Design and architecture smells typically span across multiple source files

and abstractions. Furthermore, their detection involves identifying complex semantic features that makes design and architecture smell detection using machine learning methods difficult. The research community may build on the results presented in this study and further explore optimizations to the presented models, alternative models, or innovative model architectures to address the detection of complex design and architecture smells.

Smell samples used for training and evaluation are highly imbalanced naturally. We observed that in the best case it could be 15 negative samples per positive sample while it may go up to as skewed as 588 negative samples per positive sample (refer to Table XIV). Compared to other deep-learning models, Autoencoders fit naturally in this context, because they are more robust to class imbalance. We anticipate that future research work would explore the potential of autoencoders in more detail.

Beyond smell detection, proposing an appropriate refactoring to remove a smell is a non-trivial challenge. There have been some attempts [151], [152] to separate refactoring changes from bug fixes and feature additions. One may exploit the information produced from such tools and the power of deep learning methods to construct tools that propose suitable refactoring mechanism.

VI. Threats to Validity

Threats to the validity of our work mainly stem from possible faults in the employed tools, our assumption concerning similarity of both the programming languages, and generalizability and repeatability of the presented results.

A. Construct Validity

Construct validity measures the degree to which tools and metrics actually measure the properties that they are supposed to measure. It concerns the appropriateness of observations and inferences made on the basis of measurements taken during the study.

In the context of using deep learning techniques for smell detection, we use Designite and DesigniteJava to detect smells in C# and Java code respectively and use these results as the ground truth. Relying on the outcome of two different tools may pose a threat to validity especially in the case of transfer-learning. To mitigate the risk, we make sure that both the tools use exactly the same set of metrics, thresholds, and heuristics to detect smells. Also, we ensure the smell detection similarity by employing automated as well as manual testing.

To address potential threats posed by representational discrepancies between the two languages we ensure that Tokenizer generates same tokens for same or similar language constructs. For instance, all the common reserved words are mapped to the same integer token for both the programming languages.

B. Internal Validity

Internal validity refers to the validity of the research findings. It is primarily concerned with controlling the extraneous variables and external influences that may impact the outcome.

In the context of our investigation, exploring the feasibility of applying transfer-learning for smell detection, we assume that both programming languages are similar by paradigm, structure, and language constructs. It would be interesting to observe how two completely different programming languages (for example, Java and Python) can be combined in a transfer-learning experiment.

C. External Validity

External validity concerns generalizability and repeatability of the produced results. The method presented in the study is programming language agnostic and thus can be repeated for any other programming language given the availability of appropriate tool-chain. To encourage the replication and building over this work, we have made all the tools, scripts, and data available online [138].

VII. Conclusions

The interest in machine learning-based techniques for processing source code has gained momentum in the recent years. Despite existing attempts, the community has identified the immaturity of the discipline for source code processing, especially when it comes to identifying quality aspects such as code smells. In this paper, we establish that deep learning methods can be used for smell detection. Specifically, we found that CNN, RNN, and autoencoder deep learning models can be used for code smell detection, though with varying performance. We did not find a clearly superior method between 1D and 2D convolution neural networks. Further, our results indicate that RNN performance is not consistently better than convolutional networks. Our experiment on applying transfer-learning proves the feasibility of practicing transfer-learning in the context of smell detection.

With the results presented in the paper we encourage software engineering researchers to build over our work as we identify ample opportunities for automating smell detection based on deep learning models. There are grounds for extending this work to a wider scope by including smells belonging to design and architecture granularities. Furthermore, there exist opportunities for further exploiting results and coupling with deep learning methods for identifying suitable refactoring candidates. From the practical side, the tool developers may induct the deep learning methods for effective smell detection and use transfer-learning to detect smells for programming languages where no appropriate code smell detection tools are available.

Acknowledgement

This work is partially funded by the SENECA project, which is part of the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID) under grant agreement number 642954 and by the CROSSMINER project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 732223.

We would like to thank Antonis Gkortzis, Theodore Stassinopoulos, and Alexandra Chaniotakis for generously contributing effort to our DesigniteJava project.

This work was supported by computational time granted from the National Infrastructures for Research and Technology S.A. (GRNET S.A.) in the National HPC facility — ARIS — under project ID pa180903-smellsDL.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Programs*, 1st ed. Addison-Wesley Professional, 1999.

[2] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217303114

[3] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[4] R. Marinescu, "Measurement and quality in object-oriented design," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Universitatea Politehnica din Timisoara, Timisoara, Romania. IEEE, Dec. 2005, pp. 701–704.

[5] M. Salehie, S. Li, and L. Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws," in *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, University of Waterloo. IEEE Computer Society, Jun. 2006, pp. 159–168.

[6] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[7] T. Sharma, P. Mishra, and R. Tiwari, "Designite — A Software Design Quality Assessment Tool," in *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities*, ser. BRIDGE '16. ACM, 2016.

[8] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[9] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Polytechnic School of Montreal. ACM, Sep. 2012, pp. 278–281.

[10] G. Czibula, Z. Marian, and I. G. Czibula, "Detecting software design defects using relational association rule mining," *Knowledge and Information Systems*, vol. 42, no. 3, pp. 545–577, Mar. 2015.

[11] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," in *QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software*. IEEE Computer Society, Aug. 2009, pp. 305–314.

[12] ——, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," in *Journal of Systems and Software*. Ecole Polytechnique de Montreal, Montreal, Canada, 2011, pp. 559–572.

[13] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, and E. Aïmeur, "SMURF: A SVM-based incremental anti-pattern detection approach," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, Ptidej Team. IEEE, Dec. 2012, pp. 466–475.

[14] S. Bryton, F. Brito E Abreu, and M. Monteiro, "Reducing subjectivity in code smells detection: Experimenting with the Long Method," in *Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*, Faculdade de Ciencias e Tecnologia, New University of Lisbon, Caparica, Portugal. IEEE, Dec. 2010, pp. 337–342.

[15] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[16] D. D. Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. D. Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 00, March 2018, pp. 612–621. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SANER.2018.8330266

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[19] T. N. Sainath, B. Kingsbury, G. Saon, H. Soltau, A.-r. Mohamed, G. Dahl, and B. Ramabhadran, "Deep convolutional neural networks for large-scale speech tasks," *Neural Networks*, vol. 64, pp. 39–48, 2015.

[20] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2015, pp. 103–112.

[21] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[22] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying Architectural Bad Smells," in *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Mar. 2009, pp. 255–258.

[23] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[24] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural bad smells in software product lines: An exploratory study," in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14 Companion. ACM, 2014, pp. 12:1–12:6.

[25] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, ser. QoSA '09. Springer-Verlag, 2009, pp. 146–162.

[26] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2014.

[27] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A New Family of Software Anti-patterns: Linguistic Anti-patterns," in *CSMR '13: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Mar. 2013, pp. 187–196.

[28] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems & Software*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011.

[29] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting

code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[30] S. Fu and B. Shen, "Code Bad Smell Detection through Evolutionary Data Mining," in *International Symposium on Empirical Software Engineering and Measurement*, Shanghai Jiaotong University, Shanghai, China. IEEE, Nov. 2015, pp. 41–49.

[31] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-Smell Detection as a Bilevel Problem," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, pp. 6–44, Oct. 2014.

[32] A. Ouni, R. G. Kula, M. Kessentini, and K. Inoue, "Web Service Antipatterns Detection Using Genetic Programming," in *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Osaka University. ACM, Jul. 2015, pp. 1351–1358.

[33] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[34] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2018.

[35] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, pp. 2195–1721, 2017.

[36] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[37] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[39] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[41] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[42] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *AT&T Labs. [Online] http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[43] J. Martens, "Deep learning via hessian-free optimization." in *ICML*, vol. 27, 2010, pp. 735–742.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[45] D. J. Felleman and D. C. Van Essen, "Distributed hierarchical processing in the primate cerebral cortex," *Cerebral Cortex*, vol. 1, no. 1, pp. 1–47, 1991.

[46] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[47] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[48] O. Z. Kraus, J. L. Ba, and B. J. Frey, "Classifying and segmenting microscopy images with deep multiple instance learning," *Bioinformatics*, vol. 32, no. 12, pp. i52–i59, 2016.

[49] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[50] O. M. Parkhi, A. Vedaldi, A. Zisserman *et al.*, "Deep face recognition." in *BMVC*, vol. 1, no. 3, 2015, p. 6.

[51] S.-M. Lee, S. M. Yoon, and H. Cho, "Human activity recognition from accelerometer data using convolutional neural network," in *Big Data and Smart Computing (BigComp), 2017 IEEE International Conference on*. IEEE, 2017, pp. 131–134.

[52] O. Abdeljaber, O. Avci, S. Kiranyaz, M. Gabbouj, and D. J. Inman, "Real-time vibration-based structural damage detection using one-dimensional convolutional neural networks," *Journal of Sound and Vibration*, vol. 388, pp. 154–170, 2017.

[53] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[54] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE, 2013, pp. 273–278.

[55] T.-H. Wen, M. Gasic, N. Mrkšić, P.-H. Su, D. Vandyke, and S. Young, "Semantically conditioned lstm-based natural language generation for spoken dialogue systems," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1711–1721.

[56] Y. Wang, M. Huang, L. Zhao *et al.*, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 606–615.

[57] C. Baziotis, N. Pelekis, and C. Doulkeridis, "Datastories at semeval-2017 task 4: Deep lstm with attention for message-level and topic-based sentiment analysis," in *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, 2017, pp. 747–754.

[58] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.

[59] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[60] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[61] G. E. Hinton and R. S. Zemel, "Autoencoders, minimum description length and helmholtz free energy," in *Advances in neural information processing systems*, 1994, pp. 3–10.

[62] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1096–1103.

[63] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," in *International conference on artificial neural networks*. Springer, 2011, pp. 52–59.

[64] R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, and C. D. Manning, "Semi-supervised recursive autoencoders for predicting sentiment distributions," in *Proceedings of the conference on empirical methods in natural language processing*. Association for Computational Linguistics, 2011, pp. 151–161.

[65] S. C. AP, S. Lauly, H. Larochelle, M. Khapra, B. Ravindran, V. C. Raykar, and A. Saha, "An autoencoder approach to learning bilingual word representations," in *Advances in neural information processing systems*, 2014, pp. 1853–1861.

[66] M. Chen, Z. Xu, K. Weinberger, and F. Sha, "Marginalized denoising autoencoders for domain adaptation," *arXiv preprint arXiv:1206.4683*, 2012.

[67] C. Zhou and R. C. Paffenroth, "Anomaly detection with robust deep autoencoders," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 665–674.

[68] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, 2014, pp. 4–11.

[69] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "A machine-learning based ensemble method for anti-patterns detection," 2019.

[70] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*,

vol. 21, no. 3, pp. 1143–1191, Jun 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4

[71] D. K. Kim, "Finding bad code smells with neural network models." *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 7, no. 6, 2017.

[72] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering*, 2019.

[73] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning." in *ENASE*, 2018, pp. 137–146.

[74] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[75] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[76] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[77] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[78] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *IJCAI*, 2017, pp. 3034–3040.

[79] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 683–693.

[80] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 651–654.

[81] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.

[82] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 2016, pp. 39–40.

[83] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning." in *AAAI*, 2017, pp. 1345–1351.

[84] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 222–232.

[85] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 574–584.

[86] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 599–609.

[87] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2017, pp. 440–450.

[88] C. V. Alexandru, S. Panichella, and H. C. Gall, "Replicating parser behavior using neural machine translation," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 316–319.

[89] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[90] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[91] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[92] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[93] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[94] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 542–553.

[95] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 318–328.

[96] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016, pp. 2091–2100.

[97] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, 2016, pp. 1606–1612.

[98] J. Ott, A. Atchison, P. Harnack, N. Best, H. Anderson, C. Firmani, and E. Linstead, "Learning lexical features of programming languages from imagery using convolutional neural networks," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 336–339. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196359

[99] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3324916

[100] L. Rantala and M. Mäntylä, "Predicting technical debt from commit contents: reproduction and extension with automated feature selection," *Software Quality Journal*, vol. 28, pp. 1551—-1579, 2020. [Online]. Available: https://doi.org/10.1007/s11219-020-09520-3

[101] F. Zampetti, A. Serebrenik, and M. Di Penta, "Automatically learning patterns for self-admitted technical debt removal," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 355–366.

[102] U. Azadi, F. A. Fontana, and M. Zanoni, "Poster: machine learning based code smell detection through wekanose," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2018, pp. 288–289.

[103] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[104] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[105] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 433–437.

[106] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for Smell Detection," in *2016 IEEE 24th International Conference on Program*

Comprehension (ICPC), Universita di Salerno, Salerno, Italy. IEEE, 2016, pp. 1–10.

[107] F. Chollet, *Deep learning with python*. Manning Publications Co., 2017.

[108] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

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

[110] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.

[111] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1073–1085. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[112] H. Babii, A. Janes, and R. Robbes, "Modeling vocabulary for big code machine learning," *arXiv preprint arXiv:1904.01873*, 2019.

[113] V. Markovtsev, W. Long, E. Bulychev, R. Keramitas, K. Slavnov, and G. Markowski, "Splitting source code identifiers using bidirectional lstm recurrent neural network," *arXiv preprint arXiv:1805.11651*, 2018.

[114] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 37–48.

[115] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 49–60.

[116] R.-M. Karampatsis and C. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *arXiv preprint arXiv:1903.05734*, 2019.

[117] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 334–345.

[118] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing." in *AAAI*, vol. 2, no. 3, 2016, p. 4.

[119] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," in *International Conference on Machine Learning*, 2015, pp. 1093–1102.

[120] G. Robles, "Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 171–180.

[121] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[122] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 433–436.

[123] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[124] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[125] V. Efstathiou and D. Spinellis, "Semantic source code models using identifier embeddings," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 29–33.

[126] E. Grave, P. Bojanowski, P. Gupta, A. Joulin, and T. Mikolov, "Learning word vectors for 157 languages," in *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[127] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1412–1421.

[128] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[129] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9512-6

[130] D. Spinellis, Z. Kotti, and A. Mockus, "A dataset for github repository deduplication," in *17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020.

[131] T. Sharma, "Designite - A Software Design Quality Assessment Tool," May 2016, http://www.designite-tools.com. [Online]. Available: https://doi.org/10.5281/zenodo.2566832

[132] ——, "Designitejava," Dec. 2018, https://github.com/tushartushar/DesigniteJava. [Online]. Available: https://doi.org/10.5281/zenodo.2566861

[133] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[134] T. Sharma, "Codesplitjava," Feb. 2019, https://github.com/tushartushar/CodeSplitJava. [Online]. Available: https://doi.org/10.5281/zenodo.2566865

[135] ——, "Codesplit for c#," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2566905

[136] D. Spinellis, "dspinellis/tokenizer: Version 1.1," Feb. 2019, https://github.com/dspinellis/tokenizer. [Online]. Available: https://doi.org/10.5281/zenodo.2558420

[137] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection," *Journal of Systems and Software*, vol. 169, p. 110693, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121220301448

[138] T. Sharma, "tushartushar/deeplearningsmells: public release," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4571626

[139] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*. JMLR. org, 2015, pp. 448–456.

[140] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[141] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[142] Y. Gal and Z. Ghahramani, "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks," *arXiv e-prints*, p. arXiv:1512.05287, Dec 2015.

[143] A. Ng *et al.*, "Sparse autoencoder," *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[144] M. Y. Park and T. Hastie, "L1-regularization path algorithm for generalized linear models," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 69, no. 4, pp. 659–677, 2007.

[145] N. Japkowicz, C. Myers, and M. Gluck, "A novelty detection approach to classification," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI*

95)—Volume 1. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 518—523.

[146] S. Hawkins, H. He, G. Williams, and R. Baxter, "Outlier detection using replicator neural networks," in *Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery (DaWak) 2002*. Berlin Heidelberg: Springer-Verlag, 2002, pp. 170–180.

[147] G. Williams, R. Baxter, H. He, S. Hawkins, and L. Gu, "A comparative study of RNN for outlier detection in data mining," in *Proceedings of the IEEE International Conference on Data Mining*, 2002, pp. 709–712.

[148] L. A. Becker, "Effect size (es)," *Retrieved September*, vol. 9, p. 2007, 2000. [Online]. Available: https://www.uv.es/~friasnav/EffectSizeBecker.pdf

[149] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the Matthews Correlation Coefficient matters," in *Proceedings of the Evaluation and Assessment in Software Engineering*, ser. EASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 120–129. [Online]. Available: https://doi.org/10.1145/3383219.3383232

[150] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1–12.

[151] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[152] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, "Comparison of similarity metrics for refactoring detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. ACM, 2011, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985452