The Scent of Deep Learning Code: An Empirical Study

Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh
{hadhemi.jebnoun,houssem.ben-braiek,masud.rahman,foutse.khomh}@polymtl.ca
Polytechnique Montreal, Canada

ABSTRACT

Deep learning practitioners are often interested in improving their model accuracy rather than the interpretability of their models. As a result, deep learning applications are inherently complex in their structures. They also need to continuously evolve in terms of code changes and model updates. Given these confounding factors, there is a great chance of violating the recommended programming practices by the developers in their deep learning applications. In particular, the code quality might be negatively affected due to their drive for the higher model performance. Unfortunately, the code quality of deep learning applications has rarely been studied to date. In this paper, we conduct an empirical study to investigate the distribution of code smells in deep learning applications. To this end, we perform a comparative analysis between deep learning and traditional open-source applications collected from GitHub. We have several major findings. First, *long lambda expression*, *long ternary conditional expression*, and *complex container comprehension* smells are frequently found in deep learning projects. That is, deep learning code involves more complex or longer expressions than the traditional code does. Second, the number of code smells increases across the releases of deep learning applications. Third, we found that there is a co-existence between code smells and software bugs in the studied deep learning code, which confirms our conjecture on the degraded code quality of deep learning applications.

CCS CONCEPTS

• **Software and its engineering → Software design techniques**.

KEYWORDS

Deep learning, code smells, code quality

ACM Reference Format:
Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. 2020. The Scent of Deep Learning Code: An Empirical Study. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387479

1 INTRODUCTION

Deep Learning (DL) is a type of machine learning that uses artificial neural network with multiple hidden layers. It has been found to be highly effective in several application domains including image recognition, speech recognition, computer games, and natural language understanding. The rise of open source DL frameworks has contributed to the democratization of deep learning technology. Thanks to scripting-based libraries such as Tensorflow and Pytorch, DL practitioners are now capable of developing functional prototypes quickly and experimenting with them. However, such prototypes mostly consist of glue code that patches together the program identifiers, external libraries, data processing functions and training algorithms [22]. That is, due to heterogeneous components and dependencies, the correctness of deep learning code might often be traded with its code quality. Such a choice might turn the deep learning code into complex software applications that are hard to comprehend, debug or even to enhance in the long run.

Sculley et al. [22] examined the long-term maintenance costs of machine learning (ML)-based software systems at Google and reported that ML-based systems encounter all the maintenance issues of traditional software systems. However, they noted that ML-based software systems suffer from an additional set of issues that arise from their statistical and data-driven nature. There have been a few earlier works that examined software bugs in deep learning frameworks [13] and analysed the software engineering practices followed by DL practitioners [1]. They suggest that poor coding practices and quick solutions often result in low-quality code containing various code smells. The presence of code smells within the software systems might incidentally degrade their quality and performance, and thus hinder their maintenance and evolution. While there have been a number of studies on the code quality of traditional software systems and a few on ML-based systems, to date, no investigation has been performed on the code quality of DL-based software systems.

In this paper, we conduct an empirical study on the quality of deep learning code using 118 open-source software systems. We first determine the prevalence and trends of code smells in the DL code, and then contrast them with the code smells from traditional software code. We also show that the amount of code smells increases in the DL applications across releases, and that code smells and software bugs often co-exist within the deep learning code. To the best of our knowledge, this is the first study that investigates the code quality of DL-based software systems. Our study answers three research questions as follows.

RQ_1: Does deep learning code smell like the traditional software code?
We collect a total of 59 deep learning systems and 59 traditional software systems from GitHub. We determine the distribution of 10 code smells in the deep learning systems, and then compare them with the smells from the traditional systems. We found no statistically significant difference between the code smell occurrences in deep learning projects and that in the traditional ones. When types of smells were considered, we found that *Long Lambda Function*,

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387479

Long Ternary Conditional Expression and *Complex Container Comprehension* are more frequent within the deep learning code than within the traditional source code.

RQ_2: What is the global trend of code smells in deep learning projects over multiple releases?
We investigate the trend of code smells found in the deep learning projects over time. We detect the code smells from each of their releases and then analyse the changes in smell instances across the releases. Our analysis suggests that the number of code smells increases across the releases in the deep learning applications.

RQ_3: Is there a co-existence between code smells and software bugs in deep learning applications?
We analyse 37,951 commits from 59 deep learning projects to determine the co-existence between code smells and software bugs within DL code. We first separate the bug-fixing commits using their labels and extract the files that were changed to fix the bugs. Then we determine whether these changed files contain code smells in our deep learning applications. We found that 62.84% of the changed files overlap with the smelly code. Furthermore, frequent smells (e.g., Long Ternary Conditional Expression, Long Parameter List) co-exist with the software bugs more often than the others. We also analysed the time spent for fixing the buggy code. We found that the buggy code from the DL applications needs more time to fix when it is smelly than when it is odorless.

2 BACKGROUND

In this section, we present the phases and challenges of DL-based software systems as well as the code smells analyzed in our study.

2.1 DL-based Software Systems : Phases and Challenges

Over the last few years, Deep Learning (DL) has shown impressive performance in various application domains such as image processing, speech recognition, and natural language understanding [8]. DL models have high learning capacity that allows them to capture increasingly complex patterns directly from data without the need of handcrafted feature engineering [12]. Traditionally, software systems are constructed deductively by writing down the rules that govern the behavior of the system as program code. However, with deep learning, these rules are inferred from training data and they are generated inductively. Here, we present the main steps of DL-based software development.

Data Collection and Preprocessing : DL practitioners first collect a representative dataset that captures the knowledge needed to perform a task. The dataset is pre-processed with numerical encoding that makes it suitable for the numerical optimization-based learning. Then the dataset is divided into three different subsets: *training dataset*, *validation dataset*, and *testing dataset*.

Model Construction: After analysing the dataset, DL practitioners design and configure their DL model by choosing the architecture, setting up the initial hyper-parameters, and selecting the mathematical components such as activation functions, loss functions, and gradient-based optimizers. At this step, they also consider requirements (e.g., target performance metrics), data complexities, and the guidelines from DL research works that addressed similar problems earlier.

Model Fitting: Once the data and model architectures are initialized, the training process starts and gradually updates the model parameters with the goal of minimizing the empirical loss estimated on the training dataset. In layman's terms, the training step systematically evolves the software decision logic towards effectively performing a target task (e.g., recognizing a human face).

Hyper-parameters Tuning: To optimize the learning process, hyper-parameters are often tuned by evaluating the models trained on them. DL practioners use different search strategies (e.g., grid-search, random search) for their hyper-parameter optimization.

Model Evaluation: Once the training step is over, the best-fit model needs to be tested on the testing dataset. The testing dataset remains unseen to the model during both training and tuning phases. The goal is to check whether the trained model performs well against the future-like data or not.

The high learning capacity of the DL models comes with an intensive-engineering process that involves data and configuration management before the training. Thus, although many of the required software components (e.g., ready-to-use routines) are provided by the DL libraries and frameworks, the development of a DL-based system warrants substantial coding and rigorous software engineering practices. Without rigorous engineering, DL-based software systems could quickly turn into the systems that are hard to maintain, debug and enhance due to unpaid technical debts. Sculley et al. [21] highlight two common software design anti-patterns found in the DL-based systems as follows.

Pipeline Jungle: The unstructured data processing pipelines that deal with multiple sources and perform different transformations (e.g., formatting, joins) might become overly convoluted and hard to maintain over time.

Glue Code: The use of DL libraries introduces a large amount of glue code in the deep learning applications. Glue code is often written to construct the arguments and to configure the general-purpose components required for the task at hand. Mature DL systems might contain about **5% DL code** that includes parameterized calls to generic libraries and about **95% glue code** that transforms the entries to be plugged into the ready-to-use routines.

We believe that the emergence of these high-level anti-patterns above is a logical consequence of the poor coding practices in DL-based software projects. DL practitioners might not be totally aware of the spread of these anti-patterns in their DL code, which could be costly in the long run.

In this paper, we examine the code of deep learning projects written in Python and look for occurrences of poor coding practices. More specifically, we investigate the occurrences and impact of 10 Python code smells described in Section 2.2.

2.2 Code Smells

In software engineering, the term code smell was first coined by Kent Beck [11], to describe symptoms in the source code of an application that indicate poor design or implementation choices [24]. These smells do not prevent the program from working. However, they are a violation of the best practices that may increase the risk of software bugs or failures in the future.

Our study is based on an existing Python-based tool named Pysmell [27]. We consider 10 code smells for our study, as were considered by Chen et al. [6]. To determine thresholds for code smells,

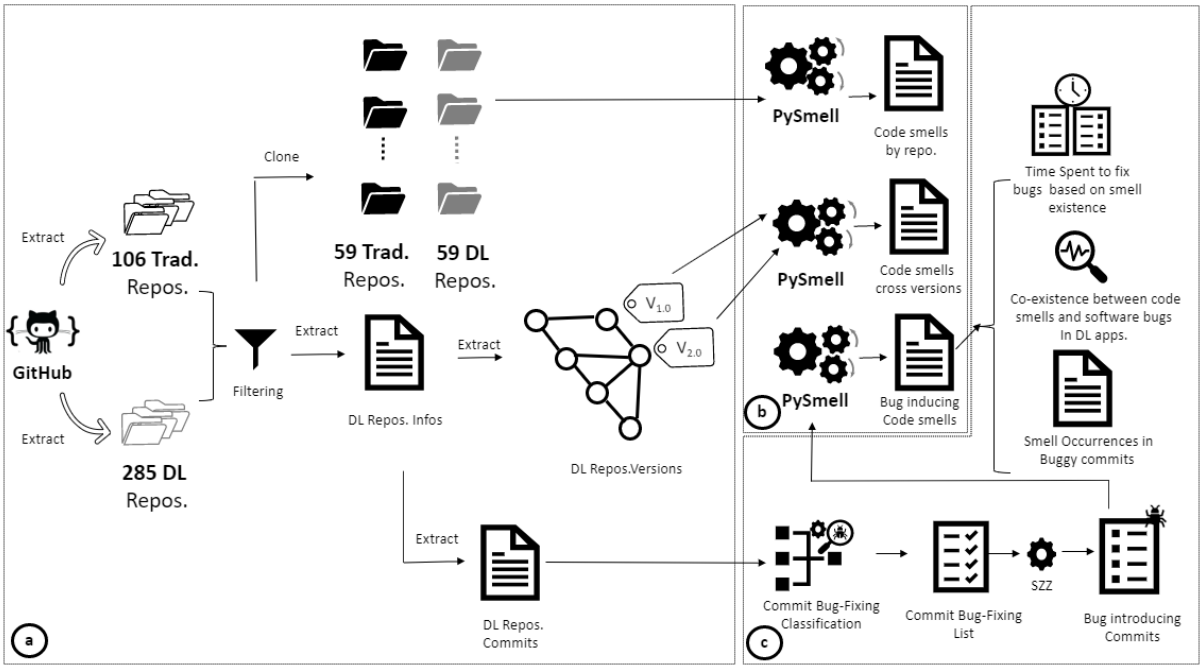

Figure 1: Schematic diagram of the empirical study -(a) Subject system collection and filtration, (b) Code smell detection, and (c) Code smell-bug co-existence analysis

we use an experience-based strategy where the thresholds are defined by 101 experienced developers with 4-10 years of experience. They are also active contributors on popular python projects from GitHub. We select five code smells that are related to object-oriented programming and another five code smells that were defined by Chen et al. [6] from their analysis of bad coding patterns in real world python systems. Below is the list of the 10 selected code smells.

Long Parameter List (LPL) [10]: A method or a function that has a large number of parameters.

Long Method (LM) [10]: A method or a function that is extremely long.

Long Scope Chaining (LSC) [10]: A method or a function that has a deep nested closure.

Large Class (LC) [10]: A class that has a large number of source code lines.

Long Message Chain (LMC) [4]: An expression for accessing an object using the dot operators through a long sequence of attributes or method calls.

Long Base Class List (LBCL) [6]: When a class extends too many base classes due to the multiple inheritances that Python language supports, it makes code hard to understand.

Long Lambda Function (LLF) [6]: An anonymous function that is extremely long and complex in terms of conditions and parameters.

Long Ternary Conditional Expression (LTCE) [6]: A ternary conditional expression that is extremely long.

Complex Container Comprehension (CCC) [6]: One-line comprehension list, set or dictionary that contains a large number of clauses and filter expressions.

Multiply-Nested Container (MNC) [6] a container (including set, list, tuple, dict) that is deeply nested.

3 METHODOLOGY

Fig. 1 shows the schematic diagram of our empirical study. It has three major steps. First, deep learning-based and traditional software systems are carefully selected from GitHub for the study (Fig. 1-(a)). Each of the software systems (a.k.a., repositories) is pre-processed and prepared for code smell detection. Second, we detect code smells using PySmell tool from each of the releases of DL-based and traditional systems (Fig. 1-(b)). Third, we collect bug-fixing commits and their changed source files to determine the co-existence between code smells and software bugs (Fig. 1-(c)). The following subsections discuss these steps in details.

3.1 Subject System Collection & Filtration

System Collection: We attempt to contrast between DL-based and traditional software systems in terms of their code quality (e.g., presence of code smells). Thus, we need to collect both types of systems for our study. In order to collect deep learning systems, we perform keyword search with GitHub Search API [9]. In particular, we choose a set of popular keywords related to various deep learning technology and frameworks as follows.

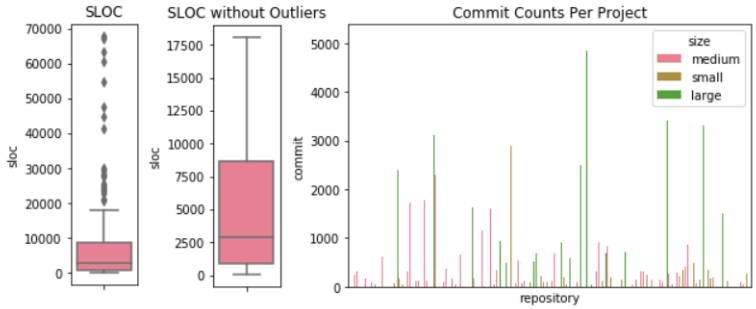

Figure 2: (a) Distribution of SLOC, and (b) Commits in the selected DL-based subject systems

Deep-learning, deep-neural-network, neural-network, CNN, RNN, convolutional neural network, recurrent neural network, Caffe, Keras, Tensorflow, Theano, tflearn, Paddle incubator-mxnet and Torch

We also limit our search to Python-based systems since Python is the most widely used programming language in the DL-based applications to date [2]. The search retrieves a total of 285 DL-based repositories with at least 57 commits each. In order to select the traditional software systems, we reuse the benchmark of Chen et al. [6]. The benchmark provides a total of 106 popular repositories with at least 1000 stars each.

System Filtration: The previous step provides a total of 391 (285 DL-based + 106 traditional) software systems. However, all of them might not be appropriate for our study. We thus manually check each of these systems and discard the inappropriate ones such as tutorials and non-popular projects. Out of 285 repositories, 139 repositories were tutorials, which left us with 146 real deep learning repositories. We also carefully select popular and mature DL projects from them by employing maturity and popularity metrics (e.g., issue count, commit count, contributor count, fork count, stars). We retain only such repositories that have at least four releases each, and discard the rest. Thus, we ended up with a total of 59 DL-based software systems. Out of 106 traditional software systems, we found that 25 systems do not exist any more, which leaves us with 81 traditional systems. However, we randomly choose 59 of them for the study, which ensures a parity in size with our DL-based systems. Thus, we ended up with a total of 118 DL-based and traditional software systems for our empirical study.

System Clustering: While 118 subject systems were selected above, we attempt to better understand them by analysing their SLOC (Source Line of Code) and number of commits. We first calculate the SLOC of each system using Radon [7], a Python-based tool, where only Python files are considered. Then we categorize the subject systems into three clusters – *small, medium* and *large* – using KBinsDiscretizer [20] from Scikit-Learn library [19]. KBinsDiscretizer accepts number of clusters as a parameter and provides balanced clusters using quantile strategy. According to Brown [3], a project is considered *small* if it has SLOC<= $10K$, *medium* if it has SLOC<= $100K$ and *large* if it has SLOC> $10M$. However, since our DL-based systems were not big enough, we adapt these thresholds for our study. In particular, we consider a system small when SLOC<= $4K$, medium when SLOC<= $15K$ and large when SLOC> $15K$. Fig. 2 shows the (a) distribution of SLOC and (b) number of commits/system in 59 DL-based subject systems. We see that DL-based systems are mostly small or medium. The largest DL-based system has a total SLOC of ≈90K. These clusters are later used for answering RQ_1 (Section 4.1).

We also analyze the releases of our deep learning subject systems. We found that the median number of release is 10, i.e., 50% of the DL-based systems have more than 10 releases. For the sake of our analysis, we thus divide the release history of each project into 10 major releases, which involves the merging of actual releases. The systems having less than 10 releases are kept as is. Table 2 shows the number of deep learning systems from three different clusters across 10 releases.

3.2 Code Smell Detection

Our smell detection strategy is based on PySmell [27] as discussed in section 2. To perform our study, we adapted the PySmell code available on GitHub to meet the needs of our project. The code takes as input a folder that contains all the projects under study, and then detects different types of smells from each of the source files. PySmell is a metric-based tool that detects code smells using rules and thresholds. It marks an entity (e.g., class, function) smelly whenever the entity activates any of the predefined rules designed for the smell types. Finally, we capture the code smell occurrences for each smell type against different granularity of program entities – class, function, line. We employ several code level metrics [6] for smell detection as follows: **PAR**: number of parameters; **MLOC**: method/function lines of code; **DOC**: depth of closure; **CLOC**: class lines of code; **LMC**: length of message chain; **NBC**: number of base classes; **NOO**: number of operators and operands; **NOC**: number of characters; **NOL**: number of lines; **NOFF**: number of for clauses and filter expressions; **LEC**: length of element chain; **DNC**: depth of nested container; and **NCT**: number of container types.
Since PySmell's rules are configurable, we use appropriate thresholds to configure these rules that are derived from the work experience of expert Python developers[6]. Such a strategy has been adopted by earlier studies for smell detection[16].

Table 1 presents the strategies and the metrics' thresholds that were used to detect the code smells. All metrics and thresholds used in our empirical study were taken from an earlier work [6].

Table 1: Experience-based thresholds and strategies used by smell type via Chen et al. [6] study

Code Smell	Metric	Thresh.	Strategy
LPL	PAR	5	(PAR, HigherThan)
LM	MLOC	38	(MLOC, HigherThan)
LSC	DOC	3	(DOC, HigherThan)
LC	CLOC	29	(CLOC, HigherThan)
LMC	LMC	5	(LMC, HigherThan)
LBCL	NBC	3	(NBC, HigherThan)
LLF	NOC	48	(NOC, HigherThan)
	PAR	3	and ((PAR, HigherThan)
	NOO	7	or (NOO, HigherThan))
LTCE	NOC	54	(NOC, HigherThan)
	NOL	3	or (NOL, HigherThan)
CCC	NOC	62	((NOC, HigherThan)
	NOFF	3	and (NOO, HigherThan))
	NOO	8	or (NOFF, HigherThan)
MNC	LEC	3	(LEC, HigherThan)
	DNC	3	or ((DNC, HigherThan)
	NCT	2	and (NCT, HigherThan))

As shown in Fig. 1-(b), there are three steps in our smell detection process. First, we clone the repositories of both traditional and DL projects' GitHub repositories. Then, we run the Pysmell tool on them to detect the code smells that occurred in the files of the cloned software projects. This allows us to compare the distribution of Python code smells between traditional and DL projects. In a second step, we restore all the versions of each of the projects collected in the previous step (1-(b)). Then, we run the Pysmell tool on each release version of each project to analyse the trend of code smells over releases. In the third step, we restore the code to its status before applying a bug fixing commit. Then, we execute the Pysmell tool on each file that has been changed by an identified bug fixing commit in order to study the relationship between bugs and code smells in DL-based software projects.

3.3 Experimental Data Analysis

Detecting Bug-Fixing and Bug-Inducing Commits: Since we attempt to determine the potential correlation between code smells and software bugs, we need to detect the bug-fixing and bug-inducing commits from version control history. In order to detect the bug-fixing commits, we employ a keyword search-based approach. In particular, we use a list of keywords for the search – {bug, fix, wrong, error, fail, problem, patch}, as were used by Rosen et al. [18]. If a commit log contains one of these keywords, we consider it as a bug-fixing commit. Once a bug-fixing commit is detected, we use the SZZ algorithm [23] to identify the bug-inducing commits from the version history that introduced the bug.

Co-existence between Code Smells and Bugs: We determine the co-existence between code smells and software bugs by investigating how the bug-inducing code overlaps with the smelly code. First, we identify the files that were changed later to fix the bugs (i.e., bug-inducing files) and that also contain one or more code smells. Second, we calculate the percentage of occurrences for each smell type in these smelly, bug-inducing files. Our goal was to identify the most frequent code smells that co-exist with the bugs in the deep learning applications. Third, we analyze the distribution of the number of bug fixing commits with respect to smelly files and smell-free files.

Time spent to fix a bug that co-exists with code smells: To evaluate the cost of code smells in productivity, we compare the time taken to fix bugs when the files contain code smells and when they do not. We compute bug-fix time by measuring the time interval between the bug-introducing changes and their corresponding fixes [14]. We use Mann-Whitney Wilcoxon test to compare bug-fix time and examine the negative impact of the code smells on bug-fixing and developer productivity.

4 STUDY FINDINGS AND DISCUSSIONS

In this section, we present the results of our study in details, and answer three research questions as follows.

4.1 RQ_1: Does Deep Learning Code smell like the Traditional Software Code?

In this section, we determine the distribution of code smells in both deep learning and traditional software systems, and compare their distributions. We calculate the occurrences of code smells across different dimensions (e.g., project type, smell type) and then compare the DL-based systems with the traditional systems.

Smell Occurrences by Project Type: We compare DL-based systems with traditional systems in terms of their code smell density. First, we calculate the number of code smells in each project, and then divide them with their number of source code lines (a.k.a., SLOC). Fig. 3 shows the comparison between DL-based and traditional systems using box plots where normalized smell occurrences per source code line are considered. We perform non-parametric Mann-Whitney Wilcoxon tests, and found that there is no significant difference (i.e., p-value 0.117 > 0.05) between DL-based code and traditional code in terms of their code smell density.

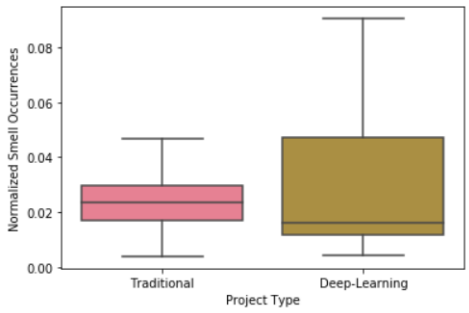

Figure 3: Smell occurrences in DL and traditional projects

Finding 1: There is no statistically significant difference between deep learning code and traditional code in terms of their code smell occurrences.

Smell Occurrences by Project Size: Although the above analysis shows no difference between DL-based and traditional systems in code smell density, we further extend our comparative analysis by considering both type and size of the systems. We calculate

Table 2: Number of small, medium and large DL-based projects across 10 releases

	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
Small Projects	18	18	18	18	18	14	12	11	11	9
Medium Projects	19	19	19	19	19	17	16	15	14	14
Large Projects	22	22	22	22	22	18	16	14	12	11

the occurrence of code smells in each of the projects from small, medium and large clusters, and compare the smells/project between DL-based and traditional projects. Fig. 4 shows our comparative analysis across three sizes (or clusters). We see that the number of code smells per project increases when the size of the project increases for both DL-based and traditional projects. Such a trend is consistent with the earlier findings with traditional systems [15]. However, we found two noticeable differences in the density of code smells between deep learning and traditional software projects when we consider the size. First, the number of code smells in the *small* DL projects is significantly higher than that of *small* traditional projects. Second, the number of code smells in the *large* DL projects is significantly lower than that of the large traditional systems. We perform Mann-Whitney Wilcoxon tests in both cases, and found p-values 0.0002 and 1.19e-05 respectively that are less than 0.05 and thus indicate the statistical significance.

Finding 2: The number of code smells per project increases with the increase in project size for both deep learning and traditional software projects. However, small DL-based projects contain more code smells than that of small traditional projects. On the contrary, large DL-projects contain comparatively less smells than that of large traditional software projects.

Complementary Manual Analysis: We manually analyse the source code of 30 deep learning projects and 30 traditional projects, and derive several meaningful insights. First, the core deep learning code that involves data pre-processing and model training tends to be smelly. Once the code is built, DL systems tend to get larger and mature by including more features for various application domains. In other words, the large DL projects tend to have a mix of core deep learning code and traditional source code. Such a mixture of heterogeneous code is often encapsulated and its functionality is exposed through a simple endpoint. Such a workaround might explain the comparatively less number of code smells in the large DL-based software systems (Fig. 4).

Smell Occurrences by Smell Type: While our above analyses show interesting findings, we further contrast the code smell distributions between DL-based and traditional subject systems in terms of code smell types. In particular, we perform statistical significance tests on these distributions across 10 different code smell types, and then collect the p-values from Mann-Whitney Wilcoxon tests. Table 3 shows the p-values from our tests across project size and code smell type. Given the comparison and p-values from the Table 3, we also divide the code smells into three groups as follows – **Group-I:** smells that occur both in DL-based and in traditional software systems with no statistically significant difference, **Group-II:** smells

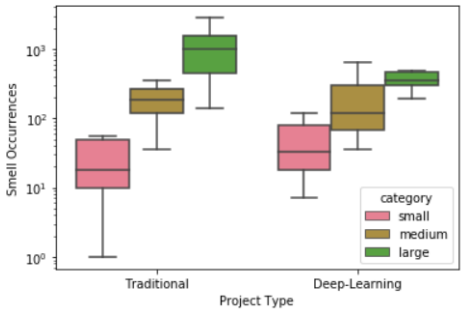

Figure 4: Smell occurrences by project type and project size (small, medium and large)

that occur more in the DL-based systems than in the traditional systems, and **Group-III:** smells that occur more in the traditional systems than in the DL-based systems. Then we further investigate the prevalence and distribution of code smells across these groups.

Group-I: Large Class, Long Method and Long Scope Chaining: Based on a significance threshold of 0.05, we found three python code smells – Large Class (LC), Long Method (LM) and Long Scope Chaining (LSC) – from Table 3 for which DL systems and traditional systems have no statistically significant difference in their smell densities. From Fig. 5-(a), we see that these code smells have similar variances across both project types. These smells are often a result of poor coding practices by the developers. Thus, they might be invariant of the type of the subject systems.

Group-II: Complex Container Comprehension, Long Ternary Conditional Expression, and Long Lambda Function: From Table 3, we found three other code smells that occur more frequently within the DL code than in the traditional code. From Fig. 5-(b), we also see that their code smell distributions are significantly different in terms of median and variance. We thus manually analyse these three code smells for further insights as follows.

Complex Container Comprehension (CCC): Container comprehension is a quick solution for constructing one-line Python objects (e.g., list, set, dict). One-line statements often become complex and hard to comprehend when more and more clauses and filter expressions are added. DL developers often choose a one-line statement to build a sequence of arguments, which results in a long-expression. They use the hard-coded sequences to initialize a list of possible hyper-parameters since these values are often found in the DL white papers and books. Besides, comprehensions are constructs that allow the containers to be built from other containers. Thus, the developer often use CC to reformat the data structure of a set of values, which makes it suitable as arguments for configuring other

Table 3: Mann-Whitney test and Cliff's Delta results between DL and Traditional Projects for each Smell Type in Total without splitting projects by size and for each size of projects.

	LC	LM	LSC	MNC	LTCE	LPL	LMC	LLF	LBCL	CCC
p-value (Total)	0.06	0.43	0.36	**5.44e-05**	**2.94e-12**	**1.22e-20**	**0.002**	**0.04**	**8.42e-08**	**0.02**
p-value (Small)	0.26	0.05	**0.02**	0.33	**1.73e-05**	0.3	**0.0005**	**0.005**	0.425	**0.035**
p-value (Medium)	0.403	0.18	0.22	**0.004**	**1.22e-06**	**0.01**	**6.4e-08**	0.37	**0.006**	**0.006**
p-value (Large)	**0.001**	0.06	0.21	**7.75e-06**	**5.40e-08**	**0.0001**	**9.29e-09**	**0.026**	**4.01e-07**	**0.02**
Cliff's Delta (Total)	0.17	0.02	0.04	0.41	0.73	0.31	0.82	0.18	0.45	0.23
Cliff's Delta (Small)	0.08	0.32	0.27	0.07	0.76	0.1	0.47	0.43	0.02	0.34
Cliff's Delta (Medium)	0.048	0.17	0.13	0.49	0.88	0.44	0.94	0.06	0.38	0.47
Cliff's Delta (Large)	0.54	0.28	0.14	0.8	0.98	0.66	0.99	0.35	0.84	0.36

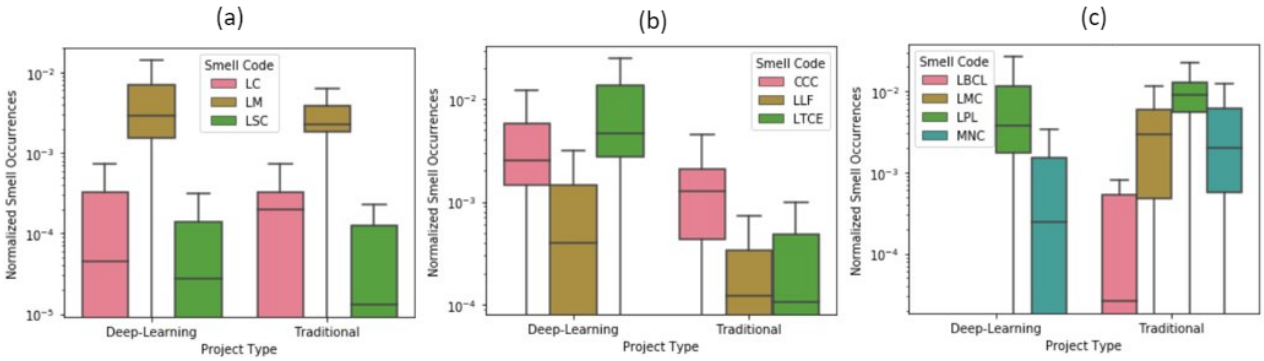

Figure 5: Smell Occurrences by Smell Type and by Project Type

DL routines. Hence, the longer the comprehension, the higher the risk of turning into CCC smell.

Long Ternary Conditional Expression (LTCE): Ternary conditional expression is a conditional variable assignment in one line that makes the code compact. It allows conditional flows into the code and replaces multiple if-else blocks with a single line of code. However, excessive use of this expression could hurt the readability of code. Besides, combination of multiple terms and expressions (e.g., lambda expression) could make this conditional expression unnecessarily complex. DL developers often use ternary conditional operators either to execute a particular routine or to conditionally assign a particular value to a configuration setting. This helps them switch between DNN design and hyper-parameter tuning and determine the impact of their different choices. Our study also reports the high occurrences of LTCE in the DL projects.

Long Lambda Function (LLF): Lambda function is a single-line function. While it is easy to define or use, the function could be hard to manage and maintain when many complex operations are involved. DL developers often choose lambda functions to carry out data processing by creating anonymous function at run-time and then by sending them to appropriate DL routines as parameters. However, the function becomes long and complex when multiple data elements are handled simultaneously by these routines.

Group-III: Long Base Class List, Long Message Chain, Long Parameter List and Multiply Nested Container: Using a significance threshold of 0.05, we found four code smells from Table 3 that occur more frequently in the traditional code than in the DL code. From Fig. 5-(c), we see that their distribution is comparatively lower for the DL-based code. We thus analyse these four types of code smells, and attempt to explain the findings as follows. First, two smells – Long Base Class List (LBCL) and Long Message Chain (LMC) – are mostly related to object-oriented programming (OOP), which might explain their low occurrences in the Python-based deep learning code. Due to the scripting nature of Python-code, DL practitioners might not be interested to use the OOP paradigm of Python language. Second, we also observe low occurrences of Long Parameter List (LPL) and Multiply Nested Container (MNC) smells in the deep learning code. These smells arise from the complexity of the code that involves long parameter list and nested logic. Since DL practitioners implement their applications using a data-driven training process with ready-to-use routines from the libraries, the odds of long parameter list and deep nested logic occuring is lower.

Finding 3: Long lambda expression **(LLF)**, Long ternary conditional expression **(LTCE)** and Complex container comprehension **(CCC)** smells are more frequent within the deep learning code than in the traditional software code.

4.2 RQ_2: What is the global trend of code smells in deep learning projects over multiple releases?

Although we have studied the distribution of code smells in DL applications and compared it with such distribution in the traditional

applications, we also want to further analyse their prevalence over time. We thus investigate the global trend of code smells exhibited by deep learning projects over time. To perform this analysis, we compute the density of code smells per version for each of the DL projects. Then, we categorize each project into our pre-defined size-based categories (i.e., small, medium and large). Then we plot the smell occurrences over versions for each size-based category. Fig. 6 shows the global trends of code smell occurrence in our subject systems.

From Fig. 6, we see that the smell occurrences have an increasing trend. This can be explained intuitively by the increase in source code size (SLOC) and complexity over the life cycle of the DL projects. Our result is also consistent with previous findings on the traditional projects [5]. However, most of the DL projects are new projects that are yet to mature. Thus, the comparison with traditional systems in terms of evolution of code smells could be more effective and more fair when these DL projects would reach similar level of maturity. However, the increasing trend of DL code smells suggest that DL practitioners might not have paid enough attention to the quality of their code even after releasing several versions. Thus, our findings confirm the need for early refactoring of code smells by deep learning practitioners to avoid a costly maintenance in the later releases.

Finding 4: The amount of code smells contained in deep learning applications increases gradually over the subsequent releases of the applications.

4.3 RQ_3: Is there a co-existence between code smells and software bugs in deep learning applications?

We consider that code smells and software bugs co-exist if the bug-fixing code overlaps with the code containing smells. In particular, we detect the files that were changed to fix the bugs and that also contain one or more code smells at the same time. We present our analysis using several dimensions (e.g., overlap ratio, smell type) as follows.

Table 4 shows the likelihood of co-existence with software bugs for different types of code smells. Based on our analysis, we found that, in DL-based software projects, 62.48% of the buggy files (i.e. changed by a bug-fixing commit) contain at least one occurrence of code smell.

Finding 5: About 62.48% of the bug-fixing files overlap with the smelly code. Thus, code smells and software bugs are likely to co-exist in the deep learning applications.

We also present the overlap ratio in more granular level by computing the percentage of the smelly, buggy files per type of smell (Table 4). We provide the percentage of smelly, buggy files containing at least one occurrence of a particular smell. We found that at least 40% of the smelly, buggy files contain LTCE, CCC , LM, or LPL smells. On the other hand, the remaining smells (e.g., MNC, LLF, LSC and LC) could be found in less than 10% of smelly and buggy files.

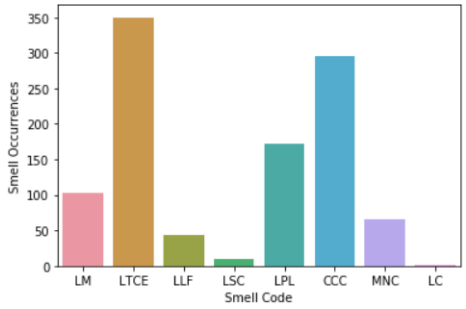

Figure 7: Smell Occurrences in Buggy Commits

Moreover, we present Figure 7 that shows the number of code smell occurrences for each of the eight smell types that partially overlap with the bug-fixing code within the deep learning systems. These substantial differences of total number of instances between smell types reinforce our previous finding and confirm that four code smells (LTCE, LM, LPL, CCC) frequently overlap with the bug-fixing code, whereas the other code smells (LLF, LSC) do not overlap much with the bug-fixing code.

Given our findings and analysis above, Long Ternary Conditional Expression (LTCE), Complex Container Comprehension (CCC), and Long Parameter List (LPL) are the types of code smells that could more likely lead to software bugs or failures. In particular, two Python-based smells –LTCE and CCC – bear more risks than other Python-related code smells given our findings in the Fig. 4. Deep learning developers often rely on advanced functionalities and the grammatical flexibility of Python language for rapid development. Unfortunately, such development practices turn the deep learning code into complex structures (e.g., LTCE), which ultimately can lead to software bugs and failures.

Our results heavily rely on the density distribution of various code smells (e.g., Fig. 4) found in our subject systems. Thus, they also reinforce the fact that the presence of code smells within the systems might increase the chance of software bugs.

Finding 6 : Long Ternary Conditional Expression, Complex Container Comprehension and Long Parameter List are more associated with the software bugs than the other smells in deep learning applications.

We also compare the number of bugs corrections that are performed to either smelly and smell-free source code files. We analyze the distribution of the number of bug fixing commits with respect to smelly files and not-smelly ones (Figure 8). We found that files containing at least one code smell have significantly higher number of bug corrections throughout the project than the files that do not contain any code smell. The Mann-Whitney Wilcoxon test yields a significant p-value of $3.6e-150 < 0.05$ with a medium Cliff's Delta effect size (i.e., 0.28).

Finding 7: Smelly files tend to have more related bug fixing commits than non-smelly files, which indicates that they might be more prone to faults. It might also be the sign that bugs occurring on smelly files require multiple fixes. In our future work, we plan to investigate this hypothesis in more details.

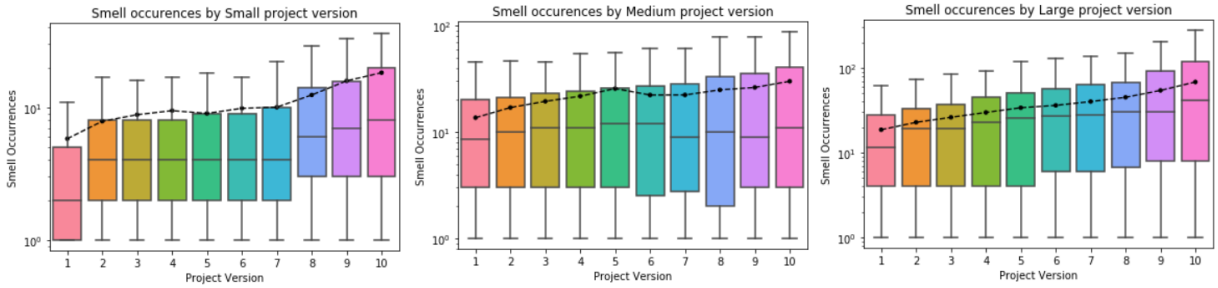

Figure 6: Trend of code smells in DL projects over time

Table 4: The Overlap Ratio Percentage between Buggy Files and Smelly Files by Code Smell Type

Code Smell	LM	LTCE	LLF	LSC	LPL	CCC	MNC	LC	LMC	LBCL
Overlap Ratio %	47.57	48.85	7.546	3.5144	47.5	43.14	6.25	7.27	0.094	0.23

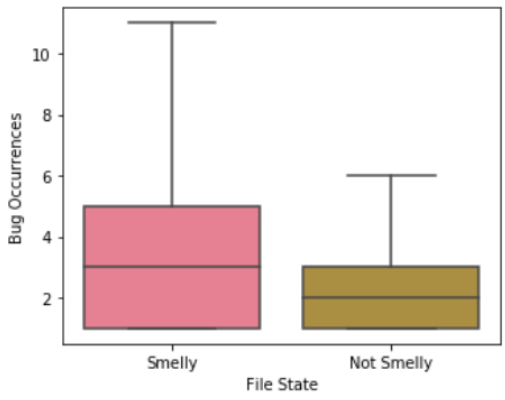

Figure 8: Number of Bugs Correction per Buggy File and per State of File (Smelly or not)

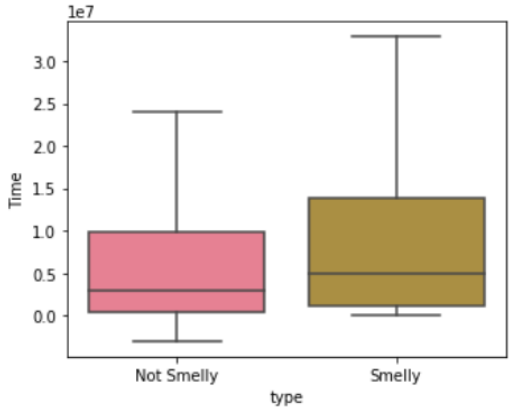

Figure 9: Time to Fix Bugs Distribution by Buggy Commit when it is Smelly and when Not

To gain further insights on the effort needed to fix the bugs in both smelly and smell-free code, we compare the time spent to fix buggy code containing code smells with the time spent to fix buggy code without code smells. We found that fixing the code with code smells takes significantly longer time than fixing the code without any code smells. We perform Mann-Whitney Wilcoxon test and obtained a significant p-value of $2.02e-10 < 0.05$. Furthermore, Fig. 9 shows the distribution of time spent for fixing the buggy code. We see that median bug-fix time for smelly code is significantly higher than that of smell-free code. Based on the above analysis, we find significant evidence that code smells might have a significant impact on bug-fixing time in deep learning based systems.

Finding 8: The presence of code smells has a significant impact on the bug-fixing time of deep learning applications. Time to fix the bugs within smelly code is significantly higher than the bug-fixing time of smell-free deep learning code.

5 RESEARCH IMPLICATIONS

In this section, we discuss several implications of our findings.

Smelly one-line long and complex statements: In the deep learning code, decision logics are learned statistically from the data rather than from complex control flows, nested loops and branches. The DL practitioners might abuse the feature of one-line statements including container comprehensions, ternary conditional operators, and lambda functions with the intention of compacting the code. Our analysis on the prevalence of code smells in the DL projects shows that these one-line statements tend to be longer and involve both complex conditions and sophisticated operations. These two code smells are the most frequent in the DL code. To improve their code quality, DL practitioners should avoid these quick solutions and refactor these one-line statements by decomposing them into manageable separate functions. Besides, the hard-coded long containers like lists or sets are also considered to be CCC code smells since they hinder the code readability and comprehension. Thus, the DL practitioners should also consider the separation of concerns in their application by cutting off the initial configuration options

from the code and adopt maintainable structured files like JSON or XML that can be easily loaded as Python objects.

An increasing trend of code smells over versions: The increasing trend of code smells in DL applications call for the development of techniques and tools that can help DL practitioners refactor their code and improve the code quality.

Co-existence between software bugs and code smells To boost productivity, deep learning developers should become aware of the costs of poor coding practices. It is important to be more conscious about code smells and their peril rather than reaching an non-manageable code state, where the only solution is to write the code again from the scratch.

6 THREATS TO VALIDITY

We now describe the threats to the validity of our study.

Construct Validity threats concern the relation between theory and observation. In our study, threats to the construct validity are mainly due to measurement errors. We use the PySmell tool to detect smells in both types of project (e.g., traditional and DL software project). Relying on the outcome of PySmell tool may pose a threat to validity. To mitigate the risk, we use the experience-based strategy that relies on the thresholds pre-defined by active, experienced python developers. Thus, our thresholds are possibly more appropriate than the thresholds that are defined statistically by analysing the traditional python projects. We are aware that our results can be affected by the presence of false positives and false negatives. However, Chen et al.[6] reported a precision above 82% and a recall of 98% for the experience-based strategy, which are acceptable performance, especially for the recall.

Internal Validity threats concern our selection of projects which could have influenced the results. In our investigation, the searching requests based on GitHub topics and keywords may pose an internal threats to validity. However, we filter the projects using maturity and popularity metrics such as issues count, commits count, contributors count, forks count and stargazers count in order to eliminate all the tutorials and the shared code snippets, and keep only real engineered DL projects.

External Validity threats concern the possibility to generalize our results. In our work, we focused only on python projects, which may reduce the generalizability to all types of DL projects. However, it is important to mention that Python is the most popular and most used programming language [2] in the DL community; so our empirical study on the code quality of DL software projects written in python can spawn useful insights on the software quality of existing DL software systems.

7 RELATED WORK

This section reports about recent research works, around the scope of our work, dealing with the degree of software engineering practices adopted in DL applications and investigating the quality of DL frameworks.

Software Engineering practices in DL applications Amershi et al. [1] propose a survey of Software Engineering (SE) practices for developing Artificial Intelligence (AI) systems. They interviewed Microsoft developers to understand how they work on developing AI applications. They asked the developers about the SE practices that they have used for AI systems and the benefits obtained from using the practices. Wan et al. [25] studied the features and impacts of machine learning towards software development. They compare various aspects of software engineering and work characteristics in both the machine learning systems and non machine learning software systems.

Investigating the quality of DL frameworks Islam et al. [13] studied the characteristics of DL systems' bugs (their types, root causes and effects) through analyzing stack overflow posts and popular framework's issues. Another study related to deep learning quality came from Zhang et al. [26] where they studied deep learning applications built on top of TensorFlow [17] by collecting their program bugs from GitHub and Stack Overflow. They identified the root causes and symptoms of the collected bugs. They also studied the detection and localization challenges of these bugs.

To the best of our knowledge, the present work is the first empirical study on deep learning that investigated the software quality of open source DL final projects (i.e., DL application programs that use DL frameworks and libraries) through mining their GitHub software repositories.

8 CONCLUSIONS

In this paper, we perform a comparison of smells occurrences between traditional and deep learning applications. We analyze a total of 118 repositories (59 deep learning + 59 traditional). We make the following observations:

1) No significant difference: There is no statistically significant difference in the code smell occurrences between deep learning and traditional software systems.

2) Prevalence of code smells in deep learning projects: The most frequent smell types found are *Long Ternary Conditional Expression*, *Complex Container Comprehension*, and *Long Lambda Function*.

3) Violations of the best practices: DL practitioners might not be aware of the code smells in their code, which possibly explains the increasing trend of smell occurrences across the software releases.

4) Code smells lead to bugs: Our findings confirm that the presence of code smells may increase the chances of bugs occurrence.

Ours is the first work that extensively investigates the code quality of 59 open-source, deep learning applications. It can help the researchers better understand the code quality and maintainability of the deep learning applications that are likely to grow in the coming years. Similarly, it can help the practitioners to calibrate their development practices by detecting and refactoring their code smells that are also likely to grow. Our replication package[1] can also be used for replication and third-party reuse.

ACKNOWLEDGMENTS

This work is supported by Fonds de Recherche du Quebec (FRQ) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: a case study. In *Proceedings of the*

[1] https://github.com/Hadhemii/DLCodeSmells

41st International Conference on Software Engineering: Software Engineering in Practice. IEEE Press, 291–300.

[2] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The open-closed principle of modern machine learning frameworks. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 353–363.

[3] Paul C Brown. 2011. *TIBCO Architecture Fundamentals*. Addison-Wesley.

[4] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

[5] Alexander Chatzigeorgiou and Anastasios Manakos. 2010. Investigating the evolution of bad smells in object-oriented code. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 106–115.

[6] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. 2018. Understanding metric-based detectable smells in Python software: A comparative study. *Information and Software Technology* 94 (2018), 14–29.

[7] Python community. 2019. Radon. Retrieved November 10, 2019 from https://pypi.org/project/radon/

[8] Shaveta Dargan, Munish Kumar, Maruthi Rohit Ayyagari, and Gulshan Kumar. 2019. A Survey of Deep Learning and Its Applications: A New Paradigm to Machine Learning. *Archives of Computational Methods in Engineering* (2019), 1–22.

[9] GitHub developers. 2019. GitHub Rest API Search topics. Retrieved November 19, 2019 from https://developer.github.com/v3/search/#search-topics

[10] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[11] Martin Fowler, Kent Beck, and W Roberts Opdyke. 1997. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[13] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. *arXiv preprint arXiv:1906.01388* (2019).

[14] Sunghun Kim and E James Whitehead Jr. 2006. How long did it take to fix bugs?. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 173–174.

[15] A Gunes Koru, Dongsong Zhang, and Hongfang Liu. 2007. Modeling the effect of size on defect proneness for open-source software. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 10–10.

[16] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 7.

[17] Ladislav Rampasek and Anna Goldenberg. 2016. Tensorflow: Biology's gateway to deep learning? *Cell systems* 2, 1 (2016), 12–14.

[18] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 966–969.

[19] scikit-learn developers (BSD License). 2019. scikit-learn Machine Learning in Python. Retrieved December 07, 2019 from https://scikit-learn.org/stable/

[20] scikit-learn developers (BSD License). 2019. sklearn.preprocessing.KBinsDiscretizer. Retrieved December 07, 2019 from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.KBinsDiscretizer.html

[21] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine learning: The high interest credit card of technical debt. (2014).

[22] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.). Curran Associates, Inc., 2503–2511. http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

[23] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.

[24] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 403–414.

[25] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. 2019. How does Machine Learning Change Software Development Practices? *IEEE Transactions on Software Engineering* (2019).

[26] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 129–140.

[27] Chen Zhifei. 2018. Pysmell a tool for detecting code smells in Python code. Retrieved November 01, 2019 from https://github.com/chenzhifei731/Pysmell