Semantics and Assessment papers

Jonathan Simmons, Jan Svacina

6 December, 2019

Important links:

- I. paper Git
- II. paper Git
- YouTrack
- WIKI

1 Worksheets

Jonathan Simmons

Number of tasks:	8
Number of commits:	15
Number of hours:	125
Lines of code:	2300

More to be found: Worksheet - Jonathan

Jan Svacina

Number of tasks: 4	
Number of commits:	15
Number of hours:	130
Lines of code:	2000

More to be found: Worksheet - Jan

Iteration I - II: Paper 1

Word Count Final Draft	4,755
Bibliography:	27 Papers
Researched Papers:	64

Iteration III: Paper 2

Word Count: Draft 1	4,715
Bibliography:	35 Papers
Researched Papers:	85

Semantic Code Clone Detection for Enterprise Applications

Jan Svacina
Computer Science, Baylor University
Waco, Texas
jan_svacina2@baylor.edu

Jonathan Simmons
Computer Science, Baylor University
Waco, Texas
john_simmons2@baylor.edu

Tomas Cerny
Computer Science, Baylor University
Waco, Texas
tomas_cerny@baylor.edu

ABSTRACT

Enterprise systems are widely adopted across industries as methods of solving complex problems. As software complexity increases, the software's codebase becomes harder to manage and maintenance costs raise significantly. One such source of cost-raising complexity and code bloat is that of code clones. These circumstances create an opportunity to implement code clone detection for the specific purpose of lowering enterprise application maintenance costs. Many researchers have proposed code clone detection techniques, however, there is a lack of solutions targeted explicitly toward enterprise applications, and even less dedicated to the semantic code clones that exhibit the same behavior between clone pairs but differ in syntactic structure. Therefore, our proposed approach to solve semantic code clone detection for enterprise frameworks uses control flow graphs (CFGs) to represent an enterprise application and applies various proprietary similarity functions to compare enterprise targeted metadata for each pair of CFGs. This approach enables us to detect semantic code clones with high accuracy within a time complexity of $O(n^2)$ where n is equal to the number of CFGs composed in the enterprise application (usually around hundreds). We demonstrated our solution on a blind study utilizing a production enterprise application.

CCS CONCEPTS

• **Software and its engineering → Reusability**; **Software verification and validation**; *Software maintenance tools*; • **Theory of computation** → Graph algorithms analysis;

KEYWORDS

Source Code Analysis, Code Clone Detection, Semantic Clone, Enterprise Software, Software Engineering

ACM Reference Format:

Jan Svacina, Jonathan Simmons, and Tomas Cerny. 2020. Semantic Code Clone Detection for Enterprise Applications. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341105.3374117

1 INTRODUCTION

Enterprise applications are ubiquitous and essential for the modern world since they address issues far and wide; such as corporate payroll, medical records, mail and package records or tracking, analytical tools such as cost or agricultural, calculating credit scores and many more [18]. These comprehensive problems require complex solutions using enterprise frameworks [1, 10, 12], which in turn require complex code constructs such as web services or service-oriented architectures [10]. Throughout the years, technologies and systems develop constantly, which in turn, develops and increases the complexity, dependency requirements, and legacy code backlog for these technologies and systems. These changes drastically increase maintenance costs [4]. Yearly, corporations pay from 20% to 25% of their total fees and costs as just maintenance costs [9, 17]. For instance, Oracle earned 2.7 times the revenue of their licensing costs through maintenance and support revenue streams, as divulged in their annual financial reporting for the year 2014 [17]. Therefore, transitively speaking, unnecessary complexity of a software system drives up to a quarter of the costs for development, making complexity one of the most important topics to pay attention to for companies desiring to lower costs and raise development efficiency.

Complexity can be reduced by detecting and avoiding code clones, which constitutes from 10% to 23% of large software projects [15, 21]. This bulk of duplicated, redundant code has many causes; one potential cause being when poor code practices cause companies to throw more developers onto projects to manage them, in turn, causing confusion and miscommunication and multiplying the code clones and complexity through both new and preexisting coding malpractices. Adding developers to teams to deal with systems that are already riddled with damaging code clones without removing the damaging code clones themselves can only end up adding complexity and legacy code, in turn drastically increasing the maintenance and support costs the company now has to pay [4]. Developers are spending hundreds of hours of precious, well-paid time just to keep the system afloat.

As shown, when the complexity of a system increases, proportionally too does the cost of supporting that very same system [4]. Developers on legacy-rich systems may have to scour thousands or hundreds of thousands of lines of code to find bugs, sometimes the very same bug appearing in dozens of locations in the codebase due to code cloning [23]. Rereading and learning legacy code is extremely time consuming, and with developers being a commodity for these massive, multi-market, legacy code filled, software corporations, training professionals on old code in order just to be able to fix and maintain the system is expensive and reduces hours that could be invested in new solutions and products. However, if the codebase was less saturated with code duplication, developers would potentially only need to change code in a single or few areas to perform even the very same bug fixes that would take numerous refactorings on a system full of code clone caused coupling.

Code bases can be reduced and cleaned for these purposes with better practices enforced with the help of code clone detection [23]. Specifically, semantic code clones, as it is possible that developers co-developed the same behaviors synchronously rather than merely copy-pasted each other's work. This tangled with the fact that massive enterprise applications often have many methods that essentially do the same work (such as: fetch data, tweak, post to repository) but have drastically different impacts and responsibilities depending on context leads us to be convinced that semantic code clones are of far more interest and impact to an enterprise application than any other type. Therefore, since code clones cause complexity and raise costs for enterprise applications; we pose that semantic code clone detection can provide meaningful results to the fields of development, quality of life, and maintenance with respect to the software engineering of large enterprise applications.

We propose a method in which we represent an enterprise system as a set of control flow graphs (CFG), where nodes are represented by method statements and edges by calls between methods. Each CFG is associated with its semantic meaning in the system. For instance: what responsibilities the method has (database persistence, communication with the user, etc), data attributed to the method (the input and output of the method), and the enterprise-specific implications of the method (security, entry-point, etc). These enhanced CFGs are compared one with another by a global similarity function. Out similarity function runs in $O(n)$ and comparing each CFG pair results in $O(n^2)$ combinations. This method keeps both the time complexity and number of CFGs n low. According to the results from our similarity function, we can categorize the code clones. We conducted a blind study on a production enterprise application and found 5 % of code clones in all categories. This paper will highlight our solution to the topic of semantic code clone detection in enterprise applications to help enterprise software providers reduce maintenance costs and analyze several desired metrics. In Section 2, this paper will outline the state of the art and the other approaches to code clone detection focusing on the methodologies of general code clone detection and then narrowing to the focus of semantic code clones. Then, Section 3 will outline our proposed method of detection followed by Section 4 - a case study on an enterprise application. Lastly, we will conclude by summarizing the results from our experiment and highlighting important notes and concepts that can be gleaned through our research.

2 RELATED WORK

2.1 State of the Art

Code clones are blocks of code that have been copied from some source and pasted elsewhere, not necessarily always from the parent system [23]. There are 4 code clone types or classes: type 1 - exact clones, type 2 - parameterized clones, type 3 - near-miss clones, and type 4 - semantic code clones [7, 16, 22, 23]. Type 1 is self-descriptive, a block of code is a type 1 clone if an exact copy of the source code can be found elsewhere. Type 2 pertains to the same structure of code, however, variables or function calls may have different names. Type 3 clones have the same flow and structure but with changed names, as with type 2, and injected subroutines. The focus for this paper, type 4 clones, or semantic clones, are those which have the same behavior but different structure or method of approach [23].

Much research has been contributed to the field of code clone detection, from types 1 to types 4 alike. However, the research into developing tools focused on enterprise systems is underdeveloped - and as emphasized from the introduction - clearly well needed. One tool name DSCCD (Dynamic and Static Code Clone Detector) [19] is able to detect semantic code clones - the most challenging of the 4 types to detect - at a rate of accuracy at 66%. The tool was developed to weigh the benefits of run time versus the reduction of false positives. The approach does as intended; however, a thorough analysis of a codebase for semantic code clone detection is naturally a computation-time consuming task. The case study done over DSCCD had 12 semantic code clones written into it, and in order to get their 66% overall accuracy rating, it took over 426 seconds in one study [19]. The benefit of this system and their contribution to the field was not necessarily moot, they utilized both dynamic and static analysis via program dependency graphs (PDGs) and abstract memory states (AMS). The PDGs provide a higher level flow analysis for the semantic clone detection, and the AMS provides a quick, lower-level analysis [19]. While using AMS helped lessen the run time and lowered false positive rates, AMS methods cannot handle scopes beyond single methods. This flaw renders them much less useful for enterprise applications, where the flow of method calls is more important for determining duplicate behavior due to separation of concerns making some methods extremely short.

For example, consider the following; let there be some method A that performs an action a. Create some new method B such that B calls A and returns the action a. It is trivial to see that B does the same exact thing as A, however, if the flows of these two methods are not analyzed and compared, they will not be tagged as semantic clones though it is clearly evident that they are. Semantic clone detection should be agnostic of lines of code.

One approach that was able to resolve many of these qualms was done with the implementation of deep learning; research done by White et al. lead to the development of a deep learning algorithm that was capable of analyzing massive codebases with extremely low false-positive rates [26]. White et al. managed to get 93% true positives [26], taking only around 3 seconds using a model trained off of ASTs and 35 seconds using a proprietary greedy algorithm

[26]. White et al. discovered dozens of types 1 through 3 clones in multiple systems and 5 type 4 clones [26]. White et al. had developed software that could analyze systems with exceptionally large codebases (such as Hibernate or Java JDK); the caveat being that running on a new system requires training the model on that system, and the training takes an equally exceptional amount of time. Their example of using Java JDK, with over 500,000 lines of code (not unreasonable for large enterprise applications), took 2,977 seconds to train via the less accurate AST method and 14,965 via their greedy learning method [26]. Considering that enterprise applications are explicitly based on business logic that may change and the constant evolution of such systems, any given enterprise application will have to be constantly re-fed into the model for re-training to provide an accurate model for what code clones may look like.

White et al. are not the first to attempt code clone detection via machine learning approaches, Yu et al. propose a similar method by running two simultaneous neural networks over each pair of code snippets and categorize them neatly into one of many types of clones (such as type 1, type 2, strongly type 3, type 3, or weak type 3 that they interchange as potentially type 4) [27]. The approach provided by Yu et al. is wonderful and boasts strong accuracies surpassing even 96%, however, a similar pitfall of complexity and training time pends the approach lackluster for enterprise solutions. Yu et al. do not divulge much information about the performance of their algorithms - beyond that training takes several hours - so their utility as an enterprise application code clone detection tool is severely limited.

Other attempts at machine learning-based analysis such as that by Sheneamer et al. [24], which uses 15 machine learning algorithms, and [8] are competitive in accuracy and performance at run time, however, are system-specific in that they similarly need to be trained for each code base and do not consider the meta-information provided by enterprise structures and patterns.

The tool CCFinder by Kamiya et al. is an example of code clone detection that has been implemented and is capable of discovering types 1 through 3 code clones efficiently and effectively [14]. Kamiya et al. focus heavily on maintainability and can help users determine if a code clone is safe to remove or reduce with impact to the system [14], however, they acknowledge inter-method flows are challenging to capture and they focus exclusively on source code analysis, thus this tool is not beneficial for large and complex enterprise systems and their dependency on inter-flow communication. Because CCFinder is not able to capture type 4 clones, there must be even some code clones technically of the 3rd type, such as described in the example provided above, that cannot be found due to the lack of flow analysis. So, even tools that are fantastic for types 1 through 3 code clones may not provide even a necessarily useful analysis for enterprise applications.

There are dozens of implementations of code clone detection, many even specializing in semantic types. However, these implementations are strictly theoretical and academic and are hard to implement and use in real-world applications. Other tools such as Agec, by Kamiya et al. [13] and the algorithm by Tekchandani et al. [25] provide code clone detection solutions for type 4 specifically, but also fall short in these same ways.

2.2 Program Representation

We intend to show that to effectively gather semantic code clones of an enterprise system, the optimal choice is to use control flow graphs or (CFGs) to represent the code segments in question. An CFG is a special type of call graph where we let the nodes represent the methods of the system and the edges as calls to other methods within the system. Our preference toward this method of program representation over other methods (token-based [5], abstract syntax tree (AST)[6], abstract memory state (ASM)[3], program dependency graph (PDG)[11], etc.) is to capture more meta-information regarding the context of the code clones or methods with regards to an enterprise architecture.

Some meta-information we may want to collect and display, and potentially analyze further in future research, could be where code clones are being found in the enterprise system based on GRASP [20] patterns and what layer of the application they are found in [18]. This information, that is only accessible in a higher-level abstraction such as an CFG, could allow for our program to eventually be filtered to only analyze service modules or controller modules; which would help developers decide whether their service/controller/etc. classes could be further split or merged to improve cohesion and reduce unnecessary coupling. Not only would performance greatly improved if users could filter code clone detection by concern but also would the code clone detection be more meaningful having knowledge of where in the application they are found - which can be found and used in the CFGs via meta-information used when developing enterprise applications such as annotations in Java code, such as that provided by the common and widely used Spring Framework [2], or similar means in other enterprise frameworks.

As evident through our state of the art, it is evident to see that AST and token-based approaches are highly popular methods of program representation for finding code clones both semantic and syntactic, so our choice to use CFGs is explicit and oriented particularly toward enterprise applications. We pose that the benefit to an CFG, which is a type of control flow graph, over that of an AST is that ASTs provide too low-level a depiction of the program and can consume too much memory to analyze, especially since we are not interested in clone detection that requires such low-level, syntactic knowledge.

Code clone detection is a widely studied field, but it is lacking in its depth with respect to enterprise applications. It is not the case, however, that business domain related code clones have never been researched, but many glances into this field left many questions unanswered and only emphasized the need for such a tool than provided one [16].

3 PROPOSED METHOD

We base our idea of detecting semantic code clones by focusing primarily on the semantic meaning of an CFG, rather than on the structure itself. We used CFGs to represent the enterprise system.

Semantic properties (hereby referred to only as properties) are derived from the metadata that are associated in each method in the form of configuration files or annotations. We examine properties of each graph by applying a global similarity function as shown in the Definition 3.1. Properties of the CFG bring higher value to identifying code clones because programs in enterprise systems tend to be repetitive in their structure but differ in meaning of the data in the input and output of the program [18]. In other words, not every structural repetition of code is a code clone, but a semantic repetition is very likely to be a code clone. Our approach consists from four stages, graph transformation, graph quantification, similarity comparison, and classification as shown in Figure 1.

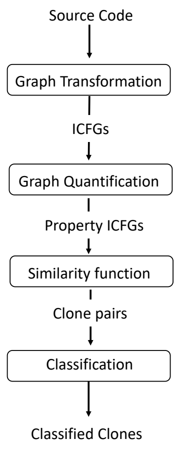


Figure 1: Schema of the algorithm

In the first transformation phase, we transform Java source code into an CFG. We used Java Reflection and Javassist libraries to scan the code for all declared methods, then, for each declared method, we get each method call within its body. We used depth first search to construct a graph for each method that does not have a parent method call. Such method is an entry-point to the enterprise application. For illustration consider an example of a system, where an endpoint method *create* in the *PosController* that calls *savePos* method in the service *PosService*. Next, *PosService* makes two procedure calls, first to some third party API, and the second to *PosRepository*. The schema of this code is depicted in the Listing 1 and the resulting CFG is shown in Figure 2.

```
@Controller
public class PosController{
    @PreAuthorize("hasRole('USER')")
    @RequestMapping(value = "/pos", method =
         RequestMethod.POST)
    public POS create (@RequestBody Pos pos) {
        return service.save(pos);
    }
}


@Service
public class PosService {
    public Pos savePos(Pos pos){
        Props p = restTemplate.postObject("/props");
        pos.setProps(p);
        return repository.save(pos);
    }
}


@Repository
public interface PosRepository {
    Pos save(Pos pos);
}
```

Listing 1: Source code example

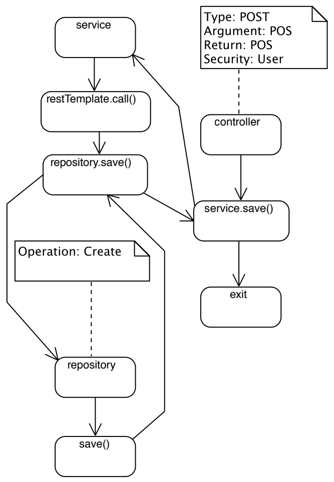


Figure 2: Example of Control flow graph

Next, we quantify each declared method by associating a set of properties P with a method as shown in Figure 2. The set of properties varies by the type of the method as shown in the Table 1. Each method type a has set of associated properties that correspond to its role in the system. All types share argument types, return types, and metadata associated with the methods. The metadata is used to describe the purpose of the method in the system (input handler, database connector, etc.) and associated properties, e.g. HTTP type. Thus, our utilization of CFGs provides additional meaning that determines individual component's roles in the system.

Method Type	Similarity name	Weight	Properties
Controller	ctr	3	arguments, return type, HTTP method, security
Services	fc	1	arguments, return type
Message calls	rfc	2	IP, port, http type, arguments, return entity
Repository	rp	2	Arguments, Return type, Database operation

Table 1: Classification of properties

In the next phase, we apply a global similarity function on properties P of two arbitrary CFGs, as shown in the Definition 3.1. The global similarity function G applies the similarity function for each Method type and multiplies the result by the weight coefficient that corresponds to the importance of a particular method type in the system as shown in the Table 1. Method type *Controller* has the highest significance because it denotes the input, output, and type of operation of a particular entry point. These properties tend to be unique in the system and therefore have a weight of 3. Method type *Repository* persists data to stable storage, and *Message calls* triggers actions usually via HTTP calls on some third party. Both of them have a significant impact on the system and thus have a weight of 2. Method type *Services* have the least importance; service methods are usually reused within the system and thus have weights around 1.

Definition 3.1 (Global similarity).

$$G(A, B) = \sum_{i=1}^{k} w_i \times sim_i(a_i, b_i) / \sum_{i=1}^{k} w_i$$

Where k is the number of attributes, w_i is the weight of importance of an attribute i, $sim_i(a_i, b_i)$ is a local similarity function taking attributes i of cases A and B, and w is a weight coefficient corresponding to the importance of the method type in the system.

We also provide definitions for various other local similarity functions as shown in Definition 3.2. The function takes as input properties of each method and evaluates each attribute. The similarity function for a controller method evaluates its arguments, return type and HTTP method. These similarity functions are all specially targeted for each method type in the system, but all of the methods share argument types and return types object types.

The similarity function ctr targets controller methods, which is a pattern for method that handles input from the user. These methods are usually in REST formats and thus accept HTTP requests; hence the the HTTP method type (GET, POST, PUT, DELETE). Next, it evaluates property security. Endpoints usually have restricted access based on the roles of individual users within an enterprise system. This security measure is referred to as a Role Based Access Control. We include definition of permission roles, for instance *user* or *admin*, into the metadata and thus including in the similarity function.

The similarity function rfc compares all method calls from one system to another system, e.g. HTTP calls from CFG A to B are retrieved and proportioned into a ratio.

When comparing two function calls, similarity function rfc takes into account IP, port, HTTP type, arguments type and return type. The similarity function fc does the same as rfc, but for method calls in the system.

Lastly, the similarity function rp compares methods working with databases by evaluating database operations as shown in Table 1.

Definition 3.2.

$$sim(a_i, b_i) = ctr(a_i, b_i) + fc(a_i, b_i) + rfc(a_i, b_i) + rp(a_i, b_i)$$

The last stage is applying classification of the similarity between CFGs in the system. We classify graphs based on their global similarity into three categories5 2. In the first category are the pairs of CFGs that are similar in all local similarity groups or differ in only one; these correspond to an interval of global similarity within [1.0, 0.91]. Category B has a larger tolerance, thus it encompasses code clones with one or two different local functions. Finally, category C has global similarity within the range [0.8, 0.71] and refers to pairs that differ in 2 to 3 functions.

Classification Type	Global similarity	Characteristics
A	1.0 - 0.91	Same or differs in one function
B	0.9 - 0.81	Differs in 1-2 functions
C	0.8 - 0.71	Differs in 2-3 functions

Table 2: Classification of code clones

4 CASE STUDY

For our case study, we used the enterprise application (EA) for managing and evaluating test questions. The EA was developed at Baylor University as part of an NSF grant proposal for Central Texas Computational Thinking, Coding and Tinkering. The application used microservice architecture and Spring Boot [2] with a set of API methods using standard procedures of multilayered application such as controllers, services, repositories and Role Based Access Control authorization. We analyzed the application in a blind study to detect semantic code clones across the application. In the next part, we will discuss an example of a semantic code clone and the overall results of our study.

We present an example of derived properties from two CFGs, CFG_A andCFG_B derived from the EA as shown in the Table 3. Both of the CFGs have the same input (object *ExamDTO*), output (an

	CFG_A	CFG_B
Controller - *ctr*		
arguments	ExamDTO	ExamDTO
return type	Exam	Exam
HTTP Method	POST	POST
Security	Admin	User
Service methods - *fc*	3	3
Rest methods - *rfc*	2	2
Repository - *rp*		
Database Operation	create	create
arguments	Exam	Exam
return type	Exam	Exam

Table 3: Example of properties of 2 CFGs

object *Exam*), use the same HTTP method *POST*, use a function of that also has the same inputs and outputs, and persist the same object type with the same database operation *create*.

Properties of both graphs CFG_A and CFG_B from the Table 3 were evaluated by local similarity functions as described in the Table 4. Similarity functions fc, rfc, rp give a full match result, whereas the similarity function ctr shows a lower match value due to the different signatures of RBAC security.

	Similarity	Weight	Weighed SIM
ctr	0.75	3	2.25
fc	1	1	1
rfc	1	2	2
rp	1	2	2
total	**3.75**		**7.25**

Table 4: Example of results of similarity function

The Table 4 shows total similarity 3.75 and weight 7.25. Since the maximum possible value of similarity is 8, the graphs CFG_A and CFG_B have a similarity 0.908. This value falls into category A on our scale from Table 2, thus, this is an example of two strongly semantically similar CFGs. We used weighed all of the similarities in order to reflect their importance in the system. For example, controllers are extremely important since they define what data is accepted and produced. Methods working database and services making rest calls also have high significance as these operations are specific to business rules in the enterprise system. Ordinary method calls are not weighed as much since they are repetitive in the system.

Totals	Count
number of pairs	6
number of CFG	20
number of combinations	190

Table 5: Results quantification

We applied our approach in the blind study. Table 5 shows that we derived 20 CFGs from this EA, which comes out to 190 combinations in total. After applying similarity functions on each pair, 6 pairs had a similarity index above 0.71 and thus fell into respective categories as shown in the Table 6, which shows that one pair of CFGs was strongly similar and 5 were fairly similar - which account for 3% of all combinations.

Category	Count
A	1
B	3
C	2

Table 6: Categorizing found clones

The percentage of actual clones shows that our approach is not prone to include false positives but rather reduces all combinations into few pairs that are guaranteed to be relational. This is in part caused by having a high sensitivity or weight based on input and output types. Types of arguments and return values are incredibly important because the same constructs intended for other data types will tend to have the exact same behavior - such as a repository method to save a question to the questions table will semantically behave the same as a repository method to save a test to the tests table, both are necessary and cannot be removed from the application due to semantic similarity alone. This sensitivity with the weights avoids including structurally identical but semantically different CFGs as semantic clones in our results.

4.1 Threats to Validity

Any given code clone detection tool falls under the scrutiny of determining between an output giving false positives or true positives. We set weights for local similarity functions that correspond to real importance in enterprise systems, however, we did not execute the experiment under various settings in an attempt to produce an optimal solution. Semantic type clones require a low threshold in order to detect, therefore we set the classification classes to be within the first third of our scale.

5 CONCLUSION

Our method for semantic code clone detection targets enterprise applications, a massive industry, yet an area of this field that has barely been studied. Because of our method of using CFGs we are able to capture more information about the system with regards to its architecture, which provides us with the means to analyze more criteria and calculate more metrics at a more efficient rate than if we were to use a more storage intensive method like ASTs or tokens.

The task of finding code clones in any code base is not a trivial one, with finding semantic code clones via some form of graph traversal being an NP-Complete problem. Therefore, our ability to produce reasonable results efficiently with the complexity of $O(n^2)$ is impressive when considering that our method of CFG generation produces only tens of operations *n*. Our method of building CFGs is also efficient, needing to scan the codebase only once using a depth first search to check all methods and build out their children list.

Another benefit to our approach is the extensibility. When it comes to enterprise applications, extensibility is a marketable property - as the development of microservice architectures exists thanks to the principle of extensibility. So, thanks to our extensibility at both a micro and macro level, it would not be too much of an investment to transform this tool into a microservice that could be utilized by other microservices. The macro-extensibility is the capability to become a module in some other suite, is not the only type of extensibility present. More micro-extensibility exists since, to expand on this tool, developers need only to add new local similarity functions to capture new metrics or other kinds of enterprise framework properties. So, the inner workings of our tool itself are similarly extensible. A tool like ours could be an essential boon to quality assurance teams for software providing companies world-wide.

For future work, new metrics or other programming language support could be added. For example, there could be a measurement for procedural entropy of the system by running checks on each git commit and calculate the degradation and code clone accumulation over a time period. In the future we could implement the means to measuring greater distances between CFGs using the meta-information and enterprise design patterns to analyze whether a controller class is behaving too much like a service class or etc., the possibilities for introducing new metrics are endless thanks to our method of developing an enterprise application code clone detection tool using enterprise architecture methodologies.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049

REFERENCES

[1] [n. d.]. Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle. https://www.oracle.com/java/technologies/java-ee-glance.html

[2] [n. d.]. Spring Projects. https://spring.io/projects/spring-boot

[3] A. Agapitos, M. O'Neill, and Anthony Brabazon. 2011. Stateful program representations for evolving technical trading rules. 199–200. https://doi.org/10.1145/2001858.2001969

[4] R. Banker, S. Datar, C. Kemerer, and D. Zweig. 1993. Software complexity and maintenance costs.

[5] H. A. Basit and S. Jarzabek. 2007. Efficient Token Based Clone Detection with Flexible Tokenization. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 513–516. https://doi.org/10.1145/1287624.1287698 event-place: Dubrovnik, Croatia.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (Sept. 2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[8] L. Buch and A. Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 95–104. https://doi.org/10.1109/SANER.2019.8668039

[9] Chris Doig. 2015. Calculating the total cost of ownership for enterprise software. https://www.cio.com/article/3005705/calculating-the-total-cost-of-ownership-for-enterprise-software.html

[10] Wu He and Li Da Xu. 2014. Integration of Distributed Enterprise Applications: A Survey. *IEEE Transactions on Industrial Informatics* 10, 1 (Feb. 2014), 35–42. https://doi.org/10.1109/TII.2012.2189221

[11] Y. Higo and S. Kusumoto. 2009. Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *2009 16th Working Conference on Reverse Engineering*. 315–316. https://doi.org/10.1109/WCRE.2009.39

[12] Allen Jin. [n. d.]. DCOM Technical Overview. https://docs.microsoft.com/en-us/previous-versions/cc722925(v%3dtechnet.10)

[13] T. Kamiya. 2013. Agec: An execution-semantic clone detection tool. In *2013 21st International Conference on Program Comprehension (ICPC)*. 227–229. https://doi.org/10.1109/ICPC.2013.6613854

[14] T. Kamiya, S. Kusumoto, and K. Inoue. 2001. A Token based Code Clone Detection Technique and Its Evaluation. (Jan. 2001).

[15] C. Kapser and M. Godfrey. 2003. *Toward a Taxonomy of Clones in Source Code: A Case Study.*

[16] R. Koschke, I. Baxter, M. Conradt, and J. Cordy. 2012. Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071). *Dagstuhl Reports* 2, 2 (2012), 21–57. https://doi.org/10.4230/DagRep.2.2.21

[17] Michael Krigsman. [n. d.]. Danger zone: Enterprise maintenance and support. https://www.zdnet.com/article/danger-zone-enterprise-maintenance-and-support/

[18] M. Foemmel E. Hieatt R. Mee R. Stafford M. Fowler, D. Rice. 2002. *Patterns of Enterprise Application Architecture.* Addison Wesley.

[19] H. Nasirloo and F. Azimzadeh. 2018. Semantic code clone detection using abstract memory states and program dependency graphs. In *2018 4th International Conference on Web Research (ICWR)*. 19–27. https://doi.org/10.1109/ICWR.2018.8387232

[20] D. Rao. [n. d.]. GRASP Design Principles. ([n. d.]).

[21] C. Roy, J. Cordy, and R. Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (May 2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[22] C. Kumar Roy and James R Cordy. [n. d.]. A Survey on Software Clone Detection Research. ([n. d.]), 115.

[23] N. Saini, S. Singh, and Suman. 2018. Code Clones: Detection and Management. *Procedia Computer Science* 132 (Jan. 2018), 718–727. https://doi.org/10.1016/j.procs.2018.05.080

[24] A. Sheneamer and J. Kalita. 2016. Semantic Clone Detection Using Machine Learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 1024–1028. https://doi.org/10.1109/ICMLA.2016.0185

[25] R. Tekchandani, R. Bhatia, and M. Singh. 2018. Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis. *The Journal of Supercomputing* 74, 9 (Sept. 2018), 4199–4226. https://doi.org/10.1007/s11227-016-1832-6

[26] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, Singapore, Singapore, 87–98. https://doi.org/10.1145/2970276.2970326

[27] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang. 2019. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 70–80. https://doi.org/10.1109/ICPC.2019.00021

Assessment of Significant Code Segments in Code Base using Deep Learning

[redacted]
[redacted]
[redacted], [redacted]

[redacted]
[redacted]
[redacted], [redacted]

[redacted]
[redacted]
[redacted], [redacted]

ABSTRACT

Developers, quality assurance managers and testers spends significant time taking over someone else's codebase on daily basis. The most significant portion of this tasks is understanding the raw code. Moreover, codebase is nowdays consisted of variety of languages with addoption of microservices, which introduces boilerplate code and repetition. This creates immense challenges in static analysis, which supports limited number of languages and provides limited support for understanding of the raw code at the current state of the art. We propose a method that is language agnostic and provides developer with assessment of significant code segments in the codebase. We propose method including deep learning techniques involving decision tree regression model for labeling code statements and logistical regression for clustering interesting parts of the code. We applied the method on state of the art open source project.

CCS CONCEPTS

• **Software and its engineering → Reusability**; **Software verification and validation**; *Software maintenance tools*; • **Theory of computation** → Graph algorithms analysis;

KEYWORDS

Keyword, KeyWord

ACM Reference Format:

[redacted], [redacted], and [redacted]. 2020. Assessment of Significant Code Segments in Code Base using Deep Learning. In *Proceedings of ACM SAC Conference (SAC'20).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Developers have many responsibilities in developing the software. The software must complete the task, fulfill quality requirements and easy to read. To accomplish these tasks, developers maintain a mental model of the system. This involves data and behaviour of the system, also specifics of technologies, which greatly vary based on the experience of the developer. Lack of code documentation is notoriously known, because developers are not willing to or lacking time to document the flow of thoughts properly, leaving others to reconstruct the mental model of the author or authors of the code base. Documenting and maintaining the documentation of the software already make up to 60 % of the time spent on the development.

Researchers was aware of this challanges and proposed over last 15 years number of tools and methods to lower the overhead and help developers to better understand the links between documentation and code base. There are number of tools using mostly plugins into IDE that links parts of code base to documentation by embedded links, or even tags with semantic meaning, hyperlinks to navigate quickly in the code base from one place to the other. These tools increase the burden on developers and adds new tasks. The developer have to not only keep the mental model of the system and all underlying technicalities, but have to splitt the focus on creaitg links with the documentaion. As this might be the necessitty to accomplish coherency with the documentation, study from LaToza and col. showed that switiching context between coding and documentation contributes to distractions. Moreover, the connection between code and documentation is not the primary concern of programmer when taking over the code base. As the study showed, the primary concern is understanding the raw code, which constitutes of 42 % of time dedicated to studying someone elses code. There are very few attempts in helping developers to understand and assess the raw code. The primary source of understanding the code is asking a colleague which in turn causes distractions and lower the productivity.

Our general idea is to provide developers with tool that will provide assessment of significant segments of the code that one should focus in order to understand the code base. This method decreases the need for distracting colleagues and lowers the time of browsing the boilerplate code. Such tool is meant for quality assurance managers that needs to have quick, yet comprehensive undrestanding of code that was being used in the core part of the project, it is also meant for developers that needs to get yourself oriented in the code base before they begin to investigate in the detail and thus eliminating in person clarifications.

Our tool utilizes the deep learning techniques for evaluating the code statements and code segments and thus making the whole process language agnostic, which is a desirable property for recent

SAC'20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/xx.xxx/xxx_x

development techniques that include wide range of languages being used together systems using microservice architecture, which become de facto standard in recent years.

We contribute to the following:

- propose language agnostic static code analysis method
- utilize the method for detecting interesting parts of code base
- discuss the importance of the method in the development cycle for developers
- illustrating the method on the state of the art codebase

This paper is organized as follows. Section II summarize the related work. In section III, we describe our language agnostic method for static code analysis and propose its utilization for detecting interesting parts of code base. We evaluate the method in section IV on the open source project. Section V discusses possible threads to validity and what measures we took to mitigate them. Section VI concludes the paper and provides summary of the findings.

2 RELATED WORK

This section will bear witness to the two primary approaches conducted in our experiment: classifying code statements into a list of stereotypes and clustering statements into interesting and uninteresting. Classifying lines of code itself is an interesting problem, with implications in various NP-Complete problems pertaining to source code analysis. Namely, code clone detection. The experiment outlined in this paper come about as a potential solution to the problem of Semantic Code Clone detection being too slow and inefficient to run on large enterprise code-bases in a routine manner - where the benefits of doing such are evident as helping reduce code bloat, making bug fixing easier, and making code more readable and understandable, all of which drastically reduce development costs [4, 12, 16]. The proposal originally was that of using only segments of code determined to be "interesting" as inputs to the semantic code clone detector in order to reduce run time with a potentially minimal impact on results. This lead, however, to another utilization of code classification: code readability and understand-ability in general case.

2.1 Motivation

The ability to understand code is one that obviously impacts the developmental efficiency of programmers in general, especially when hiring new programmers or implementing a legacy codebase. Developers today spend nearly half of their valuable time fixing bugs and roughly 15% to 21% making code more maintainable [12, 16]. An exuberant amount of time for these developers is also spent switching between productive development and reading documentation (if even available) or examining code []. Developers also agree that 55% to 66% of their difficulty in understanding code comes from code previously written by other developers or just understanding the behavior of some source code CITE. Studies have also shown that developers can spend upward of 50% to 80% of their time maintaining code and making it more maintainable for the future [10]. Not only this, but some software engineers reported that 55% to 95% of their workload relates to maintenance tasks such as bug fixing, documentation, modification of preexisting features, and so forth [10]; and that software comprehension caused around 60% of their devoted time [13]. Hence, previously written code that is not well written and understood can heavily diminish developer productivity, and developer productivity is already massively hindered by maintaining code they are currently writing. Therefore, it is evident that maintainability and understand-ability of a codebase is absolutely crucial for productive and successful software engineering.

These delays in development and loss of productivity due to code understand-ability and maintenance issues directly impact the cost of development [16]. Maintenance and bug fixing alone can cost companies up to 25% of their budget, and companies employing contractors can spend up to 20% of their annual budget requesting services and fixes [12, 16]. Clearly, maintenance is not a cheap aspect of software engineering, and any increase in code comprehension that can reduce maintenance needs will also lower the overall development costs.

2.2 Definition of Terms

For future discussion, we will be defining segments of code within our contest as a set of code statements; where code statements are a minimum of one logical line of code per statement which can span multiple lines where each statement has one causal meaning. This definition is necessary to describe our proposed method below, where it was important to differentiate between lines of code and logical lines of code in order to be able to specify the input to our model as ignoring comments and necessary semantics that can vary from language to language but overall do not impact the code performance, such as opening and closing braces in Java or multiline strings used as block comments in Python. This definition lets statements contain multiple logical lines of code such as if an expression spans multiple lines. Lastly, we define interesting as whether or not a given code segment has a high value impact on the overall codebase.

2.3 State of the Art

Our tool focuses on the use of machine learning to label individual code statements. Through researching the state of the art, it was found that many tools attempt to label statements via static code analysis [5, 6, 9, 14, 15] however fail on the front of being too specialized in what language they analyze or types of statements they interpret (such as Regex [6]). Other tools rely entirely on human interaction by manually providing labels of some form or the other [11, 13, 19, 21, 24, 26]. Lastly, many tools within the state of the art use dynamic code analysis [3, 7, 22, 27]; an approach that is beneficial, but too reliant on availability of bytecode and suffers from the same issues described for the static code analysis tools. Overall, we saw a heavy emphasis on human interaction dependent tools over automated ones.

Of all the tools researched, there was a blatant trend of focusing heavily on a single IDE or Programming language rather than considering the concept of universal semantics within algorithmic languages. Some tools propose interpreting the source code via graphs of some sort [14], requiring manually creating a meta-model or understanding from such graphs. We pose that one added benefit of our approach tends toward more language-agnosticism than many other tools. Not only does our tool provide this agnostic perspective, allowing for ubiquity throughout the domain of software engineering, but it also provides the ability to generate informative perspectives of a code base at an atomic granularity, an advantage over many tools that were focused instead on high level perspectives or direct generation of meta-models. Many of these meta-models these tools focused around pertained to how well some code is written, generating documentation from code, and code behavior from source code such as whether it is functional or entertainment based [25] and if code has malicious intent [20]. With the atomic granularity we are able to generate, we pose that many problems above tools attempt to solve can also be solved from our data set, where the focus of this paper, hence our meta-model of interest, was whether given segments of code are interesting or not.

Since the prime directive of our research has been laid out, and we have touched on pre-existing tools attempting to solve code classification, our focus will now switch to describing the benefits of our tool compared to those mentioned previously and the current state of the art. Primarily, the efficiency of training a machine learning algorithm over writing a parser. Without machine learning, it would be necessary to write some form of parser in conjunction with static code analysis tools (or, if unavailable, the creation of such) in order to heuristically analyze and label lines of code based on weighted conditions and heuristics. This approach is extremely time consuming, challenging, and tedious [8, 17]. Even after accomplishing this method, the issue of determining experimentally accurate, optimized weights and heuristic subroutines for an experiment alone is a challenging and esoteric area of research across fields [23]. Hence, the speed and ease of creating modern machine learning models [18] has a heavy advantage over the archaic method of writing parsers or manually labeling code statements. Many of the tools in this field, as previously stated, depend on creating singular meta-models – meaning that developing a parser to produce such a meta-model would be highly specific to a given issue - since many language choices pertain to the use cases attributed to such language - and could not be utilized for designing other meta-models. Hence, this machine learning approach is extensible and modular to as many meta-models as a code-base may contain.

An additional advantage to the machine learning approach over a parser is that of community involvement and crowd-sourcing: with traditional algorithmic approaches there exist large open-source communities, however, when considering the numerous amounts of programming languages actively used in the field, if a developer wanted to analyze their code base they would either have to design a parser as stated above, or wait for the open-source community to do it themselves. Similarly, if bugs or issues are found with a given parser, it is up to the primary code caretakers or open-source contributors to fix these bugs in their free time. If instead of parser a machine learning model was created, then rather than open-sourcing the individual parsing algorithms, authors would only need to crowd-source the training data: users could provide manually labeled data to the model for it to train on, improving accuracy and language reachability. Rather than needing to write a new parser from scratch to interpret more programming languages, all that is needed is raw, prewritten source code that has been labeled. This benefit is twofold; firstly, many languages can have radically dissimilar syntax and semantics that would make writing new parsers an incredibly arduous task incapable of much code reusability, and secondly, adding new language support relies only on code that has already been written and labeled – the labeling can be done with the methods provided in tools described above.

The impetus behind using machine learning for code classification can of course be backed by research done by Facebook in static code analysis powered by machine learning for the purpose of preemptive bug finding [2] or with TabNine and their tool that provides code auto-completion based on interpreting static code via machine learning [1]. Many other tools have similarly been able to analyze source code behavior to some level, and even to provide contexts to developers in a sense we have discussed of our proposal, however once again they rely too heavily on a singular IDE or language.

Therefore, we propose our novel method of machine learning aided static code analysis, agnostic of the source programming language. We have shown that this is a widely studied field that is lacking in a tool of the atomic granularity we provide, as there exist minimal multi-purpose meta-model generating tools such as ours. We pose that machine learning provides more benefits to this field than naive parser algorithms based on the high extensibility of a machine learning model.

3 PROPOSED METHOD

Our proposed method is to create a machine learning model to classify code statements into labels of their stereotype and then cluster said labels through another machine learning model to determine whether or not a code statement is what we determine to be "interesting" or not. Our proposed method proceeds in two parts, Classifying code statements, and clustering the statements into a meta-model.

3.1 Classifying Code Statements

We define an interesting code statement to be one that contains diverse labels with high code impact (e.g: not print statements, log statements, or basic variable assignment). The labels we have defined and used experimentally have been decided to be agnostic of the programming language - as each are basic features of popular, core programming languages - and more or less ubiquitous. The labels are as follows:

List 1: Classification Labels

(1) Conditional
  - Ternary
  - Else
  - If
  - Else if

(2) Loop
  - Do
  - While
  - For
  - Foreach

(3) Primitive assignment
(4) Non-primitive assignment
(5) Method Declaration
(6) Method call
(7) Uninteresting
(8) Return
(9) Other

The "other" category is a catch-all for those features which are not available in every single language. Given all the features of every language, the union of each label except the "Other" label corresponds to the conjunction of all common features among programming languages observed; conditional statements, loops, variable assignments, and method/function calling. Given that some features considered ubiquitous may not necessarily be so, for example the Foreach loop structure in C and early C++ not being supported, missing classifications can just be ignored, hence our results are not impacted. Examples of what we consider to be "other" would be lambda expressions and "goto" calls for one. These have a large impact on the running of the code, so by default we shall consider those caught by the "other" label to be proportionally more important to the result than the other labels.

The "Uninteresting" label is that which we declare by default to not be impactful to the code under any circumstance. This would entail logging statements, print statements, and things like multi-line strings in Python used as comments. Because these are such trivial statements, we did not implement it into the classification model - though it could be if just included in the training data set - we decided to just leave it out of our input by not scraping for this information when generating the input.

We began research using linear discriminant analysis (LDA) for classifying labels to test the feasibility of our code, and we were able to distinguish between conditional statements and loop statements with 96% accuracy.

Our core model is based on a sequential Keras model based around a multi-classification support vector machine (SVM). The SVM model was chosen as it has a direct correlation to linearly independent vectors in multi-dimensional spaces and can efficiently perform such analysis for non-linear classification. This is beneficial because we consider that each statement of code can exist in only one label at a time, hence we represent a statement as a vector in multi-dimensional space where its label is the probabalistically closest unit vector in this multidimensional, linearly independent space, where each linearly independent unit vector is a possible label.

The input to our model is formatted according to Listing 1, in JSON. The classification field is labeled "N/A" for unclassified data.

Listing 1: Example Model Input

```
{
    "fileName": "some_file.py",
    "statement": "if x == True:",
    "lineNumber": 13,
    "classification": "Conditional"
}
```

Because we track line numbers in our input, we are able to track this and keep it for our output. As such, a method such as in Listing 2 may wind up looking according to the list of classifications in Listing 3.

Listing 2: Example Code Segment

```
double exampleMethod() {
    if (predicate) {
     int x = 2;
     internalMethodCall();
     Object o = new Object();
     Object o2 = factoryMethod(x);
     while (predicate) {
      x++;
      o.setVal(o2.getVal() / x);
      print(   value   :     + o.getVal());
      if (predicate) break;
     }
     print(    something     );
     return (double) o.getVal() + x;
    }
}
```

When running our model on the above method having trained it for Java syntax, we receive a JSON file containing the statements, line numbers, and classifications. With some formatting to the standard output, we can produce the results seen in Listing 3.

Listing 3: Example List of Labels generated from Listing 2

```
file::1::MethodDeclaration
file::2::Conditional_If
file::3::Assignment_Primitive
file::4::Method_Call
file::5::Assignment_NonPrimitive
file::6::Method_Call
file::7::Loop_While
file::8::Assignment_Primitive
file::9::Method_Call
file::10::Uninteresting
file::11::Conditional_Ternary
file::13::Uninteresting
file::14::Return
```

3.2 Clustering Statements into a Meta-Model

After we have labeled each statement, we can proceed to generate a meta-model from the code-base. The code-base we are concerned with and have produced, is whether or not a segment of code (set of statements) is "interesting" or not.

Our initial approach to deciding this was based on the idea of entropic weights. We had assigned an entropic weight to each label variety and then performed various methods of entropy calculations on the segments to determine whether it as a whole is of note or not. Methods we approached were naive string entropy calculation by considering each label's weight as a metric for weighing the entropy from 0.0 to 1.0, however we found this to be inaccurate when compared against segments we determined by hand to be important whose entropic value showed otherwise. This proved to be an issue with both the approach and our experimental weights.

Our second and more heuristic approach was based around a moving selection window which would move randomly between the statements provided in order and would use the entropic weights to create sub-segments with entropy and try to maximize the selection window with respect to the entropy, if no window of a given threshold was found we considered the segment as a whole to be uninteresting. We found that this not only provided a run time that was $\Theta(n^3)$ but also that it again relied too heavily on the entropic weights and provided a negligent boost to accuracy of only roughly 12%, which could be considered a fluke due to the small data set we created by hand.

Finally, we settled on the approach of just creating a secondary SVM model and using hand-crafted data. We found a substantial boost to accuracy of determining whether a segment was interesting or not, however we were stunted in our results due to the lack of an extensive set of training data. The results of such can be seen in the table below. In total, 30 segments in both Python and Java were hand-created and labeled interesting or not. A benefit of this model is that a segment is not purely dependent on being a method block entirely, it may be a subset of a method's body. Though there were only 10 methods to be tested against, neither SVM got 100% accuracy. This is suspected to be a false negative rather than a false positive.

Table 1: Results from 3 different approaches and hand-crafted training data

Language	Naive	Heuristic	SVM	Segments
Python	50%	60%	90%	20:10
Java	50%	50%	90%	20:10

4 CASE STUDY

4.1 Classifying Labels

The training data we used for this method was initially created randomly through a python program we developed. We used a dictionary of possible characters forming strings of variate lengths as viarable names and generated arbitrary boolean expressions from which we formatted them first into Java syntax-based statements and on second run Python syntax-based statements. With an LDA model classifying between only 2 labels for experimenting validity with randomized data, we achieved only 61% accuracy on the untrained data set, although Keras [?] boasted a hefty 98% accuracy on the training data set. We concluded this discrepancy to be caused by over-fitting by our randomized data not providing enough variety.

To cure this over-fitting, we created a Python program to scan and collect lines of code from 10 of the top Python repositories on Github based on keywords. Running this same LDA experiment again for Conditionals and Loops we acheived a validated accuracy of 96%. With this in mind as our rationale, we continued by expanding to more label classifications.

However, due to time constraints and data sizes of said repositories, we only used roughly 10% of the Python code base in each repository in order to achieve generalization of data across multiple sets. We used naive keyword extracting and string pattern matching to generate input data to the training model.

Table 2: Python Github Repositories

Repository	Stars	Commits	Python LOC
ansible	40.8k	48,501	969,423
Django	45.7k	27,676	242,326
Sentry	23.2k	27,656	209,334
youtube-dl	58.5k	17,457	120,148
Scrapy	35.3k	7,592	27,016
Tornado	18.6k	4,142	26,698
YouCompleteMe	20.2k	2,585	10,490
Flask	47.8k	3,799	9,675
requests	41k	5,923	5,869
Httpie	44.9k	1,137	4,458

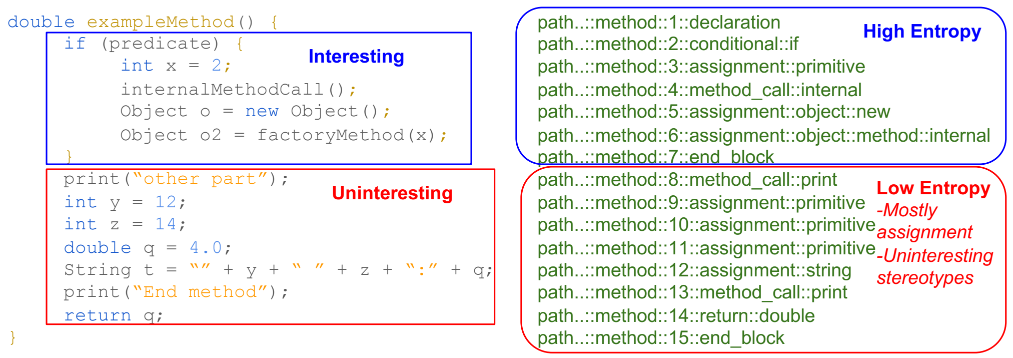
After training we were able to analyze input code snippets and classify them with 92% accuracy with all 15 labels that were applicable to Python. We believe that by increasing the data set dramatically we will be able to similarly increase the accuracy of our predictions.

4.2 Clustering Statements into Meta-Model

After validating our model for label classification, we needed to design an experiment to determine if accurate meta-models could be derived from these classifications. We proposed to select random segments of code from the repositories used in the training data and determine by hand whether they were interesting or not. We would then as input take segments that had been previously classified and run them through this new SVM model to determine "interestingness."

The problems that arise in this are that of an internal bias, potential over-fitting, and a lack of data due to how long it takes to create such data. The internal bias is based on the fact that we are creating input for our own proposition, similar to when we created our own random data and were unaware of the flaws therein. Also, since we are not the developers or experts involved in the projects our perspective of impactful code may be skewed or flawed. The potential over-fitting is always an issue when creating unlabeled data from the training data-set; we must validate that no statements in this segment input appeared in the statement input. Lastly, depending so much on manual labeling like this can be flawed due to human error and takes a lot of development time away from running the experiments and testing validity.

Because of this, we have not yet classified segments into our meta-model of interesting versus not-interesting as we are working



```
double exampleMethod() {
    if (predicate) {
        int x = 2;
        internalMethodCall();
        Object o = new Object();
        Object o2 = factoryMethod(x);
    }
    print("other part");
    int y = 12;
    int z = 14;
    double q = 4.0;
    String t = "" + y + " " + z + ":" + q;
    print("End method");
    return q;
}
```

```
path..::method::1::declaration
path..::method::2::conditional::if
path..::method::3::assignment::primitive
path..::method::4::method_call::internal
path..::method::5::assignment::object::new
path..::method::6::assignment::object::method::internal
path..::method::7::end_block
path..::method::8::method_call::print
path..::method::9::assignment::primitive
path..::method::10::assignment::primitive
path..::method::11::assignment::primitive
path..::method::12::assignment::string
path..::method::13::method_call::print
path..::method::14::return::double
path..::method::15::end_block
```

Figure 1: Example of splitting a segment into 2 sub-segments based on heuristic approach

on creating a fair, reasonable experiment. Our goal is to engage an expert in the field for some external 11th Python repository to give us a list of interesting code segments from which we can train and test our model on.

5 THREADS TO VALIDITY

We base our definition of interesting code segments as the most part of software that helps understand the general structure, meaning and strip of the unnecessary boilerplate code. The segments cannot be perfect from its nature, because every developer will have an opinion on what is and what is not important. We try to mitigate this issue by constructing learning model, which can be adjusted by the specific needs or opinion of an organization.

Another concern raises from the use of the precision of the model, its overfitting or false positive ratio. This issue can be mitigated in the long run by common contribution of number of subjects to single model. The bigger the model, the better the results will be.

6 CONCLUSION

There are new challenges in the recent years concerning development with introduction of agile techniques and microservices. Adopting someone else code has become deal for every level of developer taking up significant portion of the time.

Current solutions are language dependent and tackle understanding the code via debugging or linking documentation, but ignore the most significant part of the code adoption, which understanding of raw code. We focus on our work to deliver a language independent solution for identifying most important code segments. We base our solution on two deep learning models. First model is decision tree regression model to recognize individual statements. We tested the model on open source projects and received precision over 90 % in all tested categories. The second one is logistical model for identifying the segments.

In the future work, we want to open our model for anybody to improve its precission and to conduct more tests for different languages and projects. We will focus on accuracy and performance of the method to be suitable for real time static analysis. We seek opportunities to deploy the system to production in real development environment and measure its impact on effectivity of developers, distruption mitigation and lowering the cost when adopting a new project.

REFERENCES

[1] [n. d.]. Home. http://www.tabnine.com/
[2] [n. d.]. Infer: A Look Into Facebook's New Java Static Analysis Tool. https://blog.overops.com/infer-a-look-into-facebooks-new-java-static-analysis-tool/
[3] [n. d.]. Visual monitoring of numeric variables embedded in source code | Mobile | The Best Way to Share & Discover Documents. https://dochot.net/philosophy-of-money.html?utm_source=visual-monitoring-of-numeric-variables-embedded-in-source-code
[4] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. 1993. Software Complexity and Maintenance Costs. *Commun. ACM* 36, 11 (Nov. 1993), 81–94. https://doi.org/10.1145/163359.163375
[5] Gabriele Bavota, Luigi Colangelo, Andrea De Lucia, Sabato Fusco, Rocco Oliveto, and Annibale Panichella. [n. d.]. TraceME: Traceability Management in Eclipse. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (2012-09). 642–645. https://doi.org/10.1109/ICSM.2012.6405343 ISSN: 1063-6773.
[6] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. [n. d.]. RegViz: Visual Debugging of Regular Expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering* (2014) *(ICSE Companion 2014)*. ACM, 504–507. https://doi.org/10.1145/2591062.2591111 event-place: Hyderabad, India.
[7] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. [n. d.]. In situ understanding of performance bottlenecks through visually augmented code. In *2013 21st International Conference on Program Comprehension (ICPC)* (2013-05). 63–72. https://doi.org/10.1109/ICPC.2013.6613834 ISSN: 1092-8138.
[8] Der Universität Bern, Markus Kobel, Prof Dr, Horst Bunke, Tudor Girba, Prof Dr, and Michele Lanza. [n. d.]. *Leiter der Arbeit: Prof. Dr. Oscar Nierstrasz.*
[9] Dirk Beyer and Ashgan Fararooy. [n. d.]. DepDigger: A Tool for Detecting Complex Low-Level Dependencies. In *2010 IEEE 18th International Conference on Program Comprehension* (2010-06). 40–41. https://doi.org/10.1109/ICPC.2010.52 ISSN: 1092-8138.

[10] T. A. Corbi. [n. d.]. Program understanding: Challenge for the 1990s. 28, 2 ([n. d.]), 294–306. https://doi.org/10.1147/sj.282.0294

[11] Uri Dekel and James D. Herbsleb. [n. d.]. Improving API documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering* (2009). IEEE, 320–330. https://doi.org/10.1109/ICSE.2009.5070532

[12] Chris Doig. [n. d.]. Calculating the total cost of ownership for enterprise software. https://www.cio.com/article/3005705/calculating-the-total-cost-of-ownership-for-enterprise-software.html

[13] Anja Guzzi, Lile Hattori, Michele Lanza, Martin Pinzger, and Arie van Deursen. [n. d.]. Collective Code Bookmarks for Program Comprehension. In *2011 IEEE 19th International Conference on Program Comprehension* (2011-06). 101–110. https://doi.org/10.1109/ICPC.2011.19 ISSN: 1092-8138.

[14] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. [n. d.]. Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency. 217–224. https://doi.org/10.1145/2047196.2047225

[15] Samuel Klock, Malcom Gethers, Bogdan Dit, and Denys Poshyvanyk. [n. d.]. Traceclipse: an eclipse plug-in for traceability link recovery and management. In *Proceeding of the 6th international workshop on Traceability in emerging forms of software engineering - TEFSE '11* (2011). ACM Press, 24. https://doi.org/10.1145/1987856.1987862

[16] Michael Krigsman. [n. d.]. Danger zone: Enterprise maintenance and support. https://www.zdnet.com/article/danger-zone-enterprise-maintenance-and-support/

[17] Bruce McCorkendale, Xue Feng Tian, Sheng Gong, Xiaole Zhu, Qingchun Meng, Ge Hua Huang, and Wei Guo Eric Hu. [n. d.]. (54) SYSTEMS AND METHODS FOR COMBINING. ([n. d.]), 15.

[18] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, and David Cournapeau. [n. d.]. Scikit-learn: Machine Learning in Python. ([n. d.]), 6.

[19] M. Revelle, T. Broadbent, and D. Coppit. [n. d.]. Understanding concerns in software: insights gained from two case studies. In *13th International Workshop on Program Comprehension (IWPC'05)* (2005-05). 23–32. https://doi.org/10.1109/WPC.2005.43 ISSN: 1092-8138.

[20] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. [n. d.]. Automatic analysis of malware behavior using machine learning. 19, 4 ([n. d.]), 639–668. https://doi.org/10.3233/JCS-2010-0410

[21] Martin Robillard and Frédéric Warr. [n. d.]. ConcernMapper: Simple view-based separation of scattered concerns. 65–69. https://doi.org/10.1145/1117696.1117710

[22] André Santos. [n. d.]. GUI-driven code tracing. 111–118. https://doi.org/10.1109/VLHCC.2012.6344495

[23] Paul J. H. Schoemaker and C. Carter Waid. [n. d.]. An Experimental Comparison of Different Approaches to Determining Weights in Additive Utility Models. 28, 2 ([n. d.]), 182–196. https://doi.org/10.1287/mnsc.28.2.182

[24] Philipp Schugerl, Juergen Rilling, and Philippe Charland. [n. d.]. Beyond generated software documentation — A web 2.0 perspective. In *2009 IEEE International Conference on Software Maintenance* (2009-09). 547–550. https://doi.org/10.1109/ICSM.2009.5306385 ISSN: 1063-6773.

[25] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. [n. d.]. Automated Static Code Analysis for Classifying Android Applications Using Machine Learning. In *2010 International Conference on Computational Intelligence and Security* (2010-12). 329–333. https://doi.org/10.1109/CIS.2010.77 ISSN: null.

[26] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. [n. d.]. Shared Waypoints and Social Tagging to Support Collaboration in Software Development. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work* (2006) *(CSCW '06)*. ACM, 195–198. https://doi.org/10.1145/1180875.1180906 event-place: Banff, Alberta, Canada.

[27] Ben Swift, Andrew Sorensen, Henry Gardner, and John Hosking. [n. d.]. Visual code annotations for cyberphysical programming. In *2013 1st International Workshop on Live Programming (LIVE)* (2013-05). 27–30. https://doi.org/10.1109/LIVE.2013.6617345 ISSN: null.