Dissertation on

"Code Semantic Detection and Optimization"

Submitted in partial fulfilment of the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science & Engineering

UE17CS490A – Capstone Project Phase - 1

Submitted by:

Bhavna Arora	**PES1201700062**
G R Dheemanth	**PES1201700229**
Skanda VC	**PES1201700987**
Mehul Thakral	**PES1201701122**

Under the guidance of

Prof. N.S. Kumar
Professor
PES University

August - December 2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

'Code Semantic Detection and Optimization '

is a bonafide work carried out by

Bhavna Arora	**PES1201700062**
G R Dheemanth	**PES1201700229**
Skanda VC	**PES1201700987**
Mehul Thakral	**PES1201701122**

in partial fulfilment for the completion of seventh-semester Capstone Project Phase - 1 (UE17CS490A) in the Program of Study - Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period Aug. 2020 – Dec. 2020. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 7th-semester academic requirements in respect of project work.

Signature	Signature	Signature
Prof. N.S. Kumar	Dr Shylaja S S	Dr B K Keshavan
Faculty	Chairperson	Dean of Faculty

DECLARATION

We hereby declare that the Capstone Project Phase - 1 entitled **“Code Semantic Detection and Optimization”** has been carried out by us under the guidance of N S Kumar, Professor and submitted in partial fulfilment of the course requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester August – December 2020. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

PES1201700062	**Bhavna Arora**	[illegible]
PES1201700229	**G R Dheemanth**	[illegible]
PES1201700987	**Skanda VC**	[illegible]
PES1201701122	**Mehul Thakral**	Mehul

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. N S Kumar, Department of Computer Science and Engineering, PES University, for his continuous guidance, assistance, and encouragement throughout the development of this UE17CS490A - Capstone Project Phase – 1.

We are grateful to the project coordinators, Prof. Silviya Nancy J and Prof. Sunitha R, for organizing, managing, and helping with the entire process.

We take this opportunity to thank Dr Shylaja S S, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from the department. We would like to thank Dr B.K. Keshavan, Dean of Faculty, PES University for his help.

We are deeply grateful to Dr M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro-Chancellor – PES University, Dr Suryaprasad J, Vice-Chancellor, PES University for providing us various opportunities and enlightenment every step of the way. Finally, this project could not have been completed without the continual support and encouragement we have received from our family and friends.

ABSTRACT

Code semantics play a crucial part in code classification, code summarization, code search, and code optimization. This project was developed to classify programs based on the high-level logic of the programs. The developed solution uses the idea of supervised learning classification for a set of a labelled dataset of code snippets. The code is vectorized using two novel techniques based on dynamic and static analysis. The former uses a new technique to generate a code vector by approximating the given function using a neural network. The weights of the neural network being the heart of the neural network are used to generate the final vector, which represents the code/function snippet. The latter, i.e. Static Analysis uses the structural information of code to generate the representation. Our two new techniques beat the state of the art systems code2vec and neural code comprehension. The accuracies are 98% and 80% for dynamic and static analysis based techniques respectively.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER-1

INTRODUCTION

Imagine a program which can write a program. This is what is called machine programming. Machine Programming Domain deals with the automation of software development. In order to solve this open problem, code classification is the most important first order problem which must be solved. It also forms the basis for another interesting and important problem which is code optimization. It also forms an important part in many other related problems like code summarization, code search, etc as stated earlier.

In order to solve this crucial problem, we propose two new and unique solutions. One is based on dynamic analysis and the other on static analysis. In both the cases, input is in the form of a function and the output is the code semantics.

In Dynamic Analysis, the given function is executed to get the semantics. The function is seen as a black box here. In Mathematics,

"a function is defined as a mapping which associates elements in one set with exactly one element of a group of elements in another set".

It can be observed from the definition that a function can be uniquely represented if there is a way to obtain the mapping (which is unique) from the input and output for that function. This forms the main motivation for dynamic analysis. In Machine Learning, a neural network is seen as a function approximator which can approximate any function with reasonable accuracy. This is proved by the Universal Approximation Theorem (UAT). It states that

"a neural network with one hidden layer containing a sufficient but finite number of neurons can approximate any continuous function to a reasonable accuracy" .

Therefore with a neural network, a unique representation for the mapping can be obtained. Since weights form the heart of the neural network, the weights are used to build the vector which later can be used for other tasks. The neural network is trained by generating random inputs and feeding it to the function to get the outputs. The inputs and outputs are used to train the neural network from which the weights are extracted.

In Static Analysis, the function structure and the code flow play a significant role in predicting the intent of the code. The syntactic structure of the code can be captured using a parse tree. The parse tree represents the given string or code snippet according to the context-free grammar of the programming language. The parse tree is too detailed to capture the context or the structural aspect of the code.

The abstract syntactic structure of the given source code for a particular programming language can be represented in the form a tree using an abstract syntax tree (AST). Every node of the tree represents a construct (such as variable declaration, conditional statements, function call) occurring in the function snippet.
For example, the if-condition-then expression would be denoted by means of a single node with three branches.

During the source code translation & compilation process, the parser builds the Parse Tree. After building the parse tree, performing subsequent processing (eg: contextual analysis) helps to add additional information to the AST. The advantage of AST over parse tree is particularly the smaller height and smaller number of elements.

Bytecode (also called portable code) is a compact numeric code represented using constants, and references (typically numeric memory addresses) to encode the result of parsing phase of compiler and to perform semantic analysis based on type, scope, and nesting depths of program objects. Efficient execution by a software interpreter or compiler is enabled by conversion of source code to bytecode. The two ideas described above form the basis of the static analysis, which sees the code snippet as a white box. It is observed that codes having syntactic and structural similarity tend to have similar semantics. Therefore they have similar program dependence and control flow. The AST and Bytecode intermediate representations help to ignore low-level details of the source code. This allows the model to extract the main context of the program snippet, which helps in the unique representation of every program snippet.

In both dynamic and static analysis, the obtained vectorized representation is used to perform classification to detect the label of a new input code snippet.

CHAPTER-2

PROBLEM DEFINITION

Given a code snippet in the form as a function as the input, the problem is to find the high-level logic of the given input function. The problem statement can be summarized in the below diagram. In the project, python programs were used as inputs to the semantic detector.

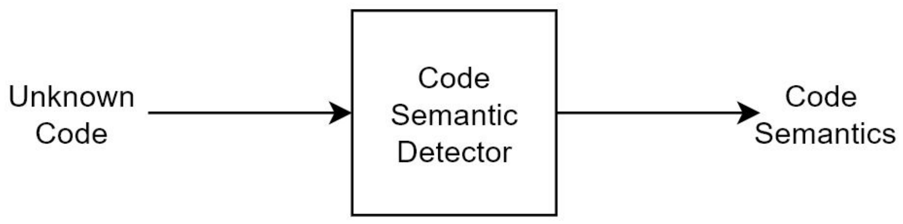
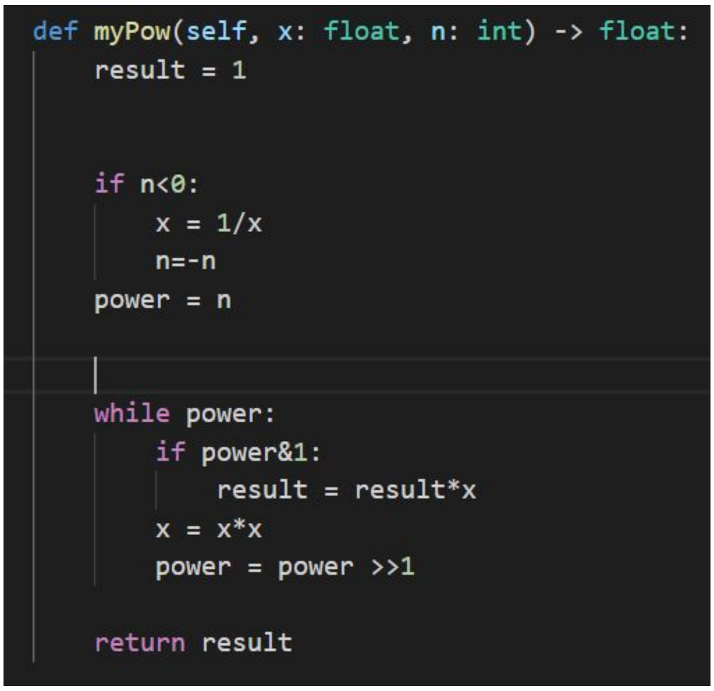


Fig 1. Block diagram of problem statement

The Input has to be a valid python program. Furthermore, the output is a python dictionary which gives possible high-level logic of the program as the key with corresponding probabilities as the values.

For Example: If Input is given as the given program snippet below, then the output will be {"POW"}.



```python
def myPow(self, x: float, n: int) -> float:
    result = 1


    if n<0:
        x = 1/x
        n=-n
    power = n


    while power:
        if power&1:
            result = result*x
        x = x*x
        power = power >>1

    return result
```

Fig 2. Python Function Snippet

The above function pow can be written in many ways, and there are different algorithms to implement the above function. However, all those implementations must be mapped to pow only. So the above problem can be seen as a classification problem. In order to classify, the program must be converted to a form which can be fed to the classification algorithms. So the function/code snippet must be converted to a vector. This is the crux of the problem.

For the purpose of classification, a dataset was generated which is a collection of around 500 programs scrapped from leetcode discussion threads spanning across 25 labels from about 8 data structure/algorithm categories.

CHAPTER-3

LITERATURE SURVEY

3.1 Code2Vec: Learning Distributed Representations of Code [1]

Uri Alon et al

The paper deals with predicting code semantics in the form of method names using Neural Network by representing the code snippet as a set of syntactic paths in its AST.

3.1.1 Introduction

The task of semantic labelling is accomplished by learning code embeddings which are low-level vector representations and a code is made of multiple vector components, hence the representation is said to be “distributed”. Such a task is difficult as the single semantic label should be representative of the entire code.

The problem is approached using neural networks. Code embedding enables the modelling of correspondence between the code snippet and semantic label. The code is mainly represented as a collection of context paths in AST which are mainly aggregated using Attention Neural Network to get a final vector representing the semantic label.

The main contributions of the paper include:

1. An attention model which learns vectors for arbitrary-sized snippets of code by extracting paths in the AST.
2. Method-name embedding to code vector mapping which could be used for other similar tasks.

3.1.2 Proposed Model

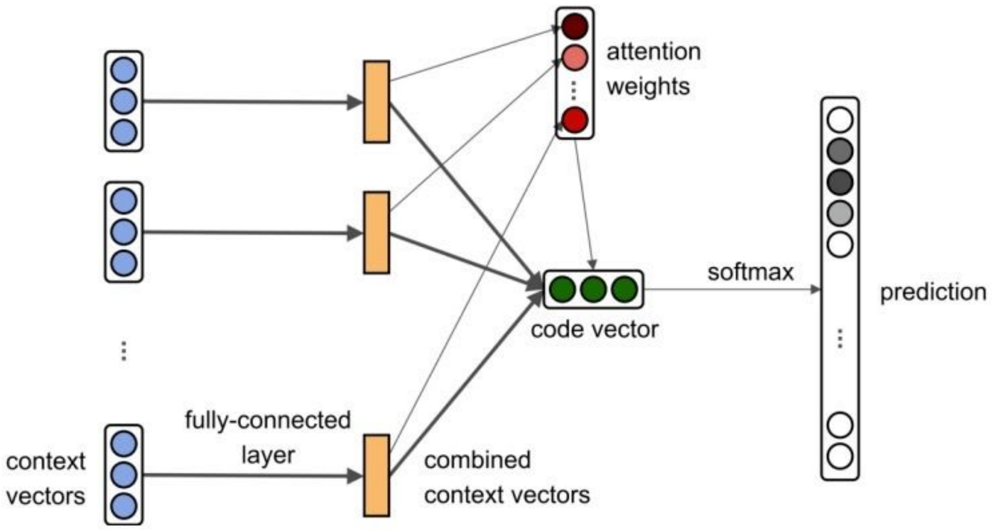
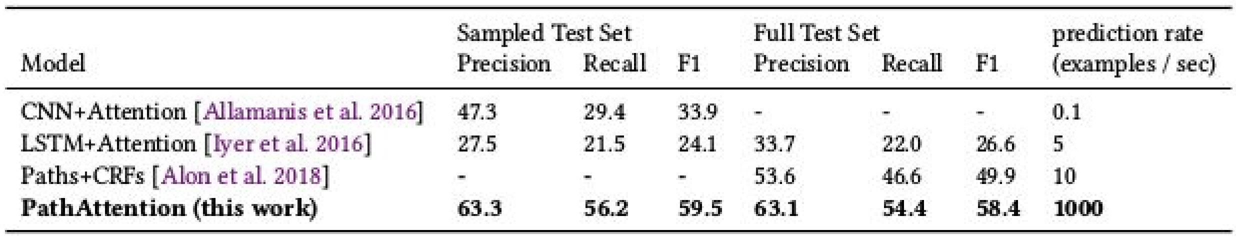


Fig3. The architecture of code2vec model

The proposed model involves first extracting contexts from AST of code in the form of a bag of context vectors (i.e. vectors representing AST paths). This bag or collection of context vectors is combined when passed through a fully-connected layer to form a single vector for each context, which are further combined using Attention NN to form a single vector for the code. The Neural Network learns to assign attention to each context in the form of weights. The combined vector passed through softmax activation is compared with *tags* vocabulary to assign a semantic label.

3.1.3 Evaluation

As a part of the evaluation, the proposed model was compared with several previous works, and the obtained results are presented in Fig4. The results show that it was able to perform better than other models undertaking a similar task.



	Sampled Test Set			Full Test Set			prediction rate
Model	Precision	Recall	F1	Precision	Recall	F1	(examples / sec)
CNN+Attention [Allamanis et al. 2016]	47.3	29.4	33.9	-	-	-	0.1
LSTM+Attention [Iyer et al. 2016]	27.5	21.5	24.1	33.7	22.0	26.6	5
Paths+CRFs [Alon et al. 2018]	-	-	-	53.6	46.6	49.9	10
PathAttention (this work)	**63.3**	**56.2**	**59.5**	**63.1**	**54.4**	**58.4**	**1000**

Fig4. Evaluation comparison between code2vec model and previous works

The learnings from the paper include:

1. The representation is able to capture code structural information which helps to determine semantically similar code.
2. It is easier to learn this representation than to train on the text of a program.
3. Unlike semantic analysis this doesn't require expert knowledge of language and don't have to manually select features to be considered while learning.
4. It can be extended to other languages by simply replacing the parser for extracting AST paths.

The shortcomings of the paper include:

1. Does not generalize well.
2. Ability to find code semantics is not good as the model is mainly trained for method name prediction.
3. Not able to produce complex and specific method names as was trained on generic names.
4. Method name prediction depends on variable names used in the code snippet.

3.2 Learning Semantic Vector Representations of Source Code via a Siamese Neural Network [2]

Wehr, David, et al

This paper presents the use of a siamese neural network to generate vectors which represent the semantics of a code snippet.

3.2.1 Introduction

The authors introduce the paper with the current scenario regarding code semantic analysis. They go on to mention that there are four different types of code clones, as stated by Roy and Cordy [5], which include functionally similar but semantically different programs (Type IV).

Furthermore, they state the primary motivation behind this paper as the lack of significant research on these Type IV programs. Moreover, most papers focus on just the syntax and structure while sidelining the semantics. The main goal of this paper is to represent syntactically different programs having the same functionality with the same vector.

The proposed contributions of the paper are:

1. Detecting code semantics from the similarity between programs
2. Can use unlabelled datasets for training which reduces the need for scarce labelled datasets
3. Using semantically similar programs to learn semantics even if their abstract syntax trees (AST) have different structures.

3.2.2 Proposed Model

The authors propose using different NLP techniques on the code snippets. The main hurdle in doing this, as mentioned in the paper is that NLP techniques work on a linear sequence of words. So to overcome this, the Abstract Syntax Tree (AST) was flattened into a sequence using the structure-based traversal (SBT). The token vocabulary of the NLP techniques was limited to the most common tokens. This flattened sequence is then inputted to the siamese neural network whose overview is given in the below diagram. The generated vectors from the model are then used to find the distance between them and the cosine similarity to determine if the programs are semantically similar or not. An LSTM based RNN is used to generate the vector from the SBT representation.

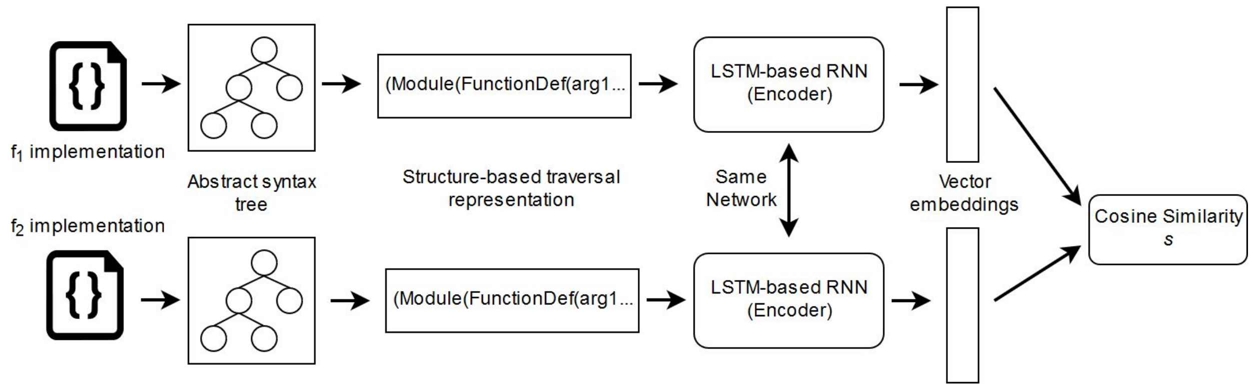


Fig5. Structure of Siamese Neural Network

3.2.3 Evaluation

The dataset used for pre-training (RNN encoding stage) was generated by scraping GitHub repositories having more than 100 stars. Furthermore, the dataset for the siamese neural network was scrapped from Hackerrank. The model was compared with the naive model (bag of words) and was found to have an improvement of around 15%. The model was not tested against existing benchmarks like BigCloneBench as the model only supports python.

3.3 Detection of Semantically Similar Code [3]

Tiantian WANG et Al

3.3.1 Introduction

This paper presents a combination metric and graph-based approach to find the semantically similar code snippets.
The metric-based approach detects near-miss similar codes but has a low precision value.
The graph-based approach considers the syntactic structure of programs as well as the data flow (as an abstraction of the semantics). However, a graph-based approach fails to detect code variations and thus lead to high computational complexity.
The combination of the metric and graph-based approach is used to detect semantic code similarity.

3.3.2 Proposed Model

The steps involved in the implementation are as follows:

i. Represent the source code as an augmented system dependence graph

ii. To lower the computational complexity, filter out dissimilar code snippet based metrics-based candidate similar code extraction

iii. Perform normalization , only on candidate similar codes to reduce computational expense

iv. Perform program matching on the normalized control dependence trees (CDT) and return the semantically similar code snippets

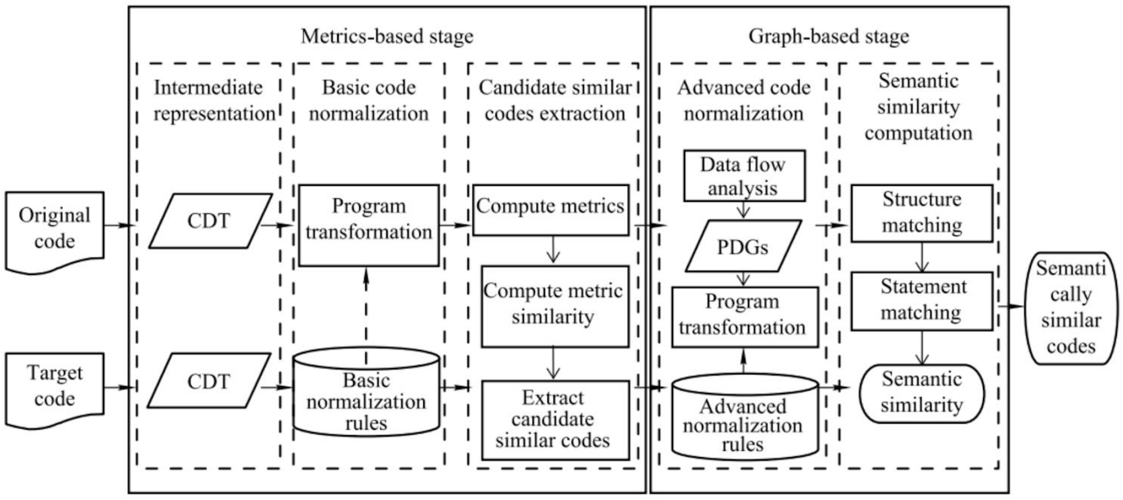

Metrics-based stage
Graph-based stage
Intermediate representation
Basic code normalization
Candidate similar codes extraction
Advanced code normalization
Semantic similarity computation
Original code
Target code
CDT
CDT
Program transformation
Basic normalization rules
Compute metrics
Compute metric similarity
Extract candidate similar codes
Data flow analysis
PDGs
Program transformation
Advanced normalization rules
Structure matching
Statement matching
Semantic similarity
Semantically similar codes

Fig6. Combined metric-based and graph-based model for similar codes detection

The data flow analysis is only performed on the selected candidate similar codes, leading to significant reduction in the complexity of SDG representation.

Code normalization is proposed to eliminate code variations, and metric similarity is computed at a modular level which improves the accuracy of the computing metric.

Even after much normalization, the computational complexity of working with trees and graphs is high.

3.3.3 Evaluation

Combined Metric and Graph Approach (CMGA) performs better than the other graph-based similar code detection approaches, as it can distinguish semantically similar code with code variations. Handling code variations requires additional time. The program matching has polynomial time complexity based on the number of nodes in control dependence trees (CDTs). The space complexity is decreased as CMGA analyzes the data flow of the Program dependence graphs (PDGs) only for the part of the program.

3.4 Machine Inferred Code Similarity : MISIM [4]

Fangke ye et Al

3.4.1 Introduction

This paper provides a neural network-based code similarity scoring algorithm, which can be implemented with various neural network architectures with learned parameters.
It improves code similarity analysis: (i) its structural representation of code, called the context-aware semantic structure (CASS), and (ii) its neural-based learned code similarity scoring algorithm.

It provides a structural representation of code specifically designed to
(i) lift semantic meaning from code syntax and
(ii) provide an extensible representation that can be augmented contextual syntax ambiguities in programming languages

3.4.2 Proposed Model

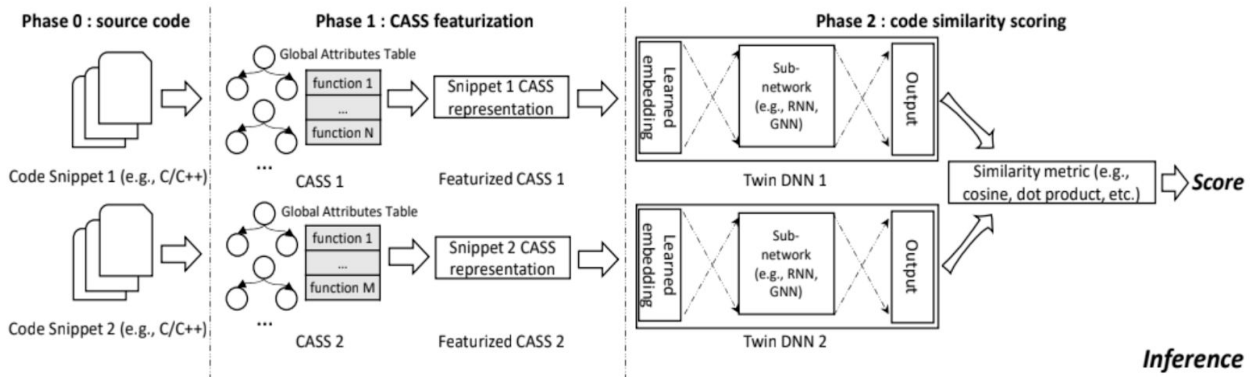

Phase 0 : source code
Phase 1 : CASS featurization
Phase 2 : code similarity scoring
Code Snippet 1 (e.g., C/C++)
Code Snippet 2 (e.g., C/C++)
Global Attributes Table
function 1
...
function N
function M
CASS 1
CASS 2
Snippet 1 CASS representation
Snippet 2 CASS representation
Featurized CASS 1
Featurized CASS 2
Learned embedding
Sub-network (e.g., RNN, GNN)
Output
Twin DNN 1
Twin DNN 2
Similarity metric (e.g., cosine, dot product, etc.)
Score
Inference

Fig7. MISIM proposed model architecture

The idea of the proposed model lies in the context aware feature extraction of the code snippet. The CASS (Context Aware Semantic Structure) consists of one or more CASS trees and an optional global attributes table(GAT). In CASS tree, the entire span of the code snippet is represented by the root node. During the construction of a CASS tree, the program tokens are mapped to their corresponding node label using the grammar of the programming language. The GAT contains exactly one entry per code snippet. It includes only the input and output cardinality values for each corresponding function, but can be extended as new global attributes are needed.

3.4.3 Evaluation

The learned embedding using CASS and GAT is passed to a RNN or GNN based neural network that generates the corresponding vectors for the input code snippets. The pairwise labels are used to train the model. Each pair of input programs are mapped to two code vectors by the model, from which a similarity score is computed. Cosine similarity in code vector space is used as a metric to compute similarity.

CHAPTER-4

PROJECT REQUIREMENT SPECIFICATION

4.1 PRODUCT PERSPECTIVE

4.1.1 Scope

The purpose of building this tool is to enable detection of semantically similar programs intended to solve the same problem but differing in their implementation.

Semantic similarity detection is an important aspect of code comprehension and code labelling which is very helpful in the current scenario where there exists large unlabelled unstructured data.

The goal of this project is to build a tool that is able to correctly predict the intent of an unseen program based on the model on the pre labelled dataset.

The scope of the tool is limited to the different labels in the dataset. The tool doesn't intend to predict a new label for a different program input based on the already existing labels.

4.1.2 Features

Major features include

1. Semantics preserving vector representation of code
2. Detection of high level logic of the program

4.1.3 User Classes and Characteristics

There are 3 major components :

1. The frontend (UI) : User interacts with this component via browser/api caller
2. The flask backend : calls the core modules
3. The core modules for code semantic detection: contains the modules which find code semantics. They are of 2 types :
    a. Dynamic Predictor
    b. Static Predictor

4.1.4 Operating Environment

1. The system can be deployed on any operating system using docker.
2. The UI can be accessed by any http client like web browsers, API callers like postman, etc

4.1.5 General Constraints, Assumptions and Dependencies

1. Input code snippet should be free from Syntactic and Semantic errors
2. Input code snippet should be a Function
3. Output Limitations - Restricted to the labels present in the dataset
4. Hardware limitations - GPU compute capability (For training the model)
5. Parallel operations : Neural Network training is a highly parallelized operation. Trained on GPUs.

4.1.6 Risks

1. System is susceptible to security vulnerabilities if the input code is not sanitized properly.

4.2 FUNCTIONAL REQUIREMENTS

1. Code Representation and Transformation
2. Dynamic Code Semantics Detection
3. Static Code Semantics Detection
4. Results Consolidation

4.3 EXTERNAL INTERFACE REQUIREMENTS

4.3.1 User Interfaces

1. User must specify the language of the code snippet being given to application
2. Requires a user friendly coding editor / file upload mechanism to take input from the user
3. Depiction of probabilities of different classes in graph/chart format
4. Specify appropriate error if the match doesn’t exist in the dataset or semantic couldn’t be detected

4.3.2 Hardware Requirements

1. GPU is necessary to increase the performance of the dynamic predictor module
2. Multi core CPU

4.3.3 Software Requirements

1. Docker : Docker is used to simplify the deployment of the entire system and make it OS agnostic.
    i. Name : Docker
    ii. Version : 19.03.13
    iii. OS : Windows, Linux, MAC
2. Tensorflow: Necessary for building Neural Network
    i. Name : Tensorflow
    ii. Version : 2.3.0
    iii. OS: Windows, Linux, MAC
3. Keras : High level API for building Neural Network
    i. Name : Keras
    ii. Version : 2.2.5
    iii. OS: Windows, Linux, MAC
4. Sklearn : python library for which has various ML models
    i. Name : sklearn
    ii. Version : 0.24

   iii. OS : Windows, Linux, MAC

5. Flask : python library to make backend
   i. Name : flask
   ii. Version : 1.1.2
   iii. OS : Windows, Linux, MAC

4.3.4 Communication Interfaces

The HTTP or HTTPS protocol(s) will be used to facilitate communication between the UI and backend.

4.4 NON-FUNCTIONAL REQUIREMENTS

4.4.1 Performance Requirements

1. Good Accuracy on a variety of classes
2. Low Prediction Time
3. Number of class of programs which can be classified

4.4.2 Safety Requirements

1. The code provided as input should follow the syntax and semantics of the input language as it would be executed by the system in order to predict the semantic properties.

4.4.3 Security Requirements

1. Although input code in any form is not stored in the system and is only processed once to predict the semantic properties, the client is responsible for ensuring that they have enough privileges and rights for the code and is not part of any group/organization's IP(Intellectual Property) .
2. As of now the client is not required to reveal his/her identity in order to use the product.

CHAPTER-5

HIGH-LEVEL DESIGN

The overall design of the semantic detection tool is shown in Fig6. The user of the application has access to the User Interface allowing him to enter a python function snippet for its semantic detection.

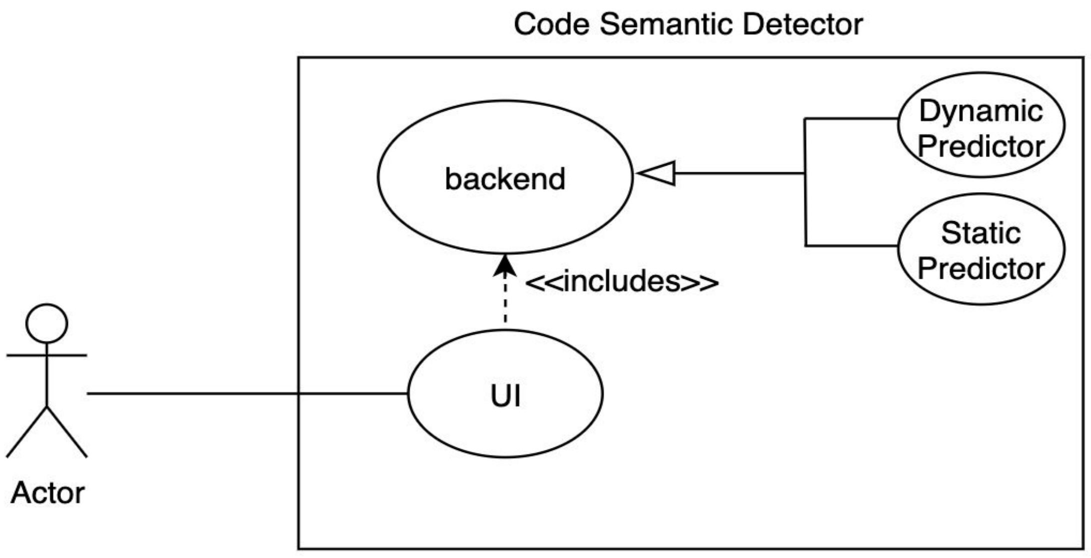


Fig8. Representation of the High Level Design

The interface relies on the backend to process the input function given by the user and provide the most suitable label. The input function is vectorized using Static and Dynamic analysis techniques. The label for the vectorized function is then computed based on the weighted average of the predictions made by the static and dynamic predictor.

CHAPTER-6

SYSTEM DESIGN

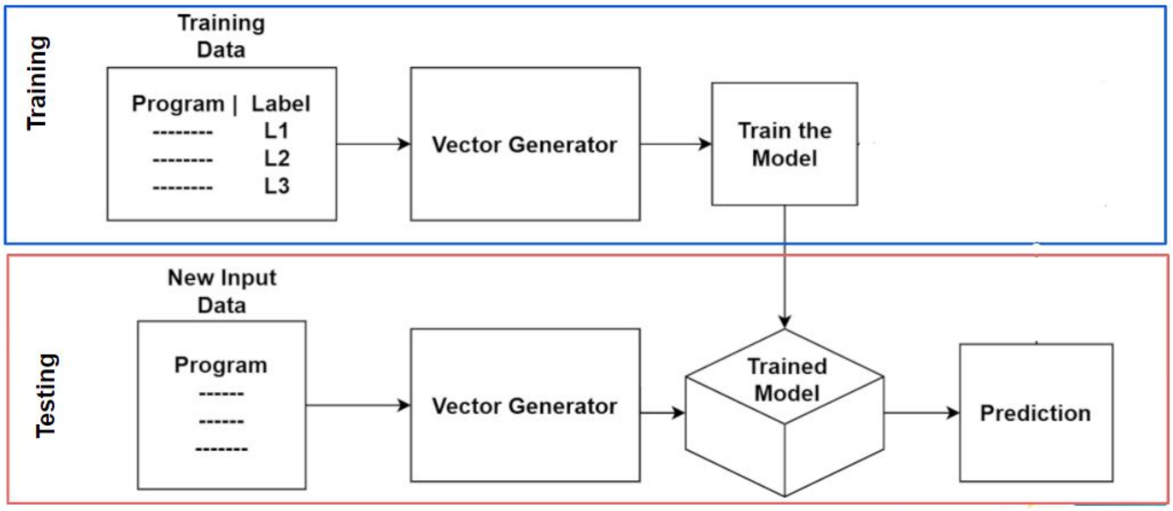


Fig9. Representation of System Design

The overview of our project is represented by Fig9.

In the training phase, the training dataset having the respective labels for the programs is passed to the vector generator which converts the programs to vectors. These generated vectors are used to train the model.

In the testing phase, a new program is passed to the vector generator which converts the program to a vector. It is then passed to the trained model which gives the required prediction.

CHAPTER-7

IMPLEMENTATION

7.1 DATASET

Dataset is composed of around 500 Python programs using Object Oriented paradigm spanning across different high level data structures or algorithms. The data structures or algorithms and the corresponding labels are as follows:

1. Array:
    a. SORT
    b. ROTATE_ARRAY

2. Math:
    a. FIB
    b. PALINDROME
    c. POW
    d. UGLY_NUM
    e. SIEVE
    f. SQRT
    g. REVERSE_INTEGER
    h. CATALAN_NUMBER

3. String:
    a. ANAGRAM
    b. STR_SEARCH

4. Linked List:
    a. CYCLE_LL
    b. REVERSE_LL

5. Tree:
    a. INORDER_TRAVERSAL

   b. VALID_BST
   c. LEVELORDER_TRAVERSAL
   d. HEIGHT_BT
   e. MAX_PATH_SUM_BT

6. Graph:
   a. COUNT_ISLANDS
   b. CYCLE_GRAPH

7. Dynamic Programming:
   a. JUMP_GAME
   b. COIN_CHANGE

8. Backtracking:
   a. SUDOKU
   b. RESTORE_IP

7.1.1 Dataset Generation

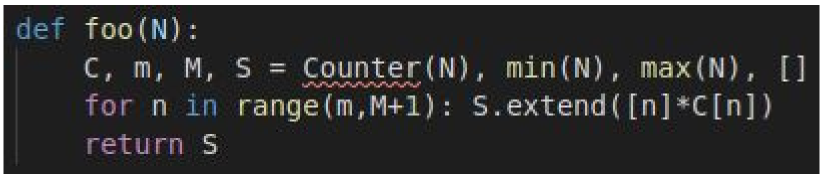
Dataset was generated by scrapping code snippets from top upvoted leetcode discussion threads. This thus not only gives model an exposure to different ways in which the problem can be approached but also the optimized way(in terms of time and space complexity) in which that problem can be solved.

7.1.2 Dataset Cleaning

Dataset cleaning involved following steps:

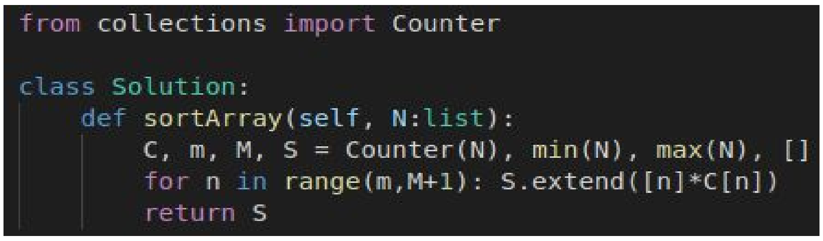
1. Adding the required imports
2. Bringing in standard format to make it executable which includes:
   a. Making it follow Object Oriented Paradigm
   b. Adding Parameter and Return data types

3. Writing required adapters for some particular labels to match the interface supported by the model
4. Running the code against unit test cases



```python
def foo(N):
    C, m, M, S = Counter(N), min(N), max(N), []
    for n in range(m,M+1): S.extend([n]*C[n])
    return S
```

Fig10(a).Before Data Cleaning



```python
from collections import Counter

class Solution:
    def sortArray(self, N:list):
        C, m, M, S = Counter(N), min(N), max(N), []
        for n in range(m,M+1): S.extend([n]*C[n])
        return S
```

Fig10(b).After Data Cleaning

Fig10. Transformation of code as part of cleaning

7.2 VECTOR GENERATION USING DYNAMIC ANALYSIS

7.2.1 Architecture

The overall architecture of the vector generator using dynamic analysis is shown in Fig10. It consists of 4 major parts. The input has to be an executable python function. The output is a vector of fixed length which can be used for other tasks like classification.

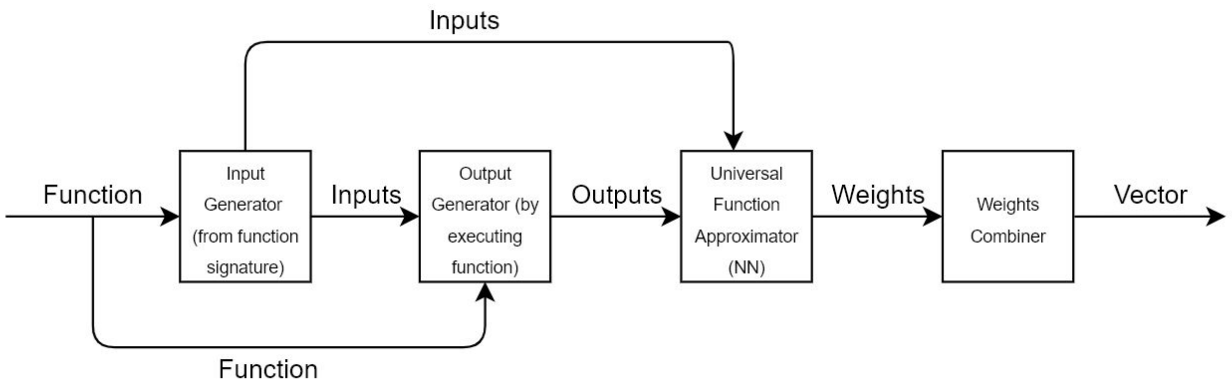
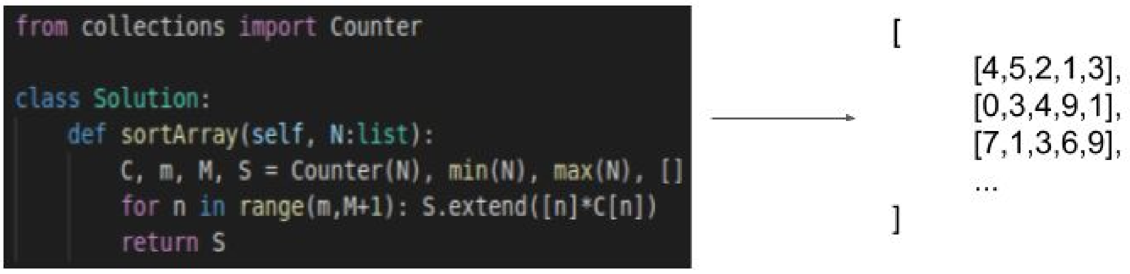

Inputs
Function
Input Generator (from function signature)
Inputs
Output Generator (by executing function)
Outputs
Universal Function Approximator (NN)
Weights
Weights Combiner
Vector
Function

Fig11. Architecture of Vector Generator using Dynamic Analysis

7.2.2 Input Generator and Output Generator

In order to approximate the Universal Function Approximator (neural network), inputs and outputs of the function form the dataset. Using this dataset, the neural network is trained. The input to the function depends on function signature. Since python is a dynamically typed language, type is a runtime mechanism. So just by looking at the function, the type can't be known. So extra type information has to be added. Python 3 introduced type annotations. This feature supports indicating the type of the variables in functions.

Based on the number of arguments and type of the arguments, random numbers are generated. Some functions have pre conditions for arguments. These arguments can only hold a restricted range of values. This is supported by passing a python dictionary as default argument for that argument. Default Precondition when no pre condition is specified is given below.



```
from collections import Counter

class Solution:
    def sortArray(self, N:list):
        C, m, M, S = Counter(N), min(N), max(N), []
        for n in range(m,M+1): S.extend([n]*C[n])
        return S
```

```
[
        [4,5,2,1,3],
        [0,3,4,9,1],
        [7,1,3,6,9],
        ...
]
```

Fig12. Input Generator

```
{"start":0,"end":11,"len_list":8,"upper_count":3,"lower_count":3,"digits_count":3,"special_count":3, "generator": lambda : random.random}
```

In the above precondition/config dictionary,

1. start and end refer to the range of values the random numbers can take
2. len_list refers to length of lists, tuples, sets in python which have to be generated
3. upper_count, lower_count, digits_count ,special_count refer to the number of uppercase alphabets, lower case alphabets, digits and special characters which must be generated in case of strings.
4. generator specifies a custom random data generator for an argument. It is a callable which takes no arguments. It can also be a string which has a function inside. The function must not take any arguments.

After generating the inputs, the corresponding outputs are generated by executing the function.

7.2.3 Universal Function Approximator and Weights Combiner

The heart of the Vector Generator using dynamic analysis is the Universal Function Approximator. This is a neural network which is generated dynamically at runtime using the dimensions of the inputs from the Input Generator. There are 2 hidden layers which consist of 6 nodes each. The output layer is also dynamically generated since the dimension of the output depends on the return type of the function.

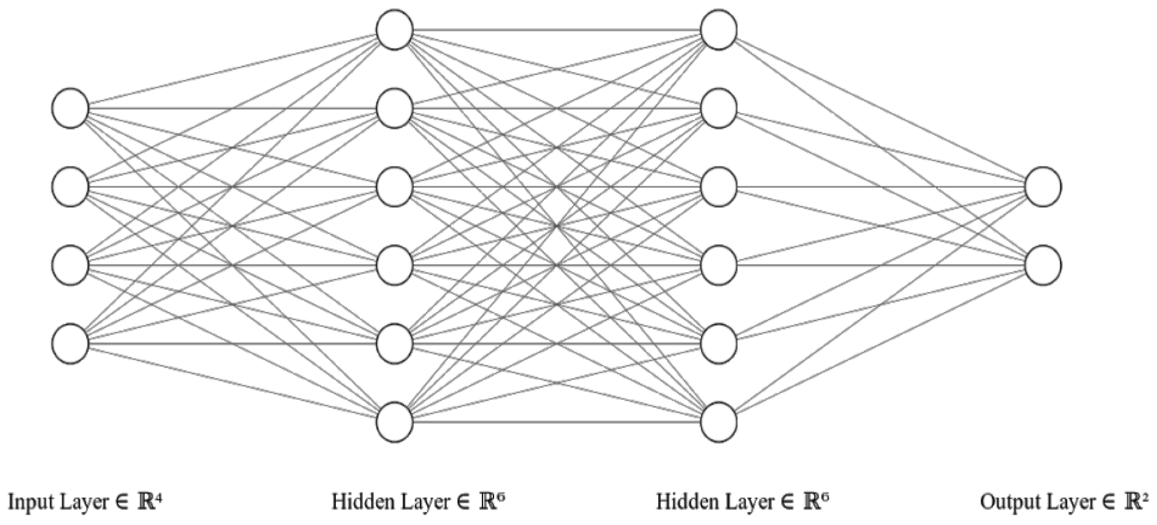


Fig13. Neural Network Architecture

Using the inputs and outputs generated previously by the input and output generators respectively, the neural network is trained to learn the function. After training, the neural network weights represent the function. In order to generate fixed length vectors, the weights from the hidden layers are combined to create a fixed length vector.

7.3 VECTOR GENERATION USING STATIC ANALYSIS

7.3.1 Architecture

The overall architecture of the vector generator using static analysis is shown in Fig 11. It consists of two significant parts, i.e. Code Structure Capturing unit and Sequence to Vector Generation Unit. The input has to be an executable python function. The function snippet is passed to Byte Code Generator and the Abstract Syntax Tree Generator to capture the context of the code. The intermediate representation received from the respective embeddings is passed to the Vector Generation Unit to convert the embedding sequence to a vector of fixed length.

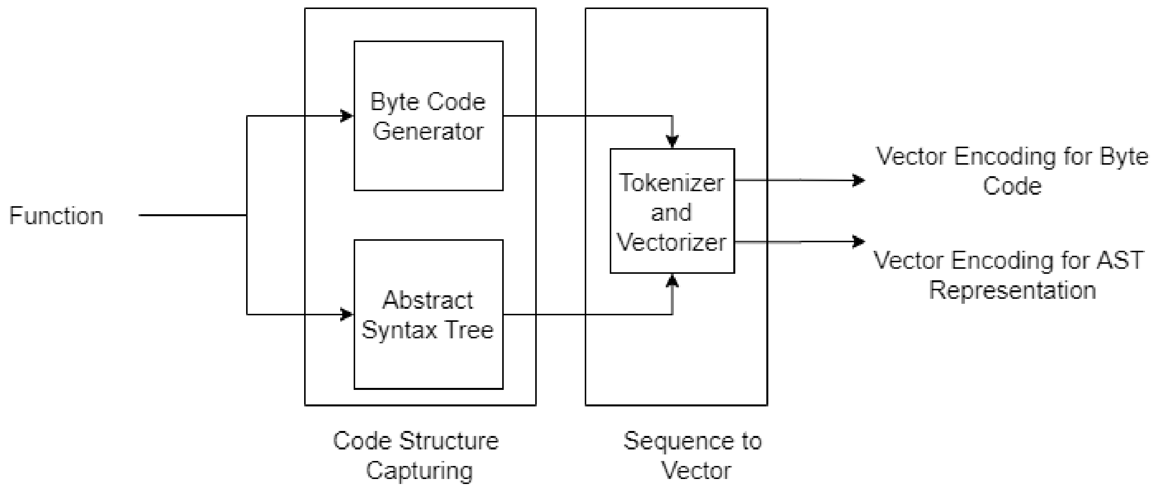


Fig14. Architecture for Vector Generator using Static Analysis

7.3.2 Byte Code Generation

Python, like many interpreted languages, compiles source code to a set of low-level instructions. This representation, called bytecode, is still not machine code and is platform-independent.

The bytecode can be very useful in semantic detection as it removes syntactic references and gives a common representation.

For example, the below program in Fig 15, is converted to its respective bytecode given in Fig 16.

```
from collections import Counter
import numba
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        return Counter(s) == Counter(t)
```

Fig15. Sample Program

```
Disassembly of <code object isAnagram at 0x7f1508df63a0, file "<ast>", line 4>:
  5           0 LOAD_GLOBAL              0 (Counter)
              2 LOAD_FAST                1 (s)
              4 CALL_FUNCTION            1
              6 LOAD_GLOBAL              0 (Counter)
              8 LOAD_FAST                2 (t)
             10 CALL_FUNCTION            1
             12 COMPARE_OP               2 (==)
             14 RETURN_VALUE
None
```

Fig16. Bytecode generated for sample program

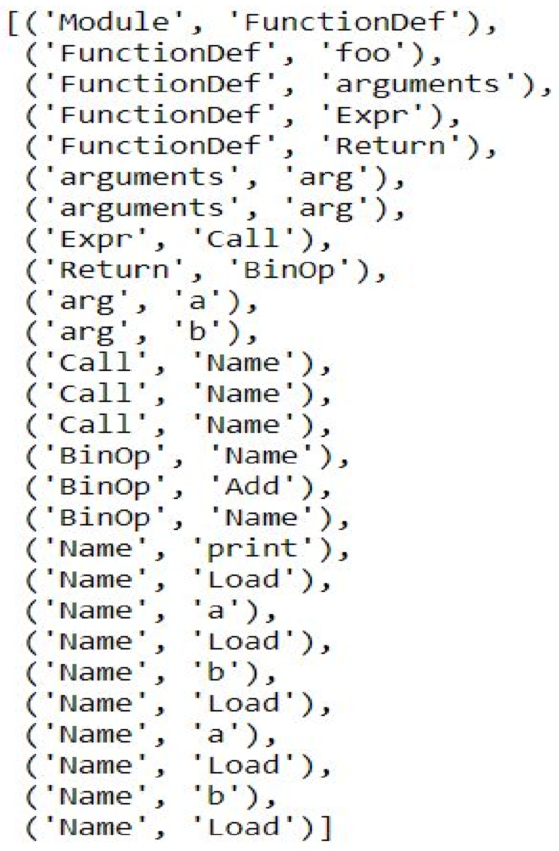
7.3.3 AST Generation

The abstract syntactic structure of the given source code for a particular programming language can be represented in the form a tree using an abstract syntax tree (AST). Every node of the tree represents a construct (such as variable declaration, conditional statements, function call) occurring in the function snippet.

It serves as an intermediate representation of the program used across multiple stages in the compiler, and significantly impacting the final output of the code compilation. The given code snippet can be checked for correct usage of the elements of the program and the language. The correctness of the program can be verified by completely traversing the tree. For example, take a sample code as:

```
def foo(a,b):
     print(a,b)
     return a+b
```

Fig1. shows the AST pairs of the generated abstract tree for the given program snippet. These AST pairs represent the nodes in the tree depicting each variable declaration, expression evaluation and function calls. This is helpful in understanding the structure of the code and identifying the general context of flow of statements



```
[('Module', 'FunctionDef'),
 ('FunctionDef', 'foo'),
 ('FunctionDef', 'arguments'),
 ('FunctionDef', 'Expr'),
 ('FunctionDef', 'Return'),
 ('arguments', 'arg'),
 ('arguments', 'arg'),
 ('Expr', 'Call'),
 ('Return', 'BinOp'),
 ('arg', 'a'),
 ('arg', 'b'),
 ('Call', 'Name'),
 ('Call', 'Name'),
 ('Call', 'Name'),
 ('BinOp', 'Name'),
 ('BinOp', 'Add'),
 ('BinOp', 'Name'),
 ('Name', 'print'),
 ('Name', 'Load'),
 ('Name', 'a'),
 ('Name', 'Load'),
 ('Name', 'b'),
 ('Name', 'Load'),
 ('Name', 'a'),
 ('Name', 'Load'),
 ('Name', 'b'),
 ('Name', 'Load')]
```

Fig17. The AST pairs of the tree generated based on sample program

7.3.4 Sequence To Vector

The sequence in which the data appears plays a crucial role in determining its context and semantic. The embeddings or representation obtained through byte code generator and ast generator cannot be fed directly to a classifier. Text data must be encoded as numbers to be used as input or output for All machine learning and deep learning models require input text data to be encoded as numbers and generate corresponding number output.

The representation is tokenized based on the attributes word_count and word_index by the Tokenizer and then fed to the vectorizer. The Tokenizer creates one vector per document i.e. the program snippet provided as input.

The total size of the vocabulary can be determined from the length of the vectors. The vector length is chosen wisely to prevent loss of information and ease the procedure of the classifier training.

7.4 CLASSIFICATION

The classification for dynamic analysis comprises the k-nearest neighbors (KNN) algorithm which takes the generated vectors as the input and classifies any new input based on the distance between the vectors. The output of the dynamic model is a set of predictions with their corresponding probabilities. The advantage of using this method is that it requires only a single program to learn.

In the static model, the classification is done using a combination of naive bayes classifier and the KNN algorithm which takes the vector encodings as input. Each model separately provides the label for the input function snippet. Both models have varied performance for the given labels. It was observed that some labels predicted well by KNN performed poorly with Naive Bayes model and vice versa. Hence, the probability of correctness of a particular label predicted by each model is different. This label wise probability for each model is pre-calculated using the testing data. The model with higher probability for the respective label contributes to the final label prediction.

The advantage of using a combination of two different classifiers allows the model to learn better and reduce the miss count. The output of the static model is the required prediction.

It requires a sufficient number of different implementations of each label to learn the representation well. The label prediction in the static model is independent of the input data type and format of the function signature unlike dynamic predictor.

CHAPTER-8

RESULT

The data structure or algorithm wise accuracies for both the models are summarized in the below table:

Data Structure/Algorithm	**Static Model**	**Dynamic Model**
Array	51.42%	100%
Math	71.1%	99.39%
String	85.71%	95.23%
Linked List	92.8%	100%
Tree	88.571%	100%
Graph	92.857%	95.23%
Dynamic Programming	90%	100%
Backtracking	91.66%	100%
Overall	**80.86%**	**98.97%**

Table1. Table displaying DS/Algorithm wise accuracy for both models

The attribute based comparison of the models is displayed below:

Attribute	**Static Model**	**Dynamic Model**
Limitations	High Structural similarity in the code snippet leads to incorrect prediction	For complex functions, pre-conditions may have to be specified.
Highlights	Independent of argument data type	Requires just one function to learn
Improvements Needed	Accuracy	Time taken to predict

Table2. Table displaying attribute-based comparison of models

CHAPTER-9

CONCLUSION FROM PHASE 1

As part Capstone Phase 1, the given unknown code snippet is labelled. A labelled dataset of programs is web scraped from common online coding platforms, covering the most commonly used programming algorithms. The classification of the code label uses a combined approach of Static and Dynamic analysis.

The static analyzer predicts based on the basis of code syntax-structure and code-flow. The dynamic analyzer approximates a given function based on the it's input and output.

The developed tool is able to predict labels for the code snippets in the dataset with an average accuracy of 90%.

CHAPTER-10

PLAN FOR CAPSTONE PHASE 2

As a part of Capstone phase 2, we plan to optimize code snippets by suggesting more optimal algorithms at the design level. Also in order to compare algorithms a composite metric will be developed to compare overall performance of algorithms . Once the high level logic of the unoptimized code snippet is known, based on the developed metric, a more optimal algorithm is suggested.

REFERENCES

[1] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), 1-29.

[2] Wehr, D., Fede, H., Pence, E., Zhang, B., Ferreira, G., Walczyk, J., & Hughes, J. (2019). Learning Semantic Vector Representations of Source Code via a Siamese Neural Network. arXiv preprint arXiv:1904.11968.

[3] Wang, T., Wang, K., Su, X., & Ma, P. (2014). Detection of semantically similar code. Frontiers of Computer Science, 8(6), 996-1011.

[4] Ye, F., Zhou, S., Venkat, A., Marucs, R., Tatbul, N., Tithi, J. J., ... & Sarkar, V. (2020). MISIM: An End-to-End Neural Code Similarity System. arXiv preprint arXiv:2006.05265.

[5] C. K. Roy, M. F. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (keynote paper),” in 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). IEEE, 2014, pp. 18–33.

APPENDIX A USER MANUAL

Steps to Install:

1. Install docker and docker-compose
2. Clone the repo
3. `sudo docker-compose up --build`

Steps to Train/Learn new Functions via UI :

1. Type `localhost:8090` in browser to access UI
2. Select model
3. Type code/upload file. The function name will be taken as label
4. Click on Learn button

Steps to Test/Predict Functions via UI:

1. Type `localhost:8090` in browser to access UI
2. Select model
3. Type code/upload file.
4. Click on Test Button

Steps to Learn new Functions via Library function call:

1. `import model1.dataset_gen as g1`
2. `g1.add_to_dataset(function,lang)`

Steps to Test/Predict Functions via Library function call:

1. `import model1.predict as mp1`
2. `m = mp1.predict(function,language)`

To determine accuracy label wise for dynamic model:

1. `cd backend`
2. Add the labels for which accuracy has to be determined in the labels list in dataset_server.py. Available labels can be referred from mapping.json

3. Run `python dataset_server.py` and the results can be seen in results.txt

Note :

1. For functions with special pre conditions like strings having only digits or only special characters or functions having types which are not supported by dynamic analysis model,the preconditions have to be explicitly specified. For ex : In case of a function validating IP address string, the string must not contain upper case, lowercase and special characters. So in that case, the function signature will be

   ```
   def IP_VAL(s:str={'upper_count':0,'lower_count':0, 'special_count':0,
   'digits_count': 8}): pass
   ```

   So here by making upper case, lowercase and special characters count as 0 and digits count as 8, the precondition becomes a string having 8 digits with no upper case, lowercase and special characters.

2. Default Precondition when no precondition is specified is

   ```
   {"start":0,"end":11,"len_list":8,"upper_count":3,"lower_count":3,"digi
   ts_count":3,"special_count":3 ,"generator": lambda : random.random}
   ```

3. start and end specify the range of random numbers to be generated. Ex : if start=1, end=1000, then numbers in the range [1,1000] are generated.
4. generator specifies a custom random data generator for an argument. It is a callable which takes no arguments. It can also be a string which has a function inside. The function must not take any arguments.
5. len_list is used to set the length of lists in case of list arguments.