DEEPCODE phase 1: Defining optimised compiled language for Code-set Summarization

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Miquel Puig i Mena

In partial fulfilment

of the requirements for the degree in

TELECOMMUNICATIONS TECHNOLOGIES AND

SERVICES ENGINEERING

Advisor: Jose Antonio Lazaro Villa

Barcelona, January 2020

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

telecom BCN

Abstract

Yet being immersed in the globalisation era, we can still perceive isolation and rivalry between software developing teams that share ambitions.

Regardless of the nature of the research, a common denominator can be spotted: they all try to get the best existing ideas on the field and apply some new improvement/s that makes their own implementation the best in a specific set of “tangible” aspects (i.e. performance, efficiency).

Nonetheless, DeepCode proposes a cooperation between parties in order to achieve greater results. DeepCode is a tool that, by processing a data-set formed by source codes representations performing a generic *task A*, learns how to implement *task A* and outperform every individual from studied data-set. Given the complexity of DeepCode, two stages are defined: *Phase 2*, that, with artificial intelligence techniques, learns which ideas in the data-set imply good code performance and *Phase 1*, which studies and builds customised code representation in order to provide *phase 2* an optimised and workable data-set.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

telecom BCN

Resum

Tot i viure en l'era de la globalització, encara podem percebre cert aïllament i rivalitat entre equips de desenvolupament de software què comparteixen objectius.

Per a una reserca genèrica, un comú denominador pot ser detectat: tots intenten filtrar les millors idees del camp en qüestió i aplicar-ne noves millores què fagin de la seva implementació la millor del mercat en un seguit d'aspectes «tangibles» (i.e. rendiment, eficiència).

No obstant, DeepCode proposa una cooperació entre rivals per aconseguir resultats més trascendents. DeepCode és una eina què, processant un data-set format per codis fonts què executen certa *tasca A*, aprèn com implementar esmentada *tasca A* i aconseguir resultats superiors a qualsevol individu en el data-set. Donada la complexitat i natura de DeepCode, dugues fases són definides: *Phase 2*, què, amb capes d'AI, aprèn quines idees en el data-set comporten bons resultats; i *Phase 1*, què estudia i munta una representació original del codi font amb l'objectiu de proveïr la *Phase 2* un data-set procesable i òptim.

Resumen

Aún viviendo en la era de la globalización, podemos percibir cierto aislamiento y rivalidad entre equipos de desarrollo de software que comparten objetivos.

Para un estudio genérico, un común denominador puede ser detectado: todos intentan filtrar las mejores ideas del campo en cuestión i aplicarle nuevas mejoras que hagan de su implementación la mejor del mercado en un seguido de aspectos «tangibles» (i.e. rendimiento, eficiéncia).

No obstante, DeepCode propone una cooperación entre rivales para lograr resultados más trascendentes. DeepCode es una herramienta que, procesando un data-set compuesto por códigos fuente que ejecutan cierta *tarea A*, aprende como implementar dicha *tarea A* y consigue resultados superiores a cualquier individuo en el data-set. Dada la complejidad y naturaleza de DeepCode, dos fases son definidas: *Phase 2*, que, con capas de inteligéncia artificial, aprende que ideas en el data-set conllevan buenos resultados; y *Phase 1*, que estudia i monta una representación original del código fuente con el objetivo de proveer a la *Phase 2* un data-set procesable y óptimo.

Revision history and approval record

Revision	Date	Purpose
0	01/12/2019	Document creation
1	dd/mm/yyyy	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Miquel Puig i Mena	miquel.puig.mena@estudiant.upc.edu
Jose Antonio Lazaro Villa	jose.lazaro@tsc.upc.edu

Written by:		Reviewed and approved by:	
Date	01/12/2019	Date	dd/mm/yyyy
Name	Miquel Puig i Mena	Name	Zzzzzzz Wwwwwwww
Position	Project Author	Position	Project Supervisor

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecom BCN

Table of contents

Abstract....2

Resum....3

Resumen....4

Revision history and approval record....5

Table of contents....6

List of Figures....8

List of Tables....9

1. Introduction....10

  1.1. Project scope....11

  1.2. Phase 1: Defining optimal compiled language for Code-set Summarization....12

    1.2.1. Statement of purpose....13

    1.2.2. Requirements and specifications....14

    1.2.3. Work plan and milestones....15

  1.3. Deviations and incidencies....16

2. State of the art of the technology used or applied in this thesis....17

3. Methodology....18

  3.1. *Architecture*....18

  3.2. *Building source code data-set*....19

    3.2.1. *Gathering the data-set with web scraping tools*....19

    3.2.2. *Homogenising a data-set under the same distribution & labeling the data-set* 20

  3.3. *Building custom AST data-set from source code data-set*....21

    3.3.1. Gathering a the Abstract Syntax Tree....22

    3.3.2. Traversing the AST....22

    3.3.3. Filtering nodes....23

      3.3.3.1. Disallowing aliases....23

3.3.3.2. Minimizing number of entities in AST.....23
3.3.3.3. Minimizing number of nodes in AST.....24
3.3.4. Enriching the AST.....24
3.3.4.1. Discriminating method calls.....24
3.3.5. Representing the customised AST.....24
4. Results.....25
4.1. DeepCode *Phase 1* Evaluation.....25
4.1.1. Custom & simple test code.....25
4.1.2. Custom & complex test code.....26
4.1.3. Transforming a 10k files data-set.....26
4.2. Evolution of a singular source code sample.....27
5. Budget.....29
6. Conclusions and future development.....30
Appendices.....32
1. DeepCode Phase1: Compilers approach.....32
2. Grouping strategy from Python's original entities.....39
3. Web scraping HackerRank's "Capitalize!" challenge.....42
4. Homogenising and labeling source code data-set.....46
4.1. Implementation: homogenising data-set.....46
4.2. Implementation: homogenising, profiling, and labeling source code.....47
5. Implementation evaluation tool.....49
Glossary.....52

List of Figures

Illustration 1: Architectural schema representing DeepCode's full scope. Notice the contribution provided by each phase of DeepCode to the full project. Input of phase 1 is a data-set formed by text source code samples and it's output is an optimised data-set....12

Illustration 2: GANTT diagram depiciting mimlestones followed while building DeepCode phase 1: Defining optimal compiled language for Code-set Summarization....................16

Illustration 3: Left screen contains a view of "Capitalize!" challenge explanation at hackerrank platform. Right screen depicts the set of solutions DeepCode is interested in. 19

Illustration 4: Representation of test code (Code 1) as AST..21

Illustration 5: Representation of test code(Code 1) as cAST..21

List of Tables

Index of Tables

Table 1: Milestones of DeepCode phase 1: Defining optimal compiled language for Code-set Summarization.........15

Table 2: Project cost.........29

Index of Graphs

Graph 1: Accumulation of number of entities seen in each sample in the data-set.........26

Graph 2: Accumulation of number of nodes seen in each sample in the data-set...........27

Index of Codes

Code 1: Snippet of code chosen to depict a generous example that includes: Disallowing Aliases (Section 3.3.3.1), Minimizing number of Entities (Section 3.3.3.2), Minimizing number of Nodes (Section 3.3.3.3), Discriminating method Calls (Section 3.3.4.1), and Representing the cAST (Section 3.3.5).........22

Code 2: Random code peeked from gathered data-set under the same distribution.This sample achieves a maximum score at HackerRank platform.........27

Code 3: AST build by Python's core compiler from source code at Code 2.....................28

Code 4: cAST build by Python's core compiler from source code at Code 2...................28

Index of Evaluations

Evaluation 1: Analysis extracted from Deepcode phase 1 evaluator logs. Notice that AST is formed by a total of 22 nodes with 13 different entities while cAST cuts it up to 15 nodes representation including only 9 different entities.........25

Evaluation 2: Analysis extracted from Deepcode phase 1 evaluator logs. Notice that AST is formed by a total of 1151 nodes with 55 different entities while cAST cuts it up to 865 nodes representation including only 51 different entities.........26

1. Introduction

This project is carried out both at the Computer Science and Language Technology departments in Lunds Tekniska Högskola (LTH) in colaboration with Universitat politècnica de Catalunya (UPC). This project is based on the previous theoretical courses Language Technology (EDAN65, LTH) and Compilers (EDAN20, LTH) from same LTH. Combination of knowledge gained in listed courses, leads to a solid background to use as a base point in order to complete Project in computer science (EDAN70, LTH) and author's bachelor thesis at UPC.

After an intense week coursing a Cybersecurity program (NeCS PhD School) mainly focused in Deep Learning applications for cybersecurity, my colleague, Marc Roig Lama, and I came up with an original idea, DeepCode.

While waiting at Mezzocorona's train station, we were talking about how we face a new project implementation when we realized that a systematic process was always followed, regardless of project's background. Research similar implementations within the community, discriminate best implemented ideas, build skeleton from seen concepts and, finally, fit it to own project's demands. Why not bulk a great amount of data and let Artificial Intelligence (AI) make the process of understanding where the good ideas are? Why not let AI build an original sample summarizing all individuals in the data-set?

It is my personal job to develop the system from scratch by myself but originality custody is shared by Marc Roig and me. To guide and help I'll be backed by Görel Hedin, from LTH's Computer Science department, Pierre Nugues, from LTH's Language Processing department, and José Antonio Lázaro, from Signal Theory and Communications department at UPC.

1.1. Project scope

Coders, Group of coders, Departments, Companies or Group of companies compete every day to get the most efficient algorithm for a specific problem. Regardless of the nature of the research, a common denominator can be spotted: they all try to get the best existing ideas on the field and apply some new improvement/s that makes their own implementation the best in a specific set of "tangible" aspects (performance, usability, efficiency, etc.). In that sense, enormous amount of time and effort is spent by each researcher analyzing and mastering the field. Additionally, told hypothetical researcher won't have any kind of assurance that a competitive approach has been discarded during the process of mastering.

In this thesis, DeepCode *phase 1*: Defining optimal compiled language for Code-set Summarization is presented. Naturally, this thesis is a partition of a bigger project: *DeepCode*.

DeepCode is an innovative tool that will try to auto-generate unique and original code. *DeepCode* system aims to generate code that solves a generic task A by learning, from a large set of different codes, data-set, how to solve task A. Notice that each element in the data-set tries, as well, to solve task A. Therefore, the concept **data-set under same distribution**, references a group of different code implementations that perform the exact same job such as compressing a file, processing a signal or solving mathematical paradigms. Furthermore, *DeepCode*'s potential innovative objective is the ability to overcome best test-performance reached by any individual from the data-set it's been learning from.

Given the complexity and nature of DeepCode, two stages are defined: *phase 2*, that, with Artificial Intelligence (AI) techniques, learns which ideas in the data-set imply good code performance and *phase 1*, which studies and builds customised code representation in order to provide *phase 2* an optimal and workable data-set. This thesis is exclusively focused in first task of the process while second task is proposed as future work. Consequently, the objective of this thesis is to build a manageable data-set under the same distribution.

1.2. Phase 1: Defining optimal compiled language for Code-set Summarization

Building and normalizing a data-set can be very tedious and time consuming. However, all the results achieved by subsequent layers will directly depend on the quality of that same data-set.

This subsection deeply describes *phase 1* of *DeepCode* but not *phase 2*. Nevertheless, *phase 2* is relevant for this section since it defines what kind of transformation has to be applied to the source code data-set in order to accomplish future intelligent decisions (See architectural model of *DeepCode* in Illustration 1).

In that sense, this section introduces an original code representation to be utilised afterwards by artificial intelligence processes.

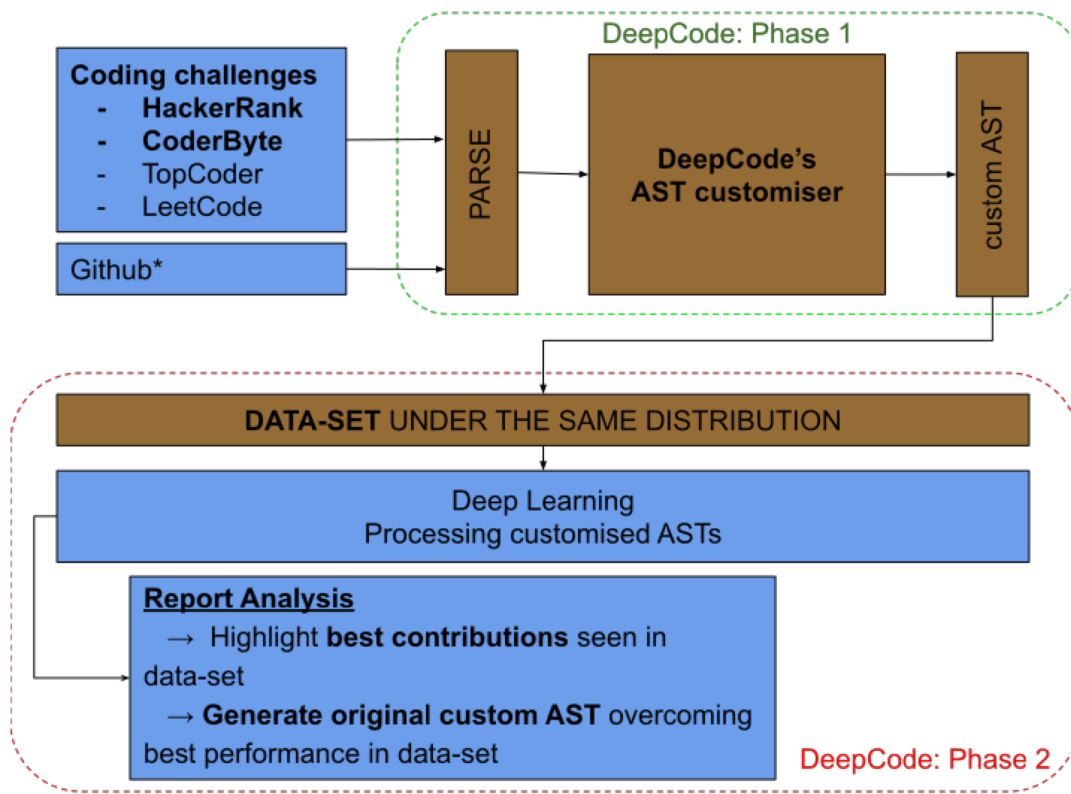


Illustration 1: Architectural schema representing DeepCode's full scope. Notice the contribution provided by each phase of DeepCode to the full project. Input of phase 1 is a data-set formed by text source code samples and it's output is an optimised data-set.

1.2.1. Statement of purpose

Most generally, source codes, irrespective of it's original language, can be represented in lower abstractions levels. From machine code [1], unintelligible from human perspective, until low level code such as Assembly. For *DeepCode*'s purposes, a middle level representation of source code must be designed. Such intermediate depiction of the code will permit any data-set under the same distribution to be homogenized, allowing, this way, to make fair comparisons among code samples under the same distribution.

Since source code can be ambiguous from a machine's point of view (i.e., In Java code, '+' symbol can be a sign of mathematical summation and a key word that concatenates Strings), a formal analysis has to be performed against the source code in text format by a compiler.

Simplistically, a compiler for a specific language will find language's accepted corpus, this action is referenced as tokeninzing. This way, every single word in the source code is categorised as a functionallity of the language. Notice that if some part of the

---

1 Machine code, also called machine language, is a computer language that is directly understandable by a computer's Central Processing Unit (CPU), and it is the language into which all programs must be converted before they can be run. Each CPU type has its own machine language, although they are basically fairly similar.[Pro06]

code could not be categorised, an error will be rised. Furthermore, an Abstract Syntax Tree[JON03] (AST) format is build by relating found tokens in a tree manner structure.

Aforementioned analysis follows a methodology composed by following steps:

- Lexical Analyzer *
- Syntactic Analyzer *
- Semantic Analyser *
- Intermediate Code Generation *
- Optimizer
- Target code generator

* This set of operations were deeply studied during LTH's EDAN65 compiler course. Without this base knowledge it would be impossible to accomplish thesis' objectives.

Concretely, at semantic analyzer layer, the system achieves a non-ambiguous representation of a source code, AST. Generally, an AST abstraction format is fed to subsequent compiler's layers such as optimization of code or Assembly code generation. For that reason, this work takes the AST representation as it's baseline. Further designed momdels will be build on top of the AST.

By default, in Python packages, an in-build Python compiler written in C Programming language is included. Additionally, Python packages include in-build front-end libraries enabling, from Python code itself, to access compiler's internals. Among other modules, *ast* front-end can be found, which converts a Python source code into it's AST representation. From ast object, a rich AST context is accessible, providing, this way, a great entry point to customise a generic AST.

By *DeepCode*'s means, tree representation of the code (AST) is very profitable. Due to time consuming processes in *phase 2*, every redundant information in the AST must be eliminated. In other words, internal AST generated by the compiler includes neglectable nodes from *DeepCode*'s perspective. In that sense, AST generated by the compiler must be simplified as much as possible without loosing valuable information. Moreover, additional important information can be appended to the original AST in order to highlight assets in the code that are not conceived by the AST. Hence, a compression followed by an enrichment strategy will satisfy sought middle level representation of source code by this thesis.

Summarizing, *DeepCode phase 1* suggests and provides a compressed and enriched code representation of the AST. Furthermore, a python tool is created which consists in gathering a data-set under the same distribution and homogenises the data-set by applying the pre designed transformation to every sample in it.

1.2.2. Requirements and specifications

DeepCode phase 1 establishes a set of requirements and specifications for each sub-division of the project. Listed below, project requirements by section:

A) Code-set retrieval formed by samples of different coding languages:

I. List of accepted coding languages for the data-set.

- LLVM compiler for C language has been deeply studied but difficulties where encountered while developing applications on top.
- For simplicity's sake, only Python2.X and Python3.X files will be taken into account when building the data-set.

II. Build data-set of N samples from online open coding challenge platforrms.

- Extraction of data-set is performed via HackerRank[2] platfrom with the help of an original web parsing tool.
- Selenium v3.141.0 for python3.6 is utilised in order to build web parsing tool.

B) Test code performance:

I. Formatting samples in data-set in a manner that they all provide same functionallity.

- File text transformation with the help of Python's regex v2019.8.19 package.

II. Define set of tests to grade each individual sample inside the data-set.

- Automation of tests performed by custom cycle code in python3.6.

C) Code normalization:

I. Define optimal custom compiled form for future analytics processes.

- Usage of python's inner compiler. Concretely, python3.6 ast front-end.

II. Homogenize Code-set with custom source code representation.

- Transformation of source code implented in python3.6 and backed in following packages:
  i. json5 v0.8.5
  ii. argparse v1.1
  iii. logging v0.5.1.2

1.2.3. Work plan and milestones

The project is an original implementation, desgined and build from scratch. For that reason, biggest block of hours are computed under study and implementation tasks. Due to deviations during the project, knonwledge used in Task 0, Natural Language Technology, has not been directly applied in the project. However it has been incredibly important in order to decide final asset of the project.

2 HackerRank is a very well known online coding challenge platform. Find it at https://www.hackerrank.com.

Task #	Task name	Task type	Start date	End date	Hours
0	Natural Language Technology	Study	01/09/2019	25/11/2019	60
1	Compiler Research	Study	01/09/2019	25/11/2019	100
2	Define optimal compiler's output format for *Phase 2*	Design	25/10/2019	25/11/2019	40
3	Gathering the data-set	Implement	01/11/2019	25/11/2019	20
4	Testing and profiling the data-set	Implement	15/11/2019	30/11/2019	40
5	Customizing compiler	Implement	01/11/2019	31/12/2019	80
6	Applying customisation to data-set	Implement	25/12/2019	05/01/2020	40
7	Compiler Documentation	Writing	15/12/2019	15/01/2020	70

Table 1: Milestones of DeepCode phase 1: Defining optimal compiled language for Code-set Summarization.

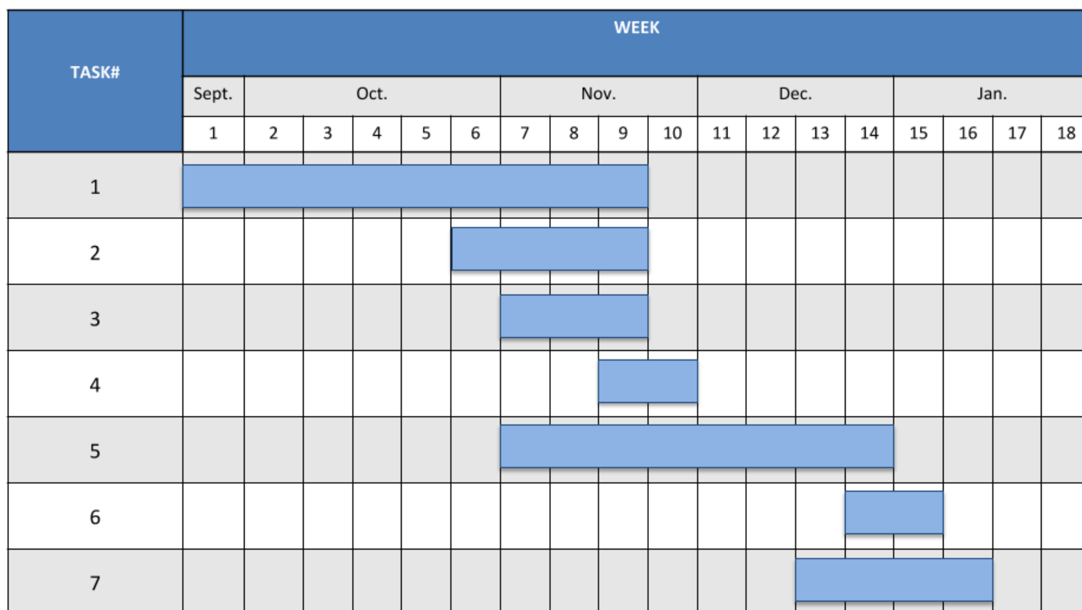


Illustration 2: GANTT diagram depiciting mimlestones followed while building DeepCode phase 1: Defining optimal compiled language for Code-set Summarization.

1.3. Deviations and incidencies

My first intention was to build from scratch *DeepCode*. Luckily, by advise of Professor Pierre Nugues from LTH, I decided to present a theoretical approach of *DeepCode phase 2* while optimising the data-set presentation and truly understanding

the potential of posessing a high quality set of data. Project proposal and workplan was agreed according to aforementioned status.

The first introduction to EDAN65 compilers course at LTH, was useful in order to recognise the considerations I had to take in order to achieve a good solution. However, I discovered that the complexity was higher than expected and not much previous work existed to rely on. In that sense, design of customised representation of the AST took more than expected and delayed following tasks.

EDAN70 project in computer science included in it's agenda a release implementation of the tool and an academic paper describing such tool that had to be delivered. Due to my unexperience in both, tagging a final release of an implementation with it's formalities and writing a paper, vast amount of time was consumed in this stages. Consequently, author decided to focus exclusevly in the process of building a data-set and provide a tool capable of gathering a data-set under the same distribution and optimise it from the point of view of future artificial intelligence layers.

Concluding, final work plan was deviated from author's first idea by only conceiving the formation of an optimised data-set. Hence, the author will propose second phase of *DeepCode* for future research.

2. State of the art of the technology used or applied in this thesis

Source code analysis has drastically evolved since AI's reborn. While code processing tools were exclusively based in complex and deterministic techniques, AI provides another dimension to existing tools. By gathering information from user experience, AI enables code processing tools to optimize itself on the fly. Currently, there's a big interest and investment from a well established community. Specifically, a big effort is invested by Integrated Development Environment (IDE) and code analysis applications such as bug finders or providing interesting and personalised hints while coding.

[APS16] provides a brief text summary of what an inputted code intends to do while [ABS14] provides a tool to learn which non-written conventions and patterns are followed by developers. Both studies have in common that they treat source code as simple text instead of utilising standardised representations such as ASTs or machine code.

Furthermore, finest results were achieved by studies utilising abstracted representations of source code. Code clone detection application [Whi+16] overcame state-of-the-art of the moment by extracting AST of it's source code data-set.

An original suggestion is found in [Moub+14], where they propose an encoded version of the AST based in [Moua+14], that converts tree structures in vectorised information. Great improvement on bug localisation was achieved by [Lia+19] by adopting the program representation proposed by [Moua+14]. Additionally, [Lia+19] included the concept of compressing the AST as much as possible in order to facilitate AI layers to complete their job with optimal results. In consequence, [Lia+19] is used as the inspiration to fulfil project's objectives.

Commonly, each programming language has it's own AST generator engine and, in many cases, more than one. This paper attempts to corroborate that an intelligent filter in current Python AST can be extremely beneficial for post processing techniques.

Low Level Virtual Machine[LA04] (LLVM) is a compiler framework that uses a build in intermediate human readable code language with slightly higher level abstractions than Assembly. Although LLVM code can be written from scratch, it's typical to produce it when compiling source code, as intermediate step for code optimization layers and/or optimal machine code generation.

Clang project provides a language front-end and tooling infrastructure for languages in the C language family for the LLVM project[UCa]. Clang is an open source project well established in the community, having up to 79k commits by the time this report is written. Although Clang is backed by a huge community and provides reliable services, this paper will choose python compiler[PDCc], a less complex compiler engine system that demands less time to familiarise with.

3. Methodology

Arrived to this point, the project has been introduced and contextualised. This work can be divided by independant modules that sequentially communicate with each other. Hence, this section describes in a technical manner how each functional block in the system is build.

3.1. *Architecture*

Within DeepCode's scope, DeepCode *phase1* commences once the text data-set is assembled (See architectural depiction at Illustration 1). For that reason, first stage involved would be how the data-set under the same distribution is gathered. From this point on, the reader should assume that the data-set formed by text source codes performing *task A* is accessible.

A data-set gathered from a code challenge platform is very useful but, some homogenisation processes need to be applied before continuing. The implementation has to be sure that every or most of the individuals in the data-set follow a similar schema. In other words, given two samples that are supposed to solve *task A* in a correct manner, they should be able to do it by being triggered with the same command. For that reason, second challenge for the project has been accomplishing an homogenised data-set under the same distribution.

Thirdly, each source code must be scored with a value from 0 to 1 where 1 is maximum score. To achieve so, this thesis analyses two parameters: score provided by the coding challenge platform and an average execution time for N executions of the sample. Notice that a unit test has to be defined for *task A* in order to extract an average execution time of the sample. Reached this stage, the homogenised data-set under the same distribution has been labeled.

Source code in text format has to be transformed to customised AST representation. Proposed tool by this report handles each element in the source code data-set individually. In this regard, the tool has been developed so it takes a single file

as input and applies a set of transformations so it can output an optimised version of Python's AST.

Python's inner compiler engine produces an AST that will be trust and used by subsequent modules. Such dependency on Python's compiler core can be assumed given the reliability provided by the existence of a vast community along Python itself using it massievly.

Parallel, python's engine is bound to report's tool in a way that AST context is shared and accessible. Thereafter, every node in retrieved AST is traversed following a visitor strategy in order to forge pursued custom AST (cAST). While visiting a node, DeepCode *phase 1* implementation decides which information is relevant from that node, if any at all, and persists it in the cAST. Notice that personalised tree carries a one to one connection between cAST nodes and their AST node pair. Concluding, fourth stage is the actual transformation of source code to cAST.

Finally, DeepCode *phase1* application grants to the user the functionality to store or output created cAST in diverse formats. This report contemplates JSON, encoded JSON[3] (eJSON), or pickle format (PyPickle) to be shown at console or saved in a file.

3.2. ***Building source code data-set***

By all means, most important step to keep the project rolling is to build a suitable data-set under the same distribution. Such task is not trivial and can be attacked from many points of view.

Nowadays enormous amount of implementations are hosted in Git based tools. Even though it could seem like a feasible source of codes, it complicates the scraping process since it's challenging to filter *task A* from a massive Git tool based like Github. In consequence, deeper study in the field was needed in order to find an online platform that groups numerous sample of codes by it's intrinsic objective. Happily, online coding challenge platfroms were encountered.

Online coding platforms consist in uploading a list of challenges with different difficulties that any subscribed user can solve and, afterwards, share it's solution. Additionally, most of the platforms allow non-premium users to see all the solutions uploaded by other people for a certain *task A*.

3\ Encoded JSON directly depends in final global DeepCode data-set. Depending in the different node entities sighted in data-set, the eJSON will cipher accordingly.

3.2.1. ***Gathering the data-set with web scraping tools***

Next step must be which platform will be scraped according to the strategy of gathering the data-set under the same distribution from an online code challenge platform. Several platforms are available but, because of author's proximity, www.hackerrank.com will be chosen. Notice that such platform doesn't provide higher quality than the rest. Nevertheless, they provide excellent services while being respected and used by a large community.

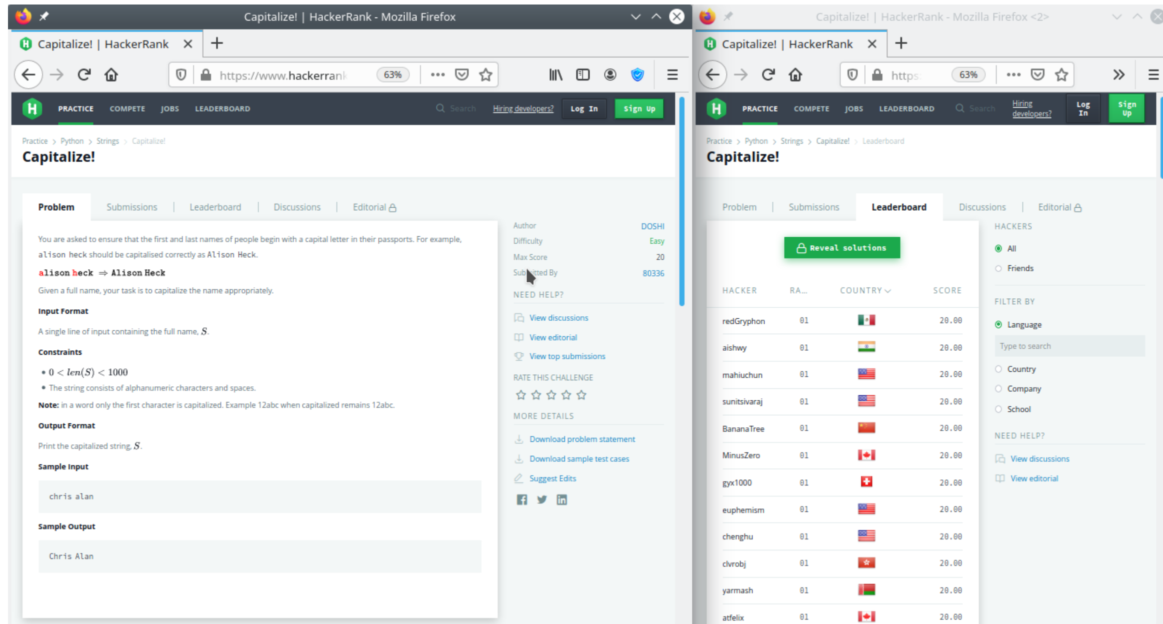


Illustration 3: Left screen contains a view of "Capitalize!" challenge explanation at hackerrank platform. Right screen depicts the set of solutions DeepCode is interested in.

A challenge within the challenge platform must be chosen. By thesis constraints, only Python samples will be considered. Hence, due to numerous Python solutions in the platform, "Capitalize!" challenge from www.hackerrank.com is selected. Notice that mentioned challenge hosts up to 79k solutions.

Building a bot capable to extract every code sample in the challenge is not straightforward. For that reason, the implementation relies in Selenium library. Selenium provides a set of atomic methods which enables the developer to automate browser searchs. In that sense, the web scraping module, tries to simulate a click on "View solution" for every solution stored in aforementioned challenge. See Appendix 3 or visit https://github.com/miquelpuigmena/DeepCodeDataset/blob/master/parser.py for actual implementation.

Bot's workflow starts by visiting the user name list of solutions available for that challenge. Notice that maximum entries per page is 100 so, when the bot finishes a page, it has to check wether it was in the last page or not and, if not, go to next page and redo the same cycle. After some investigation, it was found that utilising the bot workflow (browser click simulation) to visit a solution for each user name found ended up overloading the webpage and loosing some code samples. However, a pattern when calling the source code solution was spotted. In order to reach the actual solution, the

final link follows: "https://www.hackerrank.com/rest/contests/master/challenges/{challenge_name}/hackers/{hacker_name}/download_solution". Where challenge_name and hacker_name are variables. Additionally, certain parameters had to be appended to the GET query like a funcitonal cookie.

3.2.2. ***Homogenising a data-set under the same distribution & labeling the data-set***

Owning a full set of source codes that claim to perform *task A* is a magnific first step. However, the application has to make sure that all source codes actually call the implemented task by a generic user. In other words, it could be possible that a sample was scraped but only a function definition is included in the file, without being called nowhere.

To avoid such situation, an homogenisation action to each sample in the data-set is performed. Aforementioned action consists in verifying if function call si present and, if not, implant it to the code as text (find implementation at Appendix 4.1).

Next, source codes have to be tested and profiled. For this first approach, only execution time is taken into account but, in function of *task A* being processed, might be useful to include more measurements. Every individual in the data-set is tested N=10 times and an average time of execution is calculated. Subsequently, value representing the execution time of the sample is ponderated with the score achieved at HackerRank and extracted when web scraping it.

Finally, when every sample has been profiled and labled, it's useful to normalise it according to the values seen in the data-set. For that reason, a normalisation process is applied taking the maximum and minimum profiled values. See full implementation at Appendix 4.2 or visit https://github.com/miquelpuigmena/DeepCodeDataset/blob/master/profiler.py for actual implementation.

3.3. ***Building custom AST data-set from source code data-set***

This section deeply describes meaningful steps involved in the transformation of the data-set. Notice that told conversion is achieved by applying the same algorithm to each sample inside the data-set, modifying, this way, each individual at a time. By the end of this section, an optimised set of code representations for DeepCode will be accessible. Refer to https://bitbucket.org/edan70/2019-deepcode-miquel/src/FinalB/ for first release of the tool.

In order to provide a better depiction of the optimisations suggested by this paper, the reader can find AST's and cAST's final graphical representations (see Illustration 4 and Illustration 5 respectively) extracted from snippet of code (Code 1).

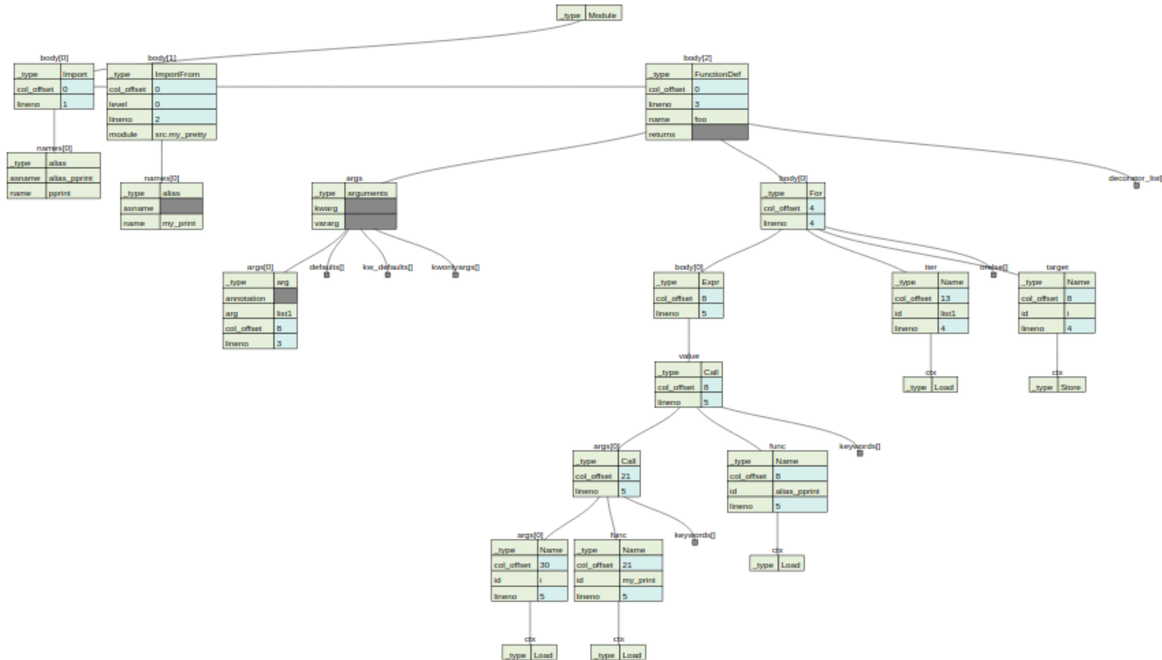


Illustration 4: Representation of test code (Code 1) as AST.

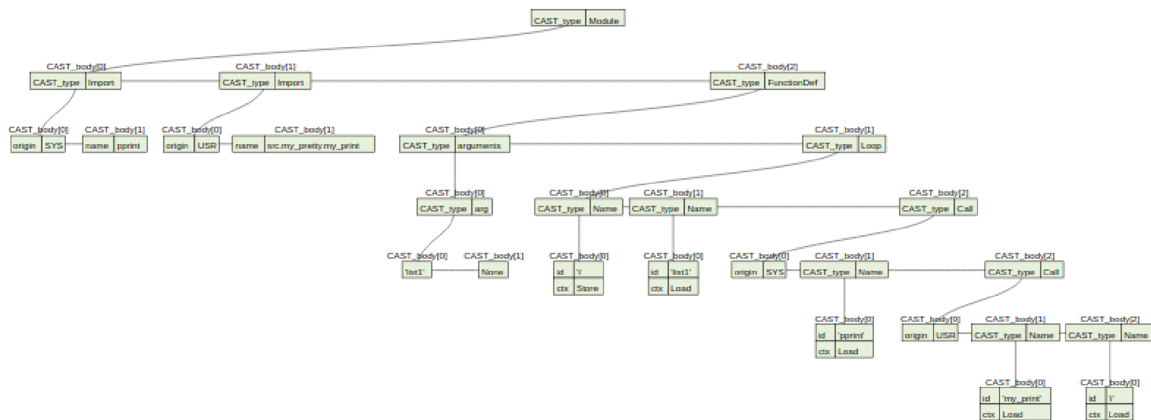


Illustration 5: Representation of test code(Code 1) as cAST.

3.3.1. Gathering the Abstract Syntax Tree

```
import pprint as alias_pprint
from src.my_pretty import
my_print

def foo(list1):
    for i in list1:
        alias_pprint(my_print(i))
```

Code 1: Snippet of code chosen to depict a generous example that includes: Disallowing Aliases (Section 3.3.3.1), Minimizing number of Entities (Section 3.3.3.2), Minimizing number of Nodes (Section 3.3.3.3), Discriminating method Calls (Section 3.3.4.1), and Representing the cAST (Section 3.3.5).

AST is a hierarchical representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code[JON03]. As stated in Section 2, state-of-the-art tools that process source code utilise AST as their inputted data format.

This project only considers Python AST abstraction, constraining, this way, to only take into account python source code.

Python's in-build front-end class *ast* hosts a method *parse()* which takes as input a string and returns an AST tree object. Internally, *ast.parse()* method call *builtin.compile()* including, as argument, *_ast.PyCF_ONLY_AST*. Stated constant implies that only AST representation is calculated while other compiler computations are skipped such as optimization filters.

Injecting *ast*'s functionalities in report's tool is straight-forward by binding them in code. Consequently, a fine AST object and it's context is reachable from DeepCode *phase 1*'s operations.

3.3.2. Traversing the AST

Multiple approaches to traverse trees are known. Concretely, this implementation considers recursive visitors technique while other approaches such as AST matchers[TEA] or cursor traversing[TEB] are also functional.

By visiting nodes, the reader should think of walking through each node in the AST from north-to-south direction. For each encountered node, the node must be analysed and, afterwards, *visit()* function called for every of it's childs.

Visitor class, inherited from *ast.NodeVisitor*, is defined. Among others, *Visitor* implements *visit_{NODE_ENTITY}()* method for every conceivable node entity in the AST. Moreover, given the case where a node entity couldn't match any pre-defined visit method, *generic_visit()* will match as a last instance. Notice that latter visit method is defined to provide robustness to the implementation and to apply a default logic.

Recursive traversal of nodes enables the *Visitor* class to build up the cAST and provide access to it from other points of the implementation.

3.3.3. Filtering nodes

Selecting substantial information from a node is crucial in order to reach report's goals. An intelligent filtering logic is applied in order to maximise compression of an AST without loosing important information.

Filtering cycle will be prompt for each traversed node in the original AST. Depending in node's entity and it's near context, different information will be persisted in the cAST. Following, an overview of the filters considered by report's implementation.

3.3.3.1. Disallowing aliases

Aliases are extremely useful in terms of smoothing the readability of a source code by humans. However, they add a counterproductive link between nodes inside the tree which must be erased.

Visitor class keeps track of aliases assigned in source code and performs a translation to it's original name when persisting information in cAST.

3.3.3.2. Minimizing number of entities in AST

Grouping related entities can be very productive for following processing techniques. By minimizing the number of conceivable entities in the cAST, the natural number of combinations that AI layers have to deal with will decay following Equation

$$Number\ Combinations = N^M$$

Where N is the total number of entities and M is the amount of nodes the tree is composed of.

Followed, a set of the most repeated groups:

- Loop group includes *For*, *AsyncFor*, and *While* node entities.
- Name group includes *Name*, and *NameConstant* node entities.
- Import group includes *Import*, and *ImportFrom* node entities.
- Op group includes *BoolOp*, *BinOp*, and *UnaryOp* node entities.

To see full grouping strategy for Python programming language, refer to Appendix 2 .

Grouping strategy entities entails a relative compression rate of expressions of 0.63 (27 AST expressions and 17 cAST expressions) and a relative compression rate of statements of 0.7 (27 AST statements and 19 cAST statements). Notice that it doesn't necessarily lead to an absolute compression rate as the ones stated before since absolute compression rate directly depends on the probability of appearance of each entity. provides absolute compression rates.

3.3.3.3. Minimizing number of nodes in AST

Naturally, compression can be achieved if nodes can be neglected without loosing information.

This paper understands a negligible node as the ones that, once dropped, can be re-injected again by deterministic means. For instance, a function *Call* node entity emerges from an *Expr* node with no other substantial information from post processing layers perspective. In this case, we can conclude that *Expr* node can be neglected from the AST.

3.3.4. Enriching the AST

Yet an enrichment of the AST can contradict the goal of maximising the compression of the AST, this report suggests that some extra information can be exceptionally advantageous for DeepCode *phase 2*.

3.3.4.1. Discriminating method calls

Knowing the origin of a function call can be convenient in order to decide which nodes should gain more attention or, even, to treat them differently. DeepCode *phase 1* classifies method calls by the ones coming from a method system (SYS), the ones created by the user (USR), and python buildins methods (NATIVE).

Firstly, the tool has to find potential python libraries accessible from the system. Such non-trivial task is performed with the help of python's *modulefinde* package. Once potential SYS and NATIVE method calls are constructed, it's a trivial problem of classifying traversed nodes whose entities are directly related to method calls (e.g., *call* or *import* nodes).

3.3.5. Representing the customised AST

Subsequent treatment of cAST by a generic group of consumers can be very heterogeneous. From consumer's point of view, could be interesting to decide which format the tool should output the cAST. Ergo, implemented tool offers three options to specify application’s output format.

By default, JSON format is used, which fits the tree representation in to a JSON object. Extending it, an encoded JSON representation can be demanded, which codifies every node representation depending in the importance of such node in the global data-set. In contrast, *PyPickle* option enables a way to output a python object representation as a *pickle* representation.

Lastly, DeepCode *phase 1* application allows the consumer to specify if chosen cAST representation will be outputted in console or written in a file.

4. Results

A great asset was achieved in the form of a data-set of ~80k source code labled samples represented with custom AST format. Even though the data-set was ready to process, it was imposible to apply studied artificial inteleligence techniques due to lack of time. For that reason, while *phase 2* has been attacked from a theoretical point of view only *Phase* 1's results were performed .

4.1. DeepCode *Phase 1* Evaluation

The experiments and evaluations are sequentially divided and sorted by increasing significance to the report.

4.1.1. Custom & simple test code

In order to provide an specific example for each of report's implementations, custom source code (Code 1) is analysed. Illustration 4 represents the original AST while Illustration 5 illustrates the transformation applied by DeepCode *phase 1*'s core.

Utilising DeepCode *phase 1*'s analysis tool (See Evaluation 1) the report defines that it's been achieved a compression rate of rate_entities=0.692, and rate_nodes=0.682.

```
DeepCode - Evaluation - INFO - Analysis AST:
total_entities=13, total_nodes=22
DeepCode - Evaluation - INFO - Analysis cAST:
total_entities=9, total_nodes=15
```

Evaluation 1: Analysis extracted from Deepcode phase 1 evaluator logs. Notice that AST is formed by a total of 22 nodes with 13 different entities while cAST cuts it up to 15 nodes representation including only 9 different entities.

4.1.2. Custom & complex test code

Secondly, this paper tries to adjust difficulty of code by analysing a more elaborated implementation. For that reason, big integer factorising algorithm[4] implmented in python is studied.

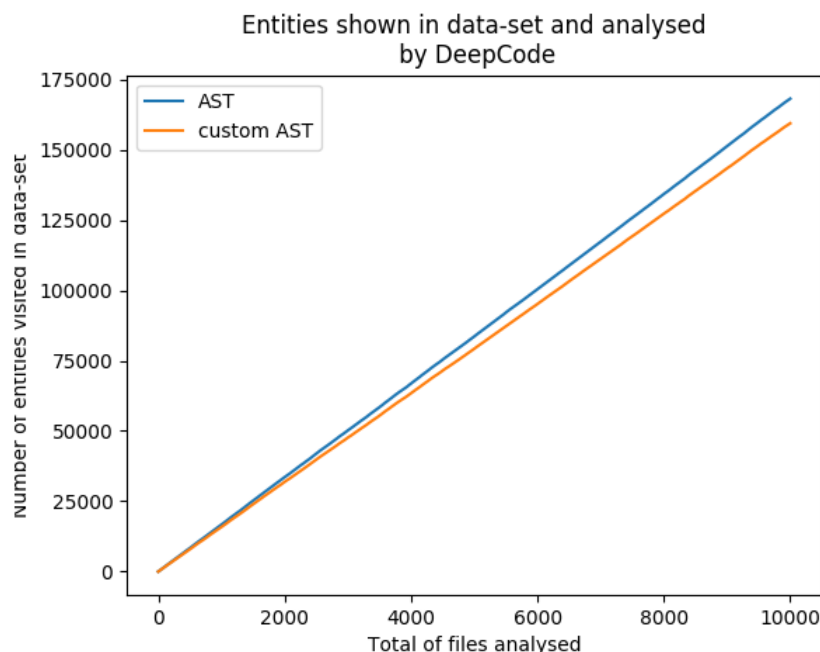
DeepCode - Evaluation - INFO - Analysis AST:
total_entities=55, total_nodes=1151
DeepCode - Evaluation - INFO - Analysis cAST:
total_entities=51, total_nodes=865

Evaluation 2: Analysis extracted from Deepcode phase 1 evaluator logs. Notice that AST is formed by a total of 1151 nodes with 55 different entities while cAST cuts it up to 865 nodes representation including 51 different entities.

One more time, from DeepCode *phase 1*'s analysis tool (See Evaluation 2), a compression rate of rate_entities=0.927, and rate_nodes=0.751 have been reached.

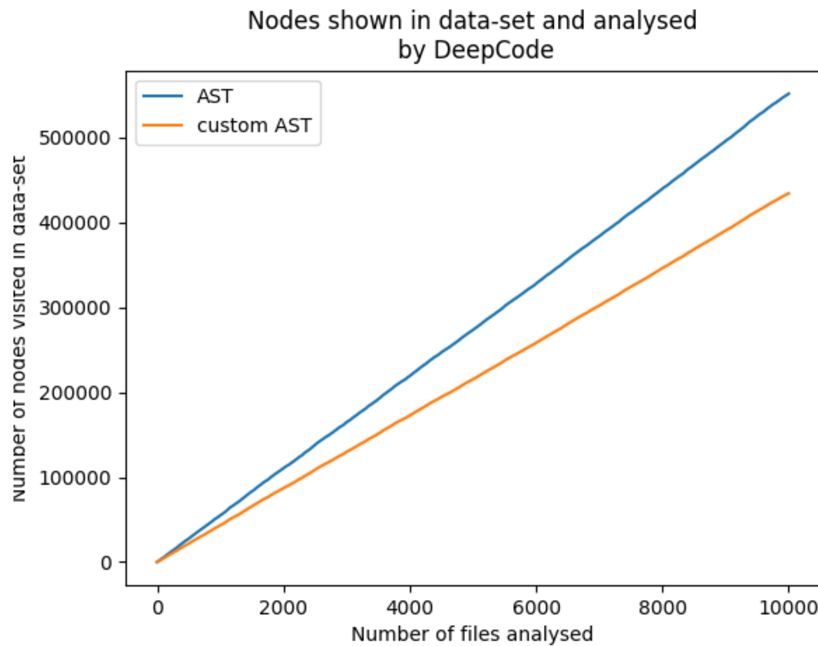
4.1.3. Transforming a 10k files data-set

Finally, this report analyses AST transformation quality in a real data-set under the same distribution. DeepCode includes a web parsing module in order to gather such amount of data.



Graph 1: Accumulation of number of entities seen in each sample in the data-set.

4 Find author's big integer factorising implementation at https://github.com/miquelpuigmena/cryptography/blob/master/factorising/factoring.py.}



Graph 2: Accumulation of number of nodes seen in each sample in the data-set.

After studying and comparing every pair of AST and cAST from each individual in the data-set, we can foresee promising results regarding the amount of needed nodes in order to represent the same information (See Graph 2). In reference to number of entities, a successful behaviour can be spotted (See Graph 1).

4.2. Evolution of a singular source code sample

In a random manner, a sample has been extracted from the data-set under the same distribution performing the task “Capitalizing!” from HackerRank. This section shows how the extracted source code evlves until it’s completely treated by the application.

Firstly, the random sample that will be considered is:

```
# Enter your code here. Read input from STDIN.
Print output to STDOUTo
import string
print (string.capwords(raw_input(), ' '))
```

Code 2: Random code peeked from gathered data-set under the same distribution.This sample achieves a maximum score at HackerRank platform.

Selected code is assigned a mean execution type of 0.0011495 seconds by the application. Ergo, this sample achieves a 0.02299 value by the profiler (mean_execution_type * HackerRank_score).

From that sample code, the following original AST is obtained:

```
{"_type": "Module", "body": [{"_type": "Import", "col_offset": 0, "lineno": 2,
"names": [{"_type": "alias", "asname": null, "name": "string"}]}, {"_type":
"Expr", "col_offset": 0, "lineno": 7, "value": {"_type": "Call", "args":
[{"_type": "Call", "args": [{"_type": "Call", "args": [], "col_offset": 23, "func":
{"_type": "Name", "col_offset": 23, "ctx": {"_type": "Load"}, "id":
"raw_input", "lineno": 7}, "keywords": [], "lineno": 7}, {"_type": "Str",
"col_offset": 36, "lineno": 7, "s": " "}], "col_offset": 7, "func": {"_type":
"Attribute", "attr": "capwords", "col_offset": 7, "ctx": {"_type": "Load"},
"lineno": 7, "value": {"_type": "Name", "col_offset": 7, "ctx": {"_type":
"Load"}, "id": "string", "lineno": 7}}, "keywords": [], "lineno": 7}],
"col_offset": 0, "func": {"_type": "Name", "col_offset": 0, "ctx": {"_type":
"Load"}, "id": "print", "lineno": 7}, "keywords": [], "lineno": 7}}]}
```

Code 3: AST build by Python's core compiler from source code at Code 2.

And the following cAST is build:

```
{"CAST_type": "Module", "CAST_body": [{"CAST_type": "Import",
"CAST_body": [{"origin": "SYS"}, {"name": "string"}]}, {"CAST_type":
"Expr", "CAST_body": [{"CAST_type": "Call", "CAST_body": [{"origin":
"NATIVE"}, {"CAST_type": "Name", "CAST_body": [{"id": "'print'", "ctx":
"Load"}]}, {"CAST_type": "Call", "CAST_body": [{"origin": "UNK"},
{"CAST_type": "Attribute", "CAST_body": [{"attr": "capwords"},
{"CAST_type": "Name", "CAST_body": [{"id": "'string'", "ctx": "Load"}]},
{"CAST_type": "Load", "CAST_body": []}]}, {"CAST_type": "Call",
"CAST_body": [{"origin": "UNK"}, {"CAST_type": "Name", "CAST_body":
[{"id": "'raw_input'", "ctx": "Load"}]}]}, {"CAST_type": "Str", "CAST_body":
["' '"]}]}]}]}]}
```

Code 4: cAST build by Python's core compiler from source code at Code 2.

Concluding, the cAST depicted at Code 4 is persisted under the name of "username`_0.02299_20200101-192032529292.deepcode.py.cAST`" and placed under the folder where the new data-set will be saved.

5. Budget

This project mainly requires a Junior engineer with a base knowledge in Python programming. Consequently, main cost of the project is due to person's salary. When it comes to proprietary software, this thesis utilises open source tools under Python language, meaning that no added cost is needed. Finally, the worker will need a computer in order to implement the solution.

Index	Concept (unit)	Details	Number of units
1	Project dedication (hours/week)		22,5
2	Project duration (weeks)		20
3	Total worked (hours)	Cell1 * Cell2	450
4	Junior engineer salary (€/hour)	Brute salary for junior software developer	13
5	Worked time cost	Cell3 * Cell4	5850
6	Computer		1000
	TOTAL (€)		6850

Table 2: Project cost

6. Conclusions and future development

This thesis has investigated an original manner of assembling an optimal data-set of source codes for future artificial intelligence processing layers. During the study process, a lack of relation between post-processing techniques and the source code text has been spotted. Thus, DeepCode phase 1 has decided to use Abstract Syntax Tree format as it's source code representation baseline to produce an optimised data-set.

Subsequently, mimicking best results found in the field, report's solution focuses in maximizing the compression rate of the AST whilst keeping intact the information contained in the original tree object. DeepCode phase 1 tool achieves promising results in terms of absolute number of nodes needed in customised AST against a generic AST extracted from compiler's engine. Additionally, the report introduces a grouping strategy that, with further fine tuning adjustments, can be extremely beneficial for AI related applications.

Lastly, this work includes the concept of appending auxiliary information to source code representations in order to facilitate and flatten the learning curve of a generic AI system. In future, new paradigms are worth studying such as subtasking a concrete task utilising deterministic means or statistically studying the impact of entity grouping strategy towards the compression rate and compression quality achieved by the tool. Nonetheless, this paper offers a unique bridge for AI systems to work in an optimal manner with source code datasets.

Bibliography

[JON03] : Jones, Joel. "bstract Syntax Tree Implementation Idioms". (2003). URL: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf. Last visited on .

[APS16] : Miltiadis Allamanis and Hao Peng and Charles A. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: CoRR (2016). URL: http://arxiv.org/abs/1602.03001.

[ABS14] : Miltiadis Allamanis and Earl T. Barr and Charles A. Sutton. "Learning Natural Coding Conventions". In: (2014). URL: http://arxiv.org/abs/1402.4182.

[Whi+16] : M. White and M. Tufano and C. Vendome and D. Poshyvanyk. "Deep learning code fragments for code clone detection". In: (2016). URL: .

[Moub+14] : Lili Mou and Ge Li and Zhi Jin and Lu Zhang and Tao Wang. "Tree-Based Convolutional Neural Network for Programming Language Processing". In: CoRR (2014). URL: http://arxiv.org/abs/1409.5718.

[Moua+14] : Lili Mou and Ge Li and Yuxuan Liu and Hao Peng and Zhi Jin and Yan Xu and Lu Zhang. "Building Program Vector Representations for Deep Learning". In: CoRR (). URL: http://arxiv.org/abs/1409.3358.

[Lia+19] : H. Liang and L. Sun and M. Wang and Y. Yang. "Deep Learning with Customized Abstract Syntax Tree for Bug Localization". In: IEEE Access (2019). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8809752.

[LA04] : C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". 2004

[UCa] : University of Illinois at Urbana-Champaign. "Clang: a C language family frontend for LLVM". (). URL: https://clang.llvm.org/. Last visited on .

[PDCc] : Python Documentation. "Python compiler package". URL: https://docs.python.org/2/library/compiler.html. Last Visted on .

[TEA] : The Clang Team. "Matching the Clang AST". URL: https://clang.llvm.org/docs/LibASTMatchers.html. Last Visted on 06/12/2019.

[TEB] : The Clang Team. "Traversing the AST with cursors". URL: https://clang.llvm.org/doxygen/group__CINDEX__CURSOR__TRAVERSAL.html. Last Visted on 06/12/2019.

Appendices

1. DeepCode Phase1: Compilers approach

DeepCode phase 1: Compilers approach

1st Miquel Puig i Mena
Student at Universitat Politècnica de Catalunya (UPC)
Lunds Tekniska Högskola (LTH)
Lund, Sweden
miquel.puig.mena@lu.lth.se

***Abstract*—Yet being immersed in the globalisation era, we can still perceive isolation and rivalry between software developing teams that share ambitions. Nonetheless, DeepCode proposes a cooperation between parties in order to achieve greater results. DeepCode is a tool that, by processing a data-set formed by source codes performing a generic *task A*, learns how to implement *task A* and outperform every individual from studied data-set. Unfortunately, no existing tool is capable of homogenizing a source code data-set in a manner that can be post processed by AI tools. DeepCode *phase 1* targets to fill this gap in the field by offering a standardised data structure representing a generic source code. Furthermore, this paper presents a practical application that generates aforementioned representation given a well-formed source code file.**

***Index Terms*—Abstract syntax tree, Tree traversal, Code auto-generation, Deep Learning.**

I. Introduction

Coders, Group of coders, Departments, Companies or Group of companies compete every day to get the most efficient algorithm for a specific problem. Regardless of the nature of the research, a common denominator can be spotted: they all try to get the best existing ideas on the field and apply some new improvement/s that makes their own implementation the best in a specific set of "tangible" aspects (performance, usability, efficiency, etc.). In that sense, enormous amount of time and effort is spent by each researcher analyzing and mastering the field. Additionally, told hypothetical researcher won't have any kind of assurance that a competitive approach has been discarded during the process.

In this paper, DeepCode is presented, a new tool that will try to auto-generate unique and original code. DeepCode system aims to generate code that solves a generic *task A* by learning, from a large set of different codes, data-set, how to solve *task A*. Notice that each element in the data-set tries, as well, to solve *task A*. Therefore, the concept data-set under same distribution, references a group of different code implementations that perform the exact same job such as compressing a file, processing a signal or solving mathematical paradigms. Furthermore, DeepCode's most innovative objective is the ability to overcome best test-performance reached by any individual from the data-set it's been learning from.

Given the complexity and nature of DeepCode, two stages are defined: *Phase 2*, that, with Artificial Intelligence (AI) techniques, learns which ideas in the data-set imply good code performance and *phase 1*, which studies and builds customised code representation in order to provide *phase 2* an optimal and workable data-set.

This paper deeply describes *phase 1* of DeepCode but not *phase 2*. Nevertheless, *phase 2* is relevant for this paper since it defines what kind of transformation has to be applied to the source code data-set in order to accomplish future intelligent decisions (See Figure 1). In that sense, this paper suggests, builds and provides an original code representation to be utilised afterwards by AI processes.

Most generally, source codes, irrespective of it's original language, can be represented in lower abstractions levels. From machine code[1], unintelligible from human perspective, until low level code such as Assembly. For DeepCode's purposes, a middle level representation of source code must be generated. Such intermediate depiction of the code allows any data-set to be homogenized, allowing, this way, to make fair comparisons among code samples under the same distribution.

Arbitrary code analysis and transformation is accomplished by generating an abstraction of the source code called Abstract Syntax Tree [2] (AST). Generally, AST abstraction format is fed to subsequent compiler's layers such as optimization of code or Assembly code generation. This paper is highly focused in AST generation stage.

Installed Python packages include an in-build Python compiler written in C Programming language. Additionally, Python packages include in-build front-end libraries enabling, from Python code, to access compiler's internals. Among other modules, *ast* [3] front-end can be found, which converts a Python source code into an AST. From *ast* object, a rich AST context is accessible, providing, this way, a great entry point to transform the AST.

By DeepCode's means, tree representation of the code (AST) is very profitable. Due to time consuming processes in *phase 2*, every redundant information in the AST must be eliminated. In other words, internal AST generated by the compiler includes neglectable nodes from DeepCode's perspective. In that sense, AST generated by the compiler must be simplified as much as possible without loosing valuable information. Hence, above strategy satisfies sought middle level representation of source code.

[1]Machine code, also called machine language, is a computer language that is directly understandable by a computer's Central Processing Unit (CPU), and it is the language into which all programs must be converted before they can be run. Each CPU type has its own machine language, although they are basically fairly similar. [1]

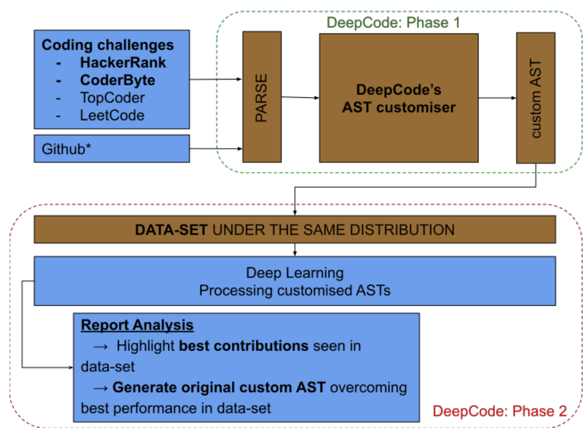


Fig. 1: Architectural schema representing DeepCode's full scope. Notice the contribution provided by each phase of DeepCode to the full project.

Summarizing, this paper suggests and provides a compressed and enriched code representation that will be used to feed further post-processes layers within DeepCode scope.

The rest of this paper is organized as follows. Section II introduces similar existing ideas in the field. Section III characterizes AST architecture and suggests a feasible transformation while section IV describes how to achieve customised AST. Section V evaluates results achieved by testing the application against realistic use cases. Suggested extensions to paper's work are found in section VI and, finally, section VII concludes with a summary of report's achievements.

II. Related work

Source code analysis has drastically evolved since AI's reborn. While code processing tools were exclusively based in complex and deterministic techniques, AI provides another dimension to existing tools. By gathering information from user experience, AI enables code processing tools to optimize itself on the fly. Currently, there's a big interest and investment from a well established community. Specifically, a big effort is invested by Integrated Development Environment (IDE) and code analysis applications such as bug finders or providing interesting and personalised hints while coding.

[4] provides a brief text summary of what an inputted code intends to do while [5] provides a tool to learn which non-written conventions and patterns are followed by developers. Both studies have in common that they treat source code as simple text instead of utilising standardised representations such as ASTs or machine code.

Furthermore, finest results were achieved by studies utilising abstracted representations of source code. Code clone detection application [6] overcame state-of-the-art of the moment by extracting AST of it's source code data-set.

An original suggestion is found in [7], where they propose an encoded version of the AST based in [8], that converts tree structures in vectorised information. Great improvement on bug localisation was achieved by [9] by adopting the program representation proposed by [7]. However, [9] included the concept of compressing the AST as much as possible in order to facilitate AI layers to complete their job with optimal results.

Commonly, each programming language has it's own AST generator engine and, in many cases, more than one. This paper attempts to corroborate that an intelligent filter in current ASTs can be extremely beneficial for post processing techniques.

Low Level Virtual Machine [10] (LLVM) is a compiler framework that uses a build in intermediate human readable code language with slightly higher level abstractions than Assembly. Although LLVM code can be written from scratch, it's typical to produce it when compiling source code, as intermediate step for code optimization layers and/or optimal machine code generation.

Clang project provides a language front-end and tooling infrastructure for languages in the C language family for the LLVM project [11]. Clang is an open source project well established in the community, having up to 79k commits by the time this report is written. Although Clang is backed by a huge community and provides reliable services, this paper will choose Python compiler [12], a less complex compiler engine system that demands less time to familiarise with.

III. Architecture

Within DeepCode's scope, DeepCode *phase 1* commences once the text data-set is assembled (Figure 2). Proposed tool by this paper handles each element in the data-set individually. In this regard, CLI module provides an entry point to DeepCode *phase 1* where each sample is injected in an iterative way.

Python's inner compiler engine produces an AST that will be trust and used by subsequent modules. Such dependency on Python's compiler core can be assumed given the reliability provided by the existence of a vast community along Python itself.

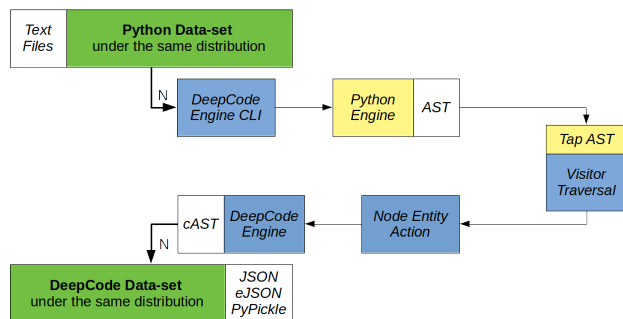


Fig. 2: Architectural model of DeepCode *phase 1*. From gathered python text files[2], DeepCode's CLI triggers transformation cycle for each individual sample in the data-set. Notice that final version of cAST can be outputted in JSON, eJSON, or as PyPickle.

Parallel, Python's engine is bound to report's tool in a way that AST context is shared and accessible. Thereafter, every node in retrieved AST is traversed following a visitor strategy in order to forge pursued custom AST(cAST). While visiting a node, paper's implementation decides which information is relevant from that node, if any at all, and persists it in the cAST. Notice that personalised tree carries a one to one connection between cAST nodes and their AST node pair.

Finally, DeepCode *phase 1* application grants to the user the functionality to store or output created cAST in diverse formats. This report contemplates JSON, encoded JSON[3] (eJSON), or pickle format (PyPickle) to be shown at console or saved in a file.

IV. Implementation

This section deeply describes meaningful steps involved in the transformation of the data-set. Notice that told conversion is achieved by applying the same algorithm to each sample inside the data-set, modifying, this way, each individual at a time. By the end of this section, an optimal set of code representations for DeepCode will be accessible.

In order to provide a better depiction of the optimisations suggested by this paper, the reader can find AST's and cAST's final graphical representations extracted from snippet of code (Listing 1) at Figure 3 and Figure 4 respectively.

A. Gathering the Abstract Syntax Tree

AST is a hierarchical representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code [2]. As stated in section II, state-of-the-art tools that process source code utilise AST as their inputted data format.

This project only considers Python AST abstraction, constraining, this way, to only take into account Python source code. Even though it might appear a limitation, it is enough to proof paper's objective.

Python's in-build front-end class *ast* hosts a method *parse()* which takes as input a string and returns an AST tree object. Internally, *ast.parse()* method calls *builtin.compile()* including, as argument, *_ast.PyCF_ONLY_AST*. Named constant implies that only AST representation is calculated while other compiler computations are skipped such as optimization.

Injecting *ast*'s functionalities in report's tool is straightforward by binding them in code. Consequently, a fine AST object and it's context is reachable from DeepCode *phase 1*'s operations.

B. Traversing the AST

Multiple approaches to traverse trees are known. Concretely, this implementation considers recursive visitors technique while other approaches such as AST matchers [13] or cursor traversing [14] are also functional.

By visiting nodes, the reader should think of walking through each node in the AST from north-to-south direction. For each encountered node, the node must be analysed and, afterwards, *visit()* function called for every of it's childs.

Visitor class, inherited from *ast.NodeVisitor*, is defined. Among others, *Visitor* implements *visit_{NODE_ENTITY}()* method for every conceivable node entity in the AST. Moreover, given the case where a node entity couldn't match any pre-defined visit method, *generic_visit()* will match as a last instance. Notice that latter visit method is defined to provide robustness to the implementation and to apply a default logic.

Recursive traversal of nodes enables the *Visitor* class to build up the cAST and provide access to it from other points of the implementation.

C. Filtering nodes

Selecting substantial information from a node is crucial in order to reach report's goals. An intelligent filtering logic is applied in order to maximise compression of an AST without loosing important information.

Filtering cycle will be prompt for each traversed node in the original AST. Depending in node's entity and it's near context, different information will be persisted in the cAST. Following, an overview of the filters considered by report's implementation.

1) Disallowing Aliases: Aliases are extremely useful in terms of smoothing the readability of a source code by humans. However, they add a counterproductive link between nodes inside the tree which must be erased.

Visitor class keeps track of aliases assigned in source code and performs a translation to it's original name when persisting information in cAST.

2) Minimizing number of entities in AST: Grouping related entities can be very productive for following processing techniques. By minimizing the number of conceivable entities in the cAST, the natural number of combinations that AI layers have to deal with will decay following equation 1.

$$num_combinations = N^M \tag{1}$$

Where N is the total number of entities and M is the amount of nodes the tree is composed of.

Followed, a set of the most repeated groups:

* Loop group includes *For*, *AsyncFor*, and *While* node entities.
* Name group includes *Name*, and *NameConstant* node entities.
* Import group includes *Import*, and *ImportFrom* node entities.
* Op group includes *BoolOp*, *BinOp*, and *UnaryOp* node entities.

Grouping strategy entities entails a relative compression rate of expressions of 0.63 (27 AST expressions and 17 cAST expressions) and a relative compression rate of statements of 0.7 (27 AST statements and 19 cAST statements). Notice

[3] Encoded JSON directly depends in final global DeepCode data-set. Depending in the different node entities sighted in data-set, the eJSON will cipher accordingly.

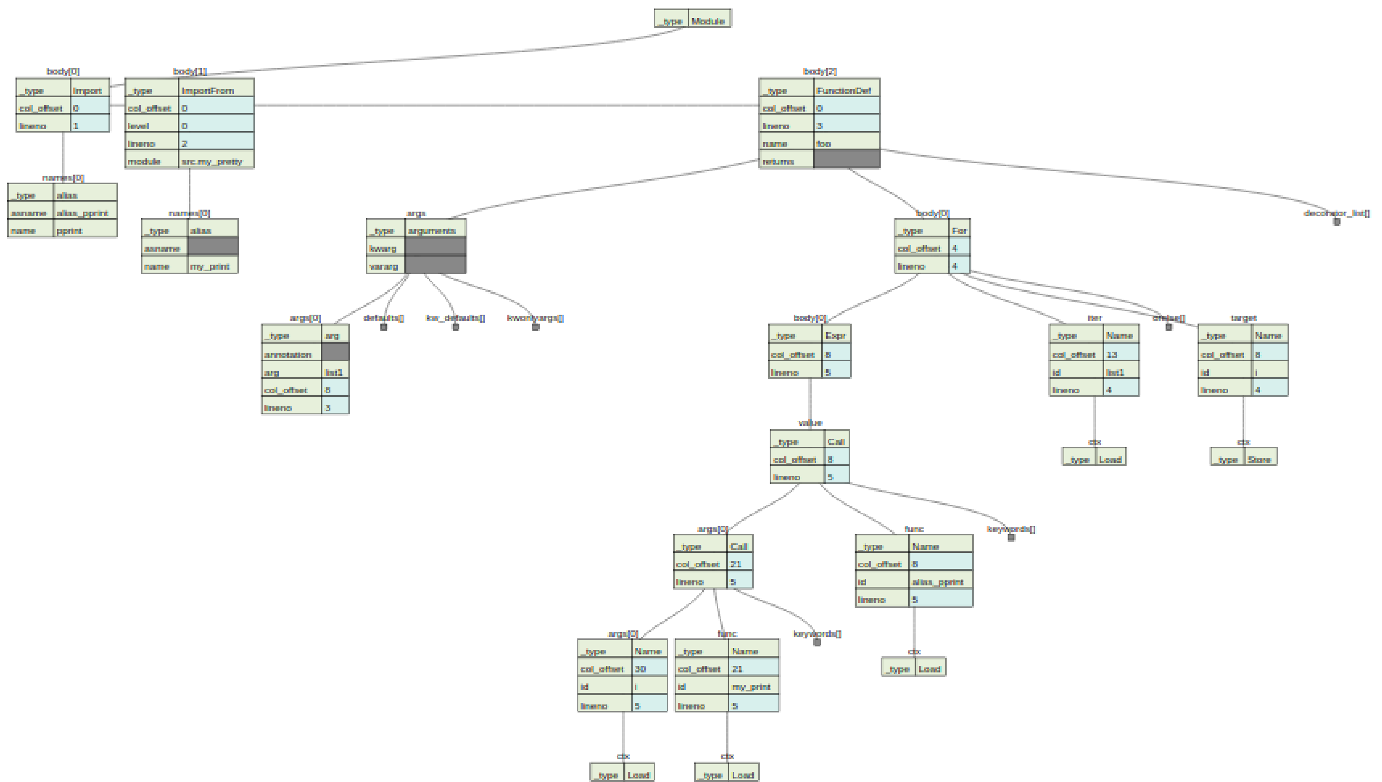


Fig. 3: Representation of test code(1) as AST.

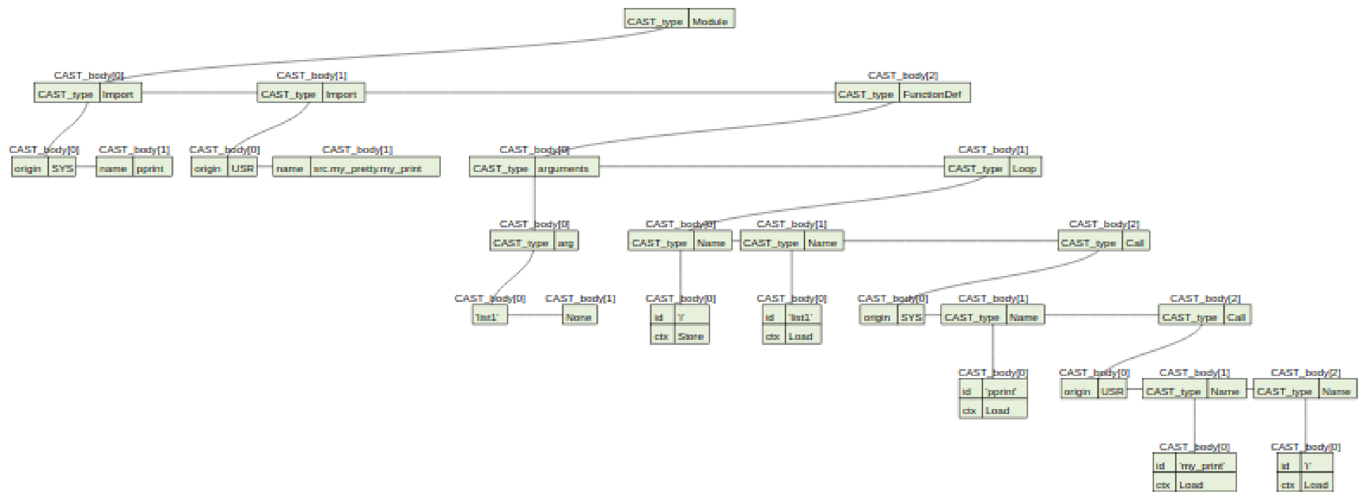


Fig. 4: Representation of test code(1) as cAST.

that it doesn't necessarily lead to an absolute compression rate as the ones stated before since absolute compression rate directly depends on the probability of appearance of each entity. Section V provides absolute compression rates.

3) Minimizing number of nodes in AST: Naturally, compression can be achieved if nodes can be neglected without loosing information.

This paper understands a negligible node as the ones that, once dropped, can be re-injected again by deterministic means. For instance, a function *Call* node entity emerges from an *Expr* node with no other substantial information from post processing layers perspective. In this case, we can conclude that *Expr* node can be neglected from the AST.

```
import pprint as alias_pprint
from src.my_pretty import my_print

def foo(list1):
    for i in list1:
        alias_pprint(my_print(i))
```

Listing 1: Snippet of code chosen to depict an example of *Disallowing Aliases* (IV-C1), *Minimizing number of Entities* (IV-C2), *Minimizing number of Nodes* (IV-C3), *Discriminating method Calls* (IV-D1) and *Representing the cAST* (IV-E)

D. Enriching the AST

Yet an enrichment of the AST can contradict the goal of maximising the compression of the AST, this report suggests that some extra information can be exceptionally advantageous for DeepCode *phase 2*.

1) Discriminating method Calls: Knowing the origin of a function call can be convenient in order to decide which nodes should gain more attention or, even, to treat them differently. DeepCode *phase 1* classifies method calls by the ones coming from a method system (SYS), the ones created by the user (USR), and Python buildins methods (NATIVE).

Firstly, the tool has to find potential Python libraries accessible from the system. Such non-trivial task is performed with the help of Python's *modulefinder* package. Once potential SYS and NATIVE method calls are constructed, it's a trivial problem of classifying traversed nodes whose entities are directly related to method calls (e.g., *call* or *import* nodes).

E. Representing the customised AST

Subsequent treatment of cAST by a generic group of consumers can be very heterogeneous. From consumer's point of view, could be interesting to decide which format the tool

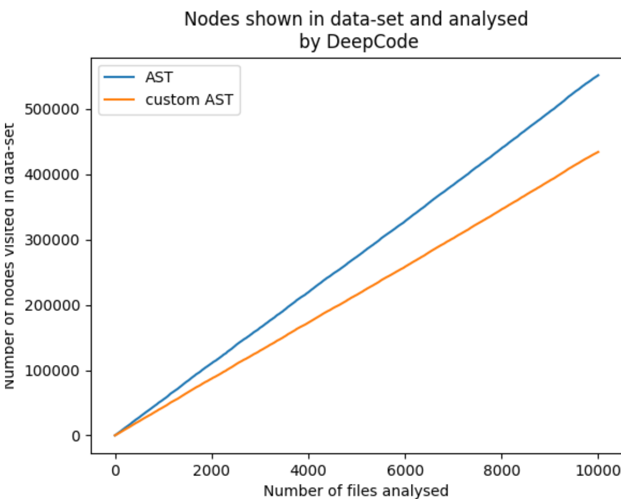


Fig. 5: Accumulation of number of nodes seen in each sample in the data-set.

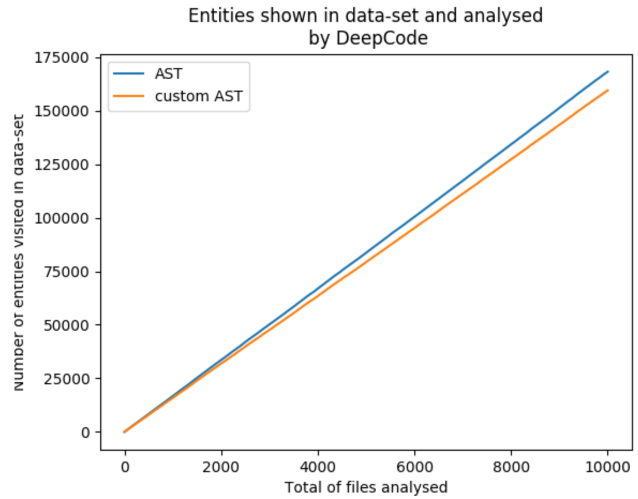


Fig. 6: Accumulation of number of entities seen in each sample in the data-set.

should output the cAST. Ergo, implemented tool offers three options to specify application's output format.

By default, JSON format is used, which fits the tree representation in to a JSON object. Extending it, an encoded JSON representation can be demanded, which codifies every node representation depending in the importance of such node in the global data-set. In contrast, *PyPickle* option enables a way to output a Python object representation as a *pickle* representation [15].

Lastly, DeepCode *phase 1* application allows the consumer to specify if chosen cAST representation will be outputted in console or written in a file.

V. Experiments

The experiments and evaluations are sequentially divided and sorted by increasing significance to the report.

A. Custom-simple test code

In order to provide an specific example for each of report's implementations, custom source code (Listing 1) is analysed. Figure 3 represents the original AST while Figure 4 illustrates the transformation applied by DeepCode *phase 1*'s core.

Utilising DeepCode *phase 1*'s analysis tool (see results in box below) the report defines that it's been achieved a compression rate of rate_entities=0.692, and rate_nodes=0.682.

DeepCode - Evaluation - INFO - Analysis AST:
total_entities=13, total_nodes=22
DeepCode - Evaluation - INFO - Analysis cAST:
total_entities=9, total_nodes=15

B. Custom-complex test code

Secondly, this paper tries to adjust difficulty of code by analysing a more elaborated implementation. For that reason, big integer factorising algorithm[4] is studied.

One more time, from DeepCode *phase 1*'s analysis tool (see results in box below), a compression rate of rate_entities=0.927, and rate_nodes=0.751 have been reached.

```
DeepCode - Evaluation - INFO - Analysis AST:
total_entities=55, total_nodes=1151
DeepCode - Evaluation - INFO - Analysis cAST:
total_entities=51, total_nodes=865
```

C. Transforming a 10k files data-set

Finally, this report analyses AST transformation quality in a real data-set under the same distribution. DeepCode includes a web parsing module in order to gather such amount of data.

After studying and comparing every pair of AST and cAST from each individual in the data-set, we can foresee promising results regarding the amount of needed nodes in order to represent the same information (Figure 5). In reference to number of entities, a successful behaviour can be spotted (Figure 6).

VI. Future work

This section suggest two lines of work that can directly benefit paper's implementation.

This report suggests that there's margin of improvement regarding current grouping techniques. Adding new group or fine tuning existing groups might lead to higher compression rate in terms of number of entities in cAST.

Moreover, DeepCode *phase 2* tries to discover every *sub task A* that constructs *task A*. When distinguished, each sub task is processed independently with the aim to identify how beneficial, compared to aligned sub tasks in the data-set, the sub task is to accomplish *task A*. From this perspective, custom AST can be enriched in a manner that identifies and divides each *sub task* within *task A*'s code.

VII. Conclusions

This paper has investigated an original manner of assembling an optimal data-set of source codes for future artificial intelligence processing layers. During the study process, a lack of relation between post-processing techniques and the source code text has been spotted. Thus, DeepCode *phase 1* has decided to use Abstract Syntax Tree format as it's source code representation baseline to produce an optimised data-set.

Subsequently, mimicking best results found in the field, report's solution focuses in maximizing the compression rate of the AST whilst keeping intact the information contained in the original tree object. DeepCode *phase 1* tool achieves promising results in terms of absolute number of nodes needed in customised AST against a generic AST extracted from compiler's engine. Additionally, the report introduces a grouping strategy that, with further fine tuning adjustments, can be extremely beneficial for AI related applications.

Lastly, this paper includes the concept of appending auxiliary information to source code representations in order to facilitate and flatten the learning curve of a generic AI system.

In future, new paradigms are worth studying such as subtasking a concrete task utilising deterministic means or statistically studying the impact of entity grouping strategy towards the compression rate and compression quality achieved by the tool. Nonetheless, this paper offers a unique bridge for AI systems to work in an optimal manner with source code datasets.

[4]Find big integer factorising implementation at https://github.com/miquelpuigmena/cryptography/blob/master- /factoris-ing/factoring.py.

References

[1] T. L. I. Project. (2006) Machine code definition. [Online]. Available: http://www.linfo.org/machine_code.html

[2] J. Jones, "Abstract syntax tree implementation idioms," *Pattern Languages of Program Design*, 2003. [Online]. Available: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf

[3] P. Documentation. Abstract syntax trees. [Online]. Available: https://docs.python.org/3/library/ast.html

[4] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[5] M. Allamanis, E. T. Barr, and C. A. Sutton, "Learning natural coding conventions," *CoRR*, vol. abs/1402.4182, 2014. [Online]. Available: http://arxiv.org/abs/1402.4182

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

[7] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang, "Building program vector representations for deep learning," *CoRR*, vol. abs/1409.3358, 2014. [Online]. Available: http://arxiv.org/abs/1409.3358

[8] H. Zhang, S. Wang, X. Xu, T. W. S. Chow, and Q. M. J. Wu, "Tree2vector: Learning a vectorial representation for tree-structured data," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5304–5318, Nov 2018.

[9] H. Liang, L. Sun, M. Wang, and Y. Yang, "Deep learning with customized abstract syntax tree for bug localization," *IEEE Access*, vol. PP, pp. 1–1, 08 2019.

[10] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis transformation," 04 2004, pp. 75– 86.

[11] U. of Illinois at Urbana-Champaign. Clang: a c language family frontend for llvm. [Online]. Available: https://clang.llvm.org/

[12] P. Documentation. Python compiler package. [Online]. Available: https://docs.python.org/2/library/compiler.html

[13] U. of Illinois at Urbana-Champaign. How to write recursiveastvisitor based astfrontendactions. [Online]. Available: https://clang.llvm.org/docs/RAVFrontendAction.html

[14] T. C. Team. Traversing the ast with cursors. [Online]. Available: https://clang.llvm.org/doxygen/group__CINDEX__CURSOR__TRAVERSAL.html

[15] P. Documentation. Python object serialization. [Online]. Available: https://docs.python.org/3/library/pickle.html

2. Grouping strategy from Python’s original entities

-- ASDL's 5 builtin types are:

-- identifier, int, string, object, constant

module Python

{

mod = Module(stmt* body, type_ignore *type_ignores)

| Interactive(stmt* body)

| Expression(expr body)

| FunctionType(expr* argtypes, expr returns)

-- not really an actual node but useful in Jython's typesystem.

| Suite(stmt* body)

stmt = FunctionDef(identifier name, arguments args,

stmt* body, expr* decorator_list, expr? returns,

string? type_comment)

| AsyncFunctionDef(identifier name, arguments args,

stmt* body, expr* decorator_list, expr? returns,

string? type_comment)

| ClassDef(identifier name,

expr* bases,

keyword* keywords,

stmt* body,

expr* decorator_list)

| Return(expr? value)

| Delete(expr* targets)

| Assign(expr* targets, expr value, string? type_comment)

| ~~AugAssign(expr target, operator op, expr value)~~

→ Assign

| ~~AnnAssign(expr target, expr annotation, expr? value, int simple)~~

→ Assign

-- use 'orelse' because else is a keyword in target languages

| ~~For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)~~

→ Loop

| ~~AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)~~

→ Loop

| ~~While(expr test, stmt* body, stmt* orelse)~~

→ Loop

| If(expr test, stmt* body, stmt* orelse)

| ~~With(withitem* items, stmt* body, string? type_comment)~~

→ FlowMgmt

| ~~AsyncWith(withitem* items, stmt* body, string? type_comment)~~

→ FlowMgmt

| ~~Raise(expr? exc, expr? Cause)~~

→ FlowMgmt

| ~~Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)~~

→ FlowMgmt

| ~~Assert(expr test, expr? msg)~~

→ FlowMgmt

| Import(alias* names)

| ~~ImportFrom(identifier? module, alias* names, int? level)~~

→ Import

| ~~Global(identifier* names)~~

→ Identifier

| ~~Nonlocal(identifier* names)~~

→ Identifier

| Expr(expr value)

| Pass | Break | Continue

-- XXX Jython will be different

-- col_offset is the byte offset in the utf8 string the parser uses

attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

-- BoolOp() can use left & right

expr = BoolOp(boolop op, expr* values)

| NamedExpr(expr target, expr value)

| BinOp(expr left, operator op, expr right)

| UnaryOp(unaryop op, expr operand)

| Lambda(arguments args, expr body)

| IfExp(expr test, expr body, expr orelse)

| Dict(expr* keys, expr* values)

| Set(expr* elts)

| ~~ListComp(expr elt, comprehension* generators)~~

→ List

| ~~SetComp(expr elt, comprehension* generators)~~

→ Set

| ~~DictComp(expr key, expr value, comprehension* generators)~~

→ Dict

| GeneratorExp(expr elt, comprehension* generators)

-- the grammar constrains where yield expressions can occur

| Await(expr value)

| Yield(expr? value)

| ~~YieldFrom(expr value)~~

→ Yield

-- need sequences for compare to distinguish between

```
    -- x < 4 < 3 and (x < 4) < 3
    | Compare(expr left, cmpop* ops, expr* comparators)
    | Call(expr func, expr* args, keyword* keywords)
    | FormattedValue(expr value, int? conversion, expr? format_spec)
    | ~~JoinedStr(expr* values)~~
                    → List
    | Constant(constant value, string? kind)

    -- the following expression can appear in assignment context
    | Attribute(expr value, identifier attr, expr_context ctx)
    | Subscript(expr value, slice slice, expr_context ctx)
    | Starred(expr value, expr_context ctx)
    | Name(identifier id, expr_context ctx)
    | List(expr* elts, expr_context ctx)
    | Tuple(expr* elts, expr_context ctx)

     -- col_offset is the byte offset in the utf8 string the parser uses
     attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)
     | ExtSlice(slice* dims)
     | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
          | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)

excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
               attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
          expr* kw_defaults, arg? kwarg, expr* defaults)

arg = (identifier arg, expr? annotation, string? type_comment)
      attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)
```

```
    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)

    type_ignore = TypeIgnore(int lineno, string tag)
}
```

3. Web scraping HackerRank's "Capitalize!" challenge

```
import logger
from selenium.webdriver import Firefox
from selenium.webdriver.firefox.options import Options
from selenium.webdriver.common.by import By
from selenium.common.exceptions import NoSuchElementException
from datetime import datetime

from urllib.request import Request, urlopen

persisted = 0
failed = 0
logger = logger.get_logger("WebParser")

def parse_challenge(driver, challenge, summary, page_number=1) -> dict:
    assert 'persisted' in summary
    assert 'failed' in summary
    URL_BY_CHALLENGE_PAGE = "https://www.hackerrank.com/challenges/{challenge}/leaderboard?limit=100&page={page_number}"
    css_path_to_table_body = 'html body#hr_v2.hr-community div#content div.ui-kit-root ' \
                    'div.body-wrap.community-page.challenges-page.leaderboard-page div.theme-m.new-design ' \
                    'div.community-content div.challenge-view.theme-m div.challenge-leaderboard ' \
                    'div.container.panes-container div.left-pane div.ui-tabs-wrap ' \
                    'div.tab-list-content.tab-content section.theme-m.ui-leaderboard ' \
                    'div.general-table-wrapper ' \
                    'div.general-table div.ui-table.ui-leaderboard-table.first-col-raised div.table-body > div'
    max_page_number = get_max_page(driver, URL_BY_CHALLENGE_PAGE.format(challenge=challenge, page_number=page_number))
    while page_number < max_page_number:
        logger.debug("="*100)
        logger.debug("Page number: {}.".format(page_number))
        url = URL_BY_CHALLENGE_PAGE.format(challenge=challenge, page_number=page_number)
        driver.get(url)
        table_body = driver.find_elements(By.CSS_SELECTOR, css_path_to_table_body)
        parse_ranking_table(table_body, summary)
        page_number += 1
        logger.debug("-"*100)
    return summary


def get_max_page(driver, url) -> (int, Exception):
    css_path_to_last_page = 'html body#hr_v2.hr-community div#content div.ui-kit-root ' \
                    'div.body-wrap.community-page.challenges-page.leaderboard-page div.theme-m.new-design ' \
```

```
                'div.community-content div.challenge-view.theme-m div.challenge-leaderboard ' \
                'div.container.panes-container div.left-pane ' \
                'div.pagination-wrap.clearfix.pagination-wrapper.mlT.leaderboard-pagination ' \
                'div.ui-pagination.theme-m ul li.page-item.last-page a.page-link'
    driver.get(url)
    try:
        last_page_num = driver.find_element(By.CSS_SELECTOR, css_path_to_last_page).get_property("textContent")
    except Exception:
        raise Exception("Couldn't find last page number. Might be triggered by new page format, url migration, ...")
    return int(last_page_num)


def parse_ranking_table(table: list, summary: dict) -> (None, Exception):
    csspath_from_table_to_hackername = 'div.table-row-wrapper ' \
                                       'div.table-row.flex div.table-row-column.ellipsis.hacker ' \
                                       'div.d-flex.justify-content-between.ellipsis a'
    csspath_from_table_to_hackerrank = 'div.table-row-wrapper div.table-row.flex ' \
                                       'div.table-row-column.ellipsis.rank div.ellipsis'
    csspath_from_table_to_hackerscore = 'div.table-row-wrapper div.table-row.flex ' \
                                        'div.table-row-column.ellipsis.score div'
    for i, row in enumerate(table):
        try:
            hacker_name = row.find_element(By.CSS_SELECTOR, csspath_from_table_to_hackername)\
                .get_property("textContent")
            hacker_rank = row.find_element(By.CSS_SELECTOR, csspath_from_table_to_hackerrank)\
                .get_property("textContent")
            hacker_score = row.find_element(By.CSS_SELECTOR, csspath_from_table_to_hackerscore)\
                .get_property("textContent")
            hacker_info = [hacker_name, hacker_rank, hacker_score]
            new_file_name = "_".join(hacker_info) + '_' + datetime.now().strftime("%Y%m%d-%H%M%S%f")
            hacker_code = get_code_from_hacker(hacker_name)
            str2file(string=hacker_code, file_name=new_file_name, summary=summary)
        except NoSuchElementException:
            summary.update({'failed': int(summary.get('failed')) + 1 })
            logger.error("Selenium element not found.")
        except Exception:
            summary.update({'failed': int(summary.get('failed')) + 1 })
            logger.warning("Couldn't parse challenge from user '{}'.".format(hacker_name))
        else:
            summary.update({'persisted': int(summary.get('persisted')) + 1 })

def get_code_from_hacker(hacker_name: str) -> (str, Exception):
    URL_CODE_BY_HACKER_PRIMARY = 'https://www.hackerrank.com/rest/contests/master/challenges/capitalize' \
                          '/hackers/{hacker_name}/download_solution?primary=true'
```

```
    URL_CODE_BY_HACKER_NOT_PRIMARY = 'https://www.hackerrank.com/rest/contests/master/challenges/capitalize/hackers' \
                            '/{hacker_name}/download_solution'
    MY_COOKIE = 'Cookie: hackerrank_mixpanel_token=1e57a69d-f61d-439f-a5df-feee2ca14447; ' \
            '_ga=GA1.2.1424175566.1568020400; ' \
            '_mkto_trk=id:487-WAY-049&token:_mch-hackerrank.com-1568020400269-75078; ' \
            '_fbp=fb.1.1568020400679.1722645904; ' \
            '__utma=74197771.1424175566.1568020400.1568020411.1568725370.2; ' \
            '__utmz=74197771.1568020411.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none); ' \
            '_biz_uid=aca96a3b00d548e08eb62c3e3f88bf5e; ' \
            '_biz_nA=1; ' \
            '_biz_pendingA=%5B%5D; ' \
            'show_cookie_banner=false; ' \
            'hrx_candidate=BAhJImt7InRlc3RfaGFzaCI6IjZkamJjdGdsbDVoIiwiZW1haWwiOiJtaXF1ZWxwdWlnbWVuYUBnbWFpbC5jb20iLCJjcmVhdGVkX2F0IjoiMjAxOS0wOS0yMVQxNDo0Mjo1MC44MTVaIn0GOgZFVA%3D%3D--d280e3f7353c5e718b6074478aced782f1ab5da7; ' \
            'userty.core.p.6bd7b3=__2VySWQiOiI0M2Q2NDI5Yzc3YTRhN2YwMWI5ZGViMDQxZTM4NGRkOSJ9eyJ1c; ' \
            '_gaexp=GAX1.2.2u09ecQTSny1HV02SEVoCg.18292.0; ' \
            '_gid=GA1.2.1505727548.1577189209; ' \
            'hrc_l_i=T; ' \
             '_hrank_session=65210a8e26e55a2725b74db9d29e20fdb98cd3e2249e8973443ce6db5c12a9e14a15465e84bdb71c814793920c6e3b92634ee9d8e36792563874b9cc48ec9aba; ' \
            'user_type=hacker; ' \
            'h_r=university_recruiting; ' \
            'h_l=header_top; ' \
             'mp_bcb75af88bccc92724ac5fd79271e1ff_mixpanel=%7B%22distinct_id%22%3A%20%221e57a69d-f61d-439f-a5df-feee2ca14447%22%2C%22%24device_id%22%3A%20%2216d154c77247c-02e693c3de5c0c-7e2c6752-100200-16d154c7726b%22%2C%22%24user_id%22%3A%20%221e57a69d-f61d-439f-a5df-feee2ca14447%22%2C%22%24search_engine%22%3A%20%22google%22%2C%22%24initial_referrer%22%3A%20%22https%3A%2F%2Fwww.google.com%2F%22%2C%22%24initial_referring_domain%22%3A%20%22www.google.com%22%7D; ' \
                 'mp_86cf4681911d3ff600208fdc823c5ff5_mixpanel=%7B%22distinct_id%22%3A%20%2216d154cb0ec37d-0ac00a0b0ef8138-7e2c6752-100200-16d154cb0ed220%22%2C%22%24device_id%22%3A%20%2216d154cb0ec37d-0ac00a0b0ef8138-7e2c6752-100200-16d154cb0ed220%22%2C%22%24initial_referrer%22%3A%20%22https%3A%2F%2Fwww.hackerrank.com%2Faccess-account%2F%3Fh_r%3Dhome%26h_l%3Dheader%22%2C%22%24initial_referring_domain%22%3A%20%22www.hackerrank.com%22%7D; ' \
            'react_var=false__cnt2; ' \
            'react_var2=false__cnt2; ' \
            'metrics_user_identifier=318f5f-00e1551a1309dee6aa6946772f0862b191df1dae'
    try:
        req_not_primary = Request(URL_CODE_BY_HACKER_NOT_PRIMARY.format(hacker_name=hacker_name))
        req_not_primary.add_header('Cookie', MY_COOKIE)
        return urlopen(req_not_primary).read().decode('utf-8')
    except Exception:
        logger.debug("Request failed without PRIMARY: " +
                URL_CODE_BY_HACKER_NOT_PRIMARY.format(hacker_name=hacker_name))
    try:
        req_primary = Request(URL_CODE_BY_HACKER_PRIMARY.format(hacker_name=hacker_name))
        req_primary.add_header('Cookie', MY_COOKIE)
        return urlopen(req_primary).read().decode('utf-8')
    except Exception:
```

```
        logger.debug("Request failed with PRIMARY: " +
                     URL_CODE_BY_HACKER_PRIMARY.format(hacker_name=hacker_name))
    raise Exception("Imposible to establish connection")


def str2file(string: str, file_name: str, summary: dict) -> (None, Exception):
    path = 'samples/' + file_name + '.deepcode'
    logger.info("Persisting '{}'. Summary status: Persisted={}, Failed={}"
                .format(file_name, summary.get('persisted'), summary.get('failed')))
    try:
        with open(path, 'w') as file:
            file.write(string)
            file.close()
    except Exception as e:
        logger.warning("Unable to write '{}' in file.".format(file_name), e)
        raise e


if __name__ == "__main__":
    options = Options()
    options.headless = True
    summary = {'persisted': 0, 'failed': 0}
    try:
        logger.debug("WebParser Succesfully Started")
        driver = Firefox(options=options)
        parse_challenge(driver, challenge='capitalize', summary=summary)
        driver.close()
    except Exception as e:
        logger.error("Unable to parse challenge", e)
    except KeyboardInterrupt:
        pass
    finally:
        logger.debug("WebParser Finished. Persisted={}, Failed={}".format(summary.get('persisted'),
summary.get('failed')))
```

4. Homogenising and labeling source code data-set

4.1. Implementation: homogenising data-set

```
def add_main_function_call(dir: str, file_name: str, test_word: str):
    import regex as re
    function_def_pattern = re.compile('def .+\(.+\):')
    path = "{path}{file}".format(path=dir, file=file_name)
    with open(path, 'r') as file:
        s_file = file.read()
        # Check if solve function is in file but not called
        func_def = function_def_pattern.search(s_file)
        if func_def:
            # get_func_name where format is 'def foo(oof):' get foo
            func_name = func_def.group(0).replace("def ", "").split("(")[0]
            function_calls = re.findall(func_name+'\(.+\)', s_file)
            # If no more than 1 call to func_name(.*), meaning the definition, append a call to that function
            if len(function_calls)<=1:
                to_append = "\nprint({func}('{test}'))".format(func=func_name, test=test_word)
                appender = open(path, 'a')
                appender.write(to_append)
```

4.2. Implementation: homogenising, profiling, and labeling source code

```
import os
import subprocess
import logger

from numpy import mean as npmean

logger = logger.get_logger("Profiler")
expected_test_output = "Myword Is Here"
test_input = "myword is here"


def profile(dir="samples/", extensions=('.py')):
    for subdir, dirs, files in os.walk(dir):
        N = 10
        num_files = len(files)
        for iteration, f in enumerate(files):
            times = list()
            ext = os.path.splitext(f)[-1].lower()
            if ext in extensions:
```

```
        add_main_function_call(dir, f, test_input)
        for _ in range(N):
            out = strace_to_file(dir, f)
            out_lines = out.split('\n')
            test_output = out_lines.pop(0)
            try:
                assert (expected_test_output in test_output)
            except AssertionError:
                msg = "Wrong output provided by file '{}'. Obtained='{}', Expected='{}'."\
                    .format(f, test_output, expected_test_output)
                logger.debug(msg)
                break
            else:
                total_line = out_lines[-2] # get second last position (total)
                total_time = total_line.split()[1] # get float value of total time
                times.append(float(total_time))
    average_time = npmean(times)
    new_python_file_name = f.split('_')
    new_python_file_name[1] = str(average_time) # Put average time in name
    s_new_python_file_name = "_".join(new_python_file_name)
    os.rename(dir + f, dir + s_new_python_file_name)
    logger.debug("{curr}/{max} File '{f}' profiled and saved under '{o}'. "
                 .format(curr=iteration, max=num_files, f=f, o=s_new_python_file_name))

    if(iteration%10==0 and iteration != 0):
        pass
        #exit(1)


def strace_to_file(dir: str, file_name: str) -> str:
    b_error = bytes("Error", 'utf-8')
    path = "{path}{file}".format(path=dir, file=file_name)
    out, strace_file = execute_strace(path, test_in=test_input, python_v="python2")
    if b_error in out:
        out, strace_file = execute_strace(path, test_in=test_input, python_v="python3")
        if b_error in out:
            logger.error("Couldn't run python file: '{}'".format(path))
            if os.path.exists(file_name):
                os.remove(file_name)
    s_out = out.decode('utf-8')
    return s_out


def execute_strace(file_path: str, test_in: str, python_v: str) -> (bytes, str):
    strace_file_name = file_path.replace("py", "strace")
```

```
    cmd = ["strace", "-c", python_v, file_path]
    ps = subprocess.Popen(('echo', test_in), stdout=subprocess.PIPE)
    out = subprocess.Popen(cmd, stdin=ps.stdout, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    stdout, _ = out.communicate()
    return stdout, strace_file_name


def add_main_function_call(dir: str, file_name: str, test_word: str):
    import regex as re
    function_def_pattern = re.compile('def .+\(.+\):')
    path = "{path}{file}".format(path=dir, file=file_name)
    with open(path, 'r') as file:
        s_file = file.read()
        # Check if solve function is in file but not called
        func_def = function_def_pattern.search(s_file)
        if func_def:
            # get_func_name where format is 'def foo(oof):' get foo
            func_name = func_def.group(0).replace("def ", "").split("(")[0]
            function_calls = re.findall(func_name+'\(.+\)', s_file)
            # If no more than 1 call to func_name(.*), meaning the definition, append a call to that function
            if len(function_calls)<=1:
                to_append = "\nprint({func}('{test}'))".format(func=func_name, test=test_word)
                appender = open(path, 'a')
                appender.write(to_append)


if __name__ == "__main__":
    profile()
```

5. Implementation evaluation tool

```
from src.logger import get_logger

logger = get_logger('evaluation')

def analyse(original_tree, custom_tree):

    import json

    original_tree_dict = json.loads(original_tree)

    node_appearances_AST = dict()

    iter_tree(original_tree_dict, node_appearances_AST, type_key="_type")

    num_entities_ast = len(list(node_appearances_AST.keys()))

    num_nodes_ast = sum(list(node_appearances_AST.values()))

    ast_analysis = {'total_entities': num_entities_ast,

                    'total_nodes': num_nodes_ast,

                    'entities': list(node_appearances_AST.keys())}


    custom_tree_dict = json.loads(custom_tree)

    node_appearances_cAST = dict()

    iter_tree(custom_tree_dict, node_appearances_cAST, type_key="CAST_type")

    num_entities_cast = len(list(node_appearances_cAST.keys()))

    num_nodes_cast = sum(list(node_appearances_cAST.values()))

    cast_analysis = {'total_entities': num_entities_cast,

                     'total_nodes': num_nodes_cast,

                     'entities': list(node_appearances_cAST.keys())}

    logger.debug("Analysis AST: Different entities '{}', Total nodes '{}'".format(num_entities_ast, num_nodes_ast))

    logger.debug("Analysis cAST: Different entities '{}', Total nodes '{}'".format(num_entities_cast, num_nodes_cast))

    return {'ast': ast_analysis, 'cast': cast_analysis}



def iter_tree(d, entities_tree, type_key):

    for k, v in d.items():

        if k == type_key:

            if v in entities_tree:

                curr_value = entities_tree.get(v)

                entities_tree.update({v: curr_value + 1})

            else:

                entities_tree.update({v: 1})

        if isinstance(v, dict):
```

```
            iter_tree(v, entities_tree, type_key)

        elif isinstance(v, list):

            [iter_tree(element, entities_tree, type_key) for element in v if isinstance(element, dict)]




if __name__ == "__main__":

    #D1={"CAST_type": "Module", "CAST_body": [{"CAST_type": "Import", "CAST_body": [{"origin": "SYS"}, {"name": "pprint"}]}, {"CAST_type": "Import", "CAST_body": [{"origin": "UNK"}, {"name": "src.my_pretty.my_print"}]}, {"CAST_type": "FunctionDef", "CAST_body": [{"CAST_type": "arguments", "CAST_body": [{"CAST_type": "arg", "CAST_body": ["'list1'", "None"]}]}, {"CAST_type": "Loop", "CAST_body": [{"CAST_type": "Name", "CAST_body": [{"id": "'i'", "ctx": "Store"}]}, {"CAST_type": "Name", "CAST_body": [{"id": "'list1'", "ctx": "Load"}]}, {"CAST_type": "Expr", "CAST_body": [{"CAST_type": "Call", "CAST_body": [{"origin": "SYS"}, {"CAST_type": "Name", "CAST_body": [{"id": "'pprint'", "ctx": "Load"}]}, {"CAST_type": "Call", "CAST_body": [{"origin": "UNK"}, {"CAST_type": "Name", "CAST_body": [{"id": "'my_print'", "ctx": "Load"}]}, {"CAST_type": "Name", "CAST_body": [{"id": "'i'", "ctx": "Load"}]}]}]}]}]}]}

    D1 = {"CAST_type": "Module","tests":{"inner":1}, "CAST_body": [{"CAST_type":"Name", "CAST_body": [{"origin": "SYS"}, {"name": "pprint"}]}]}

    node_appearances = {}

    iter_tree(D1, node_appearances)

    print(node_appearances)
```

Glossary

Glossary Index

AI: Artificial Intelligence...10

LTH: Lund Tekniska Högskola...10

UPC: Universitat Politècnica de Catalunya...1o

AST: Abstract Syntax Tree...13

LLVM: Low Level Virtual Machine...14

json: JavaScript Object Notation...15

IDE: Integrated Development Environment...17

LLVM: Low Level Virtual Machine...14