Software Development Insights

Metrics for Developers' Profiling

Elpida Bantra and Madhura Kashikar

A thesis submitted to University College Dublin in part fulfillment of the requirements of the degree of M.Sc. in Business Analytics

Michael Smurfit Graduate Business School,
University College Dublin

September, 2017

Supervisors: Dr. Brendan Cody-Kenny and Prof. Michael O' Neill

Head of School: Professor Ciarán Ó hÓgartaigh

Dedication

I would like to dedicate my third dissertation to my family: Giannis, Evi, Vasso, Kira and Pike and thank them for all of their sacrifices and support.

Elpida

I would like to dedicate my dissertation to my family: Girish Kashikar, Dr.Jaishree Kashikar, Amruta Kashikar and thank them for constant support and encouragement.

Madhura

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Business motivation . . . 2

2 Literature Review **5**
- 2.1 Introduction . . . 5
- 2.2 Available Metrics . . . 5
  - 2.2.1 Metrics that measure the overall quality and maintainability . . . 6
  - 2.2.2 Communication Metrics . . . 7
  - 2.2.3 Network Metrics . . . 8
- 2.3 Metrics which Fail and Metrics which Succeed . . . 9
- 2.4 The Gap in the Literature . . . 11

3 Defining the New Metrics **12**
- 3.1 Designing the New Metrics . . . 12
- 3.2 Definitions . . . 20
- 3.3 Expected Insights . . . 21

4 Methodology, Tools and Demonstration with the Software **23**
- 4.1 Why GitHub . . . 23
- 4.2 How to pick a repository . . . 24
- 4.3 Biases . . . 25
- 4.4 Data Extraction . . . 26
- 4.5 Git . . . 27
- 4.6 RepoGrams . . . 28
- 4.7 Data Storing and Cleaning . . . 28
- 4.8 Making everything to work as one System . . . 29
- 4.9 Demonstration of Metric one and three with the Software . . . 31
- 4.10 Demonstration of Metric two with the Software . . . 32
- 4.11 Demonstration of Metric four with the Software . . . 33

5 Testing and Results **35**
5.1 Repositories 35
5.2 Facing Encoding Errors 40
5.3 Results 41
5.3.1 Comparative study for every metric and Summary 46
5.3.2 Comparative study of all the developers for all the metrics 50
5.3.3 Summary on Testing and Results 53
5.4 Theoretical Validation 53
5.5 Validation of the new metrics 55
5.6 Key Findings 55
5.6.1 Limitations 56

6 Conclusion **58**
6.1 Overall Review of the four new Metrics 58
6.2 Suggestions for Future Work 59
6.2.1 Relative Areas 59
6.2.2 PCA 59

Appendix A - Links for the code of the brand new metrics **62**

Appendix B - Metric results for Merlin and Return repositories **63**

References **68**

List of Abbreviations

1. **API** application programming interface
2. **ROI** ratio of imports
3. **ROME** ratio of maintenance effort
4. **ROAL** ratio of added lines
5. **RODL** ratio of deleted lines
6. **ROV** ratio of variability
7. **ROF** ratio of functionality
8. **ROLB** ratio of logic building
9. **ROEI** ratio of efficiency improvement
10. **LOC** lines of code
11. **SLOC** source lines of code
12. **PCA** principal component analysis

List of Figures

1. **Figure 5.1** Ratios of metric for repository Tacotron
2. **Figure 5.2** Ratios of metric for repository Sudoku
3. **Figure 5.3** Ratios of metric for repository Numpy Exercises
4. **Figure 5.4** Ratios of metric for repository Transformer
5. **Figure 5.5** Comparison of Ratio of Import metric among all developers
6. **Figure 5.6** Comparison of Ratio of Maintenance effort metric among all developers
7. **Figure 5.7** Comparison of Variability metric among all developers
8. **Figure 5.8** Comparison of Ratio of Functionality metric among all developers
9. **Figure 5.9** Comparative study of metrics for all developers

Acknowledgement

We would like to thank Dr. Brendan Cody for trusting us to work on his ideas and for always being there for us to discuss all our thoughts, help us to stay on schedule and successfully complete this challenging project. Dr. James McDermott for his guidance, on how an appropriate academic text should be.

In addition, we would like to thank Tonia, Giorgos, Anna and Kopal Srivastava for reading our text and suggesting changes.

Abstract

The purpose of this thesis is to define and design four novel software metrics. These metrics are helpful in finding the skills that various developers display in a software development project.

The key in the software development procedure is to understand a high level view of a project and the system's architecture. This information helps in finding the necessary building components and realizing which one should be optimized to reduce the total complexity.

The business problem is to gain this high-level view and at the same time reduce the complexity of the software development process.

The brand new metrics that have been analyzed and tested, help in solving this problem. The characteristics of the people that wrote the code are related with the code's complexity.

From the literature it is known that metrics are difficult to apply and can fail. Hence, four well designed novel ratios, that are independent from the project's size are designed and validated.

These metrics provide new software development insights like understanding adaptability and changes in coding behavior of each developer. Hence, opening new perspectives of tackling this business problem. Furthermore, it is known in the literature that approximately 15 percent of the developers are core and the rest are peripheral developers. This fact has been confirmed in this study. Also, it has been observed that associate developers work on debugging, complexity reduction of the code and react easily with unseen code. Core developers bring novelty in the code and have deep understanding of the problem statement.

Chapter 1

Introduction

The objective of this chapter is to frame the topic and explain the business motivation.

1.1 Context

Due to the immense growth of the software development industry, we observe the presence of many tools and platforms, that helps developers to do a better job. As more tools are being produced to offer help to the developers, an effort to understand their logic and use, is also taking place.

The tools number is increasing dramatically, but the complexity of the software development process is not decreasing. In order to achieve that, we need to gain a high-level view of the software development process. This concerns the architecture of a system that can be used for developing a software product. This information delivers a framework of a complete system, thus helps finding the main components that would be developed for the product. The question is to understand the size of the problem, as well as the subsystems or the subcomponents. Also, the series of actions aiming at integrating these parts to a system might be a risky or a challenging undertaking.

A high-level view adds the required specifics to the project description so as to identify a suitable model for coding. The high-level view depicts the structure of the system. The structure may comprise the database architecture, application architecture (layers), as well as the navigation through the software and technology architecture. Since, it is difficult to have this idea right at the beginning of the project, it becomes difficult to identify what set of skills

is required for its completion and success. As a consequence, metrics are believed to be crucial in contributing to individuals profiling for the business area.

According to Forbes, the US Department of Labor confers that the price of bad hire is at least 30 percent of the employee's first year earnings. Given this, it is important for a company to identify the skill-set of an employee correctly; so, if a company needs to recruit programmers, a software that identifies each candidates skill-set from his/her participation and contribution on projects elaboration would be very helpful.

Furthermore, recruiting programmers based on a company's projects demands, will possibly have a serious impact on the quality of the software development.

Version control systems like GitHub, contain the complete project history/-data, along with information about individual contributor's work for the whole period of development. It is believed, that this data, along with the new metrics that have been designed, will surely contribute to the understanding of the skills that each part brings in to the development process. More specifically, the information in the open source code repositories (public repositories), where multi-developer projects are visible, is of high importance. All of this free data can be used to gain insights about the software development process.

1.2 Business motivation

In 1967, Conway stated, that "The Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

It is believed that understanding of the process of development and developers network, is the key to comprehend the companys structure and to overcome organizational problems about operations, so as to achieve a higher business turnover. The success of a company's software products and services is largely dependent upon the work its software developers do. The quality of their work entirely depends on the capability of the employee to execute the task efficiently. Moreover, as there are long running software projects, more information concerning the skills required to deal with them in the most satisfactory way, would be very useful. In addition to all these, having an insight about the skills that each programmer has, the formation of the most suitable team for the project would be easier for the company. This procedure shows the

skills that a particular programmer excels in.

Depending upon the projects demand of collaborative dynamics, it will managers to build good teams and individuals to improve their self-awareness. The purpose is not to form a judgment concerning each persons contribution, but to create metrics that identify different type of contributors. Even though the the count of lines added or deleted is observed , it is not done to compare with co contributors or to realize who has the largest count, but so as to observe the quality of the executed work. The main reason is that a programmer can write a long code which is computationally less efficient than another piece of code written by his/her peer, which may be less in terms of number of lines, but capable of performing more efficiently. Thus, most of the authors do not value the contributors according to the amount of contribution based on a count of lines.

Furthermore, a software that could return insights about the character and the experience of a programmer from a piece of his work, could be used later in the hiring process. Instead of passing through different stages for a job application, such as interviews with the HR department and team assessments, a programmer could be evaluated directly from his coding.

More details about how these metrics are going to help the business procedure are given as follows.

A. At *team* level:

- **Decide the balance of the team:** Depending on the skill set required and thus choosing people with similar skills as required by the project. As presented later on in the second chapter, at the beginning of the project it is difficult to get the entire outline of the project in terms of functionalities to be implemented, size of the team needed, or technology architecture.

  Thus, it is difficult to choose a developer who has proficiency in many languages or to go for an individual developer, because, if it is needed to implement the functionality in the various modules of the project simultaneously, it will be difficult for the same person to do so. A projects processing may implement a database system, the security of the system, the user-interface and many more such functionalities. If according to the projects deadline, any of them must be carried out at the same

time (parallel), then even though a developer in a team is aware of the technology needed to develop it, yet due to parallelism in implementation time line project manager will need to include more people with these skills to fulfill the work.

B. At *individual* level:

- **Every programmer, will be able to understand how he/she contributed as part of the overall team and improve his work by comparing it with their peers.**
- **Understanding their strengths and weaknesses**: by identifying the projects at which they are good and what types of problems they would like to work on in the future, are part of their personal development.

  These strengths and weaknesses can be identified by the number of import statements used by a developer in his/her code. It is possible to get insight about a developer's knowledge in particular programming language from these import statements, as they show his/her understanding of it. Also, the ratio of the number of non-trivial changes made in particular code by a developer to the trivial changes made in other projects, can be studied. This ratio shows again his understanding of the language and of the topic.

- **Understanding how fast they can learn and adapt**: as a first step for self-improvement and which is the best peer to take help from. When the commit history of a contributor is observed for the time period of the development, depending upon whether the changes contributed are significant or insignificant, their ability to learn can be assessed.

  If the changes remain insignificant, there is no improvement, however, if the changes grow from insignificant to significant it can be said that there is self improvement.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, an attempt is made to categorize the different metrics that were used for the analysis of the software development. Also, the evolution of the topic and the philosophy of the researchers at each time to search for a certain type of software metrics is analyzed. For this purpose, sixteen articles from 1968 to 2016 were gathered. After studying them it can be noticed that there are papers for the quantitative evaluation of different properties, either they are about the software or about the team that built it. There is no reference of metrics that could measure statistics for the programmers individually. That is the reason why it is important to open a discussion in this research topic and define new metrics that study the software with a totally different way. New metrics means new insights for the software development process, which means reduction in its complexity. Specifically, a new category of metrics will give a new perspective of studying the software.

2.2 Available Metrics

Below, the literature is organized in metrics categories. It must be underlined here that there are numerous categorizations of software metrics, but only the metrics related to the topic are studied. They are organized in categories according to the motivation of finding them. The purpose is to highlight how the focus of the researchers changed over time and by studying metrics in isolation turned to studying metrics in networks. The list with all the available metrics is in the link below:

http://www.aivosto.com/project/help/metricssample/metricslist.html

2.2.1 Metrics that measure the overall quality and maintainability

In the first category of the literature review, articles with metrics, that measure the quality [1],[2] and the maintainability [3] of the code, are presented.

Various software metrics are available, that measure the maintainability and the overall quality of a project, but they do not reveal much about the team that built the source code.

These metrics, are the first ones to be studied. In this point, it is significant to explain why these metrics are important and were initially studied.

Consider a scenario where a company has completed development of a software product. Some parameters that show success are: within the budget, on time delivery and expected performance. Are these parameters enough to consider that the project has great a quality and a maintainability? The answer is no, because there are more parameters that have to be checked, such as the easiness of the adaptability of the program, the easiness to integrate with other programs, the easiness to understand it, use it and modify it.

Suddenly, there are so many questions that have not been answered and the managers are exposed to them when the distribution of the product begins and mistakes like these are spotted. It would be more expensive to correct them later than to predict them before the distribution. Furthermore, the whole program can be rejected from the market. Hence, it is obvious that just like testing, the measurement of these parameters is of high importance.

Their common characteristic is that these metrics study only the software. In the same category, papers that measure the validation of the proposed metrics [4] have been added, as validation is of high importance, as well.

The first effort for the identification of metrics, that achieved the evaluation of the software quality, was in 1968 by Rubeyand and Hartwick [5]. Below are given metrics that can be used for measuring the quality [1]:

1. **Structures**: for measuring the average size of program modules.

2. **Robustness**: for measuring the fraction of statements with potential singularities (divide, square root, logarithm, etc).

In 1987, Robach proposed to predict the maintainability, with software metrics [6]. Below, some metrics that can be used for predicting the maintainability are presented [3].

1. **nom**: Number of methods
2. **size1**: Number of semicolons per class
3. **size2**: Number of methods plus number of attributes
4. **change**: Number of lines changed per class in its maintenance history

In this category it is becoming more clear which questions and needs led the software engineers to search for metrics that control costs and schedules. DeMarco stated that "you cannot control what you cannot measure" [7]. There is a need of better understanding the software to reduce its complexity and control it, which leads to the need of metrics, which measure the software in a different way.

2.2.2 Communication Metrics

In this category, there are more recent articles that have been published after 2006. These are about the association between the metrics and the developers' team. Slowly, the research on this topic is moving from searching for metrics that give insights only for the code to searching for metrics that characterize the teams that created it.

In this category, the articles demonstrate communication metrics which help to gain insights for the above purpose [8], [9], [10] and validate the classification of the developers into two categories. Some of the metrics that are measuring the communication among the developers during the development of a project are:

1. **Interaction frequency**: The degree of interactions between two developers.
2. **Computing commit coreness**: The number of core files in the commit to the total number of files in the commit.

Furthermore, very related to this work are papers with metrics that examine what people do and how developers are organizing in open source software projects.

In 2016, Kangning Wei et al contributed with their paper [11] in politeness theory, that supports that core and peripheral developers use more positive politeness strategies rather than negative ones. The model that holds on the description of the structure of the developers in these open source repository projects today describes that there is no hierarchy, but a core of developers (usually the ratio of them is 15/100) who are the top contributors. The rest of them are the peripheral developers, who are working at a lower level which demands less technical skills. It is an open question here if the peripheral developers because of that do have less technical skills or not. This is something that has to be investigated.

More information is available about how a group is distinguished into two different groups of developers [12]. In Crowston et al (2006), three different analysis approaches are compared in order to find the best for the identification of the core developers group.

2.2.3 Network Metrics

The researchers understood in 2016, that these systems should not be examined in isolation, but in the socio-technical networks where they were created. Now, there are network metrics available and with them the developers are classified into two categories [13]. Also, there are articles that study the structural software complexity and the socio-technical networks [14]. Articles like these explain why every piece of information that is gained for the process helps to reduce the complexity of the software. All the components that are associated with the process must be studied separately and in combinations. Below, some metrics are presented that examine the communication among developers from a network perspective. In addition metrics that help in distinguishing the contribution in the changes between two different groups, the technical core and the peripheral one are presented.

1. **LOC**: Lines of code count, lines contributed by each developer.
2. **The sum of added and deleted lines**. The core developers are responsible for the majority of changes.

3. **Number of commits**: It is the number of commits a developer has authored, a commit represents a single unit of effort for making a logically related set of changes to the source code. Core developers achieve a higher commit count than peripheral developers.

4. **Number of emails sent**: Mail count is the number of mails a developer contributed to the developer mailing list. Core developers participate more intensively and consistently.

5. **Eigenvector centrality**: Is the global importance of a developer (high eigenvector centrality have the developers who have many connections in total and connections with developers that also have a big number of connections).

6. **Hierarchy**: Is observed in networks that have nodes arranged in layered structure, such that small cohesive groups are embedded within large and less cohesive groups.

2.3 Metrics which Fail and Metrics which Succeed

Here, is examined previous work about what characteristics have been noticed to make a metric to fail or success.

According to Norman E. Fenton and Martin Neil (1998), the mis-application of software metrics is mainly reasoned by the fact that they are usually used **in isolation** [15]. It can be noticed that the papers in the last two categories given above, where there is an effort to study the metrics in networks, are more recent. This is because around 2000 the scientists realized that after 30 years of studying and producing software metrics, it is better to study them in networks.

Another key that differentiate the academic metrics, which had success in the industry and were used by the companies, is their **simplicity**. The industrialists find it easier to understand and collect LOC (lines of code) type metrics and reject esoteric metrics [15]. This had as a result to create metric models where two elements are missing: the uncertainty and the combination of different evidences.

Simplicity is a factor of successful metrics, because as Munson and Khosghoftaar asserted in 1992:

> "There is a clear intuitive basis for believing that complex programs have more faults in them than simple programs."

Additionally, the **validation** of a metric makes it more attractive to be used. Since, the esoteric metrics have not been validated more than the simple ones, there is less motivation to be used [15].

Another factor for failing of metrics is their **relevance** with the industrial needs. When the academic metrics have mainly focused on small projects and they need the calculation of parameters which they cannot be measured in practice, is reasonable why they are not applicable in business.

In 1994, Glass stated, that

> "What theory is doing with respect to measurement of software work and what practice is doing are on two different planes, planes that are shifting in different directions."

wanting to prompt this irrelevance.

In general, the software metrics have some characteristics that make them not very practical. The main characteristics that concern this work are:

1. They are not independent on programming language and/or underlying platform.
2. It is difficult in some cases to understand which software metric should be applied in order to achieve a purpose.

Since a software metric cannot be designed to measure features in projects that have been built in different programming languages, the first task for someone that wants to build a software metric is to decide in which programming language it will measure characteristics of the project.

The next feature that must be taken under consideration is metrics' clear purpose of existence. Software metrics that do not have anything new to give academically or industrially or are not applicable must be avoided to be produced.

2.4 The Gap in the Literature

There are metrics for measuring the overall quality and maintainability of the code, communication metrics and network metrics. It is noticed that a category of metrics that help to profile the individuals from their contribution is missing. Until now, not a single metric for this purpose has been proposed.

Moreover, the difference between the academic and the industrial success of a metric has been studied, in order to be explained why so many discovered metrics are not used from the business world.

It is declared that a gap in the literature that has been spotted is the identification, definition and demonstration of new metrics that support the building, in the future, of models that give insights for the individuals.

Taking into consideration that successful industrial metrics are metrics that were not studied in isolation, were validated, were simple and relevant with the industrial needs, the business problem is better framed. In conclusion, the business problem that is addressed is the creation of successful software metrics that profile individuals and are easy to adapt by the industry.

On top of that, it has been reported [16] that even though Python is a very useful language for embedded systems, there are only a few and limited metrics proposed in online articles [17],[18],[19] for it. There is a rich literature about software metrics that evaluate the overall quality of a project written in C++, Java and in other object oriented languages and for technologies such as XML and Web services [20,21,22], but there are no metrics until 2010 [16] that are for Python. The increasing popularity of Python and the fact that it is a comfy programming language for software development show why there is a demand for software metrics for Python.

Chapter 3

Defining the New Metrics

3.1 Designing the New Metrics

There are different kinds of metrics according to their values:

1. absolute - eg this project is 100.000 lines of code long
2. ratio - eg the complexity of the code is twice as big as it was before the last contributor
3. interval - eg a programmer's capabilities are between the 80th and 90th percentile of the population ability
4. ordinal - eg a programmer's capabilities are low/average/high
5. prediction - eg the effort that is required to build a project according to its functionality
6. nominal - eg the languages that were used to build a project were Python,Java and C++
7. direct - eg number of lines of code (LOC)
8. indirect - eg the ratio of written code by one programmer is the written code from this contributor / the written code by all contributors

The new metrics are ratios, because their normalization plays an important role. The value of each metric should give some information independently of

the size of the project or the number of contributors.

Proposed Metrics:

1. **Ratio Of Imports**

   Its purpose
   This metric gives an overview of how many libraries a programmer has imported and how many classes or functions he/she has written or called. This shows how much a developer is well aware of the libraries and the packages that are available in a particular programming language.

   Also, gives information about the count of lines of code that he/she has written from scratch or has reused from previously written code.

   What it measures
   In order to measure this metric, the number of the times that a statement/library/API is appearing in the source code is measured. This can be done by text analysis, searching for the words "+import", "+def" and "+class" in the source code.

   The symbol "+" is used before the key words in text analysis. This symbol shows that this line was inserted by a developer and the first word of the line code was the corresponding key word. Like this it is avoid to measure lines where the key words were deleted or used as comments.

   Assuming that TI is the number of the total imported and called statements, libraries and APIs, the denominator of this ratio is calculated.

   The next step is to calculate the number of contributors N in a source code, these are numerators of the ratio. Assume that $n_1, ..., n_N$ are the variables that correspond to the number of the times a developer imported or called a statement, a library or an API. The ratio for each contributor results by dividing the numbers $n_1, ..., n_N$ by the number TI.

   Naming this ratio as **Ratio Of Imports**, it is described mathematically from the type below:

Mathematically
ROI(i)= $\frac{n_i}{TI}$, where $i = 1, ..., N$ and N is the number of contributors

Then the following equality holds

$\sum_{i=1}^{N} n_i = TI.$

2. **Ratios of Maintenance Effort**:

Its purpose
The LOC metric has been under discussion since 1976 [1]. This metric has been widely considered to measure the maintainability of the code. Lines of code comes basically under the size metrics. Large lines of code is associated with more faults in the code.

With the help of these two novel ratios it is attempted to understand to what extent the changes have been made by the developer in the code. Since only the changes that have been made in the commits are examined, only the changes that were made in files with .py extension will be taken under consideration .

The ratios of maintenance effort metric has a dual nature. To assess from the added lines by a developer his/her contribution and from the deleted lines his/her effort of fixing the bugs or pruning the code.

What it measures

Two ratios are calculated:

2A: Ratio Of Added Lines

In order to measure this metric, the number of the times that a line added is measured. This can be done by using Subprocess.

Assuming that TI is the number of the total added lines, the denominator of this ratio is calculated.

The next step is to calculate for the number of contributors N in a source code, the numerators of the ratio. Assume that $n_1, ..., n_N$ are the variables that correspond to the number of the times a developer added a line. The ratio for each contributor results by dividing the numbers $n_1, ..., n_N$ by the number TI.

Naming this ratio as **Ratio of Added Lines**, it is described mathematically from the type below:

Mathematically
ROAL(i)= $\frac{n_i}{TI}$, where $i = 1, ..., N$ and N is the number of contributors

Then the following equality holds

$\sum_{i=1}^{N} n_i = TI.$

2B: Ratio Of Deleted Lines

In order to measure this metric, the number of the times that a line is deleted is measured.

Assuming that TI is the number of the total deleted lines, the denominator of this ratio is calculated.

The next step is to calculate for the number of contributors N in a source code the, numerators of the ratio. Assume that $n_1, ..., n_N$ are the variables that correspond to the number of the times that a developer deleted a line. The ratio for each contributor results by dividing the numbers $n_1, ..., n_N$ by the number TI.

Naming this ratio as **Ratio of Deleted Lines**, it is described mathematically from the type below:

Mathematically
RODL(i)= $\frac{n_i}{TI}$, where $i = 1, ..., N$ and N is the number of contributors

Then the following equality holds

$\sum_{i=1}^{N} n_i = TI.$

3. **Ratio of Variability**:

   Its purpose
   This proposed metric measures the amount of novel code developed by a developer. This can be measured according to the variability of the libraries/functions/classes that he/she brought in the code.

   Studying the new code in a source code and referring to the developer that first added this code, this metric not only shows how much a developer is aware of the different libraries and packages available in a particular programming language, but also how comfortable he/she is in switching among different techniques and functions.

   In addition, this directly gives information on the count of lines of code that are written from scratch by him/her.

   The measure of this metric is a subset of the ratio of contribution, as it does not measure the whole contribution of a developer but only the new things that he/she brought in the code. This time the novelty is studied.

   What it measures
   By analyzing the text in the revision files all the different elements that were imported in the source code by each developer can be found.

   In order to measure this metric, the number of the times that a statement/library/API is appearing in the source code is measured. This can be done by text analysis, searching for the words "+import", "+def" and "+class" in the source code.

   The symbol "+" is used before the key words in text analysis. This symbol shows that this line was inserted by a developer and the first word of the line code was the corresponding key word. Like this it is avoid to measure lines where the key words were deleted or used as comments.

Reading the code through the revision files, it becomes clear when the very first time the import of a new element is found and the developer who did it.

Assuming that TI is the number of the total imported statements (which are imported only once), libraries and APIs, the denominator of this ratio is calculated.

The next step is to calculate the number of contributors N in a source code, the numerators of the ratio. Assume that $n_1, ..., n_N$ are the variables that correspond to the number of the times that a developer imported for the first time a statement, a library or an API. The ratio for each contributor results by dividing the numbers $n_1, ..., n_N$ by the number TI.

Naming this ratio as **Ratio of Variability**, it is described mathematically from the type below:

Mathematically
ROV(i)= $\frac{n_i}{TI}$, where $i = 1, ..., N$ and N is the number of contributors

Then the following equality holds

$\sum_{i=1}^{N} n_i = TI.$

4. **Ratios Of Functionality**:

Its purpose
The first thing that should be cleared here is that this metric consists of two ratios and the bigger is examined.

The first ratio is about how many of the number of the classes and functions that a developer uses are first written by him. The second ratio is about how many of the number of the classes and functions that a developer uses are written by him/her and by the other developers.

This can be done by text analysis, searching for the words "+def" and "+class" in the source code.

The symbol "+" is used before the key words in text analysis. This symbol shows that this line was inserted by a developer and the first word of the line code was the corresponding key word. Like this it is avoid to measure lines where the key words were deleted or used as comments.

This metric gives an comparison between the new code that a developer has contributed to the project and his/her contribution on extending the logic of the code.

The rewritten code (second ratio) means modifying the code or adding new work to already written piece of work or debugging. The developer works more either on his/her own work to improve the quality of his/her code or on the work of someone else's code. This metric is useful to understand if the developer is core or peripheral.

This metric is useful to understand if the developer can contribute new ideas or is better at improving the present code or present logic.

Through the history of commits it is obvious which developer first coined the logic from the time stamp and then how the changes were made.

What it measures

Two ratios are calculated:

4A: Ratio Of Logic Building

In order to measure this metric, the number of the times that a programmer brought a new function or class in the code is calculated.

This can be done by text analysis, searching for the words "+def" and "+class" in the source code.

The symbol "+" is used before the key words in text analysis. This symbol shows that this line was inserted by a developer and the first word

of the line code was the corresponding key word. Like this it is avoid to measure lines where the key words were deleted or used as comments.

Assuming that TI is the number of count of the total times each function and class is written first time in the code by the developer , the denominator of this ratio is calculated.

The next step is to calculate the different number of contributors N in a source code, the numerators of the ratio. Assume that $n_1, ..., n_N$ are the variables that correspond to the number of the times that a developer wrote a new function or class in the code. The ratio for each developer results by dividing the numbers $n_1, ..., n_N$ by the number TI.

Naming this ratio as **Ratio Of Logic Building**, it is described mathematically from the type below:

Mathematically
ROLB(i)= $\frac{n_i}{TI}$, where $i = 1, ..., N$ and N is the number of contributors

Then the following equality holds

$\sum_{i=1}^{N} n_i = TI.$

4B: Ratio Of Efficiency Improvement

In order to measure this metric, the number of the times that a programmer edited a function or a class is calculated.

Assuming that TI is the number of the total times each function and class was edited by any developer, the denominator of this ratio is calculated.

The next step is to calculate for the number of contributors N in a source code, the numerators of the ratio. Assume that $n_1, ..., n_N$ are the variables that correspond to the number of the times that a developer edited a function or class in the code. The ratio for each contributor results by dividing the numbers $n_1, ..., n_N$ by the number TI.

Naming this ratio as **Ratio Of Efficiency Improvement**, it is described mathematically from the type below:

Mathematically
ROEI(i)= $\frac{n_i}{TI}$, where $i = 1, ..., N$ and N is the number of contributors

Then the following equality holds

$\sum_{i=1}^{N} n_i = TI.$

If the **ROLB ratio** is bigger then the developer has huge contribution in logic building and if the **ROEI ratio** is bigger then the developer has huge contribution in improving the efficiency of the code.

3.2 Definitions

In this section, definitions for the novel metrics are proposed.

1. **ROI metric** is defined as the ratio of the number of imports and calls of libraries, packages, functions and classes by one developer to the total number of imports and calls by all the developers.

2. **ROME metric** is defined as the two ratios, ROAL and RODL ratio, that are ratios of the number of added/deleted lines by one developer to the number of added/deleted lines by all the developers, respectively.

3. **ROV metric** is defined as the ratio of the number of imports of new libraries, packages, functions and classes in the code, by one developer to the total number of imports by all the developers.

4. **ROF metric** is defined as the two ratios, ROLB and ROEI ratio, that are:

   1. ROLB ratio: the number of the functions and the classes that a developer has worked on and they were written by the developer to the total number of different functions and classes that were written by all

the developers.

2. ROEI ratio: the number of the functions and the classes that a developer has worked on and they were written by another developer to the total number of the functions and the classes that were committed by all the developers.

3.3 Expected Insights

These new metrics are expected to give insights by answering the following questions:

1. How much does a persons developing ability change over time?

   This is useful to understand the progress in ones programming abilities. Especially, this is very important in those scenarios where the developer has shown significant improvement in coding abilities.

   Managers that want to create a team for a project with long time frame of development would be interested for developers that show this ability.

2. Which metrics are correlated?

   This will give information about how the metrics are interrelated with each other after the metrics have been tested.

3. How adaptable is a developer?

   This is useful to understand how quickly or frequently does a developer modify or interact with new unseen code.

4. Does the role that a developer plays in a project changes over time?

   Various contiguous periods over a time period of a project in question [13] can be examined. After comparing the contribution of a developer in different projects and checking if he/she has selected to work on different ideas, using different programming language and providing new novelty,

someone could measure his/her adaptability. For this purpose, all of the new metrics give insights.

On combining these novel metrics with some of the already existing metrics, a better understanding of the software development process could be gained. Now, the information about the characteristics of the software that someone wants to build is available along with the characteristics of the developers that are going to build it. Hence, a predictive model about the building time, the total complexity and the successful integration of the project could be built.

Chapter 4

Methodology, Tools and Demonstration with the Software

In this chapter, the number of tools that were used for the purpose of this work can be found, and also the reason why the authors picked them. In addition, tools that could be used in the future, not only to optimize this work but also to expand it, are presented here. Methodologies and tools are combined in order to demonstrate the four new metrics with the software. In the end, the complete logic of the code that was built for these four metrics has been explained.

4.1 Why GitHub

The data source for this study was GitHub Repositories and the programming language for the implementation was Python. The files that give the information about major interactions among the developers and their interaction with the code are of high importance. For example, the log of commits made by the contributors.

GitHub in recent years has carved its space as the default coding platform, by rapidly evolving and providing a simple platform to developers to develop and keep software projects.

Apart from catering to the developer's need, in recent time, GitHub is, also, catering for the non-developers activities. It was attempted to mine GitHubs

revision files for respective chosen repositories.

Overall, the below mentioned activities were executed:

- Study GitHubs developer platform and different commands to understand the functionalities that can be served using those commands.
- Study the graphs that were created after the results were obtained.

A GitHub user's public profile might consist of one or more repositories. These repositories can be the ones that have been created or have been forked from another user. The forked projects are also visible in the public profile. Apart from forking projects, a user can bookmark a project, which is known as **stargazer** of the project. The profiles, where the user is a contributor were examined and the ones where the user is a stargazer were avoided.

In GitHub, there is a score system with stars, which enhance the understanding of project's popularity. Only public repositories were studied.

4.2 How to pick a repository

Certain things have been taken under consideration, when picking an open source repository, such as:

1. **The Number of Contributors**: If there is only one contributor in the repository, studying the revision files does not yield any useful results, as all the ratios will be equal to one. Whereas if there are multiple contributors, the results obtained in terms of ratios can be studied to understand correlations among the various metrics and the different characteristics that the programmers show.
2. **The Number of commits**: The changes in the source code show the participation of each contributor. Hence, to understand the development cycle, its important to study the commits. Higher the number of commits, there is possibility to get a better understanding of the development process.

3. **Programming Languages**: The repositories of projects that have been developed using the python programming language are suitable to obtain accurate results out of these metrics. Other object-oriented programming languages will also give results if they have the same key words, but the authenticity of those results may be questionable.

4. **Status of the repository**: Whether the repository is active or inactive. Inactive repository means that the development work is finished and that there is a revision file which may not change any further. By studying these kind of revision files, the ratios that obtained will not change any further and the conclusions made can be finalized. For the active repositories, the ratios obtained might change over time and the conclusions that have been made on repositories may also require alterations until a final revision file is obtained.

Moreover, the big repositories are very interesting, where there are many contributors. But all of these can be checked after finding the repositories and then deciding to measure them or not. Starting from one repository and then finding relative repositories with following the developers across projects, would be a meaningful method. The purpose is to detect the developers' role and characteristics in different works in order to avoid biases and see if they have changed over time.

Another aspect that is important is to identify the nature of the repository. Many projects use the repositories just as a back-up device, or for committing final or working code. A repository like this does not contain sufficient data about the developing process, hence it needs to be identified in order to avoid erroneous or biased findings.

4.3 Biases

The problem here is that examining only one project for one developer and trying to reach conclusions means that biases are more likely to occur. The term 'biases' refers to the fact that some results might have occurred by chance and not by fair objective.

For example, it is logical for a developer of a project, in the beginning of his/her career to have different values for these ratio metrics, than in a project after having obtained some years of experience. Understanding and becoming more familiar with programming and new technologies is expected to increase

some metrics like for example the ROF metric.

Also, a developer can be a core developer in one project and associate in another, hence the programmer will have ROI and ROV metric big and small ratios, respectively. So, biases are likely to occur here, too.

As far as the period of developing the source code is been concerned, it is very important for the profiling. The key here is to not only understand the current programming situation of a developer, but also his/her evolution through time.

The role of a developer in each project is of utmost importance. A way to avoid biases is to build a model which will take under consideration the different roles that a developer can play in a project. More specifically, including all the roles that a developer could have in a project and then giving the ratios that he/she has scored on the projects that he/she built, the coefficients of the model will change. So, having a model that consists of two factors, core and associate roles, after deciding if in a particular project a developer had a core or peripheral role, the calculated ratios that occur will be under the coefficient of the corresponding factor of the model.

In the current study, in order to avoid biases in the results that classify individual contributors, the projects chosen were diverse. Diversity was considered in terms of source lines of code, number of developers, days since first commit, stars and programming language.

4.4 Data Extraction

Data extraction is core to this research process. Python libraries were used in data extraction. The data extraction for this research was implemented using the following Python library and a python module:

1. Pygit2
2. Subprocess

Software Repository Mining
The data extraction from the repository can be called as software repository mining. The revision files obtained for each repository to be studied have been mined. The two approaches that were used are explained below:

1. **Subprocess in Python**:
   The subprocess enables to run the command line commands from a Python program. There are various commands available to obtain the git log which shows the contribution in the code by each developer. This is needed to obtain the diff patch for each commit, which is very important for gathering the required data to design the metrics.

2. **Pygit2 Library**: The Pygit2 library becomes very useful to extract the diff objects, the developer, their information and the commit id. All the data is stored in object and so working with Pygit2 is helpful.

Revision Files:
Using the revision files, the user can trace the changes that were made by contributors line by line and see the way in which the parts of the file were developed over time. In GitHub, the user can see this in blame view. Locally, this file can be accessed from the command line.

4.5 Git

Git is a version control system and works in a decentralized fashion. It is different from GitHub in the following way: GitHub is a platform for Git repositories, while Git is a tool. GitHub is the service for the project using Git.

In Git, developers can have local copies and can commit local changes, without any need of being connected to a centralized server. Everything is done locally in a local Git Repository. Later, these commits can be pushed in GitHub. These pushed commits are stored in a remote repository (special online repository) from where they can be pulled anytime. Any local file can be synchronized with a location online (GitHub repository).

There are different kinds of branches in GitHub for avoiding the overlapping of work among the developers. Having a common master branch (main codebase) each developer can create a local branch in order to work in isolation. Then, the developer writes commits within this local branch and pushes it in GitHub. There, creating a pull request, explains the changes that this new branch contained in the master branch. The rest of the authors decide to

merge this branch to the master branch (or not) and then the isolated commits of this author are added to the top of the master branchs history. After this, the developer goes to the master branch of his/her local machine and can pull down these commits.

Git as an open source control system is often used for Python projects. One of the main characteristics of Git is that it works well with text files. The new metrics are developed with the latest version of Git (version 2.13.0) and the operating system command line.

4.6 RepoGrams

There is, already, a tool which uses different software metrics for examining the revision files of a repository. This tool is called Repograms and as it is explained in GitHub, can help in gaining the following insights:

- **who made commits**
- **how often and how much code did their commits change**
- **the number of programming languages used in a project over time**
- **how often developers changed code across different modules (directories)**
- **the use of branches in the repository**

In addition, an important property of RepoGrams is the option to add a new metric, so the new proposed ratio metrics can be added with other software metrics. This tool wasn't used for the purpose of this dissertation, but it is mentioned here as a helpful tool for future work. It could be used on building models with the combination of different metrics.

4.7 Data Storing and Cleaning

Storing
To store the repositories locally, they are cloned from Github. This is done to avoid latency that may occur if the data is to be fetched each time from GitHub or because of the need of being connected to the network. This can

be achieved with git clone, which is a command that creates a clone of the repository into a new directory.

Creating a clone file
Below is given the procedure that has to be followed in order to create a clone file:

1. **Go to the main page of the desired repository in GitHub**
2. **Click the "download or clone" option, which is under the repository's name**
3. **Copy the clone URL for the repository, which is in the clone with HTTPs section**
4. **Open the terminal in your computer**
5. **Select the desired for storing working directory**
6. **Type the command: git clone " the URL "**

Git and GitPython
The important task is to access the repository and study it. To perform such activities, Pygit2 library is helpful. Pygit2 provides object model access to the repositories.

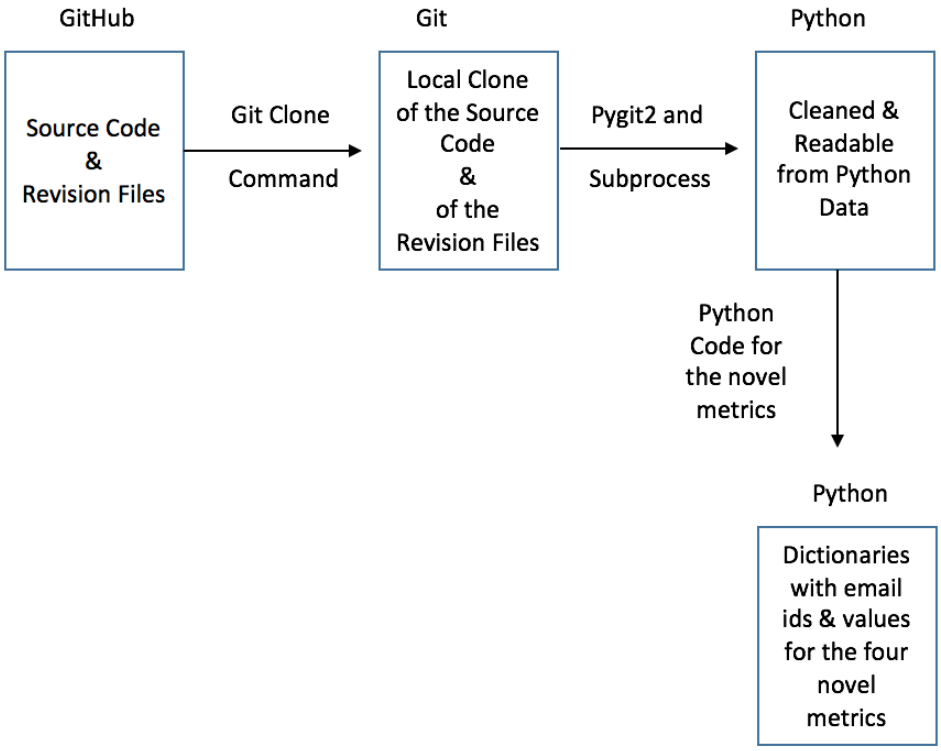
By initializing a repository object access to the repository is created. This allows access to the commit history. This history is useful to understand the interaction between developer and the repositories.

Cleaning
Pygit2 library is used to make the Git revision files readable. Using the pygit2 diff object, the code patches for every commit has been obtained. These code patches are saved in text files. The cleaning of the files is done by performing text analysis on the output files.

4.8 Making everything to work as one System

The following sketch shows how all the above are combined in one system.

A connected system was designed, where the data, which is available in the clones, is imported into the system by running python code on the local git repositories that generate the diff patch. Further, this diff patch is parsed and the metrics are calculated. The output will be the four values of the four metrics for all the developers of the project.



After cloning a repository locally and running the Python code, results are extracted in text files.

4.9 Demonstration of Metric one and three with the Software

Starting with the first and third metric, since they are closer in logic, what has been added in the source code by each developer is examined. The key here is to study what was imported in the code, by whom and in which frequency. This knowledge can give information about the developer's profile.

It is common knowledge that every function in a Python code starts with the word "def" and every class with the word "class". Moreover, when a library or a package is imported, it is imported with the word "import". Even when there is a need to add only one function from a library/package, the word "import" is used again. So, every time that an item is imported in the code or a function or a class is created by a developer, one of the three mentioned above words appears in the source code. This is the logic of text analysis, where it tries to spot words in a text and see their repetition.

The main difference between the calculations of the first and the third ratio: the first metric takes under consideration the total number of times that a developer added libraries, functions and classes, while the third metric takes under consideration the number of the different libraries, functions and classes that a developer added in the source code.

More specifically, in the first case the quantity is counted, while in the second one the novelty that someone brought in the project.

For the calculation of these two ratios, four different values, two numerators and two denominators, must be calculated.

The numerator of the ratio that comes up from the first metric is always equal or greater than the numerator that comes up from the third one, for each programmer. Since, it includes all the imports, while the third ratio's numerator is only about the different imports that a developer added in the source code.

The denominators concern all the developers, so the first ratio's denominator is about the total imports added by all the developers. Similarly, the third ratio's denominator is the number of all the different imports that were added in the code by every developer.

In this manner, it becomes more clear who is the core developer that first imports/calls most of the libraries and built the most of the functions. By developing the most necessary functions for the project's creation, for first time , the developer gains the title of core developer. Also, it can be safely assumed that the developers that only reuse functions and classes, that were built by other developers, are associates. This type of developers are expected to have low third ratio and is assumed that they work more on the debugging of the code and less on its development.

Using the logic that was explained above, text analysis was done directly on the source code, that was written by each developer and the first and third metrics were built. Firstly, this code takes the revision files of a project, through git commands, and corresponds all the commits numbers with the corresponding developer. Secondly, by keeping the history files in the chronological order that the commits happened, it stores who was the first one that built a function and who was the developer that just called it later (which is important for the third ratio). Thirdly, by running these commit ids through subprocesses and pygit2 diff object, the source code that is included in each commit is obtained and text analysis is applied on it. Lastly, having dictionaries, lists and counters to save and count the imported or called functions, classes and libraries, the two ratios are calculated.

The form of the output that subprocess and pygit2 diff patch returns the source code, allows to avoid including extra code for comments removal. The output's form is: + and - starting lines, to declare if this line was added or deleted by a developer. This helps in the text analysis. For example, in a plain text, searching only for the word "import", it could be repeated in a comment and lead to the miscalculation of the imported items. Now, searching for "+import" this mistake can be avoided.

The output of the code for the two ratios includes the name of the programmer and under it the corresponding ratio.

4.10 Demonstration of Metric two with the Software

Metric two takes into account the number of lines that were added and deleted by each developer. This metric shows the measure of initiative towards main-

tenance of the code. Using subprocess, the output is obtained from the git log –stat command, which returns the count of the insertions and the deletions made by the author.

The insertions and deletions that were done in README.MD files are deducted from the count. Here, a separate count of insertions and deletions is maintained. Using the text analysis, the ratios are taken for all the contributors. The output is saved in a text file, where the names of the contributors and the corresponding ratios under them are included.

4.11 Demonstration of Metric four with the Software

The metric four focuses on measuring which developer has written a function from scratch and which developers have contributed in editing functions and classes. The purpose here is to identify which developers have the ability to understand the requirements better and write a logic for the code, and which ones can work on an enhancement of the code or on error removal or similar activities.

The keywords in Python like '+def', '+class', can be used to identify the lines in the output that contain the functions on which each author had worked upon. To get the required output the git commands through Subprocess and pygit2 diff patch are used.

Working with author e-mail id's is better than with the names, because the conflicts when two authors might have the same names can be handled. Apart from pygit2 diff patch, the commit id for authors were used and were given as input to git show command. This gave the code patches for each author in the text file. All this output is taken into a text file and later text analysis is performed on the file.

Instead of searching 'def','class', '+def','+class' has been considered. Each code patch obtained gives the patch in increasing order of time, hence the patch committed first will be parsed and saved against the email id of the developer. A dictionary of the written and extended functions is maintained to count the corresponding functions for each author. Two values are obtained

for each programmer.

The output shows the two ratios for each author against their email ids.

Chapter 5

Testing and Results

After designing the new metrics and cleaning the data, the Python code evaluates their values for each contributor of the project. Taking different projects (popular and not) and following a specific contributor in different projects (cross-project developers) the new metrics are tested. Hereafter, the results must be analyzed.

Unit testing as well as integration testing is done on the code.

5.1 Repositories

Tacotron
The first repository that was cloned and has been used to run the metric code was Tacotron. This repository can be found on the following link:

https://github.com/Kyubyong/tacotron

This project was chosen to test the code because, it has relatively small number of commits, contributors and has good ratings in term of stars, which reflects the popularity of the repository.

Project's Characteristics
Number of commits: 80 commits
Stars: 479
Contributors: 3
Lines of Code: 985

The number of contributors is 3 and the number of commits is 80, therefore this repository is suitable to run as a prototype (because it is small and the validation will be easy). The code for metrics 1,2,3,4 takes the output in text file, hence, it is necessary to test with the code that produces log of small size.

The Tacotron project was cloned twice. Firstly, to run the unit test, to see the git log outputs and the git diff patch output. Then, to run each metric and integration testing to see the results obtained from all the four metrics. The codes have also been timed to see how long it takes to generate the output.

Results

Author:	1	2a	2b	3	4a	4b
Candelwill:	0.0647976	0.0017	0.99722	0	0	0.033613445
Spotlight:	0.133680375	0.0414	0.00277	0	0	0.11092437
Longinlove:	1.6533749	0.956802	0	2	1	0.791596639

Metric 2a shows the ratio for lines of code added. Metric 2b shows the ratio for lines of code deleted. Metric 4a shows the ratio of number classes and functions that were first initiated by the developer, metric 4b shows ratio of classes and functions edited by the developer.

The time of execution for the code took for metric 1,3,4: 3.1559 seconds.

The time required to execute metric 2 is: 0.015657424926757812 seconds.

Another observation that has been made during the testing process is that author Longinlove had two accounts to contribute but made commits from same email id.

Apart from Tacotron these metrics were tested on repositories named Python and ZeroNet. The links for these are:

Python :https://github.com/TheAlgorithms/Python

ZeroNet: https://github.com/HelloZeroNet/ZeroNet

Python

Project's Characteristics
Number of commits: 248
Contributors: 41
Stars:1, 484

The repository Python was chosen because it has higher number of commits and higher number of contributors than Tacotron repository. The repository is very popular, which can be confirmed by the 1,484 stars for the repository. The main aim to study such a huge.

ZeroNet

Project's Characteristics
Number of commits: 1551
Contributors: 69
Stars:9,189

Once the Python repository gave the successful results, the performance of the designed code on repositories larger than Tacotron and Python should be tested. The ZeroNet repository sufficed these conditions with 1551 commits, 69 contributors, and 9,189 stars. The number of commits is an important aspect, because the size of diff patch being produced largely depends on it. The bigger the number of commits, the more the needed space for the storage of the text file being produced. The git diff patch takes longer to get printed to the output file. While testing the ZeroNet project, encoding errors appeared.

The contributors of the Tacotron project were tracked. The repositories in which Python was the language of development and had developers common to Tacotron have been cloned. The repositories in which the developers of the Tacotron are lone contributors are ignored.

As the developer "Longinlove" has higher ratios in Tacotron project, the projects for him/her were first cloned. The selected repositories are: Transformer, Sudoku, Numpy.

Then the repositories for Spotlight and Candelwill were selected. The selected repositories are: Merlin and Returnn.

Sudoku

This repository has 4 contributors, but 5 email id which have logged in commits. Here, the developer who is common to Tacotron has contributed with two email ids.

Project's Characteristics
Number of commits: 20
Contributors: 4
Stars: 503

The Results Obtained are:

Author:	1	2a	2b	3	4a	4b
Longinlove	0.0590322	0.087662338	0	0	0	0.3
KPark	1.409677419	0.896103896	0.005649718	2	1	0.7
mcemilguneyy@gmail.com	0	0.003246753	0.005649718	0	0	0
pascal@borreli.com	0	0.00974026	0.971751412	0	0	0
eirik@morland.no	0	0.003246753	0.016949153	0	0	0

The execution time is for metric 1,3,4: 0.3382396697998047 seconds.

The execution time for metric 2 is: 0.06826090812683105 seconds.

Numpy Exercises

While testing the repository Numpy Exercises, a new challenge came up. The values in the fields for metric 1,3,4 are zero, because the diff patch did not have the words like "+def" or "+import" at all. Hence, it has been observed that the designed code for metrics has not been helpful to draw any conclusions about the contributions made by the developers for this repository.

Project's Characteristics
Number of contributors: 2
Stars: 753
Number of commits: 40

Output:

Author:	1	2a	2b	3	4a	4b
kiendang@Dangs-MacBook-Pro.local	0	0	0.071428571	0	0	0
Longinlove	0	0.970588235	0	0	0	0
KPark	0	0.029411765	0	0	0	0
prakash.nitin63@gmail.com	0	0	0.928571429	0	0	0

The time required for the execution of metric 2 is: 0.08798575401306152 seconds.

The time required for execution of metric 1,3,4 is: 1.475914478302002 seconds.

Output as obtained on console:

The ratio for metric 1: Counter()
The ratio for metric 3: Counter()
Metric 4 part 1: Counter()
Metric 4 part 2: Counter()

Transformer

The repository Transformer is the third repository that was cloned, as it also has an developer common to Tacotron.

Project's Characteristics
Number of Contributors: 2
Number of Commits: 7
Number of stars: 420

This repository has the least number of commits.

The results obtained are:

Author:	1	2a	2b	3	4a	4b
Longinlove	3	0.8947	0	3	1.384615385	1.082568807
Arthur	0	0.1053	1	0	0	0

The time required to execute metric 2 is: 0.022881746292114258 seconds.

The time required to execute metric 1,3,4 is : 2.90655517578125 seconds.

Merlin

In this repository Candelwill is the developer, who was also a contributor for Tacotron. The repository was chosen to analyze behavior of Candelwill as a developer. The output contains no contribution of Candelwill.

Project's Characteristics

Number of Contributors: 19
Number of Commits: 235
Number of stars: 261

These outputs add no information to analyze the type of contribution by developer Candelwill because there is no output produced for him. One reason that the execution time is huge can be that the patches are huge in terms of lines added.

Returnn
The developer Spotlight is the one common between Tacotron and Returnn. Due to many commits in this repository, running the code on Returnn was a huge challenge as it took too long to output the diff patch.

Number of commits: 4550
Number of contributors: 18
Stars:47

The output of metric code for this repository adds no useful information as there is no ratio of metrics that measure for developer spotlight.

5.2 Facing Encoding Errors

It has to be underlined here that not only the language (Python) that the project was written is important in order to take correct results, but also the encoding type.

This means that there are some characters in the .py files which cannot be recognized by the encoding utf-8. Encoding is used to translate a Unicode string into a sequence of bytes by default in Python 3. Furthermore, the encoding

error is not caused by the programming language but because of the encoding used by the console. To fix such an error in windows, the below written command can be run:

chcp 65001

Or it can be fixed by making the relevant encoding settings in tools like in Pycharm.

In MacOS X the encoding error can be fixed if any developer normalizes the command, in this case utf-8. Apart from this, setting the python sys default encoding to UTF-8 is another way of dealing with the error.

There is more information about these encoding errors and how to deal with them in Python developer community's articles [23],[24].

5.3 Results

In this section, the graphical representations of all the metrics for each of the repository are given. The graphs have been produced in Tableau business intelligence tool 10.3. The data source is the excel sheet that has all the results obtained from metric code. To maintain the anonymity of the developers, numerical aliases have been randomly assigned to the email ids.

1. Tactron

   From the below given graph, the reader can understand the behavior for each developer of the Tacotron repository. It can be seen from the graph that developer with alias 3 has imported most of the statements, has brought a lot of novelty and has written most of the classes and functions. A behavior which show that this author is a core developer in the project. The developer with alias 2 shows efforts in deletion of the number of lines of the code and editing the pre written functions/ classes. The developer with alias 1, has a contribution in maintenance effort of the code and in editing the pre written functions.

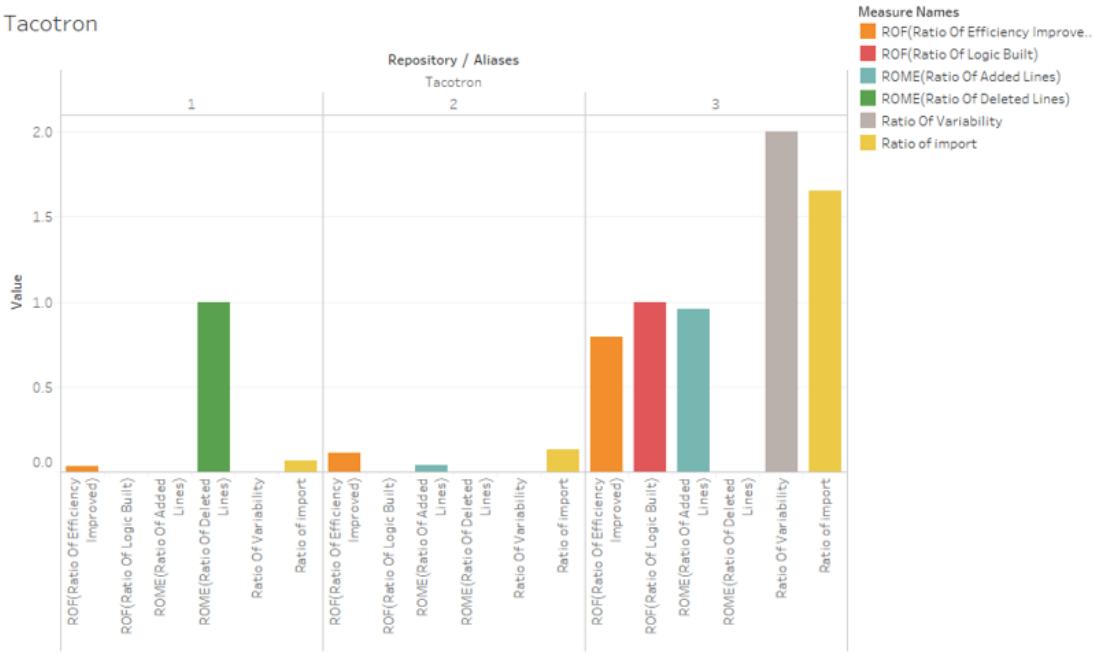


Figure 5.1 Ratios of metric for Repository Tacotron

2. Sudoku

   From the below given graph, the reader can understand the behavior of each developer of the Sudoku repository. Here again, there is the distinctive difference in the way that the developers have contributed. The developer with alias 3 has significant contribution in importing libraries, bringing novelty, writing the logic and expanding on it, as compared to the other developers who have only made a contribution in the maintenance of the code.

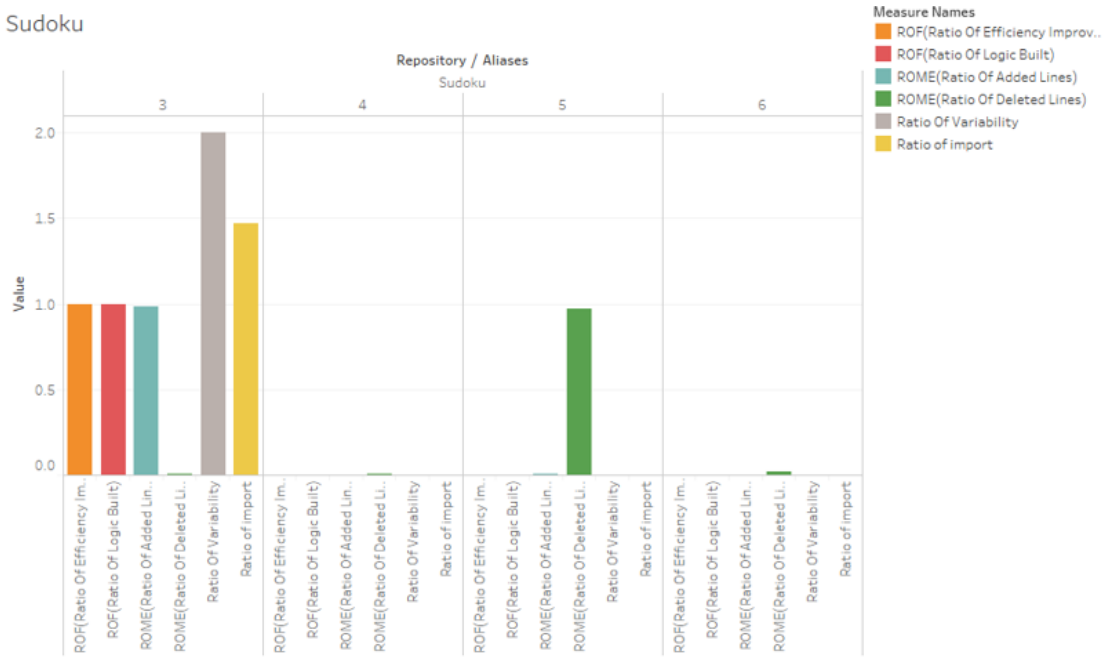


Figure 5.2 Ratios of metric for Repository Sudoku

3. Numpy Exercises

   From the below given graph, the reader can understand the behavior of each developer of the Numpy Exercises repository. As mentioned in the previous section, the output only reflects the maintenance effort of the developers. The developer with alias 3 is contributing largely in inserting lines of code. Developers with aliases 7 and 8 contribute in deletion of lines of code.

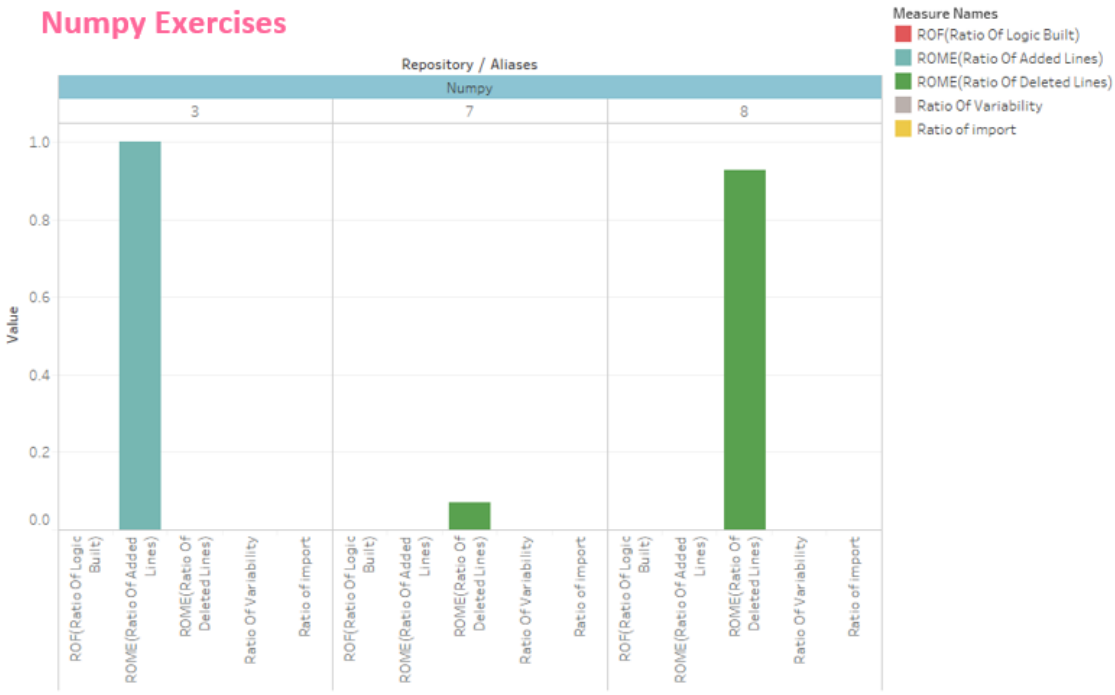


Figure:5.3 Ratios of metrics from Numpy Exercises repository

4. Transformer

   From the below given graph, the reader can understand the behavior of each developer of the Transformer repository. The graph shows the behavior of the developer with alias 3 and 4. Developer with alias 3 has significantly contributed in importing libraries, editing the pre existing function/classes. This behavior indicates that he/she must be the core developer for the repository, furthermore the developer with alias 4 has contributed only in the maintenance effort of the code.

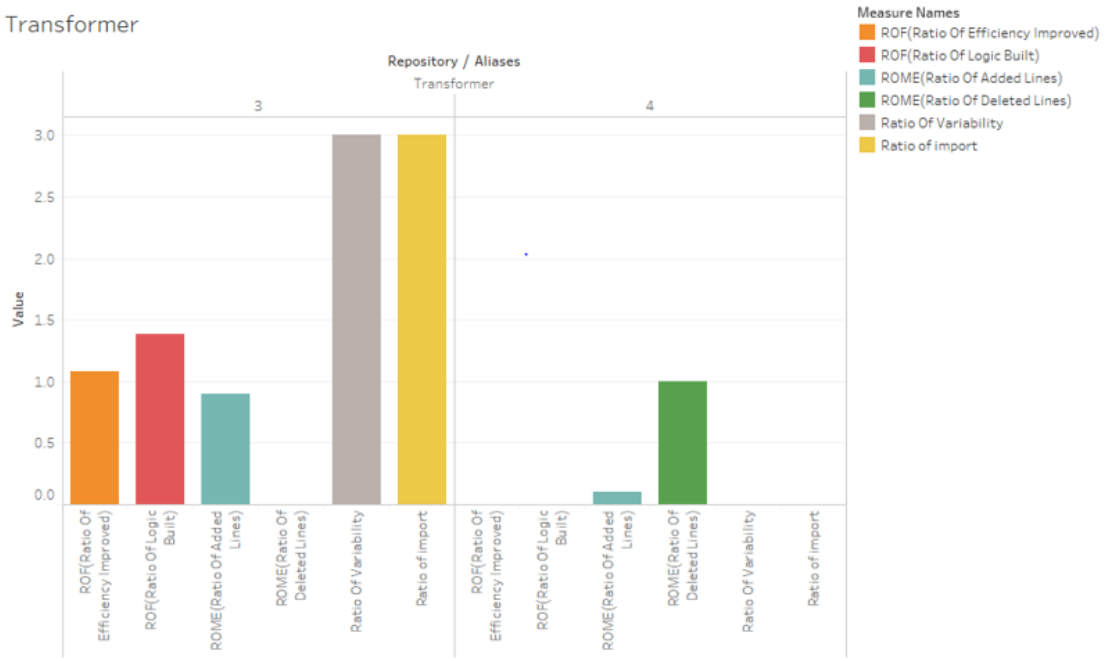


Figure 5.4 Ratios of metric for Repository Transformer

5.3.1 Comparative study for every metric and Summary

In this section, all the graphs that were plotted, for each metric and for all developers, for each repository, are explained. This was done to study the contribution behavior of developer with alias 3, who is common to all repositories.

5. **Metric 1 - Ratio of Import**

Below, graph for metric 1 is presented. Metric 1 reflects how many libraries have been imported by the developer in the code. This graph illustrates how the developer with alias 3 has contributed in repositories and whether there is any pattern in his behavior.

From the graph it can be observed, that the developer with alias 3 is the one who imports a large number of the libraries in every repository that he/she has contributed to. Hence, it could be claimed that he/she is aware of the programming language in depth as compared to the other developers in the project.

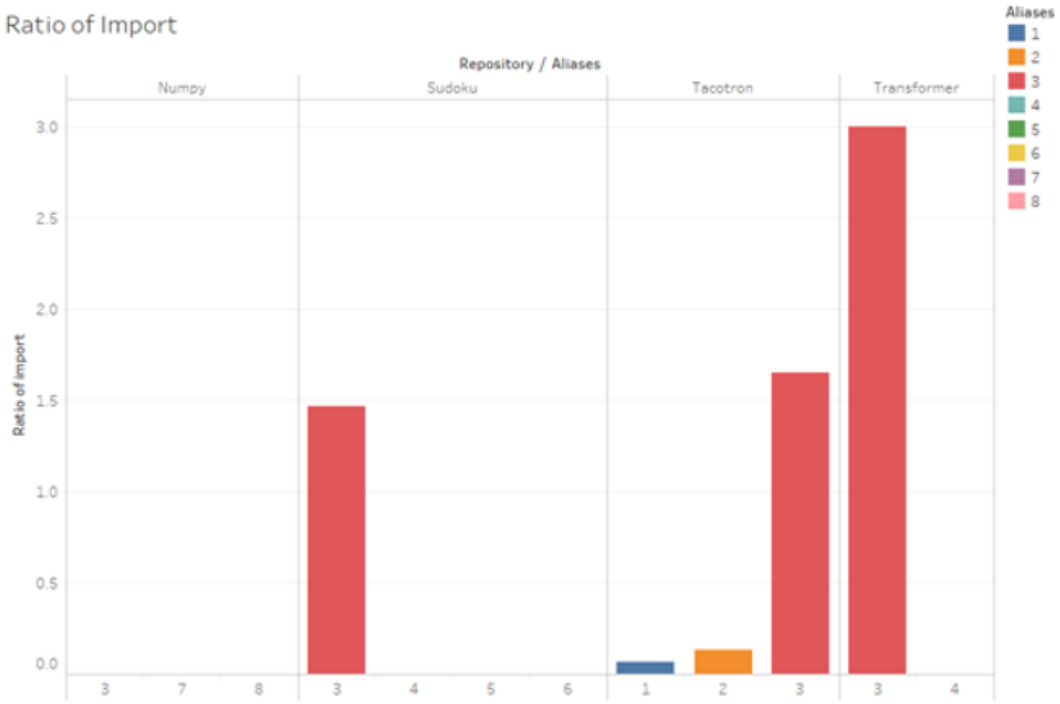


Figure 5.5 Comparison of Ratio of Import metric among all

6. **Metric 2 - Ratio of maintenance effort**

The Metric 2 reflects the maintenance effort of the developers. From the graph, it can observe that except for the developer with alias 3, the other developers do not show any constant behavior for metric 2.

For the lines of code added, the developer with alias 3 has a huge share in developing the code in all the repositories where he/she is part of. Whereas, the developer with alias 3 and 4 have contributed satisfactorily in the repositories that have been studied for them, however, this is a too small sample space to conclude about their software developing behavior. The developer with alias 5 and developer with alias 1 have a very small contribution in the repositories that have been studied for them.

For the lines of code deleted, the developer with alias 3 follows a pattern.

Figure 5.6 Comparison of Ratio of Maintenance Effort metric among all developers

He/she, generally contributes only a bit or at all on removing the lines of code or editing the lines of code. For the developer with alias 4, nothing can be concluded for his/her behavior in terms of lines deleted. Only in one project he/she has contributed significantly and only a little bit in others. The developers with alias 5,1 and 8 have contributed largely in deleting the lines of code.

7. **Metric 3 - Ratio of variability**

Metric 3 reflects how much novelty has been brought in the code. From the graph that is given below, it can bee noted that there is a significant pattern in the behavior of the developer with alias 3. In all the repositories that he/she has contributed to, he/she has the largest contribution. Developer with alias 1 has, also, contributed in bringing novelty, but not as developer with alias 3.

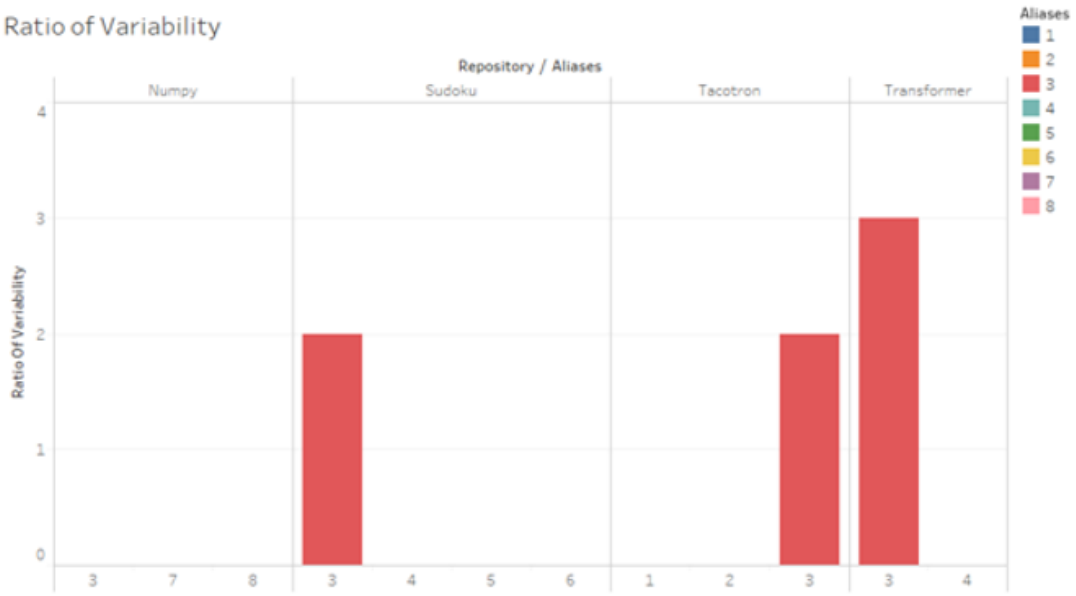


Figure 5.7 Comparison of Ratio of Variability metric among all developers

8. **Metric 4 - Ratio of functionality**

Metric 4 reflects whether the programmer writes functions and classes for the first time or expands on others' work. This metric has two parts, the first one calculates the ratio of function/classes written by developers for the first time and the second part focuses on the functions/classes edited by them.

In the graph below, for developer with alias 3 there is pattern for metric 4A and for metric 4B, as well. It can be seen that he/she has largely contributed in all the four repositories in terms of writing functions/classes and editing them. The observation in terms of number of repositories for other developers is not sufficient to conclude anything about their pattern in contributions.

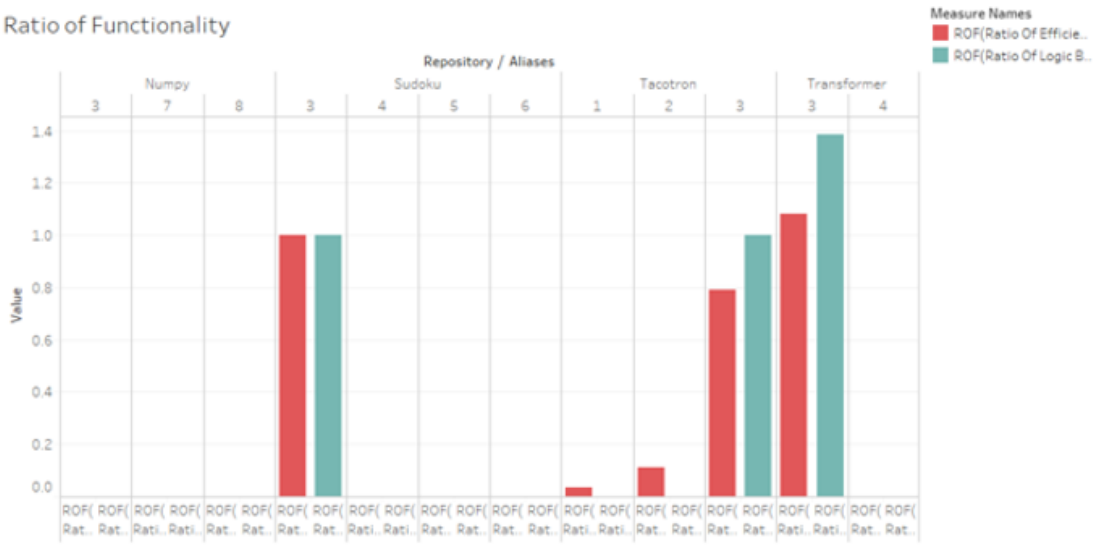


Figure 5.8 Comparison of Ratio of Functionality metric among all developers

5.3.2 Comparative study of all the developers for all the metrics

This section compares the ratios that were obtained for all the developers.

1. **Developer 1**

   Developer 1's profile has been studied for the repository Tacotron. Developer 1 is peripheral developer for Tacotron repository. He/she has minute contribution in importing libraries and huge contribution in deleting lines of code, which shows that he/she has worked in debugging the code or pruning it.

2. **Developer 2**

   Developer 2's profile has been studied for the Tacotron repository. When compared with other developers for this repository it can commented that developer 2 is a peripheral developer for Tacotron repository.

   Developer 2 has contributed in importing libraries, inserting lines of code, and editing the pre written functions or classes. This means that developer 2 has worked in testing the code/ debugging the code, and can respond towards editing code written by others.

3. **Developer 3**

   Developer 3's profile has been studied for all the four repositories namely Tacotron, Numpy Exercises, Sudoku and Transformer. There is clear pattern in his/her software development across all the repositories.

   It can be observed that the developer 3 has high contribution in metric 1, which shows that he/she has imported most of the libraries for the code that is being developed in the respective repositories. Hence, developer 3 is the core developer for all the repositories.

   Apart from this, another vital observation is that this developer does not make an effort to prune the code, as there are no values for metric 2, which measures the lines of code deleted. Hence, along with being a core developer, he/she also has extensive knowledge of libraries in Python language, and brings novelty to the code.

4. **Developer 4**

   Developer 4's profile has been studied only for the repository named Transformer. From the graph it can be observed that developer 4 has contributed towards inserting lines of code and deleting lines of code. The higher ratio is for deleting the lines of code. This indicates that he/she is more of a peripheral developer and works towards reducing the complexity of the code.

5. **Developer 5 and Developer 6**

   Developer 5's and developer 6's profiles have been studied only for the repository named Sudoku. Developer 5 is the peripheral developer along with developer 6. Both have contributions only in deleting lines of code. From these observations it can be said that both of these developers work towards debugging the code.

6. **Developer 7 and Developer 8**

   Developer 7's and developer 8's profiles have been studied only for the repository named Numpy Exercises. Not much can be concluded about these two developers profile, because the output of the metric code on git diff patch for this repository does not give any additional information for metric 1,3 and 4.

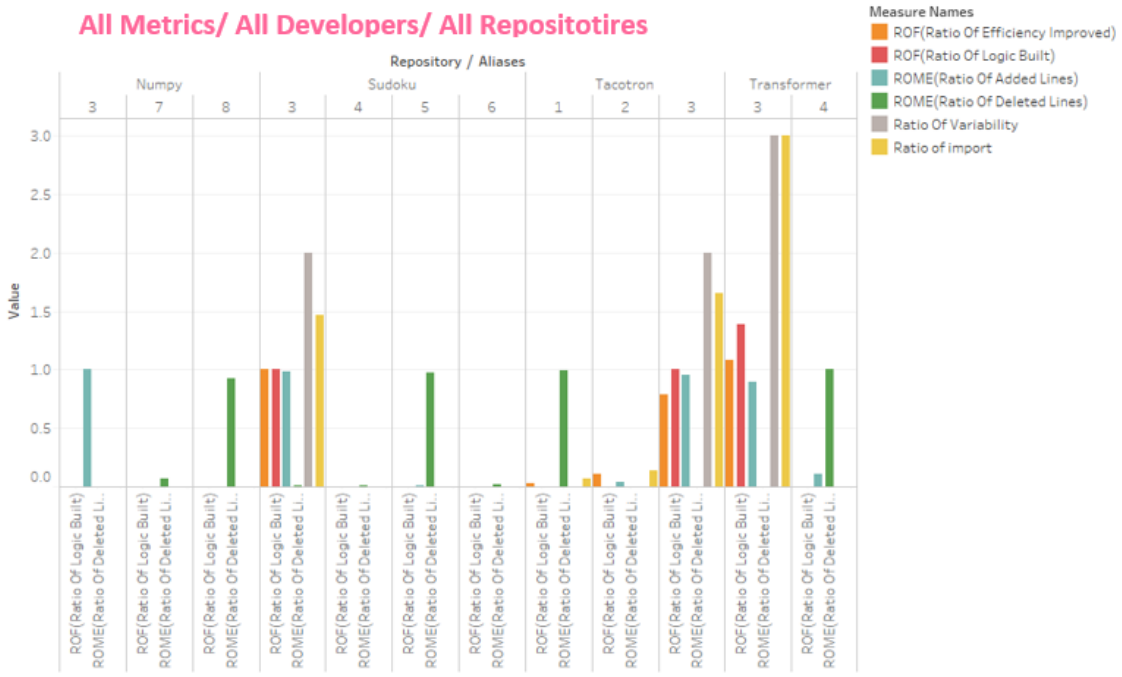


Figure:5.9 Comparative study of metrics for all developers

5.3.3 Summary on Testing and Results

In this section, there are observations that were made while testing.

1. **Time to run code:**
   The time that was taken to produce output by the code is largely dependent on the number of commits and the size of code patch obtained for each commit id. When the number of commits is huge along with long code patches, it takes longer to calculate the metrics.

2. **Key finding about git diff patch:**
   It has been observed, that certain code patches do not have the keywords like import, def, classes, and thus for such code patches, except for the metric 2, obtaining ratios for the rest of the metrics is not possible by the designed metrics code. Hence, for such code patches it is difficult to comment on the nature of contributions made by the developers.

3. **Key findings for developer with alias 3:**
   For all the repositories that have been studied for the developer with alias 3, it has been observed that there is a pattern in the way that he/she writes the code. Among the other members of his/her group, he/she is seen to be the person who includes the greatest number of lines of code and imports the most of the various libraries. Moreover, he/she is the person who acquires a large portion of the interest in code and has put in the most rationale of the code. It has been watched that, in a single repository he/she makes commits with more than one email id. In a case like this, the obtained ratio is an addition of the ratios obtained for all of his/her email ids. The repositories in which developer with alias 3 is a part, it is observed that these have high popularity, which can be concluded by its stars.

4. **Certain ratios are more than one:**
   Since some ratios (metric 1,3 and 4) are sums of ratios with values from 0 to 1, the resulting ratio is between 0 and 2 or 0 and 3.

5.4 Theoretical Validation

In this section, literature on how new proposed software metrics are validated theoretically is given.

Since, it is essential that the measures that were used for individual profiling are valid, validation is a crucial part of this study. Below, it is given a small number of articles that proposed a way of validation of software metrics. This helps to understand that there are many different ways for validation and someone should pick what suits the most on what he/she wants to validate.

In 1988, Weyuker [25] recommends that the evaluation can be achieved by spotting a set of desirable properties that a metric must possess and then examining a new metric decide if it is valid or not according to if it exhibits these properties.

In 1992, Schneidewind [26] proposes an empirical validation process. In this process, a software metric is validated according to if it is associated with one or more other measures of interest or not.

In 1990, Melton [27] and in 1994 Fenton [28] suggest that a metric is valid only if it obeys the Representation condition of measurement theory.

In 1991, Fenton and Kitchenham [29] show two different aspects of validation. One of them is for identifying the usefulness of a measure for predictive purposes and the other one is for identifying the extent to which a measure characterizes a stated attribute, they said.

The common questions in these papers were, between others, the questions below:

1. how to validate a measure
2. how to assess the validation work of others
3. when it is appropriate to apply a measure in a given situation

In conclusion, there are many theoretical ways to validate a software metric. The observation of the results in combination with the expected results, logic assumptions, and visualization of the results give some nuggets that whether the researcher is on the right way or not. There is, also, the empirical validation, which could not happen for this work, since there are no others validated metrics for individuals profiling. So, this difficult task would be better to say that needs more time, more work and more testing.

5.5 Validation of the new metrics

In this section, the correctness of the results produced by the new designed metrics is examined. The correctness is concluded by the observations made for developer with alias 3, by studying 4 repositories in which he/she has been a developer.

It has been observed that for 'metric 1' the developer has ratio large than 1 for all the repositories he/she has contributed too similar is the case for metric 2A, metric 3, metric 4A and 4B.

Hence, the given metrics are clearly able to prove the coding ability of this developer and thus can be confirmed that he/she is the core developer of these repositories.

The previously mentioned expected insights can be now confirmed from the results obtained by answering below questions for the developer with alias 3.

1. How much does a person's developing ability change over time?

   As seen for the developer with alias 3, it can be commented that the developers developing ability has not changed over time. He/she is the one who writes maximum code for all the repositories he/she has contributed to.

2. How adaptable is a developer?

   For the same developer, the high values of the ratio (close to one) 4B show that this developer is highly adaptable and can modify or interact with new and unseen code.

5.6 Key Findings

1. **Confirmation of: 15 percent of the developers in the repository are core developers**
   In the Literature review chapter it has been mentioned that for any repository about 15 percent of the total developers are the core developers.

From the current obtained results it can be seen that out of 8 developers 1 developer is the core developer, hence this hypothesis is confirmed.

2. **Confirmation of the expected insights about the core and the peripheral developers**
   The double significance of the fact that the studied repositories where public repositories in GitHub are:

   Firstly, it is reasonable that there is only one core developer. A person that decided to develop code without payment and uploaded it in GitHub, is working on his/her own idea and it is natural to have a high level understanding of the project.

   Furthermore, it is a rare case of two or more people that are strangers and not colleagues, who want to build the same thing at the same time and communicate to each other for building the project together. That's why there are so many small public projects in GitHub and most of them that were not created for a company have only one core developer. This does not mean that they are not collaborative, since they allow other developers to interfere. Also, they feel confident about their work when they have their code is public in order to be replicated, judged, optimized and used from anyone.

   Secondly, the peripheral developers are people passionate with software development and are experienced developers. Because, it is very difficult to work on unseen code and to debug or to reduce the complexity of it. The associate programmers have the experience, the talent and the passion to do it.

   It is observed, from the new metrics that the main task of the peripheral developers is to work on before written functions and classes and delete lines. They do not add novelty but they are improving the functionality and care about project's maintenance.

5.6.1 Limitations

These limitations are about the scope of test cases, the study about various developers' profile and the type of the git diff patch that is obtained.

1. **Scope of test cases: Size of the repositories**

The tests that have been performed are limited to the repositories that have been studied. These repositories are small in size. As a result the number of developers is small and no repositories with many core developers were checked. In order to obtain more insights about a core developer it is better to compare him/her with another core developer.

2. **Programming Language and sole developer projects**

   The effort was to study the developers of the Tacotron repository. Each developer's GitHub profile was checked for Python repositories and only the repositories where the developer was not the sole developer have been chosen. Only for developer 3, such meaningful repositories were obtained and cloned. Hence, this work studied mainly developer 3.

3. **Git Diff patch**

   The study hugely depends on the git diff patch obtained for the repositories. If the code patches are such where there are no keywords like '+def','+class','+import' the metrics 1,3 and 4 are not obtained, hence drawing any conclusions is not possible. Such challenges were found in Numpy Exercises repository.

4. **Open source code**

   Only public repositories were studied. The developers that completed these projects were not paid by any company to build them. This means that nobody created these teams and the projects are small.

5. **Studying the number of source lines of code**

   The source lines of code indicate the total number of lines present in the code file. A repository can have either one or more than one files for which developers commit changes. If the repository contains few .py files it is possible to manually calculate the source lines of code for the repository, otherwise it is difficult to calculate source lines of code. To achieve this user can make use of preexisting metric available to calculate the source lines of code (LOC metric). In this work a hint is given for the project's size from the commits' number that is written in the characteristics of each project.

The results show that the new metrics could be used to provide an overall assessment of the developers that built a software. However, more empirical studies are needed before these results can be generalized.

Chapter 6

Conclusion

6.1 Overall Review of the four new Metrics

Closing this work, it can be said that the goals that were set have been fulfilled with success. Four new metrics for individuals profiling were defined, specified through text analysis, calculated for different projects and eventually validated.

It is understandable from their definitions that metrics one, three and four are correlated. The first metric studies the size of the project that a developer deals with, by counting the number of the imports, functions and classes that he/she "touched". The third one studies the novelty that the developer brought by counting the number of the imports, functions and classes that he/she wrote. Metric four has two parts, comparing them answers the question, if the developer has worked more on developing a new code or on debugging and code replication. This happens by counting the functionality, hence the number of the functions and the classes that the developer wrote and worked on. Overall, these three metrics study the amount of work a developer undertook and his/hers role in the project (core - associate).

The second metric is not related to the other three, it studies the number of the lines of code added and deleted by a developer. This metric reveals how much a programmer tries to reduce the project's complexity and optimize the results. It could be combined with the others in the following way:

a) In case where the person with the biggest score in the second metric is a core developer, then this person is very dedicated to his/her work. Since, he/she was there from the beginning of the project and did not lose his/her interest

in the project. He/she is still trying to optimize the code, which means that he/she is passionate about his/her work.

b) In case that the individual with the biggest score in the second metric is a peripheral developer, then this person has experience and is learning fast. Since, it is easier for a developer to write their own code from than to debug another developer's work, this person has joined the project at a later (not while the structure is built) stage and has the ability to reduce the complexity while understanding it at the same time. An associate that not only wants to work with new unseen code, but also he/she is able to get a quick and good idea of the code. Such an associate also has the required experience that will be be helpful in removing any errors.

6.2 Suggestions for Future Work

6.2.1 Relative Areas

Besides the nuggets of future work and optimization that were left in each chapter, here are more ideas for the expansion of this work.

After finishing building the software for the proposed metrics, the purpose is to find more metrics. The social network is one such area associated with the developers' network, hence useful tools could be found there.

Another area is the pattern recognition, as there is a need for metrics that clearly distinguishes initially, the characteristics of the developers.

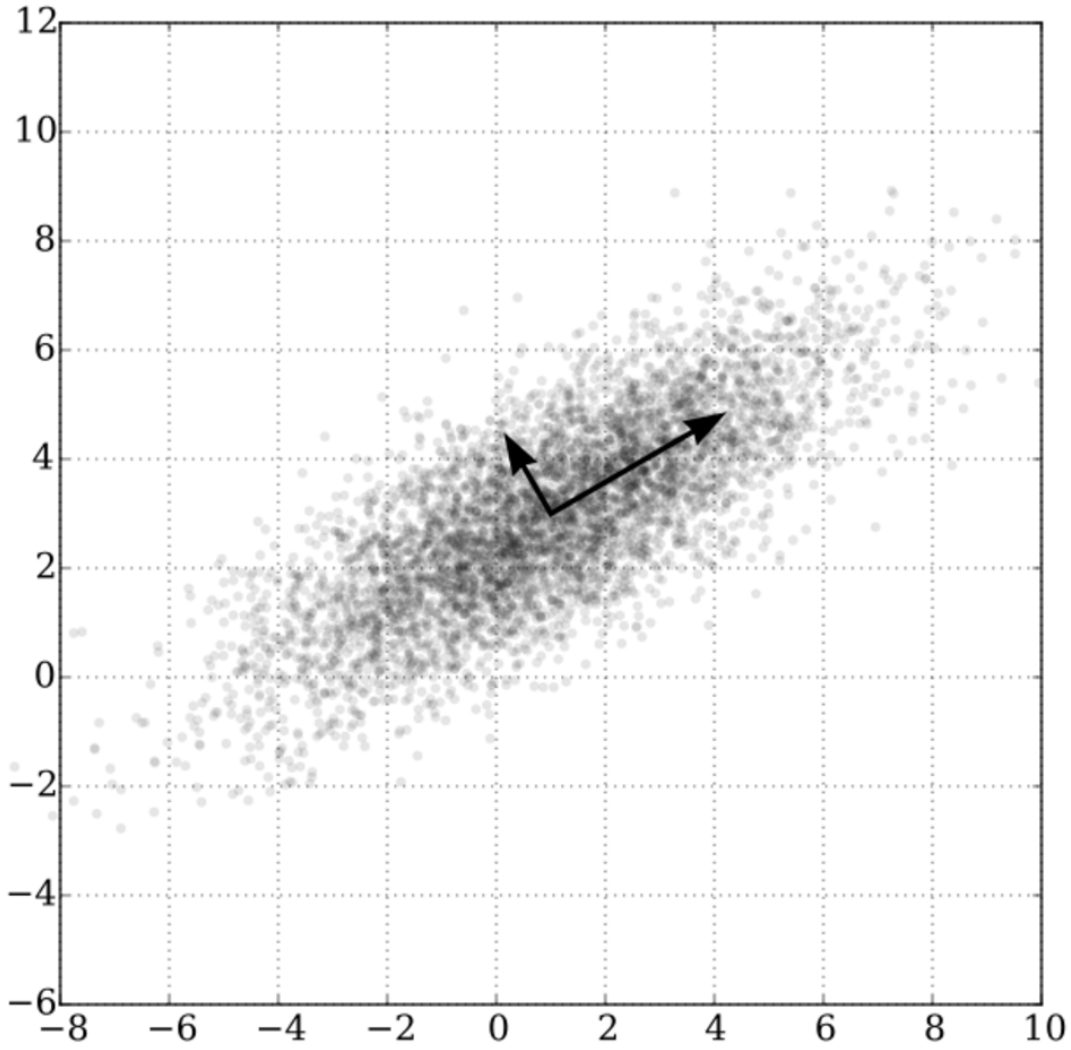
Predictive Models have been built for the available metrics [8], [30], [31]. So, after individually studying the existing metrics, it is possible to study the relationships among them and their different combinations for model construction. The model creation is the next logical step.

6.2.2 PCA

Despite the already referred tools that can be used in the future for optimization, there is PCA that could be used to create predictive models and do exploratory data analysis.

This statistical tool is used to compress the data and study the main characteristics from big data sets. It reminds a lot of canonical correlation analysis, because it works with orthogonal transformation.

To begin with its description, there is a set of observations, which has possibly correlated variables and the purpose is to end up to a set that includes values of linearly uncorrelated variables.



So, in the picture above the reader can see that the distribution lies on a two

dimensional space. This is the known as Eigen problem, in which it is expected to find the maximum variance. This is done by projecting all the points on the axes x and y. If the lengths of the variances on the axes x,y are a and b accordingly, then simply the biggest one is chosen. When the available dataset has many dimensions, it is difficult to study them graphically and even more to compare them and decide which one is more important.

PCA flattens the dataset by a meaningful way into two or three dimensions. Studying the dataset by directions, a Principal Component (PC) has the length of the maximum distance between two elements of the dataset to this direction. So, PC1 spans the direction of the first most variation, PC2 spans the direction of the second most variation and so on for all the dimensions of the dataset. Another thing that can be done at this point is to score the data according to extent they influence these PCs.

Every time two dimensions are studied, starting with the ones that have the bigger variations on the data. The linear combination of the above scores is taken for the influence of every element from PC1 and PC2 and is multiplied with the coordinates of the element on x and y, correspondingly. One single value for the PC1 and PC2 is obtained. Now, the PCs with the biggest values effect the dataset more, usually the first two are much more important than the others. In addition, not only the samples can be compared with this method can, but also the redundant information can be ignored by reducing the number of dimensions. Going to lower dimensions the visualization of the data becomes easier and patterns are revealed easier.

Principal Component Analysis can be used to find out which developer's measures are correlated with high starred projects. This can contribute in the validation of the new metrics.

Appendix A - Links for the code of the brand new metrics

The reader is encouraged to use the Python code that was developed, expand it, optimize it and move it one step further.

There is a lot of work to be done in this area, in order to fill the gap in the literature and the business needs. In an effort to make this easier, suggestions for future work have been provided and the Python codes for the new metrics has been uploaded in GitHub in a public project.

The link for the code of metric 2 is:

$https://github.com/elpidabantra/ratiometrics/blob/master/finalmetric2codegithub.py$

The link for the code of metrics 1,3 and 4:

$https://github.com/elpidabantra/ratiometrics/blob/master/Metriccodeonethreefour.py$

In addition, the text developed online in Overleaf and can be found in:

$https://www.overleaf.com/10545461kjsmmqfzkycz\#/39386983/$

Appendix B - Metric results for Merlin and Return repositories

Merlin

Output of Repository Merlin: The output:

The inserted lines are:
¡ dev1@gmail.com ¿: 0.004051316677920324,
¡dev4@gmail.com¿: 0.014854827819041188,
¡dev5@gmail.com¿: 0.0, ¡dev6@gmail.com¿: 0.0006752194463200541,
¡dev12@gmail.com¿: 0.08642808912896692,
¡ Dev2@gmail.com ¿: 0.012829169480081027,
¡ dev3@gmail.com ¿: 0.015530047265361242,
¡dev11@gmail.com¿: 0.004051316677920324,
¡dev10@gmail.com¿: 0.0054017555705604325,
¡dev9@gmail.com¿: 0.03376097231600,
¡dev8@gmail.com¿: 0.018906144496961513,
¡dev7@gmail.com¿: 0.0013504388926,
¡dev13@gmail.com¿: 0.11141120864280891,
¡dev14@gmail.com¿: 0.0027008777852802163,
¡dev15@gmail.com¿: 0.0,
¡dev16@gmail.com¿: 0.6880486158001351,
¡dev17@gmail.com¿: 0.0, ¡dev18@gmail.com¿: 0.0

The deleted lines are :
¡ dev1@gmail.com ¿: 0.006956521739130435,
¡dev4@gmail.com¿: 0.0017391304347826088,
¡dev5@gmail.com¿: 0.0034782608695652175,
¡dev6@gmail.com¿: 0.08695652173913043,
¡dev12@gmail.com¿: 0.0,

¡ Dev2@gmail.com ¿: 0.03826086956521739,
¡ Dev3@gmail.com ¿: 0.022608695652173914,
¡dev11@gmail.com¿: 0.6991304347826087,
¡dev10@gmail.com¿: 0.02608695652173913,
¡dev9@gmail.com¿: 0.0017391304347826088,
¡dev8@gmail.com¿: 0.0017391304347826088,
¡dev7@gmail.com¿: 0.0034782608695652175,
¡dev13@gmail.com¿: 0.08347826086956522,
¡dev14@gmail.com¿: 0.01391304347826087,
¡dev15@gmail.com¿: 0.0,
¡dev16@gmail.com¿: 0.0,
¡dev17@gmail.com¿: 0.010434782608695653,
¡dev18@gmail.com¿: 0.0

The time required to execute the metric 2 is: 0.06798577308654785

The ratio for metric 1 [ROI]:
Counter(dev16@gmail.com: 1.0314994986396437,
dev13@gmail.com: 0.853474323899216,
dev12@gmail.com: 0.8528718285022882,
Dev3@gmail.com: 0.08718219213708274,
dev4@gmail.com: 0.036926954984104296,
dev8@gmail.com: 0.03528166357146258,
dev9@gmail.com: 0.03306190774460354,
dev11@gmail.com: 0.01957335540548167,
dev6@gmail.com: 0.01957335540548167,
dev7@gmail.com: 0.012914087924904533,
Dev2@gmail.com: 0.008820415892865646,
dev17@gmail.com: 0.006600660066006601,
dev10@gmail.com: 0.0022197558268590455)

The ratio for metric 3 [RON] Counter:
(dev16@gmail.com: 2.6022727272727275,
dev12@gmail.com: 0.2159090909090909,
dev4@gmail.com: 0.18181818181818182)

Metric 4 part 1:
Counter(dev16@gmail.com: 1.375,
dev12@gmail.com: 0.125)

Metric 4 part 2:
Counter(dev16@gmail.com: 0.5396039603960396,
dev13@gmail.com: 0.3564356435643564,
dev12@gmail.com: 0.3547854785478548,
Dev3@gmail.com: 0.056105610561056105,
dev8@gmail.com: 0.026402640264026403,
dev9@gmail.com: 0.026402640264026403,
dev17@gmail.com: 0.006600660066006601,
dev4@gmail.com: 0.006600660066006601,
Dev2@gmail.com: 0.006600660066006601,
dev11@gmail.com: 0.0033003300330033004,
dev7@gmail.com: 0.0033003300330033004,
dev6@gmail.com: 0.0033003300330033004)

The execution time required for metric 1,3,4 is 106.7184374332428 seconds.

Return

The output of repository Return:

The output The ratio for metric 1 [ROI]:
Counter(dev19@gmail.com: 0.9775827314295851,
dev25@gmail.com: 0.7555779262285437,
dev28@gmail.com: 0.6980335592691695,
dev99@gmail.com: 0.1818815952972964,
dev199@gmail.com: 0.16015206460689912,
dev199@gmail.com: 0.07525139997302688,
dev043@gmail.com: 0.032680672613391124,
dev003@gmail.com: 0.027660933188297876,
dev033@gmail.com: 0.02276266357869654,
dev042@gmail.com: 0.014654286377918439,
dev032@gmail.com: 0.01173286991095706,
dev10909@gmail.com: 0.0082171366519795,
dev1099@gmail.com: 0.006912138469382629,
dev044@gmail.com: 0.006507660746847053,
dev04466@gmail.com: 0.0033908311329480982,
dev09@gmail.com: 0.003174821382574147 3,
dev045@gmail.com: 0.0031392387868558963,
dev002@gmail.com: 0.003134139989650325,
dev046@gmail.com: 0.0017796239779741328,
dev0466@gmail.com: 0.0016309216793771236,

dev01@gmail.com: 0.0013890312113501667,
dev08@gmail.com: 0.0013890312113501667,
dev02@gmail.com: 0.0006908469848563939,
dev001@gmail.com: 0.0006738753010726583)

The ratio for metric 3 [RON] Counter:
(dev19@gmail.com: 1.4362888933132711,
dev043@gmail.com: 0.8726631917132043,
dev28@gmail.com: 0.4162045544302393,
dev199@gmail.com: 0.16050553283526392,
dev199@gmail.com: 0.07527016838401608,
dev042@gmail.com: 0.03040965066599648,
dev25@gmail.com: 0.008658008658008658)

Metric 4 part 1:
Counter(dev19@gmail.com: 0.8641618497109826,
dev043@gmail.com: 0.3699421965317919,
dev28@gmail.com: 0.31213872832369943,
dev199@gmail.com: 0.07514450867052022,
dev199@gmail.com: 0.031791907514450865,
dev042@gmail.com: 0.008670520231213872,
dev25@gmail.com: 0.005780346820809248)

Metric 4 part 2:
Counter(dev19@gmail.com: 0.6019913869271596,
dev25@gmail.com: 0.4662798816165885,
dev28@gmail.com: 0.4293931923359716,
dev99@gmail.com: 0.11054885169326586,
dev199@gmail.com: 0.09682601469288896,
dev199@gmail.com: 0.04543179669688038,
dev043@gmail.com: 0.018508220726982893,
dev003@gmail.com: 0.017093304169987705,
dev033@gmail.com: 0.014031770746445717,
dev042@gmail.com: 0.009264923044603855,
dev032@gmail.com: 0.007164173571335891,
dev10909@gmail.com: 0.00526423103301266,
dev044@gmail.com: 0.003963619962062936,
dev1099@gmail.com: 0.003926547912534678,
dev04466@gmail.com: 0.002057498748818328,
dev045@gmail.com: 0.002008069349447318,

dev09@gmail.com: 0.0019493719376942422,
dev002@gmail.com: 0.001946282600233554,
dev046@gmail.com: 0.0010905361236229279,
dev0466@gmail.com: 0.0009947666623415941,
dev01@gmail.com: 0.0007908703899361742,
dev08@gmail.com: 0.0007908703899361742,
dev02@gmail.com: 0.00039234585750739896,
dev001@gmail.com: 0.0003892565200467108)

The execution time is 127.44491457939148

References

1. Boehm, B.W., Brown, J.R. and Lipow, M., 1976, October. Quantitative evaluation of software quality. In Proceedings of the 2nd international conference on Software engineering (pp. 592-605). IEEE Computer Society Press.

2. Barkmann, H., Lincke, R. and Lwe, W., 2009, May. Quantitative evaluation of software quality metrics in open-source projects. In Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on (pp. 1067-1072). IEEE.

3. Li, W. and Henry, S., 1993. Object-oriented metrics that predict maintainability. Journal of systems and software, 23(2), pp.111-122.

4. Jacquet, J.P. and Abran, A., 1997, June. From software metrics to software measurement methods: A process model. In Software Engineering Standards Symposium and Forum, 1997. Emerging International Standards. ISESS 97., Third IEEE International (pp. 128-135). IEEE.

5. Rubey, R.J. and Hartwick, R.D., 1968, January. Quantitative measurement of program quality. In Proceedings of the 1968 23rd ACM national conference (pp. 671-677). ACM.

6. Rombach, H.D., 1987. A controlled experiment on the impact of software structure on maintainability. IEEE Transactions on Software Engineering, (3), pp.344-354.

7. DeMarco, T., 1986. Controlling software projects: Management, measurement, and estimates. Prentice Hall PTR.

8. Dutoit, A.H. and Bruegge, B., 1998. Communication metrics for software development. IEEE transactions on Software Engineering, 24(8), pp.615-628.

9. Yu, L. and Ramaswamy, S., 2007, May. Mining cvs repositories to understand open-source project developer roles. In Proceedings of the Fourth International Workshop on Mining Software Repositories (p. 8). IEEE Computer Society.

10. Oliva, G.A., Santana, F.W., de Oliveira, K.C., de Souza, C.R. and Gerosa, M.A., 2012, September. Characterizing key developers: a case study with apache ant. In International Conference on Collaboration and Technology (pp. 97-112). Springer Berlin Heidelberg.

11. Wei, K., Crowston, K., Eseryel, U.Y. and Heckman, R., 2016. Roles and politeness behavior in community-based free/libre open source software development. Information and Management.

12. Crowston, K., Wei, K., Li, Q. and Howison, J., 2006, January. Core and periphery in free/libre and open source software team communications. In System Sciences, 2006. HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on (Vol. 6, pp. 118a-118a). IEEE.

13. Joblin, M., Apel, S., Hunsen, C. and Mauerer, W., 2016. Classifying developers into core and peripheral: an empirical study on count and network metrics. arXiv preprint arXiv:1604.00830.

14. Mens, T., 2016. Research trends in structural software complexity. arXiv preprint arXiv:1608.01533.

15. Fenton, N.E. and Neil, M., 1999. Software metrics: successes, failures and new directions. Journal of Systems and Software, 47(2), pp.149-157.

16. Misra, S. and Cafer, F., 2010. A software metric for python language. Computational Science and Its ApplicationsICCSA 2010, pp.301-313.

17. http://furius.ca/snakefood/

18. https://sourceforge.net/projects/pymetrics/

19. http://www.unixuser.org/ euske/python/index.html

20. Basci, D. and Misra, S., 2008. Entropy metric for xml dtd documents. ACM SIGSOFT Software Engineering Notes, 33(4), p.5.

21. Basci, D. and Misra, S., 2009. Data complexity metrics for xml web services. Advances in Electrical and Computer Engineering, 9(2), pp.9-15.

22. Basci, D. and Misra, S., 2009. Measuring and evaluating a design complexity metric for XML schema documents. Journal of Information Science and Engineering, 25, pp.1405-1425.

23. https://docs.python.org/2/howto/unicode.html

24. https://docs.python.org/3/howto/unicode.html

25. Weyuker, E.J., 1988. Evaluating software complexity measures. IEEE transactions on Software Engineering, 14(9), pp.1357-1365.

26. Schneidewind, N.F., 1992. Methodology for validating software metrics. IEEE Transactions on software engineering, 18(5), pp.410-422.

27. Melton, A.C., Gustafson, D.A., Bieman, J.M. and Baker, A.L., 1990. A mathematical perspective for software measure research. Software Engineering Journal, 5(5), pp.246-254.

28. Fenton, N., 1994. Software measurement: A necessary scientific basis. IEEE Transactions on software engineering, 20(3), pp.199-206.

29. Fenton, N. and Kitchenham, B., 1991. Validating software measures. Software Testing, Verification and Reliability, 1(2), pp.27-42.

30. Muthanna, S., Kontogiannis, K., Ponnambalam, K. and Stacey, B., 2000. A maintainability model for industrial software systems using design level metrics. In Reverse Engineering, 2000. Proceedings. Seventh Working Conference on (pp. 248-256). IEEE.

31. Jiang, Y., Cuki, B., Menzies, T. and Bartlow, N., 2008, May. Comparing design and code metrics for software quality prediction. In Proceedings of the 4th international workshop on Predictor models in software engineering (pp. 11-18). ACM.