Sociometrics in Software Engineering: Measuring the Contribution of a Software Engineer in a Software Project through GitHub

Sourojit Das , B.Tech

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Prof. Stephen Barrett

August 2019

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Sourojit Das

August 15, 2019

Permission to Lend and/or Copy

Acknowledgments

I would like to extend my sincerest thanks and appreciation to Prof. Stephen Barrett for his continuous motivation and valuable guidance throughout the research. Stephen gave me free rein to shape the project as per my vision, while always helping me in spots of difficulty, and steering me in the right direction whenever needed.

To my parents, I must express my deepest gratitude for providing me with everything I ever needed, and for their love and encouragement throughout my years of study. This accomplishment would not have been possible without you. Thank you both. To my friends, too numerous to name, who have helped and encouraged me every step of the way. From old friends far away in India, to new ones next door in Dublin, I express my heartfelt thanks. I wouldn't have made it without you guys. To my professors at Trinity who have taught me so well, I offer my thanks. To Trinity herself, who shone as a beacon of hope during my darkest hours, and gave me memories to cherish forever, I am ever grateful. And finally to my patrons at Tata Trusts, whose generous scholarship helped me reach here in the first place.

Sourojit Das

University of Dublin, Trinity College
August 2019

Sociometrics in Software Engineering: Measuring the Contribution of a Software Engineer in a Software Project through GitHub

Sourojit Das, Master of Science in Computer Science
University of Dublin, Trinity College, 2019

Supervisor: Stephen Barrett

Contribution is fundamental to the concept of performance analysis. It is integral to judging the worth of a person in a team, an employee in an organization, and participants in any activity in general. With the industry stressing on rightsizing and optimized workforce management, it is of paramount importance that the contribution of employees is comprehensively monitored and adequately rewarded. However, the subjective and bias-prone nature of existing processes has led to widespread employee dissatisfaction, especially in the software industry. Traditional LOC-based metrics fail to measure contribution as they do not consider the full range of activities performed by a software engineer in a project. Thus, there arises a need for a comprehensive metric to measure contribution. This research seeks to build a data-model to measure the contribution of a software engineer in a software project by using data mined from GitHub and Gitter. A high-level model for contribution is constructed and then expanded, by a top-down approach, to create a model that considers both the amount of work done, and its quality. This model represents an expert's view of measuring contribution. Using the data mined from GitHub and Gitter a data- model is constructed, from bottom-up, that seeks to identify useful signals to quantify the concepts expressed by the high-level top-down model. This data-model is conceptualized by applying the principles of Measurement Theory in software engineering and is constantly refined using defeasible reasoning to better quantify the concepts of the high-level model. Finally, an expert system is used to measure the contribution of a software engineer in a project. The proposed solution is then subjected to a sanity test against the manual evaluation methods proposed by the expert in the high-level model. The results indicate that the data model is a reasonably accurate representation of the model proposed by the expert and succeeds in providing a rank-ordering of developers consistent with that obtained from a manual evaluation of contribution using the expert's model, though some loss in conceptual fidelity is observed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter elucidates the motivation behind the research and lays out the research question, the research objectives and potential challenges in conducting the research. It is rounded off with a discussion of the technical approach proposed for solving the research question and a brief outline of the overall structure of the dissertation.

1.1 Motivation

The OED defines contribution to be "the part played by a person or thing in bringing about a result or helping something to advance."[1] Though the definition is straightforward, quantifying a person's contribution to a body of work has posed significant challenges to researchers (Amor et al. 2006;Gousios et al. 2008).

The importance in measuring the contribution of a software developer to a project stems from its intimate association with appraising the value of the software engineer to the team or the organization, and its central role in the performance appraisal process carried out in all major organizations (Thoti et al.,2015). In order to truly realize the importance of measuring contribution of software engineers in an organization, an understanding of the inherent shortcomings of the performance appraisal system, followed by major organizations, is vital.

Flippo (1976) had defined performance appraisal to be *"the systematic, periodic and an impartial rating of an employee's excellence in the matter pertaining to his present*

[1] `https://www.oxfordlearnersdictionaries.com/definition/english/contribution`,accessed:04.08.2018

job and his potential for a better job." It is commonly accepted that an organization performs best when the people in that organization perform optimally, or in other words contribute effectively to achieving the organization's goals.(Sanyal & Biswas,2014)

Most major corporations perform periodic performance appraisals of their employees to this end. While some may want to conduct these processes in a formally structured manner, others rely on informal interactions.

A 2015 study (Thoti et al.,2015) on the various performance appraisal methods followed by Multi-National Corporations(MNC's) of Indian origin, revealed a preponderance of traditional methods of appraisal. Management tended to rely on highly subjective analyses of contribution essays, project checklists and supervisor recommendations. Recent research (Thoti et al.,2015) has brought to light the increasingly negative attitude of IT employees towards such appraisal processes. A plurality of IT professionals distrust the performance appraisal process and view it to be highly prone to bias. (Gupta & Kumar,2013) Studies have also highlighted several shortcomings of traditional performance appraisal methods, primarily due its subjective nature which makes it unverifiable by others(Islam & bin Mohd Rasad,2006). Some prominent shortcomings identified by Islam & bin Mohd Rasad (2006) are -

1. **The Halo Effect**, which causes appraisers to rate all traits or behaviours by being excessively influenced by a single trait or behaviour, may lead to highly biased appraisals in which those conducting the appraisal are inadvertently blind to the bigger picture.

2. **Errors of Central Tendency** may also occur, in which, those conducting the appraisal may give ratings around the middle of the rating scale to everyone in order to play safe.

3. **Personal Prejudice** has also been found to be a major influence in performance appraisals with several employees complaining of victimization as a result of poor personal relations with supervisors and management.

Several researchers have appreciated the fact that contribution is not only context-specific but should be considered in terms of characterizing employee activity. Though tools like GitPrime[2], SonarQube[3], waydev[4], and Code Climate[5] have taken strides in helping to assess qualitative characteristics in software development, they still tend to depend on the engineer's contribution through code, while neglecting the vast array of tasks actually performed. With the continued fixation of State-of-the-Art systems on code-based metrics, there is an immense opportunity to explore the impact of non-coding tasks, such as code-base administration and engagement on chat platforms in order to obtain a comprehensive view of what contribution in software engineering actually entails.

1.2 Research Question

This dissertation explores the question whether a data-model can be constructed to calculate the contribution of a software engineer to a software project through signals obtained from an open-source project repository in GitHub.

1.3 Research Objectives

To address the research question mentioned above, we have defined the following research objectives:

1. To create a data-model for measuring the contribution of a Software engineer in a project by analyzing the entire range of developer activity over a period of time, considering both the amount of work performed and the quality of the work, as opposed to traditional LOC based measures.
2. To identify useful signals from GitHub that can serve to approximate/quantify the high-level concepts espoused in the data model.
3. To create a rule-based tool (Expert System) that would quantify the contribution of a software developer using the metrics derived from the selected GitHub and Gitter signals.

[2]`https://www.gitprime.com/`,accessed:04.08.2018
[3]`https://www.sonarqube.org/`,accessed:05.08.2018
[4]`https://waydev.co/`,accessed:05.08.2018
[5]`https://codeclimate.com/`,accessed:05.08.2018

4. To evaluate the data-model through a sanity test of the contribution values generated by the tool with the values estimated by a human expert (the author himself for the purpose of this dissertation)

5. To visualize, by means of a dashboard, the relative contributions of the software engineers to the project.

1.4 Research Challenges

1.4.1 The Nature of the Problem

Though the concept of contribution seems inherently simple and can be roughly defined to be *"an action or a service that helps to cause or increase something"*, quantifying it poses a significant challenge, as defining what actually consists of a useful contribution to a body of work is highly subjective and context-based. For instance, writing 100 LOC may be the benchmark for an effective contribution in a particular project, probably one that deals with the implementing a novel solution or a particularly hard problem. Projects dealing with more routine work, however, may see team members writing a few thousand LOC every day and pushing multiple commits. It would be inherently wrong to judge people simply on the amount of activity performed without taking into consideration the importance of that activity in the grand scheme of things, and the specific conditions under which that particular activity was performed. A good illustration would be to say that Edgar Allan Poe contributed less to the horror-fiction genre than Stephen King, simply because the latter has published more novels in that genre. Tackling this problem in a qualitative manner might give us a reasonable understanding of the key actors in a particular project and help identify some activities which can be considered to be more important than others. But researchers have struggled to quantify contribution of the identified actors or creating suitable metrics to enumerate the relative value of the activities earmarked. The advent of Agile methods of software development, and software development tools that eliminate the need of writing code have increasingly rendered traditional LOC-based metrics obsolete.

1.4.2 Inherent perils in mining GitHub

Research conducted to estimate participant contribution in software development projects have largely used GitHub or other publicly accessible VCS(Version Control Systems) to mine information and understand the mechanics of participant collaboration(Kalliamvakou et al.,2016). However, this is not without its own set of challenges, some of which have already been highlighted in previous research. (Kalliamvakou et al.,2016)

A primary concern while mining GitHub is the misalignment between the actual and mined data. For instance, a study by Kalliamvakou et al. (2016) highlighted that almost 40 % of the PR's did not appear to be merged with the master branch when mined from GitHub, even though they were, in reality.

Kalliamvakou et al. (2016) voiced other concerns centred on finding active projects for data mining such as:

1. Most projects tended to be inactive for significant periods of time.
2. A vast majority of GitHub projects (around 70%) are personal, and thus have only 1 committer.
3. Research has also established that most projects conduct some development activities outside GitHub.
4. The PR-based nature of GitHub ensures that all code merges to the main codebase (as represented by the Master branch) are for those commits which have been deemed acceptable by peer reviews. Thus, though we see the successes in creating new functionalities and fixing bugs, we do not glimpse the failures and the learning curve which are critical to have a well-rounded view of contribution.
5. Some GitHub repositories have logs of activities by non-users as well. In some cases, this is the result of the activity of users not being properly credited to their GitHub account.

1.4.3 Difficulty in using the GitHub API

Using GitHub API to mine for data also becomes a challenge, due to the API exposing a subset of the entirety of events or entities available on GitHub for that project, as well as providing limited information regarding those entities.(Kalliamvakou et al.,2016)

For instance, when we observe an Issue on GitHub site, we can clearly see all the relevant information about the Issue creator, commenters, references made to the issue and the pull request used to resolve the issue (if applicable). However, the Issue API from GitHub provides only a subset of this data and the rest has to painstakingly linked to the particular issue by mining information from a host of other API's like the Reactions API, the timeline API, etc.

1.4.4 Challenges in mining signals from Gitter chat

Mining chat boards or IRC applications to gauge participant engagement in terms of effectiveness in communication, pro-activeness in taking up issues, resolving doubts about the project in the community poses a significant challenge. The true challenge is in delineating conversations and converting them into a usable "Question asked" - "Question Answered" - "Acknowledgement to the answer" pattern. This becomes very challenging due to the inherent nature of chat boards where questions may be answered after a significant amount of time has passed, and there's no definitive way of knowing when a conversation has ended and who were the participants in that conversation without extensive manual analysis.

1.5 Overview of Technical Approach

To address the research objectives previously outlined, a high-level data-model is constructed to calculate the contribution of a software engineer to a software project. Following a top-down approach, contribution is broadly classified into three types, viz. Direct Contribution(dealing with contribution made through code directly to the codebase); Administrative Contribution(dealing with contribution made through Issue and Pull-Request handling and housekeeping activities); Guidance Contribution(based on engagement in the chat-boards, by asking pertinent questions, resolving queries and being a positive influence on the community in general). Data mining from GitHub and Gitter is done to collect signals which would help quantify the high-level concepts espoused in the data-model. A bottom-up approach is followed to generate increasingly complex metrics from simple counts of activities. This model is continuously refined through sanity testing, conducted for chosen individuals, in order to judge the correlation between the contribution values generated through the mined signals and the contribution scores given to these individuals by manual analysis. This process is repeated until a sufficient level of correlation is obtained. A dashboard is then utilized to graphically display the calculated contribution values, for engineers, for every quarter.

1.6 Dissertation Outline

- Chapter 1 introduces the project to the reader by elucidating the motivation behind the research, stating the research question and the research objectives addressed in this dissertation, listing the research challenges and giving a brief technical overview of the research.
- Chapter 2 presents an overview of the existing State-of-the-art methods employed to estimate the contribution of software engineers in a project. This chapter introduces the technology and research areas associated with the research carried out in this dissertation and reviews relevant literature.
- Chapter 3 provides details about the design decisions made during the dissertation and the reasoning for the same.
- Chapter 4 discusses the implementation details of this research. The technologies that were used are described, alongside the justification for selecting them.
- Chapter 5 provides an overview and discussion of the observations gathered throughout the research. The results of the research are also evaluated in light of the research objectives.
- Chapter 6 concludes the dissertation in which conclusions are drawn about the research and highlights the research contribution. It ends with a discussion on the future work prospects.

Chapter 2

State of Art

This chapter discusses the State of the Art solutions, in the field of Software Metrics, designed to calculate contribution and allied concepts like productivity. The chapter gives a brief background into the development of traditional software metrics, their shortcomings and discusses the advances made in academia and industry regarding the estimation of contribution as a software engineering metric. The relevant literature supporting the use of argumentation theory in crafting software metrics is also laid down, as is the use of measurement theory to create software metrics.

2.1 Background

For a thorough assessment of the present state of software metrics used in the industry, and their relative success in staying relevant given the rapid evolution of software engineering practices; it is imperative to look into its history.

2.1.1 Software Engineering Metrics - Growth and Overview

The first dedicated study on software metrics was conducted by Gilb (1977), though the origins of software metrics can be dated back to the 1960s(Fenton & Neil,1999). The Lines Of Code (LOC) measure was popularly used as a means to assess both developer productivity in terms of LOC and product quality(in terms of defects raised per Kilo Lines Of Code). Some early resource prediction models were developed by Putnam (1978) and Boehm (1981), which also utilized the LOC metric or depended on other related parameters like *"delivered source instructions."*(Fenton & Neil,1999) Akiyama (1971) pioneered the attempt to predict software quality when he put forward his proposal for a basic regression-based model for defect density in software modules. Akiyama's work (Akiyama,1971) was considered to be have used LOC as a "surrogate measure for program complexity" by authors such as Fenton & Neil (1999).

Fenton & Neil (1999), in the same work, draws our attention to the unsuitability of using LOC as a "surrogate measure" for different aspects of program size such as effort, functionality, and complexity. With the increasing diversity of programming languages by the mid-1970s, several researchers proposed various measures of software complexity. Pioneering work was done in the field of software complexity metrics by Halstead et al. (1977)and McCabe (1976), while Albrecht (1979) and subsequently Symons (1991) pioneered concepts of software metrics like function points which were inherently independent of the programming language used.

With a profusion of software metrics taking center-stage, Grady & Caswell (1987) carried out their landmark study into reporting the efficiency of software metrics used in the industry. This work set the baseline for and inspired many subsequent studies in this field.

Basili & Rombach (1988) proposed the concept of "GQM (Goal-Question Metrics)," which strove to ensure that a specific goal drove any software-metric creation activity. Though this approach was criticized for its emphasis on Top-Down metric development, and subsequent lack of focus in considering what was possible to calculated from bottom-up(Bache & Neil,1995, Hetzel,1993), it had several supporters in the industry, notably Hall & Fenton (1997) who stated that *"A metrics program without clear and specific goals and objectives is certainly doomed to fail."* This concept led to researchers focusing on quantifying concepts in software metrics, previously considered to be solely measurable through qualitative analysis, like Productivity and Contribution. In general, this approach can be credited with a bringing about a paradigm shift in the approach to crafting software metrics as now researchers focused exactly on what they wanted to measure, rather than use existing measures as an (often inadequate) approximation.

The field of software metric development soon embraced the concept of using metrics in empirical software engineering (Fenton & Neil.1999). Fenton & Neil (1999) clarified the term "empirical software engineering" to mean "evaluating the effectiveness of specific software engineering methods, tools, and technologies." A breakthrough in this field was achieved in the work of Basili & Reiter (1981) who defined a metric to have successfully passed its evaluation if the empirical results were acceptable, the experiments repeatable, and the ability to independently validate the findings of the tests performed. This was crucial as it established "a measurement theory basis for software metric activities" (Fenton & Neil,1999).

2.1.2 Software Engineering Metrics - Limitations and Recommendations

Though Software Metrics as a field of study has flourished, with a large number of texts and journals dedicated to the subject, and most major IT companies having some form of "software metric program" for in-house purposes; researchers have often highlighted the increasing irrelevance of current software metric methods in the face of the ever-evolving software industry.(Riguzzi,1996,Fenton & Neil,1999,Dhawan & Juneja,2013)

Fenton & Neil (1999) has categorically stated that *"There is no match in the content between the increased level of metrics activity in academia and industry."* The same paper(Fenton & Neil,1999) also says that *"Much academic metrics research is inherently irrelevant to industrial needs"* and proceeds to state that this irrelevance is in terms of both scope and content.

The irrelevance in scope is justified(Fenton & Neil,1999) because academic research concentrates on metrics which cannot be scaled up to satisfy industrial needs and depends on parameters which may not be easily measurable in a real-world context.

The irrelevance in content has been pointed out because academic research stresses on creating detailed code metrics, whereas the industry seeks metrics which enable more efficient processes and utilization of resources.(Fenton & Neil,1999)

Glass (1994) has emphatically stated in this context *"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."*

Fenton & Neil (1999) state that the way forward for software metric development is to be able to model both "cause and effect relationships" and "uncertainty and combination of evidence" to create metrics relevant to present day industry requirements.

It is in this background, that we must look into efforts made by academia to break free of the stranglehold of traditional software metrics and establish metrics to quantify concepts like contribution and productivity, which are considered vital to the industry.(Gousios et al.,2008)

2.2 Characterizing effort through activity

This research gives importance to quantitatively analyzing developer activity as a measure of the contribution done.The work of Amor et al. (2006) in this field is particularly significant as it argues comprehensively in favour of using activity signals to estimate developer effort, especially in the area of Free and Open Source Software (FOSS). Amor et al. (2006) states, that while determining effort is critical to all software engineering projects; there has been relatively little progress made in this field beyond the classical techniques employed.

Of the various traditional effort estimation models proposed (Cuadrado et al.,2002), the model by Boehm (1981) considers seven parameters for estimation, viz. "algorithmic cost modelling, expert judgment analogies, Parkinson's Law, pricing to win, top-down estimations, and bottom-up estimations". Conte et al. (1986)'s method of estimation proposed classification on the nature of the models before conducting an estimate.

However, researchers argue (Amor et al.,2006) that since these models are primarily dependent on the traditional LOC metric, they must be augmented by other signals of activity which can be mined from public software repositories, especially in the case of FOSS development.

Amor et al. (2006) argue that a thorough characterization of developer activity would provide a much more comprehensive estimate of effort than traditional LOC-based metrics and propose that a study of developer behaviour is more likely to produce realistic results than an analysis of the project artifacts. For FOSS projects, this is all the more important since they lack formal means to keep records of developer work and depend mainly on poorly coordinated volunteer activity.

To this end Amor et al. (2006) analyzed a set of public software repositories using various analysis tools like CVSAnalY[1] – a web-based tool for mining data from a soft-

[1]`http://cvsanaly.tigris.org/`,accessed:04.08.2018

ware repository(Robles, Koch, GonZÁlEZ-BARAHonA & Carlos,2004), MailingList-Stats[2] – a tool for analysing mailing list activity (Herraiz et al.,2006), GlueTheos[3] – a quantitative analysis tool that works on snapshots of the repository(Robles, González-Barahona & Ghosh,2004), DrJones – a tool for performing empirical analysis of software archaeology (Robles et al.,2005), CODD[4] – an "authorship extraction tool" (Matellán Olivera et al.,2003), and SLOCCount[5].

Using these tools, they (Amor et al.,2006) arrived at a comprehensive estimation of developer effort as a function of developer activity, as expressed by this set of formulae

$$cost = f(effort)$$

$$effort = g(activity) => cost = f(g(activity))$$

The work of Amor et al. (2006) helped set a strong precedent for future qualitative analysis of developer contribution and was acknowledged by Gousios et al. (2008)

Though this work did not provide a detailed explanation about identifying and measuring the sources of activity in a software project and the calculations involved in estimating the cost; it helped set the tone for the investigations into the nature of developer activity and the subsequent calculation of contribution from the same as done in this research.

2.3 Quantifying Concepts through Measurement Theory

Whereas the previous two sections dealt with concepts supporting the creation of a software metric like contribution (which was traditionally considered in purely qualitative terms), this section deals with the strategies employed in academia for creating new metrics. This section provides the background for the creation of the contribution metric defined by this research.

Whereas the previous two sections dealt with concepts supporting the creation

[2]`http://libresoft.urjc.es/indexhtml?menu=Tools&Tools=MLStats`,accessed:04.08.2018
[3]`http://libresoft.urjc.es/index.php?menu=Tools&Tools=GlueTheos`,accessed:04.08.2018
[4]`http://codd.berlios.de`,accessed:04.08.2018
[5]`http://www.dwheeler.com/sloccount/`,accessed:04.08.2018

of a software metric like contribution (which was traditionally considered in purely qualitative terms), this section deals with the strategies employed in academia for creating new metrics. This section provides the background for the creation of the contribution metric defined by this research.

A branch of measurement theory, “the representational theory of measurement” has been often used to build software metrics. In general, Measurement Theory is defined as a discipline for crafting fundamental rules for the correct use of a proposed framework.(Alexandre,2002)

The representational theory of measurement in particular seeks to formalize commonly help beliefs about the working of a system. A set of data, representative of the entity-attributes under consideration, is first gathered. This data is then manipulated in a way such that the entity relationships stay consistent. (Fenton & Bieman ,2014) According to Fenton & Bieman (2014) “our observation reflects a set of rules that we are imposing on the set of people”

To exemplify that the measurement process is a mapping process between the empirical and real-world data, Alexandre (2002) has given the following example:

“When we say that X is taller than Y, we define a binary relation between X and Y. In this case, ”taller than” is an empirical relation for height.”

The basic rules for creating metrics using the representational theory of measurement involve considering the real world as the domain and the mathematical world as the range, and then adhering to certain rules as stated by Fenton & Bieman (2014):

“...) the representation condition asserts that a measurement map-ping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations.”

The Figure 2.1 illustrates the stages of formal measurement postulated by Fenton & Bieman (2014)

A major challenge encountered in this regard is the representation of complex relationships between attributes, or when the representation of an attribute requires the combination of several aspects. This requires the measurement process to be divided into Direct and Indirect measurements. According to Alexandre (2002), *“Direct Measurement of an entity attribute involves no other attribute or entity. For e.g. Length of Source Code (LOC) or Duration of testing process (Hours).”*

Fenton & Bieman (2014) has defined Indirect Measurement of an attribute to be

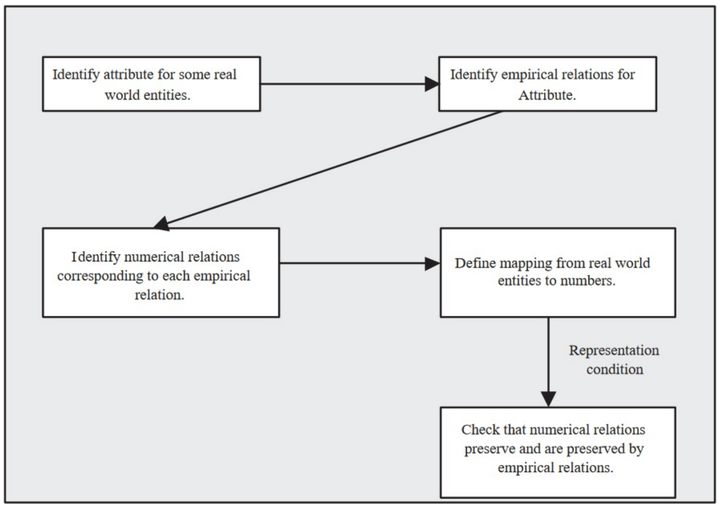

Identify attribute for some real world entities.
Identify empirical relations for Attribute.
Identify numerical relations corresponding to each empirical relation.
Define mapping from real world entities to numbers.
Representation condition
Check that numerical relations preserve and are preserved by empirical relations.

Figure 2.1: Formal Measurement Stages byFenton & Bieman (2014)

"obtained by comparing different measurements. For e.g. Number of defects divided by module size gives the Module defect density."

Another important consideration in this regard is the choice of Measurement Scales. Since Direct Measurement of an attribute tries to map the observed system to the mathematical system, it encounters problems when the mappings are of different dimensions. This necessitates the use of Measurement Scales to ensure appropriate analysis.

Five principal measurement scales have been identified, viz. **Nominal**(which classifies entities into categories), **Ordinal**(which performs rank ordering of the entities), **Interval**(which defines the distance between two value points in the domain and seeks to maintain a uniform distance or interval between all the value points), **Ratio**, and **Absolute**.

Subsequent chapters will demonstrate how these measurement scales were used in the creation of the data model to calculate contribution.

Of vital importance is the validation process followed to prove that the metrics sufficiently capture the essence of the concept they represent. According to Pfleeger et al. (1997), *"a measure is valid if it satisfies the representation condition: if it captures in the mathematical world the behaviour we perceive in the empirical world. For example, we must show that if H is a measure of height, and if A is taller than B, then H(A) is larger than H(B)."* However, Alexandre (2002) warns that such proofs are empirical in nature and difficult to replicate with sufficient rigour. Fenton & Bieman (2014) comments that *"Currently there isn't any accepted standard for validating a measure."*

Alexandre's research (Alexandre,2002) also goes on to classify software engineering metrics into three classes, viz. Processes, Products and, Resources.

Challa et al. (2011) have also outlined a way to quantify certain qualitative attributes. They propose the quantification of software quality attributes on the basis of a software quality model provided by the ISO, and define six attributes:

- Efficiency
- Usability
- Functionality
- Maintainability
- Portability
- Reliability

They (Challa et al.,2011) initially divide Software Quality into three components based on Developer, User and Manager perspective as shown in Figure 2.2

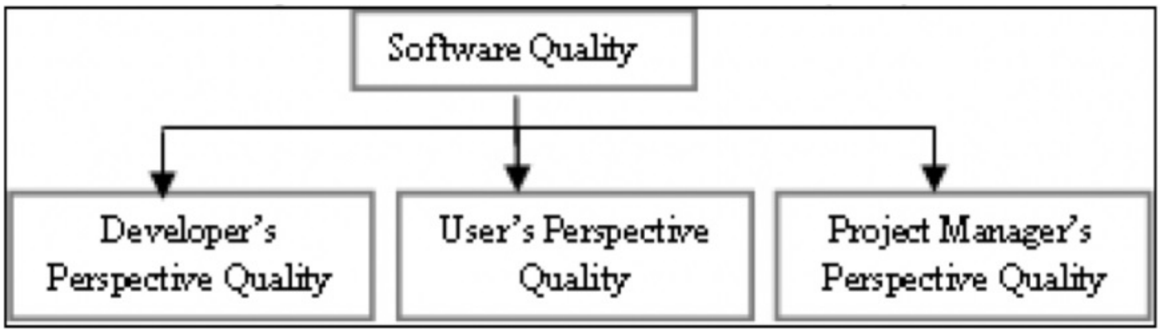

Software Quality
Developer's Perspective Quality
User's Perspective Quality
Project Manager's Perspective Quality

Figure 2.2: Software Quality Components by Challa et al. (2011)

These perspectives are further subdivided into characteristics and sub-characteristics as shown in Figure 2.3 and Figure 2.4

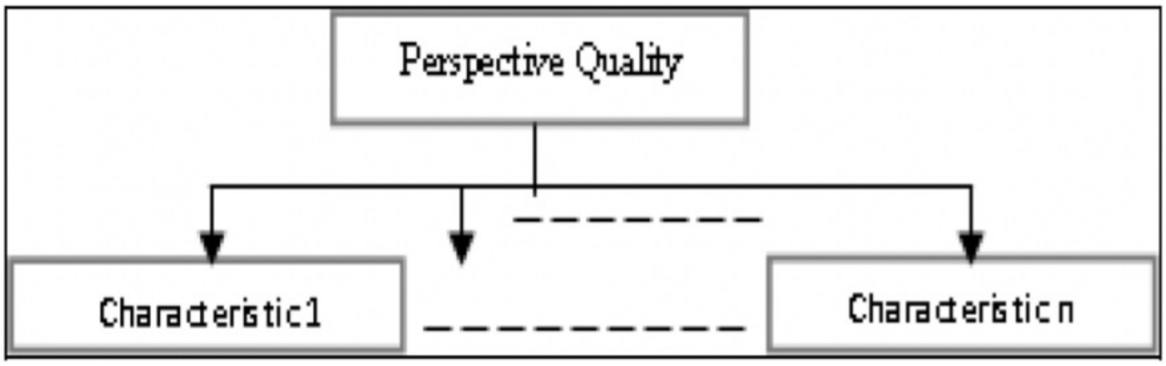


Figure 2.3: division of Perspectives into Characteristics by Challa et al. (2011)

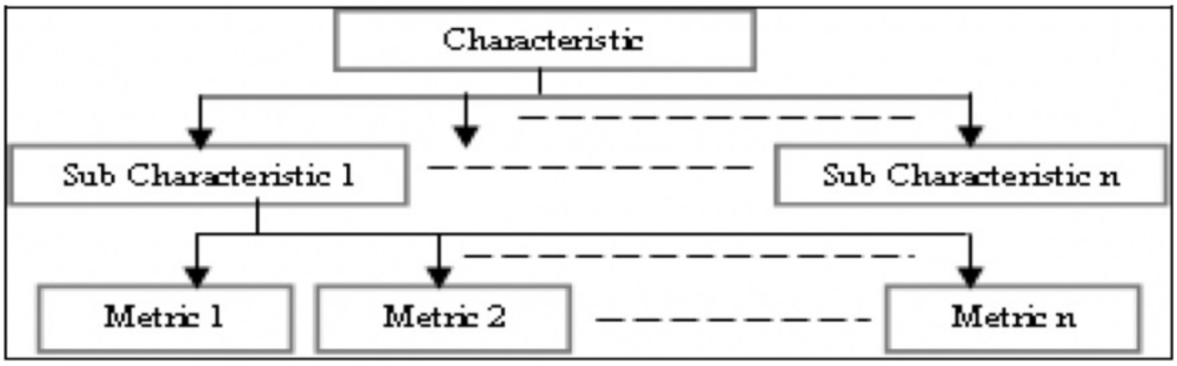


Figure 2.4: Subdivision of Characteristics by Challa et al. (2011)

Their use of hierarchical mapping of qualitative attributes to quantitative ones have been replicated in creating the data model for Contribution, where Qualitative metrics would be Direct Contribution, Administrative Contribution and Guidance Contribution; whereas Qualitative attributes are the base signals obtained from GitHub and Gitter. This is explained in further detail in the next chapter.

The work reviewed in this section helped set the base for the design of the contribution metric outlined in the next chapter.

2.4 Argumentation Theory in Software Engineering

This research uses Defeasible reasoning and Argumentation Theory for both quantifying contribution through a data-model as well as creating an evidence-based manual model for calculating contribution. This section reviews research supporting the use of such techniques in software engineering.

A key premise to the creation of the data-model for calculating contribution is the use of personal opinions to construct a high-level model and the use of evidence-based argumentation to construct, from the bottom up, an automated model which can quantify those concepts.

This approach finds resonance in research as a survey conducted by Devanbu et al. (2016) at Microsoft, required respondents to rank, in order of importance, the sources of information which helped them form their opinions about software engineering processes. It was found that respondents valued personal experience and peer opinion as primary sources of information when formulating opinions.

The creation of a manual model to estimate contribution for validating the contribution of a software engineer in a project based on the author's own perspective can be justified in the work of Rainer et al. (2003) who have established that "practitioners most valued information that was provided by other practitioners, with the ideal type of information being sourced from a local expert".

The basis for this dissertation research has been the use of *primary information*, i.e. information obtained through personal experience in creating the manual and automated models for calculating contribution.

The use of Argumentation Theory in general and defeasible reasoning in particular has been utilized in software engineering for the development of theories in this domain(Jørgensen & Sjøberg,2004, Sjøberg et al.,2008, Hannay & Jørgensen,2008)

The work of Johnson et al. (2012) propose that such efforts attempt to "generalize local observations and data into more abstract and universal knowledge". Substantial research supports the importance of this approach in developing theoretical models in software engineering especially in the field of Evidence Based Software Engineering.(Jorgensen et al.,2005, Brereton,2010)

Though our current scope of research tends not to hypothesize about the validity of the assumptions made in creating the model in a global scenario, it indeed forms a crucial part of the future work planned in this regard.

Rainer (2017) defines arguments to be "a type of persuasive reasoning with a particular structure: a structure of assertions that, through inference, are intended to support a conclusion". A simple model of evidence-based argumentation is presented in Figure 2.5 comprising of information, evidence, propositions and inferences.

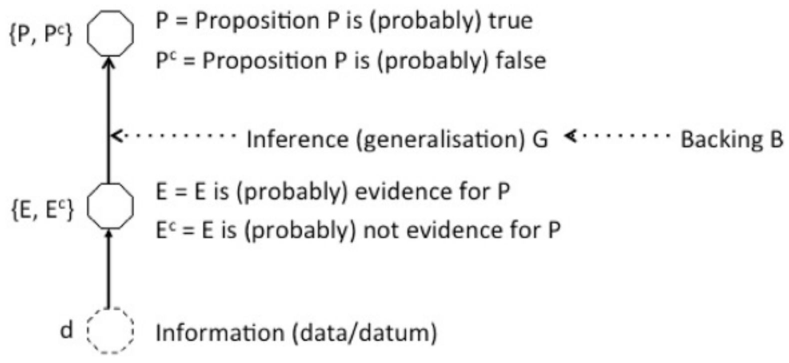

{P, P^c}
P = Proposition P is (probably) true
P^c = Proposition P is (probably) false
Inference (generalisation) G
Backing B
{E, E^c}
E = E is (probably) evidence for P
E^c = E is (probably) not evidence for P
d
Information (data/datum)

Figure 2.5: Argumentation Model by Rainer (2017)

The evidence E is inferred from the information d, and in turn the probability of proposition P is inferred from evidence E. This model borrows from previous work in this field by Schum (2001), Anderson et al. (2005), Twining (1994) and Toulmin (2003).

Defeasible reasoning is defined by Walton (2015) *"A defeasible argument is one in which the conclusion can be accepted tentatively in relation to the evidence known so far in a case, but may need to be retracted as new evidence comes in. A typical case of a defeasible argument is one based on a generalization that is subject to qualifications."*

Rainer (2017) holds forth that both software engineers as well as software researchers inherently make use of defeasible argumentation due to the constraints on research, practical decision making and the "complex, ever-changing nature of software practice".

Though defeasible argumentation has long be reviled in literature as being inherently fallible and thus unsuitable to represent expert opinion, Walton (2015) has argued that *"Although such reasoning may often be fallible, it is not always wrong, and such reasoning is very often the predominant or even the only kind of reasoning available for our decision-making...it is not helpful to condemn such[expert evidence as fallacious.*

Rather the problem is to judge in specific cases when an argument from expert opinion can properly be judged as strong, weak or fallacious."

Rainer (2017) has also recognized that *"the software engineering research community would benefit from a greater appreciation of the range of reasoning and knowledge that practitioners use".*

Walton's research (Walton,2015) has also provided strong pointers in refining the data model of the contribution created by outlining the three methods of evaluating defeasible argumentation

- attack the premise
- attack the inference
- present a counterargument

This work makes extensive use of these principles to refine the automated evaluation model to correspond the high-level concepts espoused in the top-down model as well as make a reasonable approximation of the techniques used in manual evaluation.

2.5 Related Work in Academia

2.5.1 Software Engineering research using GitHub

In recent years the adaption of several features relevant to social engineering in significant code hosting platforms has led to increased attention from researchers(Kalliamvakou et al.,2016). Qualitative studies by Begel et al. (2013), Dabbish et al. (2012), and Marlow et al. (2013) have led them to conclude that GitHub users tack the activities of co-contributors and form opinions about their role and potential capabilities. Such evidence-based observations, helps them to contribute in a meaningful way to the project and organize their contribution to complement that of others. Studies by Pham et al. (2013),and Tsay et al. (2012) studied the implications of such social behaviour on project success. Researchers have increasingly used tolls like GHTorrent and Gitminer to study testing patterns (Kochhar et al.,2013), the programming languages used (Bissyandé et al.,2013), issue reporting patterns (Bissyandé et al.,2013), and project success (Tsay et al.,2012).

Another key piece of research was carried out by Hauff & Gousios (2015) where the researchers attempted to match developer profiles from GitHub to job descriptions. Bissyandé et al. (2013) carried out a thorough study of over 100,000 projects through the GitHub API to try and estimate the correlation between the number of issues generated and project characteristics like age, team size etc. Their conclusion that issue reporting was essential to collaborative development in GitHub projects underlined the importance of contribution through issue administration as done in this research. Dabbish et al. (2012) concluded that commit and issue comments were indicative of commit quality and issue severity, which also led to these factors being considered in this dissertation. A study by Guaman et al. (2017) proposed the use of SonarQube and allied software to track code quality in software repositories. This led to the incorporation of Pylint based code analysis in this research for checking the contribution to code quality.

Though Kalliamvakou et al. (2016) have stressed upon the perils of data mining through GitHub, the study of developer behaviour and its implications in project success through GitHub has continued apace. However, most of these studies are qualitative, and little work has been done to quantify project or performance metrics through the data mined from GitHub.

2.5.2 State of the Art Research in measuring developer contribution from GitHub

A primary source of inspiration for the current research was the work of Gousios et al. (2008) in **"Measuring developer contribution from software repository data."** The authors postulated that the shift to agile development practices decreased the relevance of traditional estimation models. Their work centred on the realization that *"Apart from source code, software infrastructures supporting agile and distributed software projects contain traces of developer activity that does not directly affect the product itself but is important for the development process"* and led them to propose a model that was an ensemble of traditional approaches with new parameters mined from software repositories.

Given the importance of assessing contribution to *"monitor the rate of project development, identify implementation bottlenecks and isolate exceptional cases."* the authors propounded that software developers performed a wide variety of activities other than writing code, especially in the domain of Open Source Software (OSS). In attempting to identify indicators of contribution from actions performed in software repositories, the authors considered tasks like defect-resolution and contribution to discussion boards and mailing lists.

Given that there exists no clearly defined model to characterize contribution in software engineering, Gousios et al. (2008) borrowed from the findings of Amor et al. (2006) by using activity to determine contribution. Glass et al. (1992) proposed that "clerical" and "intellectual" types of actions be made distinct when dealing with software development. This was followed by a thorough taxonomic cataloguing of activities performed in every stage of the Software Development Life Cycle (SDLC), which remained a standard reference for both this research as well as the work done by Gousios et al. (2008).

The model ultimately developed by Gousios et al. (2008) considered source code as only a part of the activities done by a developer when estimating contribution.

For a developer d, a function $C(d)$was calculated as the sum of the total LOC coded by the developer and the contribution function $CF(d)$

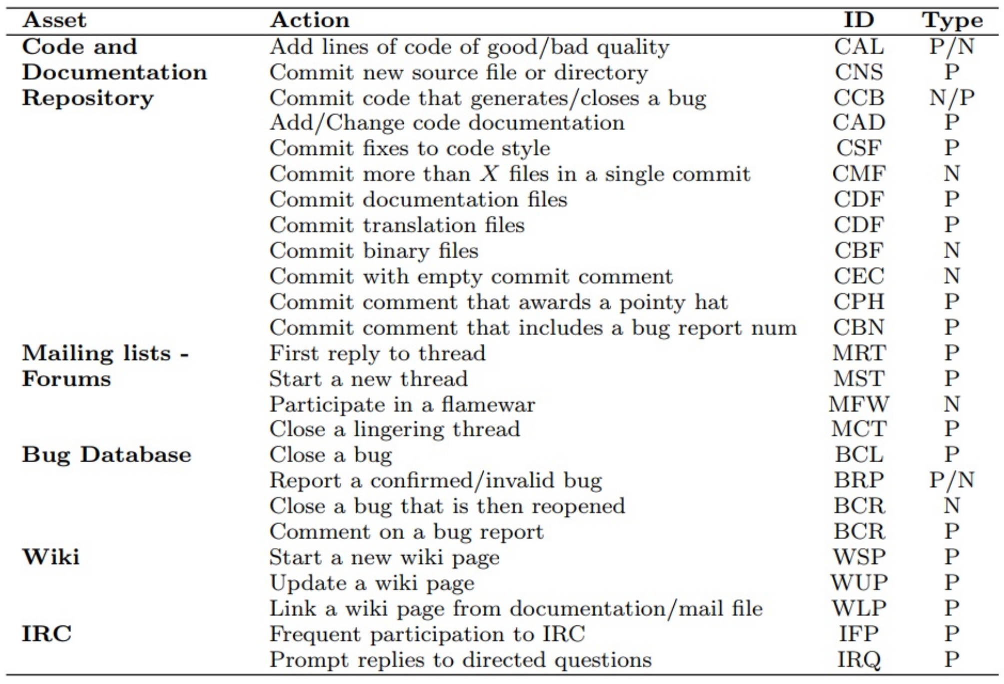
$$\text{C(d)=LoC(d)+CF(d)}$$

Utilizing a hierarchical approach suggested initially by Challa et al. (2011), the authors performed a top-down assessment of activities performed in the project. Some were adjudged to have a positive impact, whereas others were considered negative to the project performance. Figure 2.6 gives details about the project resources, the actions performed on them and their impact as postulated by Gousios et al. (2008) Given the varying importance of each considered activity to the overall project, a weight was assigned, and then this weighted sum was considered as given in the formula below

$$A^{total} = \alpha A^{rep} + \beta A^{ml} + \gamma A^{bug} + \delta A^{wiki} + \epsilon A^{irc}$$

where *"The value A^x denotes the sum of all the events affecting asset x in the project's lifetime"*(Gousios et al.,2008)

The weights α, β. γ, δ, and ϵ are evaluated to be the percentage value of the sum of all events for an asset x. Each asset type sum so calculated is broken down into a



Asset	Action	ID	Type
Code and Documentation Repository	Add lines of code of good/bad quality	CAL	P/N
	Commit new source file or directory	CNS	P
	Commit code that generates/closes a bug	CCB	N/P
	Add/Change code documentation	CAD	P
	Commit fixes to code style	CSF	P
	Commit more than X files in a single commit	CMF	N
	Commit documentation files	CDF	P
	Commit translation files	CDF	P
	Commit binary files	CBF	N
	Commit with empty commit comment	CEC	N
	Commit comment that awards a pointy hat	CPH	P
	Commit comment that includes a bug report num	CBN	P
Mailing lists - Forums	First reply to thread	MRT	P
	Start a new thread	MST	P
	Participate in a flamewar	MFW	N
	Close a lingering thread	MCT	P
Bug Database	Close a bug	BCL	P
	Report a confirmed/invalid bug	BRP	P/N
	Close a bug that is then reopened	BCR	N
	Comment on a bug report	BCR	P
Wiki	Start a new wiki page	WSP	P
	Update a wiki page	WUP	P
	Link a wiki page from documentation/mail file	WLP	P
IRC	Frequent participation to IRC	IFP	P
	Prompt replies to directed questions	IRQ	P

Figure 2.6: Categorization of activities and their impact by Gousios et al. (2008)

weighted sum of the measurable entities. The below formula shows the break-down for A^{rep}.

$$A^{rep} = \sum W_i * A_i$$

A_i denotes the total number of actions and w_i denotes the weight for each action. The model was evaluated through a qualitative study of developer's understanding of whether the weights allocated to the actions were correct, and whether the surveyed developers agreed about the positive or negative impact of certain activities to a project.

Another major challenge for the authors was to define a metric for work done in software engineering terms. Sackman et al. (1966) has proposed a model of using words as a unit of work by considering the entire code base as a body of text and then assigning space-separated elements to represent thought. Other metrics were also defined by Card & Scalzo (1999). However the authors failed to provide any conclusive evidence in their own research about how they solved this problem.

This work(Gousios et al.,2008) provided the basis for conducting the current research on measuring contribution and several concepts espoused here have been incorporated into the project. However, this project begs to differ on the assigning of absolute positive and negative attributes to certain activities like “Commit binary files” as sometimes they may be essential to the project itself (such as when they contain images), and it may require better organization in the repository. Finally special mention must be given to the work of Reddy (2018), who, through his own thesis on **”Sociometrics in Software Engineering Measuring Dedication of Software Developer through GitHub”** proposed many of the fundamental processes used in this research, including creation of data-models to compute abstract concepts in Software Engineering.

2.6 State of The Art Systems in Industry

As compared to academia, industrial applications of measuring engineer productivity, and by extension contribution, have been significant. This section outlines some of the popular applications in this domain, and how they have influenced the course of the research conducted.

The systems described in this section, only partly adhere to the comprehensive approach of calculating contribution as outlined in this research. However, they are significant in the aspect that they break traditional norms for software metric estimation and attempt to communicate essential concepts like developer and team productivity in terms easily understandable to even those outside the domain of computer science.

2.6.1 GitPrime

A system that has come in the news in recent times is GitPrime[6] due to its acquisition by Pluralsight. GitPrime seeks to measure developer productivity in a comprehensive manner and present it in a way understandable to non-IT personnel.

GitPrime allows software repository data to be mined from a variety of Version Control Systems(VCS) like Git, GitHub, GitLab, and Bitbucket and provide a transparency regarding team contribution and activity impact.(Rezvina (2019)) Using a proprietary algorithm GitPrime calculates metrics like

[6]`https://www.gitprime.com/`,accessed:04.08.2018

- Impact: This is a metric designed to calculate the difficulty involved in changing a block of code in the code base. It considers things like LOC and number of files affected to do so.
- Churn: This metric measures the rework or re-factoring done by a developer while coding
- TT100 Raw: This metric represents the time taken by a developer to produce a 100 Lines of Code irrespective of its quality.
- TT100 Productive: This metric is similar to the above one, but considers the time taken to produce 100 Lines of code after code churn has been factored into the equation.

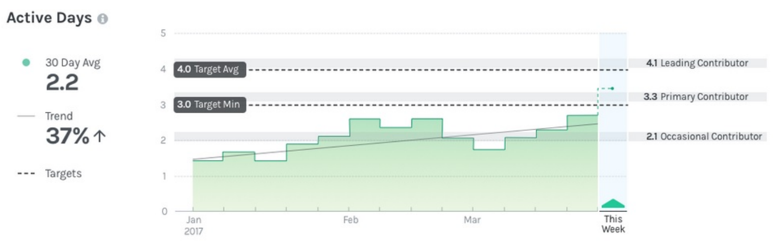


Figure 2.7: GitPrime dashboard from (Rezvina,2019)

GitPrime uses a set of well-designed dashboards to elucidate productivity in terms of its fundamental metrics, and highlight leading contributors to the project. Figure 2.7 shows an example of the GitPrime dashboard

2.6.2 Code Climate Velocity

Velocity[7] is an engineering Intelligence tool, similar to GitPrime which strives to bring transparency to software development processes.

Velocity focuses on pull request analysis in GitHub and is able to mine data from git, GitHub and BitBucket. It focuses on analyzing PR's and their metadata to monitor project progress and developer productivity. It makes the use of "data-enrichment

[7]`https://codeclimate.com/`,accessed:05.08.2018

algorithms" to provide in-depth analysis beyond counts and averages of activities performed.(Rezvina,2019)

Some of the metrics computed by Velocity to track productivity are:

- Impact: Similar to GitPrime, it also measures the difficulty in making changes to the code base.
- Rework: This metric tracks the amount of re-factoring committed on their own code and values this as a measure of developer productivity.
- Pull Request Activity Level: This metric tracks the effort behind integrating a Pull Request to the code base and the activity performed on that Pull Request.
- Review Cycles: This metric details the contribution of Pull Request reviewers in vetting a pull request before its merging with the code base. A higher Review Cycle value tends to indicate potential issues with the Pull Request.

2.6.3 waydev

Another popular tool gaining attention in the market is waydev is waydev[8], which also tracks developer productivity in a way understandable to higher management. Using metrics mined from git repositories it provides summary reports of developer activity and productivity on a weekly basis.

2.6.4 SonarQube

Though SonarQube is not a conventional productivity analysis tool in the sens that it primarily acts as a code quality platform, it bears mention in this study because of its popularity and in-depth approach to code quality analysis.(Udara,2015)

SonarQube[9] is an open source code quality management tool developed in Java and maintained using sonarsource. It can be installed locally for direct or customized use.

This static code analysis tool takes Source Code files along with relevant binary files as its input and performs analysis on the seven axes of code quality :

[8] `https://waydev.co/`,accessed:05.08.2018
[9] `https://www.sonarqube.org/`,accessed:05.08.2018

- Bugs and Potential Bugs
- Coding Standards Breach
- Duplication
- Lack of Unit Tests
- Bad Distribution of Complexity
- Spaghetti Design
- Not Enough or Too Many Comments

SonarQube acts a useful tool for developers to track code quality, for managers as a means of tracking down code changes, and a source of tracking technical debt for non-IT personnel.

Though this research does not used SonarQube for code analysis, a similar tool called Pylint is used, which operates on a similar principle.

This research has also included pointers for calculating Impact from both GitPrime and Velocity and the focus on Pull Request-based Administrative contribution is in part due to Velocity's focus on Pull Requests given the Pull-Request based nature of GitHub.

2.7 Summary

In this chapter, the State of the Art tools and research methods relevant to addressing the Research question is discussed.

A brief background of the history and evolution of software metrics pertinent to the research is provided. The limitations of traditional software estimation methods, in measuring metrics like contribution, are also established. This is important to the research as the aforementioned limitations were a primary source of motivation for conducting this study.

Literature, supporting the estimation of effort in software development through activity performed, is also reviewed to establish a theoretical base for the use of similar methods; in this research, to calculate contribution.

A brief outline of the application of Measurement Theory principles in creating software engineering metrics is discussed as it forms the foundation for creating the metric to calculate contribution.

Since Argumentation Theory using Defeasible Reasoning has been used to create the data-model for calculating contribution, relevant research in this field has been reviewed.

When reviewing related work in academia, the work of Gousios et al. (2008) has been described in some detail as it constitutes (perhaps) the sole body of work which deals with calculating developer contribution using software repository data.

Finally, some popular tools, used to measure team and developer productivity in the industry, have also been discussed.

SOTA	Code-based analysis	Issue Analysis	Pull Request Analysis	IRC/Mail/Chat-based analysis
SonarQube	Y	N	N	N
GitPrime	Y	N	Y	N
Code Climate Velocity	Y	N	Y	N
waydev	Y	N	Y	N
Gousios et al. (2008)	Y	N	N	Y
Amor et al. (2006)	Y	N	N	Y
This Research	**Y**	**Y**	**Y**	**Y**

Table 2.1: Comparison of Current Research with State Of The Art

Most of the academic studies reviewed focused on qualitative assessments of developer behaviour and there remains a substantial gap in academia in the field of assessing the contribution of software engineers in a project. In contrast, the tools being used in industry propose methods and metrics to track developer contribution as well as productivity. One of the goals of this research is to create an automated tool to assess contribution of a developer in a software project, which would conform to the mental model for contribution generated by a particular expert (in this case the author himself). To this end, a brief comparison of the functionalities of the various State-of-the-Art tools and approaches discussed and that of the current research is given in Table 2.1.

Chapter 3

Design and Method

Chapter 3 provides a detailed description of the design decisions for the research by building on the foundations, and stemming from the related work, reviewed in Chapter 2. Firstly a top-down approach is adopted to create a data-model for calculating the contribution of software engineers in a project. This model is created using empirical relationships as observed from the analysis of GitHub repositories, and is fleshed out into a full-scale model for manually calculating contribution. In the second phase, signals are mined from a particular GitHub repository to design, from bottom-up, an automated process for calculating contribution as per the concepts espoused in the high-level data model initially created. Numerical methods are then designed to ensure the consistent calculation of contribution scores across both the models.

The automated model is then constantly refined to ensure that the base metrics mined from GitHub are modelled to provide a reasonable quantification of the concepts proposed in the high-level model

An overview of the Design process followed can be seen in Figure 3.1

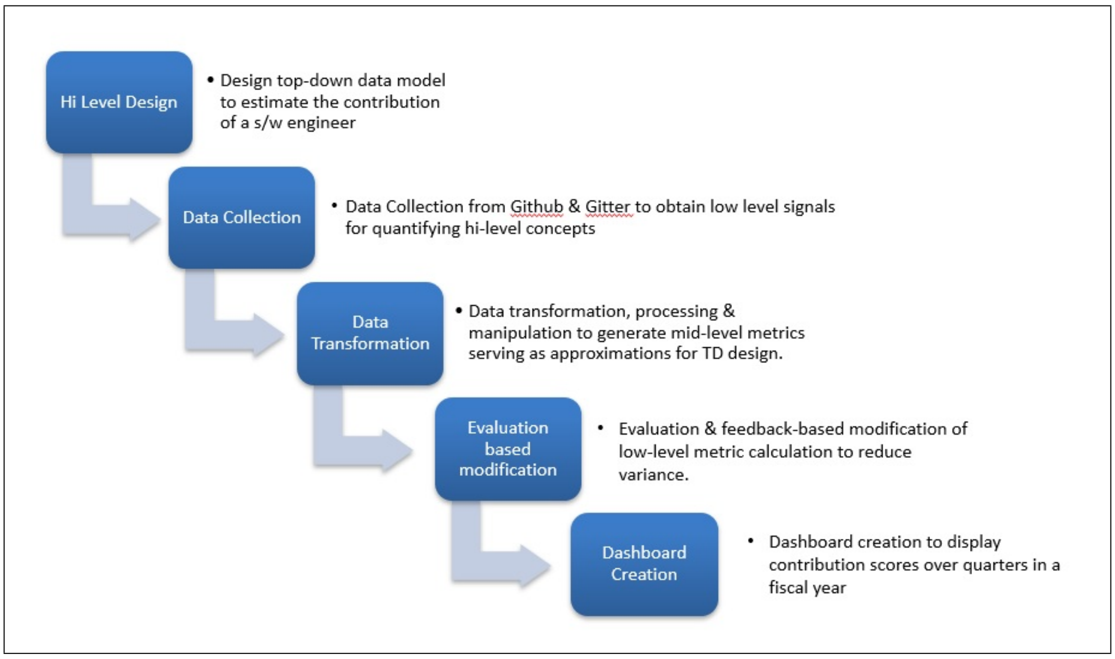

Hi Level Design
• Design top-down data model to estimate the contribution of a s/w engineer
Data Collection
• Data Collection from Github & Gitter to obtain low level signals for quantifying hi-level concepts
Data Transformation
• Data transformation, processing & manipulation to generate mid-level metrics serving as approximations for TD design.
Evaluation based modification
• Evaluation & feedback-based modification of low-level metric calculation to reduce variance.
Dashboard Creation
• Dashboard creation to display contribution scores over quarters in a fiscal year

Figure 3.1: Design and Methods Overview

3.1 Background

This section provides a summary of the theoretical concepts which form the backbone for creating the data-model for measuring contribution.

The purpose of the data-model is to quantify, and thus create a metric, for contribution according to an expert's viewpoint (in this case the author himself). The measure, so constructed, is of the form of a Goal-Question Metric as proposed by Basili & Rombach (1988) (which has been discussed in the previous section). The design methods utilized in this research follow the basic framework for creation of a Goal-Question Metric as shown below:

- List the major goals
- Derive, from the goals, the major questions that must be answered in order to meet said goals

- Decide what must be measured to answer the questions satisfactorily.

In keeping with the spirit of this framework, a high-level categorization of the types of contribution is conducted. This is followed by the creation of a set of parameters, for each category, which symbolize the major components which make up these categories of contribution. These actions help us to build a top-down abstract data model for contribution.

Finally, a bottom-up model of contribution is created using signals mined from GitHub and Gitter to determine which of these signals are useful, whether by themselves or in combination with other signals, in representing contribution as per the high-level data model constructed.

Measurement Theory concepts, discussed in the previous chapter, are also important.They provide the basis by which empirical relations between attributes can be quantified through numerical relations between the same attributes using proper measurement scaling. Figure 2.1 details the process for formal measurement principles to create software metrics.

. The guiding principle behind the design stems from an understanding that contribution be judged by evaluating the entire range of activities done by an engineer in the light of the following :

- The magnitude of the work performed, such as number and frequency of commits made, number of issues raised, etc.

- The impact of the work performed, such as significance of Pull Requests raised in resolving bugs/blockers, Importance of comments made in resolving issues, etc.

- The diversity of activities performed, such as whether an engineer has solely focused on coding as opposed to working on raising issues, engaging in productive conversations in chat boards, fixing bugs, etc

- The scale of the quantity and significance of activities performed in comparison to other participants in the project.

These principles are used to define the rule-set or the knowledge base for the inference mechanism.

3.2 Top-Down Design

This section discusses the Top-down Model for contribution, which views contribution from a management perspective. First the major domains of activity representing the ways in which an engineer could contribute to a project were broadly categorized. The broad domains were then sub-divided into domain-specific activities.

Research by Li et al. (2015), on the various attributes of a ”good software engineer”, brought to light that different attributes were considered for engineers at different hierarchical levels in an organization. This provided the cornerstone for the classification of contribution into three major components

- **Direct Contribution** which deals with contributions done to the code base of the project. All software engineering projects require engineers to do a bare minimum of coding, though the bulk of the work is done either by junior engineers and specialist coders. Though research (Amor et al.,2006) has protested the undue importance given to LOC-based metrics in analyzing developer activity, Gousios et al. (2008) considered contribution through code to be a vital aspect in judging the contribution of a software engineer to a project. This research seeks to measure direct contribution as the contribution of a software engineering through code, by evaluating both the amount of work done, as well as the quality and maintainability of the work.

- **Administrative Contribution** This component deals with the ”housekeeping” tasks associated with maintaining the project, including handling issues and pull requests. The importance of issue reporting and handling has been highlighted in literature by Bissyandé et al. (2013) and reviewed in the previous section. Given that GitHub follows a ”fork and pull” model of development (Kalliamvakou et al.,2016) in which contributors make changes to personal or ”forked” copies of the repository and initiate pull requests when they want the community to incorporate these changes in the main code base, it is important that project maintainers and owners contribute by thoroughly vetting such requests before merging the branches with the code base (Gousios et al.,2014). In most organizations, project leads would be responsible for such activities, whereas open-source projects rely on a few ”project maintainers” to perform these tasks. Often certain stakeholders in a project would dedicate most of their time to such activities in order to ensure smooth project execution.

- **Guidance Contribution** This represents activities done to guide the project team on the correct use of the project artifacts and to engender potential improvements to the project itself. In organizations as well as Open-Source Software development communities, there are experts, either professional consultants or volunteers, who spread awareness about functionalities implemented, best practices for use, and resolve questions about the project in general. They provide the support environment essential for attracting new contributors to the project, and ensure that others benefit from their expertise. Most companies have dedicated technical consultants who perform this role, whereas a few key project owners and stakeholders act as helmsmen for open-source communities.

  Previous research by Gousios et al. (2008) and Amor et al. (2006) have utilized mailing lists and IRC chat-boards to analyze the effectiveness of developer communication. This project uses the Gitter[1] for a similar purpose as Gitter is well-integrated with GitHub and actions of GitHub users can be openly and consistently traced on Gitter.

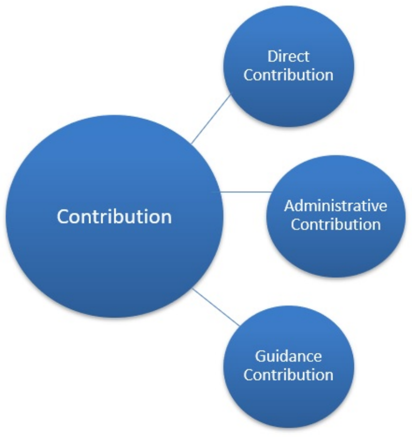


Figure 3.2: Primary Categories of Contribution

[1]`https://gitter.im/home`,accessed:06.05.2018

Figure 3.2 shows the three categories into which contribution is initially divided on the basis of a broad classification of the types of activities performed.

3.2.1 Direct Contribution

Direct Contribution is the contribution done by an engineer, directly through coding, to the project. Unlike traditional methods which focused solely on LOC and allied metrics to quantify such contribution, a more thorough approach is proposed.

Direct Contribution has been categorized to consist of

- Contribution to the Code Base
- Contribution to Code Quality
- Contribution to Code Maintainability

Figure 3.3 illustrates this categorization.

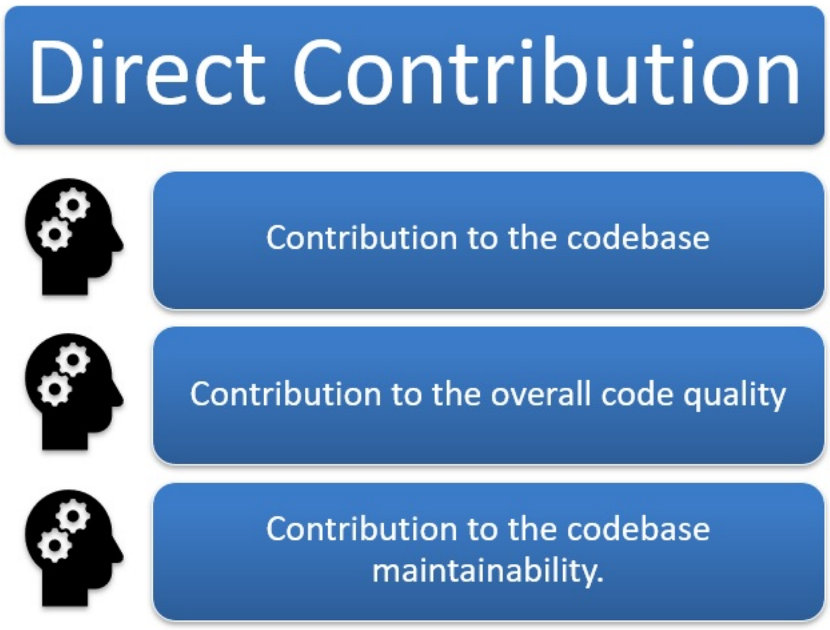

Direct Contribution
Contribution to the codebase
Contribution to the overall code quality
Contribution to the codebase maintainability.

Figure 3.3: Basic Top-Down Model for Direct Contribution

This research recognizes the fact that a simple count of the quantity of code written is pointless if the code has been written badly and if it takes significant effort to maintain that code and make it understandable to others. This model is explained in greater detail in the section 3.3.1

This model sought to refine the high-level concepts espoused in the Top-Down model into concepts that could be analyzed directly by observing the activities performed by developers in a project.

3.2.2 Administrative Contribution

Apart from coding, most engineers in organizations are engaged in tasks like flagging issues, raising defects, code reviews and fixing bugs. Participation in these tasks may not necessarily require a change in the code base itself, or the need to do any coding at all.

However, these tasks are equally important as they help streamline the process of software development and help maintain the "health" of project in general.

In GitHub, most of these activities are performed through Issues and Pull Requests, which have been recognized to be critical to project development by Bissyandé et al. (2013) and Gousios et al. (2014) respectively.

In this research Administrative Contribution is based on contribution to Issue Administration (like flagging significant issues, identifying critical defects and blockers, and tracking and ultimately resolving issues), and Pull Request Administration (like helping add new functionalities to the code base, fixing defects in the code base through Pull Requests, and reviewing Pull Requests before allowing them to be merged to the code base).

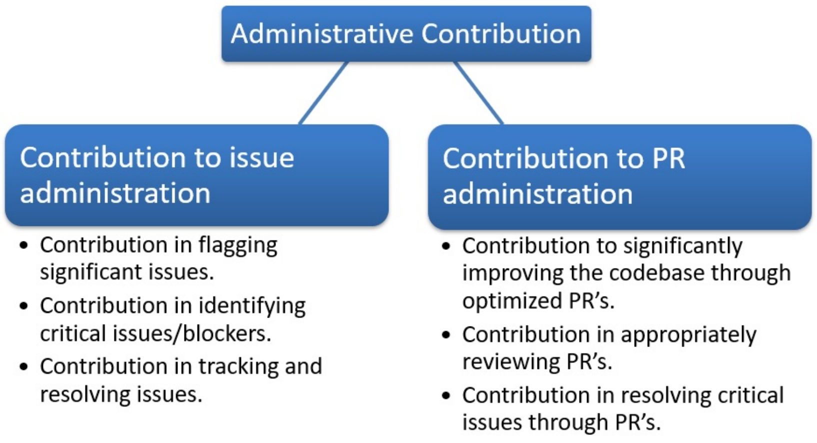

Administrative Contribution
Contribution to issue administration
• Contribution in flagging significant issues.
• Contribution in identifying critical issues/blockers.
• Contribution in tracking and resolving issues.
Contribution to PR administration
• Contribution to significantly improving the codebase through optimized PR's.
• Contribution in appropriately reviewing PR's.
• Contribution in resolving critical issues through PR's.

Figure 3.4: Basic Top-Down Model for Administrative Contribution

Figure 3.4 illustrates this breakdown. A detailed study of the Top-Down model is provided in Section 3.3.2.

3.2.3 Guidance Contribution

All engineers require a guiding hand, from time to time, to set them on the right track. Companies hire technical consultants expressly for this purpose. Senior members, and indeed anyone with relevant expertise can, and often do, provide guidance to others in the project. In Open-Source communities, this is all the more important as the community at large grapples with challenges in using the product. Existing documentation is frequently insufficient to handle particular issues and community members usually pitch in to resolve these problems. The Guidance Contribution of engineers is analyzed from their engagement in the project's official Gitter chat board.Parameters like keeping the community engaged in a positive way, resolving doubts, and raising important issues are considered for this purpose. Figure 3.5 illustrates this concept.

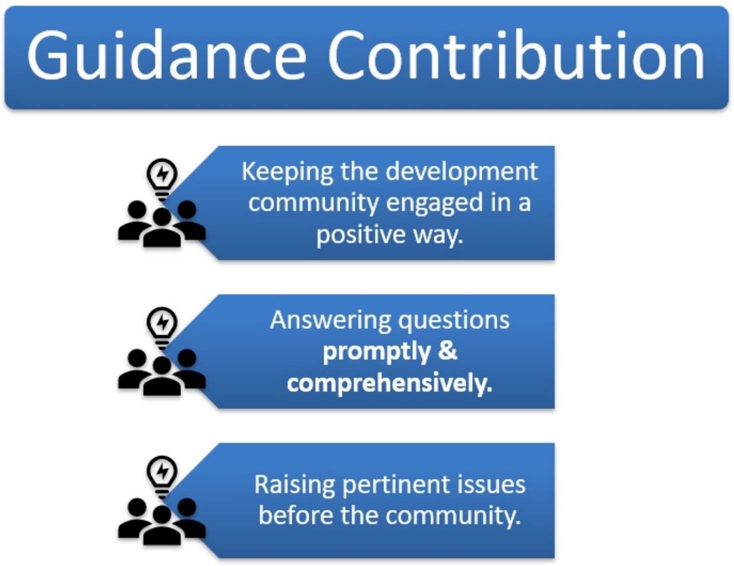

Guidance Contribution
Keeping the development community engaged in a positive way.
Answering questions promptly & comprehensively.
Raising pertinent issues before the community.

Figure 3.5: Basic Top-Down Model for Guidance Contribution

The top-down model discussed in this section deals with the high-level concepts proposed to calculate contribution.This is consistent with the primary stage of the Goal-Question Metric framework discussed in Section 3.1, viz. listing the major goals which define the metric. A more detailed discussion on each of these concepts and the creation of the contribution metric is discussed in the subsequent section.

3.3 Design of Model for Manual Evaluation

Once the Top-Down model was conceptualized, a more comprehensive model was designed to calculate contribution for developers manually. This model was built using the concepts proposed in the preliminary Top-Down Model and later used as a benchmark to evaluate the automated system built to calculate contribution using data mined from GitHub and Gitter. This data-model was built using the Goal-Question Metric framework discussed in Section 3.1, specifically the second stage of the framework which sought to *" Derive from each goal the questions that must be answered to to determine if the goals are being met."*and adhered to the principles of Measurement Theory application for Software engineering as shown in Figure 2.1

The goal of this model was to demonstrate that a contribution score could be calculated for an engineer using manual analysis and evaluation of activities performed in the project. This would later pave the way for an automated approach which would seek to quantify the concepts proposed by this model in terms of metrics obtained from GitHub.

During the conception of this model it was not considered whether the modelling would be suitably replicated from signals obtained in GitHub and Gitter, but stressed on heuristics which can be followed in a manual evaluation process.

The model for Manual Evaluation has three major parts which correspond to the three high-level concepts comprising contribution as set forth in Chapter Three, viz. Direct Contribution, Administrative Contribution and Guidance Contribution.

The underlying principle driving the assignment of contribution scores through this model is the fact that engineers must be evaluated on the quality of their work as well as the amount of work done. Furthermore, those engineers observed performing significantly better than their peers should be rewarded for their efforts, whilst those found to under-perform should be penalized. Also, those engineers performing the greatest diversity of tasks would be rewarded, though those not participating in any task would not be penalized.

The below sections describe in detail the structuring of the model for calculating each of these components

3.3.1 Components of Manually Verifiable Design Model - Direct Contribution

The manual model for calculating direct contribution to the code base emphasizes on estimating the software engineer's contribution to the code base of the project, both in terms of activity done (e.g. in terms of commits done), and the quality of the contribution made.

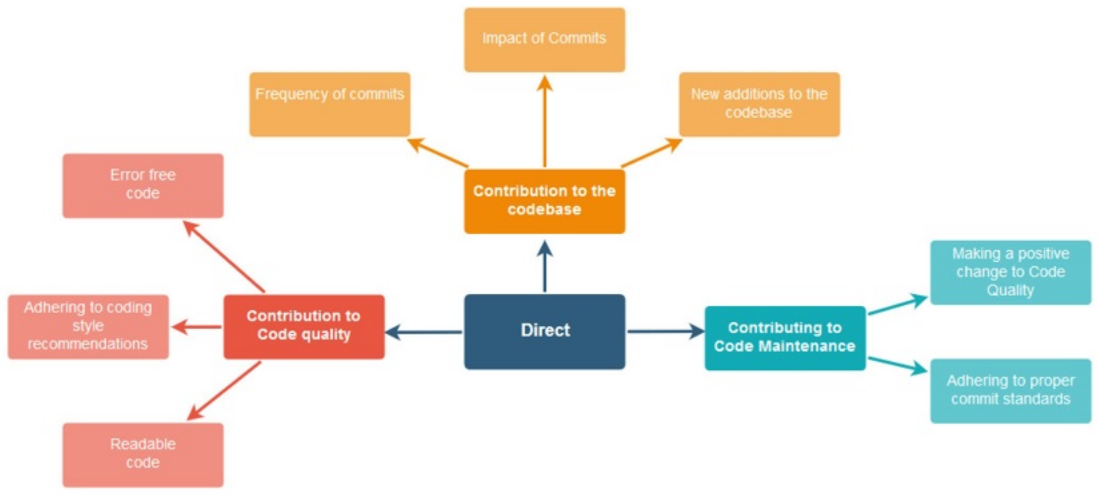

Impact of Commits
Frequency of commits
New additions to the codebase
Error free code
Contribution to the codebase
Making a positive change to Code Quality
Adhering to coding style recommendations
Contribution to Code quality
Direct
Contributing to Code Maintenance
Adhering to proper commit standards
Readable code

Figure 3.6: Detailed Model for Direct Contribution

As illustrated in Figure 3.6 the manually verifiable model for calculating Direct Contribution, depends on three main factors, which depend on sub-factors themselves –

- Contribution to the Code base
- Contribution to Code Quality
- Contribution to Code Maintenance

Contribution to the Codebase

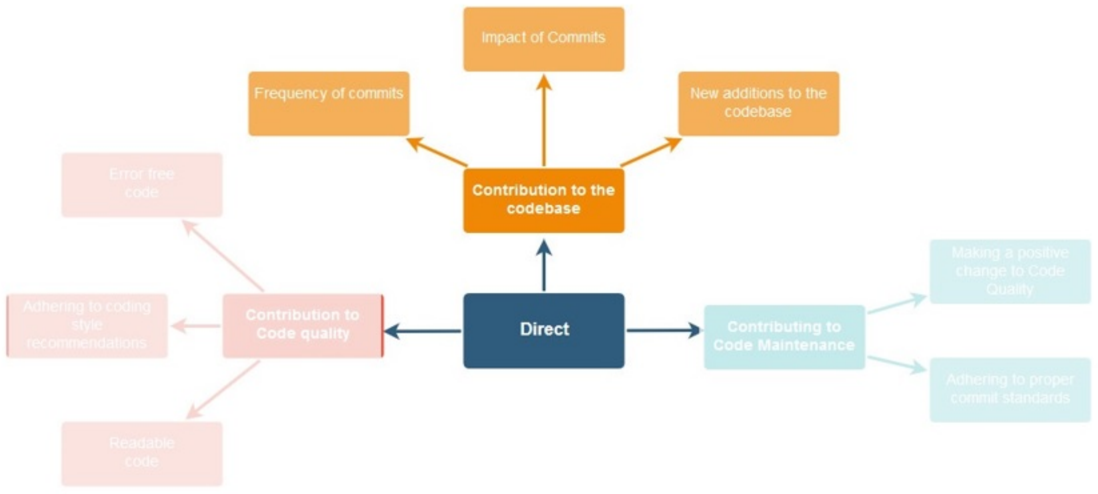

Impact of Commits
Frequency of commits
New additions to the codebase
Error free code
Contribution to the codebase
Making a positive change to Code Quality
Adhering to coding style recommendations
Contribution to Code quality
Direct
Contributing to Code Maintenance
Adhering to proper commit standards
Readable code

Figure 3.7: Modelling Contribution to Code Base

The various factors considered when checking the contribution to the code base, represented in the Figure 3.7 are described as follows:

- The frequency of commits – the frequency of commits done as a measure of contribution to the code base has been established in several research papers (Gousios et al.,2008). This metric is calculated by finding the number of commits done by the engineer over a specified period of time by manually counting commits done or even with the help of the GitHub API.

- Impact of the commits - The concept of Impact is an attempt to estimate the extent to which a commit has affected the code base. GitPrime defines Impact to be a metric which answers the question *"Roughly how much cognitive load did the engineer carry when implementing these changes?*[2] It is estimated from the number of files affected, by a developer, in a commit.

- New functionality added to the code base - Several research papers (Gousios et al.,2014) as well as state-of-the art applications in this domain like GitPrime and Velocity stress on the importance of considering the amount of new functionality added to

[2]https://help.gitprime.com/metrics/what-is-impact, accessed:04.08.2018

a code base while estimating the significance of the commit made, thus it forms a core component of the manual evaluation process of estimating the contribution.

New functionality is considered in terms of number of new files created, as well as the average number of new methods created per file.

Contribution to Code Quality

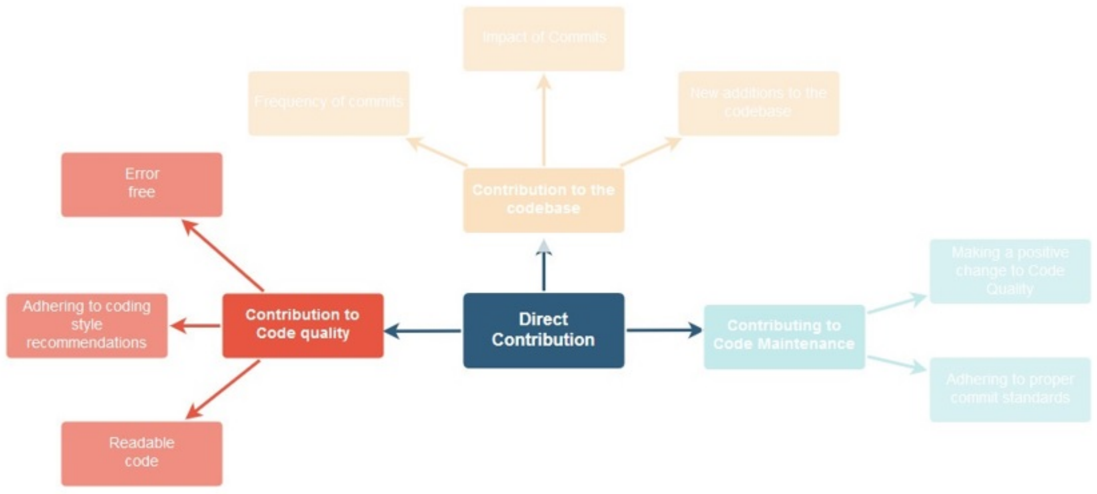

Impact of Commits
Frequency of commits
New additions to the codebase
Error free
Contribution to the codebase
Making a positive change to Code Quality
Adhering to coding style recommendations
Contribution to Code quality
Direct Contribution
Contributing to Code Maintenance
Adhering to proper commit standards
Readable code

Figure 3.8: Modelling Contribution to Code Quality

The concept of peer-code review, as a process of having source code manually inspected by engineers other than the author, has been recognized as a valuable tool for improving software quality (Sadowski et al.,2018). Fagan (1999), helped formalize a structured approach for reviewing code. Time, and research-based evidence, has helped validate the benefits of conducting code reviews, especially in finding potential bugs and vulnerabilities in the code (Shull & Seaman, 2008). However, the cumbersome and time-consuming nature of this process has proven to be a major hindrance in universal use of this process (Rigby,2012). Several organizations have turned to light-weight tools to perform code analysis in order to increase the efficiency of this process.

Considering the recommendation to use code quality estimation tools like SonarQube by Guaman et al. (2017), Pylint is used for the purpose of this research. Figure 3.8 illustrates the detailed model to calculate code quality

Contribution to Code Maintainability

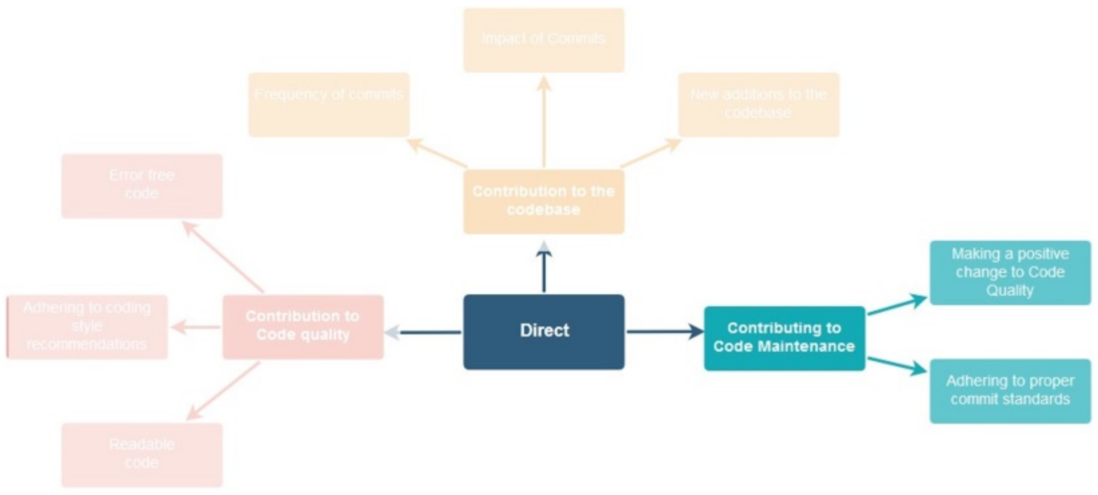

Impact of Commits
Frequency of commits
New additions to the codebase
Error free code
Contribution to the codebase
Making a positive change to Code Quality
Adhering to coding style recommendations
Contribution to Code quality
Direct
Contributing to Code Maintenance
Adhering to proper commit standards
Readable code

Figure 3.9: Modelling Contribution to Code Maintainability

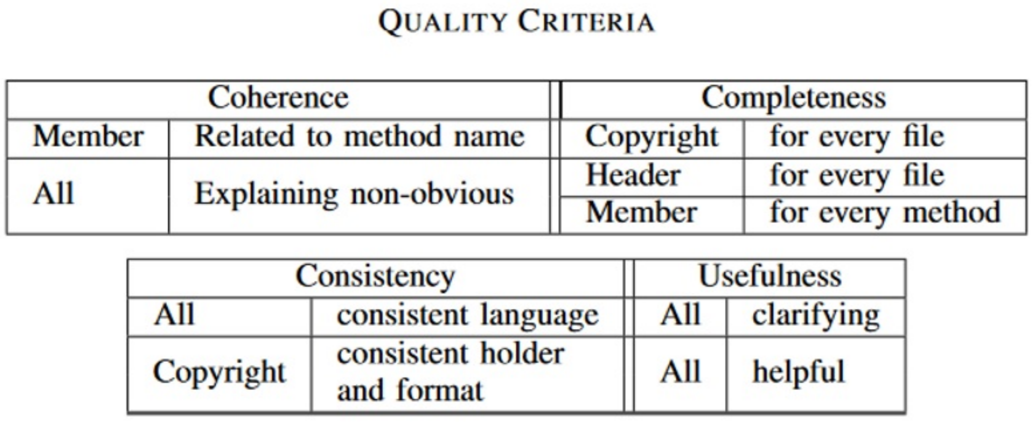
The manual evaluation model for measuring code maintainability focuses on two things.

- The change made to the quality of the code itself
- The quality of commenting done when editing a piece of code

The difference made to the code quality through a commit can be easily adjudged using a code analyzer to check the difference made to a file by multiple commits and identifying committers who have increased or decreased the overall quality of the code.

However, measuring the quality of commenting done is a more nuanced process. The significance of comments in code maintainability cannot be overstated (Steidl et al., 2013). The method for evaluating the quality of comments is based on a procedure laid down in a study by Steidl et al. (2013) in their research on this topic.

This research utilizes the code quality model defined by Steidl et al. (2013) and illustrated in Figure 3.10 for judging commit quality.



QUALITY CRITERIA

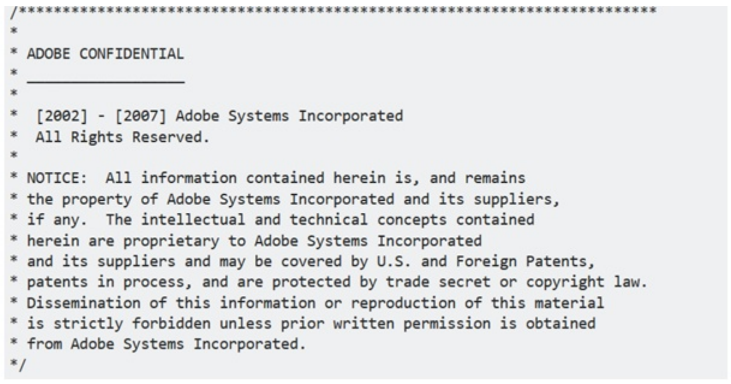
Coherence		Completeness	
Member	Related to method name	Copyright	for every file
All	Explaining non-obvious	Header	for every file
		Member	for every method

Consistency		Usefulness	
All	consistent language	All	clarifying
Copyright	consistent holder and format	All	helpful

Figure 3.10: Code Quality model by Steidl et al. (2013)

To evaluate comments using this model, the comments have to be first categorized into discrete types as proposed by Steidl et al. (2013)

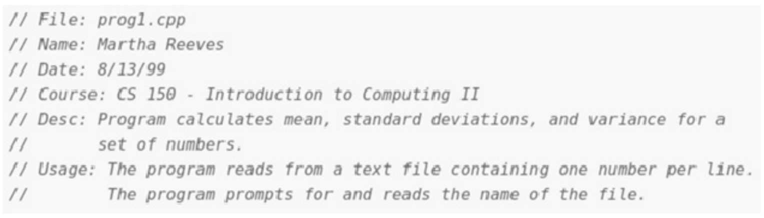
Copyright comments These include information about copyrights or licensing information associated with the code in the source file. They are to be found at the beginnings of each new file. A standard example of this is given below in Figure 3.11 -



```
/**************************************************************************
*
* ADOBE CONFIDENTIAL
* __________________
*
*  [2002] - [2007] Adobe Systems Incorporated
*  All Rights Reserved.
*
* NOTICE:  All information contained herein is, and remains
* the property of Adobe Systems Incorporated and its suppliers,
* if any.  The intellectual and technical concepts contained
* herein are proprietary to Adobe Systems Incorporated
* and its suppliers and may be covered by U.S. and Foreign Patents,
* patents in process, and are protected by trade secret or copyright law.
* Dissemination of this information or reproduction of this material
* is strictly forbidden unless prior written permission is obtained
* from Adobe Systems Incorporated.
*/
```

Figure 3.11: Copyright Comment from Adobe

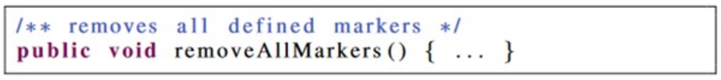
Header Comments These provide a brief description of the functionality of the code and other related information like author name, revision number, etc. A standard example of this is given below in Figure 3.12 -



```
// File: prog1.cpp
// Name: Martha Reeves
// Date: 8/13/99
// Course: CS 150 - Introduction to Computing II
// Desc: Program calculates mean, standard deviations, and variance for a
//       set of numbers.
// Usage: The program reads from a text file containing one number per line.
//        The program prompts for and reads the name of the file.
```

Figure 3.12: Header Comment Example

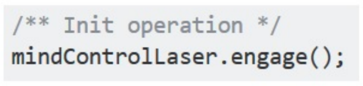
Member comments These describe method functionality and are usually situated before the member declaration. These are especially essential for documenting API's. A standard example of Header comment is given below. A standard example of this is given below in Figure 3.13 -



```java
/** removes all defined markers */
public void removeAllMarkers() { ... }
```

Figure 3.13: Member Comment Example

Inline comments These are found inside a method body and describe details about the implementation.A standard example of Header comment is given below. A standard example of this is given below in Figure 3.14 -



```java
/** Init operation */
mindControlLaser.engage();
```

Figure 3.14: Inline Comment Example

Section comments These are used to identify and provide information about a bunch of methods which have a similar functionality.

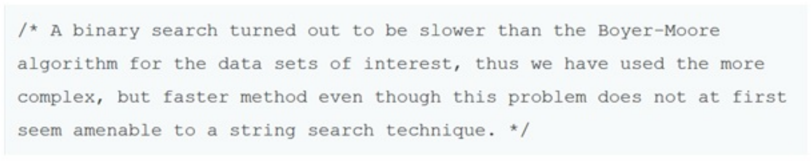
Code comments Sections of code commented out, especially if that code was used for debugging purposes.

Task comments Provides details about tasks, set out by the developer, to be implemented in the particular source file. These are typically to-do comments.

Once the comments have been categorized into the above-mentioned sections, the following quality model is used to assess the quality of the comments and the relevant comment types associated with each quality attribute.

Coherence This attribute deals with how the code and its associated comments are inter-related. For instance, member comments must be related to member names in the member declarations that follow them. Member and inline comments must also provide insights about non-trivial details (beyond the scope of the current code) to enhance understanding of the project implementation and design. This is especially important in an open-source development environment where many different developers work on the same piece of code.

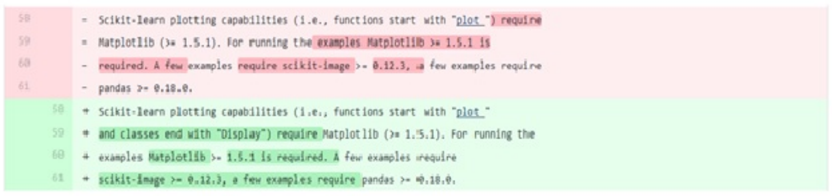
Figure 3.15 is a piece of code[3] which is essentially a member comment which provides detailed implementation notes about a section of code A not so good example of a



```
/* A binary search turned out to be slower than the Boyer-Moore
algorithm for the data sets of interest, thus we have used the more
complex, but faster method even though this problem does not at first
seem amenable to a string search technique. */
```

Figure 3.15: Good Example of a Coherent Comment

coherent comment is one that has failed to capture the essence of the method which it seeks to elucidate. Figure 3.16 is from the scikit-learn repository which. The comment highlighted in red was incomplete in its description of the plotting API and it has been subsequently replaced by a more coherent comment highlighted in green.



```
- Scikit-learn plotting capabilities (i.e., functions start with "plot_") require
- Matplotlib (>= 1.5.1). For running the examples Matplotlib >= 1.5.1 is
- required. A few examples require scikit-image >= 0.12.3, a few examples require
- pandas >= 0.18.0.
+ Scikit-learn plotting capabilities (i.e., functions start with "plot_"
+ and classes end with "Display") require Matplotlib (>= 1.5.1). For running the
+ examples Matplotlib >= 1.5.1 is required. A few examples require
+ scikit-image >= 0.12.3, a few examples require pandas >= 0.18.0.
```

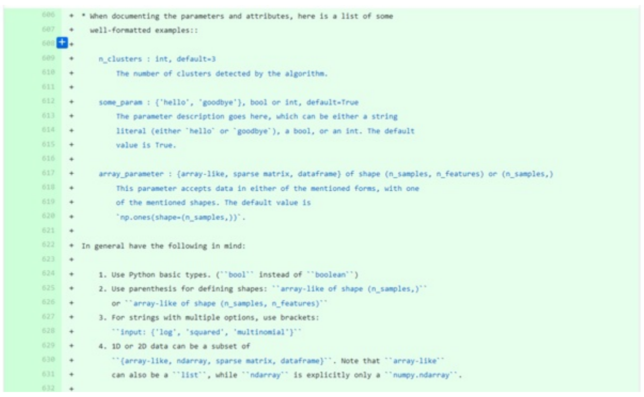
Figure 3.16: Caption

Usefulness This attribute deals with the clarity with which a comment outlines the intent of the code. The heuristic behind this is that deleting a 'useful' comment would make understanding the source code considerable harder.

The below Figure 3.17 from GitHub Scikit-Learn repository represents a set of

[3]https://blog.codinghorror.com/code-tells-you-how-comments-tell-you-why/, accessed : 04.08.2019

comments which describe in detail the procedure to correctly document parameters and attributes and adds considerable value to the body of the document.



```
+ * When documenting the parameters and attributes, here is a list of some
+   well-formatted examples::
+
+     n_clusters : int, default=3
+         The number of clusters detected by the algorithm.
+
+     some_param : {'hello', 'goodbye'}, bool or int, default=True
+         The parameter description goes here, which can be either a string
+         literal (either `hello` or `goodbye`), a bool, or an int. The default
+         value is True.
+
+     array_parameter : {array-like, sparse matrix, dataframe} of shape (n_samples, n_features) or (n_samples,)
+         This parameter accepts data in either of the mentioned forms, with one
+         of the mentioned shapes. The default value is
+         `np.ones(shape=(n_samples,))`.
+
+ In general have the following in mind:
+
+     1. Use Python basic types. (``bool`` instead of ``boolean``)
+     2. Use parenthesis for defining shapes: ``array-like of shape (n_samples,)``
+        or ``array-like of shape (n_samples, n_features)``
+     3. For strings with multiple options, use brackets:
+        ``input: {'log', 'squared', 'multinomial'}``
+     4. 1D or 2D data can be a subset of
+        ``{array-like, ndarray, sparse matrix, dataframe}``. Note that ``array-like``
+        can also be a ``list``, while ``ndarray`` is explicitly only a ``numpy.ndarray``.
+
```

Figure 3.17: Good Example of a Useful Comment

Completeness It helps enforce some global aspects of the system such as placing copyright comments at the start of every new file or placing section comments before a set of related getter and setter methods.

Consistency This attribute concerns itself with ensuring that a similar pattern is followed for commenting throughout the project. This may be to enforce a global coding standard, or the particular standards set in the project.For instance, writing comments in a language not understood by all members of the community is a poor sign of comment consistency.

The detailed description of how this model is quantified to provide a score for Direct Contribution is defined in Chapter 5

3.3.2 Components of Manually Verifiable Design Model - Administrative Contribution

The concept of calculating Administrative Contribution for a software engineer arises out of concerns expressed in research (Kalliamvakou et al., 2016) regarding the fact that there are sections of the project team who, being at senior positions, tend to contribute more by 'housekeeping' activities rather than direct coding. These people are primarily concerned with ensuring that the proper processes are in place to track major issues and that code is added to the main code base only after it is thoroughly vetted.

In GitHub, all concerns raised by the developer community are usually expressed as Issues and since GitHub follows a Pull-Request based model, any addition to the code base must be raised as a Pull Request, which then undergoes scrutiny by the community at large.

The model for administrative contribution has two major components described in Figure 3.18

- Contribution to Issue Administration
- Contribution to Pull Request Administration

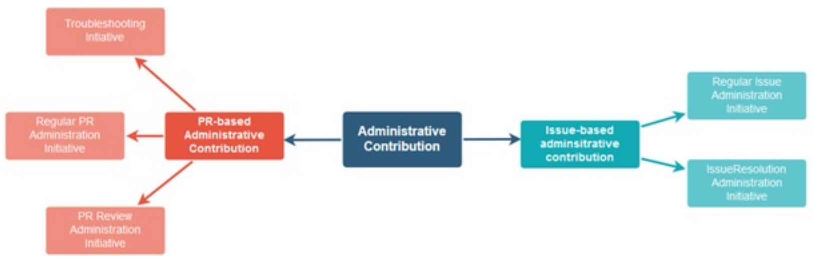

Troubleshooting Intiative
Regular PR Administration Initiative
PR-based Administrative Contribution
PR Review Administration Initiative
Administrative Contribution
Issue-based adminsitrative contribution
Regular Issue Administration Initiative
IssueResolution Administration Initiative

Figure 3.18: Detailed Model for Administrative Contribution

Issue-based administrative contribution

When considering the contribution made to Issue administration we look at a range of factors some of which are based on quantifying activities performed, while others are based on quality-assessment of the activities themselves. In broad terms this was classified as Regular Issue Administration Initiative and Issue Resolution Administration Initiative respectively. This is illustrated in Figure 3.19

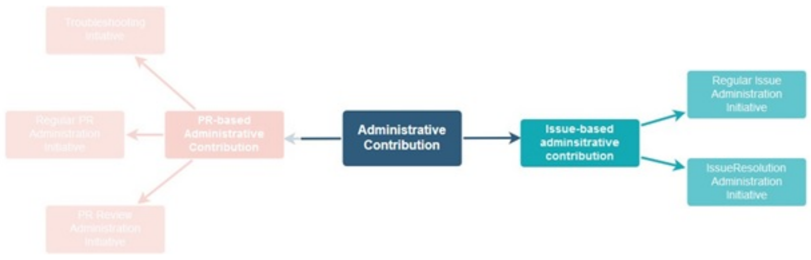

Troubleshooting Initiative
Regular PR Administration Initiative
PR-based Administrative Contribution
PR Review Administration Initiative
Administrative Contribution
Issue-based adminsitrative contribution
Regular Issue Administration Initiative
IssueResolution Administration Initiative

Figure 3.19: Detailed Model for Issue-based Administrative Contribution

Regular Issue Administration Initiative

Regular Issue Administration Initiative considered activities like Raising Issues, Closing Issues, Raising Issues for Bugs/Blockers, Closing Bug/Blocker Issues on their resolution, Tagging issues to milestones, Segregating issues into categories based on their nature.

For calculating Regular Issue Administration Initiative some components are trivial in their choice, such as numbers of issues raised and closed, and they provide a general indicator of the level of engagement of an engineer in bringing to light certain concerns in front of the community and in ensuring that said concerns are marked as closed on their resolution.

Since, GitHub does not have a dedicated Bug-tracking system, any Bugs/blockers are raised as Issues. Initially, the engineer raising the issue may not be certain that the problem encountered is a bug or a blocker, and seeks advice on the same from the community. Others then investigate the matter and decide to label it as a bug or a blocker if they deem fit to do so. This component is included in the manual estimation of contribution as a means to include defect-tracking initiatives by engineers in the project.

Assigning issues to milestones has been considered to be indicative of administrative foresight as it helps in ensuring that the issue is resolved within the time-frame set for completing that milestone.

Another consequence of GitHub not having a dedicated defect-tracking systems is that most concerns raised as issues have to be labelled into discrete categories in order to provide clarifications to the community about the nature of the problem encountered. This is done through labelling the issues with a wide variety of labels. If the labelling is not subsequently changed by another member of the community, then the engineer is considered to have correctly recognized the issue for what it is, for e.g. a bug, a blocker, a regression issue, an API issue etc.

Previous research (Gousios et al., 2014) has focused on trying to understand the significance of such labelling and has put forward evidence that different labels have differing importance relative to the project. For instance an issue labelled 'High Priority' is likely to be more critical to the project than an issue labelled 'Low Priority'. This is consistent with the assigning of criticality and priority to defects in defect-tracking tools like HP ALM.

In order to calculate a score for labelling issues, the categories are first divided into three broad sections, A, B and C, with A being of highest importance and C of least importance. Unlabelled issues were also checked individually and their importance was assigned by the author on the basis of personal understanding.

The categorization is described as follows -

Category A - Bug, Blocker, Good First Issue, Hard, Help Wanted, High Priority, Large Scale

Category B - Build/CI, Documentation, Enhancement, Medium Priority, Moderate, Regression

Category C - All the rest of the categories available.

Issue Resolution Administrative Contribution

The Issue Resolution Administrative Contribution was calculated in a more nuanced manner. It depended on assessing the importance of an engineer in contributing to the resolution of an issue.

When checking for contribution to resolving an issue an in-depth analysis of the comments made by an engineer, and the event-actions performed were analyzed and graded on the basis of the following heuristics, which were influenced by several popular blogs and articles on the topic[4,5] Some of the heuristics involved in this are -

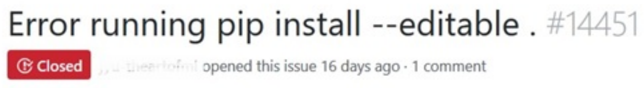
Duplication is to be avoided in raising issues. It is possible that duplicate issues are raised for the same problem, due to the contributors working without much formal communication between them, but they must be identified by the community and closed as soon as possible. Contributors will be penalized if they have raised a duplicate issue, and contributors who have identified duplicate issues will be rewarded for their action.

An issue for an issue One issue needs to be raised for every concern faced. It may seem convenient to raise a single issue for a set of problems which were encountered at the same time. However for efficient defect tracking, different issues must be raised for different aspects of the same problem. A generic example[6] can be as follows

Bad Practice : *"the form needs a subject field and the submit button doesn't work"*

Good Practice: Issue 1 – *"Add subject field to form"* Issue 2 – *"Submit button causes 405 error"*

Appropriate Titles Giving appropriate titles to issues is vital in ensuring that the community gets a sense of what the issue is about. Titles should be brief, specific and provide enough context for a person to get a sense of what the issue is about without having to go through the body of the issue. A good example form the scikit-learn repo itself is given in Figure 3.20



Error running pip install --editable . #14451

Closed [illegible] opened this issue 16 days ago · 1 comment

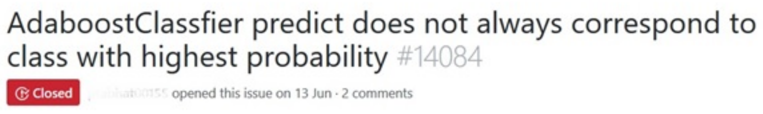
Figure 3.20: Example of Concise Title

[4]https://medium.com/nyc-planning-digital/writing-a-proper-github-issue-97427d62a20f,Accessed: 15.08.2019

[5]https://wiredcraft.com/blog/how-we-write-our-github-issues, Accessed: 15.08.2019

[6]https://medium.com/nyc-planning-digital/writing-a-proper-github-issue-97427d62a20f,Accessed: 15.08.2019

If the issue is about a potential bug in the code, then the title must contain indicators as to what the problem actually is. In Figure 3.21 is shown an issue which highlights a bug in the scikit-learn project.

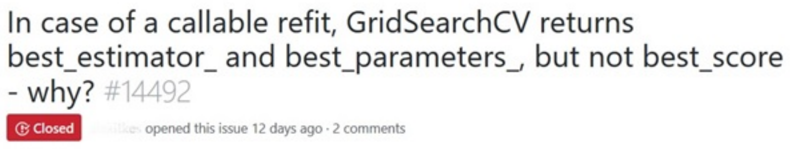


AdaboostClassfier predict does not always correspond to class with highest probability #14084

Closed [redacted] opened this issue on 13 Jun · 2 comments

Figure 3.21: Example of Concise Title

Also, it is a good practice to start writing issue titles by stating facts first and putting opinions later. Figure 3.22 showing an issue from the scikit-learn repo is a good example.



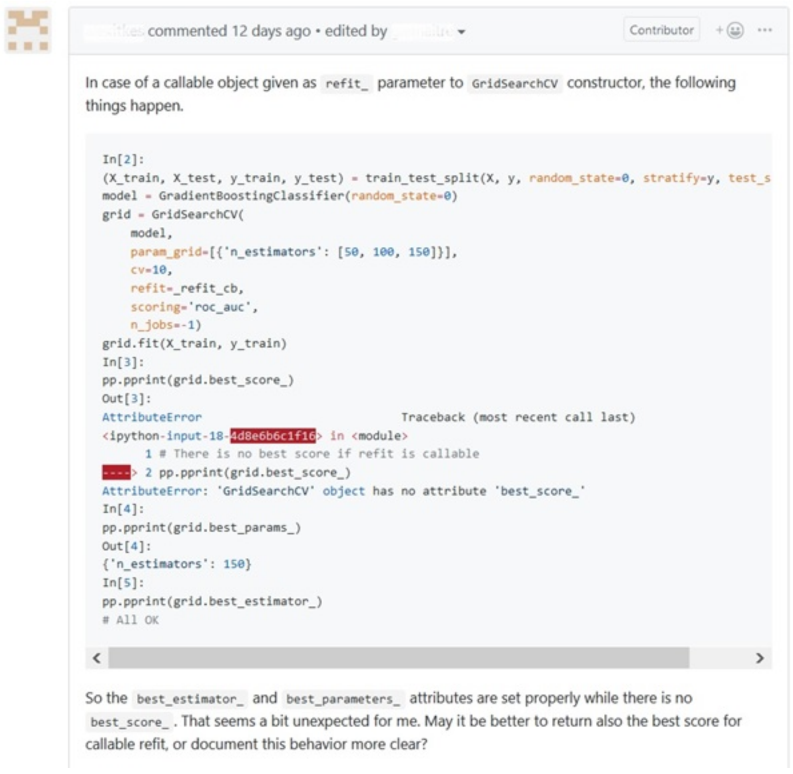
In case of a callable refit, GridSearchCV returns best_estimator_ and best_parameters_, but not best_score - why? #14492

Closed [redacted] opened this issue 12 days ago · 2 comments

Figure 3.22: Example of Factual Title

Following proper formatting Formatting is essential to making the message expressed by an issue as crystal clear as possible. GitHub provides the option to format text using Markdown and has several customized markdown options. Using markdown to format the issue-body allows for consistency in the issues raised. Also, using links for pointing to specific areas of concern or putting relevant screen-shots to augment findings is highly beneficial in improving overall comprehension.

Tagging the people responsible for coding a particular functionality, or those who have helped in issue resolution is also a great way of seeking attention or giving credit where it is due. Engineers following proper markdown styling and bolstering their findings with links and screen-shots have been rewarded for their efforts in the manual evaluation process. Figure 3.23 from the scikit-learn repo illustrates proper formatting and use of code output information to clarify the issue raised.



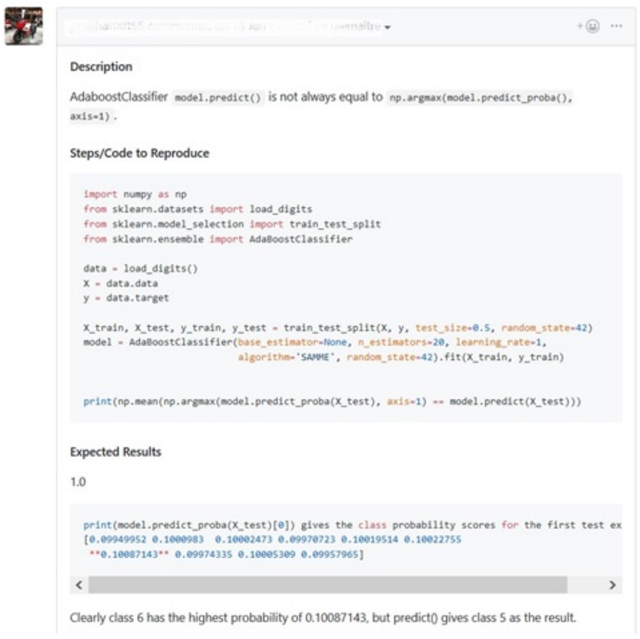
In case of a callable object given as `refit_` parameter to `GridSearchCV` constructor, the following things happen.

```
In[2]:
(X_train, X_test, y_train, y_test) = train_test_split(X, y, random_state=0, stratify=y, test_s
model = GradientBoostingClassifier(random_state=0)
grid = GridSearchCV(
    model,
    param_grid=[{'n_estimators': [50, 100, 150]}],
    cv=10,
    refit=_refit_cb,
    scoring='roc_auc',
    n_jobs=-1)
grid.fit(X_train, y_train)
In[3]:
pp.pprint(grid.best_score_)
Out[3]:
AttributeError                            Traceback (most recent call last)
<ipython-input-18-4d8e6b6c1f16> in <module>
      1 # There is no best score if refit is callable
----> 2 pp.pprint(grid.best_score_)
AttributeError: 'GridSearchCV' object has no attribute 'best_score_'
In[4]:
pp.pprint(grid.best_params_)
Out[4]:
{'n_estimators': 150}
In[5]:
pp.pprint(grid.best_estimator_)
# All OK
```

So the `best_estimator_` and `best_parameters_` attributes are set properly while there is no `best_score_`. That seems a bit unexpected for me. May it be better to return also the best score for callable refit, or document this behavior more clear?

Figure 3.23: Example of good Formatting

Clear Reproduction Steps An issue must have clear reproduction steps associated with it so that the community at large can attempt to reproduce the same issue on their own initiative. Reproduction steps must be precise and easily replicable. Providing details about the expected and actual results also helps in issue resolution. This is especially important in case of Issues dealing with Bugs. Engineers following this standard are rewarded for their actions, and vice-versa. Figure 3.24 is a good example from the scikit-learn repository



Description

AdaboostClassifier `model.predict()` is not always equal to `np.argmax(model.predict_proba(), axis=1)`.

Steps/Code to Reproduce

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier

data = load_digits()
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)
model = AdaBoostClassifier(base_estimator=None, n_estimators=20, learning_rate=1,
                           algorithm='SAMME', random_state=42).fit(X_train, y_train)


print(np.mean(np.argmax(model.predict_proba(X_test), axis=1) == model.predict(X_test)))
```

Expected Results

1.0

```
print(model.predict_proba(X_test)[0]) gives the class probability scores for the first test ex
[0.09949952 0.1000983  0.10002473 0.09970723 0.10019514 0.10022755
 **0.10087143** 0.09974335 0.10005309 0.09957965]
```

Clearly class 6 has the highest probability of 0.10087143, but predict() gives class 5 as the result.

Figure 3.24: Example of Step Reproduction

Use of Standard Templates Wherever possible, it is best to use the standard template for the project used for raising issues to maintain consistency. It also worthwhile to attach the results of investigations carried out to resolve the issue in the issue body. Engineers adhering to the template format and proactively showcasing their own efforts for issue resolution are rewarded and vice versa.

Precise Commenting in Issues When providing comments to help resolve issues, it is of paramount importance to be as precise as possible in asking questions and in giving solutions. Precise correlations must be drawn between the solution suggested and the methods considered, including relevant references.

PR-based administrative contribution

Github utilizes a "fork and pull" model in which developers create personal copies of repositories and generate a pull request when the want their changes to be incorporated into the main code base. This makes reviews of such pull requests an extremely vital part of the Github work-flow. An in-depth examination of how Pull Requests work in GitHub was conducted by Gousios et al. (2014). and this study revealed this pull-request based model to offer scope for *"fast turnaround, increased community engagement and decrease in time needed to incorporate contributions".* This research also proved that the reasons for rejecting pull requests were rarely technical. This chain of thought led to the realization that a considerable amount of effort is required to raise the pull request in a way acceptable to the project maintainers and the community at large, as well as to ensure that the pull request satisfies the functional and technical requirements set for the project. The detailed model for PR-based administrative contribution is described in Figure 3.25 below The PR-based administrative contribution

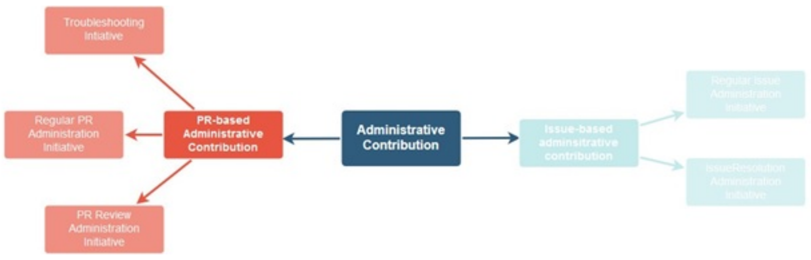

Troubleshooting Initiative
Regular PR Administration Initiative
PR-based Administrative Contribution
PR Review Administration Initiative
Administrative Contribution
Issue-based adminsitrative contribution
Regular Issue Administration Initiative
IssueResolution Administration Initiative

Figure 3.25: Detailed Model for PR-based Administrative Contribution

model in turn depends on three components -

Pull Request Regular Initiative Contribution which comprises of contribution through raising and merging pull requests. These represented fundamental administrative tasks in handling pull requests. A detailed calculation of Pull Request Regular Initiative Contribution from such tasks is provided in chapter 5

Pull Request Troubleshooting Initiative which deals with raising pull requests to fix bugs and blockers. GitHub does not have any dedicated Defect Tracking module and thus, all Bugs/Blockers raised through issues and accepted as such are handled through Pull Requests. A detailed calculation of Pull Request Regular Initiative Contribution from such tasks is provided in chapter 5.

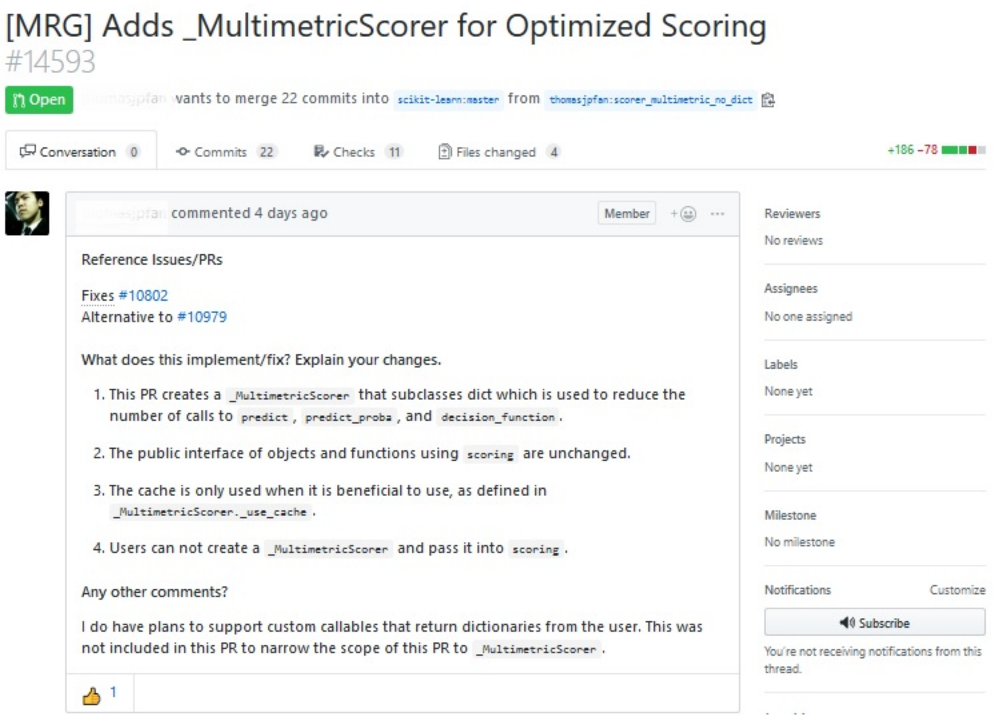
Pull Request Review Initiative deals with the activity of reviewing Pull Requests. In the "fork-and-pull" model followed by GitHub all Pull Requests raised must be thoroughly vetted by the community before merging the code branch with the main code base. A detailed calculation of Pull Request Regular Initiative Contribution from such tasks is provided in chapter 5.

Apart from this another metric is calculated. This is the *Pull Request Importance Score*. This is to represent the relative importance of the contributor raising the Pull Request, as compared to others in this domain. While the other categories dealt with concepts which could be largely represented through quantity counts, this metric seeks to judge the quality of the people raising these requests by monitoring their behaviour during Pull Request handling. An easy of approximating this value is through a check of the number of times a certain pull request was referenced by others. This would give a good indication of the importance of the Pull Request to the code base in general and in turn the importance of the contribution of the person initiating it.

However, the quality of contribution is to be judged in a more nuanced manner by considering the clarity with which the developer has raised the request.

Also considered is the willingness of other members of the community to accept the pull request into the code base. This point is evidenced by the number of corrections to the pull request sought and related clarifications sought.

For instance, a good way to ensure that the contents of your pull request are crystal clear to the community is to document the issue (if any) it seeks to address, and to clearly lay out the scope of changes represented by the pull request. The Figure 3.26 mentions an instance of a well-defined Pull Request from the scikit-learn repo in GitHub



[MRG] Adds _MultimetricScorer for Optimized Scoring #14593

Open [redacted] wants to merge 22 commits into `scikit-learn:master` from `thomasjpfan:scorer_multimetric_no_dict`

[redacted] commented 4 days ago

Reference Issues/PRs

Fixes #10802
Alternative to #10979

What does this implement/fix? Explain your changes.

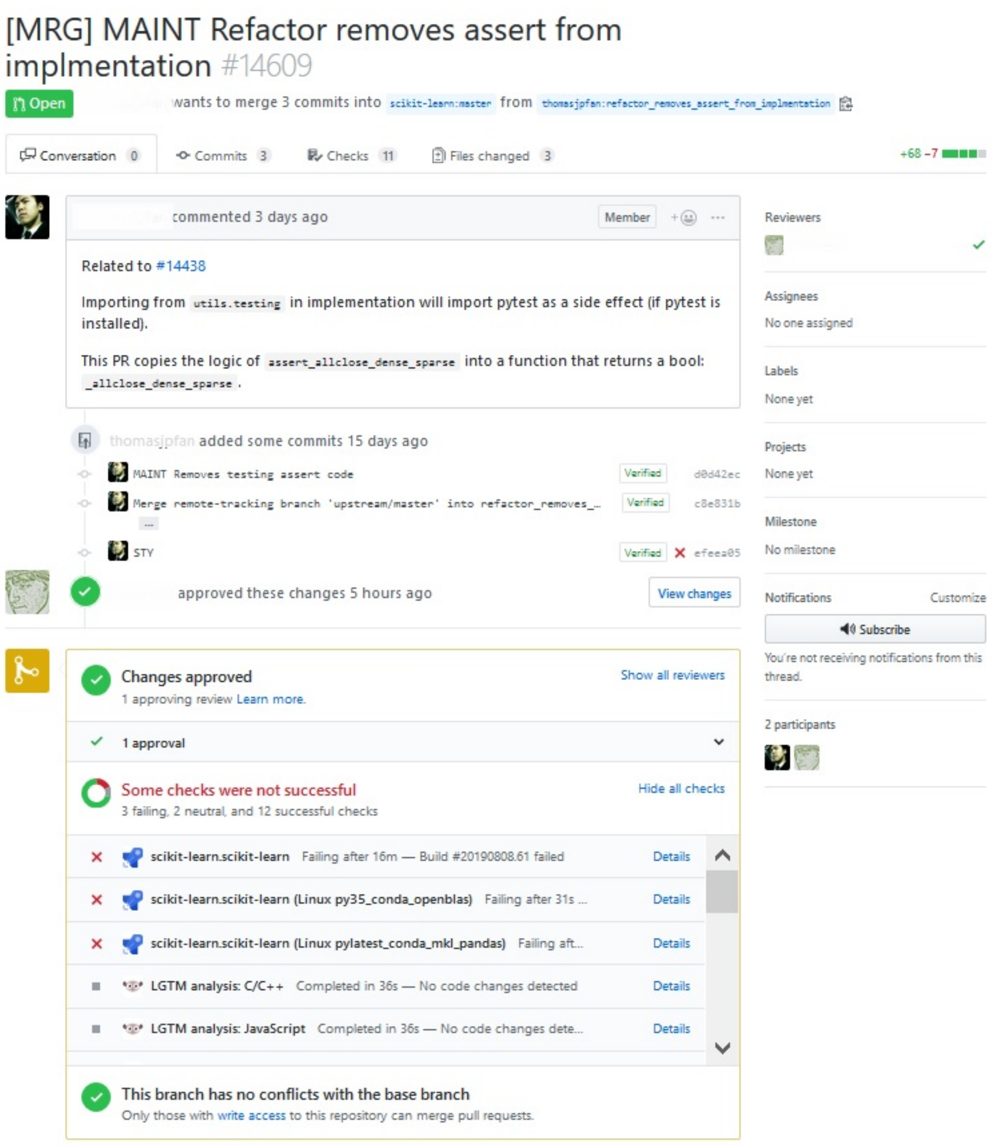
1. This PR creates a `_MultimetricScorer` that subclasses dict which is used to reduce the number of calls to `predict`, `predict_proba`, and `decision_function`.
2. The public interface of objects and functions using `scoring` are unchanged.
3. The cache is only used when it is beneficial to use, as defined in `_MultimetricScorer._use_cache`.
4. Users can not create a `_MultimetricScorer` and pass it into `scoring`.

Any other comments?

I do have plans to support custom callables that return dictionaries from the user. This was not included in this PR to narrow the scope of this PR to `_MultimetricScorer`.

Figure 3.26: A Well defined Pull Request

Another indicator of good contribution is a pull request which has been unanimously accepted by the community as evidenced in Figure 3.27 The quality of the contributions



[MRG] MAINT Refactor removes assert from implmentation #14609

Open [redacted] wants to merge 3 commits into `scikit-learn:master` from `thomasjpfan:refactor_removes_assert_from_implmentation`

Conversation 0 · Commits 3 · Checks 11 · Files changed 3 · +68 −7

[redacted] commented 3 days ago · Member

Related to #14438

Importing from `utils.testing` in implementation will import pytest as a side effect (if pytest is installed).

This PR copies the logic of `assert_allclose_dense_sparse` into a function that returns a bool: `_allclose_dense_sparse`.

[redacted] added some commits 15 days ago

```
MAINT Removes testing assert code                                  Verified   d0d42ec
Merge remote-tracking branch 'upstream/master' into refactor_removes_...   Verified   c8e831b
...
STY                                                                Verified ✗ efeea05
```

[redacted] approved these changes 5 hours ago · View changes

Changes approved · Show all reviewers
1 approving review Learn more.

✓ 1 approval

Some checks were not successful · Hide all checks
3 failing, 2 neutral, and 12 successful checks

✗	scikit-learn.scikit-learn	Failing after 16m — Build #20190808.61 failed	Details
✗	scikit-learn.scikit-learn (Linux py35_conda_openblas)	Failing after 31s ...	Details
✗	scikit-learn.scikit-learn (Linux pylatest_conda_mkl_pandas)	Failing aft...	Details
■	LGTM analysis: C/C++	Completed in 36s — No code changes detected	Details
■	LGTM analysis: JavaScript	Completed in 36s — No code changes dete...	Details

This branch has no conflicts with the base branch
Only those with write access to this repository can merge pull requests.

Reviewers: [redacted] ✓
Assignees: No one assigned
Labels: None yet
Projects: None yet
Milestone: No milestone
Notifications · Customize · Subscribe
You're not receiving notifications from this thread.
2 participants

Figure 3.27: A Unanimously Accepted Pull Request

done by the engineer in the field of Pull Request administration are factored into the contribution score assigned. A detailed example of this is given in Chapter 5

3.3.3 Components of Manually Verifiable Design Model - Guidance Contribution

The core principle behind deducing Guidance contribution is to evaluate the amount of engagement on the Gitter chat board as well as the quality of such engagement. This is illustrated by Figure 3.28 shown below

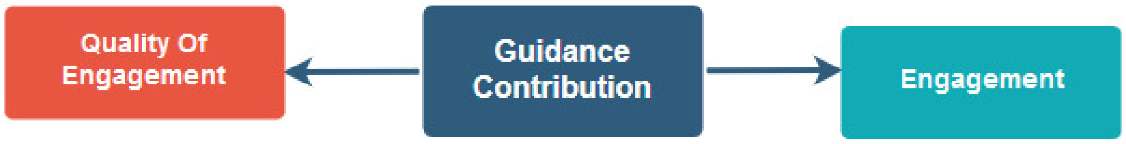


Figure 3.28: Detailed Model for Engagement Contribution

When we deal with the *amount of engagement*, concepts like the number of total messages authored, the number of times mentioned by others in chats, number of questions asked, number of questions answered all come up as vital indicators of quantifying engagement. In short, the footprint of an engineer's activity on the chat board can be measured by counts of these activities. How they sum up to calculate contribution is dealt with in chapter 5.

Quality of engagement is more nuanced and can be determined by judging the quality of conversations, an engineer has participated in, in the chat board. These conversations usually take the form of a "Question asked" - "Question Answered" - "Answer Acknowledgement" pattern, with the particular engineer's contribution to the conversation coming in one or more of these pattern types. The heuristics employed for judging the Quality of Engagement in the conversation are -

- It is considered a positive contribution if the user has avoided spamming in the blog and has stuck to meaningful conversations.
- It is considered a positive contribution if the engineer has promptly answered a question.
- It is considered a positive contribution if the engineer has answered a question thoroughly and with sufficient technical context.
- It is considered a positive contribution if the engineer has received a positive acknowledgement for the answer given.

A detailed explanation of how an engineer is scored on Guidance Contribution is shown in Chapter 5 and is also shown in the Appendix.

Total contribution is calculated by summing up the scores for Direct, Administrative and Guidance contribution from this model

This section discusses, in detail, the components of a model for contribution which can be calculated using a manual analysis of activities performed. The scoring scheme used for calculating the contribution is discussed in detail in chapter 5 and an example of using some these components for calculating contribution is discussed in the Appendix attached. This model was created to allow the contribution of an engineer in a project to be calculated through purely manual effort, although the use of tools like Pylint was considered. Some of the basic metrics dealing with counts of activities can be more easily calculated using the GitHub and Gitter API's to mine the data for substantial periods of time. Later on, when it was decided to use this model to evaluate the performance of the data-model created solely from metrics gathered from GitHub and Gitter, the model adapted to using the standardized methods of creating a Total Score for a metric component as defined later in Section 3.5.3 to quantify the concept of rewarding those engineers observed performing significantly better than their colleagues and penalizing the under-performers and utilized the Inference Mechanisms designed for the Bottom-Up data-model, to formalize the concepts of evaluating engineers on both the quality and quantity of the work performed and rewarding those who performed the greatest diversity of tasks. These concepts were stated at the beginning of this section, and were quantified using the formulae and inference systems designed for the Bottom-Up model to ensure uniformity in Measurement Scaling during evaluation.

3.4 Bottom Up Design

The Bottom Up design initially considers the base metrics mined from GitHub and Gitter and uses them, either individually or in a certain combination, to represent the concepts described in the detailed Top-Down model of contribution described above.

In some cases, a combination of metrics which provide a reasonable representation of the concepts espoused are used, even though they may not exactly correspond to the methods described in the model.

This has been done because of the time and resource constraints of the current research. This section answers the third phase of the framework for the Goal-Question-Metric, viz. "Decide what must be measured to answer the questions satisfactorily".

Reasoning Logic for metric aggregation

In the subsequent section, it will be observed that some higher-level concepts have been characterized by a single signal whereas other require a combination of signals. Before proceeding to analyze the choice of basic signals from Gitter to quantify higher level concepts, it is important to understand the logic behind the design decisions made.

This logic has been constructed using by using the principles of defeasible reasoning, as it does not claim to provide deductive proof of the suitability of the measures created, and accepts that there might be exceptions to these rules.

First, the set of base signals obtained from GitHub and Gitter are considered, then the following algorithm is applied

```
if a single base level signal is sufficient to represent a higher-level concept then
    Allocate the metric to the concept
    Calculate the Percentile-based score and Z-score for the metric
    Calculate the total score for the metric
    Evaluate the correlation with the manually assigned score

    if The correlation is sufficiently strong then
        Finalize the allocation of the signal to the concept
    else
        Obtain more base signals and repeat the process
    end if
else
    Obtain more base signals and repeat the process
end if
```

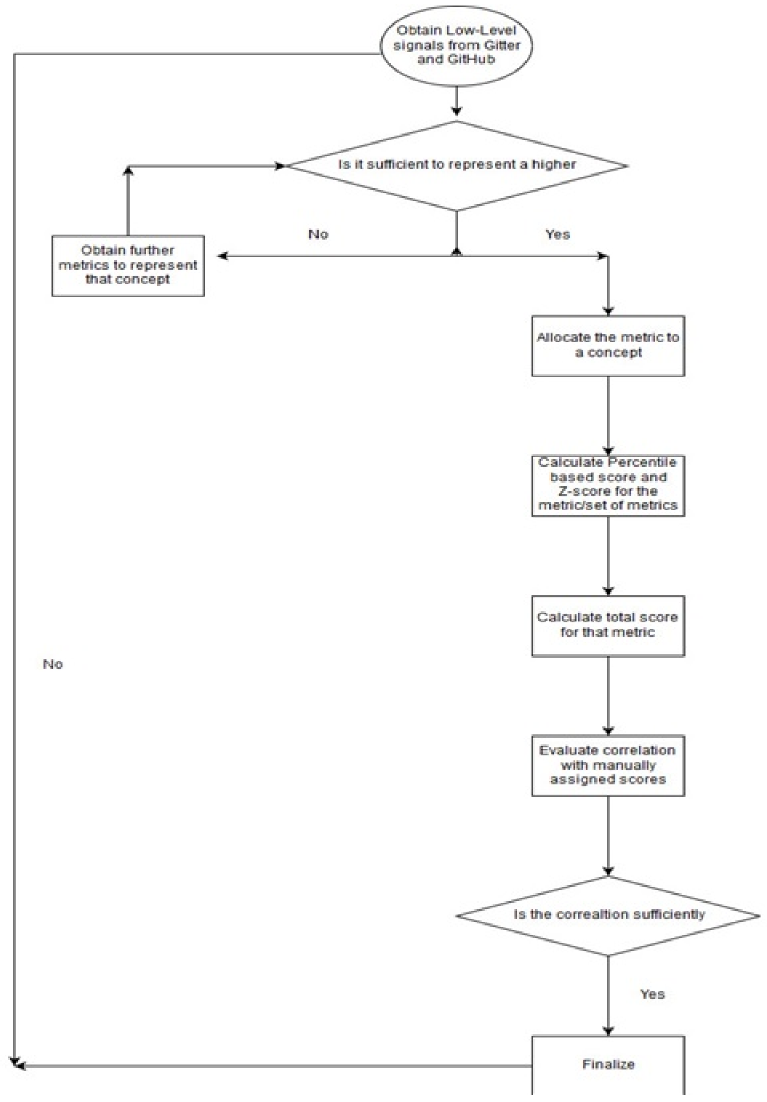

Obtain Low-Level signals from Gitter and GitHub
Is it sufficient to represent a higher
No
Yes
Obtain further metrics to represent that concept
Allocate the metric to a concept
Calculate Percentile based score and Z-score for the metric/set of metrics
Calculate total score for that metric
No
Evaluate correlation with manually assigned scores
Is the correaltion sufficiently
Yes
Finalize

Figure 3.29: Flowchart for creating Bottom-Up Metrics

This can be represented by the flowchart in Figure 3.29. An instance of a single signal used to represent a concept is that of Average Pylint Score to represent Contribution to Code Quality. As already explained in the previous sections, contribution to code quality has been determined to be effectively measured through standard code quality tools as it ensures a thorough adherence to coding standards and is far less cumbersome than a manual verification of the same. This concept was espoused in the data model suitable for manual evaluation as well, and is consistently mapped in the Bottom-Up data model from GitHub signals.

However, contribution to the code base required more than one signal from GitHub for its quantification as it relied on multiple heuristics for its conception.

This algorithm was essential in deciding the combination of signals used to represent model concepts and in refining the automated model to meet the standards of the manually verifiable process designed for calculating contribution.

3.4.1 Bottom-Up Design for Direct Contribution

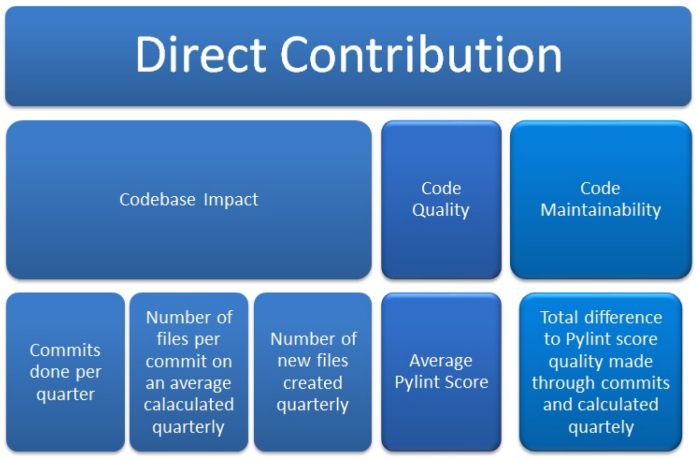

Direct Contribution
Codebase Impact
Code Quality
Code Maintainability
Commits done per quarter
Number of files per commit on an average calaculated quarterly
Number of new files created quarterly
Average Pylint Score
Total difference to Pylint score quality made through commits and calculated quartely

Figure 3.30: Bottom-Up Modelling for Direct Contribution

Figure 3.30 describes the Bottom-Up creation of the model for measuring Direct Contribution from base signals obtained from GitHub.

The bottom-up model describes the signals from GitHub obtained to represented the higher-level concepts representative of Direct Contribution, as stated in the detailed model described in the previous section.

Bottom-Up Design for code base Impact

As detailed in the previous section, contribution to the code base consisted of

- *Frequency of commits done* which was represented by the number of commits done per quarter by a developer. this proved to be completely appropriate to the task at hand.
- *Impact of the commits done* which was represented by the number of files created per commit on an average over a quarter. This is a technique which is followed by State of the Art applications like GitPrime and proved to be completely appropriate to the task at hand.
- *New additions done to the code base* which was represented by the number new files created quarterly. The count of new functions added per file was not considered as it proved to be difficult to identify new functions created in an automated way from the code base and would require a well-tuned text analyzer to be created specifically for this purpose, and was not possible within the limited time frame for this dissertation.

Figure 3.31 represents the Bottom-Up Design model for code base Impact

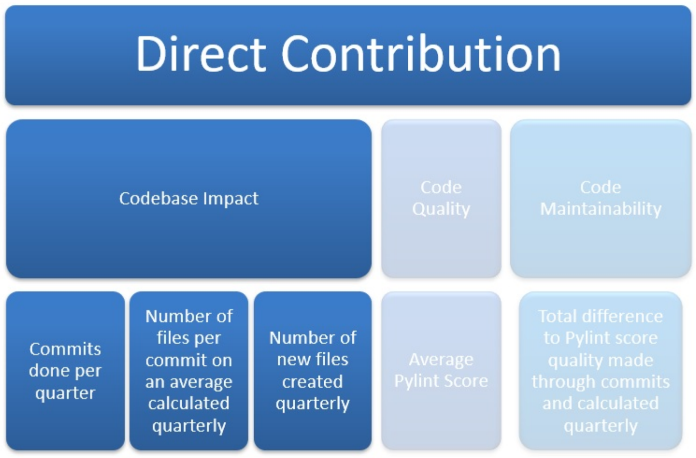

Direct Contribution
Codebase Impact
Code Quality
Code Maintainability
Commits done per quarter
Number of files per commit on an average calculated quarterly
Number of new files created quarterly
Average Pylint Score
Total difference to Pylint score quality made through commits and calculated quarterly

Figure 3.31: Bottom-Up Design model for code base Impact

Bottom-Up Design for Code Quality

As detailed in the previous section, contribution to the code quality consisted of the code quality score for the developer. This was represented by the Average Pylint score for all the commits made by the developer in a quarter. Figure 3.32 represents the Bottom-Up Design model for code quality

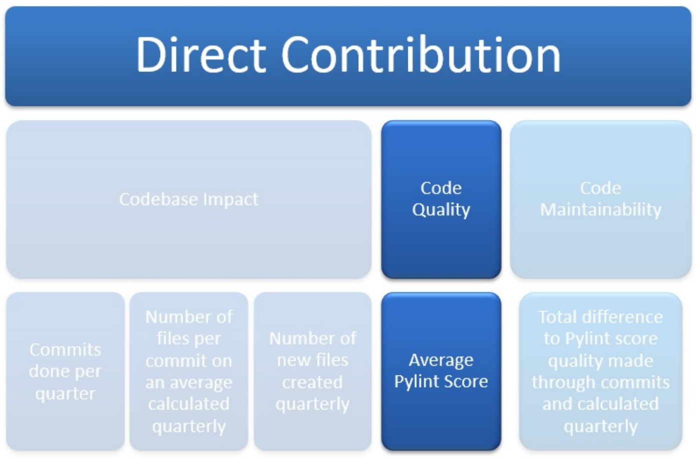

Direct Contribution
Codebase Impact
Code Quality
Code Maintainability
Commits done per quarter
Number of files per commit on an average calculated quarterly
Number of new files created quarterly
Average Pylint Score
Total difference to Pylint score quality made through commits and calculated quarterly

Figure 3.32: Bottom-Up Design model for Code Quality

Bottom-Up Design for Code Maintainability

As detailed in the previous section, contribution to the code maintainability consisted of

- *The change done to the code quality itself* which was calculated by tracking the difference in the Pylint score (for code quality) of individual files across commits and then assigning the differential score to the developer responsible for the commit (and thus the changes made to the file). This method would track whether a developer had increased or decreased the code quality of program files through changes made in the commits. However, this method would only consider files where there had been a code change and not a change in comments done
- *The quality of comments done* was not possible to be determined using automation as a highly-refined text analysis system would first be required to segregate

the commits themselves and then the quality of those commits would also have to be determined. This was de-scoped from the automated system designed as it would not be possible to create this system in the limited time-frame of the project.

Figure 3.33 represents the Bottom-Up Design model for code maintainability

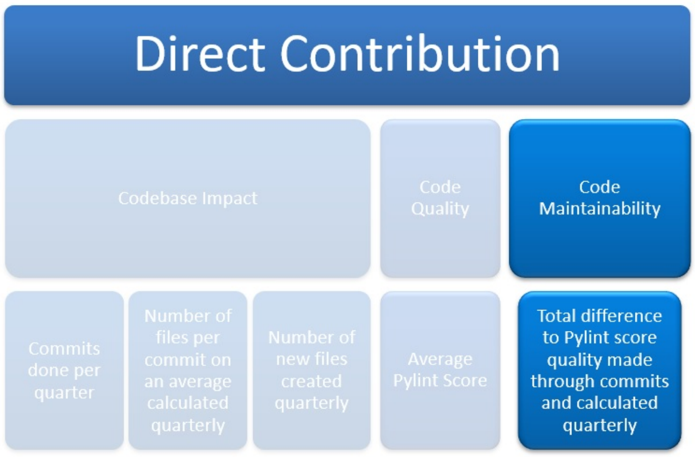

Direct Contribution
Codebase Impact
Code Quality
Code Maintainability
Commits done per quarter
Number of files per commit on an average calculated quarterly
Number of new files created quarterly
Average Pylint Score
Total difference to Pylint score quality made through commits and calculated quarterly

Figure 3.33: Bottom-Up Design model for Code Maintainability

3.4.2 Bottom-Up Design for Administrative Contribution - Issues

The Bottom-Up design for Administrative Contribution was segregated into Administrative contribution for Issues and Pull Requests. This section deals with Administrative Contribution through Issues. Figure 3.34 describes the Bottom-Up creation of the model for measuring Administrative Contribution for Issues from base signals obtained from GitHub.

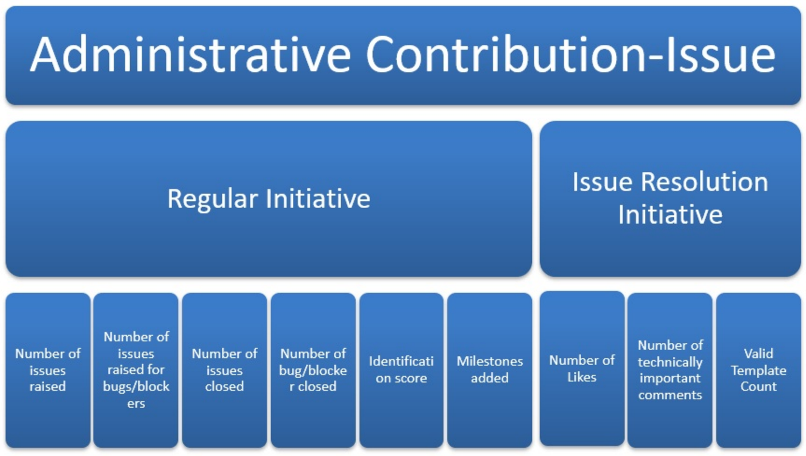

Administrative Contribution-Issue
Regular Initiative
Issue Resolution Initiative
Number of issues raised
Number of issues raised for bugs/blockers
Number of issues closed
Number of bug/blocker closed
Identification score
Milestones added
Number of Likes
Number of technically important comments
Valid Template Count

Figure 3.34: Bottom-Up Modelling for Administrative Contribution - Issues

The bottom-up model describes the signals from GitHub obtained to represented the higher-level concepts representative of Administrative Contribution from Issues, as stated in the detailed model described in the previous section.

Regular Initiative

The model for Regular Initiative Contribution for Administrative Contribution through Issues consists of the following metrics taken from GitHub

- Number of issues raised by a developer in a quarter
- Number of issues raised for Bugs/Blockers in a quarter
- Number of issues closed by a developer in a quarter

- Number of Bug/blocker issues closed
- Number of milestones added to issues by a developer over a quarter
- An identification score for labels added to categorize issues. This identification score is calculated by the formula $\sum(Number\ of\ labels\ identified * Weight-age\ of\ label\ category)$ where labels of category A,B, and C as defined in the previous sections receive weights of 3,2,and 1 respectively.

These metrics calculated from GitHub signals were exact representations of the concepts espoused in the detailed model suitable for manual evaluation. Figure 3.35 describes the Bottom-Up creation of the model for measuring Regular Initiative for Administrative Contribution for Issues from base signals obtained from GitHub.

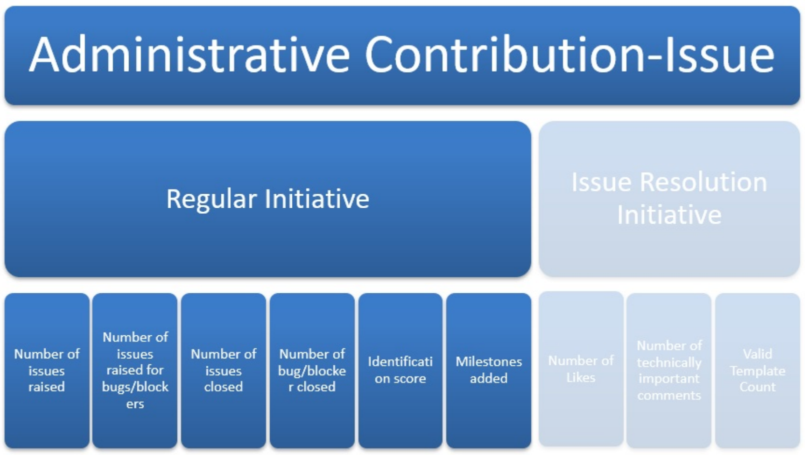

Administrative Contribution-Issue
Regular Initiative
Issue Resolution Initiative
Number of issues raised
Number of issues raised for bugs/blockers
Number of issues closed
Number of bug/blocker closed
Identification score
Milestones added
Number of Likes
Number of technically important comments
Valid Template Count

Figure 3.35: Bottom-Up Modelling for Administrative Contribution - Issues - Regular Initiative

Issue Resolution Initiative

The model for Issue Resolution Initiative Contribution for Administrative Contribution through Issues consists of the following metrics taken from GitHub

- Number of likes received by a developer's comments in Issues over a quarter. This represented the quality of the comment as reflected by its approval from the community.

- Number of technically important comments. This was calculated by counting the number of comments made by a developer which immediately led to an event in the Issue timeline, for instance a labelling activity, a cross-reference to a Pull Request. It was hypothesized that comments which are useful to the resolution of an issue are directly followed by an activity on the basis of that comment.
- Valid template count. This was done by checking in the issues raised by a developer for a standard template in the form of a *Description of the issue*, followed by the *Steps/Code to Reproduce*, *Expected Results*, *Actual Results* and *Thoughts/Comments*.

The heuristics defined for checking developer contribution to issue resolution considered factors like duplication of issues, and appropriate titling which could not be checked through an automated process as these were far too contextual for the application of standard checks. However, a lion's share of the heuristics including proper formatting, presence of reproduction steps, providing precise comments and following standard templates were represented through these metrics mined from GitHub. Figure 3.36 describes the Bottom-Up creation of the model for measuring Issue Resolution Initiative for Administrative Contribution for Issues from base signals obtained from GitHub.

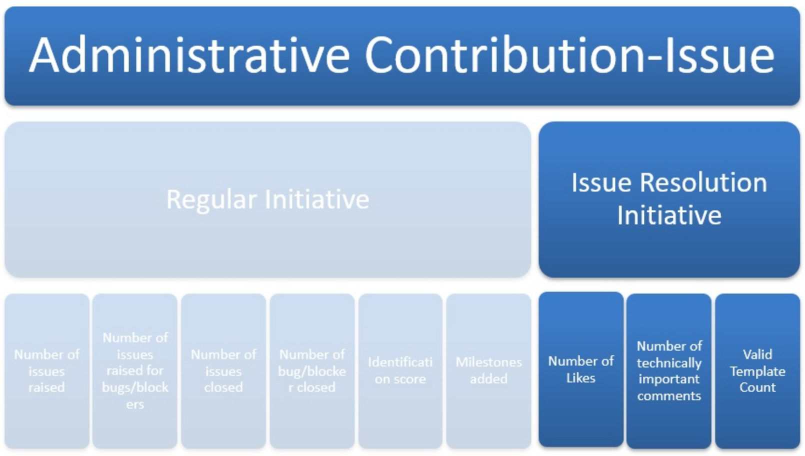

Administrative Contribution-Issue
Regular Initiative
Issue Resolution Initiative
Number of issues raised
Number of issues raised for bugs/blockers
Number of issues closed
Number of bug/blocker closed
Identification score
Milestones added
Number of Likes
Number of technically important comments
Valid Template Count

Figure 3.36: Bottom-Up Modelling for Administrative Contribution - Issues - Issue Resolution Initiative

3.4.3 Bottom-Up Design for Administrative Contribution - Pull Requests

The Bottom-Up design for Administrative Contribution was segregated into Administrative contribution for Issues and Pull Requests. This section deals with Administrative Contribution through Pull Requests. Figure 3.37 describes the Bottom-Up creation of the model for measuring Administrative Contribution for Pull Requests from base signals obtained from GitHub.

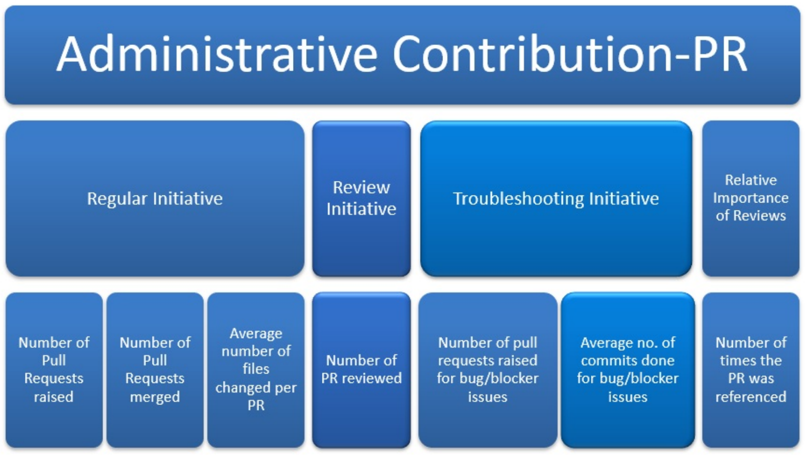

Administrative Contribution-PR
Regular Initiative
Review Initiative
Troubleshooting Initiative
Relative Importance of Reviews
Number of Pull Requests raised
Number of Pull Requests merged
Average number of files changed per PR
Number of PR reviewed
Number of pull requests raised for bug/blocker issues
Average no. of commits done for bug/blocker issues
Number of times the PR was referenced

Figure 3.37: Bottom-Up Modelling for Administrative Contribution - Pull Requests

The bottom-up model describes the signals from GitHub obtained to represented the higher-level concepts representative of Administrative Contribution from Pull Requests, as stated in the detailed model described in the previous section.

Regular Initiative Pull Requests

The modelling for Regular Initiative of Pull Requests was calculated through the following signals from GitHub

- Number of Pull Requests raised by a developer in a quarter
- Number of Pull Requests merged by a developer in a quarter
- Average Number of commits done per Pull Request in a quarter

The Number of Pull Requests raised and merged quantify the level of engagement of the developer in Pull Request administration, and the average number of commits done per Pull Request is a signal to quantify the amount of work done by the developer in the Pull Request. This model is represented by Figure 3.38

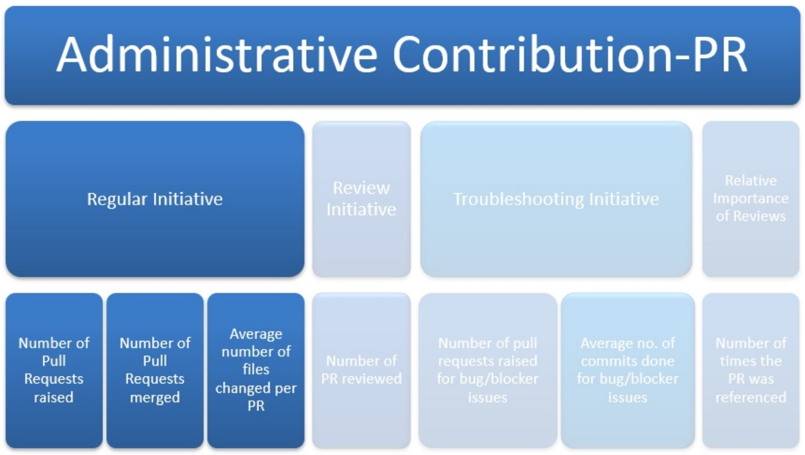

Administrative Contribution-PR
Regular Initiative
Review Initiative
Troubleshooting Initiative
Relative Importance of Reviews
Number of Pull Requests raised
Number of Pull Requests merged
Average number of files changed per PR
Number of PR reviewed
Number of pull requests raised for bug/blocker issues
Average no. of commits done for bug/blocker issues
Number of times the PR was referenced

Figure 3.38: Bottom-Up Modelling for Administrative Contribution - Pull Requests-Regular Initiative

Troubleshooting Initiative Pull Requests

This section defines the model to track the contribution of a developer in resolving defects through Pull Requests. The modelling for Troubleshooting Initiative of Pull Requests was calculated through the following signals from GitHub

- Number of Pull Requests raised by a developer for Bug/Blockers in a quarter
- Average Number of commits done per Pull Request for Bug/Blockers in a quarter

The Number of Pull Requests raised for fixing Bugs/Blockers quantifies the level of engagement of the developer in Pull Request administration, and the average number of commits done per Pull Request is a signal to quantify the amount of work done by the developer in the Pull Request. This model is represented by Figure 5.8

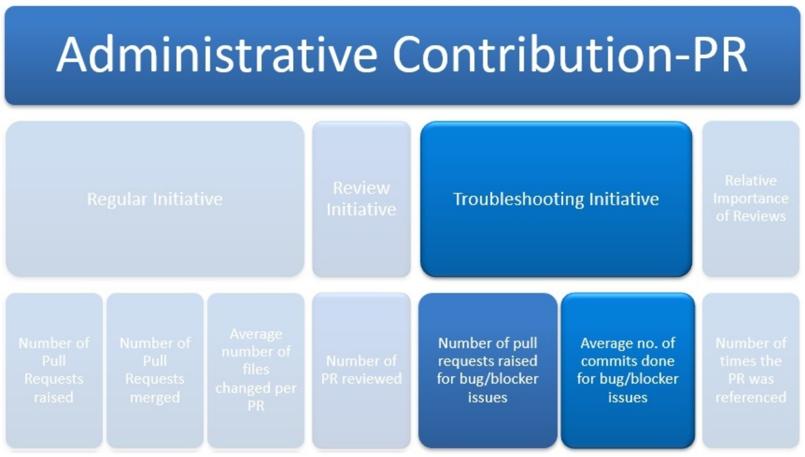

Administrative Contribution-PR
Regular Initiative
Review Initiative
Troubleshooting Initiative
Relative Importance of Reviews
Number of Pull Requests raised
Number of Pull Requests merged
Average number of files changed per PR
Number of PR reviewed
Number of pull requests raised for bug/blocker issues
Average no. of commits done for bug/blocker issues
Number of times the PR was referenced

Figure 3.39: Bottom-Up Modelling for Administrative Contribution - Pull Requests-Troubleshooting Initiative

Review Initiative Pull Requests

This section defines the model to track the contribution of a developer in reviewing Pull Requests. A major activity in Open Source Communities is to review Pull Requests before merging them to the main code base.

This is represented by the Number of Reviews done by a developer over a quarter. This model is represented by Figure 3.40

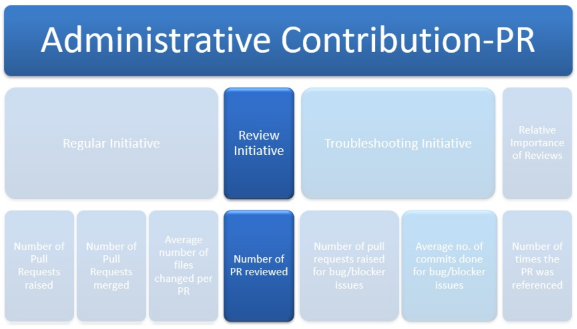

Administrative Contribution-PR
Regular Initiative
Review Initiative
Troubleshooting Initiative
Relative Importance of Reviews
Number of Pull Requests raised
Number of Pull Requests merged
Average number of files changed per PR
Number of PR reviewed
Number of pull requests raised for bug/blocker issues
Average no. of commits done for bug/blocker issues
Number of times the PR was referenced

Figure 3.40: Bottom-Up Modelling for Administrative Contribution - Pull Requests-Review Initiative

Relative Importance of Pull Requests

This section defines the model to estimate the importance of the contribution made by a contributor in raising the Pull Requests. This is measured by the Number of times the Pull Requests made by a developer are cross-referenced by other developers, which serves as an indicator of the importance of that particular Pull Request to other changes made to the code base.

Some other metrics like judging the quality of the Pull Requests raised by an analysis of the Pull Request body, and the unanimity of the community's approval of the pull request (either through direct acceptance, without suggesting changes, or positive comments) could not be estimated within the scope of this approach. This model is represented by Figure 3.41

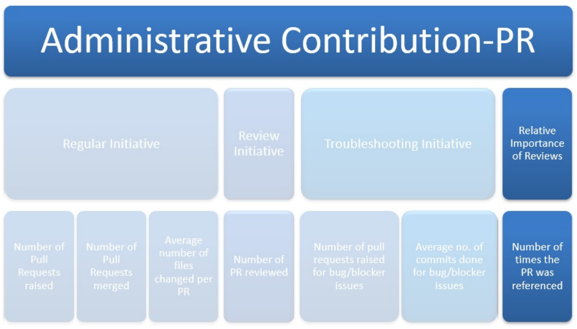

Administrative Contribution-PR
Regular Initiative
Review Initiative
Troubleshooting Initiative
Relative Importance of Reviews
Number of Pull Requests raised
Number of Pull Requests merged
Average number of files changed per PR
Number of PR reviewed
Number of pull requests raised for bug/blocker issues
Average no. of commits done for bug/blocker issues
Number of times the PR was referenced

Figure 3.41: Bottom-Up Modelling for Administrative Contribution - Pull Requests-Relative Importance

3.4.4 Bottom-Up Design for Guidance Contribution

The Bottom-Up design for Administrative Contribution was segregated into scores for Guidance - engagement and Guidance - Quality of engagement

This is represented in the Figure 3.42

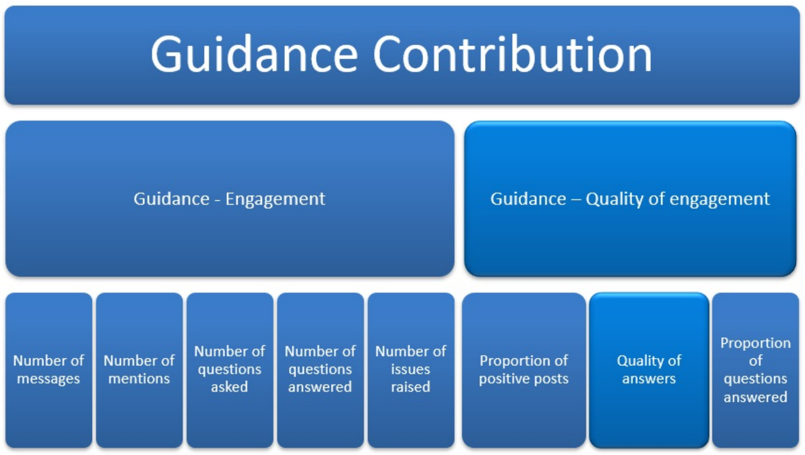

Guidance Contribution
Guidance - Engagement
Guidance – Quality of engagement
Number of messages
Number of mentions
Number of questions asked
Number of questions answered
Number of issues raised
Proportion of positive posts
Quality of answers
Proportion of questions answered

Figure 3.42: Bottom-Up Model for Guidance contribution

For calculating *Guidance-Engagement* which represents the engagement in terms of quantity of relevant activities performed on the chat board, we consider the following

- Number of messages by the engineer in a quarter
- Number of mentions by the engineer in a quarter
- Number of Questions asked by the engineer in a quarter
- Number of questions answered by the engineer in a quarter
- Number of issues raised by the engineer on the chat board (utilizing the issue reference made to the message)

These metrics model the concept of the quantity of work done as a measure of Guidance. This is illustrated by Figure 3.43

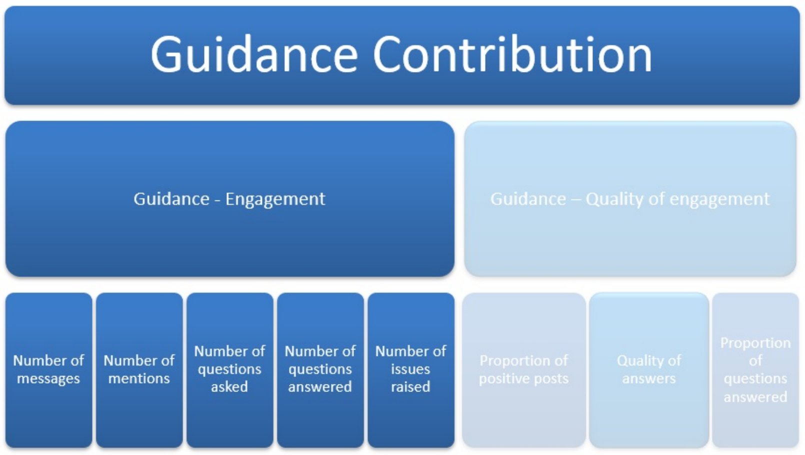

Guidance Contribution
Guidance - Engagement
Guidance – Quality of engagement
Number of messages
Number of mentions
Number of questions asked
Number of questions answered
Number of issues raised
Proportion of positive posts
Quality of answers
Proportion of questions answered

Figure 3.43: Bottom Up Model for Engagement

For calculating *Guidance- Quality of Engagement* which represents the engagement in terms of the quality of the relevant activities performed on the chat board, we consider the following

- Proportion of positive posts made by the engineer in a quarter. This is obtained by a ratio of the number of posts which returned a positive sentiment to the number of posts which returned a negative sentiment.

- Quality of answers. This is obtained as a count of the number of acknowledgements to the engineer's posts which returned a positive sentiment.

- Proportion of Questions answered. This returns a ratio of the number of times the engineer was mentioned in a question to the number of times an answer was given for that question.

These metrics model the concept of the quality of work done as a measure of Guidance. This is illustrated by Figure 3.44

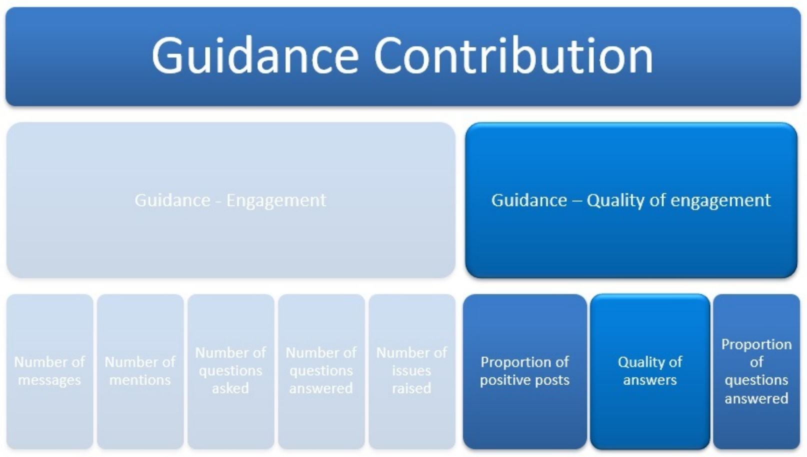

Guidance Contribution
Guidance - Engagement
Guidance – Quality of engagement
Number of messages
Number of mentions
Number of questions asked
Number of questions answered
Number of issues raised
Proportion of positive posts
Quality of answers
Proportion of questions answered

Figure 3.44: Bottom Up Model for Quality of engagement

Some simple heuristics were employed to check whether a chat was a question or an answer

- A question can be identified by the presence of question marks, or if the sentence begins with a "WH" word like "What", "Why", "Which", "Who", "When", etc.
- To identify an answer, the chat directly beneath a question is considered, as this has the highest probability of being the answer. Though there are several exceptions to this rule. Sometimes the question itself is mentioned as "¿ ¡¡Question body¿¿ in Gitter, and this helps to identify the answer.
- Acknowledgements usually follow the answer.

3.5 Inference Mechanism

The below sections define the Inference Mechanisms used to quantify contribution from the components defined in the Bottom-Up model. Expert systems tend to replicate the decision-making abilities of human experts by using a Knowledge Base or a set of rules (primarily as a set of If-Else statements) to solve complex scenarios. A similar

Knowledge base has been used for the three components of contribution with minor variations in each, and are described individually in the respective sections.

A standard mathematical transformation is used across the Inference Mechanism as it maintains the uniformity of measurements and helps scale all the measures to a single standard. The below sections define the standard mathematical formulae used and the Inference Procedure for the contribution categories.

3.5.1 Core Mathematical Formulae Used

As defined in Figure 3.29, once a signal or a set of signals are assigned to represent a concept, they undergo a standard mathematical transformation. These calculations are used across both Manual and Automated calculation of Contribution in order to enforce uniformity of values.

Percentile score

First, each raw-value obtained as a signal from GitHub is given a percentile score. This percentile score is calculated as follows -

- Calculate the First Quartile (Q1), Second Quartile (Q2) and Third Quartile (Q3) values for the entire distribution of raw values for that signal.
- Then assign a score of 1, 2, 3, or 4 based on whether the raw value is less than or equal to the Q1 value, greater than the Q1 value but less than or equal to the Q2 value, greater than the Q2 value but less than or equal to the Q3 value, and greater than the Q3 value respectively.

Z-Score

The Z-score represents the distance, in terms of number of standard deviations, a data point is away from the mean. The Z-score for the raw values is calculated using the formula
$Zscore = (Data - \mu)/(\sigma)$

Total Score

The Total score for the metric is then calculated by multiplying the Percentile score by -1,1,2, or 4 based on whether the Z-score is less than -1, greater than -1 but less than or equal to +1, greater than +1 but less than or equal +2, and greater than +2 respectively.

The reasoning behind this set of mathematical transformations is to ensure that the developers are not only graded according to their relative rank in the population, but are also awarded for performing better or worse than others in the same rank group. For instance, someone with a score of 2 and someone with a score of 20 fall in the same rank group and receive the same percentile score, however the person with a score of 2 will get a lower total score due to a lower Z-score.

3.5.2 Inference Procedure for Direct Contribution

The below section discusses the procedure to calculate Direct Contribution.

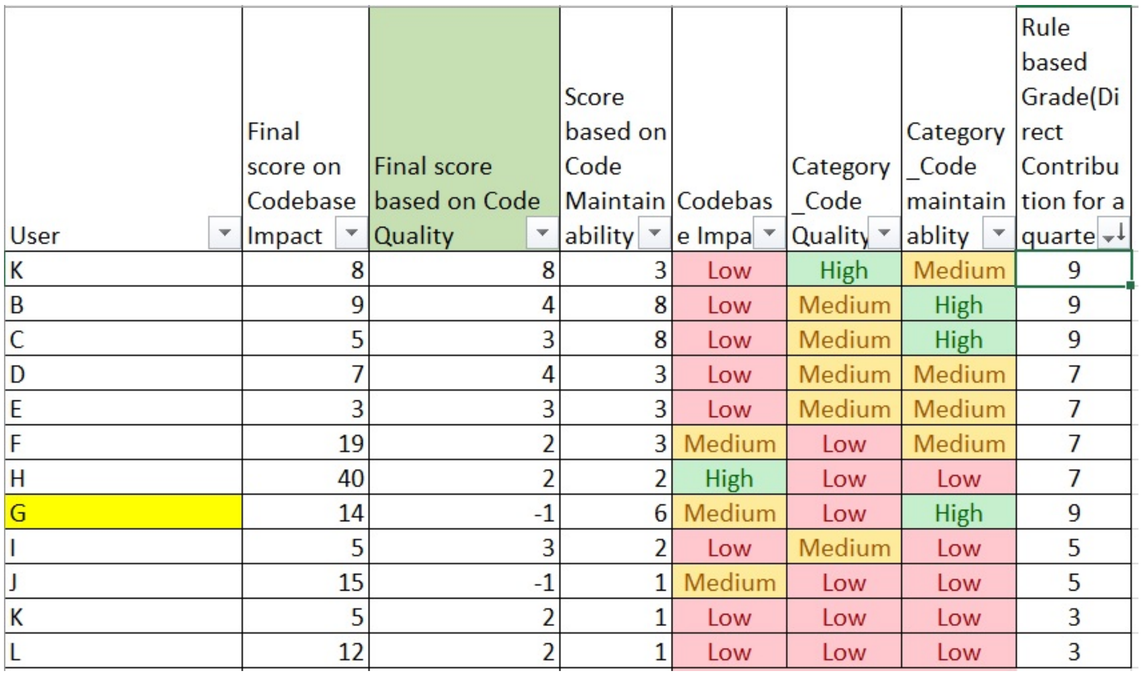
For the raw values obtained for Commits done per quarter, Number of files affected per commit calculated quarterly, and Number of new files created quarterly the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. The total scores obtained are then added to get the *Total Score for Code base Impact contribution.* Similarly total scores for Average Pylint score, and total difference to Pylint score made through commits and calculated quarterly are obtained to get the *Total Score for Code Quality contribution*, and *Total score for Code Maintainability* respectively.

Once the scores for *Total Score for Code base Impact contribution*, *Total Score for Code Quality contribution*, and *Total score for Code Maintainability* are obtained the following set of rules are applied

```
Divide the population under consider into thirds
if the total score for a component falls into the 1st third for the population then
    Allocate a label of Low and a graded value of 1 for that sub-component
else if the total score for a component falls into the 2nd third for the population
then
    Allocate a label of Medium and a graded value of 3 for that sub-component
```

```
else if the total score for a component falls into the final third for the population
then
    Allocate a label of High and a graded value of 5 for that sub-component
end if
```

Finally the graded values obtained for each of the sub-components are added to get the direct Contribution score for the engineer. Figure 3.45 illustrates this concept.



User	Final score on Codebase Impact	Final score based on Code Quality	Score based on Code Maintain ability	Codebas e Impa	Category _Code Quality	Category _Code maintain ablity	Rule based Grade(Di rect Contribu tion for a quarte
K	8	8	3	Low	High	Medium	9
B	9	4	8	Low	Medium	High	9
C	5	3	8	Low	Medium	High	9
D	7	4	3	Low	Medium	Medium	7
E	3	3	3	Low	Medium	Medium	7
F	19	2	3	Medium	Low	Medium	7
H	40	2	2	High	Low	Low	7
G	14	-1	6	Medium	Low	High	9
I	5	3	2	Low	Medium	Low	5
J	15	-1	1	Medium	Low	Low	5
K	5	2	1	Low	Low	Low	3
L	12	2	1	Low	Low	Low	3

Figure 3.45: Inference Mechanism to calculate Direct Contribution

3.5.3 Inference Procedure for Administrative Contribution

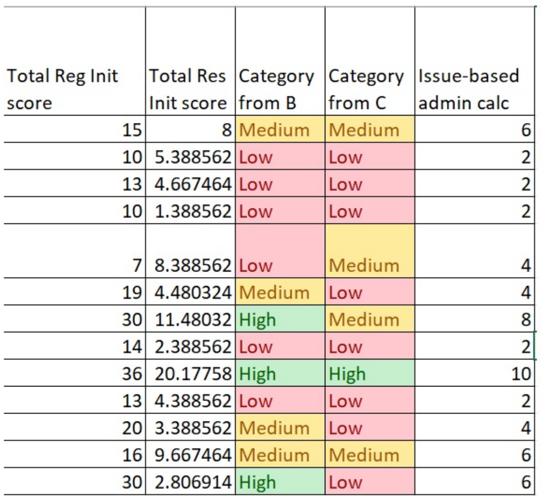
The below section discusses the procedure to calculate Administrative Contribution.

First Administrative Contribution score for Issues is calculated For the raw values obtained for Number of issues raised, Number of issues raised for bugs/blockers, and Number of issues closed, Number of Bugs/Blockers closed, Identification score, and Number of milestones added the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Regular Initiatives in Issue Contribution*

Similarly, the raw values obtained for Number of likes, Number of technically important comments, and Valid Template count the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Issue Resolution Initiatives in Issue Contribution*

Once these values are obtained the algorithm created in section 3.5.3 is used to obtain a score for Administrative Contribution through Issues.

Figure 3.46 represents this calculation procedure



Total Reg Init score	Total Res Init score	Category from B	Category from C	Issue-based admin calc
15	8	Medium	Medium	6
10	5.388562	Low	Low	2
13	4.667464	Low	Low	2
10	1.388562	Low	Low	2
7	8.388562	Low	Medium	4
19	4.480324	Medium	Low	4
30	11.48032	High	Medium	8
14	2.388562	Low	Low	2
36	20.17758	High	High	10
13	4.388562	Low	Low	2
20	3.388562	Medium	Low	4
16	9.667464	Medium	Medium	6
30	2.806914	High	Low	6

Figure 3.46: Calculation of Issue based Administrative Contribution through Inference Mechanism

For calculating Administrative Contribution through Pull Requests we calculate the total scores for the primary sub-components first. For the raw values obtained for Number of Pull Requests raised, Number of Pull Requests merged, and Average number of commits done per PR, the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Regular Initiatives in Pull Request Contribution*

For the raw values obtained for Number of Pull Requests raised for Bugs/Blockers, and Average number of commits done per PR for Bug/blocker issues, the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Troubleshooting Initiatives in Pull Request Contribution*

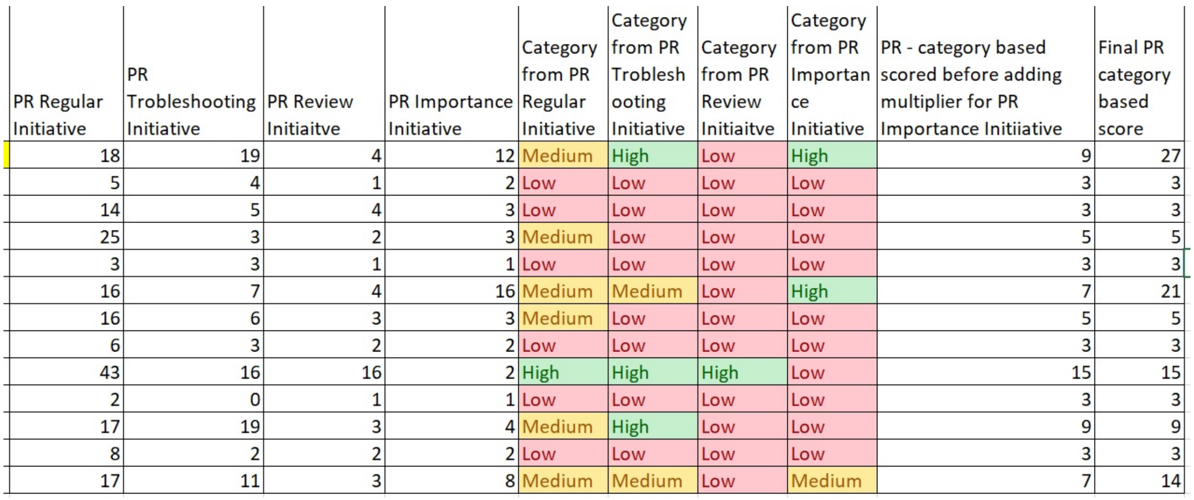
For the raw values obtained for Number of Pull Requests reviewed the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Review Initiatives in Pull Request Contribution*

For the raw values obtained for Number of times the Pull Requests was referenced the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for PR Importance*

Once these values are calculated, the process to estimate Pull Request Contribution score is as follows -

```
Divide the population under consider into thirds
if the total score for a component falls into the 1st third for the population then
    Allocate a label of Low and a graded value of 1 for that sub-component
else if the total score for a component falls into the 2nd third for the population then
    Allocate a label of Medium and a graded value of 3 for that sub-component
else if the total score for a component falls into the final third for the population then
    Allocate a label of High and a graded value of 5 for that sub-component
end if
Sum up the graded values for Total Score for Regular Initiatives in Pull Request Contribution, Total Score for Troubleshooting Initiatives in Pull Request Contribution, Total Score for Review Initiatives in Pull Request Contribution
Multiply to this core the graded value for PR Importance to obtain the final graded score for Pull Request Contributions
```

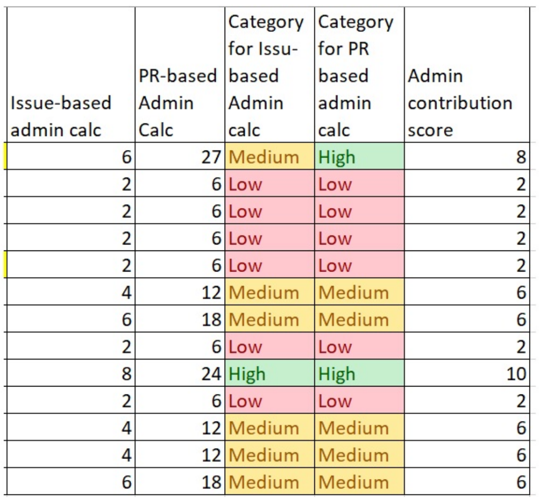
Figure 3.47 represents this calculation Finally when the Issue and PR Administrative



PR Regular Initiative	PR Trobleshooting Initiative	PR Review Initiaitve	PR Importance Initiative	Category from PR Regular Initiative	Category from PR Trobleshooting Initiative	Category from PR Review Initiaitve	Category from PR Importance Initiative	PR - category based scored before adding multiplier for PR Importance Initiiative	Final PR category based score
18	19	4	12	Medium	High	Low	High	9	27
5	4	1	2	Low	Low	Low	Low	3	3
14	5	4	3	Low	Low	Low	Low	3	3
25	3	2	3	Medium	Low	Low	Low	5	5
3	3	1	1	Low	Low	Low	Low	3	3
16	7	4	16	Medium	Medium	Low	High	7	21
16	6	3	3	Medium	Low	Low	Low	5	5
6	3	2	2	Low	Low	Low	Low	3	3
43	16	16	2	High	High	High	Low	15	15
2	0	1	1	Low	Low	Low	Low	3	3
17	19	3	4	Medium	High	Low	Low	9	9
8	2	2	2	Low	Low	Low	Low	3	3
17	11	3	8	Medium	Medium	Low	Medium	7	14

Figure 3.47: Calculating PR Contribution through Inference Mechanism

Calculations scores are obtained, we use the same algorithm defined in section 3.5.3 to calculate the Administrative Contribution Score. Figure 3.48 represents this calculation



Issue-based admin calc	PR-based Admin Calc	Category for Issu-based Admin calc	Category for PR based admin calc	Admin contribution score
6	27	Medium	High	8
2	6	Low	Low	2
2	6	Low	Low	2
2	6	Low	Low	2
2	6	Low	Low	2
4	12	Medium	Medium	6
6	18	Medium	Medium	6
2	6	Low	Low	2
8	24	High	High	10
2	6	Low	Low	2
4	12	Medium	Medium	6
4	12	Medium	Medium	6
6	18	Medium	Medium	6

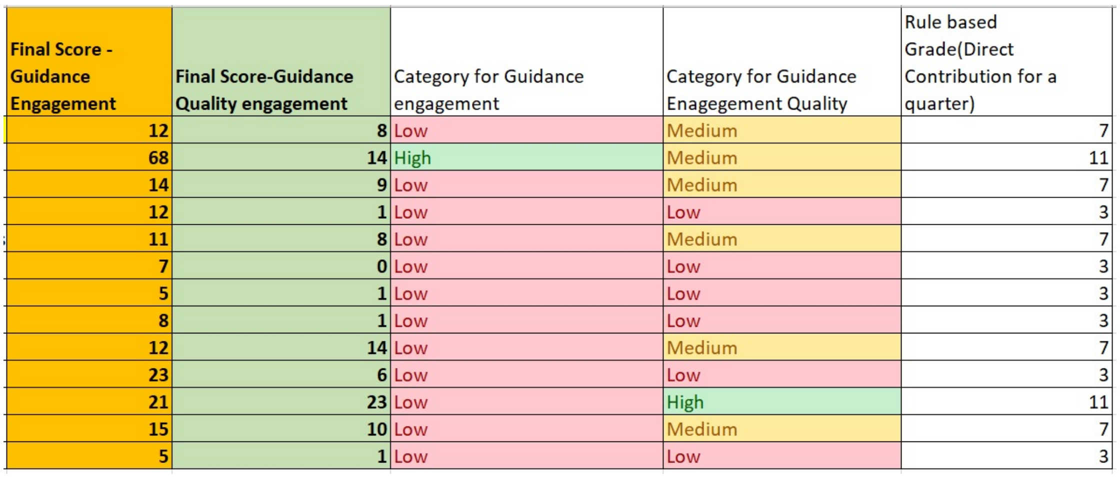
Figure 3.48: Calculation of Administrative Contribution through Inference Mechanism

3.5.4 Inference Mechanism for Guidance Contribution

For calculating Guidance Contribution we calculate the total scores for the primary sub-components first. For the raw values obtained for Number of Messages, Number of Mentions, Number of questions asked, Number of Questions answered, and Number of issues raised the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Guidance-engagement*

For the raw values obtained for Proportion of Positive posts, quality of answers, and Proportion of Questions answered the standard mathematical transformations for calculating Percentile score, Z-score and Total score defined in Section 3.5.1 are used. These values are then added to obtain the *Total Score for Guidance-Quality of Engagement*

Finally when the Guidance-engagement and Guidance-Quality of engagement Calculations scores are obtained, we use the same algorithm defined in section 3.5.3 to calculate the Guidance Contribution Score. Figure 3.49 represents this calculation



Final Score - Guidance Engagement	Final Score-Guidance Quality engagement	Category for Guidance engagement	Category for Guidance Enagegement Quality	Rule based Grade(Direct Contribution for a quarter)
12	8	Low	Medium	7
68	14	High	Medium	11
14	9	Low	Medium	7
12	1	Low	Low	3
11	8	Low	Medium	7
7	0	Low	Low	3
5	1	Low	Low	3
8	1	Low	Low	3
12	14	Low	Medium	7
23	6	Low	Low	3
21	23	Low	High	11
15	10	Low	Medium	7
5	1	Low	Low	3

Figure 3.49: calculation of Guidance Contribution through Inference Mechanism

3.6 Design Limitations

The primary limitation of the data-model created to measure contribution is the fact that the expert system knowledge base constructed is based on the opinions of only the author. However, the aim of this research was not to create a model which would be universally valid and cater to a universal definition of contribution, and future work considers exploring the possibilities of creating such a model.

A limitation of the model, created bottom-up, from metrics in GitHub is that it was not able to fully capture the entire range of heuristics proposed by the model to be used for manual validation. The implications of this are discussed in chapter 5

3.7 Summary

In this chapter, a conceptual design for contribution was created by using a top-down approach. A more detailed model was then designed from the concepts espoused by the first model which was capable of assessing the contribution of an engineer to a project by using manual verification methods. It was not considered at this stage as to whether this model would be suitable for the design of an automated system from GitHub. This was followed by a model, designed using a bottom-up approach, by using data mined from GitHub and Gitter. This model underwent refinement until it was able to reasonably represent the heuristics proposed in the manually verifiable model. For the ease of calculations a standardized set of mathematical formulae were designed which would represent the significance of the contribution of developers with respect to their peers. Finally, a knowledge base or set of rules were designed for the expert system which would calculate contribution. This expert system represented the codification of the concept of rewarding quality performance of a sufficient magnitude across the different contribution types. **Total contribution is calculated as the sum of the three contribution types**

Chapter 4

Implementation

In this chapter, the implementation details are talked about, and the technologies that used are described.

4.1 Programming Language Used

Python : Python[1] is a high-level, interpreted, general-purpose programming language. The python design philosophy is to implement a significant number of white spaces for code readability. *Python 3.7.4* is used in the project implementation. The programming language is based on Object-Oriented Programming Paradigm for writing clear, logical codes for small and large scale products. Python is used as the primary programming language because of the many Open-Source libraries available for Data Analysis implemented in Python.

Java : Python[2] is a general-purpose high level object-oriented programming language developed by Sun MicroSystems. Java is being used in the project for the development of web based dashboard using Java Servlet. The open source Java Development Kit, OpenJDK[3] is being used in the project to avoid licensing issues. Java is selected to be the language to develop the dashboard because of its platform-independent nature. There are a lot of Integrated Development Environments available for the development and easy availability of servers for the deployment of the application. The Apache

[1]`https://www.python.org/`,Last Accessed on 13 August 2019
[2]`http://oracle.com/java/`,Last Accessed on 13 August 2019
[3]`openjdk.java.net`,Last Accessed on 13 August 2019

Tomcat server is used in this project.

4.2 Using GitHub to measure contribution

Developers use version control systems to manage their projects. GitHub is a commonly used source control repository for open-source projects as well as private projects. This project analyses the scikit-learn repository from GitHub.

Tools used in the implementation are :

1. **MongoDB**[4] : It is a popular cross-platform NoSQL database, especially useful for creating a document-oriented database. since this project utilizes JSON files for analysis, MongoDB has been used.
2. **PyMongo**[5] : Is a python distribution which provides tools to interact with MongoDB from Python.
3. **PyLint**[6] : It is a static code analysis tool used for measuring code quality in Python.

The steps involved in analysis of GitHub data is :

1. **Data Gathering** : The data gathering involves the collection of data from the scikit-learn repository in GitHub for analysis.
2. **Analysis** : The collected data is analyzed and results are obtained from this module.

4.2.1 Data Gathering

The data gathering from GitHub involves Configuration management, and gathering Pull-request, Issue, and Commit data from the scikit-learn repository for analysis.

[4]`https://www.mongodb.com/`,Last Accessed on 13 August 2019
[5]`https://api.mongodb.com/python/current/`,Last Accessed on 13 August 2019
[6]`https://www.pylint.org/`,Last Accessed on 13 August 2019

Configuration Management

Configuration management refers to the management of the configuration file **config.py**. In the configuration file we have to provide the details of the git repository to be analyzed. A user token is generated and is provided in the configuration file to identify the user to obtain access to the information from the git repository. The information is collected only from the repository which is being configured in the configuration file. This configuration file also allows data mining to continue either from the last saved point, or begin anew.

Mining Commits

Whenever a user creates a new file or modifies an existing file it is committed to the repository. Information like the number of commits done by a user, the number of files generated per commit, and the number of new files created are all obtained from this module. Additionally, the quality of the commit is also analyzed by using Pylint. The **mineData/mineRepositoryCommit.py** file will fetch all the information about the commits that are made in the repository and find out the code quality using the PyLint library or API. This is saved to the MongoDB Database for future analysis.

Mining Issues

Executing the file **mineData/mineIssues.py** will fetch the information about the issues that are created in the configured repository and save it to the database. This module fetches data from the GitHub Issues API, but also the GitHub Issues/Timeline, Issues/Labels, and Issues/Milestones API's to gather the information required for analysis.

Mining Pull Requests

Once a feature is completed or an issue is resolved the code is required to be merged with the master branch of the repository. For this a pull-request is created. The details of the Pull Requests in the repository are by the **mineData/minePR.py** file. The collected information are then stored to the Database. This module requests information from the GitHub/Issues/timeline API as well as the Pull request API as GitHub considers all Pull Requests to be issues and saves their data accordingly.

4.2.2 Data Analysis

All the raw data gathered from the configured GitHub repository is stored in the database. This data is used to calculate the Administrative Contribution Score and Direct Contribution scores on a quarterly basis. This analysis is carried out by **misc/analyse_quarter_issues_pr_commits.py** and saves the final results to a .csv output file. this program contains the expert system for calculating Direct and Administrative Contribution codified as a set of If-Else rules.

4.3 Gitter Contribution Analysis

In this section we discuss the implementation details for the analysis of Gitter data. The Gitter data analysis module is implemented in Python using JupyterLab.

JupyterLab : JyputerLab is an IDE for Python from a Open-Source Project called **Project Jupyter**[7]. The JupyterLab IDE is often used by professional Data Analysts for analysis in Python. JupyterLab allows developers to run the code as small sequences leveraging the fact that Python is an interpreted programming language. This helps the user to analyze the output of individual code snippets before the completing the entire project or module.

The implementation includes

1. **Data Collection** : This stage involves gathering information from the Gitter chat groups which represent the GitHub repository we intend to analyze.
2. **Chat Classification** : This modules handles conversation delineation by segregating text into a set of questions, answer and acknowledgement texts. The body of data which are not captured by the heuristics for this approach are considered to be unclassified text.
3. **Sentiment Analysis** : Sentiment analysis is done on the conversations that are extracted from the Gitter data.

[7]`https://jupyter.org/`,Last accessed on 13 August 2019

4. **Creation of Secondary Metrics** : The primary metrics include the chat classification values and sentiment analysis values. Secondary metrics are created from the primary metrics for further evaluation and analysis in the model.

4.3.1 Data Collection

The Gitter chat data, required for analysis, is gathered using the Gitter-API [8]. The first step in the data collection process is to register the user and obtain a user-token from the developer dash board. The generated token is then added to the token file in the source code. The data is gathered from Gitter by executing the file **getdata.py**. This program will fetch the data from all chat-rooms of which the user a member. The gathered data is then saved to an archive folder as a json file with the chat-room name.

4.3.2 Chat Classification

The retrieved chat information requires to be classified as a question, an answer, or an acknowledgement for further analysis of the conversations taking place in the chat-room. For data classification a trained system or model is required and to create that model the **Natural Language Toolkit (NLTK)**[9] is used. The NLTK contains different corpora for training and testing purposes while implementing Natural Language Processing. The training and testing is done on the Net Promoter Score (NPS) chat corpus that contains real-time chat information. This is a standard chat data-set available online for Natural Language processing on real-time chat information. In **GitterAnalysis.ipynb** notebook file the function **dialogue_act_features(post)** is being used to tokenize the the chat data and train the model using a Naive Bayes Classifier on the tokenised NPS corpus implemented in NLTK module.

After training of the model using NPS dataset the gitter conversations are passed to the trained model which classifies the sentence according to its type. first as a question or an answer, and then into the nature of the question like 'clarification' or 'wh-question', or the nature of the answer like 'y-answer'.

[8]`https://developer.gitter.im/docs/welcome`,Last accessed on 13 August 2019
[9]`https://www.nltk.org/`,Last accessed on 13 August 2019

4.3.3 Sentiment Analysis

Apart from identifying the questions, answers and acknowledgements present in the chat, the sentiment of the conversation is to be identified as either positive or negative. A model requires to be trained to do the sentiment analysis. For this purpose, the Internation Movie DataBase (IMDB) review corpus [10] is used. The system is trained with the help of NLTK and Scikit-Learn[11] libraries in **Sentiment_Analysis.ipynb** notebook file. The trained model is saved as **pickle** files for future use.

The trained model is then accessed inside **GitterAnalyisis.ipynb** and the Gitter data is fed to the trained model to find the sentiment of the conversation components. The model also provides the confidence of the sentiment which is calculated by the model.

4.3.4 Creation of Secondary Metrics

The results obtained after classification and sentiment analysis of conversation components are appended to the data which gathered from the Gitter chat-room. The resulting data-frame contains the classification and sentiment as the primary metrics, and is used for the creation of secondary metrics. The secondary metrics involve things like the *total number of questions asked by the user, total number of answers given by the user and total number of positive and negative responses given by the user.* These secondary metrics are used for the evaluating Guidance Contribution. The creation of secondary metrics is done by the **calculations.ipynb** file.

4.4 Dashboard Creation

In this section we discuss the implementation of a dashboard in Java to represent the data analyzed by the modules described above.

The dashboard is created using Java Servelets and deployed using the Tomcat server. The functioning of this module can be described by the Sequence diagram in Figure 4.1 The values for Direct, Administrative and Guidance Contribution are

[10] `https://www.kaggle.com/iarunava/imdb-movie-reviews-dataset`,Last accessed on 13 August 2019

[11] `https://scikit-learn.org/stable/`,Last accessed on 13 August 2019

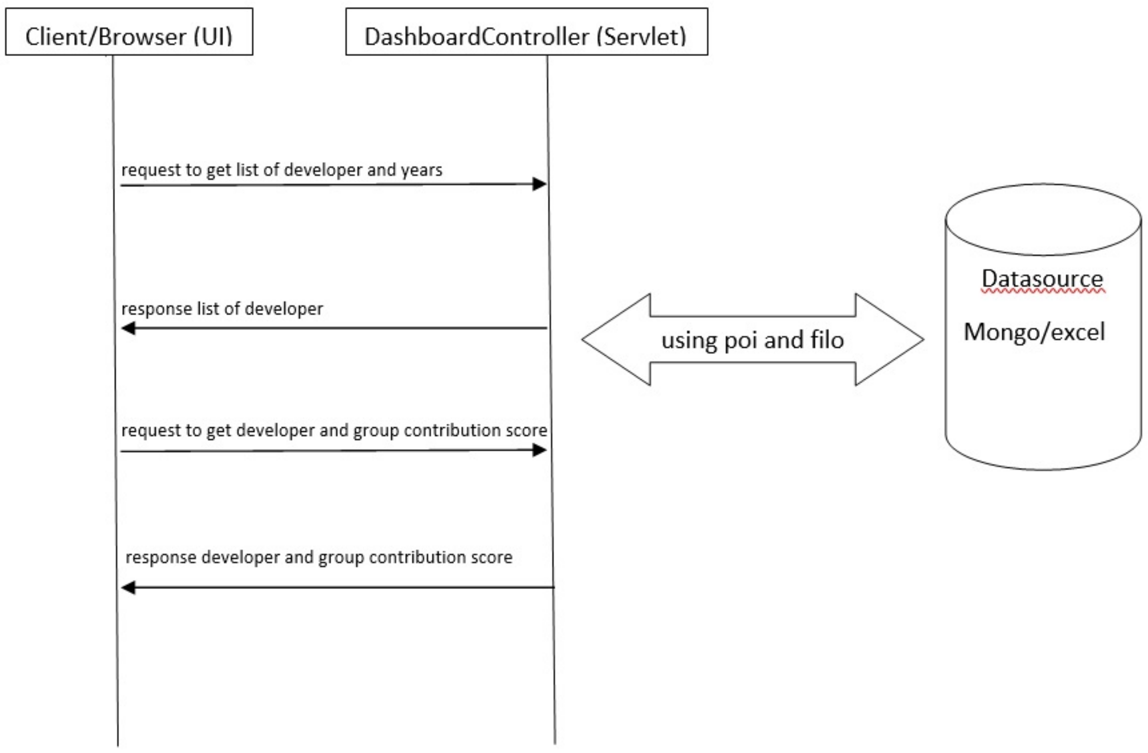

Client/Browser (UI)
DashboardController (Servlet)
request to get list of developer and years
response list of developer
request to get developer and group contribution score
response developer and group contribution score
using poi and filo
Datasource
Mongo/excel

Figure 4.1: Sequence diagram for dashboard

obtained as excel files and are manually merged to create a single document.

1. The browser/client will first fetch the developer's details and yearly contribution details from the excel file using Apache POI[12] and Fillo[13] (a java external api).

2. The End-Users will then select the developer of their choice and submit the form.

3. After the form submission, the request will go to the controller/servlet and servlet will fetch developer contribution score from the excel sheet using Apache POI. the Servlet with extract the data and send the contribution score to the JSP where highchart.js[14] will be used to create graph on dashboard

[12]`https://poi.apache.org/`,Last accessed on 13 August 2019
[13]`https://codoid.com/fillo/`,Last accessed on 13 August 2019
[14]`https://code.highcharts.com/highcharts.js/`,Last accessed on 13 August 2019

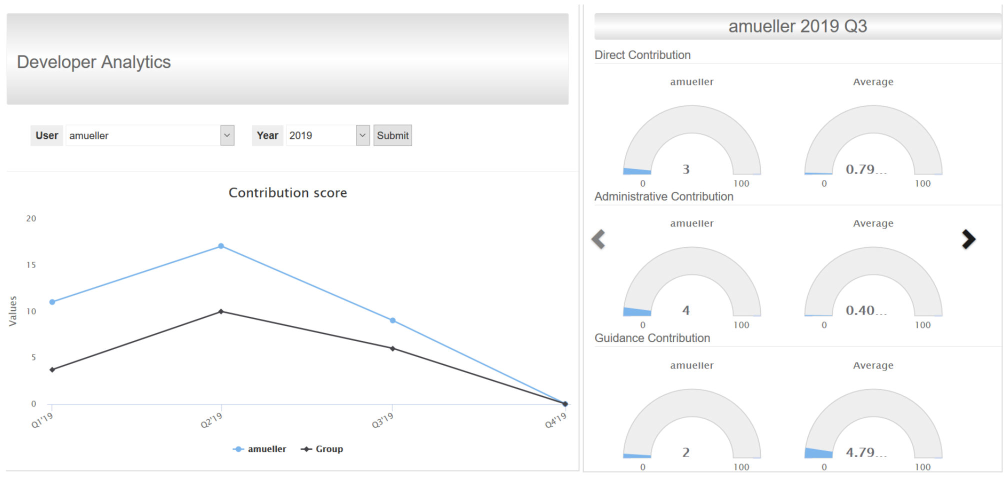


Figure 4.2: Dashboard Screen

Figure 4.2 shows the Dashboard screen. The main panel of the dashboard shows the contribution of the developer over a year, with data points showing the contribution score calculated for each quarter. The right hand side of the dashboard shows, by means of a sliding panel, the Direct Contribution, Administrative contribution, and Guidance Contribution scores for an engineer.

The drop-downs can be used to select the engineer whose contribution score is to be displayed, and the year for which the score is to be displayed.

4.5 Summary

Chapter 4 discussed the various technology decisions made in creating the system. The chapter discussed in detail the methods implemented to gather and analyze data from GitHub and Gitter and the way they are represented in the dashboard. A major limitation of the implementation, due to time constraints, is the inability to fully combine the front end of the system with the back-end, as the excel sheet which serves as input to the dashboard has to be created by combining the inputs of the two analysis modules, and then be ready for use by the dashboard.

Chapter 5

Results and Evaluation

This chapter sets out the results of the research performed and their evaluation against the standards set. An outline of the process of calculating contribution through manual means is first provided. Then the results obtained from the automated expert system are compared to the results from the manual model. The degree of correlation between the values obtained from the two methods are examined. Any differences observed in the trend and rank-ordering are also examined and discussed through pertinent examples. A detailed case study of the manual verification process is provided in the Appendix.

5.1 Scoring Process for Manual Evaluation of Direct Contribution

This section discusses the scoring process to be followed when calculating contribution through manual evaluation using the model defined in Section 3.3

5.1.1 Calculating Direct Contribution

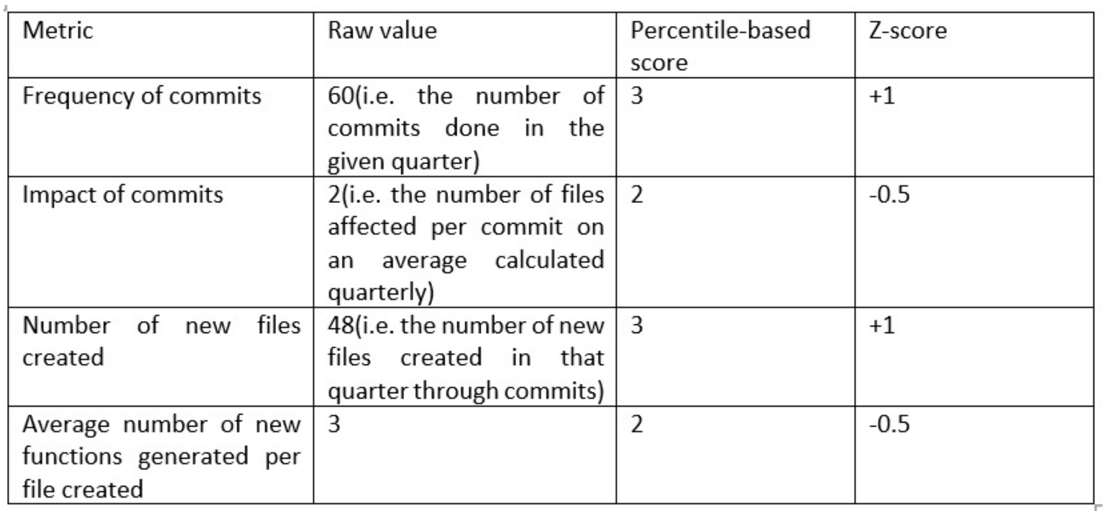
The scoring method for Direct Contribution emphasizes on estimating the software engineer's contribution to the code base of the project, both in terms of activity done and the quality of the contribution made.

5.1.2 Calculating Contribution to the code base

The *frequency of commits* is calculated as the number of commits done over a given period of time. For the purpose of this evaluation, the number of commits done over a quarter is calculated. This can be done by manually checking the number of commits from the GitHub UI or can be checked using the GitHub API. Since this is a raw count of an entity, the scoring process follows the algorithm defined in Section 3.5.3. The Total score for the metric is calculated through the Percentile Score and the Z-Score as defined in Section 3.5.3.

The Impact of the commits is also a raw value of the number of files affected per commit over a period of time. This is also handled in a similar fashion, with the Total score for the metric being calculated through the Percentile Score and the Z-Score as defined in Section 3.5.3. The *New functionality added to the database* consists of raw data scores for the number of new files created over a quarter and the number of new functions generated as a result of creating new files through commits. The Total score for each of the metrics is calculated through the Percentile Score and the Z-Score as defined in Section 3.5.3, and then their sum is considered to be the Total score for *New functionality added to the database*

. Therefore to calculate Contribution to the code base from these scores, we need



Metric	Raw value	Percentile-based score	Z-score
Frequency of commits	60(i.e. the number of commits done in the given quarter)	3	+1
Impact of commits	2(i.e. the number of files affected per commit on an average calculated quarterly)	2	-0.5
Number of new files created	48(i.e. the number of new files created in that quarter through commits)	3	+1
Average number of new functions generated per file created	3	2	-0.5

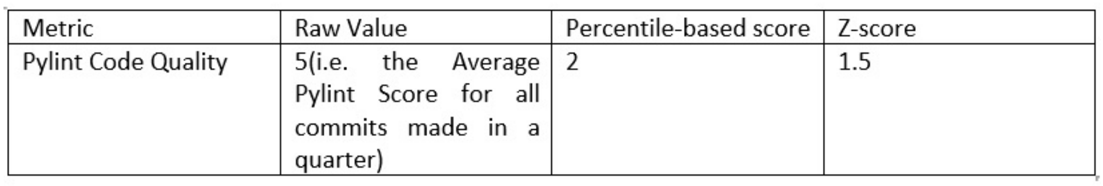
Figure 5.1: Sample scores for manual calculation of Contribution to code base

to add up the total scores for each of the components

From the example in Figure 5.1
$Contribution\ to\ the\ code\ base = \sum(Frequency\ of\ commit\ score, Impact\ of\ commits,$
$New\ functionality\ added\ to\ the\ codebase)$
$= \sum(3, 2, (3, 2))$
$= 10$

5.1.3 Calculating Contribution to code Quality

The *contribution to code quality* is calculated as the average Pylint score of commits done over a given period of time. The Total score for each of the metrics is calculated through the Percentile Score and the Z-Score as defined in section 3.5.3. From the



Metric	Raw Value	Percentile-based score	Z-score
Pylint Code Quality	5(i.e. the Average Pylint Score for all commits made in a quarter)	2	1.5

Figure 5.2: Sample scores for manual calculation of Contribution to code quality

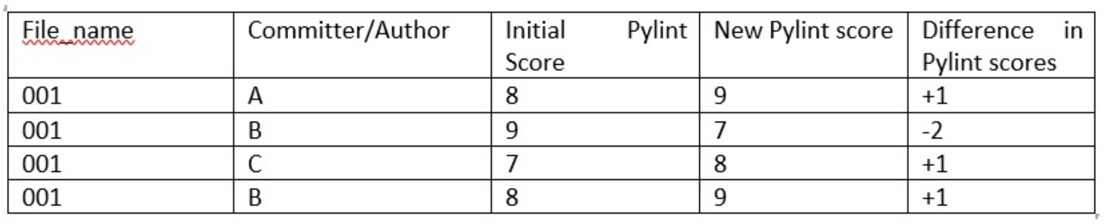
sample given in Figure 5.2
$Total\ score\ for\ Contribution\ to\ Code\ Quality\ is\ (2 * 1) = 2$

5.1.4 Calculating Contribution to Code Maintainability

Code Maintainability consists of two things, The change made to the quality of the code itself, and The quality of commenting done when editing a piece of code.

For calculating the score for the change made to the source code quality itself, we calculate the average change in code quality made to a set of files, by a developer, over a series of commits. This is calculated by assessing the difference in Pylint scores for a set of files as they undergo changes due to a series of commits. The difference between the Pylint score (indicative of code quality) of the file before the commit was made and the post-commit score is calculated and is assigned to the developer who made the commit.

For all developers under consideration a similar process is followed and the aggregation of Pylint score differences is calculated.



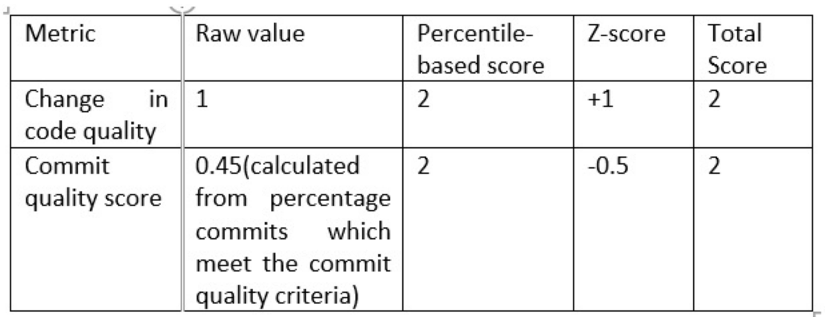
File name	Committer/Author	Initial Pylint Score	New Pylint score	Difference in Pylint scores
001	A	8	9	+1
001	B	9	7	-2
001	C	7	8	+1
001	B	8	9	+1

Figure 5.3: Sample data of change to code quality

For Developer A, on the basis of Figure 5.3, the total change made to the file code-quality score/number of changes made = (1/1) =1 Whereas, for Developer B, it will be (-2+1)/ (2) = (-1)/2= -0.5

For assessing commit quality

- For every comment found in the commit logs for the files affected by the commit, we check whether the commits are Coherent, Useful, Complete and Consistent on the basis of the quality model.
- If the comments are found to have the afore-mentioned attributes, then they are assigned a score of 1, other wise 0 is assigned.
- The percentage of comments which adhere to the quality standards is calculated.
- We consider the value of the percentage of comments adhering to quality standards made.
- We then calculate the total score on the basis of the algorithm in Section 3.5.3



Metric	Raw value	Percentile-based score	Z-score	Total Score
Change in code quality	1	2	+1	2
Commit quality score	0.45(calculated from percentage commits which meet the commit quality criteria)	2	-0.5	2

Figure 5.4: Sample Data for calculating code maintainability

Based on the sample data in Figure 5.4, Code Maintainability score can be calculated as the sum of the two-sub scores = 2+2 = 4 Direct contribution is then calculated as per the inference mechanism discussed in Section 3.5.2, in which we proceed to categorize these scores into Low, High and Medium Categories based on which third of the population an engineer falls into, for a particular score metric. Falling into a category of Low gives the engineer a score of 1, Medium gives 3 and High gives 5. These are then added to give the final Direct Contribution score for the engineer.

5.2 Scoring Process for Manual Evaluation of Administrative Contribution

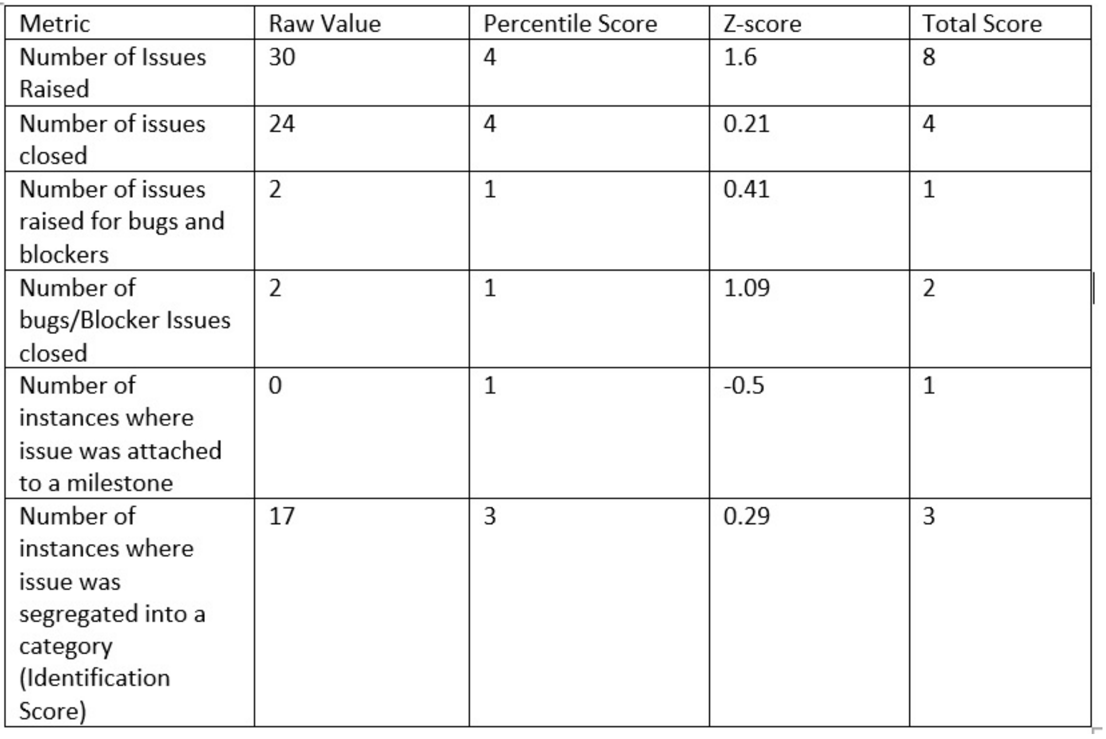
The scoring process for Manual Evaluation of Administrative Contribution constitutes the Manual scoring for the Issue-based and Pull-Request based Administrative contribution.

5.2.1 Scoring for Issue-based Administrative Contribution

This consists of calculating the scores for Number of Issues Raised, Number of Issues Closed, Number of issues raised for bugs and blockers, Number of bugs/Blocker Issues closed, Number of instances where issue was attached to a milestone and Number of instances where issue was segregated into a category (Identification Score).

All metrics excluding the identification score are directly calculated from raw counts of the activities. The identification score is determined using the formula given in Section 3.4.2, i.e. by the formula $\sum(Number\ of\ labels\ identified*Weight-age\ of$

label category) where labels of category A,B, and C as defined in the previous sections receive weights of 3,2,and 1 respectively. For the values so obtained, the total score is calculated using the algorithm given in section 3.5.3. The scores are then added up to give the *Regular Initiative Issue-based Administrative Contribution score* From Figure



Metric	Raw Value	Percentile Score	Z-score	Total Score
Number of Issues Raised	30	4	1.6	8
Number of issues closed	24	4	0.21	4
Number of issues raised for bugs and blockers	2	1	0.41	1
Number of bugs/Blocker Issues closed	2	1	1.09	2
Number of instances where issue was attached to a milestone	0	1	-0.5	1
Number of instances where issue was segregated into a category (Identification Score)	17	3	0.29	3

Figure 5.5: Sample data for Manual Calculation of Regular Initiative for Issue-based Administrative Contribution

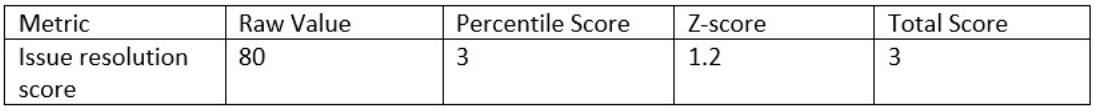
5.5 we can calculate the *Regular Initiative Issue-based Administrative Contribution score* = 8+4+1+2+1+3 = 19

For Issue Resolution Initiative Administrative Contribution Score we look at the contribution of each developer in resolving the issues, as discussed in Chapter 3, some of the major factors considered here are –

- Avoiding Issue Duplication
- Raising a single issue for every concern faced
- Proper titling of issues including indicating potential bugs in Issue title

- Properly formatting the issue-body for greater ease of understanding
- Following a standard template for raising issues and including Issue reproduction steps
- Making comprehensive issue resolution recommendations and providing relevant examples where necessary.

Indicative examples are provided in Section 3.3.2, and more details are available in the appendix. Every instance where an engineer is seen to adhere to these guidelines, a positive score of 1 is given and -1 is awarded where these guidelines are flaunted. The score so calculated is converted into a total score on the basis of the algorithm defined in section 3.5.3 As evidenced from the sample data in Figure 5.6, the *Issue Resolution*



Metric	Raw Value	Percentile Score	Z-score	Total Score
Issue resolution score	80	3	1.2	3

Figure 5.6: Sample Data for Manual Calculation of Issue Resolution Initiative Issue-based Administrative Contribution

Initiative Administrative Contribution Score, we get the value of 3. Once the *Issue Resolution Initiative score for Issue-based Administrative Contribution* and the *Regular Initiative score for Issue-based Administrative Contribution* has been computed we use the inference mechanism defined in Section 3.5.3 to calculate Issue based Administrative Contribution.

5.2.2 Scoring for PR-based Administrative Contribution

It is calculated by first evaluating the contribution from its primary components

1. Regular Initiative
2. Troubleshooting Initiative
3. Review Initiative

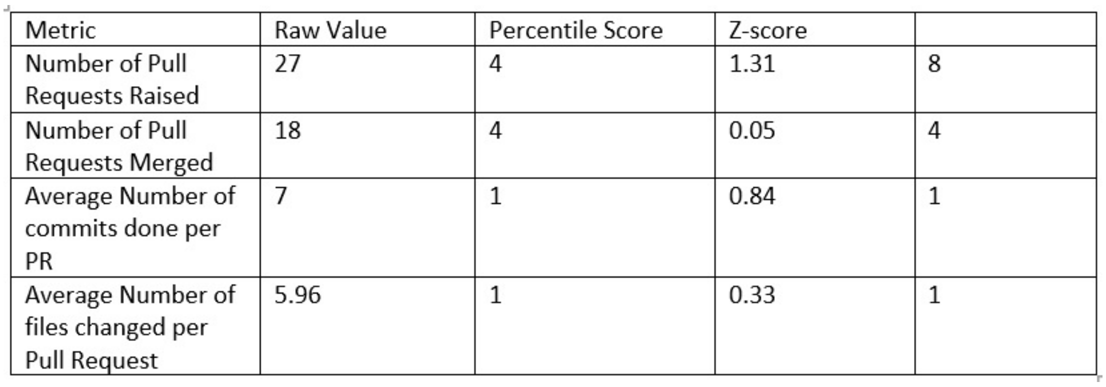
4. Pull Request Importance Score

Regular Initiative

This consists of calculating the scores for Number of Pull Requests Raised, Number of Pull Requests Merged, Average Number of files changed per Pull Request, Average number of commits done per Pull request.

The first two measures represent basic administrative tasks done by the engineer, and the latter two signify the effort made in raising the pull request. However, all these measures depend on raw data for these activities, and for the purpose of evaluation are calculated over the course of a quarter.

Once the raw measures are obtained their Total Scores are calculated on the basis of a Percentile Score and a Z-score as defined in Section 3.5.3. The sum of these total scores, so obtained are then considered to be the *Regular Initiative Score for PR-based Administrative Contribution.*

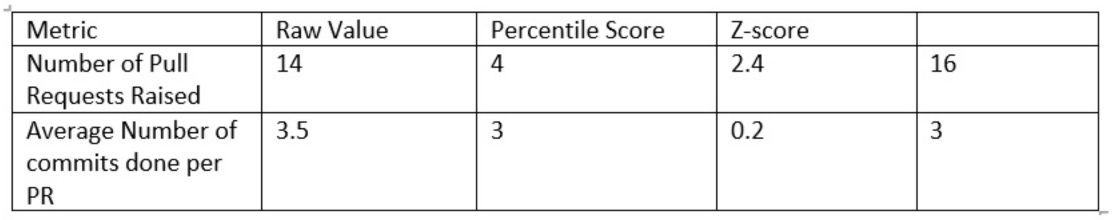


Metric	Raw Value	Percentile Score	Z-score	
Number of Pull Requests Raised	27	4	1.31	8
Number of Pull Requests Merged	18	4	0.05	4
Average Number of commits done per PR	7	1	0.84	1
Average Number of files changed per Pull Request	5.96	1	0.33	1

Figure 5.7: Sample Data for Manual Calculation of Regular Initiative Score PR-based Administrative Contribution

From Figure 5.7, we can calculate the *Regular Initiative Score PR-based Administrative Contribution* to be 8+4+1+1 = 14 The Troubleshooting Initiative score is calculated from the raw values for the number of Pull Requests Initiated for a Bug/Blocker and the Average number of commits done for a Pull Request raised for a Bug/Blocker issue. As with the Regular Initiative, the first measure represents basic administrative tasks done by the engineer, and the latter signifies the effort made in raising the pull request.

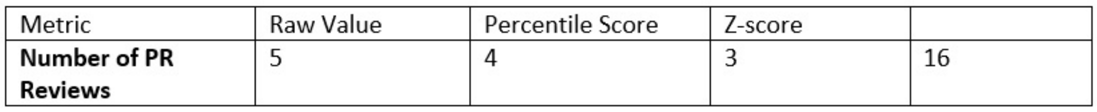
Once the raw measures are obtained their Total Scores are calculated on the basis of a Percentile Score and a Z-score as defined in Section 3.5.3. The sum of these total scores, so obtained are then considered to be the *Troubleshooting Initiative Score for PR-based Administrative Contribution.*



Metric	Raw Value	Percentile Score	Z-score	
Number of Pull Requests Raised	14	4	2.4	16
Average Number of commits done per PR	3.5	3	0.2	3

Figure 5.8: Sample Data for Manual Calculation of Troubleshooting Initiative Score for PR-based Administrative Contribution

From Figure 5.8, we can calculate the *Troubleshooting Initiative Score for PR-based Administrative Contribution* = 16+3 = 19 For calculating *Review Initiative Score for PR-Based Administrative Contribution*, we obtain the raw data value for the number of Pull Requests reviewed by an engineer over a period of time.



Metric	Raw Value	Percentile Score	Z-score	
Number of PR Reviews	5	4	3	16

Figure 5.9: Sample Data for Manual Calculation of Review Initiative Score for PR-based Administrative Contribution

From Figure 5.9, we can calculate the *Review Initiative Score for PR-based Review Contribution* = 16

To calculate the *Score for Quality of the Pull Request* we adapt a more nuanced strategy, in which the clarity with which the developer has raised the request is considered. Also considered is the willingness of other members of the community to accept the pull request into the code base. This point is evidenced by the number of corrections to the pull request sought and related clarifications sought.

Pull Requests which are raised clearly with links to existing issues (if relevant), and stating the purpose of the pull request are considered to be good examples. Pull requests which have been passed unanimously without any requirement of revisions are also considered ideal. These cases stipulate that a score of +1 be awarded to the concerned engineer for the contribution. However, Pull Requests which do not state its objectives, or the scope of which exceeds the objectives stated and are identified as such by the community are to be penalized with a score of -1.

In case a Pull Request needs a lot of revisions, the comments of the developers needs to be factored in.

Furthermore for every pull request cross-referenced a score of +1 is to be awarded to the engineer for raising a significant Pull Request, as cross-referencing is considered to be a sign of good PR-handling[1]. The final score obtained may be quite large and thus needs to be processed as per the algorithm in Section 3.5.3 to obtain a final score. Once the *Regular Initiative Score for PR-based Administrative Contribution*, *Troubleshooting Initiative Score for PR-based Administrative Contribution*, *Review Initiative Score for PR-based Review Contribution*, and *Score for Quality of the Pull Request*, we calculate the Pull Request-based administration score by the inference mechanism specified in Section 3.5.3.

5.2.3 Scoring for Guidance Contribution

Guidance contribution was measured in terms of the engagement of users in the chat board on Gitter and the quality of such engagement.

Engagement Score for Guidance Contribution

The contribution made by the amount of engagement was calculated by tracking -

1. Total Number of messages by a person in the chat board
2. Number of times the person the person was mentioned in a conversation.
3. Number of questions asked

[1]https://codeinthehole.com/tips/pull-requests-and-other-good-practices-for-teams-using-github/, accessed : 13.08.2019

4. Number of Questions answered
5. Number of issues raised in the board

The Total Score for these metrics are to be calculated using the algorithm in Section 3.5.3, and are summed up to give the Engagement score for Guidance Contribution.

Quality of Engagement Score for Guidance contribution

A score of +1 is to be given for positive contributions like asking non-trivial questions, replying promptly to questions, writing answers of high quality and technical soundness and keeping the community engaged in general.

For instance, engaging in detailed conversations may only be positive when there is a serious technical question being discussed. Otherwise, it may be considered negative if a person is wasting someone else's time by asking trivial questions

Putting a simple +1 or 'me too' in discussions on issues are not considered a positive contribution, even though they may help represent the magnitude of an issue.

Not answering questions even when mentioned or spamming are discouraged and penalized with a score of -1.

5.2.4 Scoring for Total Contribution

The total contribution is calculated as a sum of the scores obtained for Direct, Administrative and Guidance contribution.

5.3 Data Collection for Evaluation

The collection of data for evaluation was done by analyzing the activities of 13 engineers in the scikit-learn project for a quarter,viz. the first quarter of 2019, specifically the months of January, February and March. The names and other details of these engineers has been withheld to prevent bias and for confidentiality purposes.

Contribution was measured for these developers manually over that period of time by analyzing the activities done in the repository.

The contribution for these developers, and others in the project were also calculated from the automated expert system, designed and displayed on the dashboard.

Some instances of the manual contribution calculation process for engineers is outlined in the appendix.

5.4 Results and Evaluation

This section details the results for contribution scores calculated from the automated data-model and the manual evaluation process. The correlation between the two is observed and the relative pattern of the values obtained from both are compared. The rank-ordering of the developers from both the models are also compared.

A discussion of key observations from the results and inferences from and explanations of the same are then provided.

5.4.1 Results and Discussion

Table 5.1 shows the Total Contribution Scores calculated from the Data-Model as well as by Manual Evaluation. The correlation coefficient value between these two models is 0.77 which suggests a strong positive correlation between these two values.

Developer	Score from Model	Score from Manual Evaluation
A	27	28
B	19	26
C	19	22
D	15	22
E	15	26
F	11	22
G	11	20
H	11	12
I	11	20
J	11	22
K	7	12
L	7	12
M	7	18

Table 5.1: Comparison of Total Contribution Scores from model and manual evaluation

Figure 5.10 given below shows the graphical plot of these values. From both the graph and the table, it is clear that the Manual Verification process returns a con-

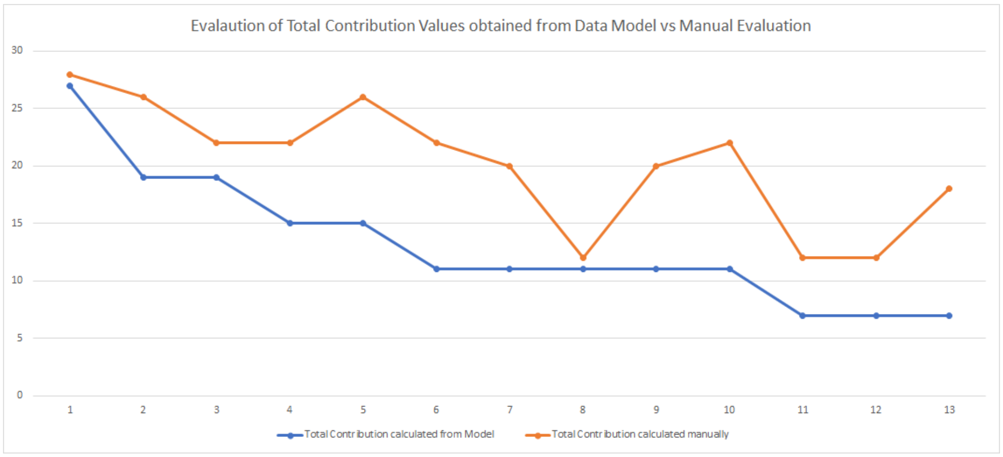


Figure 5.10: Line-graph for the total contribution scores calculated by the data-model and manual evaluation

sistently higher score for contribution than the Data-Model. There are no instances where the data-model has returned a higher value than the Manual Evaluation model, which leads us to infer that there is a definite loss of fidelity when contribution is calculated by the data model. Another interesting result is obtained when we check the rank-ordering for the engineers on the basis of the two models as described in Table 5.2 and Table 5.3

Rank	Developer	Total Contribution Score from Model
1	A	27
2	B	19
2	C	19
3	D	15
3	E	15
4	F	11
4	G	11
4	H	11
4	I	11
4	J	11
5	K	7
5	L	7
5	M	7

Table 5.2: Rank Ordering of Engineers by the Data-Model

Rank	Developer	Total Contribution Score from Manual
1	A	28
2	B	26
2	E	26
3	C	22
3	D	22
3	F	22
3	J	22
4	G	20
4	I	20
5	M	18
6	H	12
6	K	12
6	L	12

Table 5.3: Rank Ordering of Engineers by Manual Evaluation

It is evident from both the tables that Engineer A holds the top rank regardless of the evaluation method used.

Another pattern presents itself when we split the lists into 2 sections of top-6 and bottom-6 engineers, with the 7^{th} ranked person in either group being in the middle. Even though, the relative ranking may have changed slightly, Engineers A,B,C,D, and E hold the top-6 ranks in either evaluation model. The bottom six also remain relatively unchanged with the exception of Engineers J and G swapping places for the 7^{th} spot.

With respect to relative positions in the list, apart from Developer A at the top, Developer B has managed to retain the 2^{nd} spot in both rank-ordered lists

Developers D and F also remain 3^{rd} in their respective lists. Developers G and H also retain the 4^{th} spot in both listings. Finally Developers K and L remain at the respective bottom positions of 5^{th} and 6^{th} for both models.

Thus, we can draw the inference that the overall rank-ordering structure remains similar across both models, even with the loss of fidelity.

For a detailed analysis into the metrics responsible for this loss of fidelity, the Direct Contribution, Administrative Contribution, and Guidance Contribution scores obtained from both the metrics were plotted. Graph 5.11 and its associated table, Table 5.4 compare the results of Direct Contribution Scores obtained from both the models.

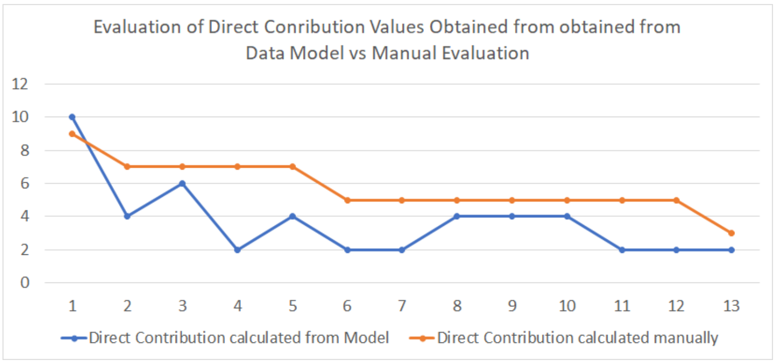


Figure 5.11: Line-graph for the Direct Contribution scores calculated by the data model and manual evaluation

It is observed that apart from a single data-point for Developer A, all the other data points show that the manual evaluation model has scored higher than the data-model.

Developer	Direct Cont. Score from Model	Direct Cont. Score from Manual
A	10	9
B	4	7
C	6	7
D	2	7
E	4	7
F	2	5
G	2	5
H	4	5
I	4	5
J	4	5
K	2	5
L	2	5
M	2	3

Table 5.4: Comparison of Direct Contribution Scores from model and manual evaluation

A deeper analysis of the components of Direct Contribution for the two models revealed that this is due to an imbalance in the scores generated by the inference mechanism for Code Maintainability.

While the code maintainability impact for all others is consistent across both models, it is found that A had got a better score in the data-model evaluation than the manual evaluation.

This was due to Developer a having missed out on some essential commenting while writing new functions and thus being penalized for it. However, as the functions were by themselves of good quality, the difference to code quality calculated through Pylint was not negative.

This will be illustrated in detail in the Appendix.

This, however, was a clear sign, that in situations where the approximations used (in this case using Pylint to check change in code quality does not work in cases where only comments have been changed) can prove to be fallible when exceptions arise. Next

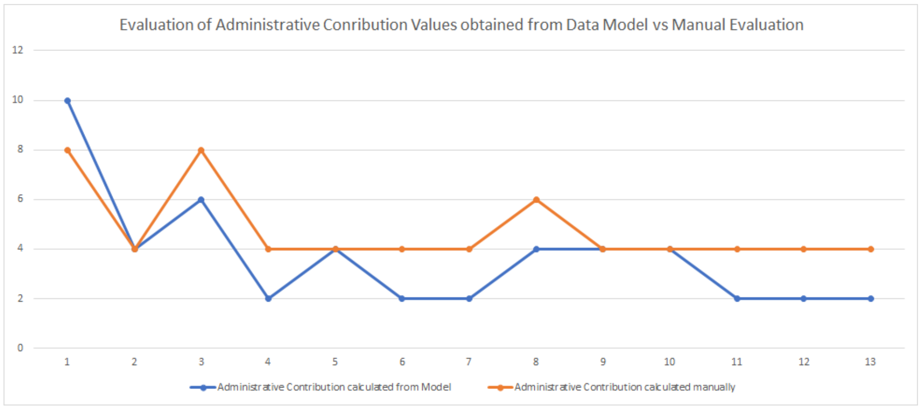


Figure 5.12: Line-graph for the Admin Contribution scores calculated by the data model and manual evaluation

the Administrative Contribution scores were analyzed for both the models. Which led to the observation that, though the manual evaluation model produced scores of greater magnitude than the Data-model, there were several instances of the scores matching. Also, the scores did not differ by more than two points during any time.

Since, the data-model had used some approximations for Issue-Resolution Initiative and Pull-Request Importance factor based on observations, it proved that those approximations did give a reasonable approximation of the quality of work done.

Cases where an important issue comment surfaced, but was not considered by the data-model due to no event following it in the timeline, led to the loss of fidelity.

Developer	Admin Cont. Score from Model	Admin Cont. Score from Manual
A	10	8
B	4	4
C	6	8
D	2	4
E	4	4
F	2	4
G	2	4
H	4	6
I	4	4
J	4	4
K	2	4
L	2	4
M	2	4

Table 5.5: Comparison of Admin Contribution Scores from model and manual evaluation

Finally the Guidance contribution scores for the two models are analyzed. The results for Guidance contribution score for both the models is analyzed and it is observed that the contribution scores from the manual evaluation are in general of a higher magnitude than the ones from the data-model. Though in some cases, the scores were the same.

Analysis laid out the fact that the cause for this was some conversations getting missed from the scope of things as they were not of a question-answer form or were too apart in time for the system to relate.

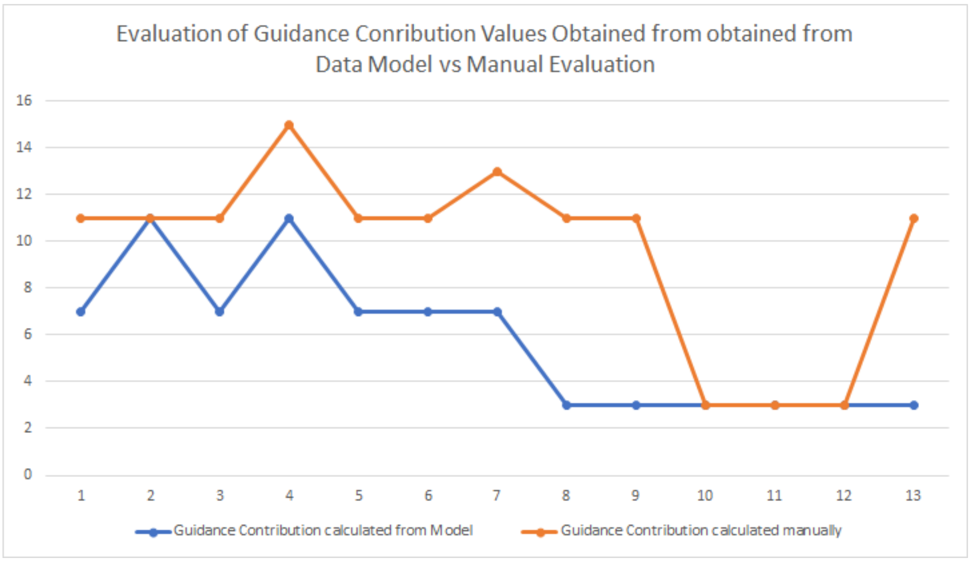


Figure 5.13: Line-graph for the Guidance Contribution scores calculated by the data model and manual evaluation

5.4.2 Summary

A comparative analysis of the contribution scores for the data model and the manual evaluation model revealed that the data model suffered a loss of fidelity because the approximations used failed in cases where the heuristics for those approximations met exceptions.

In general, there was no scenario in which the data model outscored the manual evaluation model for calculating total contribution, though an anomaly was recorded in Direct Contribution.

However, the data-model, built using signals from GitHub was able to provide a reasonably accurate representation of the high-level model(evaluated manually) by giving a consistent rank-ordering of the engineers for total contribution. The same engineer (Developer A) was first in both rank-ordered lists, with other engineers also holding the same position in both the lists. The upper half of both lists featured the same engineers (although in a slightly different order), and the lower half was similarly arranged with only one exception.

Another notable outcome of the model is represented in Figure . One of the main objectives for building this data model was to ensure that the entire range of activities

Developer	Guid. Contri. Score from Model	Guid. Contri. Score from Manual
A	7	11
B	11	11
C	7	11
D	11	15
E	7	11
F	7	11
G	7	13
H	3	11
I	3	11
J	3	3
K	3	3
L	3	3
M	3	11

Table 5.6: Comparison of Guidance Contribution Scores from model and manual evaluation

performed by an engineer would be considered for calculating contribution, rather than simply code-based metrics. This is aptly demonstrated as the *No.* 2 ranked engineer in GitHub's list of contributors for the scikit-learn project, shows a similar contribution level to the person placed at *No.* 24, thereby showing the inherent weakness of relying solely on code-based metrics and the success of the research objective of the project *"To create a data-model for measuring the contribution of a Software engineer in a project by analyzing the entire range of developer activity over a period of time, considering both the amount of work performed and the quality of the work, as opposed to traditional LOC based measures."*

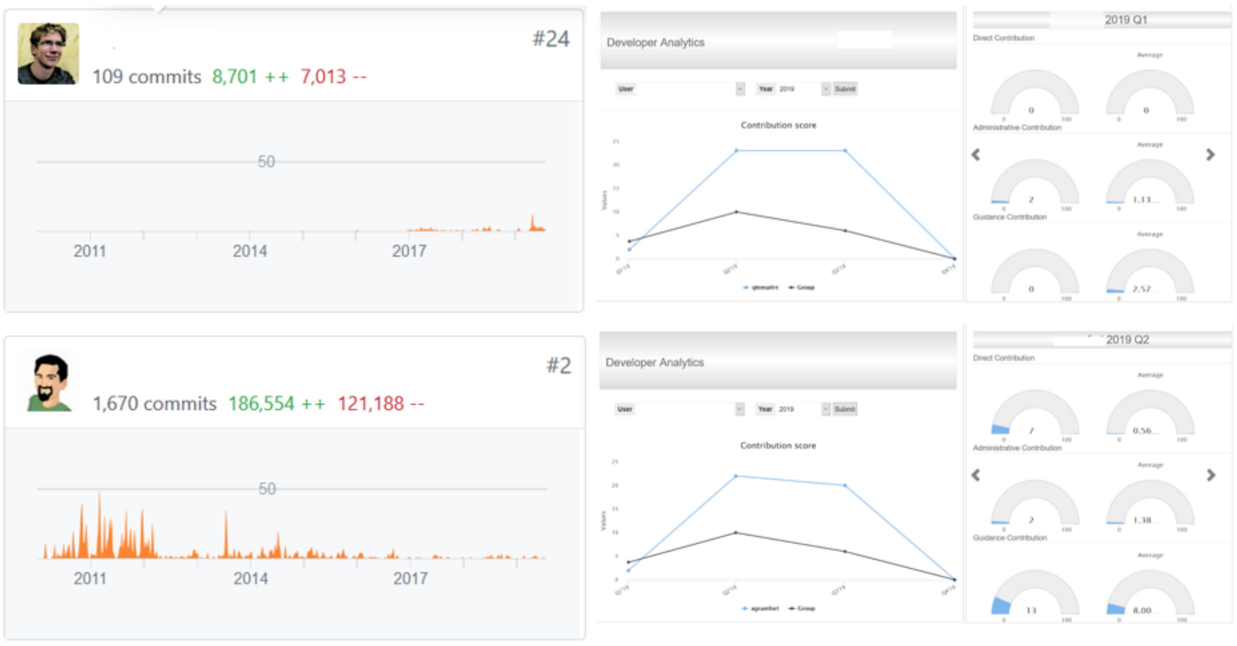


Figure 5.14: Comparable Contribution scores for engineers ranked differently by GitHub

Chapter 6

Conclusion

Chapter Six concludes the dissertation. Its goal is to show that the research objectives depicted in Chapter One have been met, to list the limitations of the current work, to state the research contribution this dissertation represents, and to discuss the future work prospects.

6.1 Objective Assessment

6.1.1 Creation of Data Model to Measure Contribution

Research Objective *To create a data-model for measuring the contribution of a Software engineer in a project by analyzing the entire range of developer activity over a period of time, considering both the amount of work performed and the quality of the work, as opposed to traditional LOC based measures.* The primary objective of this dissertation was the creation of a data-model to measure the contribution of a software engineer to a project by analyzing the various activities carried out by the developer in the project.The first phase of this work consisted of creating a high-level, top-down, mental model for contribution and then fleshing it out by expanding upon the concepts to create a detailed model that supported the calculation of contribution through manual means. This model did not consider whether the heuristics used to calculate contribution could be automated.

Then in the next phase, signals were mined from both GitHub and Gitter to create a model that would reasonably quantify the concepts espoused in the detailed top-down model. Using a bottom-up approach, the base metrics/signals from GitHub and Gitter were used to create a model was created which reasonably addressed all the heuristics for calculating contribution as expressed in the first model. On analysis, this model gave similar relative positions when rank-ordering the selected group of developers, as given by the top-down (or mental) model for contribution.

6.1.2 Mining GitHub and Gitter for useful signals

Research Objective *To identify useful signals from GitHub that can serve to approximate/quantify the high-level concepts espoused in the data model.*

Another objective of this research was to identify signals available from GitHub and Gitter which would be useful in measuring the contribution of a software developer in a project.

For this purpose, GitHub and Gitter were mined for a wide range of metrics which would help to analyze the activities done by developers in a project, and thus help quantify contribution. The metrics used for this model were later fed to an expert system to measure contribution.

6.1.3 Creation of an Expert System to measure contribution

Research Objective *To create a rule-based tool (Expert System) that would quantify the contribution of a software developer using the metrics derived from the selected GitHub and Gitter signals.*

To address the research question, an expert system was constructed which would use a set of rules as the Knowledge Base to measure contribution from a set of signals obtained from GitHub and Gitter.

6.1.4 Evaluation of the Data-Model

Research Objective *To evaluate the data-model through a sanity test of the contribution values generated by the tool with the values estimated by a human expert*

In pursuing the research goals, a sanity test was conducted to evaluate the performance of the contribution scores generated for a group of engineers in the scikit-learn project against the values obtained through a manual evaluation of the heuristics used in the top-down model. Both models gave nearly consistent relative positions to the developers when rank-ordering the assessed individuals.

6.1.5 Dashboard Creation

Research Objective *To visually assess, by means of a dashboard, the relative contributions of the software engineers to the project.*

A dashboard was created to display the contribution score of an engineer across the four quarters of a fiscal year.

6.1.6 Final Remarks

The significant result of this research has been to demonstrate that a data-model can be constructed using signals from GitHub which is consistent with a mental model of contribution created by an expert.

This clearly demonstrates the potential of using real-time data mined from software repositories to help objectively quantify concepts like contribution and maybe even performance which were considered to be purely subjective.

Though the research is not proven to be generally valid across repositories and engineer expertise and is refined to imitate the mental model of a particular expert, it nevertheless paves the way for future research to consider whether building a generalized model would be possible given the highly-contextual nature of what is understood to be "contribution".

Also, given the highly subjective and bias-prone methods followed in industry to judge contribution, this project provides a framework to quantify contribution in terms of activity performed and create a repeatable and verifiable process which can be used in the industry for performance appraisals.

6.2 Limitations

The data-model was able to accurately replicate all concepts dealing with an estimate of the quantity of work done, whereas some approximations had to be used while quantifying the quality of work done. An example is the concept of contribution to issue-resolution which was done by using a set of observed, measurable heuristics like Number of Likes obtained for comments, and categorizing those comments succeed by an event like cross-referencing or labelling to be important to issue resolution. While these served as good approximations, there were also several instances where manual evaluation would discover other engineers contribution to issue resolution, whose comments could not be captured by the scope of these heuristics.

With respect to analyzing conversations in Gitter chat, the heuristics used were capable of identifying conversations which took place within a reasonable span of time. Instances of conversations lingering on in the chat board over a period of weeks were not captured by the heuristics employed by the system.

Furthermore, the data-model was consistent in measuring contribution with respect to a specific view of contribution, and a specific project repository. Further work is needed to analyze whether the model would be consistent in measuring contribution globally, across projects as well as expert views of other engineers.

. The front-end also needs more work as it is currently able to display data from an excel or .csv file produced by the back-end system which measures contribution. Better integration of front-end and back-end systems needs to be done in order to build a more robust product.

6.3 Research Contribution

This section discusses the research contribution. A data-model was constructed which employed an expert system to measure the contribution of a developer in a project by considering a wide range of activities and not just code-based metrics like traditional approaches. This data model's results were found to be consistent with respect to the results obtained by manual evaluation of contribution of the author's own model of contribution.

The data-model produced as a result of this research considers a more comprehensive range of activities performed by engineers than existing State-of-the-Art systems as shown in Chapter 2.

The tracking of developer activity in an open-source chat board like Gitter to measure Guidance contribution is a novel approach which seeks to observe the engagement of engineers with the community at large. This is especially relevant for open-source projects which use these forums as sounding boards to judge community satisfaction with their work. These platforms are also important for improving the quality of project deliverables as many issues with the project are raised on these platforms, allowing project owners to identify and correct them.

6.4 Future Work

This research represents the first step in creating a universal model for measuring contribution that would be generalized across project repositories as well as expert opinion. Future work would focus on exploring the possibilities of creating a more comprehensive high-level model that would be generally acceptable to engineers across cultures and organizations.

Given the highly contextual nature of contribution, it would be interesting to see whether the definition of what qualifies as a "useful contribution" varies across cultures and organizations or if there is a middle ground where an understanding can be reached as to what is contribution, and how it is to be evaluated.

It would also be interesting to use advanced text analysis methods to design an automated process of evaluating the quality of code comments, issue and pull request comments and user chats on Gitter to make the data model more comprehensive.

Bibliography

Akiyama, F. (1971), An example of software system debugging., *in* 'IFIP Congress (1)', Vol. 71, pp. 353–359.

Albrecht, A. J. (1979), Measuring application development productivity, *in* 'Proc. Joint Share, Guide, and IBM Application Development Symposium, 1979'.

Alexandre, S. (2002), 'Software metrics: An overview (version 1.0)', *CETIC asbl—University of Namur, Software Quality Lab, Belgium* .

Amor, J. J., Robles, G. & Gonzalez-Barahona, J. M. (2006), Effort estimation by characterizing developer activity, *in* 'Proceedings of the 2006 international workshop on Economics driven software engineering research', ACM, pp. 3–6.

Anderson, T., Schum, D. & Twining, W. (2005), *Analysis of evidence*, Cambridge University Press.

Bache, R. & Neil, M. (1995), 'Introducing metrics into industry: a perspective on gqm', *Software Quality, Assurance and Measurement: A Worldwide Perspective* .

Basili, V. R. & Reiter, R. W. (1981), 'A controlled experiment quantitatively comparing software development approaches', *IEEE Transactions on Software Engineering* (3), 299–320.

Basili, V. R. & Rombach, H. D. (1988), 'The tame project: Towards improvement-oriented software environments', *IEEE Transactions on software engineering* **14**(6), 758–773.

Begel, A., Bosch, J. & Storey, M.-A. (2013), 'Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder', *IEEE Software* **30**(1), 52–66.

Bissyandé, T. F., Lo, D., Jiang, L., Réveillere, L., Klein, J. & Le Traon, Y. (2013), Got issues? who cares about it? a large scale investigation of issue trackers from github, *in* '2013 IEEE 24th international symposium on software reliability engineering (ISSRE)', IEEE, pp. 188–197.

Boehm, B. W. (1981), 'An experiment in small-scale application software engineering', *IEEE Transactions on Software Engineering* (5), 482–493.

Brereton, P. (2010), 'A study of computing undergraduates undertaking a systematic literature review', *IEEE Transactions on Education* **54**(4), 558–563.

Card, D. & Scalzo, B. (1999), Measurement for object-orienated software projects, *in* 'Proceedings of the 6th International Symposium on Software Metrics, Florida'.

Challa, J. S., Paul, A., Dada, Y., Nerella, V., Srivastava, P. R. & Singh, A. P. (2011), 'Integrated software quality evaluation: a fuzzy multi-criteria approach', *Journal of Information Processing Systems* **7**(3), 473–518.

Conte, S. D., Dunsmore, H. E. & Shen, Y. (1986), *Software engineering metrics and models*, Benjamin-Cummings Publishing Co., Inc.

Cuadrado, J. J., Amescua, A., Garcıa, L., Marbán, O. & Sánchez, M. I. (2002), Revision and classification of current software cost estimation models, *in* '6th World multiconference on systemics, cybernetics and informatics. Orlando, FL', pp. 339–341.

Dabbish, L., Stuart, C., Tsay, J. & Herbsleb, J. (2012), Social coding in github: transparency and collaboration in an open software repository, *in* 'Proceedings of the ACM 2012 conference on computer supported cooperative work', ACM, pp. 1277–1286.

Devanbu, P., Zimmermann, T. & Bird, C. (2016), Belief & evidence in empirical software engineering, *in* '2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)', IEEE, pp. 108–119.

Dhawan, S. & Juneja, N. (2013), 'Relevance of software metrics for a software project', *International Journal of Advanced Research in Engineering and Applied Sciences* **2**(10), 77–87.

Fagan, M. E. (1999), 'Design and code inspections to reduce errors in program development', *IBM Systems Journal* **38**(2.3), 258–287.

Fenton, N. & Bieman, J. (2014), *Software metrics: a rigorous and practical approach*, CRC press.

Fenton, N. E. & Neil, M. (1999), 'Software metrics: successes, failures and new directions', *Journal of Systems and Software* **47**(2-3), 149–157.

Flippo, E. B. (1976), *Principles of personnel management*, McGraw-Hill.

Gilb, T. (1977), *Software metrics*, Winthrop Publishers.

Glass, R. L. (1994), 'A tabulation of topics where software practice leads software theory', *Journal of Systems and Software* **25**(3), 219–222.

Glass, R. L., Vessey, I. & Conger, S. A. (1992), 'Software tasks: Intellectual or clerical?', *Information & Management* **23**(4), 183–191.

Gousios, G., Kalliamvakou, E. & Spinellis, D. (2008), Measuring developer contribution from software repository data, *in* 'Proceedings of the 2008 international working conference on Mining software repositories', ACM, pp. 129–132.

Gousios, G., Pinzger, M. & Deursen, A. v. (2014), An exploratory study of the pull-based software development model, *in* 'Proceedings of the 36th International Conference on Software Engineering', ACM, pp. 345–355.

Grady, R. B. & Caswell, D. L. (1987), *Software metrics: establishing a company-wide program*, Prentice-Hall, Inc.

Guaman, D., Sarmiento, P., Barba-Guamán, L., Cabrera, P. & Enciso, L. (2017), Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis, *in* '7th International Workshop on Computer Science and Engineering, WCSE', pp. 171–175.

Gupta, V. & Kumar, S. (2013), 'Impact of performance appraisal justice on employee engagement: a study of indian professionals', *Employee relations* .

Hall, T. & Fenton, N. (1997), 'Implementing effective software metrics programs', *IEEE software* **14**(2), 55–65.

Halstead, M. H. et al. (1977), *Elements of software science*, Vol. 7, Elsevier New York.

Hannay, J. & Jørgensen, M. (2008), 'The role of deliberate artificial design elements in software engineering experiments', *IEEE Transactions on Software Engineering* **34**(2), 242–259.

Hauff, C. & Gousios, G. (2015), Matching github developer profiles to job advertisements, *in* 'Proceedings of the 12th Working Conference on Mining Software Repositories', IEEE Press, pp. 362–366.

Herraiz, I., Robles, G., Amor, J. J., Romera, T. & González Barahona, J. M. (2006), The processes of joining in global distributed software projects, *in* 'Proceedings of the 2006 international workshop on Global software development for the practitioner', ACM, pp. 27–33.

Hetzel, B. (1993), *Making software measurement work: Building an effective measurement program*, John Wiley & Sons, Inc.

Islam, R. & bin Mohd Rasad, S. (2006), 'Employee performance evaluation by the ahp: A case study', *Asia Pacific Management Review* **11**(3), 163–176.

Johnson, P., Ekstedt, M. & Jacobson, I. (2012), 'Where's the theory for software engineering?', *IEEE software* **29**(5), 96–96.

Jorgensen, M., Dyba, T. & Kitchenham, B. (2005), Teaching evidence-based software engineering to university students, *in* '11th IEEE International Software Metrics Symposium (METRICS'05)', IEEE, pp. 8–pp.

Jørgensen, M. & Sjøberg, D. (2004), 'Generalization and theory-building in software engineering research', *Empirical Assessment in Software Eng. Proc* pp. 29–36.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M. & Damian, D. (2016), ‘An in-depth study of the promises and perils of mining github’, *Empirical Software Engineering* **21**(5), 2035–2071.

Kochhar, P. S., Bissyandé, T. F., Lo, D. & Jiang, L. (2013), Adoption of software testing in open source projects–a preliminary study on 50,000 projects, *in* ‘2013 17th European Conference on Software Maintenance and Reengineering’, IEEE, pp. 353–356.

Li, P. L., Ko, A. J. & Zhu, J. (2015), What makes a great software engineer?, *in* ‘Proceedings of the 37th International Conference on Software Engineering-Volume 1’, IEEE Press, pp. 700–710.

Marlow, J., Dabbish, L. & Herbsleb, J. (2013), Impression formation in online peer production: activity traces and personal profiles in github, *in* ‘Proceedings of the 2013 conference on Computer supported cooperative work’, ACM, pp. 117–128.

Matellán Olivera, V. et al. (2003), ‘Studying the evolution of libre software projects using publicly available data’.

McCabe, T. J. (1976), ‘A complexity metric’, *IEEE transactions on software engineering* **2**(4), 308–320.

Pfleeger, S. L., Jeffery, R., Curtis, B. & Kitchenham, B. (1997), ‘Status report on software measurement’, *IEEE software* **14**(2), 33–43.

Pham, R., Singer, L., Liskin, O., Figueira Filho, F. & Schneider, K. (2013), Creating a shared understanding of testing culture on a social coding site, *in* ‘Proceedings of the 2013 International Conference on Software Engineering’, IEEE Press, pp. 112–121.

Putnam, L. H. (1978), ‘A general empirical solution to the macro software sizing and estimating problem’, *IEEE transactions on Software Engineering* (4), 345–361.

Rainer, A. (2017), ‘Using argumentation theory to analyse software practitioners’ defeasible evidence, inference and belief’, *Information and Software Technology* **87**, 62–80.

Rainer, A., Hall, T. & Baddoo, N. (2003), Persuading developers to” buy into” software process improvement: a local opinion and empirical evidence, *in* ‘2003 International

Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.', IEEE, pp. 326–335.

Reddy, S. R. (2018), Sociometrics in Software Engineering Measuring Dedication of Software Developer through GitHub, Thesis.

Rezvina, S. (2019), 'Velocity vs. gitprime vs. diy how to choose an engineering intelligence tool'. Accessed: 2019-07-15.
URL: *https://codeclimate.com/blog/velocity-vs-gitprime/*

Rigby, P. C. (2012), 'Open source peer review–lessons and recommendations for closed source'.

Riguzzi, F. (1996), 'A survey of software metrics', *Universita degli Studi di Bologna* .

Robles, G., González-Barahona, J. M. & Ghosh, R. A. (2004), Glutheos: Automating the retrieval and analysis of data from publicly available software repositories., *in* 'MSR', Vol. 4, IET, pp. 28–31.

Robles, G., Gonzalez-Barahona, J. M. & Herraiz, I. (2005), An empirical approach to software archaeology, *in* 'Proc. of 21st Int. Conf. on Software Maintenance (ICSM 2005), Budapest, Hungary', pp. 47–50.

Robles, G., Koch, S., GonZÁlEZ-BARAHonA, J. M. & Carlos, J. (2004), Remote analysis and measurement of libre software systems by means of the cvsanaly tool, *in* 'Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)', IET, pp. 51–56.

Sackman, H., Erikson, W. J. & Grant, E. E. (1966), Exploratory experimental studies comparing online and offline programing performance, Technical report, SYSTEM DEVELOPMENT CORP SANTA MONICA CA.

Sadowski, C., Söderberg, E., Church, L., Sipko, M. & Bacchelli, A. (2018), Modern code review: a case study at google, *in* 'Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice', ACM, pp. 181–190.

Sanyal, M. & Biswas, S. (2014), ‘Employee motivation from performance appraisal implications: Test of a theory in the software industry in west bengal (india)’, *Procedia Economics and Finance* **11**, 182–196.

Schum, D. A. (2001), *The evidential foundations of probabilistic reasoning*, Northwestern University Press.

Shull, F. & Seaman, C. (2008), ‘Inspecting the history of inspections: An example of evidence-based technology diffusion’, *IEEE software* **25**(1), 88–90.

Sjøberg, D. I., Dybå, T., Anda, B. C. & Hannay, J. E. (2008), Building theories in software engineering, *in* ‘Guide to advanced empirical software engineering’, Springer, pp. 312–336.

Steidl, D., Hummel, B. & Juergens, E. (2013), Quality analysis of source code comments, *in* ‘2013 21st International Conference on Program Comprehension (ICPC)’, Ieee, pp. 83–92.

Symons, C. R. (1991), *Software sizing and estimating: Mk II FPA (function point analysis)*, John Wiley & Sons, Inc.

Thoti, K. K., BE, M. & Nagar, S. S. (2015), ‘Performance appraisal significance in the software industries in india’.

Toulmin, S. E. (2003), *The uses of argument*, Cambridge university press.

Tsay, J. T., Dabbish, L. & Herbsleb, J. (2012), Social media and success in open source projects, *in* ‘Proceedings of the ACM 2012 conference on computer supported cooperative work companion’, ACM, pp. 223–226.

Twining, W. (1994), *Rethinking evidence: Exploratory essays*, Northwestern University Press.

Udara, M. (2015), ‘An introduction to sonarqube’. Accessed: 2019-07-10.
URL: *https://milinaudara.wordpress.com/2015/01/15/an-introduction-to-sonarqube/*

Walton, D. (2015), *Argument evaluation and evidence*, Vol. 23, Springer.

Appendix A

Appendix

A.1 Calculating Contribution for Developer A by Manual Evaluation

N.B. The actual names of the developers in the scikit-learn project are withheld for confidentiality purposes.

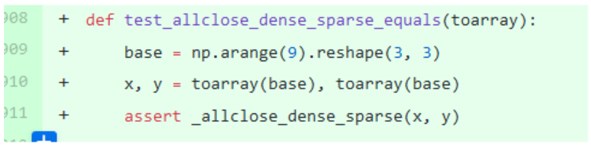
To calculate Direct Contribution, we need to first calculate separately the contribution made to the Code base, the contribution to Code Quality and the contribution to Code Maintenance. Contribution the Code Base depends on Frequency of Commits, Impact of the commits, and New functionality added to the code base

N.B. For those values calculated directly from counts of attributes available through GitHub API, examples of direct calculations and detailed procedures have already been have been shown. For those values calculated through manual assessments, pertinent examples have been shown here in the appendix to represent the chain of thought followed while assessing them.

First we begin with Developer A who was found to have a lower score in code Maintainability when analyzed manually as compared to the data-model

A reason for this may be found in the test created here which does not have a clarifying comment. This activity was penalized.

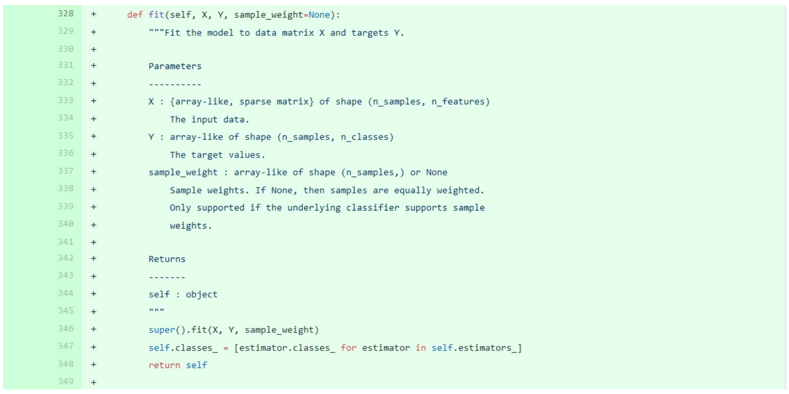
However, Developer A also wrote clear comments in other areas of code snippet, which were rewarded, for instance below figure shows well-documented comment of the



```
+ def test_allclose_dense_sparse_equals(toarray):
+     base = np.arange(9).reshape(3, 3)
+     x, y = toarray(base), toarray(base)
+     assert _allclose_dense_sparse(x, y)
```

Figure A.1: Comment-less code

parameters to be used for a function which is both coherent as well as useful (as per the code quality model) However, the percentage of such comments were rather less,

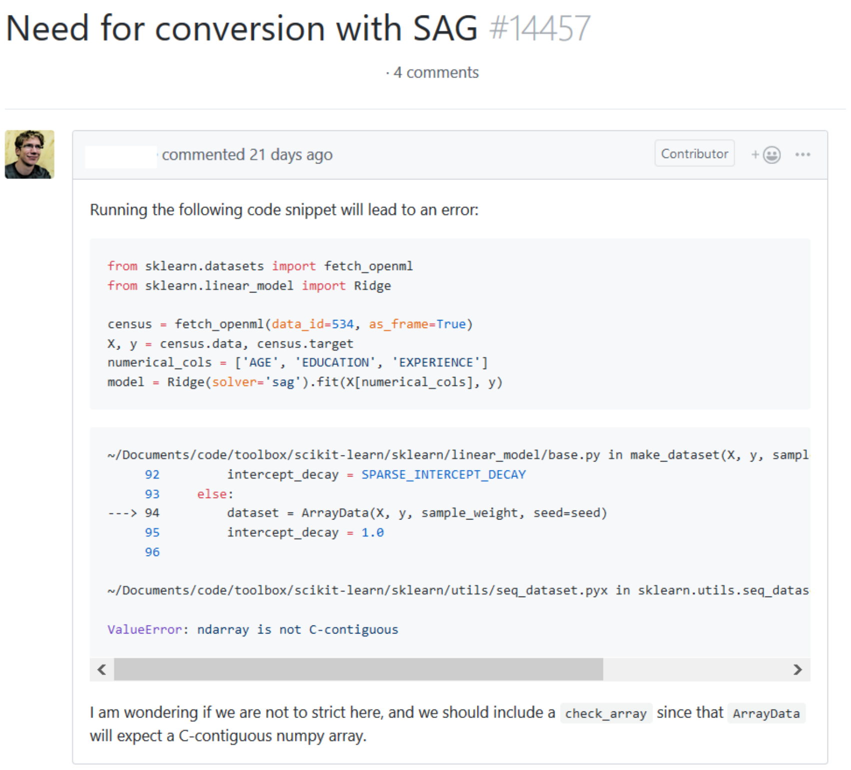


```
+    def fit(self, X, Y, sample_weight=None):
+        """Fit the model to data matrix X and targets Y.
+
+        Parameters
+        ----------
+        X : {array-like, sparse matrix} of shape (n_samples, n_features)
+            The input data.
+        Y : array-like of shape (n_samples, n_classes)
+            The target values.
+        sample_weight : array-like of shape (n_samples,) or None
+            Sample weights. If None, then samples are equally weighted.
+            Only supported if the underlying classifier supports sample
+            weights.
+
+        Returns
+        -------
+        self : object
+        """
+        super().fit(X, Y, sample_weight)
+        self.classes_ = [estimator.classes_ for estimator in self.estimators_]
+        return self
+
```

Figure A.2: Coherent and Useful Comment

in fact less than 50%.

In Issue Administration, Developer A was not consistent in raising Bugs in a standard template. Out of 4 bugs raised by Developer A, none were in the standard template used in the project and Developer A was penalized for this.


Need for conversion with SAG #14457

· 4 comments

commented 21 days ago

Contributor

Running the following code snippet will lead to an error:

```
from sklearn.datasets import fetch_openml
from sklearn.linear_model import Ridge

census = fetch_openml(data_id=534, as_frame=True)
X, y = census.data, census.target
numerical_cols = ['AGE', 'EDUCATION', 'EXPERIENCE']
model = Ridge(solver='sag').fit(X[numerical_cols], y)
```

```
~/Documents/code/toolbox/scikit-learn/sklearn/linear_model/base.py in make_dataset(X, y, sampl
     92         intercept_decay = SPARSE_INTERCEPT_DECAY
     93     else:
---> 94         dataset = ArrayData(X, y, sample_weight, seed=seed)
     95         intercept_decay = 1.0
     96

~/Documents/code/toolbox/scikit-learn/sklearn/utils/seq_dataset.pyx in sklearn.utils.seq_datas

ValueError: ndarray is not C-contiguous
```

I am wondering if we are not to strict here, and we should include a `check_array` since that `ArrayData` will expect a C-contiguous numpy array.

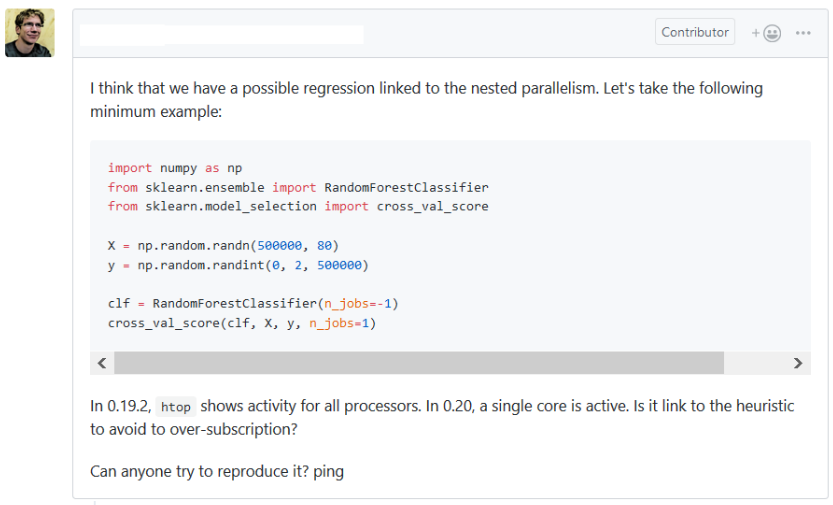
bugA.png

Figure A.3: Unformatted bug

Another example is shown below

However, one such unformatted bug did raise a pertinent issue and was rewarded.

In conversations as well Developer a was not particularly helpful in resolving issues as shown below bug2A.png



Contributor

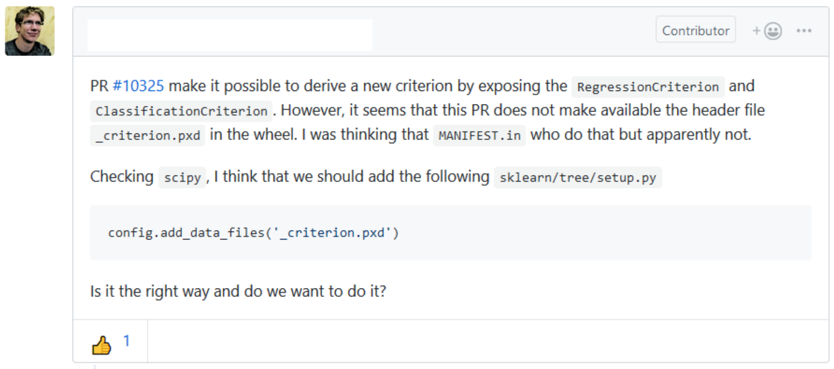
I think that we have a possible regression linked to the nested parallelism. Let's take the following minimum example:

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

X = np.random.randn(500000, 80)
y = np.random.randint(0, 2, 500000)

clf = RandomForestClassifier(n_jobs=-1)
cross_val_score(clf, X, y, n_jobs=1)
```

In 0.19.2, `htop` shows activity for all processors. In 0.20, a single core is active. Is it link to the heuristic to avoid to over-subscription?

Can anyone try to reproduce it? ping

Figure A.4: Unformatted bug2



Contributor

PR #10325 make it possible to derive a new criterion by exposing the `RegressionCriterion` and `ClassificationCriterion`. However, it seems that this PR does not make available the header file `_criterion.pxd` in the wheel. I was thinking that `MANIFEST.in` who do that but apparently not.

Checking `scipy`, I think that we should add the following `sklearn/tree/setup.py`

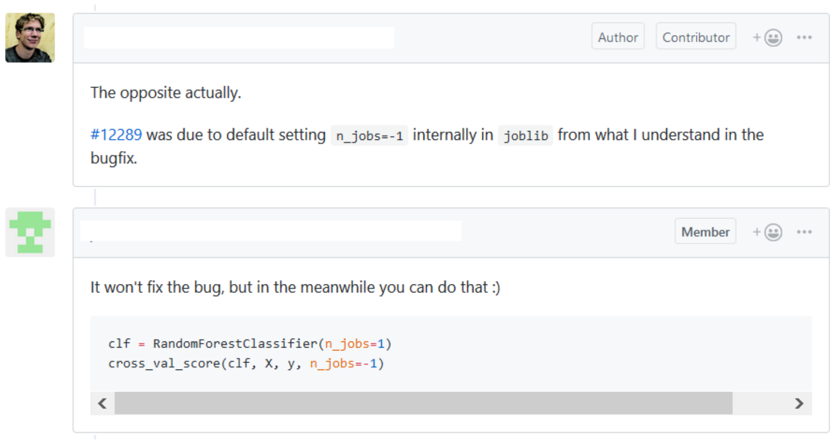
```
config.add_data_files('_criterion.pxd')
```

Is it the right way and do we want to do it?

👍 1

Figure A.5: Good Issue Raised

However, Developer A really excelled at the chat board. He was active in replying to questions


Author Contributor

The opposite actually.

#12289 was due to default setting `n_jobs=-1` internally in `joblib` from what I understand in the bugfix.

Member

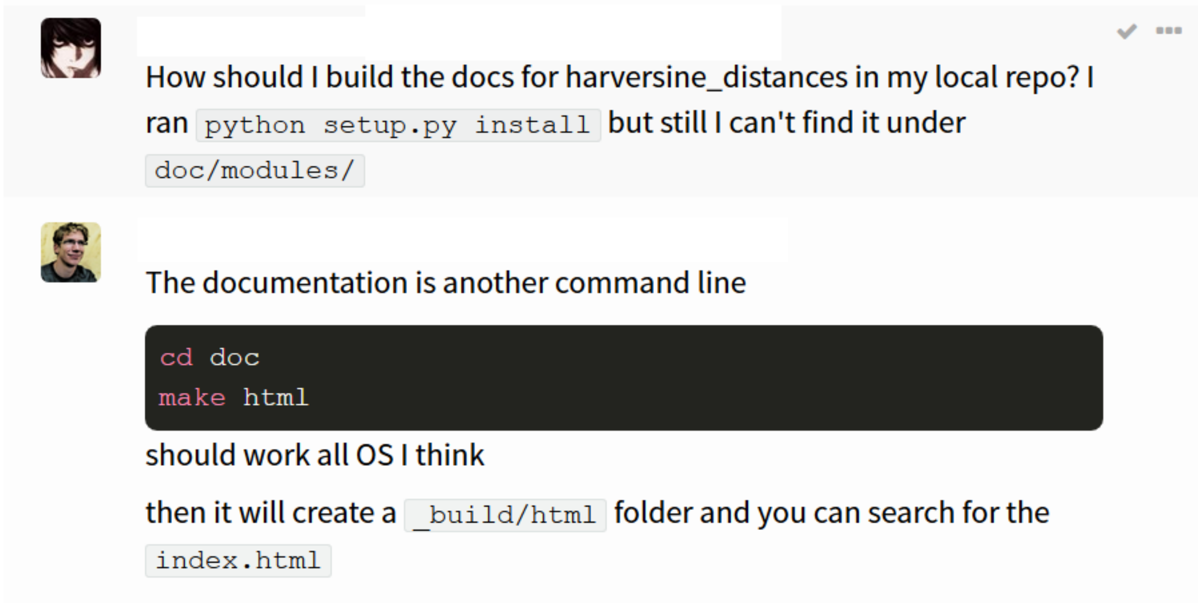
It won't fix the bug, but in the meanwhile you can do that :)

```
clf = RandomForestClassifier(n_jobs=1)
cross_val_score(clf, X, y, n_jobs=-1)
```

Figure A.6: Unhelpful Comment

Developer A was also praised by peers for his efforts and thus scored well overall in Guidance Contribution due to accumulating points for these kind of activities.

on chatA.png



How should I build the docs for harversine_distances in my local repo? I ran `python setup.py install` but still I can't find it under `doc/modules/`

The documentation is another command line

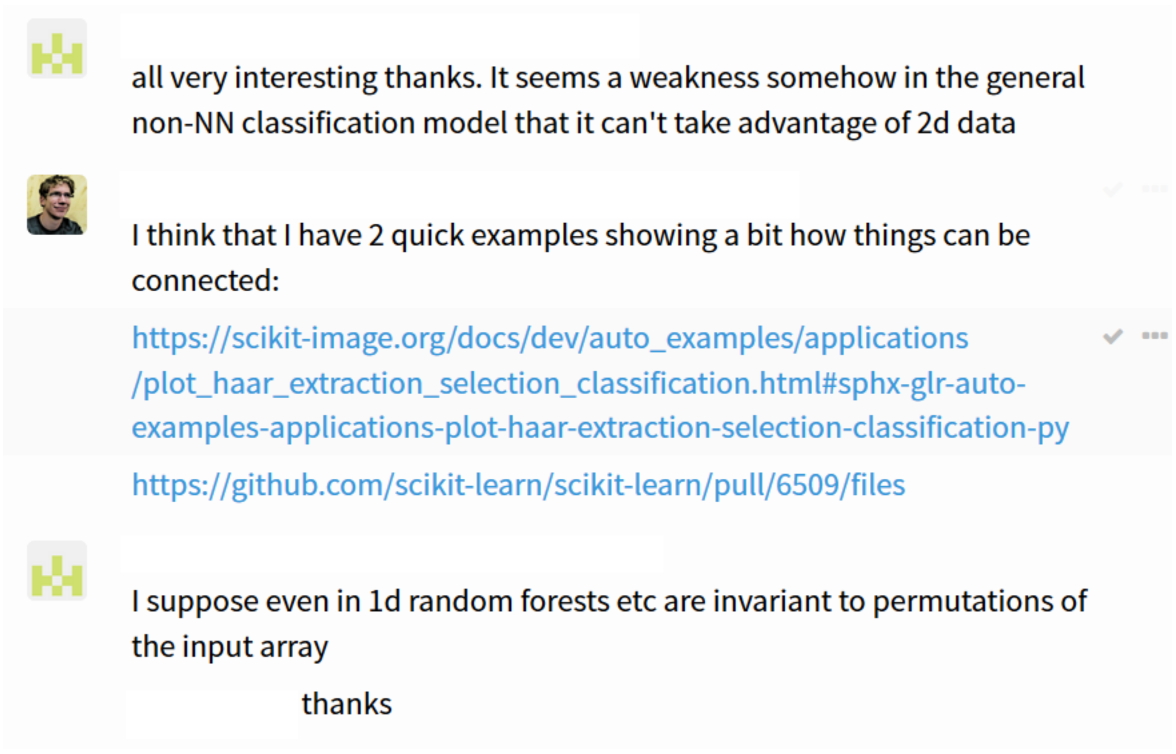
```
cd doc
make html
```

should work all OS I think

then it will create a `_build/html` folder and you can search for the `index.html`

Figure A.7: Prompt Reply by A

This sums up a manual estimate of why Developer A did not score well for Direct and Administrative Contribution, but excelled at Guidance contribution.



all very interesting thanks. It seems a weakness somehow in the general non-NN classification model that it can't take advantage of 2d data

I think that I have 2 quick examples showing a bit how things can be connected:

https://scikit-image.org/docs/dev/auto_examples/applications/plot_haar_extraction_selection_classification.html#sphx-glr-auto-examples-applications-plot-haar-extraction-selection-classification-py

https://github.com/scikit-learn/scikit-learn/pull/6509/files

I suppose even in 1d random forests etc are invariant to permutations of the input array

thanks

Figure A.8: Recognition from peers