Commit Guru: Analytics and Risk Prediction of Software Commits

Christoffer Rosen, Ben Grawi
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
{cbr4830, bjg1568}@rit.edu

Emad Shihab
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada
eshihab@cse.concordia.ca

ABSTRACT

Software quality is one of the most important research sub-areas of software engineering. Hence, a plethora of research has focused on the prediction of software quality. Much of the software analytics and prediction work has proposed metrics, models and novel approaches that can predict quality with high levels of accuracy. However, adoption of such techniques remain low; one of the reasons for this low adoption of the current analytics and prediction technique is *the lack of actionable and publicly available tools*.

We present Commit Guru, a language agnostic analytics and prediction tool that identifies and predicts risky software commits. Commit Guru is publicly available and is able to mine any GIT SCM repository. Analytics are generated at both, the project and commit levels. In addition, Commit Guru automatically identifies risky (i.e., bug-inducing) commits and builds a prediction model that assess the likelihood of a recent commit introducing a bug in the future. Finally, to facilitate future research in the area, users of Commit Guru can download the data for any project that is processed by Commit Guru with a single click. Several large open source projects have been successfully processed using Commit Guru. Commit Guru is available online at `commit.guru`. Our source code is also released freely under the MIT license.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Software Quality Analysis

Keywords

Software Analytics, Software Metrics, Risky Software Commits, Software Prediction

1. INTRODUCTION

The increasing importance and complexity of software systems in our daily lives makes their quality a critical, yet extremely difficult issue to address. Estimates by the US National Institute of Standards and Technology (NIST) stated that software faults and failures cost the US economy $59.5 billion a year [1].

Therefore, a large amount of recent software engineering research has focused on software analytics and the prediction of software quality, where code and/or repository data (i.e., recorded data about the development process) is used to provide software analytics and predict where defects might appear in the future (e.g., [2,3]). In fact, a recent literature review shows that in the past decade more than 100 papers were published on software defect prediction alone [4]. In addition to risk prediction, there has also been growing research concentrated on conflict resolution (e.g., [5,6]).

Nevertheless, the adoption of software prediction in practice has been a challenge [7–9]. One of the reasons for this limited adoption is the lack of tools that incorporate the state-of-the-art analytics and prediction techniques [9].

To address this limitation, we present a tool called *Commit Guru* that provides commit-level analytics and predictions. Commit Guru builds on the state-of-the-art analytics and prediction work to provide developers and managers with the risk levels of their commits. To derive the risk values, Commit Guru analyzes and presents a number of metrics (presented in our earlier work [10,11]) that help explain how the risk value is determined. Commit Guru is fully automated, updates daily and is capable of analyzing *any* publicly accessible git repository. Finally, Commit Guru provides a full data dump of all the commits, along with their associated metrics, to facilitate research efforts in the area of software quality analytics and prediction.

2. A WALKTHROUGH OF COMMIT GURU

The main goal of Commit Guru is to provide developers and managers with analytics and predictions about their risky commits. Similar to prior work, we consider a risky commit as a commit that introduces a bug in the future [11, 12]. Commit Guru can be used by developers, managers and gatekeepers to identify commits that have introduced bugs. An overview of Commit Guru is shown in Figure 1. Commit Guru is composed of a **backend**, which is responsible for all of the analysis of the commits and a **front-end**, which is responsible for the visualization of the analyzed data. To illustrate how both, the backend and front-end, work to support Commit Guru, we provide a brief walkthrough in this section.

2.1 Front-end of Commit Guru

When a user visits `commit.guru`, they first see the homepage shown in Figure 2. From the homepage, a user can view repositories that have already been analyzed or analyze a new repository.
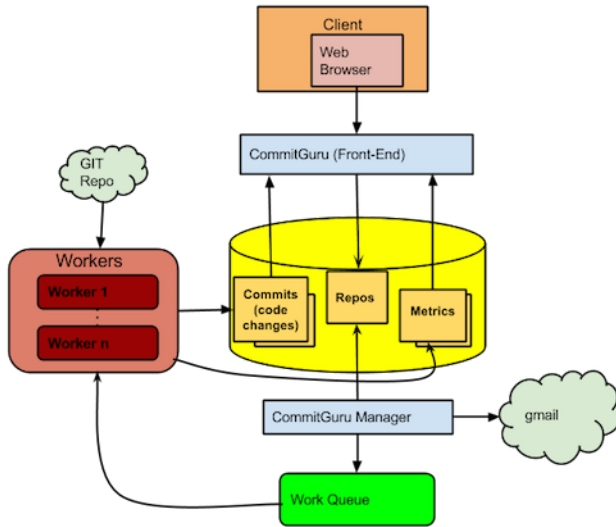
Figure 1: Overview of Commit Guru

To analyze a git repository, a user enters the URL of an publicly accessible Git repository and (optionally) their e-mail address (numbered 1 in Figure 2). The e-mail address is only used to notify the user when their repository has been fully analyzed. By default, all repositories that are submitted are viewable by all visitors of Commit Guru, however, a user can select if they do not want their analyzed repository to be publicly viewable. A user also has the option of creating an account to keep track of all of his or her repositories.

Once a repository has been fully analyzed, we present it in the recent repositories section (numbered 2 in Figure 2). Any visitor of Commit Guru can click on the repository and quickly have an overview of different analytics about the repository. These include:

- **Project commit statistics**, which show the total number of commits and the percentage of commits that are risky (and not risky).

- **Median metric values**, which show the median values for the risky and non-risky commits. For example, if the median number of lines of code (LOC) added for risky commits is 100 lines and for non-risky commits if 50 lines, then they are represented as such. The values for the risky commits are shown in red, whereas the values for the non-risky commits are shown in green. We also measure whether the difference in medians between the risky and non-risky commits is statistically significant or not. For the statistically significant differences, we highlight them by adding an asterisk '*' next to the metric's name.

In addition to the viewing the different analytics at the project level, a user can view commit-level analytics. Clicking on the 'commits' tab (labeled number 1 in Figure 3) allows the user to view all of the individual commits of the project. In this tab, there exists two views: historical and predictive data. A user can switch views by modifying the display dropdown (labeled number 2 in Figure 3). The **historical data view** show all commits in the repository and highlights risky commits in red based on those that are determined to be bug-inducing (i.e., risky). These commits, once expanded, provide a link to the commit that fixed the bug in it. Users can also apply a number of filters, e.g., filter commits by a

specific developer, order the commits based on their age. In the **predictive data view**, we show only the recent commits made in the last three months. In the prediction view, we leverage a regression model to evaluate the riskiness of each commit.

In both the predictive and historical data views, a user can click on the individual commits to view specific metric values (labeled number 4 in Figure 3). For each commit, we present the metric value for the specific commit. If the value of the metric is below the median of the non-risky commits, then it is coloured in green; if it is higher than the value of the risky commits, it is coloured in red; and if it is higher than the non-risky commits, but lower than the median value of the risky commits, then the metric value is coloured in yellow. Users can also apply a number of filters, e.g., filter commits by a specific developer, order the commits based on some criteria. Lastly, a user can click on the 'options' tab (labeled 3 in Figure 3) to see when the repository was ingested and last analyzed. Furthermore, a user can obtain a csv file containing all of the commits and their corresponding metrics.

2.2 Backend of Commit Guru

To accomplish the aforementioned analysis, Commit Guru's backend performs three sequential steps: ingestion, analysis (to calculate analytics and determine historical risky commits), and prediction (of potentially risky commits). The "Commit Guru Manager" tracks the status' of all projects and appropriate adds tasks (e.g, ingestion or analysis) to a work queue. A thread pool object maintains a pool of worker threads and assigns tasks from the work queue to available workers. This allows the system to perform multiple tasks parallel in the background. Below, we describe in a nutshell how the backend of Commit Guru works.

When a user requests a new repository to be analyzed, the front-end creates a new row in a table called `repositories`. The `repositories` table stores the name, location URL, user's email, creation date, and assigns the repository the "Waiting to be Ingested" status. The "Commit Guru Manager" observes this, and adds an ingestion task into the work queue. Our thread pool will then pop this task from the work queue and dispatch an available worker thread to start processing it. In this case, it will start ingesting the code changes in the repository and classify them (more details about the classification are provided in Section 2.2.1).

Once a worker thread has finished the ingestion task, it changes the repository's status in the table as "Waiting to be Analyzed" and saves the ingestion date in the `repository` table. The "Commit Guru Manager" notices this and triggers a new analysis task into the work queue. A free worker is dispatched and begins linking the fix-inducing commits to the bug-inducing commits. The median values of the metrics for both, risky and non-risky commits, are also calculated (more details about the analysis step are provided in Section 2.2.2).

When analysis is done, the worker changes the status of the repository to "Waiting to Build Model". The "Commit Guru Manager" again sees his and creates a new thread to build the regression model using the 'R' statistical package. After the models have been built, it stores the regression model's coefficients into a `metrics` table (more details about the prediction of risky commits are provided in Section 2.2.3.). It also creates a complete CSV data dump of all the code changes with their code change measures and whether they were bug-inducing. This file is saved on the disk to make it accessible for download by users. Next, it saves the date of the dump into a column in the repository table. We create new data dumps for each repository every month. As the "Commit Guru Manager" knows the last time a repository was ingested, it will also add tasks to pull new changes every day and update the
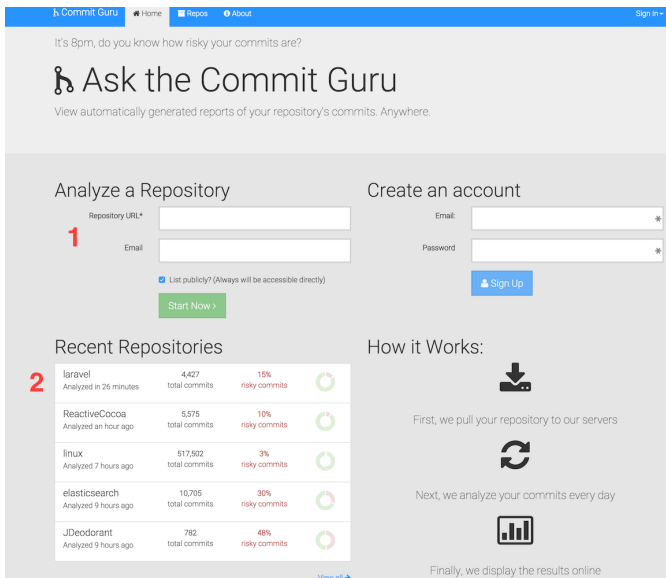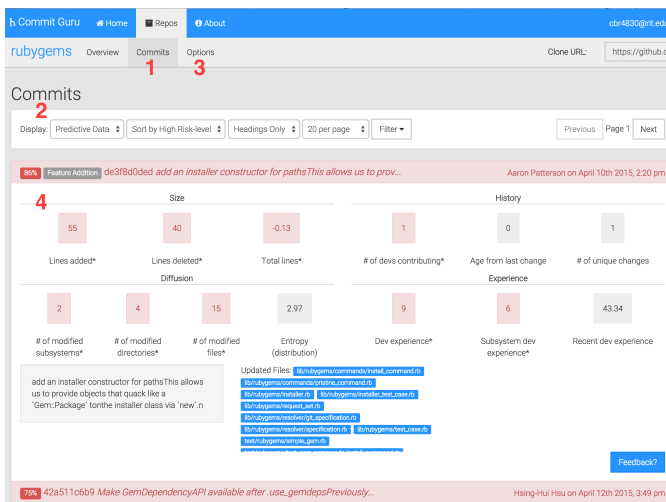
Figure 2: Commit Guru's Home Page

Figure 3: Commit View

median values and model's coefficients.

Finally, when all of this is done, the "Commit Guru Manager" changes the repository's status to "Analyzed". The user is also notified via e-mail, using Google's SMTP Server with a gmail account we created for Commit Guru, that the repository has been fully analyzed. The user can now view the riskiness of the project's code commits and download data dumps of this data

2.2.1 Ingestion and Classification

As stated earlier, the first phase a project will enter is the ingestion and classification phase. For all new projects, we download its repository and revision history, and store all of the project's data locally on the Commit Guru server. Then, we parse each commit in the revision log and record each of the 13 change-level metrics (presented in our earlier work [11]) for each commit. After extracting the commit metrics from the commit, we semantically analyze the log made by the developer in order to classify the type of commit (e.g., corrective, feature addition). We leveraged the prior work of

Table 1: Words and Categories Used to Classify Commits [13]

Category	Associated Words	Explanation
Corrective	bug, fix, wrong, error, fail, problem, patch	Processing failure
Feature Addition	new, add, requirement, initial, create	Implementing a new requirement
Merge	merge	Merging new commits
Non Functional	doc	Requirements not dealing with implementations
Perfective	clean, better	Improving performance
Preventive	test, junit, coverage, asset	Testing for defects

Hindle *et al.* [13] and use the list of keywords shown in Table 1 to classify commits. If the commit log contains one of the keywords in our defined categories, the commit is labeled as belonging to the corresponding classification. While it is possible for a commit to belong to several categories, we limit them to only one category in the following order: corrective, feature addition, non functional, perfective, and preventive. For example, if a developer fixes both, an issue and adds a new feature, it would be labeled only as a corrective change. A similar approach was used by the authors of the SZZ algorithm [14] to determine commits that were fixing.

2.2.2 Analysis and Determining Risky Commits

Next, we start linking the risky (i.e., bug-inducing) commits to those that are fixing commits. We link these these fixing commits to the risky commits by performing a diff to find which lines of code were modified. Using the annotate/blame function of the SCM tool, we can discover in what previous commits those lines of codes were modified. These commits are then marked as a potential location where the bug was introduced, i.e., potential risky commits.

To perform the diffs on each fixing commit in a project, we pipe the output of the diff into bash and echo each line back with our own delimiters at the beginning and end of each line. We do this as it is not sufficient to simply inspect each line by separating them through the newline character, since a line of code can contain arbitrary newline characters. We split this output into regions, or by the files modified. The first line contains information about the file name, which we collect. First, we check if the file name ending is in our list compiled from Wikipedia containing 144 source code and script file extensions [15]. We do not consider other types of files because a change in a document file is likely not the source of a bug. Including other file types may make the algorithm confused as it would consider code churn and documentation change as the same type of event. Additionally, it slows down our algorithm significantly for projects that have large change logs included in their updates, as the algorithm has to mark each modified line individually.

Next, we look for our delimiter to find the line that contains information about the lines of code modified and the code diff. Each region can have multiple delimiters and we refer to these as sections in the region. We divide the region up initially into two parts: the first part contains the file information and the second part contains the information about the lines of code modified followed by the code chunk. We split the second part into two more additional parts, which separates the information about the lines of code and the code chunk. From the information about the lines of code modified, we extract which line number the diff begins and mark it as the current line number. We then begin looking at the code diff and if it begins with a '-' character, we can be certain that it was a code modification. Then, we record the line number and its correspond-

ing file name and increment the current line number. We do this since code additions that did not exist in previous versions of the file may modify the line numbers. After this has been completed for all sections in the region, we perform the SCM blame/annotate function on all modified lines of code for their corresponding files on the fixing commit's parent. This will return the commit identifications that previously modified them. We determine these commits to have introduced the bugs corrected by the fixing commit and mark them. Once we have successfully linked a fixing commit to a bug-inducing commit, we mark the fixing commit as successfully linked so to not redo this every time a repository is analyzed. Only new fixing commits need to be linked, as we have already found which change(s) introduced the bugs in those fixing commits.

Once the fixing and the risky commits are labeled, we calculate the 13 commit-level metrics derived in our prior work [11]. For each metric, we calculate its median value for two sets of commits, risky and non-risky. We also perform a Wilcox test to ensure that the difference in medians is statistically significant (i.e., p-value $<$ 0.05). Then, we use the median values to present the project level analytics and to provide the user some perspective on where each commit falls compared to the set of risky and non-risky commits. For example, if the set of risky commits have a median LOC added of 100 lines, where as the non-risky set has a mean of 20 lines, then if a commit have 10 lines it is mostly safe (as far as this one metric is concerned).

2.2.3 Building Predictive Models

One main criteria/assumption that needs to be satisfied for the SZZ (and our algorithm that determines risky commits) to work is that enough time needs to have elapsed since a commit, for the commit to have the chance to be fixed. Therefore, we also build a predictive model that trains on historical data and makes predictions for commits performed in the last three month period.

Our prediction model is a logistic regression model, although in theory we can apply any other model. We build the prediction model incrementally by adding one metric to the model at a time. If the metric is statistically significant and does not cause any previously added metric to no longer be significant, it is added to the model. We repeat this for all 13 metrics in the following order: lines added, lines deleted, lines total, no. subsystems, no. directories, no. files, no. developers, age, no. unique changes, experience, recent experience, subsystem experience, and entropy. Once we build the model, we use the coefficients of the model to predict the risk of the commits made in the last 3 months. When a user submits a repository to be analyzed, they can see the predicted risk of their commits for any commit made in the last 3 months using the predictive view of Commit Guru.

3. CONCLUSION

In this paper, we present our tool, Commit Guru, that performs analytics and predicts risky software commits. Commit Guru uses 13 metrics introduced in our prior work [11] to give its analytics and make its predictions. Commit Guru is publicly available on the web at `http://www.commit.guru`. Practitioners can use Commit Guru to identify recent commits that are more likely to contain bugs and better understand the overall quality of a software project. This can be useful to find those changes to prioritize verification activities such as code inspections as well as seeing which quality change metrics might be good indicators for bugs. Researchers can use Commit Guru to study a large collection of quality commits and their metrics from any GIT SCM based open source repositories for future research in this area.

4. REFERENCES

[1] "The economic impacts of inadequate infrastructure for software testing." http://www.nist.gov/director/planning/upload/report02-3.pdf.

[2] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007, pp. 1–7.

[3] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, pp. 181–190.

[4] E. Shihab, "An exploration of challenges limiting pragmatic software defect prediction," PhD Thesis, Queen's University, 2012.

[5] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Crystal: Precise and unobtrusive conflict warnings," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 444–447.

[6] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 342–352.

[7] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin, "Future of mining software archives: A roundtable," *IEEE Software*, vol. 26, no. 1, pp. 67 –70, Jan.-Feb. 2009.

[8] A. Hassan, "The road ahead for mining software repositories," in *Frontiers of Software Maintenance, 2008*, Oct. 2008, pp. 48 –57.

[9] E. Shihab, "Pragmatic prioritization of software quality assurance efforts," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 1106–1109.

[10] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 62:1–62:11.

[11] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 757–773, 2013.

[12] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 181–196, 2008.

[13] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: A taxonomical study of large commits," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08, 2008, pp. 99–108.

[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[15] "List of file formats," http://en.wikipedia.org/wiki/List_of_file_formats, Jun. 2015.

APPENDIX

A. SOURCE CODE AVAILABILITY

Our source code is freely available under the MIT license. The back-end of the Commit Guru system can be viewed and dowloaded at `https://github.com/CommitAnalyzingService/CAS_CodeRepoAnalyzer`. The front-end can be found at `https://github.com/CommitAnalyzingService/CAS_Web`.

B. COMMIT GURU DEMONSTRATION

To demonstrate Commit Guru in action, we walk through mining and analyzing the code changes for the open source project pip. This is a popular tool for installing Python packages. The project's source code is publicly available on GitHub, which is a popular place for developers' to store projects using git as its version control system. Pip can be seen on GitHub via the web at `https://github.com/pypa/pip`. It has 3,944 code changes with 195 contributors and currently has 86 pull requests.

B.1 Walk Through Analyzing Pip

Adding Pip: We first get the url to pip's git repository. This is all we need to mine the project's code changes and see those that are risky through the Commit Guru web tool. We navigate to `commit.guru` with our web browser. Here, we can input pip's git repository url and our email address as shown in Figure 4. After we click "Start Now", the repository will start to get processed by Commit Guru.

Overall Commit Statistics: When Commit Guru has finished processing a repository, it will send an e-mail notification to the owner, if this was provided. It will also show up by default to the recent repositories list on the home page unless disabled. When we click on the link given by the e-mail or the repository in the recent repositories table on the home page, we see the overall commit statistics as shown in Figure 5. Here, we can quickly see that 20% of code changes are marked as bug-inducing (number 2 in Figure 5). Additionally, we can view the averages for each of the commit metrics (number 3 in Figure 5) for the changes that were bug-inducing (colored in red) or not (colored in green). Metrics with an asterisk('*') indicate that the differences between the bug-inducing and non bug-inducing commits are statistically significant. For pip, we see that all metrics are statistically significant. Interestingly, we also learn that the more developers that touched a file for this project, the less likely it was to contain a bug. Other useful information can also be found from this page, particularly for researchers. Hovering over the blue question mark next to the metric name will display a popup describing it. For instance, hovering over subsystem dev experience metric will show a popup explaining that this metric represents the number of commits the developer made in the past to the subsystems that are touched by the current commit (number 4 in Figure 5).

Viewing Code Changes: In the overview page, there is a navigation bar for the repository to switch between different pages of the repository (labeled as 1 in Figure 5). After clicking on the "Com-mits" tab in the repository navigation bar, we see a list containing the commits for this repository. By default, it shows an historical view as shown in number 1 in Figure 6. Commits colored in red indicate that they are bug-inducing (number 3 in Figure 6). On the left-side for each commit, we also show it's classification (if available). It is also possible to filter for specific commits by clicking on the filter drop-down menu (number 2 in Figure 6) and specifying the commit hash, commit message, author email, or the commit classification.

Finding Risky Commits: Changing the display from historical to predictive data in the drop down on the commits page (number 1 in Figure 6) will change the view to show all code changes in the past three months and how likely they contain bug(s). There is an additional element on the left-side for each commit (number 1 in Figure 7). This is a percentage number that indicates the probability of it containing a bug. It is color coded accordingly: green indicates low risk, yellow indicates medium risk, and red indicates high risk. However, in the predictive data view, a red colored commit indicates that it is more probable to contain a bug using our regression algorithm instead of indicating that it is bug-inducing (historical data view). Here, we can browse through all the recent code changes and easily inspect those that are most likely to introduce bugs by sorting on high risk-level (number 2 in Figure 7). For example, we can see that commit 17352765f0 is likely to introduce a new bug, making it a good target for conducting additional verification activities (i.e., code review).

Inspecting Individual Commits: In both the historical and predictive data displays, it is possible to inspect individual commits by clicking on it as shown in Figure 8. This will expand the box, showing all the individual code change metrics (number 1 in Figure 8), the commit message (number 2 in Figure 8), the updated file names (number 3 in Figure 8), an option to give feedback whether it was useful (number 4 in Figure 8), and a link to the commit that fixed a bug it contained (number 5 in Figure 8), if applicable. Each metric is color coded according to it's risk factor based on our median model. If the metric is less than the average of non-buggy commits, it is green (low risk). If it is between the average of a buggy and non buggy commit, it is colored yellow (medium risk). Finally, if it is the same as the average of a buggy commit or higher, it is colored as red (high risk). This is useful for quickly understanding why a certain commit was flagged as risky in the predictive data view and seeing a summary of which files were changed and by whom.

Downloading Data Dumps: In the repository navigation bar, clicking on "Options" will take us to a view that allows us to download a data dump of the code changes for the repository as shown in Figure 9. It also shows the last time Commit Guru downloaded code changes from the repository, the last time they where analyzed, and when the data dump was done. Clicking on the "Download dump (.csv)" button (number 1 in Figure 9) will start the download of a csv file containing all code changes with their change measures and whether they are marked as containing a bug. Figure 10 shows an example of the contents in the data dump after doing this for the Pip project.
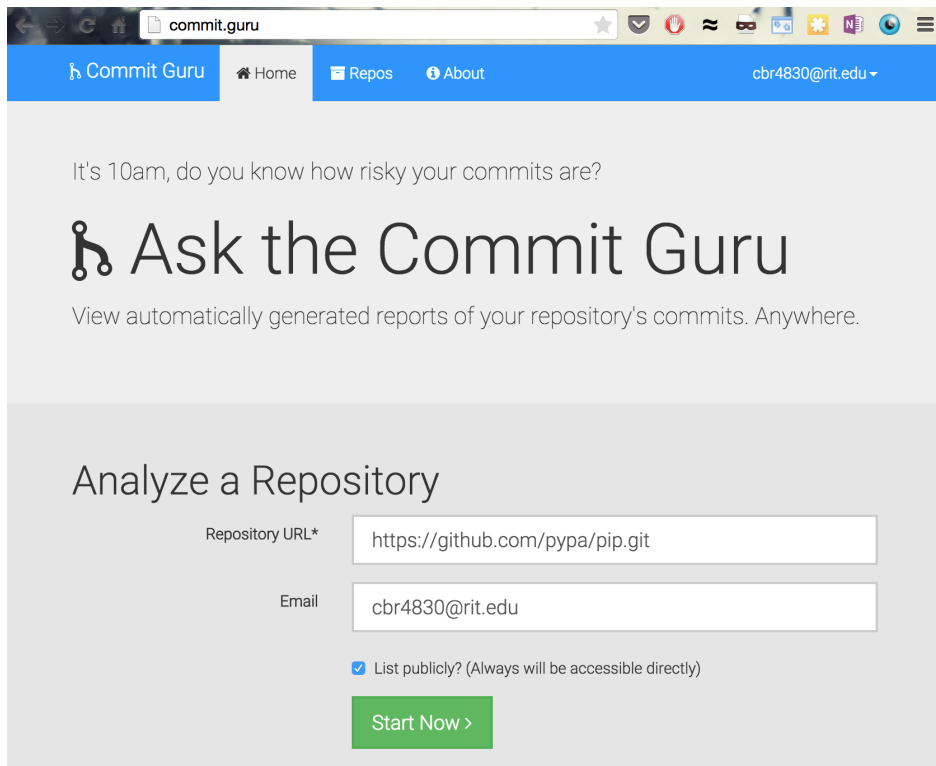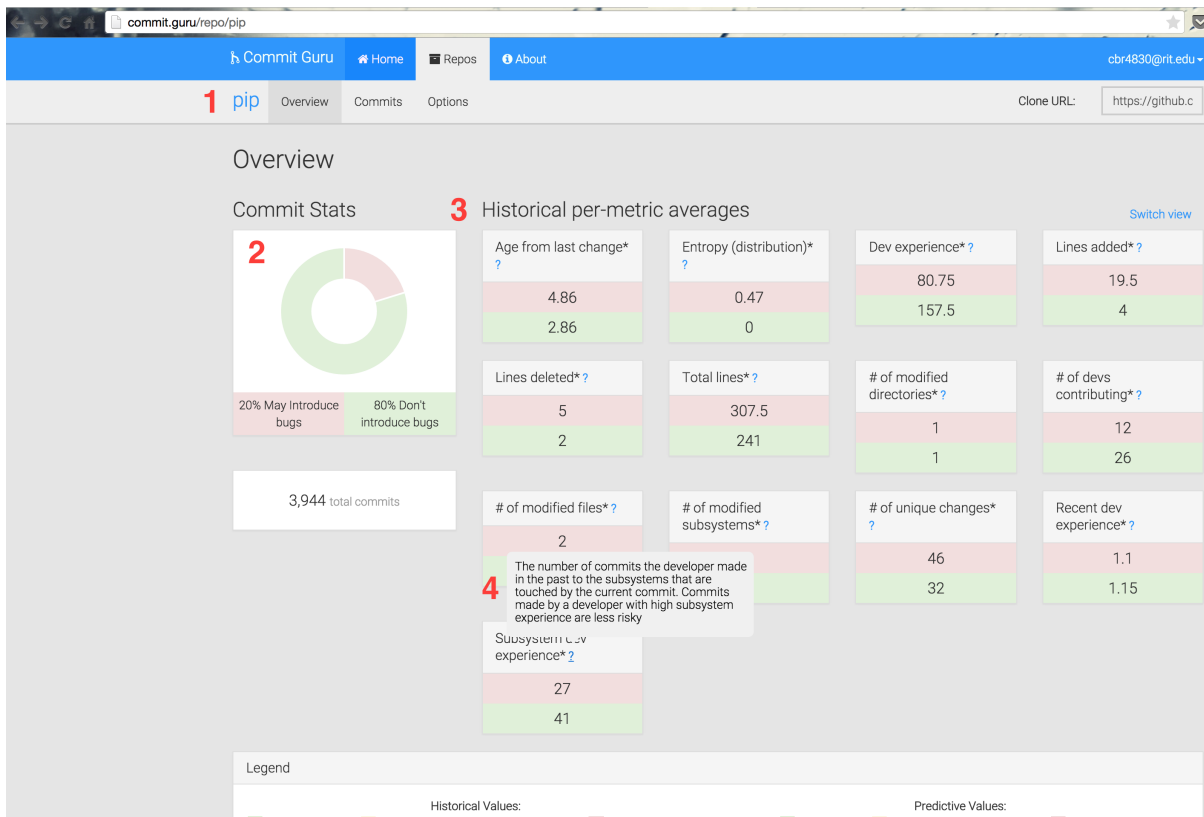
Figure 4: Adding a Repository

Figure 5: Commit Statistics

Figure 6: Commits - Historical Display

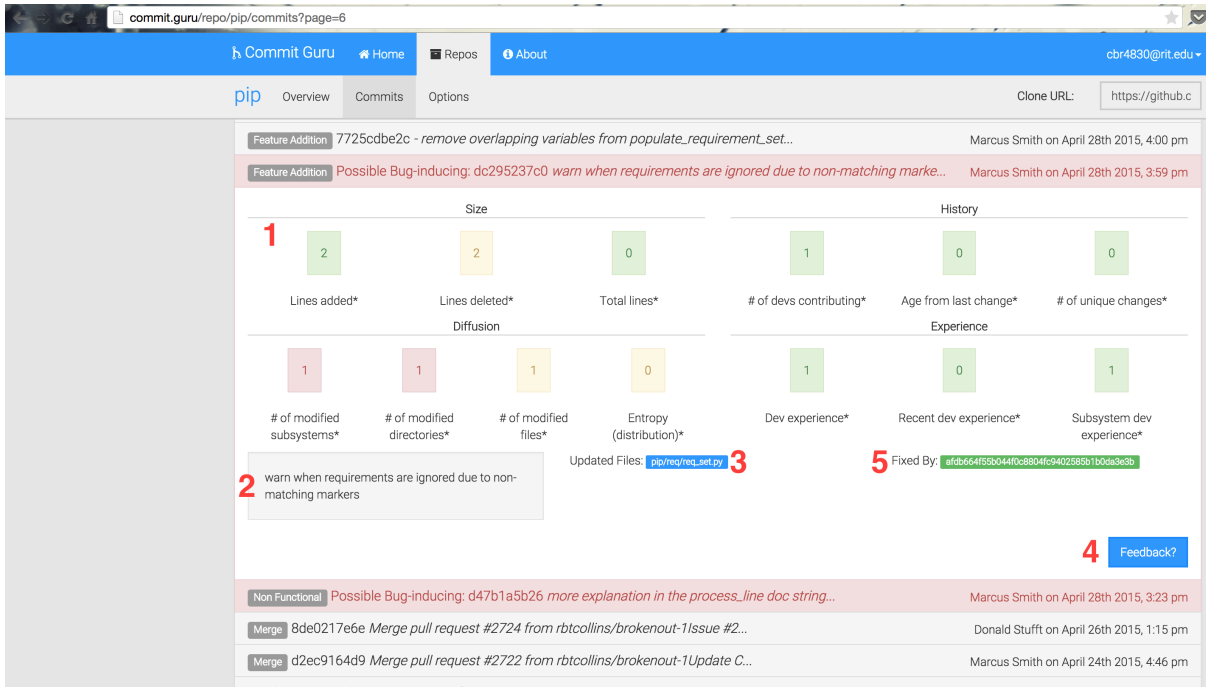**Figure 7: Commits - Predictive Data Display**

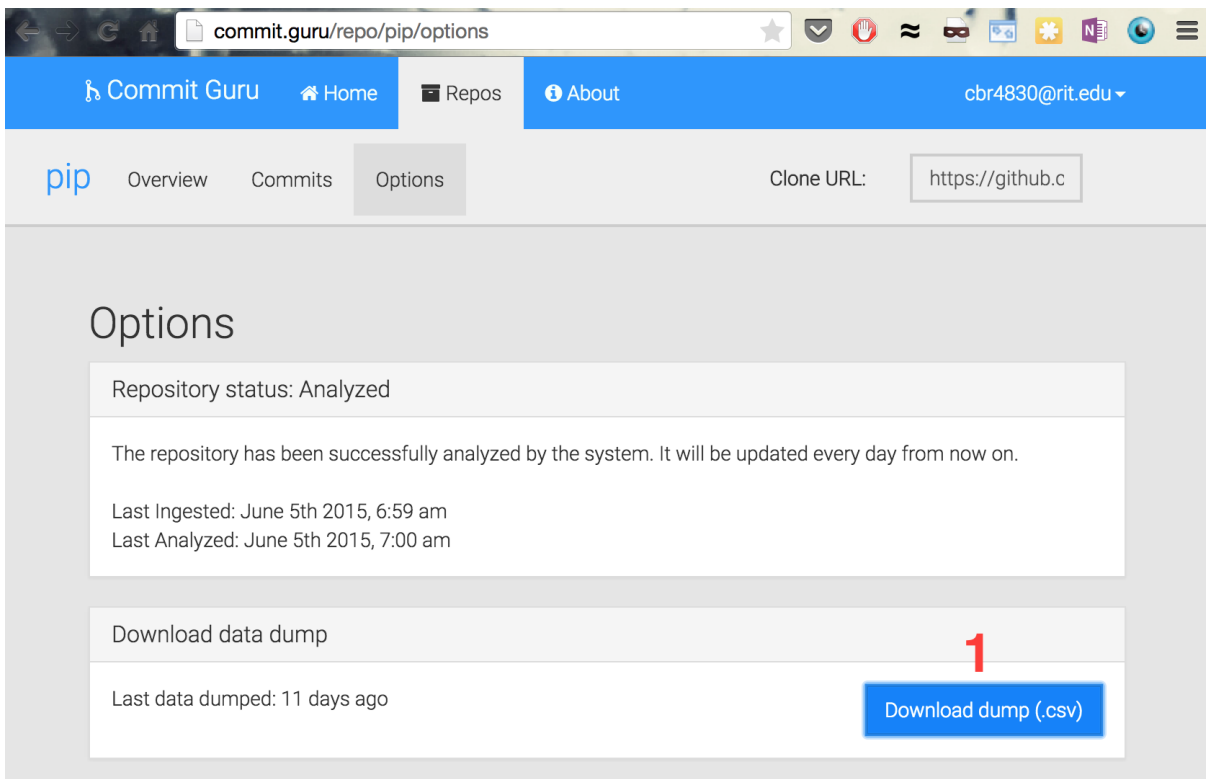**Figure 8: Inspecting an individual commit**

Figure 9: Downloading Data Dump

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	ns	nd	nf	entrophy	la	ld	lt	ndev	age	nuc	exp	rexp	sexp	is_buggy
2	1	1	2	0.99923284	92	92	142.5	2	6.88537037	9	4.5	1.07263266	5	TRUE
3	1	1	1	0	114	7	39	1	7.02335648	1	3	1.14238207	3	TRUE
4	1	1	1	0	13	10	2	1	0.00420139	1	1	239.016529	1	TRUE
5	1	1	1	0	6	4	0	1	0	0	0	0	0	TRUE
6	1	1	1	0	62	53	35	2	0.00090278	12	15	1108.69231	15	TRUE
7	1	1	2	0.87086447	25	23	14.5	2	1.43711227	9	9.5	1.34789892	10	TRUE
8	2	2	2	0.26676499	9	13	147.5	1	2.68585648	8	16.5	1.28821006	3	TRUE
9	1	1	2	0.91829583	43	50	0	1	0	0	0.5	0	1	TRUE
10	1	1	1	0	57	60	-2	2	0.18396991	4	6	6.43567159	6	TRUE
11	1	2	3	0.96656787	26	26	2.33333333	2	-0.2608565	1	1	-0.092614	2	TRUE
12	1	1	1	0	56	26	16	1	0.06148148	3	4	17.2650602	4	TRUE
13	1	1	1	0	290	273	0	1	0	0	0	0	0	TRUE
14	2	2	3	0.92184106	54	44	97.6666667	1	0.09143519	7	11	7.6835443	2	TRUE
15	3	3	4	1.72309134	21	21	79	1	2.91365451	5	7.5	1.15019368	1	TRUE
16	1	1	1	0	2	2	9	1	3.87741898	1	5	1.25790352	3	TRUE
17	3	4	5	1.3346353	429	108	0	1	0	0	2	0	2	TRUE
18	1	1	1	0	3	3	548	41	80.589456	60	32	1.01240857	17	TRUE
19	1	1	1	0	11	3	329	37	177.265	40	0	0	0	TRUE
20	1	1	1	0	41	0	734	40	1.26334491	33	66	1.79154948	12	TRUE

Figure 10: Data Dump Contents