Overview of the MAPS Plagiarism Detector

Mateo Kocaj, Manthan Thakker, Sakshi Tonwer, Preethi Anbunathan

Team 114

Problem Statement:

Plagiarism is a prevalent and considerable problem at many universities. Currently, there are a plethora of modern day detection systems for detecting plagiarism in essays, research papers, and other text-based assignments, however, not many systems exist for programing assignments. To add on to this issue, detecting plagiarism in programming assignments can be much more difficult since they are often implemented in multiple files and plagiarism tactics go beyond textual differences.

Some of these tactics that go undetected in programming classes, while using traditional systems, include but are not limited to:

- Changing the name of variables
- Changing the name of classes
- Changing the name of methods
- Splitting the code into multiple functions or files

Although some code plagiarism detection systems do currently exist, one of the more popular being Stanford MOSS, they lack a user interface and can only be applied to two assignments at a time making them nearly impractical for a large classroom setting. Therefore, many professors are only left to rely on their teaching assistants to hopefully recognize similarities between assignments.

The rising demand of the computer science field brings in a more students every year, increasing the number of cases of cheating along with it. There is a clear demand for a much more user-friendly system that can detect plagiarism in multiple student assignments. Our team set out to solve this issue by developing a multi-user web application that detects plagiarism in python code of multiple submitted assignments.

Result Overview:

The system developed employs three different types of algorithmic strategies and allows the user to choose the strategy they wish to use. Alternatively, the user can select an "all strategies" option which combines all three strategies and has been trained to return similar results to that of the Stanford MOSS system.

In order to ease the already burdensome process of submitting and grading assignments, the system was designed to integrate with GitHub so that there is only one point of submission for students, therefore, limiting any sort of middleman headaches. The teaching staff can login to the application, add a new course using the GitHub repo URL (if not already done so), and all assignments for the given URL will automatically populate on the assignments page. The teaching staff can then select a strategy and generate the results for a given assignment. Results can be re-generated at any given time and a history of the last generated results is stored in a database and viewable at any point up until new results are generated for the assignment.

Once results have been generated for an assignment the teaching staff is able to go to the results page and view a list of comparisons with each comparison showing the main folder name of the two students, the strategy used to generate the result, and a percentage match of between the two students. The list is grouped via strategy used, sorted in descending order by percentage, and color coded to achieve further ease of use.

A further breakdown of a result is shown when a user selects a given result. In the comparison page, snippets of code are grouped together to show similarities between them, where the line of code in question is highlighted yellow.

To manage the work-flow the team followed agile methodology and used Jira for issue tracking. In order to maintain a standard of quality, Jenkins and SonarQube were set up to enforce an 80% code coverage in order to merge any new feature to the master branch, ensuring that all code was tested properly. Once a new feature was ready for merging, a pull-request was created which had to be approved by another teammate. Figure 1 below shows a history of Jira issues created and completed, while Figure 2 is a SonarQube report outlining overall code quality on master.

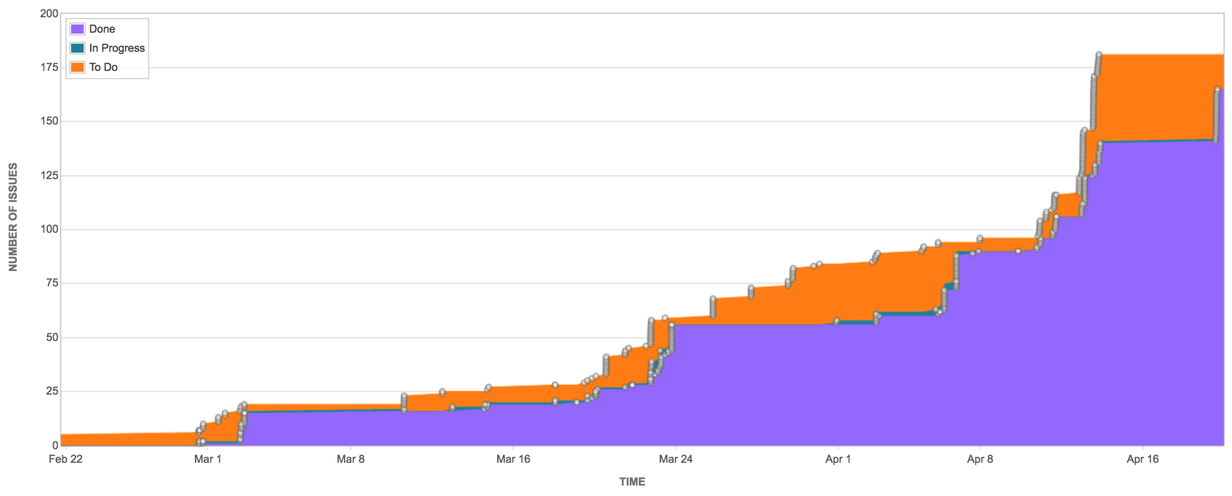
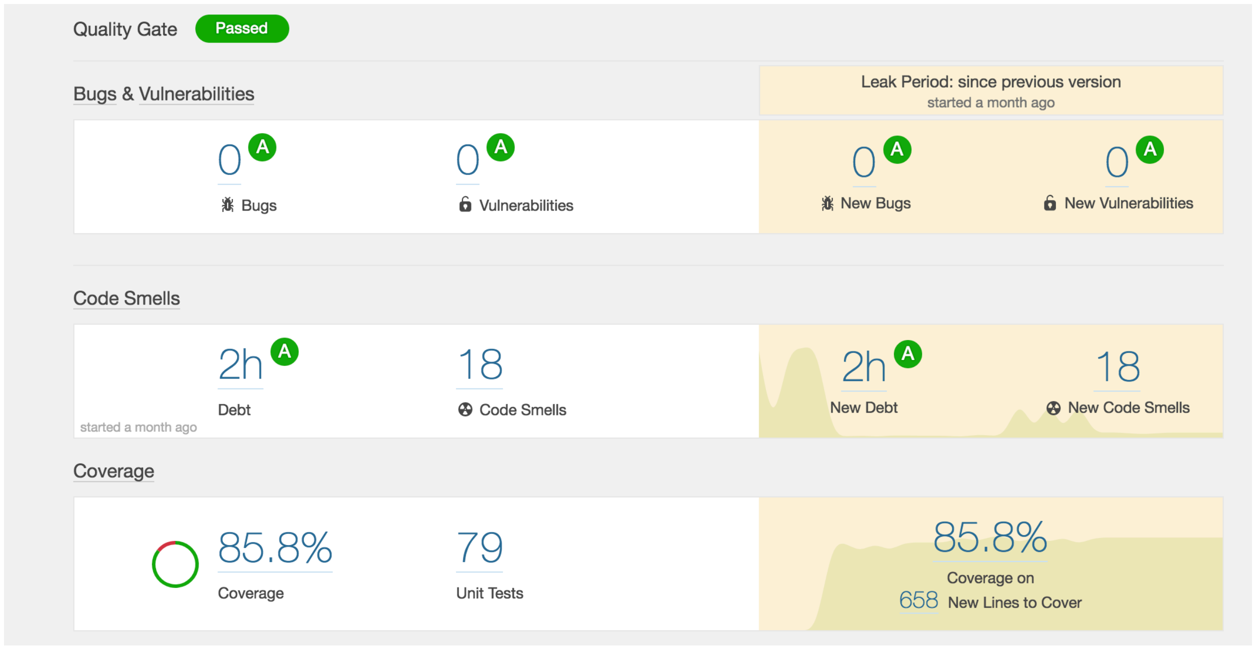


Figure 1. Number of Open, Closed, and In Progress issues over time.



Quality Gate Passed

Bugs & Vulnerabilities

Leak Period: since previous version
started a month ago

0 A Bugs

0 A Vulnerabilities

0 A New Bugs

0 A New Vulnerabilities

Code Smells

2h A Debt

started a month ago

18 Code Smells

2h A New Debt

18 New Code Smells

Coverage

85.8% Coverage

79 Unit Tests

85.8%
Coverage on
658 New Lines to Cover

Figure 2. SonarQube report outlining total number of code smells, unit tests, and overall percent coverage.

Development Process:

Agile methodology was followed throughout the development process and the work was divided into three parts encompassing the requirements, design, and implementation phase. The implementation phase was further broken down into three sprints with each sprint having a review at the end of it where the overall achieved functionality and code quality was discussed with a teaching assistant.

Phase A: Requirements

i. **Inception**: based on the problem statement a series of questions were created by the team in order to gain further clarification on the scope of the project.

ii. **Elicitation**: a meeting was held with the stakeholder (professor) to gain a deeper insight on the requirements and receive answers on the prepared questions.

iii. **Elaboration, Specification**: uses cases and a UI mockup were created based on the requirements elicited by the client.

iv. **Validation**: Phase A was submitted to the client for feedback where the client validated our understanding of the requirements phase.

Phase B: Design

Applying the use cases and UI mockup created in Phase A as a starting point, a system design that reflected the architecture was constructed that included:

i. UML diagram for the major components of the system and relationship between them.

ii. UML diagram for key data structures that were manipulated by and/or exchanged between components.

iii. Java interfaces that reflected these components and data structures.

Phase C: Implementation

The implementation phase was split into three sprints, each of two weeks length, where the development team followed a set pattern:

i. Meetings at the start of the sprint to decide on features to be implemented and stretch goals to be achieved. Each feature was broken down to tasks which were assigned to individuals. Interfaces were discussed and agreed upon which served as service specifications.

ii. Tasks were independently implemented, and meetings held on alternate days to discuss the current progress and challenges/blockers being faced.

iii. All tasks were integrated into a single feature at the end of the week and deployed after review.

iv. Sprint review held on the second week to validate all work done so far and gain any feedback by the client (TA)

By breaking down stories into smaller tasks which were developed independently, a faster development process was achieved without restricting the work of others. Additionally, lower coupling and a higher cohesion was achieved since features were broken down to a point where subtasks are able to be used by other services.

1. Sprint 1: Comprised mostly of project setup including environments, technology, infrastructure and quality checks. Spring Boot was set up as the project framework along with Angular 5 as the frontend framework and MongoDB as the database. The project was deployed to an AWS EC2 instance and Jenkins and SonarQube was deployed for code manageability and quality. Implementation of a basic comparison strategy along with login and register functionality were achieved. Overall, all base expectations and stretch goals were achieved for this
2. Sprint 2: At the end of this sprint two sophisticated comparison strategies were employed for detection which included longest common subsequence and document fingerprinting. Along with the algorithms, the UI was fully implemented for adding a new course and assignment. Successfully integrated the Git API in order to clone repositories where assignments were handed in. Log4j was set up as a next generation asynchronous logging framework that sends e-mails to the system admin on any system failures. Overall, the team managed to hit all base expectations as well as all stretch goals for the sprint.
3. Sprint 3: Most of the base goals in this sprint were already achieved in sprint 2, so for this sprint more focus was put on the refinement of backend functionality as well as user interface. This was achieved by polishing up frontend routing, creating help and information boxes throughout the application, implementing progress bars so that the user is always aware of what the system is doing, and displaying more understandable error texts in the frontend when something went wrong. Admin functionality was also

implemented where usage statistics were displayed along with user manageability functionality that displays all registered users with the functionality to delete a user. Furthermore, a domain name was purchased for the project (www.plagiarismpython.com) and google analytics was set up to track system statistics such as real time visitors, visitor demographics, visitor usage patterns throughout the application, and much more.

Retrospective:

The Good:

- The overall consensus of the team is that this project has definitely proved a valuable learning experience. The requirements gathering phase where teams were to elicit specifications from the client by taking into consideration the language they used with non-tech speakers was a beneficial exercise that some CS students do not get a chance to practice regularly.
- Following the agile methodology has helped us gain a much greater insight on one of the more popular tech world methodologies and the experience gained throughout the software development lifecycle gave us a better perceptive when it comes to approaching and breaking down larger problems into much more manageable subtasks.
- Smart commits simplified the developed process and helped the team track progress more efficiently.
- Following good practices by:
    - Integrating loggers
    - Integrating SonarQube
    - Enforcing quality through an 80% quality gate on Jenkins before deploying to AWS
    - Using Jira to manage issues
    - Requiring pull-requests to be reviewed by a teammate

  While understandably a headache at times, have ultimately helped us achieve much greater software quality and manageability.

The Bad:

- Dev-ops issues related to AWS servers, Jenkins, and SonarQube were a major headache and the lack of understandable documentation made it very challenging to fix which consumed a lot of time during sprints.
- Using AWS costs money.