Investigating Patterns in Software Project Attributes Hosted in an Open Source Code Repository

Filip Variciuc
variciu2@uic.edu

Omaid Khan
okhan23@uic.edu

Arnav Dahal
adahal3@uic.edu

ABSTRACT

As new and upcoming programmers, the world of software can be daunting at times. It is nice to start out programming in a language that is easy to read, easy to write, and most importantly, is plentifully documented. More important than this, however, is having examples of code snippets, comprehensive and tested libraries, and an active community from which one can reference for information. We pose a number of questions that a newcomer might ask about various languages that he/she is considering to start learning. We explain the rationale behind these questions and then answer the questions based on facts we generated from attributes found in the metadata of 10,000 GitHub repositories. The goal of this project is to analyze what makes GitHub repositories popular and how that may help a new programmer get affiliated with multiple languages.

1. INTRODUCTION

We download the top 1,000 most popular repositories for the top 10 most popular languages on GitHub. We then took the metadata from GitHub's API and put it into a NOSQL database (MongoDB). We then used MongoDB in conjunction with Rapid Miner in order to find patterns on the data. Many of the patterns were revolved around language that the project was written in. This data was filtered and uploaded into a relational database running MySQL. From there we were able to perform requests to the MySQL server to aggregate data and perform computations. Both the NOSQL database and relational database were hosted on Google's Cloud Platform.

2. IMPLEMENTATION

In this section, we describe how the system was designed in order to obtain the information that led to the answer of our questions:

2.1 Components

We hosted our components on Google's Cloud Platform. We deployed a virtual machine with an instance of MongoDB and another SQL Database instance of MySQL. Logging warnings, errors, and debugging information was facilitated through Google Stackdriver Logging.

2.1.1 GitHub Developer API

The GitHub Developer API allows us to retrieve metadata about repositories on GitHub using RESTful requests that return JSON payloads. The API also facilitates the use of OAuth and client secret authentication. We used the latter for our requests. We send in structured queries that request project repositories of a certain language and sort it by popularity. Each payload return is considered a page. A page contains the data for 30 repositories. To get 1,000 projects (which is the maximum that GitHub API will return), we retrieve up to the 34th page of results. We then parse these payloads for the meta data of types NAME, SIZE, STAR_GAZER_COUNT, WATCHERS_COUNT, LANGUAGE, HAS_ISSUES, HAS_DOWNLOADS, HAS_WIKI, HAS_PAGES, FORKS_COUNT, OPEN_ISSUES, WATCHERS, SCORE, FORKS.

Analyzing this kind of data is much more manageable than analyzing source code. The main reason for this is that this data is already in a structured format (JSON), and implementing the right libraries to extract its payload is easier than answering questions about source code file contents, where all entities are essentially words that must be broken up into language keywords (on an individual language basis). GitHub imposes a limit to how much data can be requested via their API, referred to by them as a rate limit. Once this rate limit is reached (30 requests/minute), the developer's request is rejected with a timeout. We circumvent this limitation by implementing a fault tolerance mechanism that waits until making the next request (much like a machine gun needs time in between bursts of activity to prevent overheating).

2.1.1* Issues with GitHub Developer API

The WATCHERS_COUNT and WATCHERS type seemed to be bugged and just take the value of the STARGAZER_COUNT and it does not reflect the actual number of watchers for the repositories. For this reason, we exclude this from the results later. We are unsure about whether GitHub knows about this issue; we have not submitted a ticket regarding this.

2.1.2 MongoDB

The JSON form of the metadata pulled from the GitHub API was in a different format from one that is insert-able into MongoDB. Therefore, we were forced to construct a new JSON object of the parsed metadata and insert it into the MongoDB after specifying an IP address and port number. This new JSON form also simplifies working with Rapid Miner.

2.1.3 Rapid Miner

Rapid Miner aided us in visualizing our data with ease. Rapid Miner generated tables, charts, and graphs that gave us a better understanding of what the GitHub metadata looks like and what sort of interesting questions we could draw from it. We used a JDBC connector to connect the MongoDB instance to Rapid Miner.

2.1.4 MySQL

Before uploading project information to the relational database, we had to decide which attributes of the project we would discard, as not all of the data in the JSON payload that got uploaded to the NOSQL database was relevant in our findings. Having the filtered information in database form made querying the data an easier and more flexible experience. Unfortunately, setting up the MySQL instance for our particular project was not all done programmatically. We had permission issues creating the database object programmatically. Therefore, we created the database object inside the instance itself (after connecting into it remotely). The code required to create the database is:

CREATE DATABASE FinalProjectDB;

Everything else, from creating the tables and inserting the data into it, to formulating queries and delegating them to the server, was all managed programmatically. This is the code executed to create the tables in the relational database:

```
statement.execute(
  "CREATE TABLE LanguagesTable(" +
    "languagesTableID INT NOT NULL AUTO_INCREMENT,"
    "language VARCHAR(100) NOT NULL," +
    "PRIMARY KEY(languagesTableID)" +
    ");")
statement.execute(
  "CREATE TABLE ProjectsTable(" +
    "projectsTableID INT NOT NULL AUTO_INCREMENT,"
    "_id VARCHAR(100) NOT NULL," +
    "fork_count INT NOT NULL," +
    "stargazer_count INT NOT NULL," +
    "watcher_count INT NOT NULL," +
    "size INT NOT NULL," +
    "open_issues_count INT NOT NULL," +
    "languagesTableID INT NOT NULL," +
    "has_issues BOOLEAN NOT NULL DEFAULT FALSE," +
    "has_downloads BOOLEAN NOT NULL DEFAULT FALSE,"
    "has_wiki BOOLEAN NOT NULL DEFAULT FALSE," +
    "has_pages BOOLEAN NOT NULL DEFAULT FALSE," +
    "PRIMARY KEY(projectsTableID)," +
    "FOREIGN KEY (languagesTableID) REFERENCES
LanguagesTable(languagesTableID)" +
    ");")
```

3. QUESTIONS AND FINDINGS

1. As a beginner programmer, getting started is the most difficult aspect of learning a new programming language. Our data suggests that PHP, Python, and Ruby were the top 3 programming languages with repositories containing a wiki page. [Figure 3A]

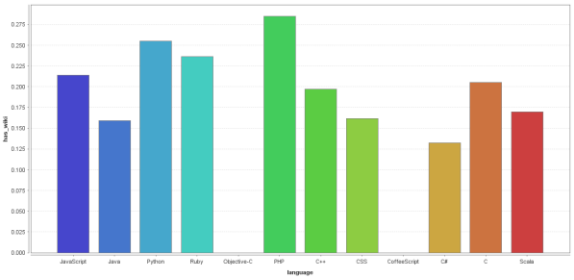
2. When learning one's first programming language, it is critical to have a well-developed and active community. Our data suggests that JavaScript dominates all other programming languages when it comes to the number of stargazers and forks. On average, JavaScript repositories have an estimated 1,250 forks per repository and a total stargazer count of 6,250. [Figure 3B and Figure3C]

3. It is helpful to have a language with working examples to follow from code snippets. Our results show that 6.25% of Ruby repositories are downloaded. This suggests that those repositories prove useful to a significant portion of programmers. [Figure3D]

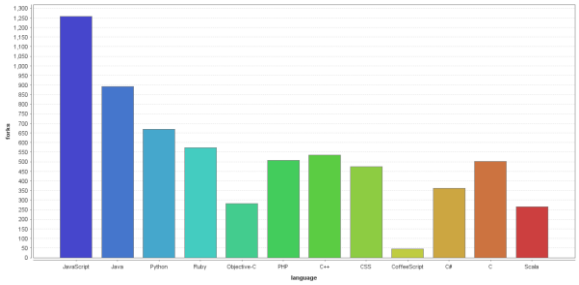
4. It is important to encounter as few obstacles as possible when debugging a problem. New Ruby programmers will encounter that only 3.5% of Ruby repositories have unresolved issues. [Figure3E]

5. Our results show that JavaScript repositories have more forks and fewer open issues than any other language in our data set. Conversely, Python repositories has fewer forks and more open issues than any other language in our data set. This suggests that JavaScript targets a more active and stable community. [Figure3F]

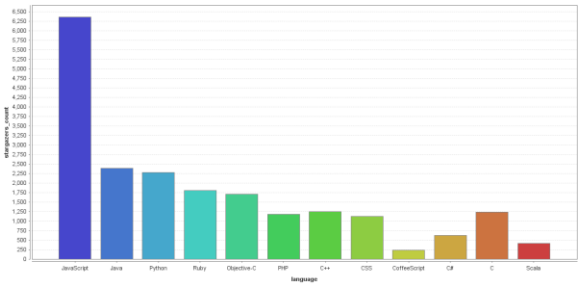
6. The JavaScript repositories have the greatest number of watchers amongst all other languages within our data set, reassuring that the JavaScript community is very active and well up-to-date. [Figure3G]



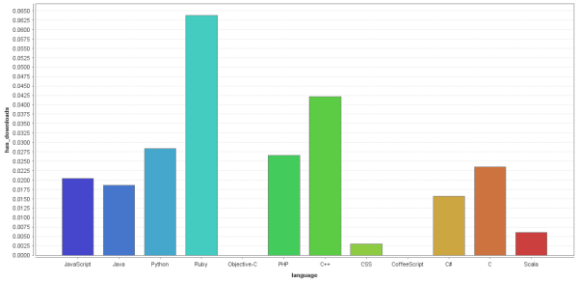
[Figure3A]: Language vs Wiki



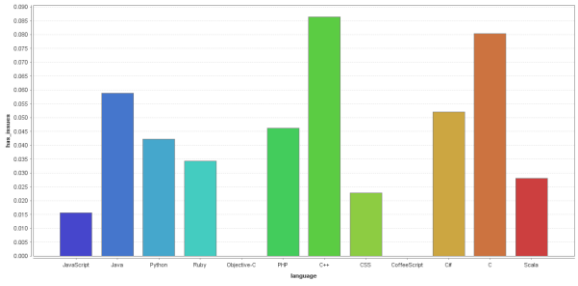
[Figure3B]: Language vs Forks



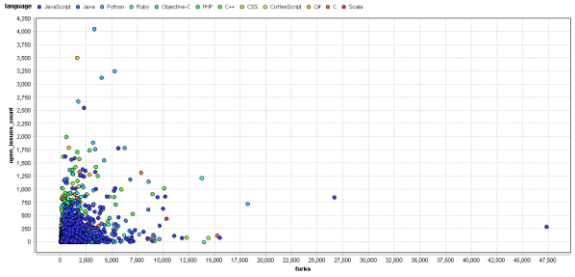
[Figure3C]: Language vs StarGazers



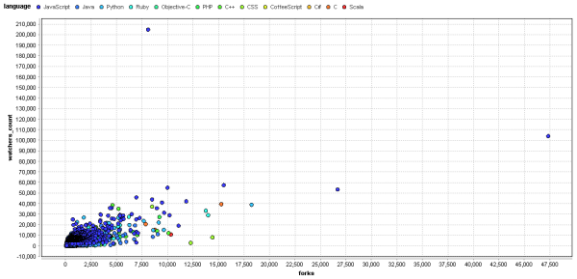
[Figure3D]: Language vs has_downloads



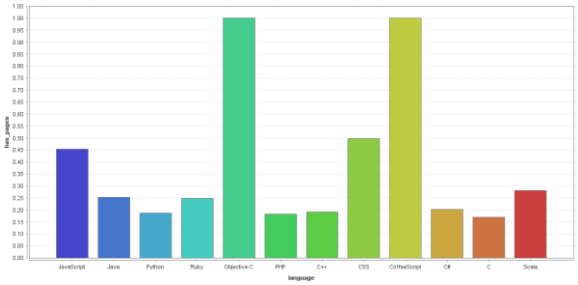
[Figure3E]: Language vs has_issues

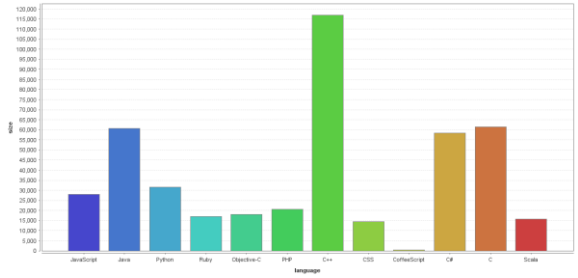


[Figure3F]: Forks vs open_issues



[Figure3G]: Forks vs Watchers





4. TESTING AND LOGGING

4.1 Apache JMeter

We used Apache JMeter to perform basic load and stress testing on our MySQL instance with different numbers of virtual users who request different services at the same time.

These tests were simply HTTP default requests, selected directly from the JMeter menu. All that was supplied was the remote information of the VM. Results are shown in [Figure4A] and [Figure4B]. Results indicate that our MySQL instance was able to sustain an average of 54

requests per minute. This helps explain the runtime of our application when the time came to insert little payloads into the MySQL database. Load test results on Google's provided hardware were consistent throughout multiple iterations.

We decided to not load test our MongoDB VM, as this is not queried by multiple clients. In fact, MongoDB was only used to provide a transition from raw data to a relational database. For this reason, there was no fruit in simulating load from multiple clients on a server that experiences load only from the application manager.

[Figure4A]: Apache JMeter Load Test graph results

Label	# Samples	Average	Min	Max	Std. Dev.
HTTP Request	1000	54	44	175	6.88

[Figure4B(i)]: Apache JMeter Load Test summary results

Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
0.00%	10.0/sec	1.65	1.86	168.0

[Figure4B(ii)]: Apache JMeter Load Test summary results

4.2 Simple Logging Facade for Java (SLF4J)

In this project, we decided to implement logging to an external file as a means of keeping track of program status. This was achieved through a Java logging API implemented using the simple façade pattern. The reason we chose SLF4J is because its implementation separates itself from its client API. All configurations are made within code or other `.properties` file, meaning the backend of the logging framework is built at runtime.

4.3 ScalaTest

Testing an application like this one is difficult for amateur developers like us. We did try our best at incorporating ScalaTests into our project. We wrote tests that utilized a payload modifying the JSON node values testing the functionality of our parser.

Generally, these tests are focused on ensuring the regex parser in `githubRegex.scala` is working as intended and correctly parsing JSON fields. The accuracy of this parser is critical due to our heavy reliance on the MongoDB data set to be as accurate as possible because this data is pulled by Rapid Miner and the MySQL database, allowing our the application to properly predict the attributes of a given GitHub repository.

5. CONCLUSION

In this paper, we describe a system which parses meta data from 10,000 repositories from 10 modern programming languages. From this data, we formulate 6 questions, explain the rationale behind them, and use gathered evidence to answer them to better assist new programmers who are trying to make a calculated decision on which language is a good first language to start learning based on our findings.