Team Name: Test Bayes
Team Members:
Adavya Bhalla (bhalla)
Aditya Jhamb (aditya97)
Avidant Bhagat (avidant)
Ethan Mayer (emayer4)
Steven Austin (saaustin)

Project Report

Motivation :

Software applications nowadays have become extremely complicated and difficult to test and verify. This can be attributed to the fact that the problems we are solving have become increasingly complex and developers cannot afford to make a product that is not technically sound or something that has not been exhaustively verified and tested. More often than not, testing takes a lot more time and effort than the actual solution to the problem. A tool that reduces the amount of time it takes to test a given solution/implementation to a problem, would be a major boost to the current development and testing process. Moreover, the number of tests nowadays are also substantial and when we run a test suite, we expect it to indicate results quickly. However, if a test suite is very large and is ordered in a way such that the first half or more passes but the others fail, then the developer loses out on precious time as the tests could have been ordered in a way particular order to indicate errors first and fail fast. In such a scenario, the developer is likely to save a lot of time and come up with a solution quicker since the developer was just working on the code. Besides, if the tests that failed had an intelligent ordering i.e. debugging the tests that failed would lead to a lot of errors being discovered in contrast to multiple failed tests indicating the same error, then the job of the developer is made even easier since the tests that failed initially can help the developer discover a lot of bugs in different areas of the project quicker.

Our approach:

The re-ordering of tests can be treated as a probabilistic problem where we take advantage of not only the independent probability of each test failing but also the correlated probabilities of that test conditioned on other tests. The probabilistic treatment ensures two things : 1) Tests that usually fail after an update are tested first and 2) The ordering is intelligent in the sense that these tests indicate possibly different bugs. This happens because the tests reorder themselves at run-time. Tests are re-ordered based on the outcome of the previous tests. This ensures that the order is

interesting (i.e. it is more likely to fail fast and indicate diverse errors). To achieve this goal, we can have the following high level approach :

1. We maintain a list of the independent probabilities of all tests failing on different versions of the same project. This list is constructed as we run the test suite on different versions of the implementation. For example, to construct the list of probabilities, we go through 100 different versions of the same large project and find out the probabilities of individual tests failing.
2. Next, we have to construct multiple data structures that store all the conditional probabilities of these tests. If there are n tests then we can implement this using two n x n matrices. The first matrix stores the probability of a test failing given that another test succeeded, while the second matrix stores the probability of a test failing, given that another test fails.
3. Finally, we first run a test that has the highest probability of failing regardless of other tests. If this test fails, then we iterate through the second matrix and find out which test has the highest probability of failing given that the first test failed. If it passes, then we do the same thing with the first matrix.

Interaction:

JUnit is an annotation based test system that can easily run large test suites by providing a convenient framework for developers to use. JUnit gives a test suite class that a developer can extend, and fill with test methods, that can then be easily run with a simple command (or simple GUI interaction). JUnit essentially is an API that a developer can use to wrap around there tests to make development quicker. However, JUnit works with a static ordering of tests, which does not allow for tests that are most likely to fail to actually fail first, causing for a lot of wasted time.

Our system will not have a GUI, as it will be completely command line based, so as to be used by all systems, instead of just those that have GUIs. Our design will interface with JUnit as an extension, meaning we will not modify JUnit in any way, and will simply act as a wrapper for JUnit, interfacing JUnit in a way similar to how a user would interface with JUnit. We will run tests one by one, in the order that we have determined mathematically.

The user will be able to interface our design through a very similar way to JUnit. They will be able to run all tests, run specific tests, and run a certain group of tests. Our design also creates a notion of a failure probability difference threshold (epsilon), and a moving average amount. When given a scenario where two tests have roughly the same failure probability (difference less than or equal to epsilon), we will run the shorter running test, instead of the longer one, so that the shorter one can fail first. This makes our program run not only the most likely to fail, but the shortest tests that are most likely

to fail. Our moving average is used to not incorporate test runs that are too old, as tests often change while the developer is in the process of debugging. This moving average can be tuned (increased or decreased) to incorporate more or less most recent test runs in the failure probability calculations. The user can also choose to not reorder tests, just for sanity purposes. All of these configurations will be modified through the pom configuration file in the maven project that our design will be a part of, since our design will use maven as its build system.

These changes will not affect the JUnit architecture, as we are only acting as an extension of JUnit, not actually modifying JUnit itself, leaving JUnit to be a tool that our design uses under the hood.

Implementation:

We have decided to make 2 implementations for the actual test ordering. It is not actually 2 completely different implementations, it is just the same code but ordered in a different way. The first implementation involves a greedy heuristic where we first run the test with the most probability of failing and then based on the outcome of the first test greedily decide which test to run next based on conditional probability. We would break ties on the greedy decisions by factoring in time taken to run the test, conditional probability, total probability, and a running average. The second implementation would be to just use everything from the start instead of waiting for ties. Evaluation will be critical in deciding which implementation will work better. Here are a few details about the data structures we plan on using:

1. **HashMap for total probability -** We want to store the total probability for each test passing in a HashMap where the hashCode is a code created by the name for each test and the value is the probability for each test passing. The value will be in a fraction object either created by us or an implementation we find online if it is able to match all the operations we would want to perform on it.
2. **HashMap for runtime -** We want to store the runtime for each test in a HashMap where the hashCode is a code created by the name for each test and the value is the runtime for the given test. The runtime will be stored as a double with seconds as the unit of time.
3. **Matrix for Conditional probability -** This is where we still have certain grey areas. We want to store the conditional probability in a matrix so that if we access the value for a test x by test y index we get the probability for test x passing given test y either passes or fails. There will be 2 of these matrices to facilitate faster calculations and decrease computing time. One way of doing this is to have a HashMap that itself has a HashMap as a value and then the second HashMap contains a fraction object as value which is the probability we seek. A

2D array implementation might also suffice. These are the 2 options we are currently looking into and right now we like the HashMap implementation better.

We've created two runners that can actually run the JUnit tests. One runner is an extension of the BlockJUnit4Runner class, which allows a user to use the @RunWith(OurRunner.class), and just run their test suites as if they were using JUnit natively. However, this implementation only allows for reordering within one class, meaning JUnit would spin up an instance of our runner for each test class that is ran, instead of one for all test classes. We had a second implementation that scans the testing directory for all test suites, using JUnit annotations, and manually runs the tests, ordering them irrelevant from the test class they reside in. The cons of this second implementation is that since JUnit does not easily support the reordering of methods from a culmination of test classes, it's difficult to use JUnit as much as a backend, causing us to do some JUnit test processing manually, making our code more obtrusive to JUnit that originally planned.

As far as our mathematical model goes, we are forced to use a fraction class that allows us to represent numerators and denominators separately, as it is mathematically impossible to perform our bayesian analysis without these. Our bayes calculations requires us to only add to the numerator or denominator, meaning that we cannot represent current running probabilities of test failure as numerators or denominators. As such, we implemented a class that can properly represent fractions.

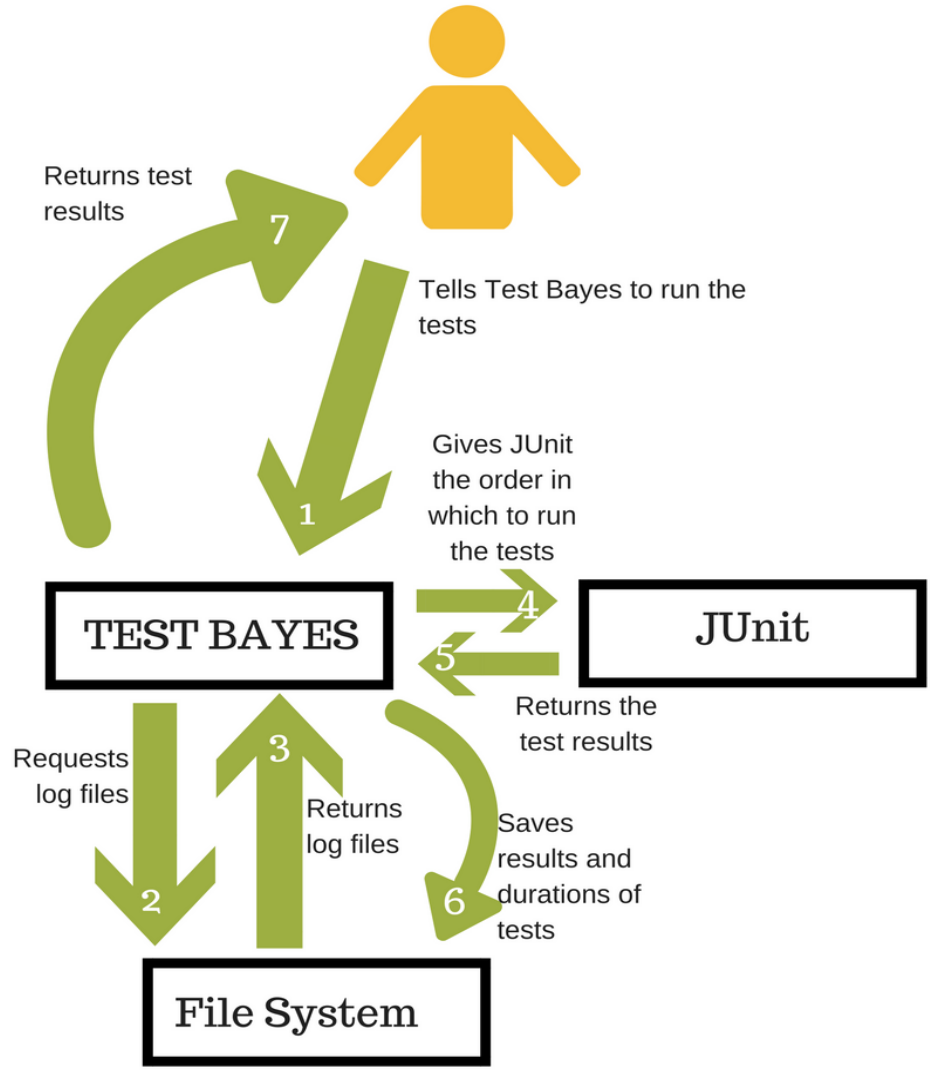
We decided to implement our log files in a JSON style format, with one file per test run, and one line per file. This makes reads and writes very fast, as we do not have to constantly open, readlines, edit, and close files, we can just open, readline, and close; file creation is also a very fast operation. Our data will be saved in a simple JSON format, denoting the test name, success or failure, and the time it took to actually run. This is the only information necessary for the bayes math to be calculated, since we are generating joint probability tables of tests based on their success/failures, and doing further epsilon reordering based on tests of similar failure, to reorder based on time at that point.

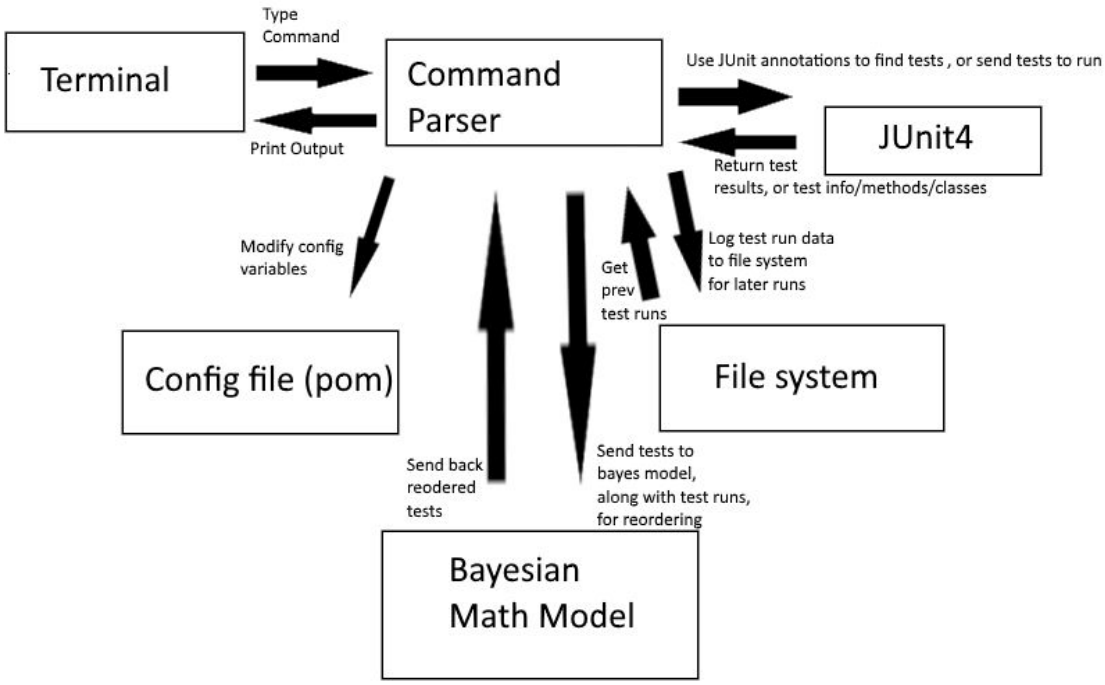
On the very first run, where there is no test data at all, each test will be run as if it were being run by JUnit, without our tool at all, meaning that each test will be deemed a 50/50 chance of passing or failing. Then, after that first run, the bayes probabilities will come into play, along with our moving average; tests will then be reordered.

How is it currently done:

There is currently no system in place that does this. Developers will actually run their tests sequentially, and wait for them to pass or fail. Testing frameworks, especially with JUnit, do not have support for dynamic ordering. This dynamic ordering would be the first of its kind. Read the “Similar Works” section of the user manual to gain more insight into why other projects out there do not quite address the problems we are tackling in our design.

Architecture:





These design represent the way our modules interact with both the user, and themselves. The first diagram depicts how the user and JUnit interact with our model, and the second model gives a more in depth understanding of how the “Test Bayes” module in our first diagram works.

In our second diagram, we have a command parser (main portion of the tool) that determines if the user wants to modify the test reordering config, or if the user wants to run tests. JUnit is used to parse annotations, along with run the actual JUnit tests. The file system is either sent new data to log, or is queried for existing data. The main brain of this tool is the Bayesian Math Model. This model takes in the previous run information, and returns the mathematically best ordering of the tests, in the order they should be run. The file structure returns a JSON model of the previous test runs. The math model takes in a list of method names, along with the JSON data needed for bayes calculations, and returns a list of methods to be run. The command going into the parser could be a list of test directories to run, or a flag that states all tests should be run. JUnit requires you to query it with a test suite class, requiring us to either have the user put all test classes in a dummy test suite using the @SuiteClass annotation, or to have our command parser generate the suite on the fly.

Evaluation:

To evaluate our code, we will be using two metrics to compare against two alternative ways of ordering tests. We will track both the time and number of tests run until the first failure, and compare our results for these figures to a test suite run in the default junit ordering, and a randomly ordered test suite. We will run these tests on multiple repositories (with commits that have a failed test build), along with general debugging and intermediary evaluation with the CSE 331 RatPoly code, and compare how our ordering fairs on each execution. We expect that test Bayes ordering will improve dramatically after a few runs of the test suite. An example data table we would generate is as follows:

Repository Names	Time taken to first failure			Number of tests to first failure		
	JUnit Ordering	Random Ordering	TestBayes Ordering	JUnit Ordering	Random Ordering	TestBayes Ordering

Main Demographics:

Since this is not an existing method of testing in industry, there is a huge user base to use this resource. These companies have codebases that can take days to run through all tests, just for one small package. Moreover, nearly every developer has had to run tests that take a significant amount of time, only for them to fail near the very end. This cases development to be slowed, making it very cumbersome for a developer to actually debug on larger projects.

Impact:

As a group, we feel as if the impact of this project is obvious to every developer who has ever run a test suite. Our tool will save developers a huge amount of time in running test suites with its ordering feature. Individual coders will feel the benefits in their personal time being saved, and the frustration of waiting for tests to complete reduced. Larger organizations could literally measure their monetary savings if they

compared the average wait time of a test suite before and after implementing our tool, correlating to higher productivity. Overall, this tool will dramatically cut down on idle time in the testing process.

Every developer has had to write test suites before. No matter the size, testing is a great benchmark that many developers use to gauge how correct their code is, sometimes even going so far as to writing the tests first, before even implementing their code. For this reason, it's not hard to see that every developer could benefit from this tool, since they would be able to rapidly determine the failure points of their code, instead of waiting for long test suites to run. The scenario that can be imagined is a developer in industry who has to work with large test suites that can take many hours to run. Our group has had several members work at internships where test suites could take anywhere from 10 minutes, to 30 minutes, to overnight, just to finish running a small portion of the tests that the entire codebase has, only to find that some random test failed near the end. The use case for our code would be not only generally useful to the common developer, but could be crucial to many industry codebases, speeding up the development process significantly.

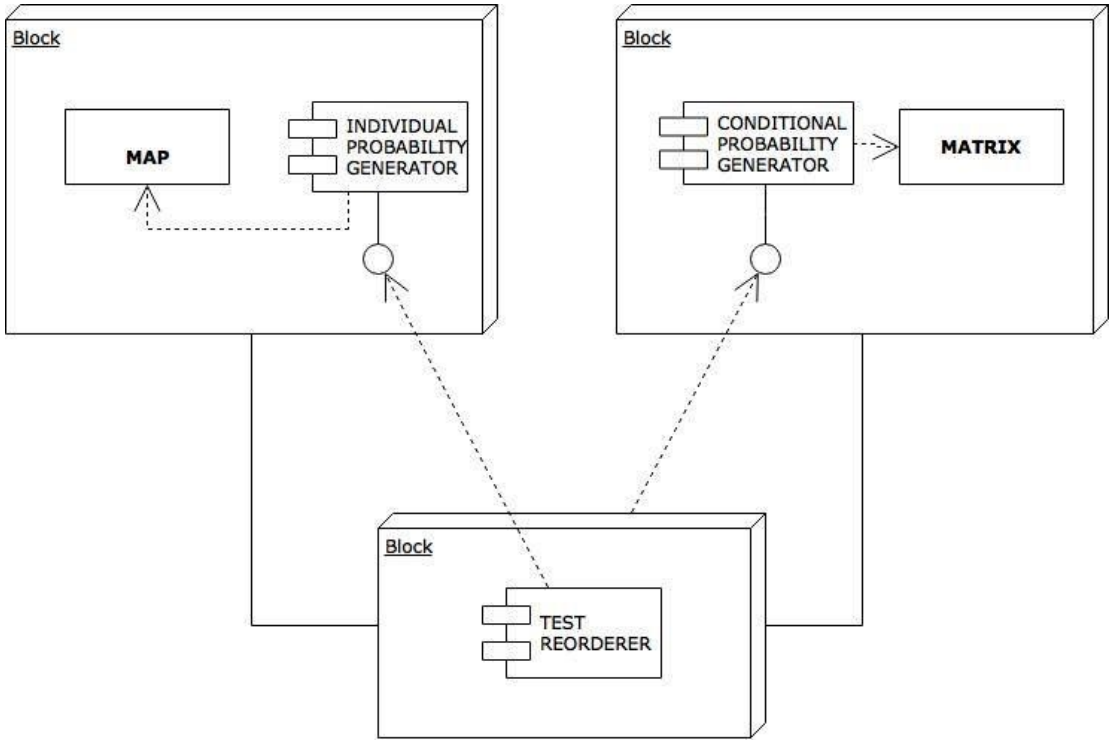
Challenges and Risks:

It is difficult to mathematically verify whether this approach will be successful or not. It sounds correct but may not be technically sound. The biggest risk is that the overhead from running this extension tool might take longer because of the initial training and construction of the two matrices and the list. Also, it might be challenging to find such a large open-source project that has so many previous versions.

Since there could potentially be a large amount of data from many test runs, we could run into the issue where the file/data processing required of the tool could outweigh the time the developer saves from test reordering. We will counteract this by incorporating a moving average with our Bayes model, so as to not consider extraordinarily large amounts of data.

Furthermore, if the data were too large, it could be a memory load on the user, meaning our approach will consist of using as little memory as possible in our file structure, only saving what is absolutely necessary.

Here is a small diagram that illustrates this procedure :



Cost:

In an ideal scenario, the costs of using this tool would be minimal. We are trying to make our solution take close to no time to reorder the tests so that this time is not noticeable by the users, and also is significantly lower than the time saved by the re-ordered tests. To augment this, we would also be working with a moving average which will help reduce the amount of time we take to process the data.

Checks for Success:

The most important factor to check for success would be the time to failure. The purpose of our project is to essentially reduce this as much as possible, and this makes it the most important factor to test success. To do this check, we would want to compare the time taken by our test order to the time taken by the standard JUnit ordering, and also to the time taken by a random ordering of the tests. By doing so, we will not only be able to guarantee that our ordering saves the developer time as compared to the JUnit ordering, but also compared to a random ordering, which is often considered a better option.

Building our Project:
Building our project is simple. All dependencies of our project are built into the maven configuration, and will be imported on build, or are existent in the repository already. To run a build and run the tests, enter:

```
git clone https://github.com/avidant/test-bayes
cd test-bayes
mvn clean verify
```

This will output the success or failure of the build, and success or failure of the tests we've written for our classes.

Verifying Builds using Travis CI:
Describe how to use the Travis CI tool to verify passing and failing builds on the repository. Describe where to find past builds and how to see whether they passed or not.

When a pull request is made, Travis CI is run automatically, and upon viewing the pull requests status online, a green check mark will pop up next to the Travis CI section indicating that the build passed, an X denoting it failed, or a different symbol denoting the test partially passed. Past builds can be found in our repository at https://travis-ci.org/avidant/test-bayes/builds/. To view individual past pull requests instead of current states of branches, go to https://travis-ci.org/avidant/test-bayes/pull_requests. You'll see that there are several passing and failing builds.

This can also be run from the command line with the "`mvn clean verify`" command from the terminal, as this is the command that Travis CI runs in the background.

Using Parameters to improve results:
We have identified some places in our code where we are using made up numbers like the starting probability for each test, the importance we have to give to each part of our mathematical model. We would like these numbers to mean something more than just numbers that seem correct. Therefore, we plan on using optimization methods to optimize these parameters for maximum performance. Once we have the initial implementation complete, we will use the initial results to optimize these parameters for a better overall performance by our tool. We can achieve this by either using ML

libraries or just small custom methods that minimize the time taken to failure based on different values of the parameter.

Schedule:
✔ Week 3: Learn about extending JUnit, and hammer out details about file format, and the mathematical model.
✔ Week 4: Begin development of base functionality, meaning the ability to reorder tests randomly, or in some given order.
✔ Week 5: Incorporate our mathematical model into the code.
Week 6: Test to see how the mathematical model performs on code.
Week 7: Measure benefit of our tool, as opposed to random ordering.
Week 8: Clean up code, add any small features or changes that are necessary.
Week 9: Test on larger code bases.
Week 10: Polish codebase, turn in

Resources Used:

http://www.baeldung.com/junit-4-custom-runners

http://junit.sourceforge.net/javadoc/org/junit/runners/BlockJUnit4ClassRunner.html

https://github.com/eugenp/tutorials/blob/776a01429eb6b7ec0d1153f88097e2cc9915e599/testing-modules/testing/src/test/java/com/baeldung/junit/BlockingTestRunner.java

https://stackoverflow.com/questions/20976101/how-to-get-a-collection-of-tests-in-a-junit-4-test-suite?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

https://junit.org/junit4/javadoc/4.12/org/junit/runner/manipulation/Sortable.html

https://ieeexplore.ieee.org/abstract/document/1007961/

https://ieeexplore.ieee.org/abstract/document/5401169/

https://ieeexplore.ieee.org/abstract/document/1510150/