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

Project Report

Motivation:

More often than not, testing takes more time and effort than the actual design and implementation phase[8]. A tool that reduces the amount of time it takes to test a program would be a major boost to the current development process. Developers often will run entire unordered, or manually ordered, test suites, which eat up testing time with inefficiencies. If the developer could order tests by failure probability, this would allow a developer to recognize that a test has failed early on, and would not have to wait for many passing tests that often don't have much to do with the bug at hand. Our tool aims to run tests such that the test suite fails early and fails fast.

Our approach:

Our approach is to identify tests with a higher probability of failure than others and run them first. However, this is very difficult, if not impossible, as the user's code would have to be scanned to determine all bugs that exist, and how those bugs affect the tests being run. This is extremely impractical; we propose a probabilistic model that determines which tests are most likely to fail given prior test run data. Our tool analyzes log files and builds a mathematical model to run the first test most likely to fail, then compute the next test most likely to fail, given that the first test passed/failed, and run it. Generalizing this, the nth test ran will be the most likely to fail, given the (n-1)th test outcome. This process repeats until all tests have been run. To achieve this goal, we have the following high-level approach:

1. Our Bayesian model constructs a list of the independent probabilities of all tests failing on different versions of the same project. It takes the past test runs as

inputs and constructs these probabilities as the number of successes divided by the number of runs, independent of any other test passing or failing. These independent probabilities are then used as input for our overall probability calculations.

2. Next, if there are n tests, the model constructs 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. The past test runs are used as input, which is then used to generate these two n x n matrices, by counting the number of instances of a test passing/failing given that some other test has passed/failed, and doing this for all tests. These matrices are then used to output a test ordering.
3. Finally, the test runner first runs the test that has the highest probability of failing, regardless of other tests. If all tests have the same probability, the lowest execution time test is chosen based on previous results. If this test fails, the model will iterate through the failure matrix and find out which test has the highest probability of failing given that the first test failed. It will otherwise do the same thing with the success matrix, finding the test most likely to fail. The two n x n probability matrices, along with the individual probability list, is used as input for the model to iterate through so it can find the next test to run, based on the outcome of the current test that was just run. As an optimization, if given the choice between several methods that have a failure probability of within epsilon of each other (set by the user), then the shortest test is chosen.

Architecture:

Figure 1: User Interaction with Test Bayes

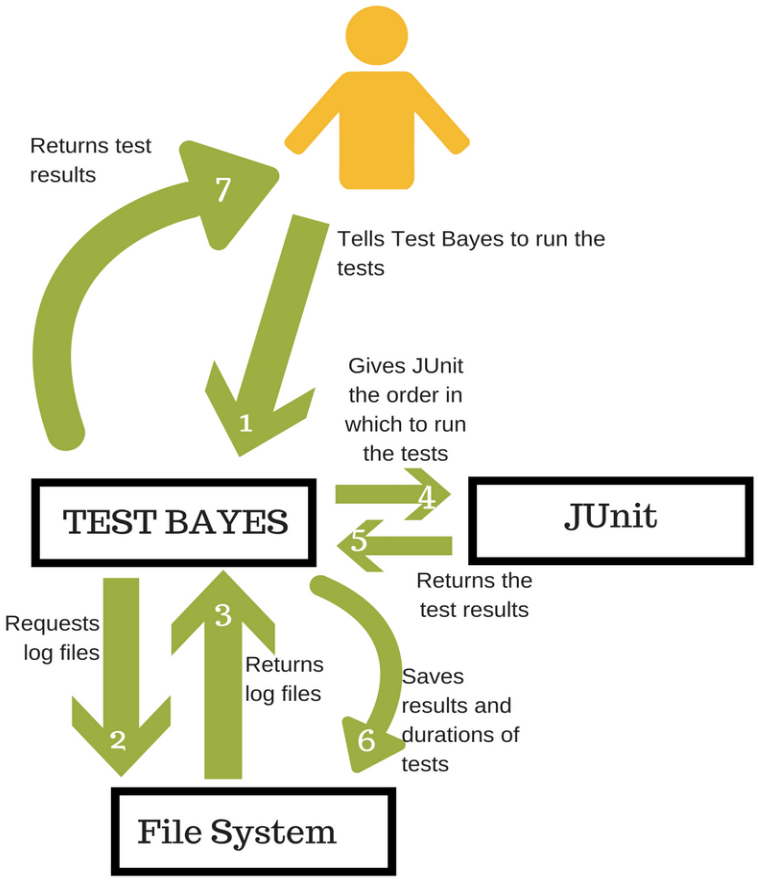
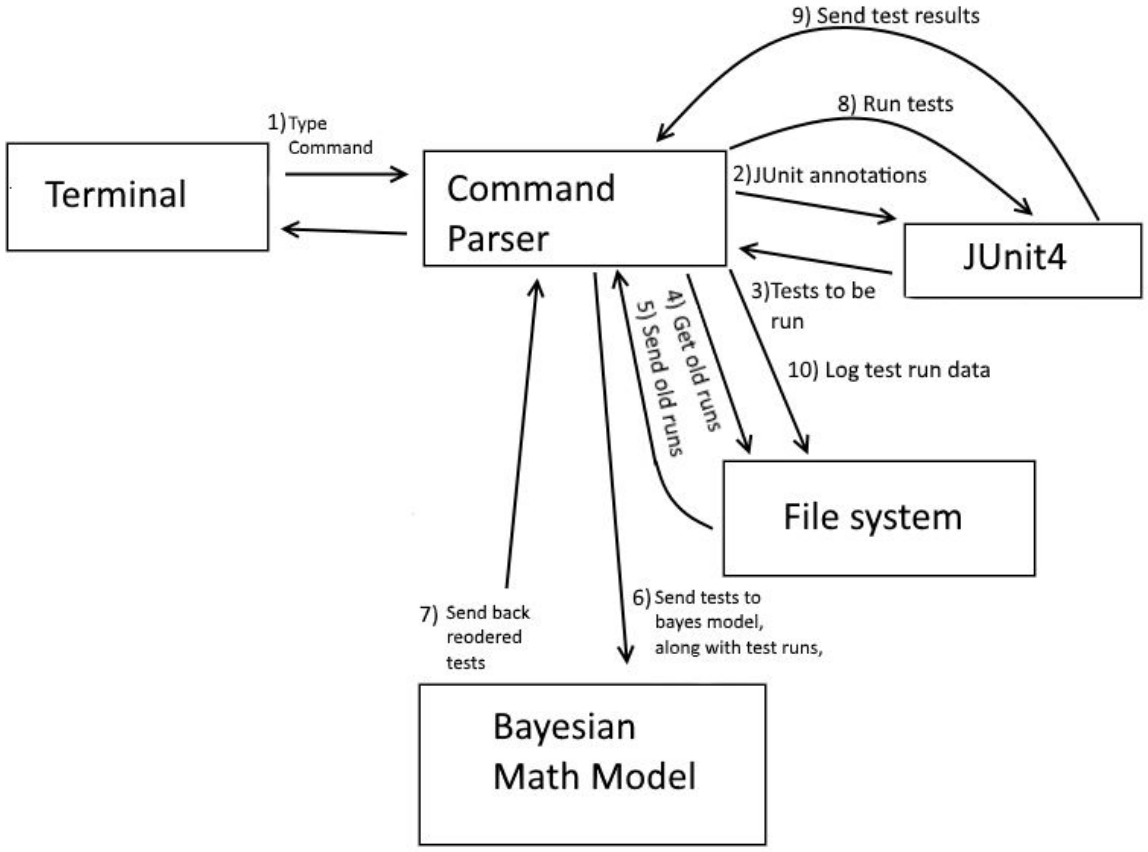


Figure 2: Interaction of the components within Test Bayess



These designs represent the way our modules interact with both the user and themselves. Figure 1 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 Figure 1 works.

In Figure 2, there are seven steps overall:

1. User inputs a command to the parser including the test directory to be run (interfaceable part of tool)
2. Parser sends JUnit before class, test, and after class annotations to JUnit4
3. JUnit4 sends back the tests to be run
4. The parser then requests old test run data for probability calculations
5. That old data is sent back to the parser
6. The Bayesian math model is sent the old run data, with the tests to run
7. The tests are re-ordered based on the probability of failure and are sent to parses

8. These tests are then run in the specified order by JUnit
9. The tests results are returned to the parser
10. The user is given the output from the tests, the current test session is logged.

Implementation:

Our implementation involves first running the test with the most probability of failing and then based on the outcome of the first test, decides which test to run next based on conditional probability. The model breaks ties on the decisions by factoring in time the test took to either pass or fail on the last run, conditional probability, total probability, and a running average. As a note, the term running average, and moving average are used interchangeably, and mean the same thing; this moving/running average is simply the last n test runs that the user specified they want as input for the Bayesian calculations our tool performs, where n is the number of previous test runs the user wants the tool to incorporate into Bayes calculations. Here are a few details about our design decisions:

1. **Total probability -** The model takes in all the executions and their results from the log files that fileio creates using test history, then it goes through each execution to calculate total probability and store it in a data structure. The value will be stored as a fraction, that stores the numerator and denominator as integers.
2. **Runtime -** The runner stores the runtime for each test in a data structure using the log files created by fileio. The runtime will be stored as a double with milliseconds as the unit of time and the sign (negative or positive) will describe whether the test passed or failed.
3. **Matrix for Conditional probability -** The model stores the conditional probability in a matrix so that if it accesses the value for a test x by test y index it will get the probability for test x passing given test y either passes or fails. There will be two matrices to facilitate faster calculations and decrease computing time.

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(IndividualClassRunner.class), and simply 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, instead of one for all test classes. We have 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. This second implementation

is run from the command line, instead of through JUnit. The drawback of the second implementation causes us to go under the hood with JUnit and run the tests manually.

As far as our mathematical model goes, it must 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. This is because Bayes Theorem requires that we add or subtract to one or both of the numerator/denominator. Our Bayes calculations require us to only add to the numerator or denominator, meaning that they cannot represent current running probabilities of test failure as only numerators or denominators. As such, we implemented a class that can properly represent fractions.

We store our log files in a simple format, with one file per test run, and one line per file containing the test names, and the time each test took to actually run. These times are negative if the test fails. This makes reads and writes very fast, as fileio does not have to constantly open, read lines, edit, and close files; it can just open, read one line, and close (an extremely fast sequence to execute). This is the only information necessary for the Bayes math to be calculated, since it is generating joint probability tables of tests based on their success/failures, and doing further epsilon reordering based on tests of similar failure time.

On the very first run, when there is no test data at all, each test will be run as if it were being run by JUnit alone, meaning that each test will be deemed a 50/50 chance of passing or failing. After that first run, the Bayes probabilities will come into play, along with our moving average, causing the tests to be reordered.

How test reordering is currently done:

Developers usually run their tests sequentially and wait for them to pass or fail or, manually reorder them such that the tests they think will fail are executed first. But their instincts are not mathematically based, and may not yield good results. Testing frameworks, especially with JUnit, do have support for dynamic ordering. This dynamic ordering would be the first of its kind. Read the “Similar Works” section of this report to gain more insight into why other projects out there do not quite address the problems tackled by our design.

Similar work:

Most work related to our project are frameworks that allow for different types of testing, rather than test reordering. One framework that is popular is the Spring Framework[7], which is designed around testing a program by giving control flow back the to framework (often called Inversion of Control, or IoC). However, this is more concerned with the type of testing, rather than test reordering. After searching on github, and google scholar, the only thing similar to our project was a research paper conducted on the time benefits of reordering test based on test failure rates. The paper stated that “the effectiveness of prioritization techniques… on JUnit has not been investigated”[6]. However, the paper stated that “numerous empirical studies have shown that prioritization can improve a test suite's rate of fault detection”[6], showing that our project has promise in being a useful tool for people to use. While previous approaches to this problem try to address it through manual reordering, we are using automatic background calculations of failure rates to mathematically work out a provably good strategy for test suite executions.

Evaluation:

In our evaluation, we need to evaluate 2 hypotheses:

1. The efficiency of our tool
2. The overhead of our tool

To evaluate our tool, we use two metrics to compare against two alternative ways of ordering tests. We 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, and compare how our ordering fares on each execution. We expect that test Bayes ordering will improve dramatically after a few runs of the test suite. Our evaluation table we would generate is as follows:

Repository Names	Number of runs for which data was collected	Time taken to first failure			Number of tests to first failure		
		JUnit Ordering	Random Ordering	TestBayes Ordering	JUnit Ordering	Random Ordering	TestBayes Ordering
331 Code	3	15ms	3.7ms	0.95ms	15	13	1
Test Bayes (Dummy Tests)	100	0.164ms	0.122ms	0.026ms	35	28	1

Efficiency:

As our initial data shows, in just 3 runs, both our metrics were clearly successful. We recognize that this is only a small data set and are thus not using this data to make significant decisions. We do feel that this signifies that we are moving in the right direction.

Overhead:

As we notice with our Dummy Tests, despite 100 previous runs, and each run having over 100 tests, the overhead is so small that the time to first failure is still very early. This makes us believe that our decision to use a moving average was a smart one and helps us ignore old logs.

Instructions to reproduce (random ordering may not be exactly reproducible):

331 Code Instructions:

1. clone https://github.com/emayer2/RationalNumbers.git via IntelliJ (or some other IDE)
2. add test-bayes.jar to classpath, along with JUnit
3. build project
4. run the RatPolyTest.java file on src/hw4/test.
5. continue running RatPolyTest.java, and notice how the time and tests until the first test failure decreases to a consistent amount, and flattens out.
6. change the @RunWith value to contain IndividualClassRandomRunner.class instead of IndividualClassRunner.class
7. run RatPolyTest.java again, and notice how the average time and tests to failure is much higher than the first runner
8. remove the @RunWith statement completely to see how JUnit runs the test natively

Test Bayes (Dummy Tests) Code Instructions:

1. Clone https://github.com/avidant/test-bayes
2. Checkout and pull the branch evaluation
3. Run eval-100-runs.sh. This will give you the result of running with TestBayes with no previous data, and then with approximately 100 runs of data
4. Change the @RunWith value in Tests.java to contain IndividualClassRandomRunner.class instead of IndividualClassRunner.class
5. Run the command *mvn test -Dtest=Tests.java*. This will give you the result of running in random order
6. remove the @RunWith statement completely to see how JUnit runs the test natively

Main Demographics:

Since this is not an existing method of testing in industry, there is a huge user base to use this resource. Companies have codebases that can take days to run through all tests even for a 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 causes development to be slowed, making it very cumbersome for a developer to actually debug large projects[8].

Impact:

As a group, we feel the impact of this project is obvious to every developer who has ever run a large test suite. Our tool will save developers time in running test suites with its ordering feature. Individually, coders will save a lot of their time and will have reduced frustration due to late test fails. Larger organizations could literally measure their monetary savings if they compared the average time until a test suite failure occurs, before and after implementing our tool, correlating to higher productivity. Since a developer would have to on average wait less time until a failure occurs, they would be able to identify, test, and fix bugs quicker. 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 they even going so far as to writing the tests before implementing their code, as we have done occasionally in our studies. 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. Our group has had several members work at internships where test suites could take anywhere from ten to thirty minutes, even overnight, just to finish running! Of course, failures near the end of the suite were common, and wasted a lot of time. 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:

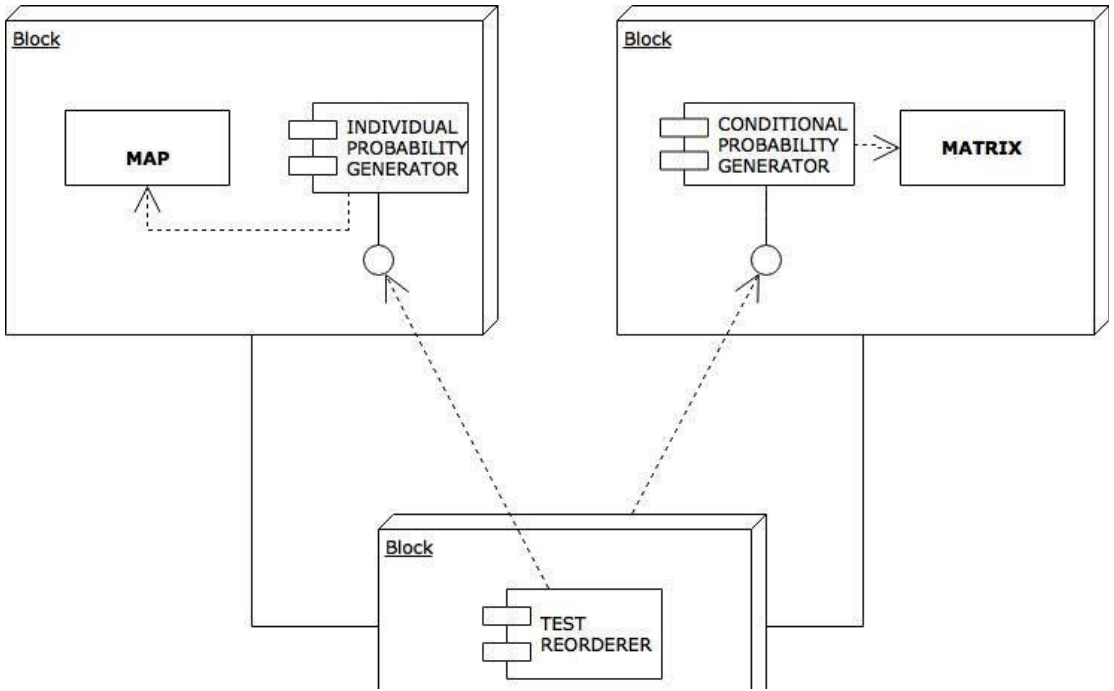
It is difficult to mathematically verify whether this approach will be successful or not. It sounds correct but may not be technically sound. The biggest risk is that the overhead from running this extension tool might take longer because of the initial training and construction of the two matrices and 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. As a result, 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 :

Figure 3:



Cost:

In an ideal scenario, the costs of using this tool would be minimal. We are trying to make our solution take miniscule time to reorder, which should be much lower than the time saved by the re-ordered tests. To augment this, our model works with a moving average to reduce the amount of time fileio takes to process the data.

Checks for Success:

The most important factor to check for success would be the time to failure, and tests that ran until first test 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.

Verifying Builds using Travis CI:

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 internally..

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. Regardless, the user will pick these parameters via command line arguments to the tests being run.

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. Measured overhead and efficiency of our tool, giving encouraging results.
✔ Week 8: Clean up code, fix bugs, add any small features or changes that are necessary.
Week 9: Test on larger code bases.
Week 10: Polish codebase, turn in

Resources Used:

[1]http://www.baeldung.com/junit-4-custom-runners
[2]http://junit.sourceforge.net/javadoc/org/junit/runners/BlockJUnit4ClassRunner.html
[3]https://github.com/eugenp/tutorials/blob/776a01429eb6b7ec0d1153f88097e2cc9915e599/testing-modules/testing/src/test/java/com/baeldung/junit/BlockingTestRunner.java

[4]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
[5]https://junit.org/junit4/javadoc/4.12/org/junit/runner/manipulation/Sortable.html
[6]https://dl.acm.org/citation.cfm?id=1116157
[7]http://spring.io/
[8]http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.370.9611&rep=rep1&type=pdf
[9]https://ieeexplore.ieee.org/abstract/document/1007961/
[10]https://ieeexplore.ieee.org/abstract/document/5401169/
[11]https://ieeexplore.ieee.org/abstract/document/1510150/
[12]https://junit.org/junit4/javadoc/4.12/org/junit/experimental/max/MaxCore.html
[13]http://automation-webdriver-testng.blogspot.com/2012/07/how-to-call-test-plans-dynamically.html
[14]http://beust.com/weblog/2008/03/29/test-method-priorities-in-testng/
[15]https://arxiv.org/pdf/1801.05917.pdf
[16]https://lib.dr.iastate.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=6493&context=etd