Team Name: Test Bayes
Team Members:

Adavya Bhalla (bhalla)
Aditya Jhamb (aditya97)
Avidant Bhagat (avidant)
Ethan Mayer (emayer4)
Steven Austin (saaustin)

Test Bayes User Manual

Reason For Use: Test Bayes is a tool designed to speed up the testing process of development significantly. Testing is a static process, that can cause developers to waste a lot of time waiting for certain tests to fail. However, with Test Bayes, tests that are likely to fail on the next run are run with higher priority, meaning they will fail first, instead of later, saving the developer a lot of time testing. This speeds up the development process significantly. Test Bayes is built off of JUnit4, a very common Java test framework, making it compatible with many projects. Test Bayes uses a Bayes oriented mathematical model to determine probabilities of test failures, and orders them accordingly.

Use: Test Bayes utilizes the maven build system to get its information about tests. By specifying the test directories in the pom.xml file, Test Bayes can find all of the tests that are to be run. By including our project in the same directory as the pom.xml, Test Bayes can function properly. Since this project is a build-off of JUnit4, JUnit4 must also be installed in the project. Test Bayes is interfaced through a terminal. There are several commands that can be used to run tests, as follows:

```
tb                              : Runs all tests in pom file.
   -h                           : Display command help info.
   -d testDir1 testDir2 …       : Runs all tests in given
                                  directories instead of all
                                  tests.
   -u                           : Will not reorder tests to be
run.
   --avg=numTestRuns            : Set the number of test runs
                                  to consider in the calculations.
   --eps=doubleValue            : Set the failure probability
                                  difference between tests.
   -c                           : Clear all test data.
   -n                           : Treat new tests as if they are
                                  not similar in fail rate to
```

```
other
                                    test runs of other methods
```

After running Test Bayes, test results will be output to the console (terminal) in the order they were run, or if the command was a parameter modification, no output will be produced. Test Bayes test runs are versioned, in that the output from test runs will be stored in files in the directory the tool is in, making the reordering exist across different machines running the code.

Customizable inputs-

The user will be able to customize the following things according to need:

1. **New tests** - The user will decide whether the tests that have no previous runs will be run first, or just treated like tests with the most average stats out of all. This means that a new test will not be given a 50/50 chance of passing or failing, but instead will be treated as if it were similar to other tests, and will be given the average failure rate of all other tests about to be run. This would be specified with the `-n` flag in the command entered.
2. **Moving average -** The user decides what number of runs they want to include in the calculation of the moving average. While this is not the only parameter our code uses to compute the order of tests, it is certainly a big part of it, and we want to be able to adapt to the user's need as much as possible. This can be set in the pom.xml file through the `--avg` flag command, as described above. The default value for this moving average value is 20 test runs.
3. **Test Similarity -** The user can finetine an epsilon parameter, which is a decimal value from 0.0 to 1.0, which determines how similar two tests' failure rates can be so as to order by time instead of failure rate. This can be set in the pom.xml file through the `--eps` flag command, as described above. For example, if epsilon was set to 0.01, and two tests had failure rates of 0.94 and 0.95, but the second test was double as long, the first (shorter) test would be run. The default value for epsilon is 0.03.

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

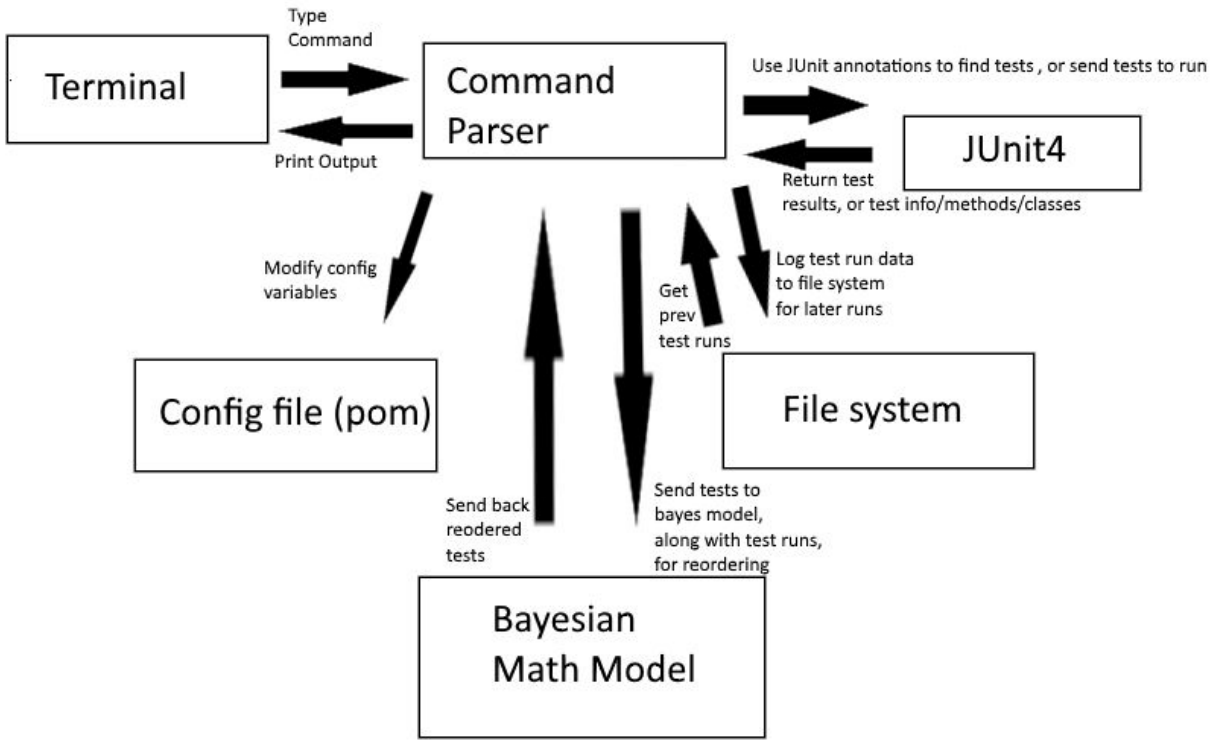
interesting (i.e. it is more likely to fail fast and indicate diverse errors). To achieve this goal, we can have the following high level approach :

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.

Feedback from last week: There was concern about the initial run of each of these tests. 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.

There was also a concern in the general use case of our tool. 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.

Furthermore, there was concern about the control flow of our design. To illustrate this, the diagram below shows how the user would interact with our design:



This diagram describes the main portion of our system. 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.

Similar work: Most work related to our project are frameworks that allow for didn't types of testing, rather than the actual act of reordering tests. 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 (as previously stated), more concerned with the type of testing, rather than test reordering itself. 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.

Further reading -

Much of this work was based off of tutorials given by links 1-5. If the user wants to further understand the workings behind what we are doing a few good places to look at would be:

(1)https://junit.org/junit4/javadoc/4.12/org/junit/experimental/max/MaxCore.html
(2)http://automation-webdriver-testng.blogspot.com/2012/07/how-to-call-test-plans-dynamically.html
(3)http://beust.com/weblog/2008/03/29/test-method-priorities-in-testng/
(4)https://arxiv.org/pdf/1801.05917.pdf
(5)https://lib.dr.iastate.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=6493&context=etd
(6)https://dl.acm.org/citation.cfm?id=1116157
(7)http://spring.io/