5 Testing

Our overall testing philosophy in this project is to test whenever human error is possible and test the outcome of several machine learning algorithms. Most of our testing for the dataset creation tool will come from unit testing and integration testing. Unit testing is critical especially as we are dealing with external services that may be out of our control. This idea is also elevated to the project level – machine learning outputs are never certain, so we much be careful about the data we feed and the inputs of the system to ensure that even if the algorithm doesn't give us what we expect, it's still performing accurately. Basically, we need to make sure all data is clean and responsibly made so the factors out of our control later will be made in good faith. This type of testing is specific to our project as it involves every aspect of the machine learning pipeline, unlike other machine learning projects where data is already available.

5.1 Unit Testing

Below is a table of our current (and possible future) units that will undergo unit testing. They roughly correlate to the methods being used in source code for ease of unit testing.

Name	Description	Test Plan
User Input	Grabs user input file and asks for parameters for label file creation.	Give sample input and ensure file location exists and writes correct file output. Edge cases include empty strings and illegal characters that would interfere with label file creation.
Abstract Syntax Tree Creation	Creates an abstract syntax tree from source code.	Out of our control as we use a tool called JavaParser for this, however, we can test it with edge cases (empty .java files).
AST Classname Gatherer	Given an AST, we traverse through the AST nodes and collect the line number and type of feature.	Create Java files that do not follow convention but can technically compile. Test parameters (low vs high level classnames) are being correctly sorted.
Label File Creation	Given a classname TreeMap, print keys and values to label file.	Test what happens when label file cannot be created, or dictionary is empty. Edge case testing.

We will use JUnit tests to ensure that these unit tests work. Our project is currently using the Maven framework which easily incorporates this testing framework.

5.2 Interface Testing

Name	Description	Test Plan
AST Internals	Units related to internal AST creation and traversal.	These will be testing with pre-defined user input and sample files. This will test the accuracy of the main driver functionality – label creation.
User Interaction	Units related to user interaction and I/O.	Ensure I/O is properly accepted and does not interfere with other internals. Data scrubbing & cleaning. Dealing with file location and type.

5.3 Integration Testing

Our most critical integration was getting AST into our program. As stated above, we will test edge cases for our AST application (javaParser). These edge cases can simply be tested with Junit stubbing. We will have consistency tests that will run against a select few files that make sure we aren't getting any variability with the AST output.

5.4 System Testing

Our system level testing strategy has been TDD. There should not be any lines that should not have a test case. This will be apparent if a line of code has been deleted or altered and an error a test case will fail. We will be using Junit and Junit libraries.

User input testing

- We will test if a valid input has been given
- We will have test for in there is invalid or no input given

AST functionality testing

- There will be junit stubs testing that methods have been called
- We will be testing that the methods have been called with expected parameters

Consistency testing

- As afore mentioned, we will be testing files and there expected output
- These tests ensure that our AST integration is functioning as expected.

Output testing

- We will test output for valid and invalid inputs
- We will test output format

- We will test output destination

5.5 Regression Testing

We will run our unit tests each time a new feature or source code is added. We cannot have the main functionality of the label file creation application break, that being accepting user input and outputting a label file. Everything else added, whether it be a new feature, minor tweaks, or improvements, must make sure all unit tests pass. We will continue using JUnit for these tests.

5.6 Acceptance Testing

This is the most important part of our testing plan. We must ensure that label files are accurate and correlate to source files accurately. If this part is inaccurate, the machine learning algorithm will have inaccurate data and have a high bias. We will not move on until all our requirements are met. The requirements will be set out by our client, Arushi Sharma, who will give the "okay." One requirement requires us to have a rate of 90% accuracy with label files. Accuracy, in this case, means not missing any features of code.

5.7 Security Testing (if applicable)

Not applicable – no live application.

5.8 Results

We have two main testing types involved in our project, JUnit and Acceptance Testing. Below is a table elaborating on our expected results and the requirements that will be verified by these test types.

Testing Type	**Expected Results**	**Requirements Verified**
Junit Testing	All JUnit tests will result in pass or fail. It will be expected that all JUnit tests pass. In any case where JUnit tests fail we will have to resolve the issue before pushing the code to our live version.	JUnit Tests ensure user use requirements are being met such as file uploading, as well as our AST integration. User requirements and AST functionality must be met at all times.
Acceptance Testing	We require and expect our label files to be accurate. Accuracy in this case will be defined as 90% accuracy rate with our label files. This is to be verified by our advisor.	Label file accuracy is critical to the success of our project. These labels will be needed for our machine learning algorithms later on.