COSC 6386 - PROGRAM ANALYSIS AND TESTING

ASSESSMENT OF TESTING IN SCIPY

SINDHU GUMMADI- 2146080

SAI SRIJA LAKKAKULA- 2138809

RAHUL TANDON- 2049205

GitHub link - https://github.com/sindhugummadi9/PAT-Project-Scipy

https://github.com/sindhugummadi9/PAT-Project-Scipy

Introduction to SciPy

In this project our team worked on building, testing, and analyzing large software projects and provided our assessment of testing done for the project. The project revolves around a stable release of the SciPy library in Python (https://github.com/scipy/scipy). SciPy library has gone over 64 releases till now and our team worked on the latest release of SciPy (1.8.0).

SciPy is a free open-source Python library mainly used for scientific and technical computing. It contains modules for optimization, linear algebra, integration, interpolation, FFT, signal and image processing, and other tasks common in science and engineering. This library is written in Python, Fortran, C, C++ with an initial release around 2001 and most stable release in 2022. SciPy is a tool in the Data Science Tools category of a tech stack, with 9.5k GitHub stars and 4.2k GitHub forks. However, there are still certain alternatives to SciPy which includes NumPy, Pandas, MATLAB, scikit-learn, TensorFlow to name a few.

In this project, we articulated our views, about the state of testing, adequacy of tests, appropriateness of testing process, keeping in mind the viewpoint of developer and of the tester to provide an acceptable level of confidence in the library.

This report includes description and analysis on following details in the SciPy library–

1. The line, branch, and function coverage of the projects at file level and on its entirety.
2. The number for test files and the number of assert statements in each file.
3. The number and location of the debug and assert statements in production files.
4. Lastly, the use of PyDriller to report when the test files have been added to the project, how often they are modified, and how many people were involved

https://github.com/sindhugummadi9/PAT-Project-Scipy

Coverage in SciPy

Code coverage essentially shows the number of lines, functions, and branches covered by the unit test cases in a project. Running a code coverage report helps us identify what code is not being used to help us write more unit tests. Code coverage can also show which branches in conditional logic are not being covered. Code coverage is calculated for a file based on the unit tests that successfully passed. Below pie chart gives an overview of the code coverage on the SciPy project with the most stable version.

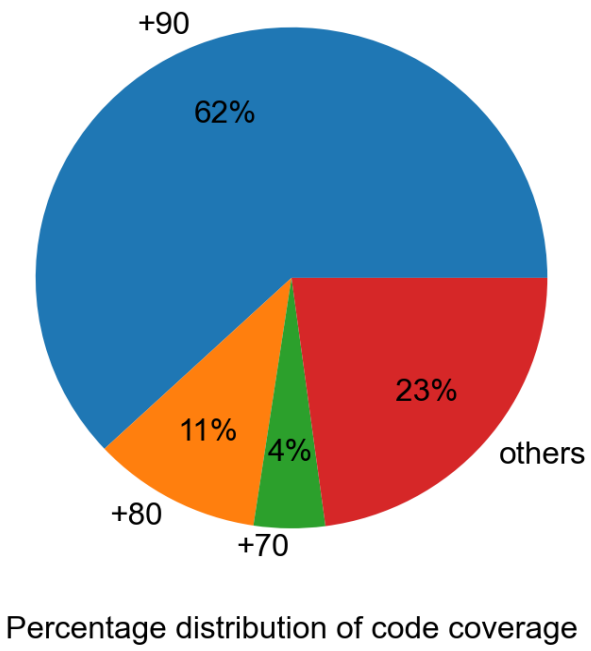

Figure 1: Percentage distribution of code coverage

SciPy has fairly good test cases which resulted in 62% of the files to have more than 90% code coverage. With 73% of files having more than 80% coverage, maintenance of the SciPy code would be relatively easier. This gives the developers and the testers an acceptable level of confidence to push new code without breaking any existing functionality. Code coverage helps both developers and testers in the same way in an aspect. The final goal is to develop software in optimal time with maximum efficiency. Adding tests allows the developer to write code faster covering all requirements and helps the tester to test all the scenarios in less time as a few of them would have been covered already in the test cases.

https://github.com/sindhugummadi9/PAT-Project-Scipy

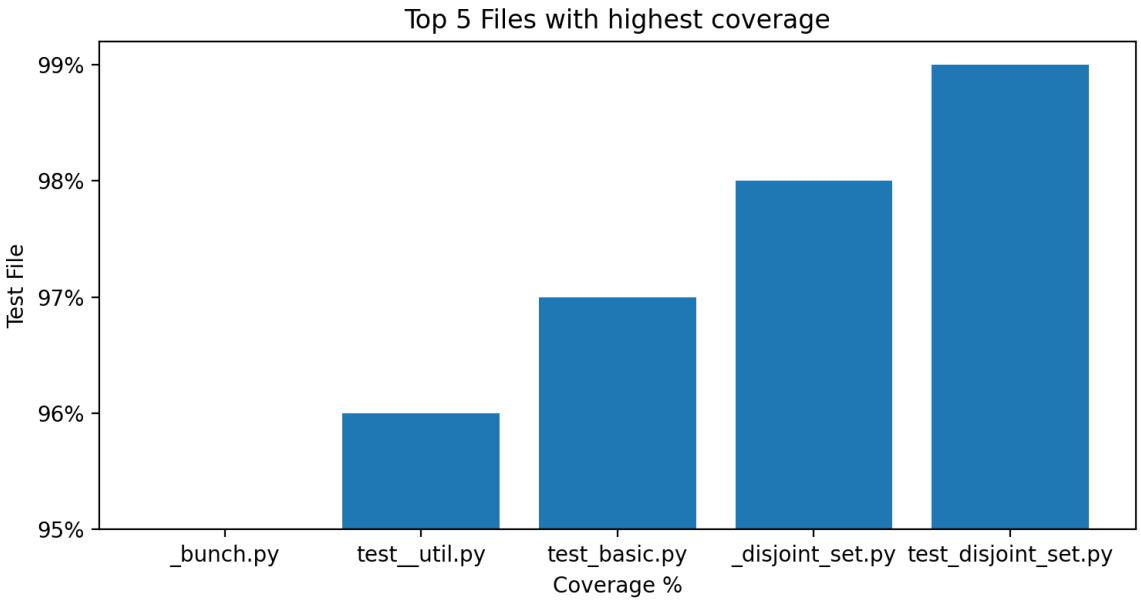

Figure 2: Top 5 files with Highest Coverage

The above figure shows the top 5 files in the SciPy code with maximum coverage, the highest being 99%.

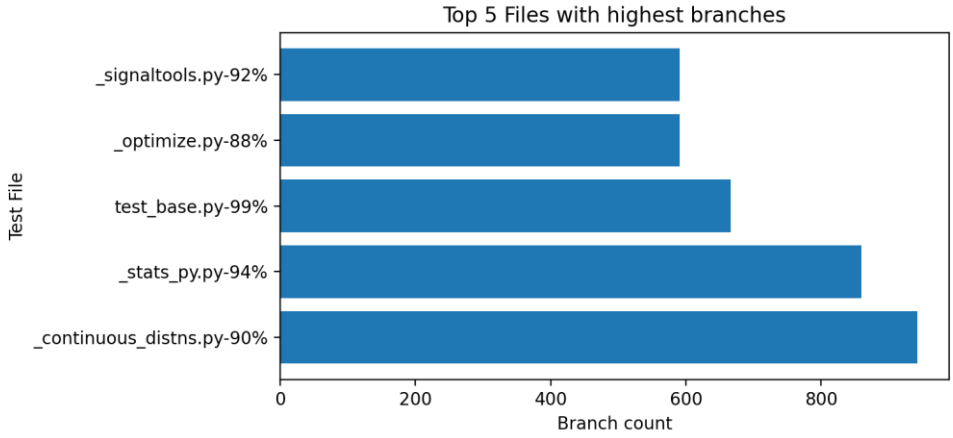

Figure 3: Top 5 files with Highest branches

The figure 3 shows the branch coverage of the SciPy repository. Branch coverage is essentially a metric that shows the tests cover all the possible execution paths in a program. Branch coverage is an important metric which can help a tester and a developer to identify whether an application has been tested to completion. A low branch coverage shows that there are scenarios in the application lacking testing. Such scenarios might contain defects that will only manifest in edge cases when the application makes it to

production. From the figure we can see that the file with the highest number of branches has about 90% coverage. With all the branches covered, a developer can be sure of all the edge cases. Part of a table showing the coverage report for a few classes can be seen below.

Coverage report: 90%

filter...

Module	statements	missing	excluded	branches	partial	coverage
scipy/__config__.py	42	26	0	18	1	28%
scipy/__init__.py	61	9	0	14	4	83%
scipy/_build_utils/__init__.py	18	8	0	0	0	56%
scipy/_build_utils/_fortran.py	189	167	0	82	0	8%
scipy/_build_utils/compiler_helper.py	77	77	0	35	0	0%
scipy/_build_utils/system_info.py	124	116	0	68	1	5%
scipy/_build_utils/tempita.py	14	14	0	6	0	0%
scipy/_build_utils/tests/__init__.py	0	0	0	0	0	100%
scipy/_build_utils/tests/test_scipy_version.py	10	1	0	2	1	83%
scipy/_distributor_init.py	0	0	0	0	0	100%
scipy/_lib/__init__.py	3	0	0	0	0	100%
scipy/_lib/_boost_utils.py	5	5	0	0	0	0%
scipy/_lib/_bunch.py	72	3	0	30	2	95%
scipy/_lib/_ccallback.py	98	13	0	42	6	85%
scipy/_lib/_disjoint_set.py	65	1	0	24	1	98%
scipy/_lib/_docscrape.py	443	125	0	224	47	65%
scipy/_lib/_gcutils.py	31	1	0	12	1	95%
scipy/_lib/_pep440.py	198	55	0	100	7	70%
scipy/_lib/_testutils.py	121	59	0	56	2	49%
scipy/_lib/_threadsafety.py	32	0	0	10	1	98%

Table 1: Coverage Report generated from build

Test files and Assertions in SciPy

Unit testing in an application helps us to isolate individual components of the system and makes sure each one of them works up to the expectations independently.

scipy\test_bunch.py	35
scipy\test_ccallback.py	12
scipy\test_deprecation.py	0
scipy\test_import_cycles.py	0
scipy\test_public_api.py	0
scipy\test_tmpdirs.py	10
scipy\test_warnings.py	0
scipy\test__gcutils.py	14
scipy\test__pep440.py	1
scipy\test__testutils.py	0
scipy\test__threadsafety.py	4
scipy\test__util.py	28
Total number of test.py files : 287	

Table 2: Number of Assert Statement in each Test File

https://github.com/sindhugummadi9/PAT-Project-Scipy

Testing ensures that all code meets quality standards before it is deployed, which provides a reliable engineering environment. Testing facilitates safe refactoring, improves code quality, reduces the cost of detecting an error and provides quick access to code documentation. In the SciPy project, there are close to ~300 tests which give a code coverage of 90% with an average of 66 assertions per file. Having these test files helps the developers to build additional features with the confidence of not breaking any existing code.

Python's assert statement is essentially a sanity check. Assertions can be used to test if certain assumptions remain true while we develop our code. If any of the assertions turn false, then we have a bug in our code. Assertions mainly help the developer in debugging the code. The primary role of assertions is to trigger the alarms when a bug appears in a program. Hence, they are also a great asset in documenting and testing the code. We can write concise and to-the-point test cases because assertions provide a quick way to check if a given condition is met or not, which defines if the test passes or do not pass.

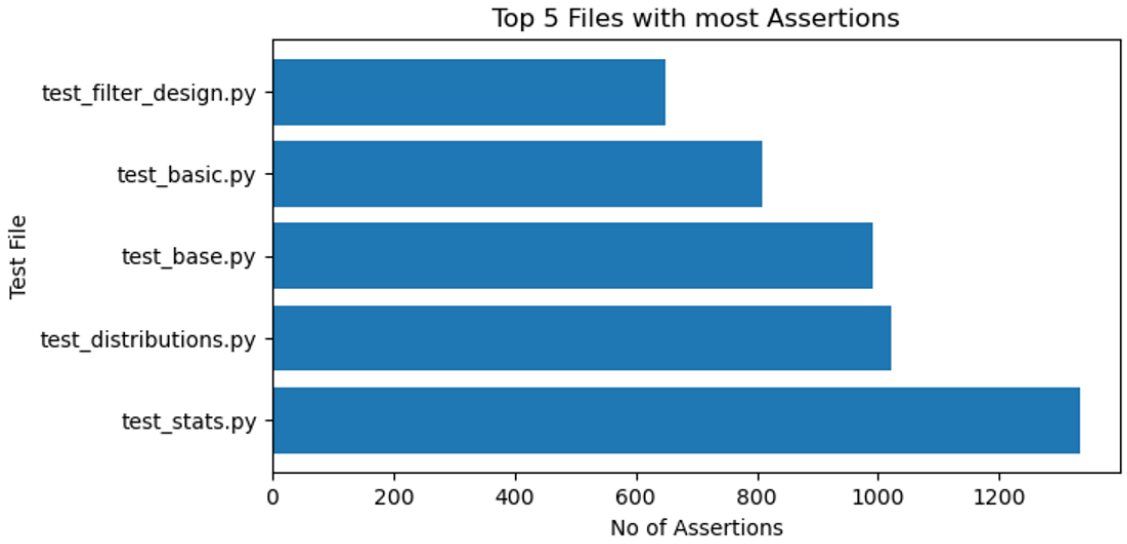

Figure 4: Top 5 files with Most Assertions

In the above figure we can see that the highest number of assertions in a file is around 1300. A decent number of asserts along with a good code coverage percentage, gives us an acceptable level of confidence in the production efficiency of our project.

https://github.com/sindhugummadi9/PAT-Project-Scipy

Debug and Assert Statements in Production Files

Given the SciPy repository didn't have any separate test folder, it is assumed that all the files inside the individual test folders of the SciPy folder belong to the test files. Any file outside the scope of the test directory is considered to be the production file. Firstly, assert and debug statements provide an insight to the developer and anyone going through the code the messages, and the flow at which the program is working. There are many modules in python which provide the debug functionality.

Any statement which has pdb.run or a statement that contains the err, log, and info statements are considered to be the debug statements in the scope of the script. There are many other debug statements that can be used in python, but these are considered to be the highly used ones and are mostly used in this project.

Firstly, any debug statement in the production files ensures that the program is working as expected and the tasks are working in the way they are expected to work. An assert statement in the production file ensures the code is working as expected and helps to understand any use case that has not been thought of while developing the module.

The sample output of the script run to find the assert and debug statements is as below. The detailed output can be found in the part3_output.csv file uploaded to the repository:

File	Number of Assert Statements	Number of Debug Statements
scipy-main/scipy/linalg/_matfuncs_inv_ssq.py	0	5
scipy-main/scipy/linalg/_testutils.py	2	0
scipy-main/scipy/linalg/_special_matrices.py	0	1
scipy-main/scipy/optimize/_tstutils.py	0	1
scipy-main/scipy/optimize/_dual_annealing.py	0	8
scipy-main/scipy/optimize/_trustregion_constr/tr_interior_point.py	0	1
scipy-main/scipy/integrate/_quadrature.py	0	1
scipy-main/scipy/_lib/_gcutils.py	3	0

Table 3: Number of Assert and Debug Statements in Production Files

https://github.com/sindhugummadi9/PAT-Project-Scipy

In this project, there were a total of 12 assert statements distributed in various files. The file _mptestutils.py in the special directory is observed to have the highest number of assert statements which is 4 and the least number of assert statements were found to be in _testutils.py in linalg directory. The detailed plot of assert files is attached below.

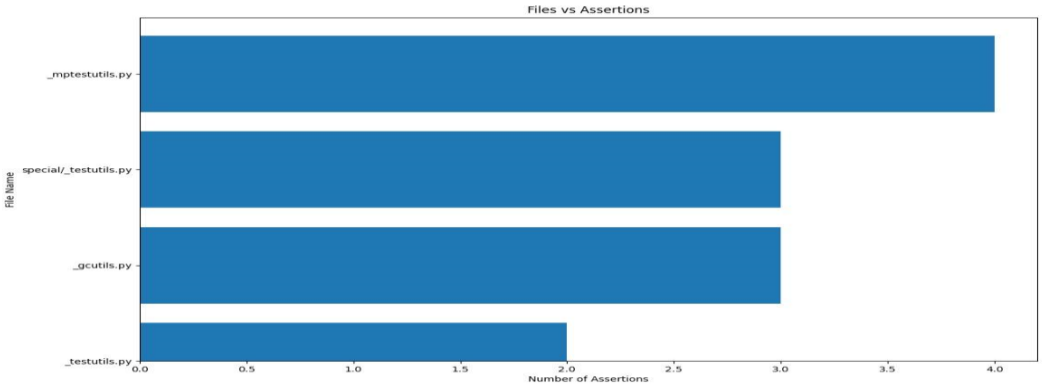

Figure 5: Top 5 files with Most Assert Statements

However, there were considerably many debug statements in the production files. There were 303 debug statements in the production files. Most of this was contributed by the _continuous_distns.py file in the stats directory. On further analysis, this file has a very important statistical processing which we believe was one of the important elements of the project. The detailed graphical analysis of the debug statements is as below:

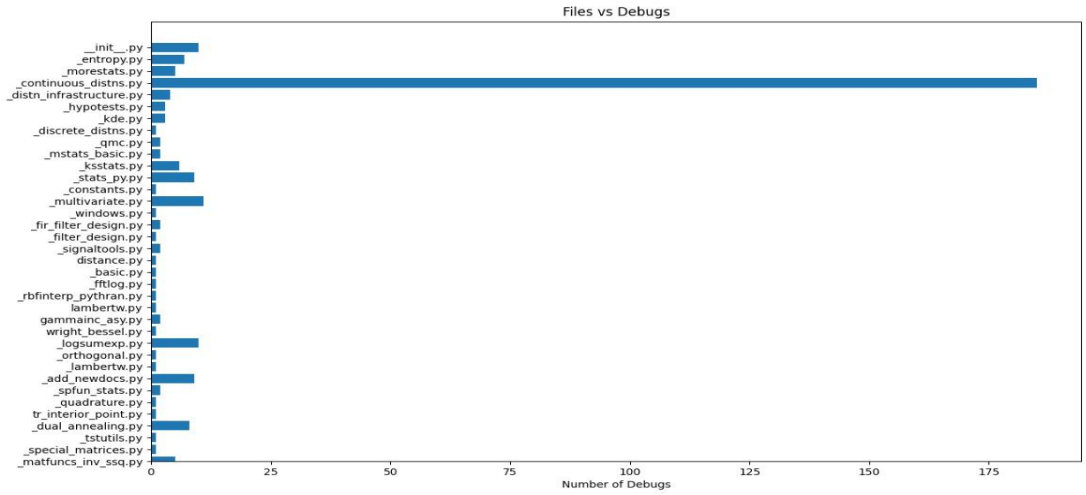

Figure 6: Number of Debug Statements in each Production File

https://github.com/sindhugummadi9/PAT-Project-Scipy

PyDriller For Assessment of SciPy Git Repository

From the docs, "*PyDriller is a Python framework that helps developers on mining software repositories. With PyDriller you can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly export CSV files".* We used PyDriller to fetch details of the SciPy repository like timeline of modifications on the repository, details about authors who contributed to the repository, commit info, etc.

The figure below provides a bar graph representation of Years to the number of Test Files added each year.

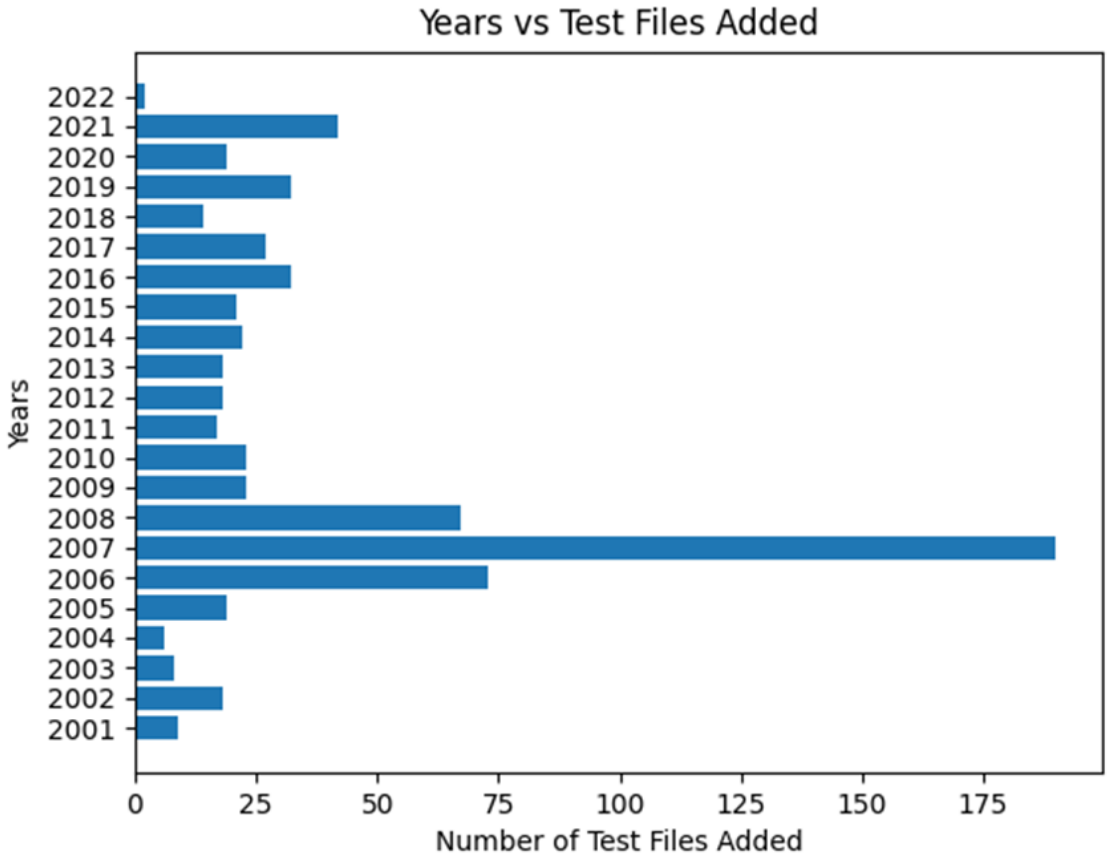

Figure 7: Number of Test Files added each Year

https://github.com/sindhugummadi9/PAT-Project-Scipy

It represents a rather irregular pattern of addition of test files to the SciPy Library Git Repository (https://github.com/scipy/scipy). It cannot be clearly commented that as years passed test files addition increased in the repository, as in some years, number of test files added were less than its previous years (2011) while in some cases it was way more than its previous years (2007). It can be interpreted that the number of test files added to the repository in a particular year is proportional to the contribution made to the SciPy library in that year. Any change in the number of test files added to the repository from the previous year indicates a better stable version of library features from its previous releases. Addition of test files can mean two things – new development script is added which requires testing or new test cases were written for previously written scripts to enhance the code coverage. Considering the year 2007, it experienced the most number of test files addition, and holds the highest contribution to the repository till now concerning test files. This provides an acceptable level of confidence in the library from the very initial stages. Post year 2008 there have been small differences with few exceptions in the addition of test files in the repository, which is indicative of either new test cases written for better code coverage of development scripts or addition of new test files for newly added development scripts.

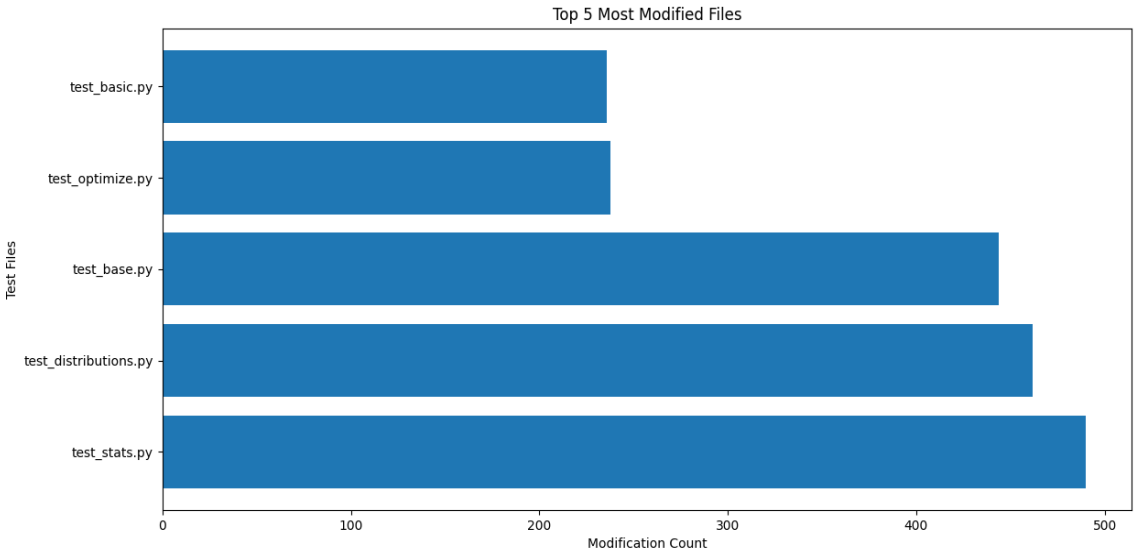

Figure 8: Top 5 Most Modified Files

https://github.com/sindhugummadi9/PAT-Project-Scipy

The figure above shows the top five test files that have been modified the most number of times in the repository.

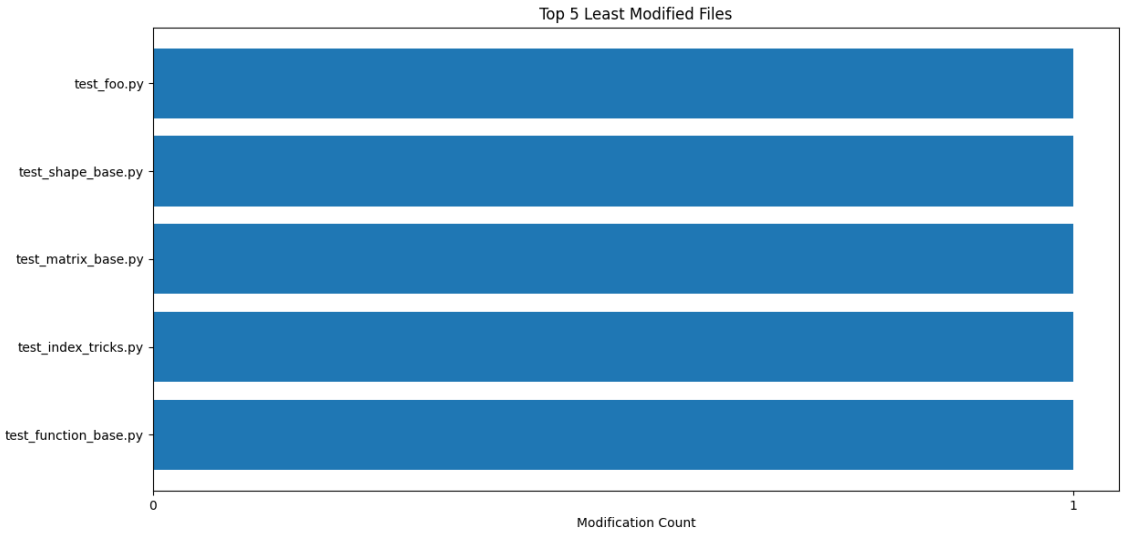

Figure 9: Top 5 Least Modified Files

The figure above shows the top five test files that have been modified the least number of times in the repository.