Identifying Test Case Combinatorial Explosions

With Python's Abstract Syntax Tree (AST) and Pytest Framework

Arun Vishwanathan

arunvishwanathan88@gmail.com

Abstract

Test matrix explosions are a growing challenge in modern software testing, especially when using parameterized tests in machine learning pipelines. This paper presents a hybrid approach combining Python's Abstract Syntax Tree (AST) analysis and the Pytest framework to map test parameterizations and reveal hidden redundancies. By applying this static analysis approach, testers can better understand test-model relationships and selectively reduce test coverage where appropriate. While this paper does not provide formal quantitative benchmarks, early usage of the technique suggested meaningful reductions in CI/CD execution time. The technique is lightweight, adaptable, and can support smarter test suite management in high-scale environments.

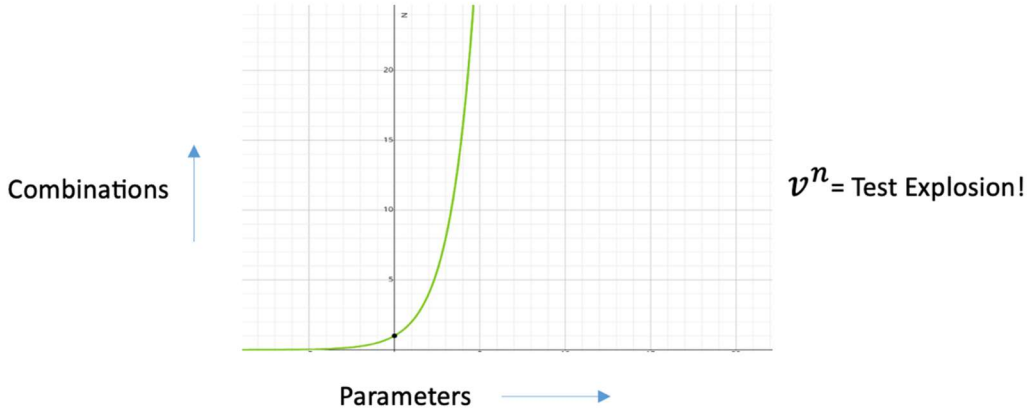
Biography

I work as a Senior Software Development Engineer in Test at Apple, Inc., in the Machine Learning organization. I create testing automation tools and frameworks for use within the team and across other teams, with a focus on efficiency and improving productivity. These tools help qualify the product stack as a whole and enrich the customer experience. As part of my role, I also present product health statistics to senior company executives.

1 Introduction

Python-based testing frameworks like Pytest are widely used for unit, integration, and system-level testing. As testers, we strive to maximize coverage across diverse workflows, from simple user interactions to complex developer pipelines. This results in several tests being identified and authored easily but becomes exponentially more difficult to manage and audit for efficiency or overlap.

As a result, test matrices tend to grow exponentially as features evolve, especially when leveraging parameterized testing.



2 The Challenge: When Parameterization Becomes Unmanageable

Parameterization enhances coverage but it can silently grow to an unmanageable scale. This is particularly common in machine learning (ML) environments, where tests involve varying model parameters. For a medium size project, unit tests could range from about 300-1000+ tests spanning over different hardware. And for much larger projects/products spanning across teams and devices, there could be thousands of tests in the suite somewhere from 2000 – 10000+.

2.1 Problems Observed

- Longer test execution times delaying feedback loops
- Redundant testing without improving code coverage
- Increased maintenance overhead, making test suite management harder
- Scaling challenges when running tests across multiple devices and platforms

To maintain efficient testing without sacrificing coverage, I aimed to analyze and optimize the test matrix strategically.

3 Analyzing and Optimizing the Test Matrix

Writing tests for machine learning models by converting them from one format to another format for optimization, tuning or other purposes, can follow the following sequence:

1. Parameterize the test function.

2. Fetch a source machine learning model and/or accompanying data.

3. Convert the fetched original model to a different format.

4. Execute the updated model under various test parameters.

Because such tests worked with some open-source models, I hypothesized that certain source models were over-represented across multiple test cases. Therefore, I needed a way to bridge information between these models and the test cases. (Step 2 and Step 4 above).

To achieve this, I used a hybrid approach combining:

- Pytest's `--collect-only` option to list all parameterized test cases
- Python's Abstract Syntax Tree (AST) for static analysis of test files

Let's first talk about using Pytest and see how it provides an insight into our test variants created with parameterized testing

4 Analyzing Test Files to Extract Parameterized Test Information

Over half of Python developers today use Pytest, making it the most popular unit-testing framework, compared to unittest which comes in second at about 24%

Consider the following parameterized test sample:

```
class TestModels():
   @parameterized.expand(itertools.product(PARAM, DEPLOYMENT_TARGETS))
   def test_modelexported_platform(param, precision):
   # Test implementation here
```

Assuming:

```
PARAM = param1, param2
DEPLOYMENT_TARGETS = target1
```

To observe how this expands, install the Pytest Python package and execute from the terminal looking only at the test file

```
> (python environment) pytest --collect-only test_*
```

The output looks like this:

```
Test session starts
collecting ...
<Package ABC>
  <Module M>
    <UnitTestCase TestModels>
        <TestCaseFunction test_modelexported_platform_param1_target1>
        <TestCaseFunction test_modelexported_platform_param2_target1>
```

This provides visibility into the expanded test matrix. However, it doesn't yet reveal which input models are used (**Step 2 of Identifying Common Patterns in Tests above**), which was critical for identifying redundancies in this case.

Let's now explore the concept of Abstract Syntax Trees in general and how you can effectively use them in Python towards the goal

5 Using AST

An Abstract Syntax Tree (AST) is a tree representation of the structure of source code. It does not require the execution of the script, but instead, Python can parse it into an AST, which allows developers to analyze code statically. AST is a technique used widely in code analysis in compilers, linters, parsers, etc. for code inspection.

Python's ast module comes built-in, so there is no need to install it separately. All that's needed is the simple import of the module

```
import ast
```

5.1 Parsing Python Code with AST

Consider a simple example `greeting.py`:

```
def greet(name):
    message = f"Hello, {name}!"
    print(message)

greet("Sam")
```

Now, to parse this file with AST:

```
import ast
# Read and parse the file
with open("greeting.py", "r") as f:
    tree = ast.parse(f.read())
# Print the AST structure
print(ast.dump(tree, indent=4))
```

You can see the output from this file. The output displays a structured code representation revealing function definitions, variable assignments, and function calls

```
Module(
  body=[
    FunctionDef(
      name='greet',
      args=arguments(
        posonlyargs=[],
        args=[
          arg(arg='name')
        ],
        vararg=None,
        kwonlyargs=[],
        kw_defaults=[],
        kwarg=None,
```

```
        defaults=[]
      ),
      body=[
        Assign(
          targets=[
            Name(id='message', ctx=Store())
          ],
          value=JoinedStr(
            values=[
              Str(s='Hello, '),
              FormattedValue(
                value=Name(id='name', ctx=Load()),
                conversion=-1
              ),
              Str(s='!')
            ]
          )
        ),
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              Name(id='message', ctx=Load())
            ],
            keywords=[]
          )
        )
      ],
      decorator_list=[]
    ),
    Expr(
      value=Call(
        func=Name(id='greet', ctx=Load()),
        args=[
          Constant(value='Sam')
        ],
        keywords=[]
      )
    )
  ],
  type_ignores=[]
)
```

5.2 Python's `ast.NodeVisitor` Class to traverse and extract information

`ast.NodeVisitor` is a base class provided by Python's `ast` module that helps you walk (or traverse) an Abstract Syntax Tree (AST) in a structured way.

`visit_*` methods handle a specific node type in the Python AST. When you traverse the tree with NodeVisitor.visit(node), Python automatically dispatches to the appropriate method based on the node type.

- `visit_FunctionDef`: called when a function definition node (i.e., a function definition like `def my_func():`) is encountered.

- `visit_Assign`: called when an assignment node (i.e., a variable assignment like `x = 1`) is encountered.
- `visit_ClassDef`: called when a class is defined (i.e., a class definition like `class MyClass:`) is encountered.

Each `visit_*` method is automatically invoked when the visitor walks over a corresponding node type in the AST.

5.3 Extracting Specific Information from the AST

Let us say that you want to inspect the function defined in the greeting script earlier. You can walk through the AST like this with a class defined to override the function visitor method:

```
class FunctionVisitor(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        print(f"Function name found: {node.name}")
        self.generic_visit(node)

# Parse and analyze the script
tree = ast.parse(open("greeting.py").read())
visitor = FunctionVisitor()
visitor.visit(tree)
```

This will output:

```
Function name found: greet
```

Now that you have a basic understanding of an AST and how to hook into the code, we can apply these concepts to our original problem.

5.4 Extracting Source Models/Data Using AST Analysis

Consider the test suite which loads the source models using a function `<my_model_loading_function>`

```
class TestTorchBasicModels(unittest.TestCase):
    @pytest.mark.parametrize(
    "quantize_fp16, deployment_target",
          itertools.product(
                [True, False],
                ["ios15", "ios16"],
          ),
     )
    def test_demo_net_from_file1(self, quantize_fp16, deployment_target):
          original_model=<my_model_loading_function>("torch_models/DemoNet.pt")
           self.convert_and_compare(original_model, (1, 1, 28, 28),
quantize_fp16=quantize_fp16, minimum_deployment_target=deployment_target)
```

A simple AST-based analysis simply walks over the code and searches for such function usages. We can define our visitor class with visitor methods overridden

```
def extract_model_data_calls(file_path):
    tree = ast.parse(f.read(), filename=file_path)
       class GetFileVisitor(ast.NodeVisitor):
```

```
        def __init__():
            self.current_class = None
            self.current_function = None
              . . .

        def visit_ClassDef(self, node):
               self.current_class = node.name
            self.generic_visit(node)
              . . .

        def visit_FunctionDef(self, node):
            self.current_function = node.name
            self.generic_visit(node)
              . . .

        def visit_Call(self, node):
            if isinstance(node.func,  ast.Name) and \
                     node.func.id == 'my_model_loading_function'):
                  package_name = os.path.dirname(file_path)
                  module_name = os.path.basename(file_path)

                  results.append({'package_name': package_name,
                                  'module_name': module_name,
                                  'file_name': node.args[0].value,
                                  'class_name': self.current_class,
                                  'test_name': self.current_function})

                  self.generic_visit(node)
```

This process pulls out **`torch_models/DemoNet.pt`**, allowing you to proceed to map model strings to the actual expanded test cases, thereby enabling redundancy analysis.

The entry for this case, in the `results` list would look like this for the above case

```
{
     'package_name': 'packageA',
     'module_name': 'moduleA',
     'file_name': 'torch_models/DemoNet.pt'
     'class_name': 'TestTorchBasicModels',
     'test_name': 'test_demo_net_from_file1'
}
```

The results can be collected across the different test files in the suite

```
ast_results = []
for file_path in test_files:
    ast_results.extend(extract_model_data_calls(file_path))
```

6 Analyzing and Optimizing the Test Matrix

After extracting test parameterizations (`pytest --collect-only`) and the input models from the AST analysis, the two datasets were combined to identify model-test relationships.

Assume the Pytest results were further parsed and were collected in the format of:

```
package::module::classname::expandedtestname
```

The elements in the `ast_results` obtained from the `extract_model_data_calls` can also be serialized to a similar fashion as a string representation.

The final mapping is then obtained by the merge of the two results based on the common prefix hierarchy of package name, module name, class name and test name present in the two updated representations:

```
mapping = combine_results(ast_results, pytest_results)
```

The following code snippet below shows the final obtained mapping that could possibly be obtained for one case:

```
{
  "torch_models/DemoNet.pt": [
    "packageA::moduleA::classnameA::test_demo_net_from_file1",
    "packageA::moduleA::classnameA::test_demo_net_quantized_from_file1",
    "packageM::moduleB::classnameF::testnameX",
    "packageM::moduleB::classnameF::testnameY"
  ]
}
```

From this mapping, you can observe that it's not just `test_demo_net_from_file1()`that references **`torch_models/DemoNet.pt`**, but also other test cases like **`testnameX`** and **`testnameY`** located in different packages/modules. This comprehensive overview of model-to-test relationships reveals the extent of model reuse across the test suite.

By identifying these overlaps, you can focus on over-tested models, guiding targeted optimizations to reduce redundancy and improve CI/CD efficiency.

By analyzing the mapping of models/data to expanded parameterized test name, a smaller test matrix was selectively applied in areas where faster turnaround was needed. The following optimizations were observed during initial usage:

- Reduced redundant tests where multiple cases used the same model to validate similar behaviors
- This has a lot of potential in narrowing a wide set of tests to a core set of smoke tests even if the overall test matrix is required to maintain a matrix with known scenario overlap (such as specific regulatory or other legal requirements).
- Improved CI/CD feedback cycles by reducing execution time without a noticeable loss in test coverage

While these results are anecdotal, they demonstrate the potential of this technique to streamline test strategy in complex environments.

6.1 Challenges/Limitations

There are some limitations with the approach as it depends on how the tests are structured:

- **AST Limitations**: Custom handling is needed when tree parsing involves nested function calls, indirect model loading, or dynamic data generation. The examples shown assume static, analyzable code structures.

- **Dynamic Model/Data Selection**: If models are loaded dynamically at runtime or fetched via indirect references (e.g., dictionaries, config files), static AST analysis may not reveal the full set of dependencies.
- **No Quantitative Benchmarking (Yet)**: This paper does not include formal measurements such as time saved or test count reduced, as the focus was on building a scalable analysis framework.
- Because we're using AST, this is limited solely to Python unit testing. But there are other AST-type libraries for other languages. So, this concept could be extended to other languages and test frameworks.

However, informal applications of the technique showed promising outcomes in reducing redundant tests and improving test feedback cycles. Systematic measurement is planned as future work.

7 Conclusion and Adaptability

By combining AST-based static analysis with Pytest's parameter collection capabilities, this approach helps testers visualize test redundancy and reduce combinatorial explosion in large test suites. It provides a way to assess which models or datasets are over-tested, enabling smarter decisions about test trimming or scheduling, leading to faster and more maintainable CI/CD pipelines.

Although this paper does not present formal performance metrics, the methodology has already been applied internally in targeted cases, yielding faster turnaround during CI/CD cycles. Future work includes integrating this analysis into continuous pipelines and collecting quantitative metrics on test reduction and efficiency gains.
This methodology can be adapted for broader testing scenarios, making it a powerful tool for modern test engineers.
It can be adapted as needed where there are repeatable static patterns in code that help visualize the source of various tests being used.

References

https://docs.pytest.org/en/stable/how-to/usage.html

https://docs.python.org/3/library/ast.html