Automated Flaky Test Detector

Overview

Flaky tests are defined as tests that experience failures intermittently. Flakiness arise from many causes ranging from numeric errors to race conditions. The existence of flaky tests are both meaningful and harmful at the same time: they are revealing possible problems in our code, but it's also costing extra CI builds to be run. Flaky tests also need to be evaluated on an individual basis to determine the correct approach to resolving them, which may involve fixing code that causes the problem to occur, or increasing the tolerance of the test.

As of the creation of this tool, there is an effort to fix the currently known flaky tests, but we want to ensure that in the future we are not continuing to add flaky tests to our code base. Our solution to the problem will be to try to identify flaky tests, before they go into the code base and cost us CI runs, so that someone can look at them and make further decisions. This will involve automatically detecting tests that have been changed by a pull request and running those tests a sufficient number of times to determine if they are flaky or not. If a flaky test is detected the pull request will be rejected or in some way marked as flaky so that the original committer can have ownership over the fix.

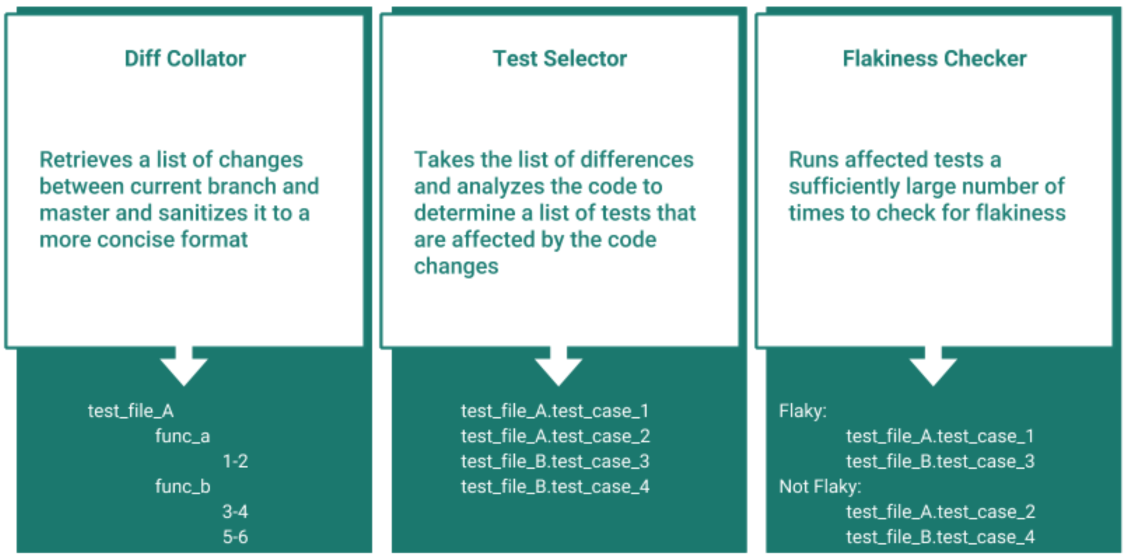
In theory, determining that a test isn't flaky would mean running the test an infinite number of times to confirm that it always succeeds; however, since we are limited in the number of test trials we can run, we need to choose a number that will balance both our confidence in the flakiness of the test and the resource consumption of the detection process. On CI, the tool will operate with a time bank, which will be distributed among tests, in order to ensure that it does not increase the duration of PR checks.

Scope

This project is currently aimed towards unit tests written in python, since these currently comprise the majority of out tests on CI. While this tool could theoretically be used to test integration tests, the time required to run these tests would make such coverage impractical. It also will not cover things such as test-order dependency or flakiness that is only reproducible in a specific environment. The main goal of this project is to try to provide a baseline coverage that can catch most flakiness with low effort and high impact.

Design

The flaky test bot is composed of three components: Diff Collator, Test Selector, and Flakiness Checker. These stages each serve as a part of the flaky test detector, but have been decoupled from each other. Each stage exposes human-understandable input and output to facilitate debugging and extensibility. For example, the the flakiness checker may be useful as a standalone tool for developers working on flaky test fixes or implementing new test cases.


Diff Collator
Retrieves a list of changes between current branch and master and sanitizes it to a more concise format
Test Selector
Takes the list of differences and analyzes the code to determine a list of tests that are affected by the code changes
Flakiness Checker
Runs affected tests a sufficiently large number of times to check for flakiness

```
test_file_A
        func_a
                1-2
        func_b
                3-4
                5-6
```

```
test_file_A.test_case_1
test_file_A.test_case_2
test_file_B.test_case_3
test_file_B.test_case_4
```

```
Flaky:
        test_file_A.test_case_1
        test_file_B.test_case_3
Not Flaky:
        test_file_A.test_case_2
        test_file_B.test_case_4
```

Diff Collator

The purpose of the diff collator is to retrieve a list of changes that have been made to the code and sanitize it before passing it to the next stage. This is done by taking the output of git diff and parsing it to obtain file names, top-level functions, and line numbers for each code change. For the purposes of the flaky test detector, only file and function names are needed for the dependency analyzer.

Usage

- Targets- By default the diff collator uses master and HEAD as the targets for git diff, but these can be specified using the --commits (-c) option, which directly compares two commits, or the --branches (-b) option, which compares the second commit to the common ancestor.
- Verbosity- 3 verbosity levels can be specified by repeating the --verbosity (-v) option (e.g. -vv corresponds to verbosity level 2). Verbosity 1 outputs just file names, 2 outputs files with functions, and 3 outputs files, functions and line numbers. Defaults to 2.
- Filters- Filter files that are included in the diff. --path (-p) can be used to specify a directory to be diffed, and --filter (-f) can be used to diff files that match a particular python regular expression. See https://docs.python.org/2/library/re.html for details.

Sample output:

```
python tools/flaky_test_bot/diff_collator.py -vvv --filter .*dependency_analyzer\.py
tools/flaky_test_bot/dependency_analyzer.py

    top-level

        19:23

        25:25

        27:31

    read_config

    find_dependents

        54:54

        58:58

    find_dependents_file

        67:71

        74:74

        103:103
```

Dependency Analyzer

The dependency analyzer must be able to handle several different cases when it comes to codes changes:

- A new test has been added
- An existing test has been changed
- A function upon which a test is reliant has been changed

Doing this within a single file simply means parsing the file for calls to function dependencies and returning the top-level callers. Across files, this is accomplished by storing cross-file dependencies in a config file, which is written in json; when a dependency in the config file is selected, its dependents are automatically selected as well. This avoids having to parse through every test file, but it also means that the config file must be updated when cross-file dependencies are changed.

Cross-file dependencies

As mentioned above, cross-file dependencies are handled using a config file. below is an example of such a file.

```
{
    "tests/python/gpu/test_operator_gpu.py": [
        "test_operator.py",
        "test_optimizer.py",
        "test_random.py",
        "test_exc_handling.py",
        "test_sparse_ndarray.py",
        "test_sparse_operator.py",
        "test_ndarray.py"
    ],
    "tests/python/gpu/test_gluon_gpu.py": [
        "test_gluon.py",
        "test_loss.py",
        "test_gluon_rnn.py"
    ],
    "tests/python/mkl/test_quantization_mkldnn.py": [
        "test_quantization.py"
    ],
    "tests/python/quantization_gpu/test_quantization_gpu.py": [
        "test_quantization.py"
    ]
}
```

Each file with other file dependencies is listed along with a list of its dependencies. In this example, test_gluon_gpu.py depends on three files: test_gluon.py, test_loss.py, and test_gluon_rnn.py. Currently this is done manually since the structure of our tests rarely changes; however, an automated solution may be worth implementing.

Usage

The dependency analyzer takes as input a list of function names and associated files in the format: <file>:<function-name> and return a list, with the same format, or top-level functions with associated files that are dependent on those in the input.

Flakiness Checker

The flakiness checker tool is a script that checks a test for flakiness by running it a large number of times. Its primary purpose is to serve as a component of the automated flaky test detection system. However, it can also be used manually by developers for flaky test fixes or when modifying test files. Before submitting a new test case or a modification to an existing test, run the flakiness checker script to test for flakiness.

Number of Trials

Below is a table indicating the number of runs needed to achieve a given confidence level with a given chance that a test passes. As a default, on CI we are using a value of 10,000 to check tests. This number will give us a relatively high confidence that we are catching flakiness even if it only occurs

success rate \ confidence	99%	99.9%	99.99%
99%	458	4,603	46,049
99.9%	687	6,904	69,074
99.99%	916	9,205	92,099

Usage

```
python flakiness_checker.py [optional_arguments] <test-specifier>
```

where <test-specifier> is a string specifying which test to run. This can come in two formats:

1. <file-name>.<test-name>, as is common in the github repository (e.g. test_example.test_flaky)
2. <directory/<file>:<test-name>, like the input to nosetests (e.g. tests/python/unittest/test_example.py:test_flaky). Note: This directory can be either relative or absolute. Additionally, if the full path is not given, the script will search whatever directory is given for the provided file.

Optional Arguments:

-h, --help print built-in help message

-n N, --num-trials N run test for n trials, instead of the default of 10,000

-s SEED, --seed SEED use SEED as the test seed, rather than a random seed

Integration

The integration plan for this project can be found here: Flaky Test Bot Integration Plan. This tool will be run as a Jenkins job that will be triggered on pull requests. The Jenkins job executes the diff collator and dependency analyzer in one step to detect modified or added test cases. If any tests are selected, then MXNet is compiled and the flakiness checker is run to check the tests for flakiness.