PLearn Programmer's Tools Guide

The tools that make the programmer's life simpler

Contents

Chapter 1

PLearn scripts

1.1 pymake

1.2 pyskeleton

1.3 perlgrep

1.4 search-replace

1.5 pytransform

1.6 cvschangeroot

Chapter 2

The PLearn test-suite

PLearn test-suite is still in development by its author Christian Dorion, but it is already usable and should probably be used by anyone who wants to ensure his code does not get broken accidentally.

Each *test* is a program or a PLearn script, which process some files (such as datasets), and outputs something to the standard output and error, and possibly to other files. The goal of the test-suite is to compare these results, run with a recently-compiled copy of PLearn, to the reference results created by the first run of the test.

The following instructions are a step-by-step example, for a test named `PL_Var_utils`, testing the `Var` class. It is not officially supported by Christian, and may not work anymore on a later version of the test-suite.

In addition, those who already know of the test-suite may find the PLearn object class `PTest` useful to write C++ tests that do not make the test-suite explode both in size and execution time (in addition to provide understandable floating numbers diff through the `PLearn diff` command applied on objects).

1. If it does not already exist, create the appropriate `test` directory, add this directory to the Subversion repository, and move into it:

   ```
   $ mkdir ${PLEARNDIR}/plearn/var/test
   $ svn add ${PLEARNDIR}/plearn/var/test
   $ cd ${PLEARNDIR}/plearn/var/test
   ```

2. *[Optional]* If your test relies on more than a couple of custom files

(data files, scripts, ...), better create its own directory:

```
$ mkdir Var_utils
$ svn add Var_utils
$ cd Var_utils
```

3. If it does not already exist, create a template `pytest.config` file:

```
$ pytest add
```

4. There are mainly two kinds of tests. If a simple `.plearn` or `.pyplearn` script is enough to run the test (e.g., if you plan to test a new PLearner class), go to step 14. If you need to create C++ code to test some specific functions, go on to step 5.

5. Create a PTest subclass template:

```
$ pyskeleton PTest VarUtilsTest
```

6. Edit the resulting files (e.g., `VarUtilsTest.h` and `VarUtilsTest.cc`):

   - fill the `PLEARN_IMPLEMENT_OBJECT` macro help:

   ```
   PLEARN_IMPLEMENT_OBJECT(
       arUtilsTest,
       "Test various functions in Var_utils",
       ""
   );
   ```

   - write your actual test code in the `perform()` method
   - store the test results, you can either:
     (a) display them (using the `pout` or `perr` PStreams, or the PLearn logging system):
     ```
     // In VarUtilsTest::perform
     pout << my_function(x) << endl;
     MAND_LOG << my_function(x) << endl;
     ```
     (b) store them in your `PTest` object options (this requires a little more work, but is actually easier to understand when the test fails):

```
// In VarUtilsTest.h
map<string, Vec> vec_results;
// In VarUtilsTest::declareOptions
declareOption(ol, "vec_results",
              &VarUtilsTest::vec_results,
              OptionBase::learntoption,
              "Test Vec results.");
// In VarUtilsTest::perform
vec_results["my_function"] = my_function(x);
```

(c) remember that the text (and PLearn binary-formatted) files that might be output by the program are also compared (no need to output them to `cout`).

7. Once your code compiles and is ready to be tested, add your new PTest in `PLearn/commands/plearn_tests_inc.h`:

```
#include <plearn/var/test/VarUtilsTest.h>
```

8. Edit your `pytest.config` file to specify how your test is supposed to be run. The name of your test should begin by "PL_" if it is a normal PLearn test.

   Typically you will run `plearn_tests`, on a `.plearn` or `.pyplearn` script (this script will describe one or more objects of your PTest subclass, with its specific options):

```
Test(
    name = "PL_Var_util",
    description = "Test various functions in Var_utils",
    program = GlobalCompilableProgram(
        name = "plearn_tests",
        compiler = "pymake",
        compile_options = ""
    ),
    arguments = "varutils_test.plearn",
    resources = [ "varutils_test.plearn" ],
    precision = 1e-06,
    disabled = False
)
```

   But if your `PTest` object does not need extra options, you can save the use of a script:

```
Test(
    name = "PL_Var_util",
    description = "Test various functions in Var_utils",
    program = GlobalCompilableProgram(
        name = "plearn",
        compiler = "pymake",
        compile_options = ""
    ),
    arguments = "PLEARNDIR:scripts/command_line_object.plearn " \
        "'object=VarUtilsTest()'",
    resources = [ ],
    precision = 1e-06,
    disabled = False
)
```

9. *[Optional]* For debug purpose, you may temporarily use a copy (say `plearn_mytests`) of `plearn_tests` in `pytest.config`, and also comment out all tests but your own test in `plearn_mytests_inc.h`. This will make compilation faster when debugging your test. Setting `compile_options` to `-opt` will also probably speed up the link time.

10. Run your test to generate results. Check everything is fine in the generated `.pytest/expected_results` hidden directory. If it is not, fix your code. Do this until you are happy with the results.

    ```
    $ pytest results -n PL_Var_util
    ```

11. If you had made any of the actions described in optional step 9, revert back to the standard test configuration.

12. Add your specific files to version control:

    ```
    $ svn add VarUtilsTest.h VarUtilsTest.cc
    ```

13. Go to step 19.

14. Write your `.plearn` or `.pyplearn` script, and make sure it runs smoothly.

15. Once you are confident it works fine, remove all the data that was generated while running it (the script must generate data files, or output to `cout` or `cerr`, otherwise it is useless).

16. Edit your pytest.config file to specify how your test is to be run:

```
Test(
    name = "PL_Var_util",
    description = "Test various functions in Var_utils",
    program = GlobalCompilableProgram(
        name = "plearn",
        compiler = "pymake",
        compile_options = ""
    ),
    arguments = "varutils_test.plearn",
    resources = [ "varutils_test.plearn" ],
    precision = 1e-06,
    disabled = False
)
```

17. Run your test to generate results. Check everything is fine in the generated `.pytest/expected_results` hidden directory. If it is not, fix your script. Do this until you are happy with the results.

```
$ pytest results -n PL_Var_util
```

18. Add your specific files to version control:

```
$ svn add varutils_test.plearn
```

19. Check that your test works fine:

```
$ pytest run
```

20. Confirm that the results obtained are correct for the current test. This will `svn add` the result files, and put the `svn:ignore` property on some files, but nothing will be committed yet.

```
$ pytest confirm
```

21. Do a `svn status` to check what files have been generated but should be ignored by version control, and ignore them:

```
$ svn propedit svn:ignore .pytest
```

And in the editor it opens, type:

```
*.compilation_log
run_results
```

22. Assuming your test passes correctly, it is ready for commit:

```
$ cd ${PLEARNDIR}
$ svn status          <-- to check which files you need to commit
$ svn commit commands/plearn_tests_inc.h plearn/var/test \
    -m "New test: VarUtilsTest"
```

Chapter 3

The speed benchmark suite

Chapter 4

External tools

4.1 cvs

4.2 valgrind

4.3 doxygen

4.4 latex2html

4.5 mpi

License

This document is covered by the license appearing after the title page.

The PLearn software library and tools described in this document are distributed under the following BSD-type license: