

Documentation for 57x Grading

David Inman

September 23, 2018

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 2 | Homework Descriptions | 2 |
| 3 | Rubric | 2 |
| 3.1 | Standard Portion (25 points) | 3 |
| 3.2 | Homework-specific Portion (75 points) | 3 |
| 4 | Creating Files for Grading | 3 |
| 4.1 | List of files needed | 3 |
| 4.2 | Submit List | 4 |
| 4.3 | Config File | 5 |
| 5 | Automated Checker (check_hw.py) | 6 |
| 5.1 | Checker dependencies | 6 |
| 5.2 | Student-facing check_hw.sh | 7 |
| 5.3 | Behavior | 7 |
| 5.4 | check_hw Tests | 7 |
| 6 | Grading Steps | 7 |
| 6.1 | Setup | 7 |
| 6.2 | Unzipping the zip file and file checking | 8 |
| 6.3 | Running student code | 8 |
| 6.4 | Evaluation | 9 |
| 7 | Code Documentation | 9 |
| 7.1 | Using Bitbucket | 9 |
| 7.2 | Code Structure | 9 |
| 8 | Potential Future Developments | 10 |
| 8.1 | Internalizing the run script | 10 |
| 8.2 | Grading special lines in the output | 10 |

1 Introduction

This document is meant to guide professors and TAs when creating and grading homework assignments for the 57x series. It is roughly structured according to the homework workflow: creating the homework description (§2), creating the rubric (§3), creating the necessary files for grading (§4), setting up the homework checker for student use (§5), and the use of the automated grader (§6). Finally, I have included some documentation for the structure of the grader and checker (§7) and future development (§8).

2 Homework Descriptions

There are a series of common mistakes students make when doing the homework for the 57X series. This is a set of best practices when writing the homework descriptions to minimize student confusion and errors, and make the homework easier to grade. The development of homework descriptions is presumed to be done by the professor.

1. Describe separate components of the homework separately.

The best practice for clarity is to separate different parts of the assignment by section. If possible, describe the expected function of the program (Viterbi, classification, etc) in one section, and then describe the format of the output in a separate section. Try to minimize component interleaving in the homework description.

2. Homeworks should include an at-a-glance summary of expected turned-in files.
3. Points should be given either in section headers or in a summary at the end. The points should add up to 75, the portion of the rubric assigned for homework-specific grading (§3.2).
4. Points should be awarded for program output, not for code structure.

This prevents the TA from needing to review the code for each student. For instance, “Program outputs a valid path (10 points)” is preferable to “Initializes lattice (10 points).”

5. Graded outputs should have a strict format.

This is particularly true within a line. The automated grader can sort lines alphabetically, but it cannot sort formats within a line. For instance, “On each line, print the probabilities for each class, `c1 prob1, c2 prob2, ...`” is too vague. “On each line, print the probabilities for each class in alphabetical order, `c1 prob1, c2 prob2, ...`” is better.

For every assignment, output should be uniform for all student code on a given input.

3 Rubric

The rubric across 570, 571, and 572 has been standardized. There are two parts: the standard component, which is graded automatically and identically across all assignments, and then the homework-specific portion, which is defined for each assignment.

3.1 Standard Portion (25 points)

There is no deviation for this component, and it is graded automatically by the grader (§6). It is graded as follows:

1. Submitted Files (10 points)
 - 1.1 Tarfile submitted (2 points)
 - 1.1.1 Exists (1 point)
 - 1.1.2 Correctly named (1 point)
 - 1.2 Readme file submitted (2 points)
 - 1.2.1 Exists (1 point)
 - 1.2.2 Correctly named (1 point)
 - 1.3 Requested files exist in tarfile (6 points)
2. Program runs as expected (15 points)
 - 2.1 Code runs to completion on input (10 points)
 - 2.2 Output of running code matches turned-in files (5 points)
3. Extra points off
 - 3.1 Carriage returns in turned-in output (-2 points)

3.2 Homework-specific Portion (75 points)

This section is based on the needs of each assignment. Although the gross distribution of points is defined by the professor in the homework description, the TA still has to make decisions about the finer distribution.

1. Some points should be assigned for correct formatting for each output file.
2. Points distributed for the program being implemented correctly should ideally depend on evaluating the program output on test files that are as modular as possible.

For instance, the grader could create an input that will test the code's ability to handle multiple labeling paths, and assign points for "Chooses the correct labeling path" based on that test. Then a separate test may handle unknown words. These test cases should not be intermixed (or there should be a separate test input that mixes them).

4 Creating Files for Grading

4.1 List of files needed

For each assignment, a series of files needs to be generated for the purposes of grading. This is an attempt to list all the necessary files. The first file (submit-file-list) is suggested to be the professor's responsibility, and the rest are suggested to be the TA's.

1. submit-file-list

This is a file that lists all the expected files in the student submission. For details, see §4.2.

2. config_hwX

This is the config file read by the automated grader. It is explained in more detail in §4.3.

3. run_hwX.sh

When the grader runs student files, it calls this shell script. This script will be called from the top directory of the student's untarred submit file, so all paths should be relative to the student's folder.

The run file can serve three functions, one mandatory and two optional:

1. It *MUST* rerun the commands that generate output the student turned in. These new output files should be put in the same location as the turned-in files with the same name, but with a `.out` appended. For example, if the student turns in a file at `Q2/lambda1.lm`, the run file should create a file `Q2/lambda1.lm.out`.

2. It may run student code on test input that the grader has created.

3. It may run any arbitrary scripts or functions that aid the grader.

4. Gold files

For every student output, there must be a gold output file. For clarity's sake, I suggest giving them the extension `.gold`. If you have withheld input or test data, there also must be gold output for those cases.

5. Withheld test input (optional)

If you are running student code on withheld test input (e.g., modular or targeted tests), these files need to be written.

The submit-file-list (1) needs to be made available to students for homework checker, so should be put on `/dropbox` with the assignment.

The rest of the files should be saved and passed on to help future TAs. The suggestion is to put the rest into a TA/instructor-only grading folder for the course (inside `/dropbox/year/course/grading`, but this will be negotiated with the professor and grading team).

4.2 Submit List

The submit-file-list lists all the files expected in the student's submitted tar file. The format is a set of newline-separated list of expected files, in relative path format. For instance, the file might look like the following:

```
run.sh
table
Q2/q2.sh
Q2/decoder.sh
Q3/q3.sh
Q3/results/data.txt
```

4.3 Config File

The config file is a plaintext file with configuration inputs separated by line. The format is similar to condor .cmd files, and there is no required order to the config file: each line stands on its own. The format and configuration lines are:

1. `zipfile = <zipfile name>`

`<zipfile name>` is the path to the zip file that was generated by Canvas when you downloaded the assignments.

2. `destination = <destination folder name>`

`<destination folder name>` is the local path to the folder where student submissions will be unzipped (for opening), or where the student submissions already exist (for evaluating).

3. `report = <name of report file>`

`<name of report file>` is the local path to the report file. If this line is omitted, the report file will be given the name `<destination folder>_report.txt`.

4. `file_structure = <submit-file-list>`

`<submit-file-list>` is the same file that the automated checker uses and that the professor creates for the assignment (see §4.2).

5. `ignore_newlines = <true|false>`

`ignore_newlines` tells the grading script whether to skip empty lines when comparing files. If omitted, it defaults to True.

6. `ignore_whitespace = <true|false>`

`ignore_whitespace` tells the grading script whether to tokenize individual lines by whitespace when comparing files. If set to False, the lines of each file are compared as-is with no processing, which blocks fuzzy matching (see below). If omitted, `ignore_whitespace` defaults to True.

7. `fuzzy_match = <float>`

`<float>` is a numerical value that specifies how different numeric values can be from one another when performing comparisons. For instance, a value of 0.1 means that student values can be up to 10% larger or smaller than the value in the gold standard. Fuzzy matching only occurs if `ignore_whitespace` is set to true. If omitted, `<float>` defaults to 0 (i.e., numeric values must be identical).

The following config lines can be repeated multiple times.

8. `repro_comparison = <student_file> <sorted|unsorted>`

This line should be repeated for as many student output files as the assignment requires. The grader will compare the file located at the *relative* path `<student_file>` to the expected file at `<student_file>.out` (which should be generated by `run_hwX.sh`).

You must specify whether the file should be **sorted** or left **unsorted** before comparison occurs. If the order of the output file is unimportant (for instance, it is the unordered output from a hash), then it is best to select **sorted**.

9. `gold_comparison = <gold_file> <student_file> <sorted|unsorted> <points>`

This line should be repeated for as many student output files as are being graded, including any test output generated by `run_hwX.sh`. `<gold_file>` is the *absolute* path to the gold file, and `<student_file>` is the *relative* path to the student file.

As with `repro_comparison`, you must specify whether the files should first be `sorted` before comparing, or left `unsorted`.

You must also specify the integer point value `<points>` assigned to matching the gold file. This is a blunt comparison: the grader will either assign full points or no points. If you wish to assign partial points, it may be easiest to tell the grader it is worth 0 points and then manually alter the grade. The grader does not check that the `gold_comparison` point values sum up to 75 (the points available for assignment after the standard portion is graded), so think carefully about how many points you want to assign to matching gold outputs.

5 Automated Checker (`check_hw.py`)

The automated checker is located under the `src` directory, and is for use by students. It confirms that their `hw.tar.gz` file has all expected files. It runs the same file-checking that the full grader does, but also checks for expected source code. The method for calling the code is:

```
python check_hw.py <languages> <submit-file-list> <tar_name>
```

5.1 Checker dependencies

1. `<languages>`

This is a file that contains a list of recommended coding languages and their executable formats. The format is line-separated, and each line lists an executable file format, followed by the source code formats. That is, each line looks like:

```
executable_extension code_extension1 code_extension2 ...
```

If a language is scripted, only the extension of the script appears, and there may be multiple lines with different executables. An example file could look like:

```
exe cs
exe c h
class java
jar java
py
pl
```

The master version of this file is called “languages” and lives in the `src` directory with the `check_hw.py` script.

2. `<submit-file-list>`

This is the same file described in 4.2.

3. `<tar_name>`

This is the name of the tar file the homework checker looks for. Following current 57x policy, it should be `hw.tar.gz`.

5.2 Student-facing `check_hw.sh`

The `check_hw.py` script is extensible and configurable. This is for flexibility for us, but not ideal for student use. The current policy is to create a wrapper, called `check_hw.sh`, inside the directory of each homework assignment. E.g., a `check_hw1.sh` in the `hw1` directory that calls into `check_hw.py` with the appropriate arguments, and so on.

5.3 Behavior

The `check_hw` script performs the following checks:

- Checks to see if the tar extracts. If it doesn't, prints an error.
- Checks for all files and folder structures listed in `submit-file-list`. Prints an error for each file and folder missing.
- Checks for windows carriage returns in all files listed in `submit-file-list`. Prints an error if any are found.
- Checks that all *.sh files begin with `#!/bin/sh`. Prints an error for each *.sh file that doesn't.
- Checks to see if there is some code from a language listed in `languages`. If not, prints a warning.
- If the tar includes binaries, checks for (expected) corresponding source code. Prints a warning if not found. If found, it prints the binary file and its (assumed) source code.
- If the script runs into an exception, it prints out some information and asks the user to email davinman@uw.edu with the tar file that triggered the exception.

5.4 `check_hw` Tests

The tests I ran for `check_hw.py` are under `data/test_hw_checker`. The directory contains a set of zip files with descriptive names, detailing what tests are run on each, and the expected results. To run all the tests, call `test_all.sh` from within the `data/test_hw_checker/` folder.

6 Grading Steps

After environment setup (6.1), there are three steps to using the automatic grader: unpacking the zip file from Canvas and grading the presence of requested files in the submission (6.2), running student code (6.3), and grading the run code (6.4).

6.1 Setup

The most important part of setting up is adding the version of the grader to your `PATH`. To do this permanently, you need to add the following line to your `~/.profile` file:

```
export PATH=$PATH:/dropbox/grading/grader57x/2.0/grader57x/scripts/
```

To use another version of the grader, change the 2.0 in the path to the preferred version. Once you have added this to your `~/.profile` your `PATH` will be set every time you log in. To use it immediately (without logging out and back in), run

```
source ~/.profile
```

For each homework, you will also need to set up the config file and gold files, as described in §4.

6.2 Unzipping the zip file and file checking

To open the zip file, use the parameter `open` when calling `grader.sh`:

```
grader.sh <hw_config> open
```

This will expand the zip file and all students' tar files into the destination folder defined in `config_file`.

The destination folder will have student submissions sorted into folders based on the student's name as given by Canvas. Each folder will be a concatenated string of the student's last, first, and middle names (in that order), and within each folder, the submitted tar file will be exploded.

Upon opening student folders, the grader will grade the "Submitted Files" section of the standard portion of the rubric (§3.1). This information will be written out to the report file defined in `config_file`, which is sorted alphabetically by student name.

If a student submits their code late or separately, use the parameter `open_student` followed by the student's name and their tar file. For example:

```
grader.sh hw1_config open_student doejohn john_hw.tar.gz
```

This will add the contents of `john_hw.tar.gz` to the folder containing other student submissions, under the subdirectory `doejohn`, and add their grade information into the report. This will integrate John Doe's submission with other student submissions, as if it had been in the original zip file.

This command cannot see if the student turned in the readme file, since readme files are turned in outside of the tar file. This script assumes the readme file is not present, if not in the tar file, and prints a warning.

6.3 Running student code

Student code is run by calling:

```
run_all.sh <hw_directory> <run_script.sh>
```

The `hw_directory` is the same as the `destination` directory in the config file (§4.3), and `run_script` is the script file you have created.

The `run_all.sh` script loops through every student directory in `hw_directory` and calls the `run_script` through `condor` within the directory. It will launch `X` `condor` commands where `X` is the number of students.

To run a single student's code, the command is:


```
run_student.sh <student_directory_path> <run_script.sh>
```

Note that `student_directory_path` must be the *full* path to the student directory, e.g. `hw4/doejohn/`.

6.4 Evaluation

After the condor scripts have all finished (you must manually check with `condor_q <username>`), you can run the evaluation component of the automatic grader. This is done by calling:

```
grader.sh <hw_config> eval_all
```

This will launch all the comparisons defined in the config file (§4.3). If any differences are found in the comparisons, a summary will be written to the report file, and diffs will be written to the student directories.

To run the evaluation on a single student, the command is:

```
grader.sh <hw_config> eval_student <student_directory_name>
```

`student_directory_name` is just the name of the student directory itself. That is, the parameter should be `doejohn`, not `hw4/doejohn`.

Note: The file comparisons can take a while, especially when calling `eval_all`, so it may be useful to wrap this command inside of a condor script of its own.

7 Code Documentation

7.1 Using Bitbucket

The grader software is managed under Bitbucket at the repository <https://bitbucket.org/davinman/grader57x>. If you wish to modify the code or check it out for yourself, you will need to create a Bitbucket account by going to bitbucket.org and request permission to edit the branch. You will also need a Bitbucket account to pull down a new version of the code and place it on Patas. I am currently using Mercurial as the management system.

Each stable version of the code is forked into its own branch. As of the publication of this document, there are three: 0.1, 1.0, and 2.0.

7.2 Code Structure

The code for the automated grader is written in Python and located inside the `grader57x` folder.

The code is separated into three files. The first is the `GradeReport.py` class, which describes the object that hosts (potentially intermediate) student grades, which are then written out to file. The second is the `Differ.py` file, which is the class that does the diff comparison between student files. The last is `grader.py`, which is the code that interprets and executes command line calls (open, evaluate, etc), and initializes and populates the `GradeReport` and `Differ` objects.

The `grader.py` code is modularized as much as possible: This includes having the `eval_all` execution iteratively call into `eval_student`, so that the `eval_student` command is guaranteed to produce the same output as `eval_all`. This includes zeroing out parts of the grade report before evaluating (this ensures that the eval scripts can be run multiple times).

Reading the config file is done in the function `read_config_file`. The code is not complex, but it is important that it align with the expected config file parameters, and that it generate any additional implicit values in the config file (for instance, summing the points in the `gold_comparison` sections).

The `GradeReport` object is initialized in two ways: the original initialization, which is done when unzipping the Canvas-generated file (under the `unzip` function), and reading the grade report on file (done under the `open_student`, `eval_all`, and `eval_single_student` functions). Generating the `GradeReport` from file is done by calling a class function within the `GradeReport` class.

8 Potential Future Developments

This section lists the most important needs for future development.

8.1 Internalizing the run script

Currently, the run script is separate from the `grader.sh` script. This is due to some difficulties Python has calling into `condor`. It would be more parsimonious if these difficulties could be overcome, and running the assignment could be called from `grader.sh` with a `run_all` and `run_student` parameter, just like the other two grading steps.

8.2 Grading special lines in the output

For some assignments, only particular lines of the output need to be graded. It would be good to add a parameter that, for certain comparison operations, would only grade lines that had a leading string in them.