Homework Guidelines for 57x

David Inman

January 1, 2018

Contents

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 a list of items for 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 to make the homework easier to grade. The development of homework descriptions is presumed to be done by the professor.

1. Descriptions of homework should be made as clear as possible.

   It is sometimes difficult to fully clarify an assignment. When writing an assignment description, 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 the following section or a separate sub-section. Try to minimize component interleaving in the description, although full separation may be impossible.

2. Homeworks should include an at-a-glance summary of turned-in files.

3. Points should be given at section headers in the homework description. These should add up to 75, the portion of the rubric assigned for homework-specific grading (§3.2). If these points are sub-divided within each section (e.g., you want to specify the number of points given for the code finishing in a period of time, or falling within an acceptable parameter), those points should be given at the top of a sub-section.

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.

   If a portion of a student's output is to be graded, that output must be uniform across all students. This means that the output must be very precisely formulated. For instance, "Epsilon arcs are followed without printing an output value (10 points)" is preferable to "Handles epsilon transitions (10 points)." Likewise, "Re-print the input string with the labeled parts of speech after each word, followed by a tab character: e.g., word1 \t POS1 \t word2 \t POS2 (...)" is better than "Print the input string with the parts of speech after each word."

6. Graded numeric values should be written to a separate output file, when possible.

   If students need to generate a table of values based on running their code, the TA has to verify the validity of it twice: In the table in the readme and in the output code. While it is worthwhile for students to collate data into a presentable format, it will make things easier to grade if the value present in the table is easily extracted from one of the output files when running the program. That is, "Table and <output file> show correct accuracy and precision trends" is preferable to "Table shows correct trends."

3 Rubric

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

assignment. This section is presumed to be done by the TAs.

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)

3.2 Homework-specific Portion (75 points)

This section cannot be schematized, but is based on the needs of each assignment. However, there are some best practices. 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 depend on evaluating the program output on test files that are as modular as possible.

   For instance, the grader should create an input that will test for 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 look for the code handling unknown words. These tests 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

   This is a file that will run the student's code. When the grader runs student files, it calls this shell script. It is possible to have multiple run files in an assignment: for example, if one needs to run within a certain time limits and others do not.

   This script will be called from the top directory of the student's untarred submit file, so all paths should be relative to that location.

   The run file(s) needs to serve two functions:

   1. It should 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, 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 should run any commands that run student code on hidden test input.

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.

Once all these files have been generated and tested on a homework, this set of files should be saved and passed on for future courses. 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 suggestion for the rest is to put the rest into a TA/instructor-only grading folder for the assignment, with the config and run files (2, 3) in the top directory, gold files (4) into a folder called `gold`, any test files (5) into a folder called `test`.

4.2 Submit List

The submit-file-list is a file that 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 full path of 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 into (for unzipping), 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. This line is optional: If it is not present, 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).

The following config lines can be repeated multiple times:

5. `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 `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`.

6. `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 output that was 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. Please note that this comparison is whitespace sensitive! If you wish to assign partial points, it may be easiest to give 0 points to this and 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.

The following config lines are for future use (see §8), and are not used by the current program.

7. `run_script = <run_hwX.sh>`

This is the place to point the config file to the `run_hwX.sh` file described in §4.1. Currently this is executed by a separate wrapper.

8. `cmd = <condor cmd file>`

`<condor cmd file>` is a generic condor file that is used to run `run_script` for all students. It should probably never be changed from the `hw.cmd` file present in the grader directory. As with #7, this is currently executed by a separate wrapper.

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 same directory as 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. This shell script calls check_hw.py with the correct parameters for the given assignment.

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 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 Automated Grader (grader.sh)

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). Most of these are run through `grader.sh`. `grader.sh` is always run with the following command:

```
grader.sh <config_file> <parameters>
```

Each stage of running the grader requires different parameters. For more on the config file, see §4.3.

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/1.0/grader57x/scripts/
```

To use another version of the grader, change the `1.0` 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`. An example is:

```
grader.sh hw1_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. An example of calling this parameter is:

```
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 command will integrate John Doe's submission with other student submissions, as if it had existed in the original zip file.

One warning about this method: It 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 does in fact take off points.

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 also the same as from the config file.

The code 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 between run output and turned-in output, and between output and gold standard, that has been defined in the config file (§4.3). If any differences are found in the comparisons, a summary of the number of differing lines will be written to the report file, and the diff itself will be written to an output file in the student directory.

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

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

Note that `student_directory_name` is the name of the student directory itself (without any containing folder). That is, the parameter should be `doejohn`, not `hw4/doejohn`.

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 two: 0.1, and 1.0.

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 two files: the GradeReport.py class, which describes the object that hosts (potentially intermediate) student grades, which are then written out to file. The other file, grader.py, is the code that interprets and executes command line calls (open, evaluate, etc), including initializing and populating a GradeReport object.

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). There is no such thing as fully self-documenting code, but I have tried to make the code as clear as possible.

The most important parts of grader.py are comparing files, reading the config file, and initializing the GradeReport object.

File comparison is fully under the compare_files() function. I use the python library difflib to diff two files, and keep track of the different lines. compare_files returns a list of lines that were different. To change how the grader does file comparison, this is the function to alter.

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 Future Grader Development

This section lists the most important needs (as I perceive them) 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.

8.3 Grading numerical values in the output

For some assignments, a numerical value must fall within a particular range. It would be good to add a parameter that looked for a number (perhaps with leading or trailing strings). The format for this should be: number, difference_percent (where the two parameters are interpreted to mean a value must fall within the range *number* $\pm$ *difference_percent*).