446-hw-base

The base for CSE 446 homeworks. This README will be copied for each homework.

If you are a TA also see README TA.

Reading Markdown files

Before you start doing anything, please make sure you have a proper way of reading markdown files. You can use any tool you would like for it. Python IDEs that we recommend for this class (VSCode or PyCharm) come with Markdown readers built-in.

Setup

You only need to do setup once, then for future homeworks you can run `conda activate cse446`.

Miniconda installation

Before you start working with this repo, you should install Anaconda.

Before clicking in the link below read notes below:

- Linux/Mac OS:
  - If using Linux/Mac please install command line version.
  - Make sure that you choose to initialize conda at startup. This will lead to fewer headaches in the future
- Windows:
  - If using Windows, you will need to use Anaconda Terminal, which uses Bash-like syntax.
- Low storage system
  - If you are low of storage (<10GB; for example attu), then Miniconda (see link below) might be a better option.

Download links

Anaconda (default)

Miniconda (if running low on disk space)

You can find more detailed instructions for installation at this link.

Environment Setup

First make sure you have at least ~5GB of free drive. From this directory run:

```
conda env create -f environment.yaml
conda activate cse446
```

First command might take long time. Especially if your connection is slow.

Then whenever you come back to work on homeworks just run: `conda activate cse446`

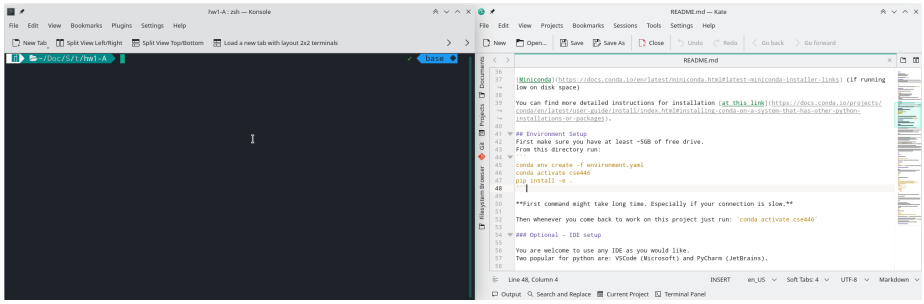

Figure 1: A quite long gif visualizing how to setup enviornment

Optional - IDE setup

You are welcome to use any IDE as you would like. Two popular for python are: VSCode (Microsoft) and PyCharm (JetBrains).

PyCharm PyCharm is tailor made for Python, so the setup should be minimal.

Just make sure you point python interpreter to the right path for the project. To do so, you should follow these instructions.

Optional: Please note, that to run linting (optional, see below) you will need to use terminal, and run `inv lint`. You can try setting up linters through *External Programs* section. If you succeed please let us know, and we will update this section for future reference :D

VSCode VSCode is general purpose IDE, so you will need to install *Python* and *Pylance* extensions (both by Microsoft).

To setup python interpreter, follow these instructions or first reload VSCode, open a folder with homework downloaded and unzipped. Click into a python file and in bottom left you should see *Python* with some version. Click into that and choose one that says "cse446".

Now whenever you open terminal it should say "(cse446)" on the left hand side, and `which python` should also output path with `cse446` in it. Sometimes, it doesn't work with the first terminal after VSCode opens, so `exit` it and reopen it (Ctrl+J). **NOTE** For further debugging see "Debugging" subsection couple of paragraphs below.

Optional: For linting you will need to *enable* flake8, mypy and *disable* pylint. Rather than following the official guide, press (Ctrl/Cmd + Shift + P), select "Preferences: Open Workspace Settings" or "Preferences: Open User Settings", in search type "flake8", "mypy" and "pylint". Each should have an option called "Python > Linting: -linter- Enabled" which you should toggle *on* for flake8 and mypy and toggle *off* for pylint.

Extra Optional: Extensions for VSCode and quite helpful apart from support for specific language. I highly recommend downloading Trailing Spaces by Shardul Mahadik and Visual Studio IntelliCode by Microsoft. Also if you are advanced LaTeX user there is LaTeX Workshop by James Yu and LaTeX Utilities by tecosaur. These require you to download and install LaTeX manually though.

Debugging If `which python` doesn't return path with `cse446` in it, and restarting terminal doesn't work, there might be an issue with settings in VSCode. It may be because of VS Code's setting, namely the inherited environment of the integrated terminal ("Terminal > Integrated: Inherit Env"). Turn it off, and reopen VS Code to see if the issue has been resolved.

Usage

Submission

When you are done with coding run.

```
inv submit
```

this will generate a `.zip` file that should be uploaded to gradescope for automated grading.

For example running:

```
inv submit
```

will result in `submission_<timestamp>.zip` generated, which should be submitted under corresponding homework coding assignment on gradescope.

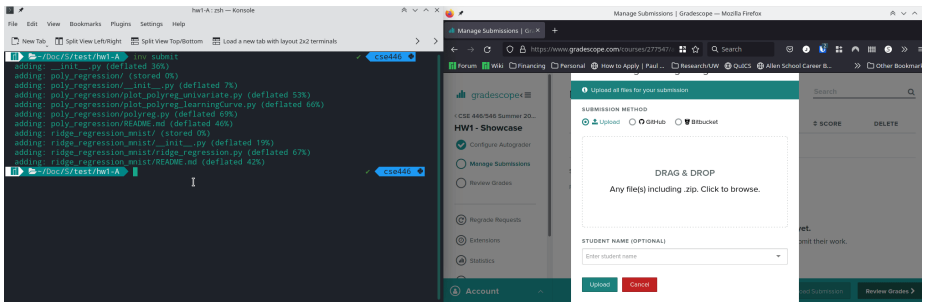

Figure 2: gif visualizing submission process

Note that if you do get a lot of `NotImplementedError("Your Code Goes Here")` errors when submitting to gradescope you .zip file might be in an improper format which makes autograder unable to detect it. **MAKE SURE TO USE `inv submit` if that happens.

Testing

In this class we will use unittest framework in python to automatically grade coding problems. Some of the tests are provided to you, so that you can validate your results.

To run tests:

```
inv test
```

The output should look something like this:

```
> inv test

FFF..
======================================================================
FAIL: test_polyfeatures_fives (public.poly_regression.test_poly_regression.TestPolyReg)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ...
AssertionError:
Arrays are not almost equal to 6 decimals

(shapes (1,), (20, 1) mismatch)
 x: array([1.])
 y: array([[5.],
       [5.],
       [5.],...
```

You can see that in the top there are 3 `F`'s and 2 `.`'s. `F`'s correspond to failed tests and `.` correspond to correct tests.

There are few things to note:

- Not all tests are equal. Some are worth more points. This will not be displayed when you run `inv test`.
- We **do not** provide you with all tests. There are many that hidden. Even if you pass all *public* tests you may still fail some *hidden* ones.

Testing specific problem

Unfortunately `unittest` framework doesn't allow for testing specific file. However, you can run tests against specific problem. To do so run:

```
inv test --problem <problem-name>
```

For example:

```
> inv test --problem poly_regression

test_fit_and_predict_cubic (test_poly_regression.TestPolyReg) ... ok
test_fit_and_predict_straight_line (test_poly_regression.TestPolyReg) ... ok
test_fit_cubic (test_poly_regression.TestPolyReg) ... ok
test_fit_hard (test_poly_regression.TestPolyReg) ... ok
test_fit_linear (test_poly_regression.TestPolyReg) ... ok
test_fit_straight_line (test_poly_regression.TestPolyReg) ... ok
test_mean_squared_error (test_poly_regression.TestPolyReg) ... ok
test_polyfeatures_fives (test_poly_regression.TestPolyReg) ... ok
test_polyfeatures_ones (test_poly_regression.TestPolyReg) ... ok
test_polyfeatures_twos (test_poly_regression.TestPolyReg) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.197s

OK
```

Linting

Linting is **not required** for this class. However, we recommend that you keep your code linted. Especially if this is your first time learning python, or if you never had a formal introduction.

For this we provided you with a simple script to track issues & possibly fix your code. To run linter:

```
inv lint
```

which will generate output

```
> inv lint
flake8 homeworks
homeworks/hw1/poly_regression/polyreg.py:103:5: E303 too many blank lines (4)
```

You can see that issue points to file (`polyreg.py`), line (105) and column (3) as well as the issue "too many blank lines".

Note that we are using 4 linters (flake8, isort, black, mypy; in that order). If you have issues that are pointed out by isort or black you can run `inv lint --apply` to automatically apply fixes.

Again, linting is **not required** for this class. If you don't want to lint your code, or prefer some other linting setup that's ok.

Kernels & Bootstrap

In this problem you will implement two kernels: Polynomial and Radial Basis Function (RBF). For each one you will: 1. Search with cross-validation to find optimal parameters 2. Plot trained function 3. Plot trained function's 5th and 95th percentiles obtained using bootstrap.

Lastly you will compare errors of RBF and polynomial kernels using bootstrap, and discuss whether one is better or not.

We recommend implementing functions in order they are provided in a file. The only exception is that you should also modify `main` function as you keep implementing other functions and solving parts A, B and C. Part D of the problem can be solved by putting A, B and C in a for-loop, while part E is quite similar to bootstrap function.

After finishing assignment run the code (for example `python homeworks/kernel_bootstrap/main.py`) and in `.pdf` submit plots that you have generated.

Binary Log Regression

In this problem you will implement logistic regression in a two class setting. Start by look at file binary_log_regression.py. You will have to implement `BinaryLogReg` class along with all of it's functions. We advise you to start with the constructor (`__init__`) and then move to any of the `mu`, `loss`, `gradient_J_weight`, `gradient_J_bias`, `predict`. Once you are finished implementing these functions you should implement `misclassification_error` and `step`. Finally, once all other functions are implemented you should write `train` function.

Note that all functions but `train` are unit tested, so that you can verify your implementation.

To run the problem do: `python homeworks/log_regression/binary_log_regression.py` from the root directory of provided zip file, and then save and add the plots to your written submission.

Neural Networks for MNIST

In this problem you will build 2 models and train them on MNIST dataset. Start by having a look at main.py file. You will see that there are two classes (`F1` and `F2`) as well as two functions (`train` and `main`) that you have to implement. We recommend implementing them in order of `F1`, `F2`, `train` and `main`.

To run the problem do: `python homeworks/neural_network_mnist/main.py` from the root directory of provided zip file, and then save and add the plots to your written submission.

Two things to note: 1. Do not use anything from torch.nn other than what is already imported. 2. There are no public tests for this problem due to number

of possible implementations. You can 3. You can find that you are not using everything imported in the `main.py` file. It is ok, not everything is necessary to finish the problem. However, they might make your code easier to read and debug.