MSDS 621 - Introduction to Machine Learning

Homework 1

Robert Clements

1 Linear and Logistic Regression Models and Gradient Descent

1.1 Turning it in

We will use Github classroom to submit assignments. For this assignment you must submit one file: - linreg.py file

1.2 Using Gradient Descent to Implement Linear Regression, Linear Ridge Regression and Logistic Regression

For this homework you will build your own library to train regression models using gradient descent. Specifically you will focus on building classes, similar to `sklearn`, that you can use to train the following three types of models:

- linear regression without regularization
- linear regression with L2 regularization
- logistic regression **without** regularization

To be able to train these models, you will need to remember these functions:

- loss function for linear regression
- loss function for ridge (linear) regression
- loss function for logistic regression
- the derivative for each of the above

Training these models means that you are finding the values of β such that you minimize the **loss function**. Recall that the loss function is a little bit different for each of the three models above. For regression it is the sum of the squared residuals. For logistic regression it is the negative-log-likelihood. For regularized regression it is the sum of the squared residuals plus the shrinkage penalty (L2). The smaller the loss, the better the model is fitting, meaning the closer the predicted values of y are to actual observed values of y.

1.2.1 Linear Regression without regularization

To train a linear regression model you need only the gradient of the loss function, and then you can implement gradient descent using an initial random value of β. Recall that the loss function can be written like this:

$$\mathcal{L}(\beta) = (y - X\beta)^T(y - X\beta)$$

where β is vector

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \dots \\ \beta_p \end{bmatrix}$$

and X is a matrix where the first column is a column of 1s. Although we know that linear regression has a closed form solution, we are still going to solve it iteratively using gradient descent. For that, we need the gradient, which is given here:

$$\nabla\mathcal{L}(\beta) = -X^T(y - X\beta)$$

1.2.2 Linear Regression with regularization

For linear regression with L2 regularization, all we need to do is change the loss function ever-so-slightly, which changes the gradient as well. The resulting loss function will minimize the sum of the squared residuals **plus** $\lambda\sum_{j=1}^{p}\beta_j^2$. Note that for linear regression with L2 regularization we **do not add a column of 1s** to X, and we estimate β_0 separately as $\bar{y}$.

1.2.3 Logistic Regression without regularization

For logistic regression without regularization we need a brand new loss function, which is the negative-log-likelihood, and its gradient. Note here that we **will add a column of 1s** to X to estimate β_0.

1.2.4 Use Adagrad

For this assignment use the **Adagrad** version of gradient descent. The algorithm that you need to code up is as so:

minimize$(X, y, \nabla\mathcal{L}, \eta, \epsilon$=1e-5, precision=1e-9)

- Let $\beta \sim 2N(0,1) - 1$ (random vector of sized p+1 (or p if excluding the intercept) with elements in $[-1, 1)$)
- $h = 0$ (sum of squared gradient history to be stored here, this is a vector of sized p+1 or p depending on if the intercept is excluded)
- $X = (1, X)$ (add first column of 1s if including the intercept)
- **repeat next two steps until** $||\nabla\mathcal{L}(\beta)||_2 \leq precision$
  - h = sum of squared partial derivatives up to, and including, the current partial derivative of the current step you are taking
    - Note that h is a vector, and that each element of h is the sum of the historical partial derivatives (squared) for each β you are estimating
  - $\beta = \beta - \frac{\eta}{\sqrt{h+\epsilon}}\nabla\mathcal{L}$
- **return** β

1.2.5 Confusion Point

You will notice that there are multiple `predict()` and `fit()` methods in the various classes. Despite having the same name, the methods exist in different class definitions. You can think of class definitions in this context as separate modules that package up a set of functions. They have different implementations, because they do different things depending on the surrounding class, but they have the same name. For example the full name of `predict()` in class `LinearRegression621` is `LinearRegression621.predict()`, which clearly differentiates it from `LogisticRegression621.predict()`.

1.2.6 Recommendations

(1) Some students get good solutions but can't get it to finish within the required 10 seconds. Look for repeated computations of expensive things like the loss gradient. By analogy, don't call f(5) many times. Call it once and save the result. Make sure that you use as many numpy vector operations as possible. Loops in Python can be quite slow. You can use the Python profiler to identify which functions take the most amount of time, which often helps to isolate speed problems. Note that the time requirements are running on

your own machine, not via github actions. (2) When it comes to debugging, you can't just look at the code and the math and declare it should work. You have to try a small known data set and then examine what the coefficients should be; compare those to the output of your optimizer. Track down where the difference comes from if any.

1.2.7 Getting Started

I recommend that you play around with the following 1D $y = (x-2)^2$ example and make sure you fully understand how it works. This will make it easier to expand into $p+1$ space. Also take a look at our gradient descent activity from class.

```
import numpy as np

def f(b) : return (b-2)**2
def gradient(b): return 2*(b-2)

b = np.random.random()   # initial guess at optimal b
niter = 7
rate = .01
for i in range(niter):
    print(f"{i:02d}: beta_1={b:.2f}, f(beta_1)={f(b):.2f}, gradient {gradient(b):.2f}")
    b = b - rate * gradient(b)
```

```
00: beta_1=0.43, f(beta_1)=2.48, gradient -3.15
01: beta_1=0.46, f(beta_1)=2.38, gradient -3.08
02: beta_1=0.49, f(beta_1)=2.28, gradient -3.02
03: beta_1=0.52, f(beta_1)=2.19, gradient -2.96
04: beta_1=0.55, f(beta_1)=2.11, gradient -2.90
05: beta_1=0.58, f(beta_1)=2.02, gradient -2.85
06: beta_1=0.61, f(beta_1)=1.94, gradient -2.79
```

The above terminates after a certain number of iterations, but we would want to continue until the gradient (or norm of the gradient) is zero or the β's or loss don't change from step to step. **An easy and efficient termination condition is to stop when the norm of the gradient vector is below some threshold.**

Once you've built the mechanism for minimizing a loss function, you can use it to optimize the regularized versions of linear regression as well as the unregularized logistic regression.

To actually begin the project, create `linreg.py` in your repo and define just the functions you will need to get basic linear regression working: `minimize()` and `loss_gradient()`. Try to debug your code using a single $x1$ vs y regression. If that works, then try with $x = (x1, x2)$. If

it fails, then you know the problem is in how you handle multiple dimensions, rather than the core of your optimization loop.

Once you have your minimizer working, add the `LinearRegression621` class and then try to get the unit test working. Please note that the starter kit has a number of defined functions in the classes that are used by the unit tests. Please follow that API.

This project looks complicated because of all the math notation, but all you are really doing is converting math notation into Python numpy operations. If you don't use numpy vector operations (like dot product), your code will be too slow and will get a grade penalty.

1.2.8 Deliverables

You should implement the following in `linreg.py`:

- `loss_gradient(X, y, B, lmbda)`
  - $\nabla\mathcal{L}(\beta) = -X^T(y - X\beta)$
- `loss_gradient_ridge(X, y, B, lmbda)` for $\beta_1, \ldots \beta_p$
  - $\nabla\mathcal{L}(\beta) = -X^T(y - X\beta) + \lambda\beta$
- `sigmoid(z)`
- `log_likelihood_gradient(X, y, B, lmbda)`
  - $\nabla\ell(\beta) = -X^T(y - \sigma(X\beta)$
- `minimize(X, y, loss_gradient, eta, lmbda, ...)`

Notice that we are passing a λ parameter to all loss and loss gradient functions. That way, the same minimize function can work with all of them. The only difference between the various regression techniques will be the loss function that is passed to the minimize function!

Next you have to implement two classes that mimic how sklearn's models work:

- class `RidgeRegression621`
- class `LogisticRegression621`

I provide you with `LinearRegression621` and you can use that to help you fill in the code for the other two classes.

1.2.9 Evaluation

Your project will be evaluated using these predefined tests: `test_regr.py` and `test_class.py`. To run these unit tests locally (on your laptop), open a Terminal and change directories to the location of your `linreg.py` and `test_regr.py` and `test_class.py`, and then run

```
python -m pytest -v test_regr.py
python -m pytest -v test_class.py
```

Please do not modify these unit test files. Here is the output that I get from my solution:

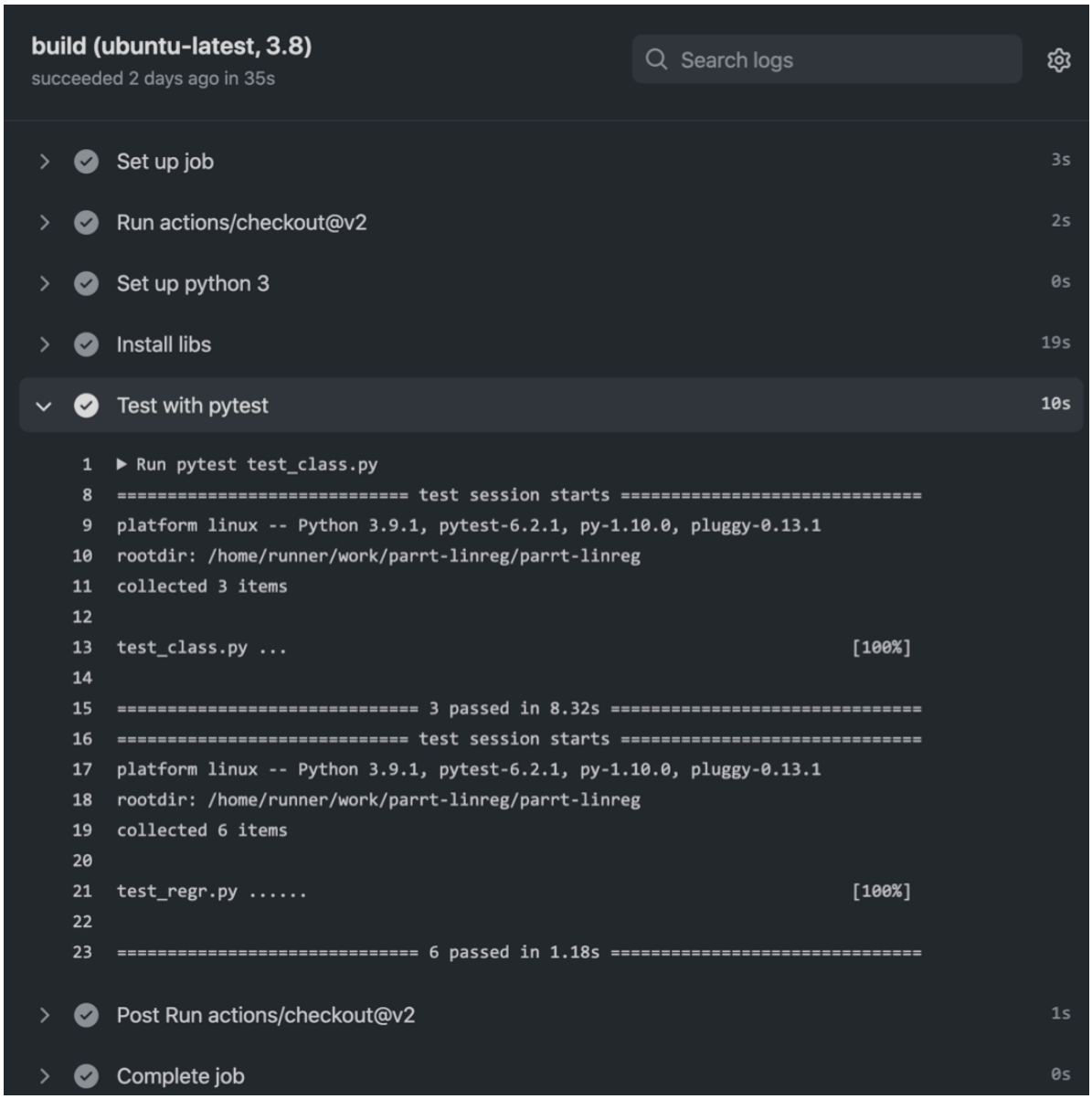
```
(base) rclements@ML-ITS-211472 linreg % python -m pytest -v test_regr.py
============================================================================= test session starts ==============================================================================
platform darwin -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0 -- /opt/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/rclements/Documents/teaching/msds621_2022/projects/linreg
plugins: anyio-3.5.0
collected 6 items

test_regr.py::test_synthetic PASSED                                                                                                                                      [ 16%]
test_regr.py::test_ridge_synthetic PASSED                                                                                                                                [ 33%]
test_regr.py::test_boston PASSED                                                                                                                                         [ 50%]
test_regr.py::test_boston_noise PASSED                                                                                                                                   [ 66%]
test_regr.py::test_ridge_boston PASSED                                                                                                                                   [ 83%]
test_regr.py::test_ridge_boston_noise PASSED                                                                                                                             [100%]


(base) rclements@ML-ITS-211472 linreg % python -m pytest -v test_class.py
============================================================================= test session starts ==============================================================================
platform darwin -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0 -- /opt/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/rclements/Documents/teaching/msds621_2022/projects/linreg
plugins: anyio-3.5.0
collected 3 items

test_class.py::test_synthetic PASSED                                                                                                                                     [ 33%]
test_class.py::test_wine PASSED                                                                                                                                          [ 66%]
test_class.py::test_iris PASSED                                                                                                                                          [100%]

============================================================================== 3 passed in 8.35s ===============================================================================
```

For full credit, you must get all 9 unit tests passing *most of the time* (at least 90%). If any test fails more often, then there is most likely still something wrong with your code. I suggest that you run your tests both locally **and** with github actions (described in the next section) so that you can verify your code works on a different computer. Late projects get a zero grade. The failure of the unit tests will be used as a guide to determine your points. In general, each failed test may result in minus 10 points.

Your unit tests should run in under 10 seconds **on your machine**. Do not use the timing from Github actions, it usually takes a little longer to run there. If either, or both of them, do not run in under 10 seconds, then this will result in minus 10 points. The grader will pull your repositories and run the unit tests on their machine in order to evaluate them.

1.2.9.1 Github Actions

With Github Actions every push to the repository on github from your laptop triggers the unit tests. All you have to do is put the test.yml file I have prepared for you into repo subdirectory .github/workflows, commit, and push back to github. Then go to the Actions tab of your repository and you'll see something like this:



Naturally it will only work if you have your software written and added to the repository. Once you have something basic working, this functionality is very nice because it automatically shows you how your software is going to run on a different computer (a linux computer). This will catch the usual errors where you have hardcoded something from your machine into the software. It also gets you in the habit of committing software to the repository as you develop it, rather than using the repository as a homework submission device.