Assignment_3

March 31, 2025

version 1.1

194.025 Introduction to Machine Learning Assignment 3: Regression & Gradient Descent

---

Welcome to the third assignment of our course **Introduction to Machine Learning**. You will be able to earn up to a total of 10 points. Please read all descriptions carefully to get a full picture of what you have to do.

Remark: Some code cells are put to read-only. Please execute them regardless as they contain important code. You can run a jupyter cell by pressing `SHIFT + ENTER`, or by pressing the play button on top (in the row where you can find the save button). Cells where you have to implement code contain the comment `# YOUR CODE HERE` followed by `raise NotImplementedError`. Simply remove the `raise NotImplementedError`and insert your code.

Some other code cells start with the comment `# hidden tests ...`. Please do not change them in any way as they are used to grade the tasks after your submission.

0.1 Implement Gradient Descent for Linear Regression (4 Points)

Implement the algorithm on the slide titled 'Gradient Descent' of the 'Basic Algorithms I' slides.

Ensure that you implement the algorithm using exactly the interface described below.

`def gradient_descent_linear_regression(X, y, initial_w, learning_rate, epochs)`

Note:

- To be on the safe side with our auto grading tool, please do not start your names for variables, functions, etc. with `solution_`
- Be careful about integer/ float division in python

[216]:
```python
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
```

[217]:
```python
# a data sample without noise to toy around, you may use different data

X = np.linspace(10, 20, 50)
y = 2.5 * X + 4.0
```

[231]:
```python
def gradient_descent_linear_regression(X, y, initial_w, learning_rate, epochs):
    '''
    X: 1d numpy array containing floats
    y: 1d numpy array containing floats
    initial_w: 1d numpy array containing exactly two float values [w_0, w_1]
               that are used to initialize w_0 and w_1
    learning_rate: float
    epochs: int

    expected output: 1d numpy array containing exactly two float values [w_0,␣
 ↪w_1]
    '''
    # YOUR CODE HERE
    raise NotImplementedError()
```

[232]:
```python
# hidden tests - DO NOT CHANGE THIS CELL
```

[233]:
```python
# hidden tests - DO NOT CHANGE THIS CELL
```

[234]:
```python
# hidden tests - DO NOT CHANGE THIS CELL
```

[222]:
```python
# hidden tests - DO NOT CHANGE THIS CELL
```

0.2 Comparison to Closed Form Linear Regression (2 Points)

Luckily, we know that we can compute w_1 and w_0 (more or less) exactly using the 'Closed Form Solution' equalities.

Implement 'Some Python Code' to compute w_1 and w_0 for the data set given in the next cell.

Fix the learning rate for your gradient descent algorithm to `learning_rate=0.000001`

How many epochs do you need to get the parameters right up to an error of `0.1` when starting at `initial_w=np.array([2,12])`?

Assign your answer to the variable `reply_number_of_epochs`

Is this surprising to you?

Hint 1: You can check if all values of two arrays `a` and `b` are at least x apart with

`np.allclose(a,b, atol=x, rtol=0)`

Hint 2: Looping over all values might not be the fastest solution.

[223]:
```
X2 = np.array([-1., -0.95918367, -0.91836735, -0.87755102, -0.83673469, -0.
↪79591837, -0.75510204, -0.71428571, -0.67346939, -0.63265306, -0.59183673,␣
↪-0.55102041, -0.51020408, -0.46938776, -0.42857143, -0.3877551, -0.34693878,␣
↪-0.30612245, -0.26530612, -0.2244898, -0.18367347, -0.14285714, -0.10204082,␣
↪-0.06122449, -0.02040816, 0.02040816, 0.06122449, 0.10204082, 0.14285714, 0.
↪18367347, 0.2244898, 0.26530612, 0.30612245, 0.34693878, 0.3877551, 0.
↪42857143, 0.46938776, 0.51020408, 0.55102041, 0.59183673, 0.63265306, 0.
↪67346939, 0.71428571, 0.75510204, 0.79591837, 0.83673469, 0.87755102, 0.
↪91836735, 0.95918367, 1.])
y2 = np.array([-16.01888673, -15.06215775, -14.66563224, -13.98386833, -13.
↪15288414 , -12.33027782, -11.92996315, -10.80118307, -10.51458662, -9.
↪65801659 , -9.00459352, -8.46469268, -7.25778365, -6.50155011, -6.38382577 ,␣
↪-5.69013565, -5.07691771, -4.64218747, -3.56640217, -2.53184116 , -2.
↪01569423, -1.09869641, -0.87069945, 0.71259073, 0.90882325 , 1.37826874, 2.
↪23373936, 2.81463612, 4.03562475, 3.63696536 , 4.13601676, 5.68708623, 6.
↪04695401, 7.2627373, 7.74624043 , 8.78626548, 8.50347204, 10.48395259, 9.
↪86322328, 10.84944206 , 11.58472135, 12.46780085, 13.21365486, 13.7756522,␣
↪14.59631027 , 15.84471266, 15.82315378, 16.15334549, 17.36419808, 17.
↪79100676])
```

[224]:
```
reply_number_of_epochs = 1  # this is certainly not enough ;)

# overwrite the variable (use the same name) with a value of your choice)

# YOUR CODE HERE
raise NotImplementedError()
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[224], line 6
      1 reply_number_of_epochs = 1  # this is certainly not enough ;)
      3 # overwrite the variable (use the same name) with a value of your choice)
      4
      5 # YOUR CODE HERE
----> 6 raise NotImplementedError()

NotImplementedError:
```

[225]:
```
# hidden tests - DO NOT CHANGE THIS CELL
```

0.3 Polynomial Regression (4 Points)

Implement polynomial regression for one-dimensional input, but arbitrary degree, as described on the slide 'What About Polynomial Regression, then?!?'. That is, the user of your method should be able to specify the degree of the polynomial to fit as a parameter. Note that a polynomial of degree p has $p+1$ entries in the vector $\mathbf{w}$. The polynomial should look as follows:

$$h(x) = \sum_{k=0}^{p} w_k \cdot x^k \,.$$

Also implement the function that, given a weight array `np.array([w_0, w_1, ..., w_p])` can compute the function values of the corresponding polynomial.

Ensure that you implement the algorithm using exactly the interface described below.

[189]:
```
import numpy as np
import numpy.linalg as la
```

[226]:
```
def poly_regression_fit(X, y, p):
    '''
    X: 1d numpy array containing floats
    y: 1d numpy array containing floats
    p: a nonnegative integer, giving the degree of the polynomial

    expected output: 1d numpy array containing the float values [w_0, w_1, ...,␣
 ↪w_p]
    '''
    # YOUR CODE HERE
    raise NotImplementedError()
```

[227]:
```
def poly_regression_transform(X, w):
    '''
    X: 1d numpy array containing floats
    w: 1d numpy array containing the float values [w_0, w_1, ..., w_p]

    expected output: 1d numpy array, same shape as X, containing function values
    '''
    # YOUR CODE HERE
    raise NotImplementedError()
```

[228]:
```
# hidden tests - DO NOT CHANGE THIS CELL
```

[229]:
```
# hidden tests - DO NOT CHANGE THIS CELL
```

[230]:
```
# hidden tests - DO NOT CHANGE THIS CELL
```

[ ]:
```
```

[ ]:
```
```