Queen Mary
University of London

MTH786U/P, Semester C, 2020/21

Assignment 5 **M. Poplavskyi**

This week marks a mixture of theoretical and practical coursework.

Smoothed modulus function

As we have seen in the lecture, the modulus/absolute value function is an important element of the LASSO problem. However, due to its non-differentiability, it is a main reason of new optimisation methods being introduced in the module. In this exercise we consider one of them, namely changing the modulus with its smooth version

1. Prove that for any $x \in \mathbb{R}$ one has $|x| = \max\limits_{p\in[-1,1]} xp$.
2. Let us now take any $\tau > 0$ and introduce the smoothed modulus function $|x|_\tau$ via
$$|x|_\tau = \max_{p\in[-1,1]} xp - \frac{\tau}{2}p^2.$$
Show that the smoothed modulus function has the closed-form solution
$$|x|_\tau = \begin{cases} |x| - \frac{\tau}{2} & |x| > \tau \\ \frac{1}{2\tau}|x|^2 & |x| \le \tau \end{cases}.$$
3. Sketch the plot of $|x|_\tau$.
4. For any vector $\mathbf{w} \in \mathbb{R}^n$ we define its Huber loss as
$$H_\tau(\mathbf{w}) = \sum_{j=1}^{n} |w_j|_\tau\,,$$
and its soft-tresholding function as
$$\mathrm{soft}_\tau(\mathbf{w}) = \mathbf{w} - \tau \cdot \nabla H_\tau(\mathbf{w})\,.$$
Evaluate $\mathrm{soft}_\tau(\mathbf{w})$ explicitly.

Proximal maps

Another approach to deal with the LASSO we will meet in the module is a so-called proximal gradient descent. It is an iterative method of solving the problem

$$\hat{w} = \arg\min_{w} \{L(w) + R(w)\},$$

where $E(w)$ is a differentiable, convex function, while R is just a convex function, continuous function that may be non-differentiable (such as modulus function for example). The main tool of the method are the proximal map prox_R defined as

$$\mathrm{prox}_R(z) = (I + \partial R)^{-1}(z) = \arg\min_{x\in\mathbb{R}^n} \left[\frac{1}{2}\|x - z\|^2 + \tau R(x)\right].$$

This maps the variable z to the minimiser of the above function.

1. Let $n = 1$, that means $R : \mathbb{R} \to \mathbb{R}$ is a one-dimensional function of one-dimensional argument and
$$\mathrm{prox}_R(z) = \arg\min_{x\in\mathbb{R}} \left[\frac{1}{2}(x - z)^2 + \tau R(x)\right].$$
Find the proximal map for
   - $R(x) := x^2$
   - $R(x) := \alpha |x|$
     Hint: if z is your input argument, make the assumption $x = \lambda z$ for your solution of the proximal map, for a scalar $\lambda \in \mathbb{R}$.
   - $R(x) := \begin{cases} 0 & x \in [-1, 1] \\ \infty & x \notin [-1, 1] \end{cases}$
   - $R(x) := \begin{cases} 0 & x \in \mathcal{C} \\ \infty & x \notin \mathcal{C} \end{cases}$, for some convex $\mathcal{C} \subset \mathbb{R}$.
2. Compute the proximal map for $R(x) := \frac{1}{2}\|Dx\|^2$ for some matrix $D \in \mathbb{R}^{m\times n}$.
3. Write the proximal map for $R(x) := aS(x - y) + b$, for $y \in \mathbb{R}^n$, constants $a, b \in \mathbb{R}$ with $a > 0$ and a convex function S in terms of the proximal map of S.
4. Write the proximal map for a function $R : \mathbb{R}^n \to \mathbb{R}^n$ defined as $R(x) := S(x) + \alpha\|x\|^2 + \langle x, y\rangle$, for fixed $y \in \mathbb{R}^n$ and $\alpha > 0$, in terms of the proximal map of S.

In the following 2 exercises you will be asked to implement K-fold cross-validation and the gradient descent methods on a given data. The assignment folder contains a helper file (external module) called `MLHelper.py`. You can import it and use any function from it by calling

[1]:
```
#import the module
import MLHelper as ml
#use a method _name_of_method with parameters _params_
ml._name_of_method(_params_)
```

Before we proceed to the problems themselves we will learn how to prepare the data.

Standardisation and de-standardisation of the data

The goal of applying *standardisation* is to make sure different features of objects are on almost the same scale so that each feature is equally important and make it easier to process by most ML algorithms. The result of standardisation is that the features will be rescaled to ensure the mean and the standard deviation to be 0 and 1, respectively. This means that for a data given by $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ we define a new, rescaled data as:

$$\hat{x}_i = \frac{x_i - \langle \mathbf{x} \rangle}{\sigma_{\mathbf{x}}},$$

where $\langle \mathbf{x} \rangle = \frac{1}{n} \sum_{j=1}^{n} x_j$, and $\sigma_{\mathbf{x}} = \sqrt{\frac{1}{n} \sum_{j=1}^{n} (x_j - \langle \mathbf{x} \rangle)^2}$ are the mean and standard deviation of $\mathbf{x}$.

1. Write two functions `standardise` and `de_standardise` to (de-)standardise the columns of a multi-dimensional array. The function `standardise` takes the multi-dimensional array `matrix` as its input argument. It subtracts the means from each column and divides by the standard deviations. It returns the standardised matrix, the row of means and the row of standard deviations. The function `de_standardise` reverses this operation. It takes a standardised matrix, the row of means and the row of standard deviations as its arguments and returns a matrix for which the standardisation process is reversed.

Model selection: cross validation

In this exercise you will work with a real housing price data. The assignment folder contains `house_prices.csv` file which you will need to read the data from. This file contains the information about $N = 1200$ houses. The data columns are:

- `StreetLength` - length of the street in front of the building

- `Area` - total area of the lot
- `Quality` - quality of building materials
- `Condition` - condition of the building
- `BasementArea` - area of the basement
- `BasementArea` - area of the basement
- `LivingArea` - total living area
- `GarageArea` - a garage area
- `SalePrice` - sale price

Your task would be to build a ridge regression using K-fold cross validation strategy for validation and a grid search strategy for optimisation over hyperparameter α.

1. Download the information of numerous houses from the `houses_prices.csv` dataset. In order to do so, download `MLHelper.py` from the QM+ module page and import the `load_housing_data` function from the module `MLHelper`. Use this function to load the data via the command `housing_data, housing_prices = load_housing_data()`. Standardise the data using the function `standrdise` you have written before.

2. Use a function `ridge_regression` available from the module `MLHelper` that takes three arguments `data_x`, `data_y`,and `alpha`, which computes and returns the solution w_α of the normal equation
$$\left(X^\top X + \alpha I\right) w_\alpha = X^\top y.$$
Here X is the mathematical representation of `data_x` and y is the mathematical representation of `data_y`.

3. Define a lambda function `linear_regression` that takes two arguments `w` and `x` and evaluates $f_w(x)$. Define another lambda function `error_measure` that takes two numeric arguments `f` and `y` and evaluates $(f - y)^2$. Finally, implement a function `validation_error` that takes 3 parameters: `validation_x`, `validation_y`, `w` and evaluates the validation error
$$\mathrm{Val}_{S_v}(w) = \frac{1}{2|S_v|} \sum_{(x_i, y_i) \in S_v} |f_w(x_i) - y_i|^2 \,,$$
between output samples $\{y_i^v\}_{i=1}^{|S_v|}$ (represented by `validation_y`) from a validation data set S_v and the outputs of the ridge regression function f_w for matching inputs $\{x_i^v\}_{i=1}^{|S_v|}$ (represented by `validation_x`) from S_v.

4. Implement a K-fold cross validation strategy as introduced in the lectures. Write a function `data_split` that takes arguments `data` and `K` and splits the data randomly into K equal (or almost equal) chunks. Write a function `KFold_validation_error` that takes arguments `data_x`, `data_y`, `alpha`, `K`, and
   - using the function `data_split`, splits the data into K chunks;

- uses $K-1$ sets to compute the ridge regression weights w_α and the remaining set as the validation set S_v for the calculation of an error;
- repeats this for all combinations, and average your results;
- returns the average regression coefficients $\langle w_\alpha \rangle$ and the corresponding validation error $\mathrm{Val}_{S_v}(\langle w_\alpha \rangle)$.

5. Implement a grid search algorithm to find an unknown parameter $\hat{\alpha}$ such that

$$\hat{\alpha} = \arg\min_{\alpha \geq 0} \mathrm{Val}_{S_v}(\langle w_\alpha \rangle) .$$

Gradient descent

The goal of this exercise is to minimise the mean squared error via gradient descent. Please follow the following steps.

1. Download and visualise the height- and weight-information of numerous individuals from the `height-weight-genders.csv` dataset. In order to do so, import the `load_measurements_data` function from the file `MLHelper.py`. Use this function to load the data via the command `height, weight, _ = load_measurements_data()`. Sort the data points in terms of their height for later visualisations. Visualise the data points with your favourite tools from the Matplotlib library.
2. Standardise the height and weight arrays from Question (1). Use the standardised height array to create a polynomial basis matrix of degree one. You can use your methods from the previous coursework, or use the functions provided in the file `MLHelper.py`. Compute the optimal weights by performing linear regression as you did in the previous coursework. A function `regression` is also provided with the file `MLHelper.py`.
3. Visualise your results from Question (2). Apply the forward model to your optimal weights and use the `de_standardise` function from Question (1). Plot your results together with the data points.
4. Write two functions `mean_squared_error` and `mean_squared_error_gradient` that implement the mean squared error and its gradient as defined in the lecture notes. Both functions take a two-dimensional NumPy array `data_matrix`, a one-dimensional NumPy array `weights` and a one-dimensional NumPy array `outputs` as arguments .
5. Implement a function `gradient_descent` that performs gradient descent to numerically approximate a minimiser of a convex function `objective` with gradient `gradient`. In addition to the functions `objective` and `gradient` that take a NumPy array as their argument, the arguments for this function are a NumPy array `initial_weights` with initial values for the first iterate, a step-size parameter `step_size` for the gradient descent step and a parameter `no_of_iterations` that controls the number of iterations. Implement the function so that it returns a NumPy array of the weights obtained after gradient descent together with a list of objective values for all iterates.

6. Use the function `gradient_descent` from Question (5) to minimise the mean squared error (and thus, to compute a solution to the linear regression problem). Experiment with different initialisations and parameters. How should the step-size parameter be chosen? Compare your result to the reference-solution from Question (2).
   Hint: make use of your functions from Question (4) and Python's lambda function.