1 Linear Regression via Gradient Descent

In this exercise, you will investigate multivariate linear regression using gradient descent.

1.1 Short theory

At a theoretical level, *Gradient Descent* is an algorithm that minimizes functions. Given a function defined by a set of parameters $\underline{\theta}$, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize a cost function. This iterative minimization is achieved using calculus, taking steps in the negative direction of the function gradient.

We define the hypothesis function that acts on a sample $\underline{x}_i = (1\, x_{i1} \ldots x_{im})^T$ as

$$h(\underline{x}_i) = \underline{\theta}^T \underline{x}_i = \underline{x}_i^T \underline{\theta} = \theta_0 + \sum_{j=1}^{m} \theta_j x_j,$$

where $\underline{\theta} = (\theta_0, \theta_1, \ldots, \theta_m)^T$ is the vector with the parameters. Given the above hypothesis function, let us try to figure out the parameters $\underline{\theta}$, which minimizes the square of the error between the predicted value $h(\underline{x})$ and the actual output y for all samples $i = 1, 2, \ldots, n$ in the training set. For that reason, let us define the cost function as

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n} \sum_{i=1}^{n} \left(h\left(\underline{x}_i\right) - y_i\right)^2, \tag{1}$$

where n is the the number of training set. The scaling by fraction $\frac{1}{2n}$ is just for notational convenience. The cost function can also be written in the following form using a matrix notation,

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n} \left(\mathbf{X}\underline{\theta} - \mathbf{y}\right)^T \left(\mathbf{X}\underline{\theta} - \mathbf{y}\right)$$

where the $n \times 1$ response vector $\mathbf{y}$ is

$$\mathbf{y} = \left(y_1, y_2, \ldots, y_n\right)^T,$$

and the $n \times (m+1)$ design matrix $\mathbf{X}$ is given by

$$\mathbf{X} = (\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_n)^T,$$

or, equivalently,

$$\mathbf{X} = \left(\mathbf{1}_{n\times 1}, \mathbf{x}_1^T, \ldots, \mathbf{x}_j^T, \ldots, \mathbf{x}_m^T\right),$$

where

$$\mathbf{x}_j = (x_{1j}, x_{2j}, \ldots, x_{ij}, \ldots, x_{nj}), j = 1, 2, \ldots, m.$$

The matrix form of the equations is useful and efficient when you're working with numerical computing tools like MATLAB or numpy. If you are familiar with matrices, you can prove to yourself that the two forms are equivalent.

Let us start with some parameter vector $\underline{\theta} = \underline{0}$, and keep changing the $\underline{\theta}$ to reduce the cost function $\mathcal{J}(\underline{\theta})$, i.e.,

$$\theta_j^{\text{new}} = \theta_j^{\text{old}} - \alpha \frac{1}{n} \sum_{i=1}^{n} \left[h\left(\underline{x}_i\right) - y_i\right] x_{ij}, \forall j \in \{1, 2, \ldots, m\}. \tag{2}$$

The update rule of the parameters can be written in a matrix form as

$$\underline{\theta}^{\text{new}} = \underline{\theta}^{\text{old}} - \alpha \frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \underline{\theta}^{\text{old}} - \alpha \frac{1}{n} \mathbf{X}^T (\mathbf{X}\underline{\theta} - \mathbf{y}). \tag{3}$$

The parameter vector after the convergence of the algorithm can be used for prediction. Note that for each update of the parameter vector, the algorithm processes the full training set. This algorithm is called *Batch Gradient Descent*.

1.2 Python Exercise

Step 1: Load and split dataset then scale features. In this step, you will need to load the Boston dataset from `sklearn` and split it. Concretely, 80 percent of the examples is used for the training set and the rest is for the test set. Then, you have to scale the features to similar value ranges. The standard way to do it is by removing the mean and dividing by the standard deviation.

Step 2: Add intercept term and initialize parameters. In this step, you need to add the intercept term to the training and the test matrices. Normally, the intercept term is the first column of your data matrices. Also, you have to initialize the parameters $\underline{\theta}$ for your model using the normal distribution with $\mu = 0$ and $\sigma = 0.5$.

Step 3: Implement the gradient and cost functions. In this step, you have to implement functions to calculate the cost and its gradient. You can calculate using a for loop but vectorizing your calculation is recommended.

Step 4: Selecting a learning rate using $\mathcal{J}(\underline{\theta})$. Now it's time to select a learning rate α. The goal of this part is to pick a good learning rate in the range of

$$0.001 \leq \alpha \leq 10$$

You will do this by making an initial selection, running gradient descent and observing the cost function, and adjusting the learning rate accordingly. You will calculate $\mathcal{J}(\underline{\theta})$ using the $\underline{\theta}$ of the current step of the gradient descent algorithm. After several steps of iterations, you will see how $\mathcal{J}(\underline{\theta})$ changes at each iteration. **Now**, build your code by following the next steps:

- Run gradient descent for about 50 iterations using your initial learning rate. At each iteration, calculate $\mathcal{J}(\underline{\theta})$ and store the result in vector **J**.
- Test the following values of α: 0.001, 0.003, 0.01, 0.03, 0.1 and 0.3. You may also want to adjust the number of iterations in order to better observe the overall trend of the curve.
- Plot $\mathcal{J}(\underline{\theta})$ for several learning rates on the same graph so as to compare how different learning rates affect convergence. In Python, this can be done by using function `plot` from `matplotlib`. Observe the changes in the cost function as the learning rate changes. *What happens when the learning rate is too small? Too large?*

Step 5: Using the best learning rate that you found, run gradient descent until convergence to find

1. The final values of $\underline{\theta}$.

2. The predicted price of houses from the test set.

2 Linear Regression via Stochastic and Mini-batch Gradient Descent

In this exercise, you will investigate multivariate linear regression using stochastic and mini-batch gradient descent.

2.1 Short Theory

2.1.1 Stochastic Gradient Descent (SGD)

Gradient descent is a powerful optimization method and can be applied to a wide range of loss functions. Nevertheless, when dealing with a big dataset, two problems arise with the vanilla batch gradient descent (GD) algorithm:

- It might be not possible to load and fit the whole training dataset at one iteration
- The optimization might get stuck at local minima

Stochastic Gradient Descent (SGD) is an effective alternative for the vanilla gradient descent. SGD is very simillar to GD, except that at each iteration, the parameter θ is updated using gradient of the cost function calculated from a single training sample, instead of all the training samples.

2.1.2 Mini-batch Gradient Descent

The vanilla gradient descent (GD) and stochastic gradient descent (SGD) both have their own disadvantages. For example, for a big dataset, the former cannot be applied due to memory constraints, and the latter is time consuming due to the large number of required iterations. As a result, an algorithm which is in-between the two would be of great importance. Mini-batch Gradient Descent is a good alternative for both GD and SGD algorithms. The Mini-batch Gradient Descent algorithm is similar to SGD, but instead of learning on a single sample at each iteration, it learns from a batch consisting of a small number of samples at a time. It has several advantages compared to SGD:

- Similar to SGD, it mitigates the problem of local minima
- It converges faster than SGD, as the number of require iterations is smaller
- It fluctuates less heaviliy compared to SGD, as the gradient is estimated from a number of training samples at a time

2.2 Python Exercise

In this exercise, you will implement a simple SGD algorithm to estimate the parameters θ for the linear regression problem. Steps 1-3 have also been done during the previous exercise on GD so you can reuse your code.

Step 1: Load and split dataset then scale features. In this step, you will need to load the Boston dataset from `sklearn` and split it as you do in previous exercise. Concretely, 80 percent of the examples is used for the training set and the rest is for the test set. Then, you have to scale the features to similar value ranges. The standard way to do this is by removing the mean and dividing by the standard deviation.

Step 2: Add intercept term and initialize parameters. In this step, you need to add the intercept term to the training and the test matrices. Normally, the intercept term is the first column of your data matrices. Also, you have to initialize the parameters $\underline{\theta}$ for your model using the normal distribution with $\mu = 0$ and $\sigma = 0.5$.

Step 3: Implement the gradient and cost functions. In this step, you have to implement functions to calculate the cost and its gradient. You can calculate using for loop but vectorizing your calculation is recommended.

Step 4: Stochastic Gradient Descent. In this step, you have to implement the SGD algorithm. At the beginning of the training, or after passing the whole training dataset one time, you need to shuffle your training dataset. It is very important to shuffle the training data and target accordingly. Follow the pseudocode provided in the class and the comment helpers in the exercise.

Step 5: Evaluate θ learned via Stochastic Gradient Descent. In this step, you need to evaluate the parameter θ that you trained via SGD in step 4.

Step 6: Mini-batch Gradient Descent. In this step, you have to implement the Mini-batch Gradient Descent algorithm. The only difference to SGD is the sampling of the training batch at each iteration.

Step 7: Evaluate θ learned via Mini-batch Gradient Descent. In this step, you need to evaluate the parameter θ that you trained via Mini-batch Gradient Descent algorithm in step 6.

3 Linear Regression via Gradient Descent with Regularization

3.1 Short Theory

Selecting a good model is challenging. From the lecture, you know that assuming too many coefficients with respect to the number of observations results in:

- Too many predictors / features
- Complex model

These effects lead to overfitting. There are many techniques to control overfitting, and in this exercise we focus on l_2 regularization. Concretely, we add a regularization term to the cost function and let the optimization algorithm penalize the model's parameters

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n}\Big[\sum_{i=1}^{n}\left(h\left(\underline{x}_i\right) - y_i\right)^2 + \lambda\sum_{j=1}^{m}\theta_j^2\Big] \tag{4}$$

where λ is the regularization coefficient. The matrix form of this regularized cost is as follows:

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n}\Big[\left(\mathbf{X}\underline{\theta} - \mathbf{y}\right)^T\left(\mathbf{X}\underline{\theta} - \mathbf{y}\right) + \lambda\underline{\theta}^T\underline{\theta}\Big] \tag{5}$$

Due to this modification, the gradient for the cost function changes to

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \frac{1}{n}\left[\mathbf{X}^{\mathsf{T}}(\mathbf{X}\underline{\theta} - \mathbf{y}) + \lambda\underline{\theta}\right] \tag{6}$$

and the update rule becomes:

$$\underline{\theta}^{\text{new}} = \underline{\theta}^{\text{old}} - \alpha\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \underline{\theta}^{\text{old}} - \frac{1}{n}\left[\mathbf{X}^{\mathsf{T}}(\mathbf{X}\underline{\theta} - \mathbf{y}) + \lambda\underline{\theta}\right] \tag{7}$$

3.2 Python Exercise

In this exercise, you will modify the functions for calculating the gradient and the cost by adding the regularization term. You will have a chance to monitor how the training process changes with different regularization coefficients. Because this exercise is based on the previous exercise of gradient descent for linear regression, the first part of loading and pre-processing the data has been done for you.

Step 1: Modify the cost and the gradient functions. In this step, you have to modify the cost and the gradient functions using the provided formulas 4, 5, 6 and 7.

Step 2: Train your model with different regularization coefficients. In this step, you train your linear regression model with different coefficients and plot the cost on the training and test sets on the same figure. By doing so, you will notice which coefficient is appropriate for your model.

Step 3: Select the regularization coefficient and visualize your prediction. You have to select the best regularization coefficient and visualize your prediction and the ground truth in the same graph. Notice when calculating the cost here, we set `lambda` to zero. The model with regularization sometimes brings a smaller cost than the model without regularization.