1 Polynomial Regression

In this exercise, you will investigate polynomial regression using least square fitting algorithm as introduced in Lecture 3.

1.1 Short Theory

In the previous exercise, we worked on multivariate linear regression using least square algorithm. We made a hypothesis that our target function is linear with respective to all the variables. This hypothesis does not work well in case we have datasets in which the target function is non-linear with respect to its variables. Polynomials of order higher than 1 are among the common non-linear functions.

Suppose our data has two variables, X_1 and X_2 and our hypothesis function $\hat{y}$ by a polynomial of order 3 on X_1 and order 2 on X_2 and has a multiplicative term X_1X_2, we can first write Y in an explicit form:

$$\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_1^2 + \beta_3 X_1^3 + \beta_4 X_2 + \beta_5 X_2^2 + \beta_6 X_1 X_2. \tag{1}$$

If we consider X_1^2, X_1^3, X_2^2 and X_1X_2 as new variables, our hypothesis function $\hat{y}$ becomes a linear funciton of six variables. The parameter of the hypothesis function has 7 dimensions, including the intercept term: $\beta = [\beta_0, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6]^T$. Appying multivariate linear regression algorithm, we can solve for β which is optimal in least square sense.

By generalizing this technique to datasets with higher number of variables and polynomials of higher orders, we can fit solve a number of real regression problems.

1.2 Python Exercise

Open the file *polynomial_regression.ipynb* In this exercise, we will work with a real dataset of house prices in Boston. This dataset contains 506 samples, each with 13 features, and the target values in range $5 - 50$ (thounsands dollars). For your reference, the meaning of each feature in this dataset is shown in Fig. 1.

```
- CRIM     per capita crime rate by town
- ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS    proportion of non-retail business acres per town
- CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX      nitric oxides concentration (parts per 10 million)
- RM       average number of rooms per dwelling
- AGE      proportion of owner-occupied units built prior to 1940
- DIS      weighted distances to five Boston employment centres
- RAD      index of accessibility to radial highways
- TAX      full-value property-tax rate per $10,000
- PTRATIO  pupil-teacher ratio by town
- B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT    % lower status of the population
```

Figure 1: Meaning of features in the Boston house dataset

Step 1: Load data

As this dataset is included in sklearn, we simply call the built-in funciton *load_boston* in sklearn to load the house features and prices, denoted as X and y, respectivelly.

Step 2: Split training and testing data

In order to fairly evaluate a machine learning algorithm, a seperated testing set is required. In this step, we split the original dataset into non-overlapping training and testing set. Only the training set will be used during parameter fitting. For convenience, a splitting of ratio 80/20 has been implemented for you.

Step 3: Make hypothesis and create virtual features

Let's hypothesize that the target function is linear with regard to all 13 variables, and contains terms which are:

- second degree of the first variable
- second degrees of eighth variable
- third and second degrees of the eleventh variable

These terms can be considered as virtual features of the data. Generate these virtual features and concatenate to X.

Note:

- Don't forget to add another variable of value 1 to account for the intercept term in the polynomial.
- You can check the *stack* or *hstack* functions of Numpy for concatenating matrices.

In the end, X should have 18 variables including the virtual features and the variable for the intercept term.

Step 4: Fit the data

Use the normal equation to estimate the parameters β. For convenience, the function to calculate the pseudo-inverse has been given. You will need to call the pseudo inverse function to estimate the parameters β

Step 5: Make prediction and evaluate

After estimating the parameter β, we can make predictions on the testing set. You will need to:

- Make predictions from X_{te} using the estimated parameteres β.
- Implement two evaluation functions, namely *Mean Square Error (MSE)* and *Mean Absolute Error (MAE)* and call them to evaluate the learned model.

2 Regularized Regression

In this exercise, you will investigate regularized regression as introduced in Lecture 3.

2.1 Short Theory

If we have a model with a lot of parameters and we allow the parameters to grow arbitrarily large, the model can "overfits" the training data, i.e. brings a very low training error but high testing error. To mitigate this problem, regularization techniques are often employed.

In case of linear regression, we can regularize the model by adding a penalty term on the square values of β. The influence of this penalty term is controlled by a parameter λ.

With X the input features, y the target values, λ the regularization parameter, the close form solution for the regularized linear regression,

$$\beta = \left(X^T X + \lambda * I\right)^{-1} X^T y \tag{2}$$

Similar to the un-regularized linear regression algorith, in this case, the predicted values $\hat{y}$ is calculated by:

$$\hat{y} = X\beta \tag{3}$$

2.2 Python Exercise

Continue with the next blocks in the the file *polynomial_regression.ipynb*.

Step 1: Fitting the data

You will need to:

- Implement the function to calculated the pseudo inverse with regularized term taken into account.
- Use the function you implement to estimate the parameter $\beta_{regularized}$

Step 2: Make prediction and evaluate

After estimating the parameter β, we can make predictions on the testing set. You will need to:

- Make predictions from X_{te} using the estimated parameteres $\beta\_regularized$.
- Call the *Mean Square Error (MSE)* and *Mean Absolute Error (MAE)* functions to evaluate the learned model.

3 Linear Regression via Gradient Descent

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

3.1 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

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n} \sum_{i=1}^{n} \left(h\left(\underline{x}_i\right) - y_i\right)^2, \tag{4}$$

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 matrix notations,

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

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

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

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.,

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{1}{n} \sum_{i=1}^{n} \left[h\left(\underline{x}_i\right) - y_i \right] x_{ij}, \forall j \in \{1, 2, \ldots, m\}. \tag{5}$$

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

$$\underline{\theta}^{new} = \underline{\theta}^{old} - \alpha \frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \underline{\theta}^{old} - \alpha \frac{1}{n} \mathbf{X}^T (\mathbf{X}\underline{\theta} - \mathbf{y}). \tag{6}$$

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

When calculating gradient, it is important to verify that your calculation is correct. A typical way to do it is to use the numerical estimation of gradient. Given the cost function $\mathcal{J}(\underline{\theta})$, the numerical estimation of gradient can be found as follows:

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \theta_i} \approx \frac{\mathcal{J}(\theta_1, \theta_2, \theta_i + \epsilon, \cdots, \theta_m) - \mathcal{J}(\theta_1, \theta_2, \theta_i - \epsilon, \cdots, \theta_m)}{2\epsilon} \tag{7}$$

3.2 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 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 it is 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: Verify that your gradient calculation is correct. In this step, you have to check if your gradient function is implemented correctly. Use the equation (7), you can compute the approximate gradient and then compare it with the output from the gradient function. The sum of squared errors should be very small ($\sim 10^{-18}$).

Step 5: 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.

While in the previous exercise you calculated $\mathcal{J}(\underline{\theta})$ over a grid of θ_0 and θ_1 values, you will now calculate $\mathcal{J}(\underline{\theta})$ using the $\underline{\theta}$ of the current stage of gradient descent. After stepping through many stages, you will see how $\mathcal{J}(\underline{\theta})$ changes as the iterations advance.

Now, build your code with the following actions:

- Run gradient descent for about 50 iterations at your initial learning rate. In each iteration, calculate $\mathcal{J}(\underline{\theta})$ and store the result in a vector **J**.
- Test the values of α at a rate of of 3 times the next smallest value (i.e., 0.001, 0.003, 0.01, 0.03, 0.1 and 0.3). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

- Plot $\mathcal{J}(\underline{\theta})$ for several learning rates on the same graph so as to compare how different learning learning rates affect convergence. In Python, this can be done by using function `plot` from `matplotlib`. Observe the changes in the cost function happens as the learning rate changes. *What happens when the learning rate is too small? Too large?*

Step 6: 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.