Sequence 2 exercises about linear regression

1. <u>Linear regression [1] (Homework)</u>

The goal of this exercise is to implement linear regression and to test it on a dataset in order to understand how it works.

Linear regression principle: having at disposal a labelled training set of data, the main goal is to estimate the parameters of a linear function able to fit the training data. The estimated linear function is then used to predict the answer for new input data. The linear regression process involves the minimization of a cost function designed to minimize over the whole training set the mean square error between the predicted value and the right value. The minimization of the cost function is done by using the iterative gradient descent algorithm.

Directory Sequence 2/Exercises contains, in the directory *Student codes and data,* the dataset and the Python codes to be used for this exercise that are:

- ✓ ex1data1.txt for the dataset
- ✓ exercice1_students_code.py which is the main program to be completed

The problem to be solved is the prediction of profits for a food truck by using a linear regression algorithm. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities. You would like to use this data to help you select which city to expand to next.
The file *ex1data1.txt* contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss.
The *exercice1.py* script has already been set up to load this data for you.

1.1 Analyzing the data

How many training samples are available?

Before starting any machine learning task, it is a good idea to have a look at the data when possible. In the present case, since the considered data are 2D, it is easy to plot them on a 2D graph. To do that, run exercise1_Students_code.py and look at the data training set.

When looking at the data distribution, what do you think about the idea to look for a linear model to fit those data?

1.2 Gradient Descent

This part concerns the estimation of the two parameters of the linear function to be used in order to model the training data set.

$$Profits_{pred} = y = \theta_0 + \theta_1 * population = h_\theta(x)$$

This estimation is going to be done by using the gradient descent algorithm which involves an iterative process for the minimization of a cost function.

1.2.1 Gradient descent updating equation

The objective of linear regression is to minimize the cost function *J(θ)*:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

where $h_\theta(x)$ is given by the linear model:

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

The θ values are the values to be adjusted to minimize the J cost by using the gradient descent algorithm. In gradient descent, each iteration performs the update:

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{(simultaneously update } \theta_j \text{ for all } j)$$

With each step of gradient descent, the parameters θ_j come closer to the optimal values that will achieve the lowest J cost.

1.2.2 Computing the cost function J(θ)

As you perform gradient descent to minimize the cost function J(θ), it is helpful to monitor the convergence by computing the cost at each iteration. In this section, you will implement a function to calculate J(θ) so you can check the convergence of your gradient descent implementation. Complete the code of the *computeCost()* function, which is a function that computes J(θ).

Once you have completed the function, the next step in *exercice1_Students_code.py* will run the *computeCost()* using successively θ=[0, 0] and θ=[-1, 2], and you will see the cost printed to the screen. You should expect to see a cost of 32.07 and 54.24 respectively.

1.2.3 Gradient Descent function

Implement the gradient descent by completing the *gradientDescent()* function. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

A good way to verify that gradient descent is working correctly is to look at the value of *J(θ)* and check that it is decreasing at each iteration. Compute and store the value of the cost function on every iteration. What is the *J* cost function value after convergence with theta values initialized to 0?

Assuming you have implemented the *gradientDescent* and *computeCost* functions correctly, your value of J should never increase, and should converge to a steady value by the end of the algorithm. Use *exercise1_Students_code.py* in order to plot the evolution of the J cost function along the iterative process. Comment the obtained curve. What do you think about the chosen number of iterations?

After running the *gradientDescent* function, *exercice1_Students_code.py* proposes to display the estimated model on the training data set. Have a look at the displayed model.

1.3 Profits prediction

Suppose the CEO wants to install a new food truck in two different cities where the numbers of inhabitants are 35.000 and 70.000 respectively. Use the estimated model in order to predict the expected profits in each case.

2. <u>Regularized linear regression (class work)</u>

The considered dataset is the Boston Housing dataset named *housing.csv*. The dataset contains 489 samples. This dataset will be used to predict the MEDV value which represents the median value of owner-occupied homes in $1000.

There are three associated features:

- RM: the average number of rooms per dwelling
- LSTAT: the lower status of the population (percent)
- PTRATIO: the pupil-teacher ratio per town

The provided directory contains the *housing.csv* dataset and a code file named *exercice2_students_code.py*.

Rmk: in this exercise, some *sklearn* functions are going to be used to solve the linear regression estimation problem.

2.1 Linear regression model as baseline predictor

Split the dataset into a training set with 70% of the data and a validation set with 30% of the data (cf. *train_test_split()*). Estimate a linear regression model on the training set (cf. *LinearRegression()* function of the sklearn.linear_model library). Report the model coefficients and the training score.

Rmk: in the case of linear regression model performance evaluation, there are two different ways: first, report the Mean Square Error on the training and the testing sets. When the MSE is lower, the model is better. At the same time, it is possible to use the *score()* attribute of your model. This score is defined in the following way:

score(X, y)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Predict the MEDV values on the validation set. Report the validation score and compare it with the training score.

2.2 Model performances increasing by adding new features?

In order to improve the model performances, it is possible to engineer new features by taking the individual features x_1, x_2, x_3 and raising them to a chosen power. To do so, it is possible to use the *PolynomialFeatures()* function from the sklearn.preprocessing module.

About the *PolynomialFeatures()* function: it generates a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

Generate new features in the training set with a polynom degree 5. But before doing that, think about normalizing the training set (cf. *StandardScaler()* from sklearn.preprocessing) between 0 and 1 to avoid too big numbers when you raise the features to a power.

Apply the same linear regressor on the new training set. Report the training score.

Predict the MEDV value on the validation set (which has not changed) and report the validation score. Conclusion.

2.3 Regularized linear prediction

Now we decide to apply regularization of the new augmented training set. The idea of regularization is to introduce a second term in the cost function in order to limit the model complexity. The influence of this second term is controlled by the lambda hypermarameter value.

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right]$$

Apply a regularized linear regressor on the new training set (look at the *Ridge()* function of the sklearn.model_linear module). Report the training score.

Predict the MEDV value on the same validation set and report the validation score. Conclusion.

Study the influence of the lambda value. Conclusion.