Week 2 - Python Machine Learning

Overview

From this week we will learn how to use PyTorch!

This week we will first focus on:

- Linear regression by the closed form solution formula, your own model
- Linear regression using `sklearn`
- Linear regression by gradient descent using `numpy`
- Linear regression by gradient descent using `pytorch`

Linear regression

Linear regression models the relationship between a set of **explanatory** (or independent) variables and a **dependent** variable with a linear function.

This model is usually written as

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_d x_d + \epsilon$$

where:

- x_i are the explanatory (independent) variables
- y is the dependent variable
- β_0, β_1... are the regression coefficients
- ϵ is the error or noise term

After finding a relationship between your explanatory and dependent variable, the linear regression model can then be used to **predict** the value of any dependent variable given a corresponding explanatory variable.

Matrix Form

The idea of the matrix form is that it represents the above equation for all samples in your data set i.e. each sample will have its own equation.

The regression coefficients remain the same for all samples and so we write it as a column matrix. Note that the number of rows corresponds to the number of features **plus one** as the intercept β_0 is included at the beginning.

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \dots \\ \dots \\ \beta_d \end{bmatrix}$$

Each sample will also have its own corresponding error $\boldsymbol{\epsilon}$ and ouptut $\boldsymbol{y}$, and so we also write these as a column vector, but the dimensions are slightly different than $\boldsymbol{\beta}$:

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ \dots \\ y_n \end{bmatrix}, \quad \boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \dots \\ \dots \\ \epsilon_n \end{bmatrix}$$

where *n* is the number of samples we have.

By defining the matrices for $\boldsymbol{\beta}$ and $\boldsymbol{\epsilon}$, we constrain the dimensions for our data matrix $\boldsymbol{X}$. It must have $d+1$ columns (so that it can be multiplied with $\boldsymbol{\beta}$) and n rows (so that we can add it to $\boldsymbol{\epsilon}$).:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1d} \\ 1 & x_{21} & x_{22} & \dots & x_{2d} \\ 1 & x_{31} & x_{32} & \dots & x_{3d} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n1} & x_{n2} & \dots & x_{nd} \end{bmatrix}$$

How do we interpret this?

- Each **row** of $\boldsymbol{X}$ corresponds to a different sample in our data set
- Each **column** of $\boldsymbol{X}$ matches with a regression coefficient or feature (excluding the 1's column)

Let us combine everything!

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1d} \\ 1 & x_{21} & x_{22} & \dots & x_{2d} \\ 1 & x_{31} & x_{32} & \dots & x_{3d} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n1} & x_{n2} & \dots & x_{nd} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \dots \\ \dots \\ \beta_d \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \dots \\ \dots \\ \epsilon_n \end{bmatrix}$$

or

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

Estimating parameters

In linear regression, the $\boldsymbol{\beta}$ parameters can take any value. But what is the right value? Ideally, the right value is the one that leads to the most accurate predictions.

This is where OLS comes in - it is designed to give you the parameters that minimise the **square of the errors**. It is an **analytic** solution.

Ordinary Least Squares

To estimate the parameters for a *simple linear regression*, we solve the following objective function:

$$\min_{\beta_0,\beta_1} \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

$$\min_{\beta_0,\beta_1} \frac{1}{2}\sum_{i=1}^{n}(\beta_0 + \beta_1 x_i - y_i)^2$$

meaning "find values of β_0 and β_1 that minimise the squared prediction error".

We can rewrite this using a dot product for *multiple linear regression*:

$$\min_{\boldsymbol{\beta}} \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i^T\boldsymbol{\beta} - y_i)^2$$

where $\mathbf{x}_i = [1, x_{i1}, x_{i2}, ..., x_{id}]^T$ and $\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, ..., \beta_d]^T$

- Note in here that d is the number of features we have per observation

Note: *That for the intercept, we multiply β_0 by 1.*

Further simplification allows us to rewrite it as:

$$\min_{\boldsymbol{\beta}} \frac{1}{2}\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$$

- Note in here that X is now a matrix

You can visualise the idea of least squares using this link. Play around with the different β_0 and β_1 values and observe how they change not only the regression line, but the squared errors.

Solution

As explained in the lecture slides, the steps to solve this equation are:

1. Calculate the derivative of the function

$$\mathbf{X}^T(\mathbf{X}\widehat{\boldsymbol{\beta}} - \mathbf{y})$$

Note: *The factor of* $\frac{1}{2}$ *disappears.*

2. Set the derivative to 0

$$\mathbf{X}^T(\mathbf{X}\widehat{\boldsymbol{\beta}} - \mathbf{y}) = 0$$

3. Solve for $\widehat{\boldsymbol{\beta}}$

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Intercept

Don't forget that to add the intercept to our feature matrix $\mathbf{X}$ we can put a column of 1s at the front like so:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1d} \\ 1 & x_{21} & x_{22} & \dots & x_{2d} \\ & & \dots & & \\ 1 & x_{n1} & x_{n2} & \dots & x_{nd} \end{bmatrix}$$

□□ Create your own Linear Regression Model

Write a Python **class** for a Linear Regression model.

Example Usage

```
model = LinearRegression(True)
model.fit(X, y)
print(model.beta)
```

Class Specification

- name: `LinearRegression`
- methods:
  - `__init__`
    - parameters:
      - `intercept` (boolean): whether to include an intercept in the model or not
  - `fit`
    - parameters:
      - `X` (numpy array): observations with shape (n_samples, n_features)
      - `y` (numpy array): target values
    - returns: nothing
  - `predict`
    - parameters:
      - `X` (numpy array): observations with shape (n_samples, n_features)
    - returns:
      - `y` (numpy array): predicted values for each sample in `X`
- attributes:
  - `beta` (numpy array): estimated model parameters i.e. $\beta_0, \beta_1, \ldots, \beta_p$. Must be set during `fit`.

□□ Task 1 - Linear regression and prediction using sklearn

We are going to use '`auction.txt`' dataset for clock auction. It has three columns, '`age`'-age of the clock; '`Bidders`'-how many bidders making an offer; '`price`'-sale price of this clock.

Please follow the following steps to complete the task.

Step 1: On the right workspace, below "Load data" add your code to read in data from `auction.txt` with `pandas`.

Step 2: Get more information about this dataset, for example, column names etc. You may consider use `pandas DataFrame`'s `info()` and `describe()`.

Step 3: Define input `X` and `y` by taking out both columns '`Age`' and '`Bidder`' for `X` and the column '`Price`' for `y`. Think about the right way from `pandas`. Distinguish pandas `DataFrame`/`Series` from data values. Normally `sklearn` functions take as inputs data values or `numpy` arrays.

Step 4: First we shall define a linear regression model, then fit it to the data. For your convenience, this step has been completed for you. Please delete `#` signs. We can add intercept by specifying `fit_intercept = True`.

Step 5: Note fit function returns a learned object (`lr_obj`) which contains all the information for the learned linear regression model. For example, you can get the linear model's intercept from this object by `lr_obj.intercept_` and beta (our β_1 and β_2) from `lr_obj.coeff_` which is a list. Add your code to get them or print their values out.

Step 6: Predict the price for a 121 year old clock with 15 bidders. The code is there for you. Please note the data reshape. Why do we do this?

□□ Task 2 - Linear regression gradient descent using Numpy

In this task, we will demonstrate gradient descent on linear regression. As shown in the lecture, linear regression model has the following loss:

$$L = \min_{\beta} \frac{1}{2N} (\mathbf{y} - \mathbf{X}\beta)^2$$

and its gradient with respect to β is given by:

$$\nabla_{\beta} L(\beta) = \frac{1}{N} \mathbf{X}^T (\mathbf{X}\beta - \mathbf{y}).$$

Then the gradient descent works by updating

$$\hat{\beta}_t = \hat{\beta}_{t-1} - \alpha \nabla_{\beta} L(\hat{\beta}_{t-1})$$

for $t = 1, ..., T$

For the linear regression model, we have the closed-form formula for the exact solution. Why do we consider using the gradient descent for an approximation solution. One of the main reasons is that the computational bottle-neck of finding the matrix inverse in the closed-form solution $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$. When the matrix size is very large, it is very very time consuming to get the matrix inverse. So an approximation solution is welcomed.

Step 1: Define the Gradient Descent

According to the formulas for gradient and gradient descent updating, please add your code in Line 36 and 38 to complete the `Gradient_Descent_LinearReg` function.

Step 2: Apply the GD to an example:

Here we generate synthetic data from a linear model

$$f(x, \beta) = \beta_0 + \beta_1 x,$$

where $\beta_0 = 4, \ \beta_1 = 1.5,$

Add noise to true values $t = f(x, \beta) + \epsilon,$ where ϵ comes from a normal distribution with mean=0, standard deviation = 0.1

We will see whether we can find out an estimate to $\beta_0 = 4, \beta_1 = 1.5.$

Fitting model by using function "Gradient_Descent_LinearReg", set parameters as follows:

- Learning rate: $\alpha = 0.0005$
- Max number of iterations: $numIterations = 10000$

According to the above information, please add your code to complete line 70. Then click Run to test the program.

(You can try different parameters and see if there is any difference)

Task 3 - Linear regression gradient descent using Pytorch

In this task, we will introduce you to `Pytorch` library that is popular within deep learning community and thus can implement state-of-the-art models with high efficiency. But, its syntax and data structure are different from `numpy`. The two main differences to `numpy` is:

- The basic data structure is Tensor. A Tensor is a n-dimensional arrays. It basically looks the same as `numpy` arrays, but it contain gradient information of itself with respect to a function and it can also run on GPUs to accelerate its numerical computations.
- Its automatic differentiation allows gradients to be computed without specifying the details as in Task 2.

Pytorch is available on ED, however you need install it if you want to use on your own machine. Please go to https://pytorch.org/get-started/locally/ and follow the instructions according to your device.

Before we show you how to use auto differentiation, let's see some basic examples of Pytorch syntax:

```
import torch
# for replicability, use this line at the start of your program
torch.manual_seed(0)

# we can define the torch tensor similar to numpy
a = torch.tensor([[1., 2.],
                  [3., 4.]])
b = torch.rand(2,2)

print(a)
print(b)
```

Basic matrix operations are also similar to numpy

```
import torch
# for replicability, use this line at the start of your program
torch.manual_seed(0)

# we can define the torch tensor similar to numpy
a = torch.tensor([[1., 2.],
                  [3., 4.]])
b = torch.rand(2,2)

# transpose
print(a.t())
print()

# Addition, subtraction
print(a + b)
print(a - b)
print()

# matrix multiplication
print(torch.matmul(a,b))
print(a @ b)
print()

# elementwise multiplication; elementwise power
print(a * b)
print(a ** 2)
print()

# shape
print(a.shape)
print()

# reshape
print(a.view(4,1))
```

Convert between numpy array and torch tensor

```
import torch
# from numpy to torch tensor
a_numpy = np.array([[1, 2],
                    [3, 4]])
a = torch.from_numpy(a_numpy)
print(type(a_numpy))
print(type(a))

# from torch tensor to numpy
a_back_to_numpy = a.numpy()
print(type(a_back_to_numpy))
```

Automatic differentiation

The key idea of automatic differentiation in Pytorch is to keep track of a variable and its computational graph. For this, we need to differentiate between a normal torch tensor and a variable that requires gradient to be tracked. See the following example on how to compute the gradient automatically.

(Verify for yourself whether this gradient matches manually calculated gradient.)

```
import torch

X = torch.tensor([[1., 2.],
                  [3., 4.]])
y = torch.tensor([[2.], [3.]])

# initalize a variable you wish to compute gradient for
# (which is a torch tensor that requires gradient)
b = torch.tensor([[0.], [0.]], requires_grad = True)
```

```
# mse loss
loss = ((X @ b - y)**2).sum()

# compute the gradient wrt b at [0,0]
loss.backward()

print(b.grad)
```

Linear regression by gradient descent using pytorch

Step 1: Prepare Synthetic Data

We reuse the code from Task 2 to create synthetic data.

Step 2: Convert data to torch tensor

We first need to convert data X, y to torch tensor object. We can set which device we want to execute the command (i.e. cpu or gpu). Note we need to have a consistent dtype. `DoubleTensor` is 64 bits floating point and `FloatTensor` is 32 bits floating point. Essentially, `DoubleTensor` has higher precision but `FloatTensor` requires less memory.

Step 3: Set Parameters

Next, we need to initialize our parameters β. The most important part is that we need to set `requires_grad = True` to tell Pytorch that this is a parameter to be optimized and you need to keep track of its gradient in the computational graph.

Step 4: Follow the standard steps

Then we can define the whole optimization process. Note in Pytorch, there is no gradient descent. But they have **stochastic gradient descent (SGD)**, which is more efficient when sample size is large. This is because it computes gradient on a subset of samples. We can utlize SGD to implement gradient descent by passing the entire dataset every iteration.

- Define your *loss* function (objective function). In the case of linear regression function, this is the mean square function, which is defined in Pytorch as `torch.nn.MSELoss()`
- Define an *optimizer* with the parameter variable to be optimized and learning rate
- In a loop, continuously
  - (1) calculate your loss function based on your model output and the target
  - (2) call backward() over your loss function (this is to calculate all the gradients)
  - (3) call optimizer's step() function to perform the gradient descent to update the parameters
  - (4) call optimizer's zero_grad() function to clear up the gradient as we no longer need it after updating the parameters. Some people move (4) to (1).

I would like to suggest you carefully read the code on the right workspace. Make a copy to your own machine and play with it.