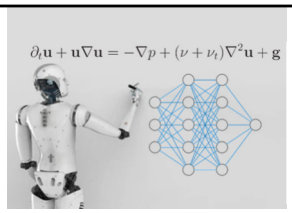


Hands on Machine Learning
for Fluid Dynamics

7 – 11 February 2022

von KARMAN INSTITUTE
FOR FLUID DYNAMICS

Lecture 4

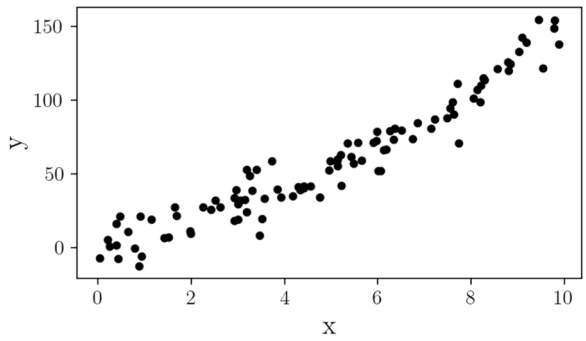
Tutorial Exercise 1: Regression with Uncertainties

Miguel Alfonso Mendez
mendez@vki.ac.be

Slide 1

Consider Today's dataset

```
# Generate data
n_p=100
np.random.seed(10)
x_s=np.random.uniform(0,10,n_p)
y_s=2*x_s+2+np.random.normal(loc=0,scale=10,size=len(x_s))+x_s**2
```



Slide 2

Step 1: Fitting Lines and Parabolas

Build two Python functions to perform the linear regression of the data using the analytic solution we have derived today:

$$\mathbf{y} = \mathbf{X}\mathbf{w} \longrightarrow \mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

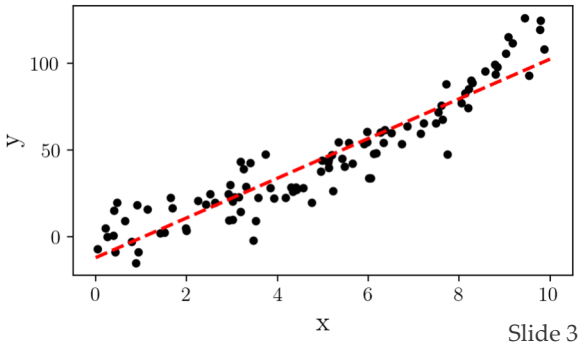
$$\mathbf{X} = \begin{bmatrix} 1 & x_0 \\ | & | \\ 1 & x_{n_p-1} \end{bmatrix} \in \mathbb{R}^{n_p \times 2} \qquad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

```
def line_s_fit(x,y):
    # take input vectors x,y and fit a line
    # with coefficients w_0 and w_1

    return w
```

Validate your results with numpy's polyfit. Then, build another function that uses a second order polynomial.

Note that the exact same approach can be used to fit higher order polynomials.



Slide 3

Step 2: Robust (Regularized) Fitting

As we will see more in details in the Lecture 9, it is often a good idea to increase the robustness of the regression by adding a regularization. Let's see it here from a numerical stability point of view. We are essentially solving the following linear system

$$\mathbf{Hw} = \mathbf{b} \quad \text{with} \quad \mathbf{H} = \mathbf{X}^T\mathbf{X} \quad \mathbf{b} = \mathbf{X}^T\mathbf{y}$$

In a linear system, perturbations in the RHS leads to perturbation in the solution following the classic inequality:

$$\frac{||\delta\mathbf{w}||}{||\mathbf{w}||} \leq \kappa(\mathbf{H})\frac{||\delta\mathbf{b}||}{||\mathbf{b}||}$$

Condition number

Condition number of H:
ratio of largest to smallest eigenvalue

$$\kappa(\mathbf{H}) = \lambda_M/\lambda_m$$

To regularize means to improve (decrease) the condition number. A simple approach (Ridge regression) consist in adding a small number in the diagonal.

$$\mathbf{H}' = \mathbf{H} + \alpha\mathbf{I}$$

Slide 4

Step 2: Robust (Regularized) Fitting

Here are some questions for you

1. What is the condition number of the Hessian for the linear and the quadratic problem?
2. Derive a formula to link the Ridge coefficient alpha to the largest condition number you tolerate
3. Prepare two python functions to compute the weights for the linear and quadratic fit taking as used input a limiting condition number.

```
def Quadratic_s_fit_red(x,y,k_A_l=1000):
    # take input vectors x,y and fit a line
    # with coefficients w_0 and w_1
    n_p=len(x)
    X=np.zeros((n_p,3))
    X[:,0]=
    X[:,1]=
    X[:,2]=
    # Compute the Hessian
    H=X.T@(X)
    # compute the eigenvalues, the largest (l_M) and the smallest (l_m)
    Lambd=np.linalg.eig(H); l_M=np.max(Lambd[0]); l_m=np.min(Lambd[0]);
    alpha=
    H_p=H+alpha*np.identity(np.shape(H)[0])
    w=np.linalg.inv(H_p).dot(X.T).dot(y)
    return w
```

Note: check that the regularized Hessian has the condition number you set! (see Python's function np.linalg.cond)

Slide 5

Step 3: Uncertainty Analysis

We are interested in analyzing the uncertainties of the regression for both regularized and un-regularized regressions. We follow an ensemble approach to compute uncertainties directly from the posterior population

1) Write a function that computes the regression using the analytical tools for n_e times, using each time 70% of the data for training and 30% of the data for validation.

2) From the n_e results, compute the average in sample and out of sample errors.

3) Then, given an arbitrary grid x_g, provide the mean prediction and the associated uncertainties using the ensemble statistics:

$$\mu_y(x) = \frac{1}{n_e}\sum_{j=0}^{n_e-1} y_j(x) \qquad \sigma_y^2(x) = \frac{1}{n_e}\sum_{j=0}^{n_e-1}(y_j(x) - \mu_y(x))^2$$

4) For each regression, store the resulting weights. Then look at the distribution (histogram) of each weight.

Slide 6

```
def Boot_Strap_Model_2_esamble(X,y,x_g,n_e=500,tp=0.3):
    Pop_Y=np.zeros((len(x_g),n_e)) # posterior population
    mu_y=np.zeros(len(x_g)) # mean prediction
    Unc_y=np.zeros(len(x_g)) # Uncertainty in the prediction
    J_i_mean=np.zeros(n_e) # in sample error
    J_o_mean=np.zeros(n_e) # out of sample error
    w_e=np.zeros((3,n_e)) # Output distribution of weights
    # Take back the x vector
    x=X[:,1]
   # Loop over the ensamble
    for j in range(n_e):
     # Split the
     xs, xss, ys, yss = train_test_split(x,y, test_size=tp)
     # Fit the weights
     w_s=
     # Assign vectors to the distributions
     w_e[:,j]=w_s
     # Make in-sample prediction----------------------------
     y_p_s=
     # In-sample error
     J_i_mean[j]=
     # Make out-of sample prediction (and errors)
     y_p_ss=
     # Out of sample error
     J_o_mean[j]=
     # Fill the population matrix
     Pop_Y[:,j]=

    # Compute the mean and the uncertainty
    mu_y=
    Var_y=
    Var_Y_model=
    Var_y_tot=
    Unc_y=1.96*np.sqrt(Var_y_tot)

    return mu_y, Unc_y, J_i_mean, J_o_mean, w_e
```

Here's a possible structure
The general idea is:

(1) Initialize all the vectors that will be exported and the matrices that will be filled

(2) Loop over the ensemble. Use scikit learn's function train_test_split

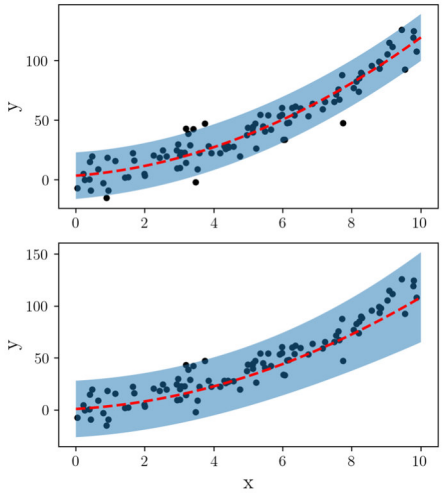
(3) Train the regression on the training data and compute performances on both training and test data

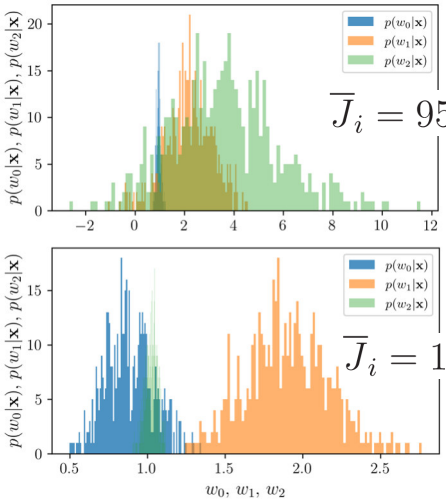
(4) Compute the ensemble statistics using python's function np.mean and np.std

Slide 7

Step 3: Uncertainty Analysis (2)

Here are the uncertainties and the distribution of weights for the ordinary and the regularized problem (taking k_A max= 1000)





Ordinary

$\overline{J}_i = 95.3 \quad \overline{J}_o = 106.0$

Regularized

$\overline{J}_i = 196.4 \quad \overline{J}_o = 187.2$

Slide 8

Step 4: Batch Gradient Descent

Write a python function to implement the mini-batch gradient descent, taking as inputs a function to compute costs and gradients, a decaying learning rate, a given number of epochs and a batch size.

First, you should write the following functions

```
def cost(w,X,y):


   return J
```

```
def grad(w,X,y):


   return Nabla_J
```

Then you should aim for a function of this form:

```
def Batch_GD(cost,grad,w0,X,y,eta_0,decay,n_epochs,n_batch):

   return w_opt, w_evolution, Err_SGD
```

Slide 9

```python
def Batch_GD(cost,grad,w0,X,y,eta_0,decay,n_epochs,n_batch):
    # Prepare the loop per epoch
    n_iter=
    # number of points and features
    n_p,n_f=np.shape(X)
    # Initialize batch sample Design Matrix
    X_b=
    # Current estimate of w
    w=w0
    # Initialize the weight evolution and the error evolution
    Err_SGD=np.zeros(n_iter); #Err_SGD[0]=cost(w,X,y)
    w_evolution=np.zeros((n_f,n_iter)); #w_evolution[:,0]=w0

    for j in range(n_iter):
      # Select randomly some data points for the batch
      # Note that replace=False means that there is no repetition
      Indices=
      # Construct the matrix X_b
      X_b=X[Indices,:]; y_b=y[Indices]
      # Get the current cost
      Err_SGD[j]=cost(w,X_b,y_b)
      #Get the gradient
      Nabla_J_w=grad(w,X_b,y_b)
      # Compute the learning rate
      eta=eta_0/(1+decay*j)
      # Weght update
      w=
      # Store the result in the history
      w_evolution[:,j]=w
      # Message
      Mex='Iteration: {:d}; Epoch: {:d}; Cost: {:.3f}; Grad_abs: {:.3f}'\
          .format(j,j//n_batch,Err_SGD[j],np.linalg.norm(Nabla_J_w))
      print(Mex)

    w_opt=w # Final result on the weight
    return w_opt, w_evolution, Err_SGD
```

Here's a possible structure

The general idea is:

(1) Take some random indices for the batch selection

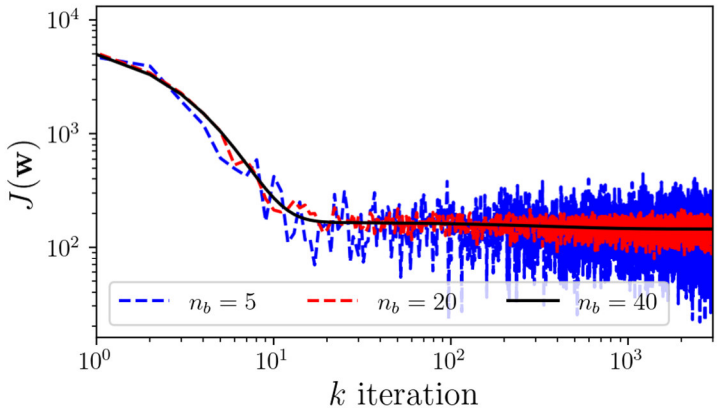
(2) Sample the data

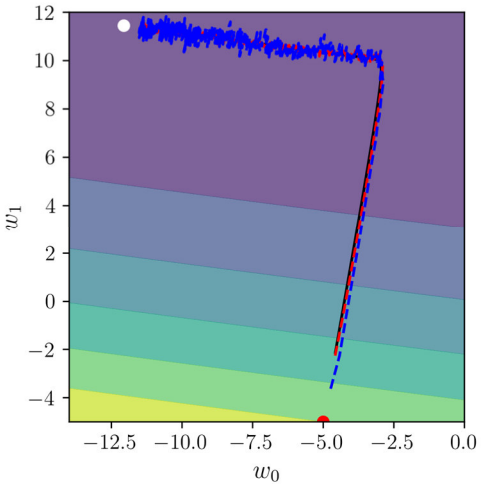
(3) Evaluate cost and gradients

(4) Update the current estimate of the weights

Slide 10

Evolution of the Learning Process





Example results for the regression of a line using ordinary (unregularized) least square

The end!

Slide 12