

Hands on Machine Learning for Fluid Dynamics

7 – 11 February 2022



Lecture 9 Linear Tools for Nonlinear Regression and Super-Resolution





Miguel Alfonso Mendez
mendez@vki.ac.be

Slide 1

Python Files for this Lecture



VKI Lecture Series and Events - Site for Participants > Hands on Machine Learning for Fluid Dynamics > Python_Codes > **Lecture_9**

 Name ▾	Modified ▾	Modified By ▾	Created ▾	Created By ▾	Description
 File_1_Bases.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
 Exercise_2.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	
 Exercise_1.py	A few seconds ago	Miguel Alfonso Mendez	A few seconds ago	Miguel Alfonso Mendez	

Slide 2

Table of Contents



1. Linear Models for Regression: bases and feature scaling
2. Tikhonov and Lasso Regularization
3. A brief note on Support Vector Regression
4. Physics Constrained (Informed) Regression

Slide 3

Table of Contents



1. Linear Models for Regression: bases and feature scaling

2. Tikhonov and Lasso Regularization
3. A brief note on Support Vector Regression
4. Physics Constrained (Informed) Regression

Linear Regression as Projection onto Bases

We have seen in lecture 1 and lecture 4, that the regression of first and second order polynomials reduces to the following linear problem:

$$\mathbf{y} = \mathbf{X}\mathbf{w}$$

Fitting a line Fitting a parabola

$\mathbf{X} = \begin{bmatrix} | & | \\ 1 & x \\ | & | \end{bmatrix} \in \mathbb{R}^{n_p \times 2}$
 $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$
 $y = w_0 + w_1 x$

$\mathbf{X} = \begin{bmatrix} | & | & | \\ 1 & x & x^2 \\ | & | & | \end{bmatrix} \in \mathbb{R}^{n_p \times 3}$
 $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$
 $y = w_0 + w_1 x + w_2 x^2$

Slide 5

The General Framework

We can generalize the previous architecture as follows

$$\mathbf{y} = \Phi(\mathbf{x}) \mathbf{w} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ | \\ w_{n_b} \end{bmatrix} \in \mathbb{R}^{n_b}$$

$$\Phi(\mathbf{x}) = \begin{bmatrix} | & | & \dots & | \\ \phi_0(x; \mathbf{p}_0) & \phi_1(x; \mathbf{p}_1) & \dots & \phi_{n_b}(x; \mathbf{p}_{n_b}) \\ | & | & \dots & | \end{bmatrix}$$

Basis Matrix

Basis Function:
 $\phi_r(x; \mathbf{p}_r)$
 function parameters

This leads to a model of the form: $y = \sum_{r=0}^{n_b-1} \phi_r(x; \mathbf{p}_r) w_r$

Thus we see that the training of this model is the same as the one we have seen in Lecture 1 and Lecture 5:

Just replace \mathbf{X} with $\Phi(\mathbf{x})$

$$J(\mathbf{w}) = \|\mathbf{y} - \Phi(\mathbf{x})\mathbf{w}\|_2^2$$

$$\mathbf{w}^* = (\Phi^T(\mathbf{x})\Phi(\mathbf{x}))^{-1}\Phi^T(\mathbf{x})\mathbf{y}$$

Slide 6

Some classic Bases

The zoology of basis function is vast. We consider three cases, all defined as a function of a distance parameter:

$$d_r(x) = x - x_r^*$$

1) Sigmoidal Functions

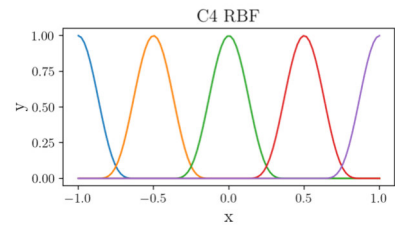
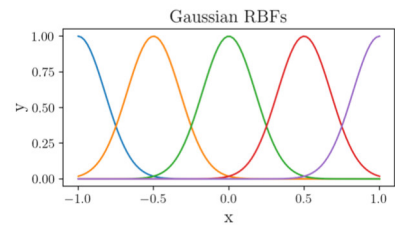
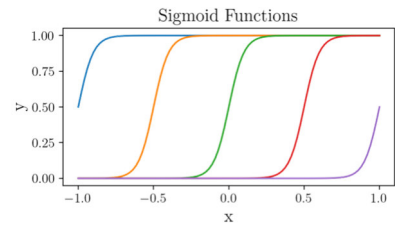
$$\phi_r(x; x_r^*, c_r) = \frac{1}{1 + \exp(-d_r(x)/c_r)}$$

2) Gaussian Radial Basis Functions (RBF)

$$\phi_r(x; x_r^*, c_r) = \exp(-c_r^2 d_r(x)^2)$$

3) Home made (C4) RBF

$$\phi_r(x; x_r^*, c_r) = \begin{cases} (1 + d(x)/c_r)^5 (1 - d(x)/c_r)^5 & |d| \leq c_r \\ 0 & |d| > c_r \end{cases}$$



Some classic Bases (Python):



```
%% Sigmoid Basis function
def sigmoid(x,x_r=0,c_r=0.1):
    z=(x-x_r)/c_r;
    phi_r=1/(1+np.exp(-z))
    return phi_r
```

```
%% Gaussian Basis function
def Gauss_RBF(x,x_r=0,c_r=0.1):
    d=x-x_r # Get distance
    phi_r=np.exp(-c_r**2*d**2)
    return phi_r
```

```
%% Custom Basis function
def C4_Compact_RBF(x,x_r=0,c_r=0.1):
    d=x-x_r # Get distance
    phi_r=(1+d/c_r)**5*(1-d/c_r)**5
    phi_r[np.abs(d)>c_r]=0
    return phi_r
```

Slide 8

A note on feature scaling



Many bases and many algorithms work better (and are easier to train) if they work with **normalized (or scaled)** data. This means that the input space can be mapped onto $[-1,1]$ or $[0,1]$.

Classic approaches are

Min-Max normalization

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Standardization

$$x' = \frac{x - \text{mean}(x)}{\text{std}(x)}$$

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler

# Create a scaler object and then fit
scaler = MinMaxScaler();
# or
scaler=StandardScaler()

scaler.fit_transform(x_test.reshape(-1,1))
x_prime=scaler.transform(x_test.reshape(-1,1)) # Scale
x_back=scaler.inverse_transform(x_prime)[: ,0] # Invert
```

Scikitlearn offers many other options;

see [Help Scikit learn](#)

Here is a nice comparison of various scalers versus outliers:

[Tutorial Scaler Scikit Learn](#)

Slide 9

Table of Contents



1. Linear Models for Regression: bases and feature scaling

[2. Tikhonov and Lasso Regularization](#)

3. A brief note on Support Vector Regression

4. Physics Constrained (Informed) Regression

Slide 10

Regularization and constrained optimization

In lecture 4, we have introduced the notion of regularization as a mean to ensure the numerical stability of the regression problem. We revisit it in the framework of constrained optimization

Consider the
constrained
optimization

$$\min_{\mathbf{x}} f(\mathbf{x})$$

s.t. $g(\mathbf{x}) = 0$

The minimum of the function must satisfy:

$$df = \frac{\partial f}{\partial x_1} dx_1 + \frac{\partial f}{\partial x_2} dx_2 + \dots = 0$$

This means:

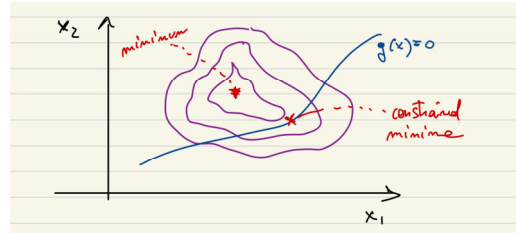
$$df = \nabla f \cdot d\mathbf{x} = 0$$

Similarly, we can write

$$dg = \frac{\partial g}{\partial x_1} dx_1 + \frac{\partial g}{\partial x_2} dx_2 + \dots = 0$$

This means:

$$dg = \nabla g \cdot d\mathbf{x} = 0$$



$$\nabla f + \lambda \nabla g = 0$$

Lagrange Multiplier

$$\nabla(f + \lambda g) = \nabla \mathcal{A} = 0$$

Auxiliary function

The constrained optimization is an unconstrained optimization for the auxiliary function

Slide 11

Practice

Solve the following,
constrained optimization problem:

$$\begin{cases} f(x, y) = 2x^2 + 3y^2 \\ \text{s.t. } 2x + y = 1 \end{cases}$$

Slide 12

Tikhonov, Lasso and Elastic Nets

The only difference between constrained optimization and regularization is in how we deal with the Lagrangian multiplier. In optimization, we solve for it. In regularization, we impose it.

In practice, it is as if we are satisfying an unknown constraint!

We thus define a new performance measure: a **regularized least square cost function**

$$J(\mathbf{w}) = \|\mathbf{y} - \underbrace{\Phi(\mathbf{x})\mathbf{w}}_{\text{prediction}}\|_2^2 + \underbrace{\alpha R(\mathbf{w})}_{\text{penalization term}}$$

dataregularization function

- Classic choices:
- $R(\mathbf{w}) := \|\mathbf{w}\|_2$ Tikhonov regularization (Ridge Regression)
 - $R(\mathbf{w}) := \|\mathbf{w}\|_1$ Lasso Regularization
 - $R(\mathbf{w}) := \gamma_1 \|\mathbf{w}\|_1 + \gamma_2 \|\mathbf{w}\|_2$ Elastic Net
 - $R(\mathbf{w}) := 0$ Ordinary Least Square
- promote 'small' \mathbf{W}
promote 'sparse' \mathbf{W}

Slide 13

The solution for Ridge Regression

Recalling the useful vector differentiation rules

$$\nabla_{\mathbf{x}}(\mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x}) = 2\mathbf{A}^T \mathbf{A} \mathbf{x} \quad \nabla_{\mathbf{x}}(\mathbf{x}^T \mathbf{A}^T \mathbf{b}) = \mathbf{A}^T \mathbf{b} \quad \nabla_{\mathbf{x}}(\mathbf{b}^T \mathbf{A} \mathbf{x}) = \mathbf{A}^T \mathbf{b}$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}, \mathbf{y}) \propto \nabla_{\mathbf{w}} \left[(\mathbf{y} - \Phi(\mathbf{x})\mathbf{w})^T (\mathbf{y} - \Phi(\mathbf{x})\mathbf{w}) + \alpha \mathbf{w}^T \mathbf{w} \right]$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}, \mathbf{y}) \propto \nabla_{\mathbf{w}} \left[\mathbf{w}^T \Phi^T(\mathbf{x}) \Phi(\mathbf{x}) \mathbf{w} - \mathbf{w}^T \Phi^T(\mathbf{x}) \mathbf{y} + \mathbf{y}^T \Phi(\mathbf{x}) \mathbf{w} + \mathbf{y}^T \mathbf{y} + \alpha \mathbf{w}^T \mathbf{w} \right]$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}, \mathbf{y}) \propto 2 \left(\Phi^T(\mathbf{x}) \Phi(\mathbf{x}) \mathbf{w} - \Phi^T(\mathbf{x}) \mathbf{y} + \alpha \mathbf{w} \right)$$

Observe that the Hessian is $\mathcal{H}(J(\mathbf{w}; \mathbf{x}, \mathbf{y})) = 2\Phi^T(\mathbf{x})\Phi(\mathbf{x}) + 2\alpha\mathbf{I}$

The analytic solution is thus: $\mathbf{w}^* = (\Phi(\mathbf{x})^T \Phi(\mathbf{x}) + \alpha \mathbf{I})^{-1} \Phi(\mathbf{x})^T \mathbf{y}$

Slide 14

L1 versus L2 Penalties: Geometry

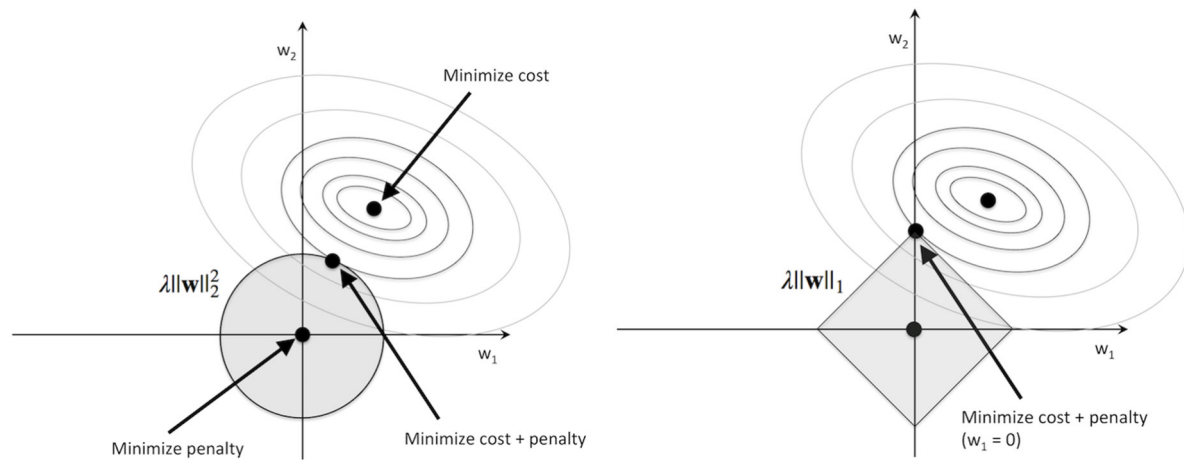


Figure from <http://rasbt.github.io/mlxtend/>, by Sebastian Raskha

See Hastie et al, Statistical Learning with Sparsity (see [this link](#))

Slide 15

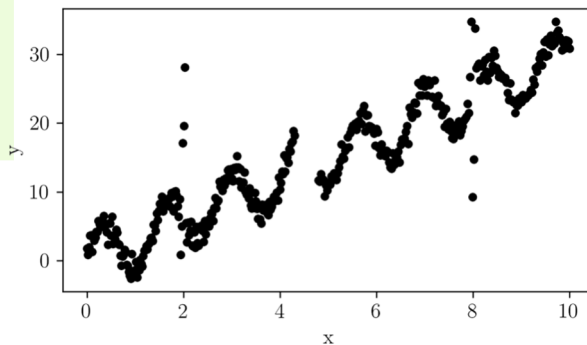
L1 versus L2 Penalties: Practice!



Consider the following dataset

```
### Problem to Fit
x1 = np.linspace(0, 4.3, 200, endpoint=True)
x2 = np.linspace(4.8, 10, 200, endpoint=True)
x=np.concatenate((x1,x2))
# Create the deterministic part
y_clean= 3*x+(x/100)**3+4*np.sin(3/2*np.pi*x)
# Add (a seeded) stochastic part
np.random.seed(0)
y=y_clean+1*np.random.randn(len(x))
# Introduce some outliers in x=2 and x=8
G1=10*np.exp(-(x-2)**2/0.005)*np.random.randn(len(x))
G2=15*np.exp(-(x-8)**2/0.005)*np.random.randn(len(x))
y_final=y+G1+G2
```

We have redundancy, lack of data and outliers...!



Slide 16

L1 versus L2 Penalties: Practice!



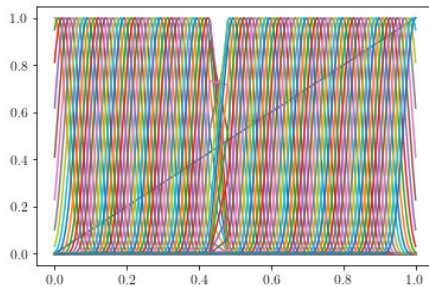
- 1) Compute the feature matrix for the three regressors we have seen, after scaling the data using min/max scaler. Take 100 equally spaced bases with equal c and add a linear term. Then compute the condition number for the associated Hessians.
- 2) For the best of the three, compute the OLS regression and fit the results on a grid with 300 points.
- 3) Compute the same regression with Ridge or Lasso regression, taking $\alpha = 0.01$. Compare the weights in the two cases.
- 4) Consider Ridge regression. Perform a K-fold analysis with $K=10$ for various values of α . Plot the out of sample error as a function of α . What is the best value?

Slide 17

Question 1

Preparing the bases. Here's how to do it for the Gaussian RBFs

```
# Check the basis:
Phi_C4_X=PHI_C4_X(x_prime[:,0], x_b, c_r=0.1)
plt.figure(1)
plt.plot(x_prime,Phi_C4_X)
```



We thus focus on the C4 case

```
## 2 Gaussian Basis function
def Gauss_RBF(x,x_r=0,c_r=0.1):
    d=x-x_r # Get distance
    phi_r=np.exp(-c_r**2*d**2)
    return phi_r

def PHI_Gauss_X(x_in, x_b, c_r=0.05):
    n_x=np.size(x_in)
    Phi_X=np.zeros((n_x,n_b+1)) # Initialize Basis Matrix on x
    # Add a constant and a linear term
    Phi_X[:,0]=x_in
    # Loop to prepare the basis matrices (inefficient)
    for j in range(0,n_b):
        # Prepare all the terms in the basis
        Phi_X[:,j+1]=Gauss_RBF(x_in,x_r=x_b[j],c_r=c_r)
    return Phi_X
```



Repeating the same structure for the other bases, and using 'reasonable parameters':

```
# Compute the condition number
H_Gauss=Phi_Gauss_X.T@Phi_Gauss_X
k_H_Gauss=np.linalg.cond(H_Gauss)
print('rcond for Gauss RBF: {:.3f}'.format(k_H_Gauss))
```

rcond for sigmoid: 8928603296478483456.000
rcond for Gauss RBF: 716494466506850048.000
rcond for C4 RBF: 737783083725.101

Slide 18

Question 2

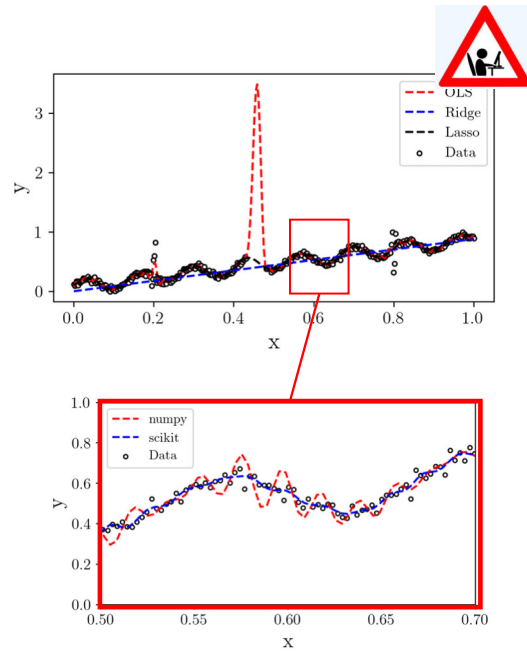
The OLS, as done in Lectures 1 and Lecture 4 reads

```
##% OS prediction for the case of
H_C4=Phi_C4_X.T@Phi_C4_X
# Train Model
w_C4=np.linalg.inv(H_C4).dot(Phi_C4_X.T).dot(y_prime)
# Make prediction on the nex x
x_test=np.linspace(0,1,300)
# Prepare the basis on the new data
Phi_X_test=PHI_C4_X(x_test, x_b, c_r=0.1)
# Predictions on the new data
y_prime_pred=Phi_X_test.dot(w_C4)
```

Using sklearn:

```
##% We can use sklearn:
from sklearn.linear_model import LinearRegression
reg = LinearRegression(fit_intercept=False).fit(Phi_C4_X, y_prime)
# look for the coefficients:
w_2=reg.coef_.T
```

You will note that the results are not the same....
This problem is very sensitive!



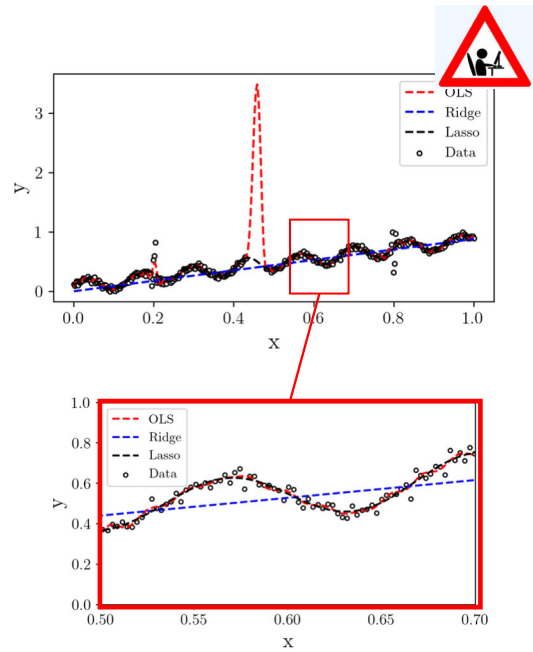
Slide 19

Question 3

We use scikit learn from now on.

```
#### Ridge Regression
from sklearn.linear_model import Ridge
Rid = Ridge(fit_intercept=False, alpha=0.01);
Rid.fit(Phi_C4_X, y_prime)
w_reg_R_s=Rid.coef_
# Regression:
y_prime_pred_L=Rid.predict(Phi_X_test)
```

```
#### Lasso Regression
from sklearn.linear_model import Lasso
Las = Lasso(fit_intercept=False, alpha=0.01);
Las.fit(Phi_C4_X, y_prime)
w_reg_L_s=Las.coef_
# Regression:
y_prime_pred_R=Las.predict(Phi_X_test)
```



Slide 20

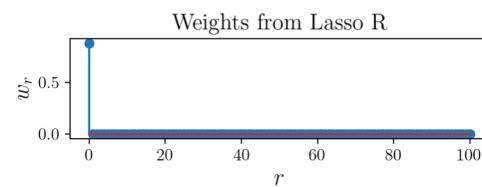
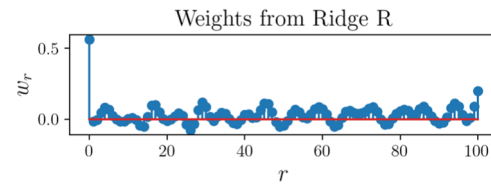
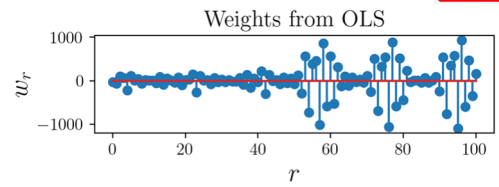
Question 3

We use scikit learn from now on.

```
##% Ridge Regression
from sklearn.linear_model import Ridge
Rid = Ridge(fit_intercept=False,alpha=0.01);
Rid .fit(Phi_C4_X, y_prime)
w_reg_R_s=Rid.coef_
# Regression:
y_prime_pred_L=Rid.predict(Phi_X_test)
```

```
##% Lasso Regression
from sklearn.linear_model import Lasso
Las = Lasso(fit_intercept=False,alpha=0.01);
Las.fit(Phi_C4_X, y_prime)
w_reg_L_s=Las.coef_
# Regression:
y_prime_pred_R=Las.predict(Phi_X_test)
```

You should know understand why Lasso regression is considered as a 'Sparsity promoting' regression!



Question 4



We now perform a K-fold validation to select the best Ridge regression parameter.

```
##% Question 4
# Here we perform the K-fold validation
K=10 # number of folds
alphas=np.linspace(0.0001,0.1,100)
J_out=np.zeros(len(alphas))
from sklearn.model_selection import KFold
# Create the k-fold split object
kf = KFold(n_splits=K, random_state=3, shuffle=True)
```

Here we prepare the kf object for the splitting, using shuffling and a given seed.

```
# Loop over given alphas
for j in range(len(alphas)):
    print('alpha '+str(j)+' of '+str(len(alphas)))
    # Select one alpha
    alpha=alphas[j]
    # Initialize the out of sample error vector
    count=0; J_out_fold=np.zeros(K)
    # Loop over the folds
    for train_index, test_index in kf.split(Phi_C4_X):
        # Get the training and test sets
        Phi_X_train, Phi_X_test = Phi_C4_X[train_index], Phi_C4_X[test_index]
        y_train, y_test = y_prime[train_index], y_prime[test_index]
        # Fit the model on the training set
        Rid = Ridge(fit_intercept=False, alpha=alpha)
        Rid.fit(Phi_X_train, y_train)
        # Test the model on the testing set
        y_prime_pred=Rid.predict(Phi_X_test)
        # Collect all the out of sample errors
        J_out_fold[count]=np.mean((y_prime_pred-y_test)**2)
    # Take the mean out of sample error over the folds
    J_out[j]=np.mean(J_out_fold)
```

Here we loop over a broad range of alpha's, and train and test the model each time

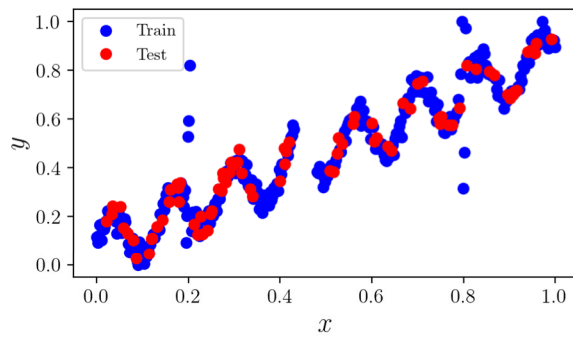
Here we loop over the folds, to train and test the model

Fit and test in each fold!

Collect out of sample error

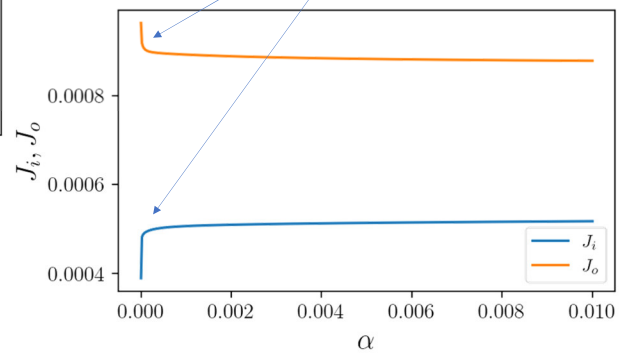
Question 4: Results

The k fold splitting



This problem is too simple to see a significant impact of alpha... but note the elbow...!

You should always be looking for these



Slide 23

Table of Contents



1. Linear Models for Regression: bases and feature scaling
2. Tikhonov and Lasso Regularization
- 3. A brief note on Support Vector Regression**
4. Physics Constrained (Informed) Regression

Slide 24

Support Vector Regression

Historically developed for classification, Support Vector Machines are also powerful tools for regression.

These are also known as Support Vector Regressor (SVR). The model is formally the same as in the linear regression tools we have seen so far, but the cost function definition is different.

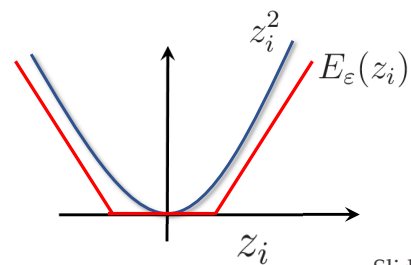
Let us define the error vector as $\mathbf{z} = \mathbf{y} - \Phi(\mathbf{x})\mathbf{w}$, i.e. $z_i = y_i - \Phi(x_i)\mathbf{w}$

Now change the cost function from this $J(\mathbf{w}) = \|\mathbf{z}\|_2^2 + \alpha \|\mathbf{w}\|_2^2$

To this: $J(\mathbf{w}) = \|\mathbf{w}\|_2^2 + C \sum_{i=0}^{n_p} E_\varepsilon(z_i(\mathbf{w}))$ $C > \mathbb{R}$

Where we introduced the ε -sensitive function

$$E_\varepsilon(z_i) = \begin{cases} 0 & \text{if } |z_i| \leq \varepsilon \\ |z_i| - \varepsilon & \text{if } |z_i| \geq \varepsilon \end{cases}$$



Slide 25

Support Vector Regression (2)

The result is thus the definition of an 'epsilon-insensitive tube' within which we would expect to have most of the data. The literature behind this optimization is vast and include amazing tricks linked to the notion of duality and kernel tricks (see [Bishop](#) and [Deisenroth et al](#))

We skip these details. Here's how to implement it in scikit-learn:

```
##% Support vector regression in python
from sklearn.svm import SVR
# Create SVR object
svr=SVR(kernel='rbf',gamma=100,C=10,epsilon=0.05)
# Fit the regressor
svr.fit(x_prime,np.ravel(y_prime))
# Make predictions:
y_p_SVM=svr.predict(x_test.reshape(-1,1))
# Look for the epsilon sensitive tube
y_p_eps=y_p_SVM+svr.epsilon
y_m_eps=y_p_SVM-svr.epsilon
```

Play with C , γ and ϵ !

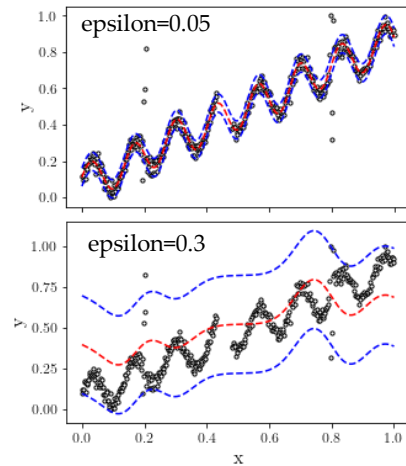


Table of Contents



1. Linear Models for Regression: bases and feature scaling
2. Tikhonov and Lasso Regularization
3. A brief note on Support Vector Regression
4. Physics Constrained (Informed) Regression

Slide 27

Physics Constrained (or informed) ML

The idea of ‘informing’ a machine learning algorithm about physics is very attractive and has been popularized by the so called ‘physics informed’ neural networks (see [Raissi et al](#)).

This idea can be viewed more formally in the framework of constrained optimization and leverages the fact that most regression tools in machine learning are differentiable. Therefore, they can be used for solving PDES in a meshless fashion (see [Sperotto et al](#))

Consider the following simple ODE:
$$\begin{cases} y'' + 2y' + y = 0 \\ y(0) = 1 \\ y(1) = 3 \end{cases} \quad \dots \text{whose solution is:} \quad y(x) = e^{-x} + (3e - 1)x e^{-x}$$

What happens if we look for a solution of the form
$$y(x) = \sum_{r=0}^{n_b-1} w_r \phi_r(x) \quad ?$$

Slide 28

From continuous to discrete (with no 'mesh'!)

For this simple example, taking an arbitrary set of n_x points (not really a mesh!), we get

$$\begin{bmatrix} \phi_0'' + 2\phi_0' + \phi_0 & \phi_1'' + 2\phi_1' + \phi_1 & \dots & \phi_{n_b}'' + 2\phi_{n_b}' + \phi_{n_b} \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix} \begin{bmatrix} w_0 \\ \vdots \\ w_{n_b} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

This is a least square problem that is very similar to the training problem.

$$\mathcal{L}(\mathbf{x}) \mathbf{w} = \mathbf{0}$$

$$\mathcal{L}(\mathbf{x}) \in \mathbb{R}^{n_x \times n_b}$$

$$\mathbf{x} \in \mathbb{R}^{n_b \times 1}$$

$$\mathbf{0} \in \mathbb{R}^{n_x \times 1}$$

We then have constraints which are also a linear:

$$\begin{bmatrix} \phi_0(0) & \phi_1(0) & \dots & \phi_{n_b}(0) \\ \phi_0(1) & \phi_1(1) & \dots & \phi_{n_b}(1) \end{bmatrix} \begin{bmatrix} w_0 \\ \vdots \\ w_{n_b} \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \Rightarrow \Phi(x_\Gamma) \mathbf{w} = \mathbf{c}$$

Slide 29

A classic constrain problem

Using again Lagrange multipliers, we seek to minimize the following:

$$J(\mathbf{w}) = \|\mathcal{L}(\mathbf{x})\mathbf{w}\|_2^2 + \lambda^T (\Phi(x_\Gamma)\mathbf{w} - \mathbf{c}) = 0$$

The minima occurs when $\nabla J(\mathbf{w}) = 0$

This takes us to (ease the notation a bit):

$$\begin{aligned}\nabla_{\mathbf{w}} J &= 2\mathcal{L}^T \mathcal{L} \mathbf{w} + \Phi_\Gamma^T \lambda = \mathbf{0} \\ \nabla_\lambda J &= \Phi_\Gamma \mathbf{w} - \mathbf{c} = \mathbf{0}\end{aligned}$$

... which we can write as

$$\begin{bmatrix} 2\mathcal{L}^T \mathcal{L} & \Phi_\Gamma^T \\ \Phi_\Gamma & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{c} \end{bmatrix}$$

An efficient algorithm to solve this problem is proposed by [Sperotto et al](#)

Slide 30

Results from the Meshless Integration

Here's the code to prepare the first and second derivatives

```
def C4_RBF_x(x, x_r=0, c_r=0.1):
    d=x-x_r # Get distance
    phi_r=(-10*d*(d-c_r)**4*(d+c_r)**4)/(c_r**10)
    phi_r[np.abs(d)>c_r]=0
    return phi_r

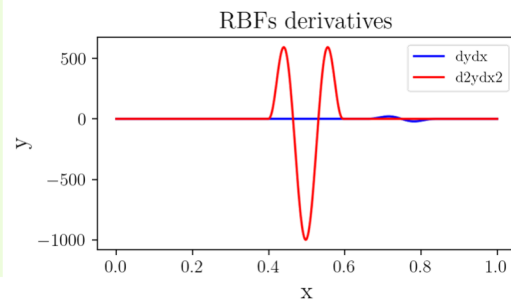
def C4_RBF_xx(x, x_r=0, c_r=0.1):
    d=x-x_r # Get distance
    phi_r=(-10*d*(d-c_r)**3*(d+c_r)**3*(3*d-c_r)*(3*d+c_r))/(c_r**10)
    phi_r[np.abs(d)>c_r]=0
    return phi_r
```

Here we prepare the related bases

```
# Function that prepares the three derivatives
def PHI_C4_X(x_in, x_b, c_r=0.1):
    n_x=np.size(x_in); n_b=len(x_b)
    Phi=np.zeros((n_x,n_b+1)) # Initialize Basis Matrix on x
    Phi_x=np.zeros((n_x,n_b+1)) # Initialize Basis Matrix on x
    Phi_xx=np.zeros((n_x,n_b+1)) # Initialize Basis Matrix on x

    # Add a constant and a linear term on Phi, which means:
    Phi[:,0]=x_in; Phi[:,1]=np.ones(len(x_in)); Phi[:,2]=np.zeros(len(x_in))
    for j in range(0,n_b): # Loop to prepare the basis matrices (inefficient)
        # Prepare all the terms for Phi, Phi_x, Phi_xx
        Phi[:,j+1]=C4_RBF(x_in,x_r=x_b[j],c_r=c_r)
        Phi_x[:,j+1]=C4_RBF_x(x_in,x_r=x_b[j],c_r=c_r)
        Phi_xx[:,j+1]=C4_RBF_xx(x_in,x_r=x_b[j],c_r=c_r)

    return Phi, Phi_x, Phi_xx
```



We will do into the details for fast solution of this system (see [Sperotto et al](#))

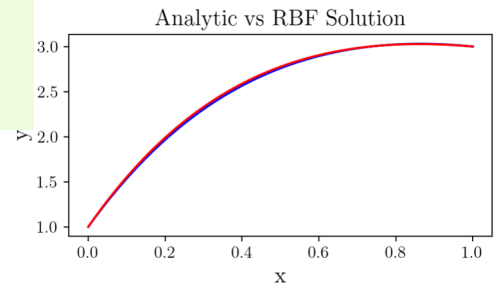
Slide 31

Results from the Meshless Integration

Here's the code to generate the linear system from the previous slide

```
%% RBF Solution procedure
Phi, Phi_x, Phi_xx=PHI_C4_X(x, x_b, c_r=0.1)
# Create the matrix L and the bloc A
L=Phi_xx+2*Phi_x+Phi; A=2*L.T@L
# Create the matrix Gamma_T
x_gamma=np.array([0,1])
Phi_Gamma, Phi_x_Gamma, Phi_xx_Gamma=PHI_C4_X(x_gamma, x_b, c_r=0.1)
# Assembly the global system
Rows_H_1=np.hstack((A,Phi_Gamma.T))
Rows_H_2=np.hstack((Phi_Gamma,np.zeros((2,2))))
A_star=np.vstack((Rows_H_1,Rows_H_2))
b_star=np.hstack((np.zeros(n_b+1),np.array([1,3])))
# Solve the system approximately using the Pseudo Inverse
x_sol=np.linalg.pinv(A_star,rcond=1e-15).dot(b_star)
```

```
# Get the weights
w=x_sol[:n_b+1]
# Assembly the solution
y_Sol_RBF=Phi.dot(w)
# Prepare the Analytic solution for comparison
y_Sol_Analyt=np.exp(-x)+(3*np.e-1)*x*np.exp(-x)
```



Note the approach based on the pseudo inverse (pinv).
This scales poorly for large problems.

A much more efficient method is described by [Sperotto et al](#)

Slide 32

Computing pressure from PTV/PIV

arXiv.org > physics > arXiv:2112.12752

Physics > Fluid Dynamics

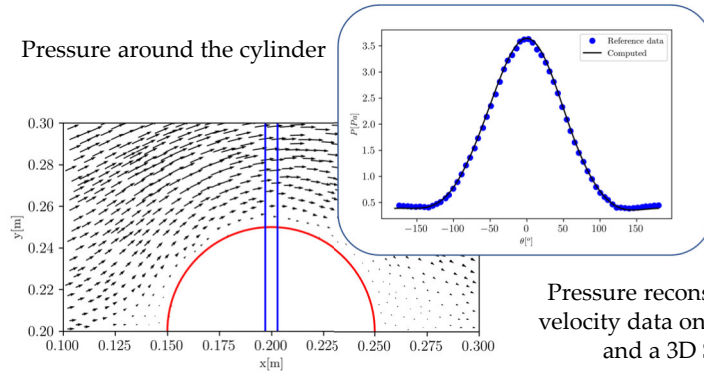
[Submitted on 23 Dec 2021]

A Meshless Method to Compute Pressure Fields from Image Velocimetry

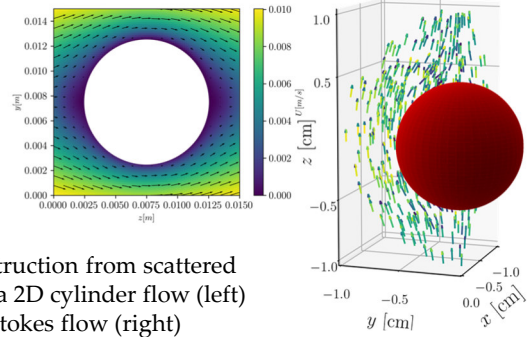
Pietro Sperotto, Sandra Pieraccini, Miguel A. Mendez



Pressure around the cylinder



Pressure field



Pressure reconstruction from scattered velocity data on a 2D cylinder flow (left) and a 3D Stokes flow (right)

Slide 33

The end!

Slide 34