Machine Learning 2023/2024 (2^{nd} semester)

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Master in Electrical and Computer Engineering

Department of Electrical and Computer Engineering

A. Pedro Aguiar (pedro.aguiar@fe.up.pt), **Aníbal Matos** (anibal@fe.up.pt), **Andry Pinto** (amgp@fe.up.pt), **Daniel Campos** (dfcampos@fe.up.pt), **Maria Inês Pereira** (maria.ines@fe.up.pt)

FEUP, Mar. 2024

Notebook 04

Part 1 - Linear Regression with regularization

The Generalization problem

Understanding the following challenges is crucial for machine learning practitioners:

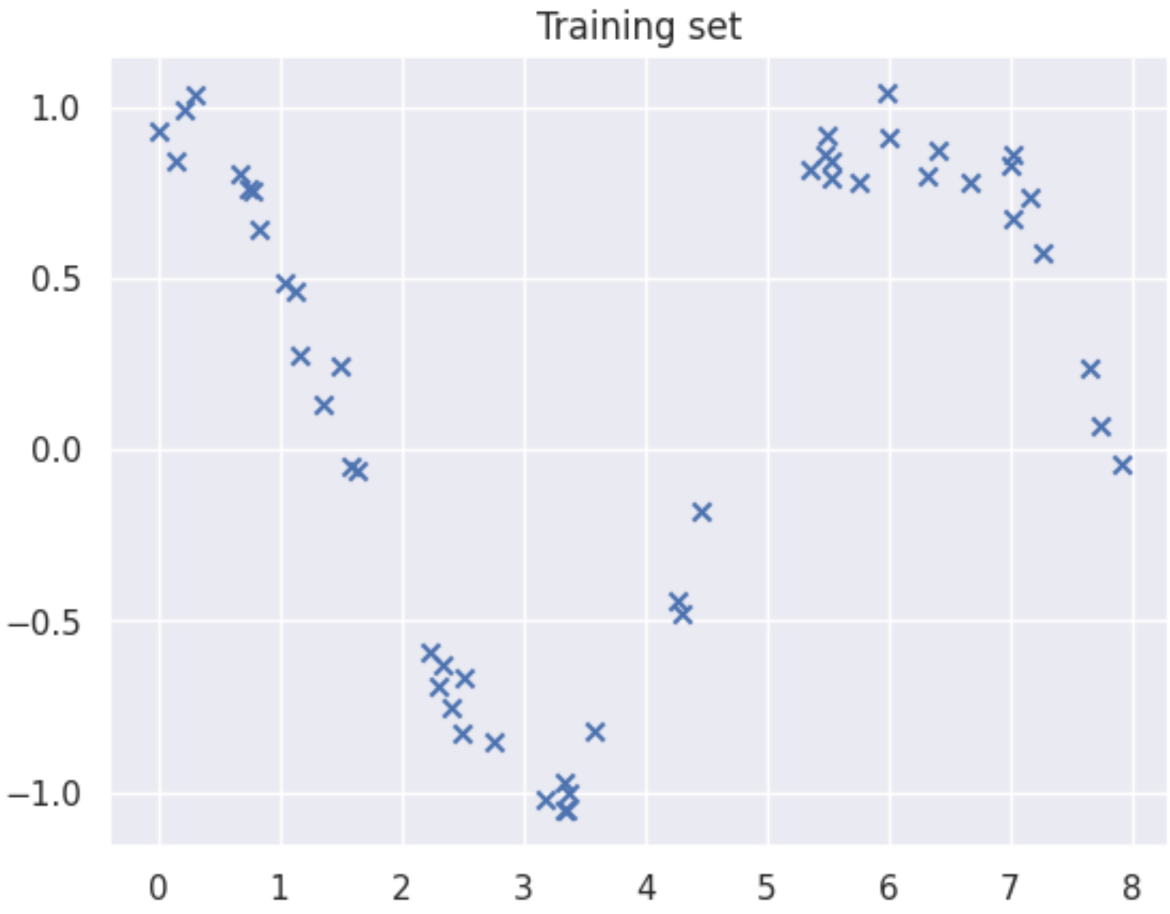
1. How does the model perform on ('training') data which has experienced before?
2. How does a model perform on previously unseen ('testing') data?
3. How should a model be selected?

Those questions constitute the basics of **underfitting** and **overfitting**.

Example:

Let's produce noisy data that will be fitted later on by different models (with polynomial and Gaussian kernels).

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
from scipy import linalg
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures

rng = np.random.RandomState(1)

size_training = 50
x_limit    = 8

#Create some noisy data that will be used for trainning models
x_training = x_limit * rng.rand(size_training)
y_training = np.sin(x_training+3.1415/2) + 0.1 * rng.randn(size_training)   # s

plt.scatter(x_training, y_training, marker="x")
plt.title("Training set");
```

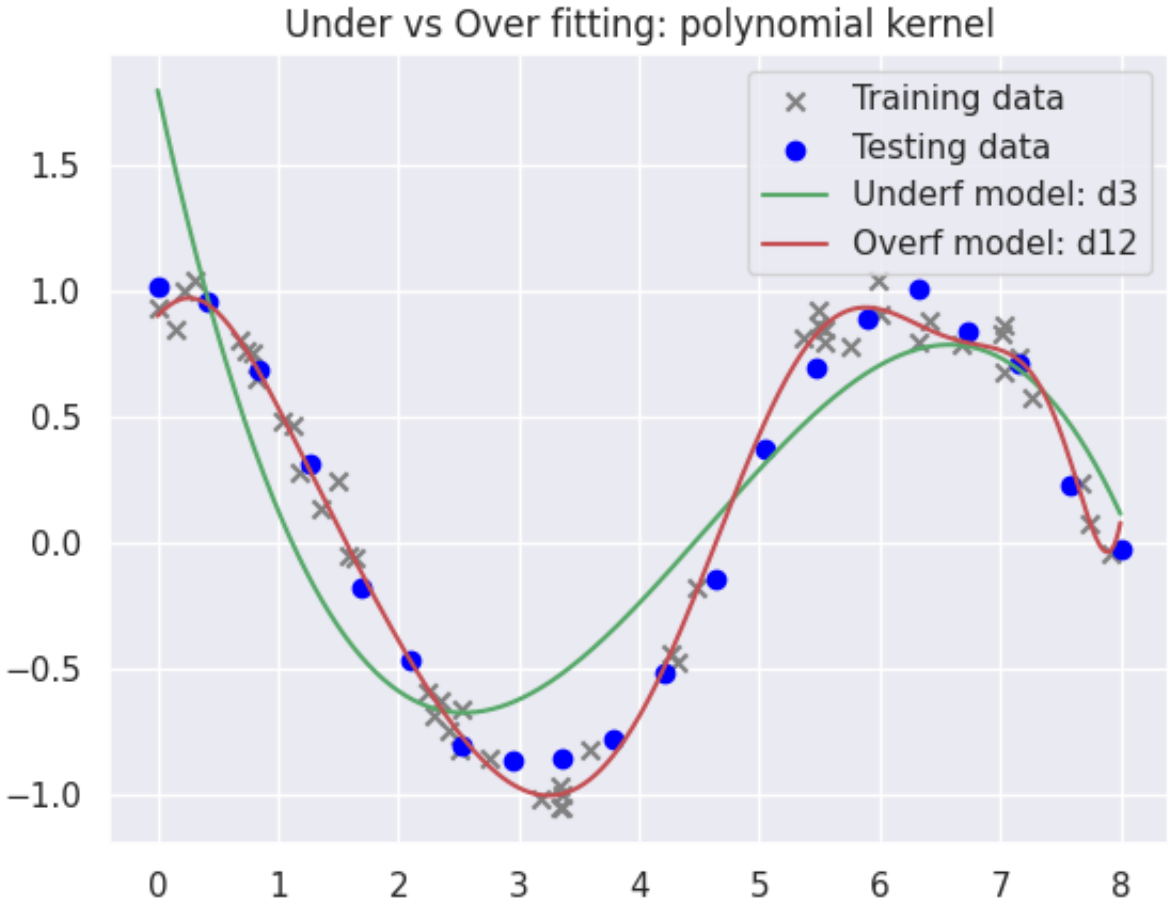


1) Using models based on Polynomial Kernels to fit this training set.

```
# Using sklearn
from sklearn import linear_model
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score
```

```
#testing set
size_testing   = 20
x_testing      = np.linspace(0, x_limit, size_testing)
y_testing      = np.sin(x_testing+3.1415/2)  + 0.1 * rng.randn(size_testing)

# continous model (for drawing)
draw_size = 500
x_draw = np.linspace(0,x_limit,draw_size)

#Overfitting Model
nb_degree_over       = 12
model_overfitting    = make_pipeline(PolynomialFeatures(nb_degree_over), Linear
model_overfitting.fit(x_training[:, np.newaxis], y_training)
print("Training MSE (Poly Model Overfitting):", mean_squared_error(y_training,

y_overfitting        = model_overfitting.predict(x_testing[:, np.newaxis])
m_error_overfitting  = mean_squared_error(y_testing, y_overfitting)
print("Testing MSE (Poly Model Overfitting):", m_error_overfitting)

y_over_draw = model_overfitting.predict(x_draw[:, np.newaxis])

#Underfitting Model
nb_degree_under       = 3
model_underfitting    = make_pipeline(PolynomialFeatures(nb_degree_under), Line
model_underfitting.fit(x_training[:, np.newaxis], y_training)
print("Training MSE (Poly Model Underfitting):", mean_squared_error(y_training,

y_underfitting        = model_underfitting.predict(x_testing[:, np.newaxis])
m_error_underfitting  = mean_squared_error(y_testing, y_underfitting)
print("Testing MSE (Poly Model Underfitting):", m_error_underfitting)

y_under_draw = model_underfitting.predict(x_draw[:, np.newaxis])

#Plots
legend_fitting_model_overfitting = 'Overf model: d{}'.format(nb_degree_over)
legend_fitting_model_underfitting = 'Underf model: d{}'.format(nb_degree_under)
plt.scatter(x_training, y_training,  color="gray", marker="x", label='Training
plt.scatter(x_testing, y_testing,  color="blue", marker="o", label='Testing dat
plt.plot(x_draw, y_under_draw, color="g", label=legend_fitting_model_underfitti
plt.plot(x_draw, y_over_draw, color="r", label=legend_fitting_model_overfitting
plt.legend(loc ="best")
plt.title("Under vs Over fitting: polynomial kernel")
plt.show()
```

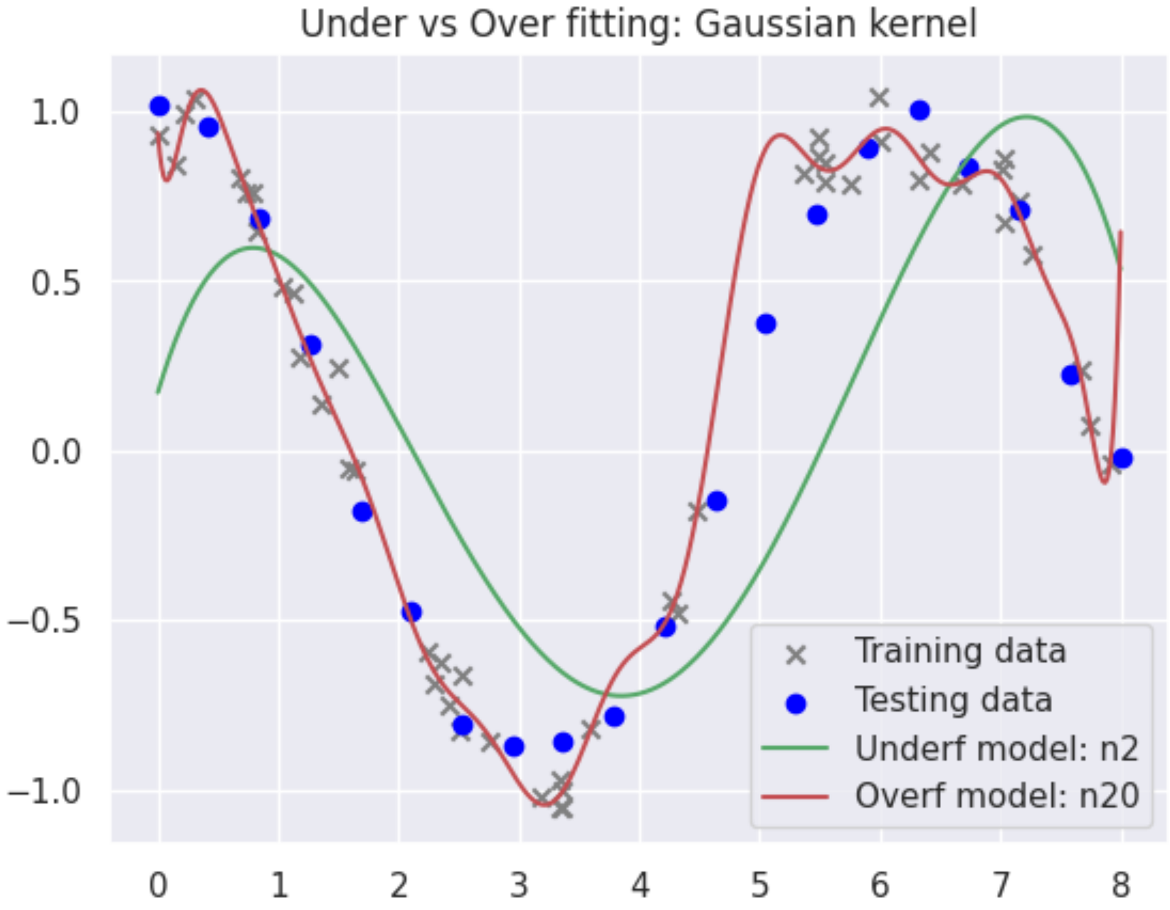
```
Training MSE (Poly Model Overfitting): 0.0042804398954628745
Testing MSE (Poly Model Overfitting): 0.007293292125722
Training MSE (Poly Model Underfitting): 0.11017920596696776
Testing MSE (Poly Model Underfitting): 0.09202245464570646
```



2) Using models based on Gaussian Kernels to fit this training set.

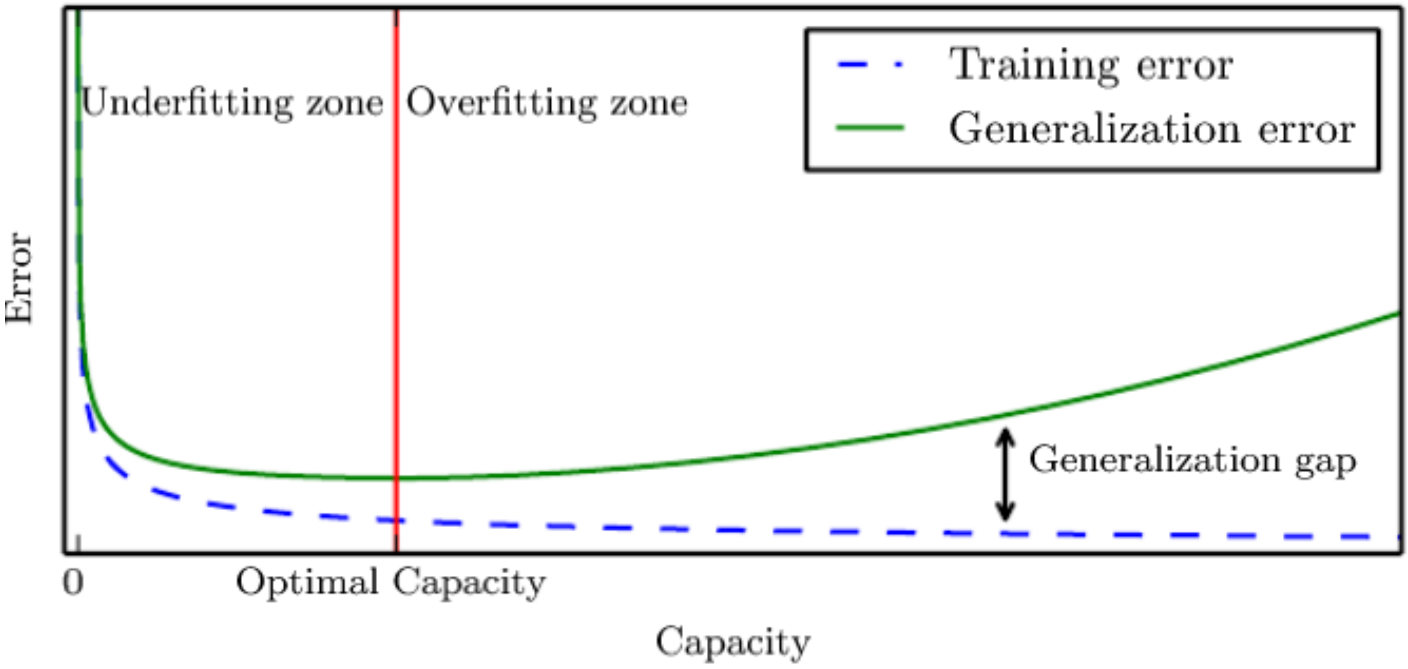
```
from sklearn.base import BaseEstimator, TransformerMixin
"""
  Gaussian models
  -> Uniformly spaced
  -> One-dimension input
"""
class GaussianBases(BaseEstimator, TransformerMixin):
    def __init__(self, N, width_kernel = 3.0):
        self.N = N
        self.width_kernel = width_kernel

    @staticmethod
    def _gauss_basis(x, y, width = 3.0, sigma_f=1, axis=None):
        arg = (x - y) / width
        return sigma_f*np.exp(-0.5 * np.sum(arg ** 2, axis))/(2 * width**2)  ##
    def fit(self, X, y=None):
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_kernel * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_, self.width

#Overfitting Model
n_models_gauss_over    = 20
gauss_model_overfitting = make_pipeline(GaussianBases(n_models_gauss_over), Lin
gauss_model_overfitting.fit(x_training[:, np.newaxis], y_training)
print("Training MSE (Gauss Model Overfitting):", mean_squared_error(y_training,

yfit_overfitting        = gauss_model_overfitting.predict(x_testing[:, np.newax
m_error_overfitting     = mean_squared_error(y_testing, yfit_overfitting)
print("Testing MSE (Gauss Model Overfitting):", m_error_overfitting)

ydraw_over = gauss_model_overfitting.predict(x_draw[:, np.newaxis])

#Underfitting Model
n_models_gauss_under     = 2
gauss_model_underfitting = make_pipeline(GaussianBases(3), LinearRegression())
gauss_model_underfitting.fit(x_training[:, np.newaxis], y_training)
print("Training MSE (Gauss Model Underfitting):", mean_squared_error(y_training

yfit_underfitting        = gauss_model_underfitting.predict(x_testing[:, np.new
m_error_underfitting     = mean_squared_error(y_testing, yfit_underfitting)
print("Testing MSE (Gauss Model Underfitting):", m_error_underfitting)

ydraw_under = gauss_model_underfitting.predict(x_draw[:, np.newaxis])

#Plots
legend_fitting_model_overfitting = 'Overf model: n{}'.format(n_models_gauss_ove
legend_fitting_model_underfitting = 'Underf model: n{}'.format(n_models_gauss_u
plt.scatter(x_training, y_training,  color="gray", marker="x", label='Training
plt.scatter(x_testing, y_testing,  color="blue", marker="o", label='Testing dat
plt.plot(x_draw, ydraw_under, color="g", label=legend_fitting_model_underfittin
plt.plot(x_draw, ydraw_over, color="r", label=legend_fitting_model_overfitting,
plt.legend(loc ="best")
plt.title("Under vs Over fitting: Gaussian kernel")
plt.show()
```

```
Training MSE (Gauss Model Overfitting): 0.0027875487865052385
Testing MSE (Gauss Model Overfitting): 0.04623096259514714
Training MSE (Gauss Model Underfitting): 0.23090736685120572
Testing MSE (Gauss Model Underfitting): 0.21898176325660207
```



Underfitting means that the model is too simple and there is still progress to be made because the training error (considering a loss function) is high well after training the model with several examples.

Overfitting means that the model is fitting the training data more closely than the underlying phenomenon that is being captured and, therefore, there is a large difference between training (small) and testing error (high). It usually appears when the model fits the training data much better than new test examples - lack of generalization capacity.



source: (Goodfellow et al., 2016)

A model that is highly flexible will learn spurious (or irrelevant) patterns as easily as the true associations. Thus, the model will be unable to generalizes well for unseen data. The best option to prevent a model from learning misleading patterns is to get more training data (which can be problematic for some cases). There are techniques that can be used to combat overfittiting - a high model complexity.

Regularization

A Regularization technique aims to reduce overfitting by penalizing the **model complexity**. Recalling that the linear regression minimizes the following loss function:

$$J(\theta) = \sum_{n=1}^{N} (y_n - \theta^T \mathbf{x}_n)^2$$

The searching space of θ can be restricted by controlling $||\theta||^2$ or $||\theta||_1$.

Ridge regression (ℓ_2 Regularization)

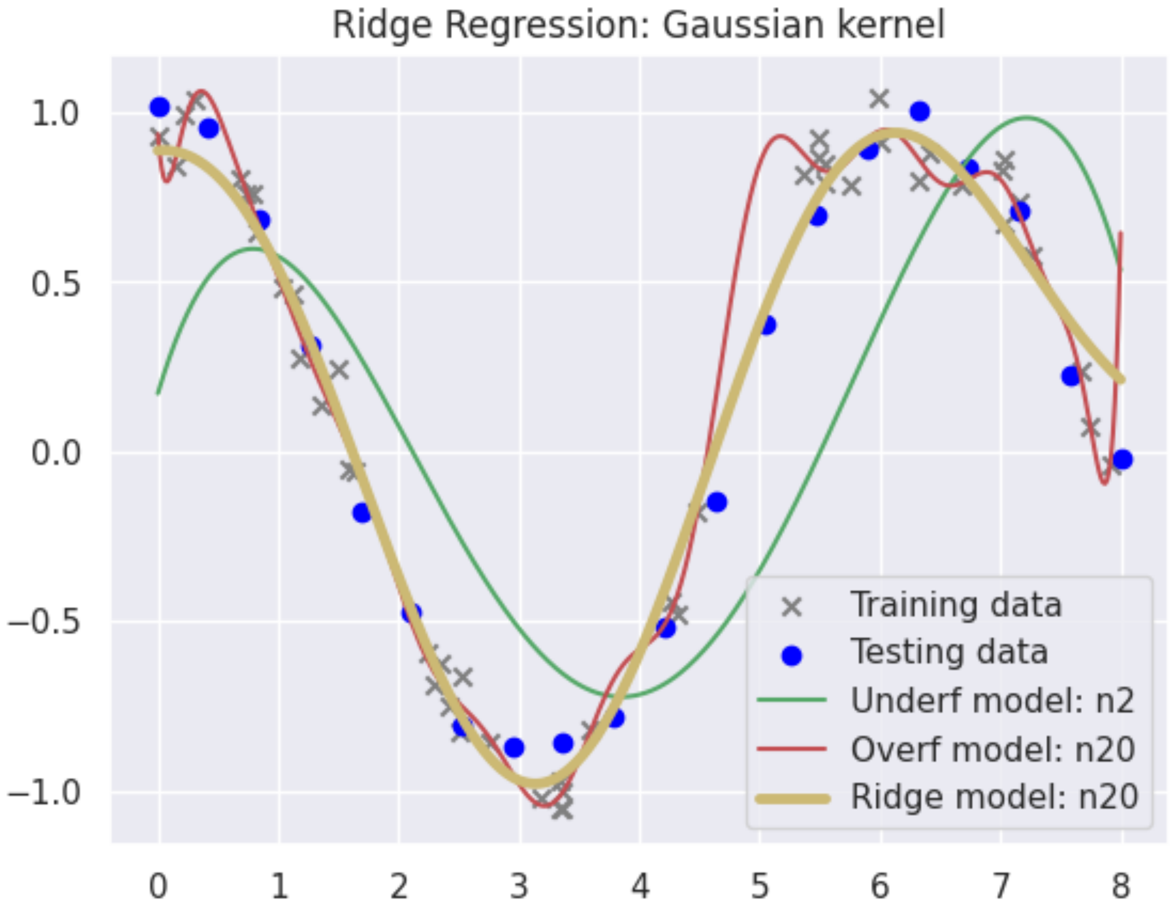
The *Ridge regression* is a ℓ_2 *regularization* that penalizes the sum of squares (ℓ_2-norms) of the model coefficients:

$$J_\lambda(\theta) = \sum_{n=1}^{N}(y_n - \theta^T \mathbf{x}_n)^2 + \lambda||\theta||^2$$

where $\lambda \geq 0$ is a free parameter that quantifies a penalty for the norm of the parameter vector. Ridge regression tries to keep the sum of squared errors small and, at the same time, it attempts to reduce the norm of the estimated vector. This type of penalized model is built into sklearn with the `Ridge` estimator:

```
from sklearn.linear_model import Ridge

#Ridge Model
gauss_model_ridge = make_pipeline(GaussianBases(n_models_gauss_over), Ridge(alp
gauss_model_ridge.fit(x_training[:, np.newaxis], y_training)
print("Training MSE (Ridge Model):", mean_squared_error(y_training, gauss_model
print("Testing MSE (Ridge Model):", mean_squared_error(y_testing, gauss_model_r

ydraw_ridge = gauss_model_ridge.predict(x_draw[:, np.newaxis])


#Plots
legend_fitting_model_ridge = 'Ridge model: n{}'.format(n_models_gauss_over)
plt.scatter(x_training, y_training,  color="gray", marker="x", label='Training 
plt.scatter(x_testing, y_testing,  color="blue", marker="o", label='Testing dat
plt.plot(x_draw, ydraw_under, color="g", label=legend_fitting_model_underfittin
plt.plot(x_draw, ydraw_over, color="r", label=legend_fitting_model_overfitting,
plt.plot(x_draw, ydraw_ridge, color="y", label=legend_fitting_model_ridge, line
plt.legend(loc ="best")
plt.title("Ridge Regression: Gaussian kernel")
plt.show()
```

```
Training MSE (Ridge Model): 0.010903318981812961
Testing MSE (Ridge Model): 0.010359667098565731
```



LASSO regression (ℓ_1 regularization)

The LASSO regularization involves penalizing the sum of absolute values (ℓ_1-norm) of regression coefficients:

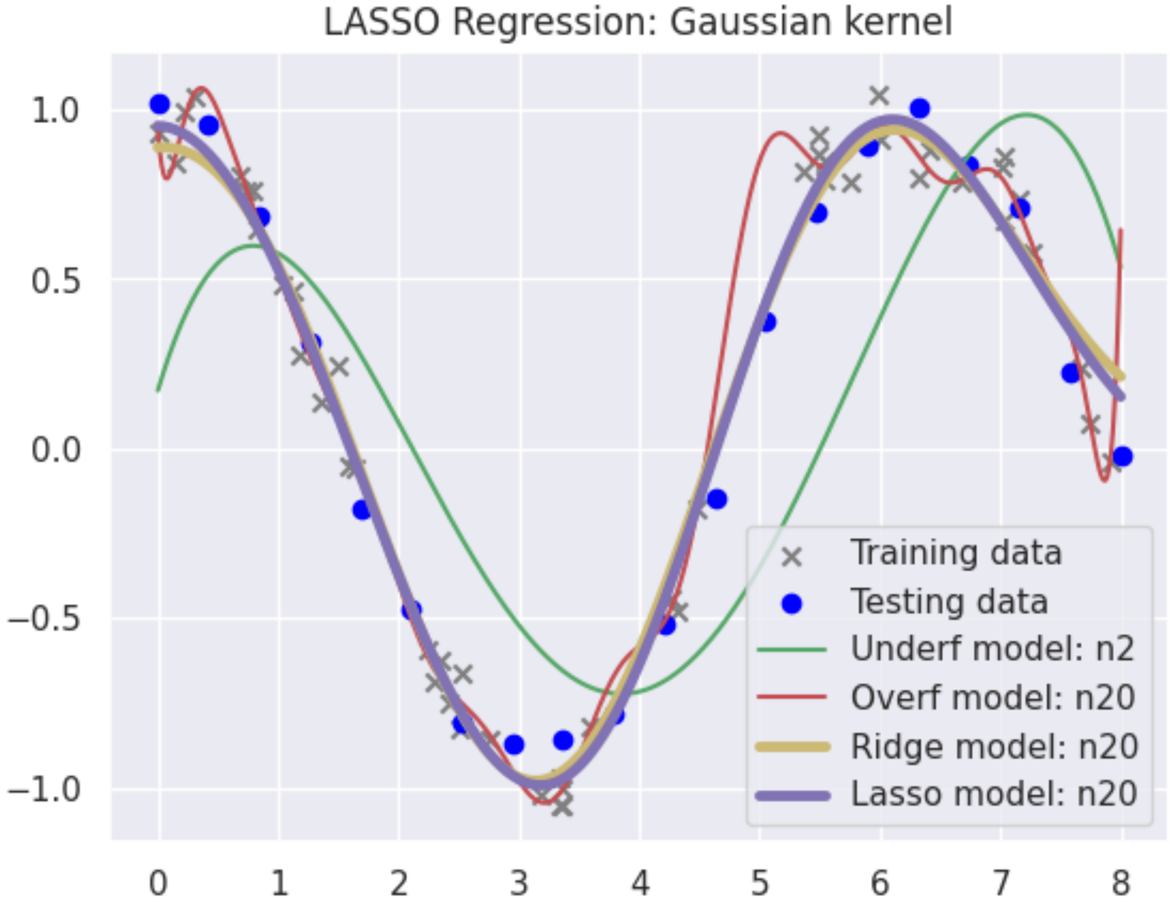
$$J_\lambda(\theta) = \sum_{n=1}^{N}(y_n - \theta^T \mathbf{x}_n)^2 + \lambda\|\theta\|_1$$

where $\|\cdot\|_1$ is the ℓ_1-norm (that is, $\|\theta\|_1 = \sum_{i=1}^{d} \theta_i$) and $\lambda \geq 0$ is a hyperparameter that tunes the strength of the penalty, and should be determined via, for example, cross-validation. Lasso regression tends to favor *sparse models* where possible: that is, it preferentially sets model coefficients to exactly zero. This type of penalized model is built into sklearn with the `Lasso` estimator:

```
from sklearn.linear_model import Lasso

#Lasso Model
```

```
gauss_model_lasso = make_pipeline(GaussianBases(n_models_gauss_over), Lasso(alp
gauss_model_lasso.fit(x_training[:, np.newaxis], y_training)
print("Training MSE (LASSO Model):", mean_squared_error(y_training, gauss_model
print("Testing MSE (LASSO Model):", mean_squared_error(y_testing, gauss_model_l

ydraw_lasso = gauss_model_lasso.predict(x_draw[:, np.newaxis])

#Plots
legend_fitting_model_lasso = 'Lasso model: n{}'.format(n_models_gauss_over)
plt.scatter(x_training, y_training,  color="gray", marker="x", label='Training
plt.scatter(x_testing, y_testing,  color="blue", marker="o", label='Testing dat
plt.plot(x_draw, ydraw_under, color="g", label=legend_fitting_model_underfittir
plt.plot(x_draw, ydraw_over, color="r", label=legend_fitting_model_overfitting,
plt.plot(x_draw, ydraw_ridge, color="y", label=legend_fitting_model_ridge, line
plt.plot(x_draw, ydraw_lasso, color="m", label=legend_fitting_model_lasso, line
plt.legend(loc ="best")
plt.title("LASSO Regression: Gaussian kernel")
plt.show()
```

```
Training MSE (LASSO Model): 0.010903318981812961
Testing MSE (LASSO Model): 0.007439415832263855
```



The estimated vector θ for the Linear Regression, Ridge and Lasso models can be seen below. The majority of the coefficients are exactly zero with the LASSO regression penalty.

```
print("Model (Overfit):\n", gauss_model_overfitting.named_steps['linearregressi
print("Ridge Model:\n", gauss_model_ridge.named_steps['ridge'].coef_)
print("LASSO Model:\n", gauss_model_lasso.named_steps['lasso'].coef_)
```

```
Model (Overfit):
 [-3.717e+07  2.627e+08 -9.827e+08  2.549e+09 -5.094e+09  8.261e+09
 -1.116e+10  1.267e+10 -1.195e+10  8.950e+09 -4.489e+09 -6.236e+07
  3.402e+09 -4.853e+09  4.564e+09 -3.270e+09  1.818e+09 -7.584e+08
  2.157e+08 -3.212e+07]
Ridge Model:
 [ 1.004  0.917  0.674  0.317 -0.089 -0.477 -0.794 -0.996 -1.041 -0.899
 -0.572 -0.115  0.368  0.753  0.937  0.873  0.582  0.149 -0.306 -0.664]
LASSO Model:
 [ 3.035  0.531  0.     0.     0.    -0.    -0.    -0.    -3.173 -0.
 -0.    -0.     0.     0.     3.156  0.699  0.     0.    -0.    -0.   ]
```

Activity 1.1

In the example above test several penalties for the Ridge and LASSO regresssions and compare the mean square errors of training and testing sets.

For Ridge test with λ=0.01, λ=0.1, λ=1 and λ=10.

For Lasso test with λ=0.0001, λ=0.001, λ=0.01 and λ=0.1.

```
# code for activity 1.1
# solution 1.1

#Ridge Model
for a in [0.01, 0.1, 1, 10] :
  print("(Ridge) alpha=", a)
  gauss_model_ridge = make_pipeline(GaussianBases(n_models_gauss_over), Ridge(a
  gauss_model_ridge.fit(x_training[:, np.newaxis], y_training)
  print("  Training MSE:", mean_squared_error(y_training, gauss_model_ridge.pre
  print("  Testing MSE :", mean_squared_error(y_testing, gauss_model_ridge.prec

#Lasso Model
for a in [0.0001, 0.001, 0.01, 0.1] :
  print("(LASSO) alpha=", a)
  gauss_model_lasso = make_pipeline(GaussianBases(n_models_gauss_over), Lasso(a
  gauss_model_lasso.fit(x_training[:, np.newaxis], y_training)
  print("  Training MSE:", mean_squared_error(y_training, gauss_model_lasso.pre
  print("  Testing MSE :", mean_squared_error(y_testing, gauss_model_lasso.prec
```

```
(Ridge) alpha= 0.01
  Training MSE: 0.008053237966544727
  Testing MSE : 0.005462014355738859
(Ridge) alpha= 0.1
  Training MSE: 0.010903318981812961
  Testing MSE : 0.010359667098565731
(Ridge) alpha= 1
  Training MSE: 0.05632997997047238
  Testing MSE : 0.05073004195505988
(Ridge) alpha= 10
  Training MSE: 0.29351449910627253
  Testing MSE : 0.25773242698292087
(LASSO) alpha= 0.0001
  Training MSE: 0.007488561678321872
  Testing MSE : 0.004455933922947243
(LASSO) alpha= 0.001
  Training MSE: 0.008914237356444752
  Testing MSE : 0.0074394158322638555
(LASSO) alpha= 0.01
  Training MSE: 0.06568650481471255
  Testing MSE : 0.07355361298863375
(LASSO) alpha= 0.1
  Training MSE: 0.5114696052079297
  Testing MSE : 0.4522947801076927
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_coordinate_de
  model = cd_fast.enet_coordinate_descent(
```

Activity 1.2

Use the classic Auto MPG dataset to build models for predicting the fuel efficiency of cars from the late-1970s and early 1980s. Before starting, it is crucial to preprocess the data because contains missing information, presence of categorical values, and features at different scales. Please analyse and execute the following code that preprocesses data accordingly.

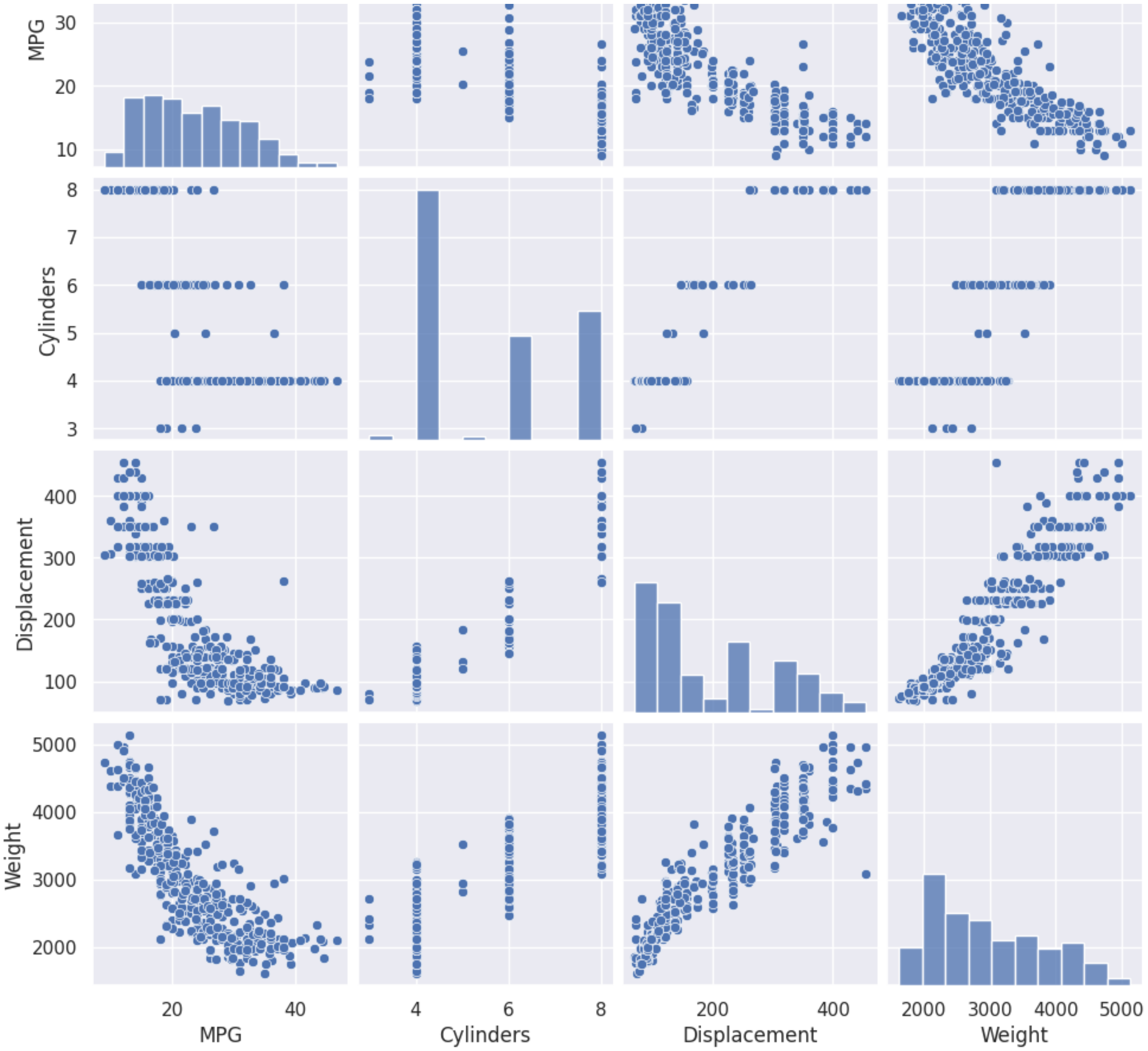
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
# Make NumPy printouts easier to read.
np.set_printoptions(precision=3, suppress=True)

# Download dataset
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-m
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
                'Acceleration', 'Model Year', 'Origin']

raw_dataset = pd.read_csv(url, names=column_names,
                          na_values='?', comment='\t',
                          sep=' ', skipinitialspace=True)

dataset = raw_dataset.copy()

# Drop NaN values
dataset = dataset.dropna()

# One-hot encoding for "Origin" column, which is categorical and not numeric.
dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3: 'Japan'})
dataset = pd.get_dummies(dataset, columns=['Origin'], prefix='', prefix_sep='')

dataset
#dataset.tail()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	E
0	18.0	8	307.0	130.0	3504.0	12.0	70	
1	15.0	8	350.0	165.0	3693.0	11.5	70	
2	18.0	8	318.0	150.0	3436.0	11.0	70	
3	16.0	8	304.0	150.0	3433.0	12.0	70	
4	17.0	8	302.0	140.0	3449.0	10.5	70	
...	...	...	...	...	...	...	...	
393	27.0	4	140.0	86.0	2790.0	15.6	82	
394	44.0	4	97.0	52.0	2130.0	24.6	82	
395	32.0	4	135.0	84.0	2295.0	11.6	82	
396	28.0	4	120.0	79.0	2625.0	18.6	82	
397	31.0	4	119.0	82.0	2720.0	19.4	82	

392 rows × 10 columns

```
# Uncomment to Inspect Training Data
sns.pairplot(dataset[['MPG', 'Cylinders', 'Displacement', 'Weight']])
dataset.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	
MPG	392.0	23.445918	7.805007	9.0	17.000	22.75	29.000	
Cylinders	392.0	5.471939	1.705783	3.0	4.000	4.00	8.000	
Displacement	392.0	194.411990	104.644004	68.0	105.000	151.00	275.750	4
Horsepower	392.0	104.469388	38.491160	46.0	75.000	93.50	126.000	2
Weight	392.0	2977.584184	849.402560	1613.0	2225.250	2803.50	3614.750	51
Acceleration	392.0	15.541327	2.758864	8.0	13.775	15.50	17.025	
Model Year	392.0	75.979592	3.683737	70.0	73.000	76.00	79.000	
Europe	392.0	0.173469	0.379136	0.0	0.000	0.00	0.000	
Japan	392.0	0.201531	0.401656	0.0	0.000	0.00	0.000	
USA	392.0	0.625000	0.484742	0.0	0.000	1.00	1.000	



1.2.1. Normalize the dataset based on the minimum and maximum value of each feature ("Displacement", "Horsepower", "Weight", "Acceleration") and label ("MPG"). Divide the original dataset into the training set (70%) and a testing set (30%).

```
#1.2.1 to complete

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

All_Data  = dataset.to_numpy()

# Normalize features. see: https://scikit-learn.org/stable/modules/generated/sk
#scaler              =
#All_Data_normalized =

#Create X matrix and Y vector
Y   = All_Data_normalized[:, 0:1] #Label: "MPG"
X   = All_Data_normalized[:, 2:6] #Selecting only the features "Displacement, H
feature_custom = np.add(X[:, 1], X[:, 3]).reshape(-1, 1)  # Add feature: HorseP
X = np.concatenate((X, feature_custom), axis = 1 )
# Feature vector: "Displacement", "Horsepower", "Weight", "Acceleration", "Hors

# Split data into train and test set.
#X_train, X_test, y_train, y_test = train_test_split(...)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-7b313f305416> in <cell line: 13>()
     11 
     12 #Create X matrix and Y vector
---> 13 Y   = All_Data_normalized[:, 0:1] #Label: "MPG"
     14 X   = All_Data_normalized[:, 2:6] #Selecting only the features
"Displacement, Horsepower, Weight, Acceleration"
     15 feature_custom = np.add(X[:, 1], X[:, 3]).reshape(-1, 1)  # Add
feature: HorsePower + Acceleration

NameError: name 'All Data normalized' is not defined
```

```
#Solution 1.2.1

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

All_Data  = dataset.to_numpy()

# Normalize features. see: https://scikit-learn.org/stable/modules/generated/sk
scaler              = MinMaxScaler()
All_Data_normalized = scaler.fit_transform(All_Data)


#Create X matrix and Y vector
Y   = All_Data_normalized[:, 0:1] #Label: "MPG"
X   = All_Data_normalized[:, 2:6] #Selecting only the features "Displacement, H
feature_custom = np.add(X[:, 1], X[:, 3]).reshape(-1, 1)  # Add feature: HorseP
X = np.concatenate((X, feature_custom), axis = 1 )
# Feature vector: "Displacement", "Horsepower", "Weight", "Acceleration", "Hors

# Split data into train and test set.
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random
```

1.2.2. Determine a solution for the regression model :

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4 + a_5 x_5$$

where x_1, x_2, x_3, x_4 and x_5 is the displacement, horsepower, weight, acceleration and horsepower+acceleration of a car, respectively. The solution that was obtained is unique?

```
# 1.2.2 to complete

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

#model =
#model.fit(...)
#yfit = model.predict(...)

# The mean squared error
print("Training MSE (Linear Model): %.2f" % mean_squared_error(y_train, model.p
print("Testing MSE (Linear Model): %.2f" % mean_squared_error(y_test, yfit))

# Show model
print("Model: ", model.intercept_, model.coef_)

#Singular values
print("Model Singular values:", model.singular_)
```

```
# Solution 1.2.2

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

model = LinearRegression(fit_intercept=True)
model.fit(X_train, y_train)
yfit = model.predict(X_test)

# The mean squared error
print("Training MSE (Linear Model): %.2f" % mean_squared_error(y_train, model.p
print("Testing MSE (Linear Model): %.2f" % mean_squared_error(y_test, yfit))

# Show model
print("Model: ", model.intercept_, model.coef_)

#Singular values
print("Model Singular values:", model.singular_)
```

```
    Training MSE (Linear Model): 0.01
    Testing MSE (Linear Model): 0.01
    Model:  [0.7] [[ 9.740e-03  1.445e+12 -5.906e-01  1.445e+12 -1.445e+12]]
    Model Singular values: [6.738 2.815 1.365 0.904 0.   ]
```

1.2.3. Implement a Ridge Regressor ($X^\top X + \lambda I$) and determine a solution for the previous problem. Explain whether $X^\top X + \lambda I$ is invertible.

```
# 1.2.3 to complete

from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error

#Ridge Model
#model_ridge =
#model_ridge.fit(...)
#yfit_ridge  = model_ridge.predict(...)

# The mean squared error
print("Training MSE (Ridge Model): %.2f" % mean_squared_error(y_train, model_ri
print("Testing MSE (Ridge Model): %.2f" % mean_squared_error(y_test, yfit_ridge

# Show model
print("Model: ", model_ridge.intercept_, model_ridge.coef_)
```

```
# Solution 1.2.3

from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error

#Ridge Model
model_ridge = Ridge(alpha=0.1)
model_ridge.fit(X_train, y_train)
yfit_ridge  = model_ridge.predict(X_test)

# The mean squared error
print("Training MSE (Ridge Model): %.2f" % mean_squared_error(y_train, model_ri
print("Testing MSE (Ridge Model): %.2f" % mean_squared_error(y_test, yfit_ridge

# Show model
print("Model: ", model_ridge.intercept_, model_ridge.coef_)
```

```
    Training MSE (Ridge Model): 0.01
    Testing MSE (Ridge Model): 0.01
    Model:  [0.704] [[-0.018 -0.14  -0.554  0.041 -0.099]]
```

1.2.4 Repeat the previous exercise, but using LASSO regularization.

```
# 1.2.4 to complete ...
# Solution 1.2.4

from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error

#Ridge Model
model_lasso = Lasso(alpha=0.0001)
model_lasso.fit(X_train, y_train)
yfit_lasso  = model_lasso.predict(X_test)

# The mean squared error
print("Training MSE (LASSO Model): %.2f" % mean_squared_error(y_train, model_la
print("Testing MSE (LASSO Model): %.2f" % mean_squared_error(y_test, yfit_lasso

# Show model
print("Model: ", model_lasso.intercept_, model_lasso.coef_)
```

```
    Training MSE (LASSO Model): 0.01
    Testing MSE (LASSO Model): 0.01
    Model:  [0.697] [-0.    -0.173 -0.587  0.    -0.043]
```

Activity 1.3

Derive the least squares optimal solution with ridge regularization for the model

$$f_\theta(x) = \theta^\top \phi(x)$$

where $\theta = [\theta_0, \theta_1]^\top$, $\phi(x) = [\phi_0(x), \phi_1(x)]^\top$, and $x \in \mathbb{R}^d$. In other words, apply the (necessary) optimality condition to

$$\min_\theta \left(\sum_{n=1}^{N} \left(y_n - \theta^\top \phi(x_n) \right)^2 + \lambda \|\theta\|^2 \right)$$

Solution

Let's consider a more general case where $\theta = [\theta_0, \ldots, \theta_{K-1}]$ and $\phi(x) = [\phi_0(x), \ldots, \phi_{K-1}(x)]^\top$ so that the linear regression with ridge regularization corresponds to obtain $\hat{\theta}$ that minimizes $J(\theta)$ defined by

$$J(\theta) = \sum_{n=1}^{N} \left(y_n - \theta^\top \phi(x_n)\right)^2 + \lambda \|\theta\|^2$$

Defining

$$Y := \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_N \end{bmatrix}$$

and

$$\Phi := \begin{bmatrix} \phi(x_1)^\top \\ \phi(x_2)^\top \\ \cdots \\ \phi(x_n)^\top \end{bmatrix} = \begin{bmatrix} \phi_0(x_1) & \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_{K-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_{K-1}(x_2) \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \phi_0(x_n) & \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_{K-1}(x_n) \end{bmatrix}$$

we have

$$J(\theta) = (Y - \Phi\theta)^\top (Y - \Phi\theta) + \lambda\theta^\top\theta = Y^\top Y - 2Y^\top \Phi\theta + \theta^\top \Phi^\top \Phi\theta + \lambda\theta^\top\theta$$

and, therefore,

$$\nabla J = -2\Phi^\top Y + 2\Phi^\top\Phi\theta + 2\lambda\theta.$$

At the minimum, $\nabla J = 0$, resuklting in $(\Phi^\top\Phi + \lambda I)\hat{\theta} = \Phi^\top Y$.

As $\Phi^\top\Phi$ is at least semi-definite positive, $\Phi^\top\Phi + \lambda I$ is non-singular, so

$$\hat{\theta} = (\Phi^\top\Phi + \lambda I)^{-1}\Phi^\top Y$$

Part 2 - Estimation

Given N observations, $\mathcal{X} = \{x_1, x_2, \ldots, x_N\}$, drawn from a probability distribution. Assume that the joint pdf of these N observations is of a known parametric functional type, denoted as $p(\mathcal{X}; \theta)$, where the parameter $\theta \in \mathbb{R}^K$ is unknown. The task is to estimate its value.

Maximum Likelihood Estimation

The joint pdf, $p(\mathcal{X};\theta)$, is known as the **likelihood function** of θ with respect to the given set of observations, $\mathcal{X}$. According to the maximum likelihood method, the estimate is provided by

$$\hat{\theta}_{\mathrm{ML}} = \arg\max_{\theta}\, p(\mathcal{X};\theta).$$

Since the logarithmic function is monotone and increasing, one can instead search for the maximum of the **log-likelihood function**, that is,

$$\left.\frac{\partial \ln p(\mathcal{X};\theta)}{\partial\theta}\right|_{\theta=\hat{\theta}_{\mathrm{ML}}} = 0.$$

Activity 2.1

The exponential distribution with parameter $\lambda > 0$ is characterized by the probability function

$$p(x;\lambda) = \lambda e^{-\lambda x}, \qquad x \geq 0.$$

Obtain the maximum likelihood estimate of λ based on observations $\{x_i\}_{i=1}^{N}$ of this distribution.

Solution

Assuming that observations are independent, we have

$$p(\mathcal{X};\lambda) = \prod_{i=1}^{N} p(x_i;\lambda) = \prod_{i=1}^{N}\left(\lambda e^{-\lambda x_i}\right)$$

and

$$\ln p(\mathcal{X};\lambda) = \sum_{i=1}^{N}(\ln\lambda - \lambda x_i) = N\ln\lambda - \lambda\sum_{i=1}^{N} x_i.$$

Therefore,

$$\frac{d\ln p(\mathcal{X};\lambda)}{d\lambda} = \frac{N}{\lambda} - \sum_{i=1}^{N} x_i$$

The maximum likelihood estimate, λ_{ML}, satisfies

$$\frac{N}{\lambda_{\mathrm{ML}}} - \sum_{i=1}^{N} x_i = 0$$

or

$$\lambda_{\mathrm{ML}} = \frac{N}{\sum_{i=1}^{N} x_i}.$$

Maximum A-Posteriori Probability Estimation

The Maximum A-Posteriori Probability estimation technique, usually denoted as MAP, is based on the Bayesian theorem, but it does not go as far as the Bayesian philosophy allows to. The goal becomes that of obtaining an estimate by maximizing

$$ \hat{\theta}_{\mathrm{MAP}} = \mathrm{arg}\max_\theta{p(\theta|\mathcal{X})} = \mathrm{arg}\max_\theta \frac{p(\mathcal{X}|\theta)p(\theta)}{p(\mathcal{X})}.$$$$ \hat{\theta}_{\mathrm{MAP}} = \mathrm{arg}\max_\theta{p(\theta|\mathcal{X})} = \mathrm{arg}\max_\theta \frac{p(\mathcal{X}|\theta)p(\theta)}{p(\mathcal{X})}.$$

Since $p(\mathcal{X})$$p(\mathcal{X})$ is independent of $\theta$$\theta$, this leads to

$$ \hat{\theta}_{\mathrm{MAP}} = \mathrm{arg}\max_\theta p(\mathcal{X}|\theta)p(\theta) = \mathrm{arg}\max_\theta \{\ln{ p(\mathcal{X}|\theta)} + \ln{p(\theta)}\}.$$$$ \hat{\theta}_{\mathrm{MAP}} = \mathrm{arg}\max_\theta p(\mathcal{X}|\theta)p(\theta) = \mathrm{arg}\max_\theta \{\ln{ p(\mathcal{X}|\theta)} + \ln{p(\theta)}\}.$$

Activitiy 2.2

Assume that $\{x_1,x_2,\ldots,x_N\}$$\{x_1,x_2,\ldots,x_N\}$ are i.i.d. observations from a Gaussian distribution $\mathcal{N}(\mu,\sigma^2)$$\mathcal{N}(\mu,\sigma^2)$. Obtain the MAP estimate of $\mu$$\mu$ if the prior follows the exponential distribution $$ p(\mu) = \lambda e^{-\lambda\mu}, \lambda > 0, \mu\geq 0.$$$$ p(\mu) = \lambda e^{-\lambda\mu}, \lambda > 0, \mu\geq 0.$$

Solution

The pdf of the Gaussian distribution $p(x|\mu)$$p(x|\mu)$ is $$ p(x|\mu) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$$$ p(x|\mu) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$ so $$ p(\mathcal{X}|\mu) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2}$$$$ p(\mathcal{X}|\mu) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x_i-\mu}{\sigma}\right)^2}$$ and \begin{align} \ln p(\mathcal{X}|\mu) & = \sum_{i=1}^N\left(-\ln\left(\sqrt{2\pi}\sigma\right) - \frac12\left(\frac{x_i-\mu}{\sigma}\right)^2\right)\\ & = -N \ln\left(\sqrt{2\pi}\sigma\right)-\frac12\sum_{i=1}^N \left(\frac{x_i-\mu}{\sigma}\right)^2. \end{align}\begin{align} \ln p(\mathcal{X}|\mu) & = \sum_{i=1}^N\left(-\ln\left(\sqrt{2\pi}\sigma\right) - \frac12\left(\frac{x_i-\mu}{\sigma}\right)^2\right)\\ & = -N \ln\left(\sqrt{2\pi}\sigma\right)-\frac12\sum_{i=1}^N \left(\frac{x_i-\mu}{\sigma}\right)^2. \end{align} Since $\ln p(\mu) = \ln\lambda-\lambda\mu$$\ln p(\mu) = \ln\lambda-\lambda\mu$, $\mu_\mathrm{MAP}$$\mu_\mathrm{MAP}$ maximizes $$ M(\mu) = \ln p(\mathcal{X}|\mu)+\ln p(\mu) = -N \ln\left(\sqrt{2\pi}\sigma\right)-\frac12\sum_{i=1}^N \left(\frac{x_i-\mu}{\sigma}\right)^2 + \ln\lambda-\lambda\mu.$$$$ M(\mu) = \ln p(\mathcal{X}|\mu)+\ln p(\mu) = -N \ln\left(\sqrt{2\pi}\sigma\right)-\frac12\sum_{i=1}^N \left(\frac{x_i-\mu}{\sigma}\right)^2 + \ln\lambda-\lambda\mu.$$ The derivative is $$ \frac{dM}{d\mu} = \sum_{i=1}^N\frac{x_i-\mu}{\sigma^2} - \lambda = -\lambda + \frac{1}{\sigma^2}\left(\sum_{i=1}^Nx_i - N\mu\right).$$$$ \frac{dM}{d\mu} = \sum_{i=1}^N\frac{x_i-\mu}{\sigma^2} - \lambda = -\lambda + \frac{1}{\sigma^2}\left(\sum_{i=1}^Nx_i - N\mu\right).$$ so, $\mu_\mathrm{MAP}$$\mu_\mathrm{MAP}$ is given by $$ \mu_\mathrm{MAP} = \frac{\left(\sum_{i=1}^Nx_i\right) - \sigma^2\lambda}{N}.$$$$ \mu_\mathrm{MAP} = \frac{\left(\sum_{i=1}^Nx_i\right) - \sigma^2\lambda}{N}.$$