Lab Session –II(b) Regularization in Linear Regression
(Ridge and LASSO)

Dr. JASMEET SINGH
ASSISTANT PROFESSOR, CSED
TIET, PATIALA

Regularization

- Regularization is a technique used for tuning the function by adding an additional penalty term in the error function that reduces the magnitude of parameters .
- In regression models, we do not know which regression coefficients we should shrink by adding their penalty in the cost function.
- So, the general tendency of applying regularization in regression is to shrink the weight (regression coefficients) of all the input variables.
- Two most commonly used regularization in linear regression are:

1. Ridge Regression (L2 Normalization)
2. Least Absolute Selection and Shrinkage Operator (LASSO) Regression (L! Normalization)

Ridge Regression

- Ridge regression performs '**L2 regularization**', i.e. it adds a factor of sum of squares of coefficients in the optimization objective.
- Ridge Regression using **gradient descent optimization** works as follows:

1. Initialize $\beta_0 = 0$, $\beta_1 = 0$, $\beta_2 = 0$,…………………………… $\beta_k = 0$
2. Update parameters until convergence or for fixed number of iterations using following equation:

$$\beta_j = \beta_j\left(1 - \frac{\alpha\lambda}{n}\right) - \frac{\alpha}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \cdots \ldots \ldots \ldots .. + \beta_k x_{ik} - y_i) \times x_{ij}$$

For j=0,1,2,3……………..k

where $x_{i0}=1$ and k are the total number of features; α *is the learning rate and* λ is the regularization parameter

Ridge Regression- Contd….

Ridge Regression using **least square error fit (LSE)** works as follows:

- The optimal value of β is computed as $\boldsymbol{\beta\hat{}}$=$(\boldsymbol{X^T\ X + \lambda I})^{-1}\boldsymbol{X^T y}$.
- It will solve the problem of overfitting and multicollinearity as $|\ X^T\ X + \lambda I|$ will not be zero for correlated features.
- where, ; y= $\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_n \end{bmatrix}$; β = $\begin{bmatrix} \beta_0 \\ \beta_2 \\ \vdots \\ \beta_k \end{bmatrix}$ and $X = \begin{bmatrix} 1 & x_{11} & x_{12} \cdots & x_{1k} \\ 1 & x_{21} & x_{22} \cdots & x_{2k} \\ \vdots & \vdots & \vdots \quad \vdots & \vdots \\ 1 & x_{n1} & x_{n2} \cdots & x_{nk} \end{bmatrix}$
- $\alpha\ is\ the\ learning\ rate\ and\ \lambda$ is the regularization parameter

Implementation Ridge Regression (LSE Fit) (Step-by-Step)

- Following steps are followed for implementation of least square error fit:

1. Load the dataset.
2. Handle Null Values. remove noise, outliers.
3. Separate the dataset into X (input/independent variables) and Y (dependent feature). Scale the feature values of X in a fixed range and add a new column (in the beginning) with all values 1.
4. Split the dataset into train and test set.
5. Using the train set to find the optimal value of $\boldsymbol{\beta}\hat{}$ matrix (regression coefficients) for which cost function is minimum.
6. Predict the values of the output variable on the test set.
7. Perform the performance evaluation of the trained model.

Step 1 (Ridge-LSE): Load the Dataset

- For implementation of Ridge Regressio using least square error fit we will generate a dataset with highly correlated values.
- We have simulated a sine curve (between 60° and 300°) and added some random noise.
- Let's try to estimate the sine function using **polynomial regression** with powers of x from 1 to 15.
- Let's add a column for each power upto 15 in our dataframe.
- Since the data is generated by ourself, so there is no missing value . Noise is handled by regularization only. So, step two is not required.

Code:

```
x = np.array([i*np.pi/180 for i in range(60,300,4)])

np.random.seed(10)  #Setting seed for reproducibility

y = np.sin(x) + np.random.normal(0,0.15,len(x))

df= pd.DataFrame(np.column_stack([x,y]),columns=['x','y'])

for i in range(2,16):  #power of 1 is already there

    colname = 'x_%d'%i      #new var will be x_power

    df[colname] = df['x']**i

print(df)
```

Step 3 (Ridge-LSE): Split Input & Output Features

- Separate the dataset into X (input/independent variables) and Y (dependent feature).
- Scale the feature values of X in a fixed range.
- Add a new column (in the beginning) with all values 1.

Code:

```
X=df.drop(['y'],axis=1)

Y=df.iloc[:,1]

from sklearn.preprocessing import StandardScaler

scaler=StandardScaler()

X_scaled=scaler.fit_transform(X)

X_scaled=np.insert(X_scaled,0,values=1,axis=1)
```

Step 4 (Ridge-LSE): Train/Test Split

- We can split the train and test sets using train/test split of sklearn.model_selection as follows:

Code:

from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.3, random_state=42)

Parameters:

test_size is the percentage of test set from the total dataset; random_state s used for initializing the internal **random** number generator, which will decide the **splitting** of data into **train** and **test** indices

If random_state is None or np.random, then a randomly-initialized RandomState object is returned.

If random_state is an integer, then it is used to seed a new RandomState object.

Step 5 (Ridge-LSE): Finding Regression Coefficients

- Compute the regression coefficients for which cost function of Ridge Regressionis minimum.
- According to least square error method, the mean square error is minimum when $\hat{\beta} = (X^T X + \lambda I)^{-1} X^T y$

Code:

```
lamba=0.001
A=X_train.T.dot(X_train)
B=A+lamba*I
C=np.linalg.inv(B)
D=C.dot(X_train.T)
beta=D.dot(Y_train)
```

Step 6 (Ridge-LSE) : Predicting values on test set

- In this step, we predict the values of output variable on the test set.
- It is done by multiplying X_test set with optimal Beta matrix as hown in the code below.

Code:

```
Y_predict=X_test.dot(beta)

print(Y_predict)
```

Step 7 (LSE): Performance Evaluation

- We check the performance of the trained model by computing following error between the predicted and actual values:
- Mean Square Error
- Root Mean Square Error
- R2_score

Code:

```
error=Y_test-Y_predict
square_error=np.power(error,2)
sum_square_error=np.sum(square_error)
mean_square_error=sum_square_error/len(y_predict)
print(mean_square_error)
rms_error=np.sqrt(mean_square_error)
print(rms_error)
y_mean=np.mean(Y_test)
total_variance=np.sum((Y_test-y_mean)**2)
print(1-sum_square_error/total_variance)
```

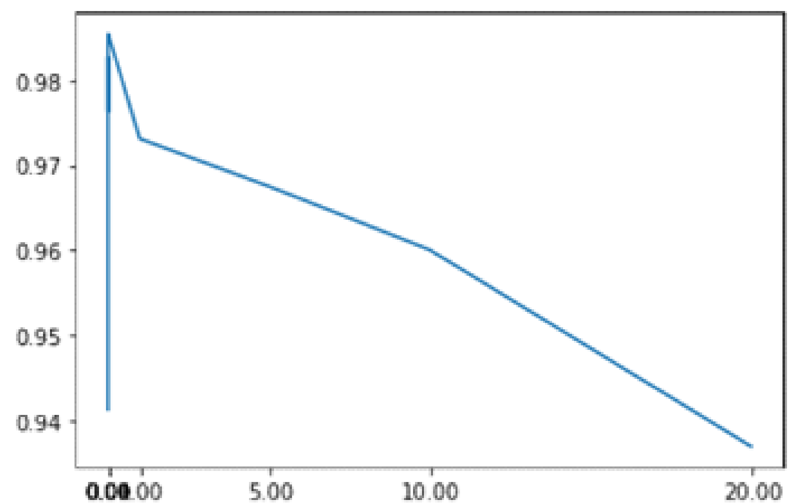
Finding value of Regularization Parameter (λ)

We can consider number of values of regularization parameter and then can check the value for which R2_score is maximum or cost of ridge regression is minimum.

```
lambda_ridge = [1e-15, 1e-10, 1e-8, 1e-4, 1e-3,1e-2, 1, 5, 10, 20]
from sklearn import metrics
R2_score=[]
for i in range(len(lambda_ridge)):
   A=X_train.T.dot(X_train)
   B=A+lambda_ridge[i]*I
   C=np.linalg.inv(B)
   D=C.dot(X_train.T)
   beta=D.dot(Y_train)
   Y_predict=X_test.dot(beta)
   R2_score.append((metrics.r2_score(Y_test,Y_predict)))
```

```
import matplotlib.pyplot as plt

plt.plot(lambda_ridge,R2_score)
```



Inbuilt Ridge Function

Inbuilt Function:

sklearn.linear_model.Ridge(alpha=1.0, *, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None)

Paramter

solver*{‘auto’, ‘svd’, “lsqr’, ‘sparse_cg’, ‘sag’, ‘saga’}, default=’auto’*

svd’ uses a Singular Value Decomposition

‘lsqr’ uses the dedicated regularized least-squares routine

‘sag’ uses a Stochastic Average Gradient descent, saga’ uses its improved, unbiased version

Example:

```
from sklearn.linear_model import Ridge

from sklearn import metrics

ridge=Ridge(alpha=0.001, normalize=True)

model=ridge.fit(X_train,Y_train)

Y_predict=model.predict(X_test)

print(metrics.r2_score(Y_test,Y_predict))
```

LASSO Regression

- LASSO regression performs '**L1 regularization**', i.e. it adds a factor of sum of absolute values of coefficients in the optimization objective
- Therefore, the regression coefficients are updated as follows in LASSO regression for fixed number of iterations:

$$\beta_j = \begin{cases} \dfrac{c_{ij}}{d_{ij}} + \dfrac{\lambda}{2} & if\ c_{ij} < -\dfrac{\lambda}{2} \\ 0 & if\ -\dfrac{\lambda}{2} \leq c_{ij} \leq \dfrac{\lambda}{2} \\ \dfrac{c_{ij}}{d_{ij}} - \dfrac{\lambda}{2} & if\ c_{ij} > \dfrac{\lambda}{2} \end{cases}$$

where, $c_{ij} = \sum_{i=1}^{n} x_{ij} \times (y_i - \sum_{h \neq j} \beta_h\, x_{ih})$ and $d_{ij} = \sum_{1=1}^{n} x_{ij}^2$ for j=0,1,2….k

Inbuilt LASSO Function

Inbuilt Function:

sklearn.linear_model.Lasso(alpha=1.0, *, fit_intercept=True, normalize=False, precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic')

Example:

```
from sklearn.linear_model import Lasso

from sklearn import metrics

lasso=Lasso(alpha=0.001, normalize=True)

model=lasso.fit(X_train,Y_train)

Y_predict=model.predict(X_test)

print(metrics.r2_score(Y_test,Y_predict))
```