Module 5 : Regression

- Regression is a modeling technique for predicting quantitative-valued target attributes.
- The goals for this tutorial are as follows:
  1. To provide examples of using different regression methods from the scikit-learn library package.
  2. To demonstrate the problem of model overfitting due to correlated attributes in the data.
  3. To illustrate how regularization can be used to avoid model overfitting.
- Reference Book

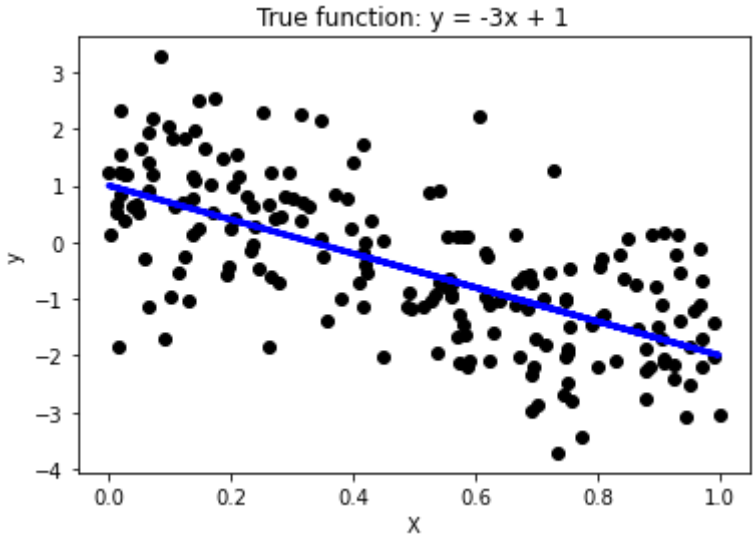
5.1 Synthetic Data Generation

Linear Regression

- Generate a random 1 - dimensional vector of predictor variables, x, from a uniform distribution.
- Linear Relationship equation : y = -3x + 1 + epsilon, where epsilon corresponds to random noise sampled from a Gaussian distribution with mean 0 and standard deviation of 1.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

seed = 1 # seed for random number generation
numInstances = 200 # number of data instances
np.random.seed(seed)
X = np.random.rand(numInstances,1).reshape(-1,1)
y_true = -3 * X + 1
y = y_true + np.random.normal(size = numInstances).reshape(-1,1)

plt.scatter(X,y,color='black')
plt.plot(X,y_true,color='blue',linewidth=3)
plt.title('True function: y = -3x + 1')
plt.xlabel('X')
plt.ylabel('y')
```

```
Text(0, 0.5, 'y')
```



5.2 Multiple Liner Regression

- Target : use python scikit-learn package to fit a multiple linear regression (MLR) model.
- Given a training set {X,y}, MLR is designed to learn the regression function $f(X, w) = X^T w + w_0$ by minimizing the following loss function given a training set $\{X_i, y_i\}_{i=1}^N$:
  $L(y, f(X, w)) = \sum_{i=1}^n \|y_i - X_i w - w_0\|^2$
  where w(slope) and w_o (intercept) are the regression coefficients.
- Given the inpu dataset, the following steps are performed:
  1. Split the input data into their respective training and test set.
  2. Fit multiple linear regression to the training data.
  3. Apply the model to the test data.
  4. Evaluate the performance of the model.
  5. Postprocessing : Visualizing the fitted model.

Step 1 : Split Input Data into Training and Test Sets

```
numTrain = 20 # number of training instances
numTest = numInstances - numTrain

X_train = X[:-numTest]
X_test = X[-numTest:]
y_train = y[:-numTest]
y_test = y[-numTest:]
```

Step2 : Fit Regression Model to Training Set

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error,r2_score

# Create linear regression object
regr = linear_model.LinearRegression()

# Fit regression model to the training set
regr.fit(X_train,y_train)
```

```
LinearRegression()
```

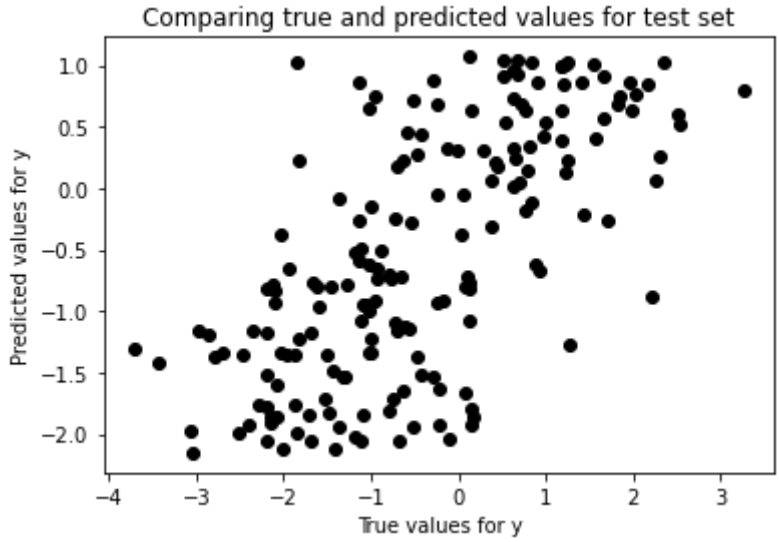
Step3:Apply Model to the Test Set

```
# Apply model to the test set
y_pred_test = regr.predict(X_test)
```

Step4:Evaluate Model Performance on Test Set

```
# Comparing true versus predicted values
plt.scatter(y_test,y_pred_test,color='black')
plt.title('Comparing true and predicted values for test set')
plt.xlabel('True values for y')
plt.ylabel('Predicted values for y')

# Model evaluation
print("Root mean squared error =
%.4f"%np.sqrt(mean_squared_error(y_test,y_pred_test)))
print('R-squared = $.4f'%r2_score(y_test,y_pred_test))
```

```
Root mean squared error = 1.0476
R-squared = $.4f
```

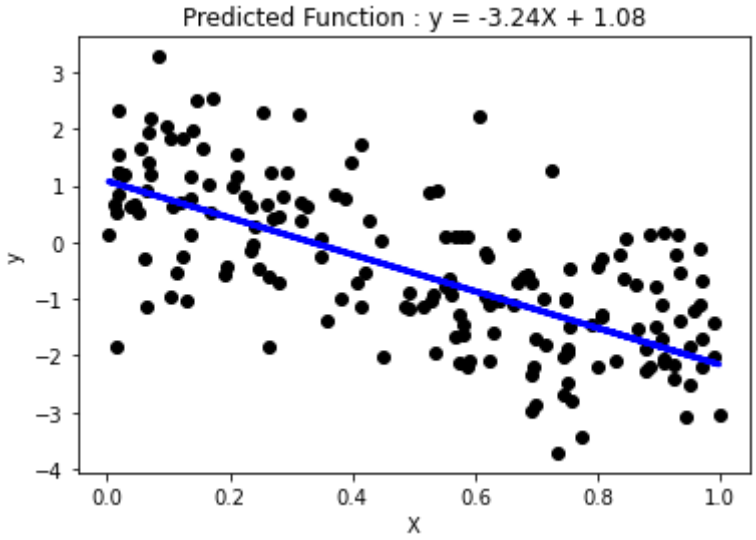


Step5:Postprocessing

```
# Display model parameters
print('Slope = ', regr.coef_[0][0])
print("Intercept = ",regr.intercept_[0]) # Step 4 : Postprocessing

# Plot outputs
plt.scatter(X_test,y_test,color="black")
plt.plot(X_test,y_pred_test,color='blue',linewidth=3)
titlestr = 'Predicted Function : y = %.2fX + %.2f'%
(regr.coef_[0],regr.intercept_[0])
plt.title(titlestr)
plt.xlabel('X')
plt.ylabel('y')
```

```
Slope =  -3.242354544656501
Intercept =  1.0805993038584834
```

```
Text(0, 0.5, 'y')
```



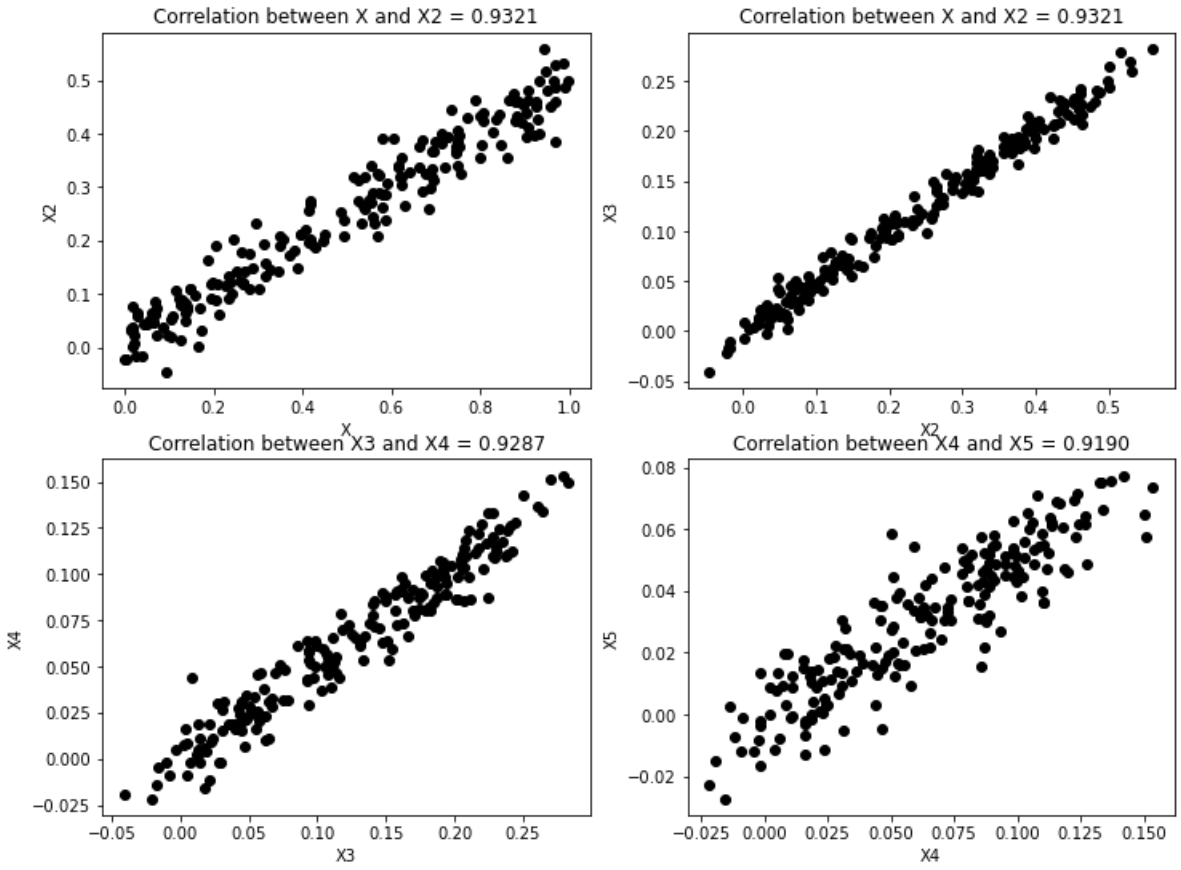
5.3 Effect of Correlated Attributes

- Target : Illustrate how the presence of correlated attributes can affect the performance of regression models.
- Create 4 additional variables : X_2, X_3, X_4 and X_5 that are strongly correlated with the previous variable X created in Section 5.1.
- The relationship between X and y remains the same as before.
- Then fit y against the predictor variables and compare their training and test set errors.

```
# First : create the correlated attributes below
seed = 1
np.random.seed(seed)
X2 = 0.5*X + np.random.normal(0,0.04,size=numInstances).reshape(-1,1)
X3 = 0.5*X2 + np.random.normal(0,0.01,size=numInstances).reshape(-1,1)
X4 = 0.5*X3 + np.random.normal(0,0.01,size=numInstances).reshape(-1,1)
X5 = 0.5*X4 + np.random.normal(0,0.01,size=numInstances).reshape(-1,1)

fig,((ax1,ax2),(ax3,ax4)) = plt.subplots(2,2,figsize=(12,9))
ax1.scatter(X,X2,color='black')
ax1.set_xlabel('X')
ax1.set_ylabel('X2')
c = np.corrcoef(np.column_stack((X[:-numTest],X2[:-numTest])).T)
titlestr = 'Correlation between X and X2 = %.4f'%(c[0,1])
ax1.set_title(titlestr)


ax2.scatter(X2,X3,color='black')
ax2.set_xlabel('X2')
ax2.set_ylabel('X3')
c = np.corrcoef(np.column_stack((X2[:-numTest],X3[:-numTest])).T)
titlesttr = 'Correlation between X2 and X3 = %.4f'%(c[0,1])
ax2.set_title(titlestr)

ax3.scatter(X3,X4,color='black')
```

```
ax3.set_xlabel('X3')
ax3.set_ylabel('X4')
c = np.corrcoef(np.column_stack((X3[:-numTest],X4[:-numTest])).T)
titlestr = 'Correlation between X3 and X4 = %.4f'%(c[0,1])
ax3.set_title(titlestr)

ax4.scatter(X4,X5,color='black')
ax4.set_xlabel('X4')
ax4.set_ylabel('X5')
c = np.corrcoef(np.column_stack((X4[:-numTest],X5[:-numTest])).T)
titlestr = 'Correlation between X4 and X5 = %.4f'%(c[0,1])
ax4.set_title(titlestr)
```

```
Text(0.5, 1.0, 'Correlation between X4 and X5 = 0.9190')
```



Create 4 additional versions of the training and test sets.

The first version, X_train2 and X_test2 have 2 correlated predictor variables, X and X2.

The second version, X_train3 and X_test3 have 3 correlated predictor variables, X, X2 and X 3.

The third version have 4 correlated variables, X,X2,X3 and X4 whereas the last version have 5 correlated variables,X,X2,X3,X4 and X5.

```
X_train2 = np.column_stack((X[:-numTest],X2[:-numTest]))
X_test2 = np.column_stack((X[-numTest:],X2[-numTest:]))

X_train3 = np.column_stack((X[:-numTest],X2[:-numTest],X3[:-numTest]))
X_test3 = np.column_stack((X[-numTest:],X2[-numTest:],X3[-numTest:]))

X_train4 = np.column_stack((X[:-numTest],X2[:-numTest],X3[:-numTest],X4[:-numTest]))
X_test4 = np.column_stack((X[-numTest:],X2[-numTest:],X3[-numTest:],X4[-numTest:]))

X_train5 = np.column_stack((X[:-numTest],X2[:-numTest],X3[:-numTest],X4[:-numTest],X5[:-numTest
]))
X_test5 = np.column_stack((X[-numTest:],X2[-numTest:],X3[-numTest:],X4[-numTest:],X5[-numTest
:]))
```

Train 4 new regression models based on the 4 versions of training and test data created i the previous step.

```
regr2 = linear_model.LinearRegression()
regr2.fit(X_train2, y_train)
regr3 = linear_model.LinearRegression()
regr3.fit(X_train3, y_train)
regr4 = linear_model.LinearRegression()
regr4.fit(X_train4, y_train)
regr5 = linear_model.LinearRegression()
regr5.fit(X_train5, y_train)
```

```
LinearRegression()
```

All 4 versions of the regression models are then applied to the training and test sets.

```
y_pred_train = regr.predict(X_train)
y_pred_test = regr.predict(X_test)
y_pred_train2 = regr2.predict(X_train2)
y_pred_test2 = regr2.predict(X_test2)
y_pred_train3 = regr3.predict(X_train3)
y_pred_test3 = regr3.predict(X_test3)
y_pred_train4 = regr4.predict(X_train4)
y_pred_test4 = regr4.predict(X_test4)
y_pred_train5 = regr5.predict(X_train5)
y_pred_test5 = regr5.predict(X_test5)
```

For postprocessing, we compute both the training and test errors of the models.

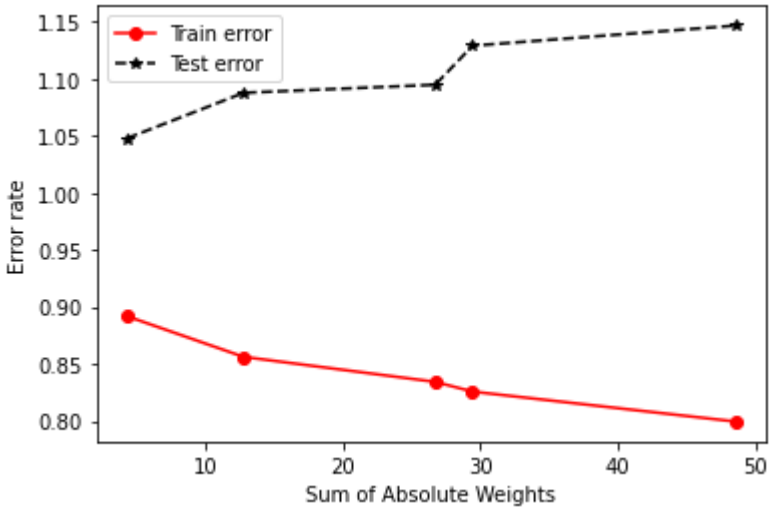
We can also show the resulting model and the sum of the absolute weights of the regression coefficients, i.e., $\sum_{j=0}^{d} \|w_j\|$, where d is the number of predictor attributes

```
import pandas as pd
import matplotlib.pyplot as plt
```

```
3
4  columns = ['Model', 'Train error', 'Test error', 'Sum of Absolute Weights']
5  model1 = "%.2f X + %.2f" % (regr.coef_[0][0], regr.intercept_[0])
6  values1 = [ model1, np.sqrt(mean_squared_error(y_train, y_pred_train)),
7   np.sqrt(mean_squared_error(y_test, y_pred_test)),
8   np.absolute(regr.coef_[0]).sum() + np.absolute(regr.intercept_[0])]
9
10 model2 = "%.2f X + %.2f X2 + %.2f" % (regr2.coef_[0][0], regr2.coef_[0][1],
   regr2.intercept_[0])
11 values2 = [ model2, np.sqrt(mean_squared_error(y_train, y_pred_train2)),
12  np.sqrt(mean_squared_error(y_test, y_pred_test2)),
13  np.absolute(regr2.coef_[0]).sum() + np.absolute(regr2.intercept_[0])]
14
15 model3 = "%.2f X + %.2f X2 + %.2f X3 + %.2f" % (regr3.coef_[0][0], regr3.coef_[0]
   [1],
16  regr3.coef_[0][2], regr3.intercept_[0])
17 values3 = [ model3, np.sqrt(mean_squared_error(y_train, y_pred_train3)),
18  np.sqrt(mean_squared_error(y_test, y_pred_test3)),
19  np.absolute(regr3.coef_[0]).sum() + np.absolute(regr3.intercept_[0])]
20
21 model4 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f" % (regr4.coef_[0][0],
   regr4.coef_[0][1],
22  regr4.coef_[0][2], regr4.coef_[0][3], regr4.intercept_[0
23 ])
24 values4 = [ model4, np.sqrt(mean_squared_error(y_train, y_pred_train4)),
25  np.sqrt(mean_squared_error(y_test, y_pred_test4)),
26  np.absolute(regr4.coef_[0]).sum() + np.absolute(regr4.intercept_[0])]
27
28 model5 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f X5 + %.2f" %
   (regr5.coef_[0][0],
29  regr5.coef_[0][1], regr5.coef_[0][2],
30  regr5.coef_[0][3], regr5.coef_[0][4], regr5.intercept_[0
31 ])
32 values5 = [ model5, np.sqrt(mean_squared_error(y_train, y_pred_train5)),
33  np.sqrt(mean_squared_error(y_test, y_pred_test5)),
34  np.absolute(regr5.coef_[0]).sum() + np.absolute(regr5.intercept_[0])]
35
36 results = pd.DataFrame([values1, values2, values3, values4, values5],
   columns=columns)
37
38 plt.plot(results['Sum of Absolute Weights'], results['Train error'], 'ro-')
39 plt.plot(results['Sum of Absolute Weights'], results['Test error'], 'k*--')
40 plt.legend(['Train error', 'Test error'])
41 plt.xlabel('Sum of Absolute Weights')
42 plt.ylabel('Error rate')
43
44 results
45
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Model	Train error	Test error	Sum of Absolute Weights
0	-3.24 X + 1.08	0.891873	1.047626	4.322954
1	-5.90 X + 5.92 X2 + 1.00	0.856157	1.087601	12.817040
2	-6.22 X + -2.30 X2 + 17.14 X3 + 1.08	0.834238	1.094661	26.744867
3	-7.16 X + 0.93 X2 + 8.39 X3 + 11.85 X4 + 1.12	0.825722	1.128861	29.453660
4	-7.16 X + 4.50 X2 + 3.52 X3 + -6.55 X4 + 25.68...	0.799399	1.146546	48.614927



The results above show that the first model,which fits y against X only, has the largest training error, but smallest test error,whereas the fifth model,which fits y against X and other correlated attributes, has the smallest training error but largest test error.

This is due to a phenomenon known as model overfitting,in which the low training error of the model does not reflect how well the model will perform no previously unseen test instances.

From the plot shown above,observe that the disparity between the training and test errors becomes wider as the sum of absolute weights of the model (which represents the model complexity) increases. Thus, one should control the complexity of the regression model to avoid the model overfitting problem.

5.4 Ridge Regression

- a variant of MLR designed to fit a linear model to the dataset by minimizing the following regularized least-square loss function:
  $L_{ridge}(y, f(X, w)) = \sum_{i=1}^{N} \|y_i - X_i w - w_0\|^2 + \alpha[\|w\|^2 + w_0^2]$
  where α is the hyperparameter for ridge regression.

Note that the ridge regression model reduces to MLR where $\alpha = 0$.

By increasing the value of α,we can control the complexity of the model as will be shown in the example below.

In the example shown below, we fit a ridge regression model to the previously created training set with correlated attributes.

We compare the results of the ridge regression model against those obtained using MLR.

```
from sklearn import linear_model

ridge = linear_model.Ridge(alpha = 0.4)
ridge.fit(X_train5,y_train)
y_pred_train_ridge = ridge.predict(X_train5)
y_pred_test_ridge = ridge.predict(X_test5)

model6 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f X5 + %.2f"%(ridge.coef_[0][0],
                                        ridge.coef_[0][1],ridge.coef_[0][2],
                                        ridge.coef_[0][3],ridge.coef_[0]
[4],ridge.intercept_[[0]])
values6 = [model6, np.sqrt(mean_squared_error(y_train,y_pred_train_ridge)),
          np.sqrt(mean_squared_error(y_test,y_pred_test_ridge)),
          np.absolute(ridge.coef_[0]).sum()+np.absolute(ridge.intercept_[0])]
ridge_results = pd.DataFrame([values6],columns=columns,index=['Ridge'])
pd.concat([results,ridge_results])
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Model	Train error	Test error	Sum of Absolute Weights
0	-3.24 X + 1.08	0.891873	1.047626	4.322954
1	-5.90 X + 5.92 X2 + 1.00	0.856157	1.087601	12.817040
2	-6.22 X + -2.30 X2 + 17.14 X3 + 1.08	0.834238	1.094661	26.744867
3	-7.16 X + 0.93 X2 + 8.39 X3 + 11.85 X4 + 1.12	0.825722	1.128861	29.453660
4	-7.16 X + 4.50 X2 + 3.52 X3 + -6.55 X4 + 25.68...	0.799399	1.146546	48.614927
Ridge	-2.24 X + -0.43 X2 + -0.14 X3 + -0.10 X4 + 0.0...	0.917456	1.052388	3.765759

By setting an appropriate value for the hyperparameter, α , we can control the sum of absolute weights, thus producing a test error that is quite comparable to that of MLR without the correlated attributes.

5.5 Lasso Regression

- One of the limitations of ridge regression is that, although it was able to reduce the regression coefficients associated with the correlated attributes and reduce the effect of model overfitting, the resulting model is still not sparse.
- Another variation of MLR, called lasso regression, is designed to produce sparser models by imposing an l_1 regularization on the regression coefficients as shown below:
  $L_{lasso}(y, f(X, w)) = \sum_{i=1}^{N} \|y_i - X_i w - w_0\|^2 + \alpha[\|w\|_1 + \|w_0\|]$

The example code below shows the results of applying lasso regression to the previously used correlated dataset.

```
from sklearn import linear_model

lasso = linear_model.Lasso(alpha=0.02)
lasso.fit(X_train5, y_train)
y_pred_train_lasso = lasso.predict(X_train5)
y_pred_test_lasso = lasso.predict(X_test5)

model7 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f X5 + %.2f" % (lasso.coef_[0],
 lasso.coef_[1], lasso.coef_[2],
 lasso.coef_[3], lasso.coef_[4], lasso.intercept_[0])

values7 = [ model7, np.sqrt(mean_squared_error(y_train, y_pred_train_lasso)),
 np.sqrt(mean_squared_error(y_test, y_pred_test_lasso)),
 np.absolute(lasso.coef_[0]).sum() + np.absolute(lasso.intercept_[0])]

lasso_results = pd.DataFrame([values7], columns=columns, index=['Lasso'])
pd.concat([results, ridge_results, lasso_results])
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Model	Train error	Test error	Sum of Absolute Weights
0	-3.24 X + 1.08	0.891873	1.047626	4.322954
1	-5.90 X + 5.92 X2 + 1.00	0.856157	1.087601	12.817040
2	-6.22 X + -2.30 X2 + 17.14 X3 + 1.08	0.834238	1.094661	26.744867
3	-7.16 X + 0.93 X2 + 8.39 X3 + 11.85 X4 + 1.12	0.825722	1.128861	29.453660
4	-7.16 X + 4.50 X2 + 3.52 X3 + -6.55 X4 + 25.68...	0.799399	1.146546	48.614927
Ridge	-2.24 X + -0.43 X2 + -0.14 X3 + -0.10 X4 + 0.0...	0.917456	1.052388	3.765759
Lasso	-2.90 X + 0.00 X2 + 0.00 X3 + 0.00 X4 + 0.00 X...	0.895692	1.043334	3.856242

Observe that the lasso regression model sets the coefficients for the correlated attributes, X2, X3, X4, and X5 to exactly zero unlike the ridge regression model. As a result, its test error is significantly better than that for ridge regression.

5.6 Hyperparameter Selection via Cross-Validation

- While both ridge and lasso regression methods can potentially alleviate the model overfitting problem, one of the challenges is how to select the appropriate hyperparameter value, α.
- In the examples shown below, we demonstrate examples of using a 5-fold cross-validation method to select the best hyperparameter of the model.

In the first sample code below, we vary the hyperparameter for ridge regression to a range between 0.2 and 1.0.

Using the RidgeCV() function, we can train a model with 5-fold cross-validation and select the best hyperparameter value.

```
from sklearn import linear_model

ridge = linear_model.RidgeCV(cv=5,alphas=[0.2, 0.4, 0.6, 0.8, 1.0])
ridge.fit(X_train5, y_train)
y_pred_train_ridge = ridge.predict(X_train5)
y_pred_test_ridge = ridge.predict(X_test5)

model6 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f X5 + %.2f" % (ridge.coef_[0][0],
 ridge.coef_[0][1], ridge.coef_[0][2],
```

```
 ridge.coef_[0][3], ridge.coef_[0][4], ridge.intercept_[0
])

values6 = [ model6, np.sqrt(mean_squared_error(y_train, y_pred_train_ridge)),
 np.sqrt(mean_squared_error(y_test, y_pred_test_ridge)),
 np.absolute(ridge.coef_[0]).sum() + np.absolute(ridge.intercept_[0])]
print("Selected alpha = %.2f" % ridge.alpha_)

ridge_results = pd.DataFrame([values6], columns=columns, index=['RidgeCV'])
pd.concat([results, ridge_results])
```

```
Selected alpha = 0.20
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Model	Train error	Test error	Sum of Absolute Weights
0	-3.24 X + 1.08	0.891873	1.047626	4.322954
1	-5.90 X + 5.92 X2 + 1.00	0.856157	1.087601	12.817040
2	-6.22 X + -2.30 X2 + 17.14 X3 + 1.08	0.834238	1.094661	26.744867
3	-7.16 X + 0.93 X2 + 8.39 X3 + 11.85 X4 + 1.12	0.825722	1.128861	29.453660
4	-7.16 X + 4.50 X2 + 3.52 X3 + -6.55 X4 + 25.68...	0.799399	1.146546	48.614927
RidgeCV	-2.74 X + -0.16 X2 + 0.09 X3 + 0.01 X4 + 0.21 ...	0.899190	1.044401	4.112120

In this next example, we illustrate how to apply cross-validation to select the best hyperparameter value for fitting a lasso regression model.

```
from sklearn import linear_model

lasso = linear_model.LassoCV(cv=5, alphas=[0.01, 0.02, 0.05, 0.1, 0.3, 0.5, 1.0])
lasso.fit(X_train5, y_train.reshape(y_train.shape[0]))
y_pred_train_lasso = lasso.predict(X_train5)
y_pred_test_lasso = lasso.predict(X_test5)

model7 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f X5 + %.2f" % (lasso.coef_[0],
 lasso.coef_[1], lasso.coef_[2],
 lasso.coef_[3], lasso.coef_[4], lasso.intercept_)
values7 = [ model7, np.sqrt(mean_squared_error(y_train, y_pred_train_lasso)),
```

```
 np.sqrt(mean_squared_error(y_test, y_pred_test_lasso)),
 np.absolute(lasso.coef_[0]).sum() + np.absolute(lasso.intercept_)]
print("Selected alpha = %.2f" % lasso.alpha_)

lasso_results = pd.DataFrame([values7], columns=columns, index=['LassoCV'])
pd.concat([results, ridge_results, lasso_results])
```

```
Selected alpha = 0.01
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Model	Train error	Test error	Sum of Absolute Weights
0	-3.24 X + 1.08	0.891873	1.047626	4.322954
1	-5.90 X + 5.92 X2 + 1.00	0.856157	1.087601	12.817040
2	-6.22 X + -2.30 X2 + 17.14 X3 + 1.08	0.834238	1.094661	26.744867
3	-7.16 X + 0.93 X2 + 8.39 X3 + 11.85 X4 + 1.12	0.825722	1.128861	29.453660
4	-7.16 X + 4.50 X2 + 3.52 X3 + -6.55 X4 + 25.68...	0.799399	1.146546	48.614927
RidgeCV	-2.74 X + -0.16 X2 + 0.09 X3 + 0.01 X4 + 0.21 ...	0.899190	1.044401	4.112120
LassoCV	-3.07 X + 0.00 X2 + 0.00 X3 + 0.00 X4 + 0.00 X...	0.892829	1.043911	4.089598

5.7 Summary

- Presents example Python code for fitting linear regression models to a dataset.
- Illustrate the problem of model overfitting and shows two alternative methods, called ridge and lasso regression, that can help alleviate such problem.
- While the model overfitting problem shown here is illustrated in the context of correlated attributes, the problem is more general and may arise due to other factors such as noise and other exception values in the data.