

▼ Module 5: Regression

The following tutorial contains Python examples for solving regression problems. You should refer to the Appendix chapter on regression of the "Introduction to Data Mining" book to understand some of the concepts introduced in this tutorial.

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.

Read the step-by-step instructions below carefully. To execute the code, click on the corresponding cell and press the SHIFT-ENTER keys simultaneously.

▼ Synthetic Data Generation

To illustrate how linear regression works, we first generate a random 1-dimensional vector of predictor variables, x , from a uniform distribution. The response variable y has a linear relationship with x according to the following equation: $y = -3x + 1 + \text{epsilon}$, where epsilon corresponds to random noise sampled from a Gaussian distribution with mean 0 and standard deviation of 1.

```
import pandas as pd
import io
import numpy as np
import os
from sklearn.model_selection import train_test_split

#Get file stored in my github
url = 'https://raw.githubusercontent.com/josephdonati/CSC-177---Project-2/master/Data/Admissi
data = pd.read_csv(url)
print("Data is received!!!")

print('Number of instances = %d' % (data.shape[0]))
print('Number of attributes = %d' % (data.shape[1]))
data.head()
```



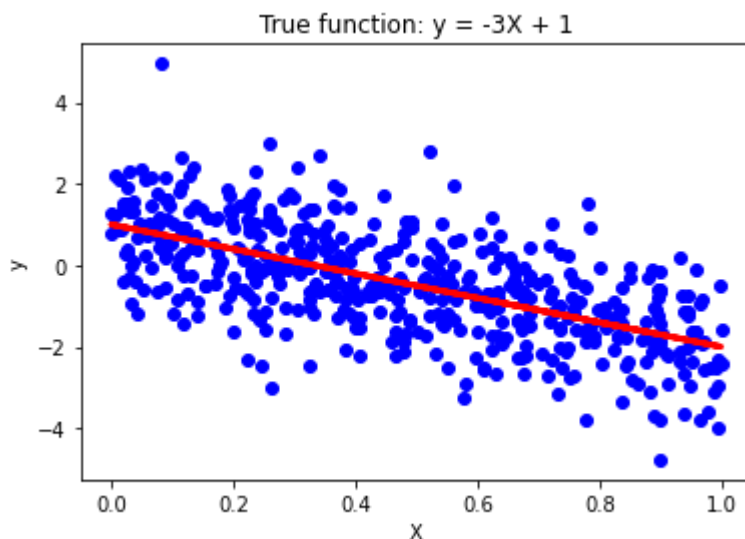
Data is received!!!
 Number of instances = 500
 Number of attributes = 9

| | Serial No. | GRE Score | TOEFL Score | University Rating | SOP | LOR | CGPA | Research | Chance of Admit |
|---|------------|-----------|-------------|-------------------|-----|-----|------|----------|-----------------|
| 0 | 1 | 337 | 118 | 4 | 4.5 | 4.5 | 9.65 | 1 | 0.92 |
| 1 | 2 | 324 | 107 | 4 | 4.0 | 4.5 | 8.87 | 1 | 0.76 |

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
url = 'https://raw.githubusercontent.com/josephdonati/CSC-177---Project-2/master/Data/Admissi
data = pd.read_csv(url)
pd.set_option('display.max_columns', None) #Display all columns for all tables in Notebook
numInstances = (data.shape[0])
print(numInstances)
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='blue')
plt.plot(X, y_true, color='red', linewidth=3)
plt.title('True function: y = -3X + 1')
plt.xlabel('X')
plt.ylabel('y')

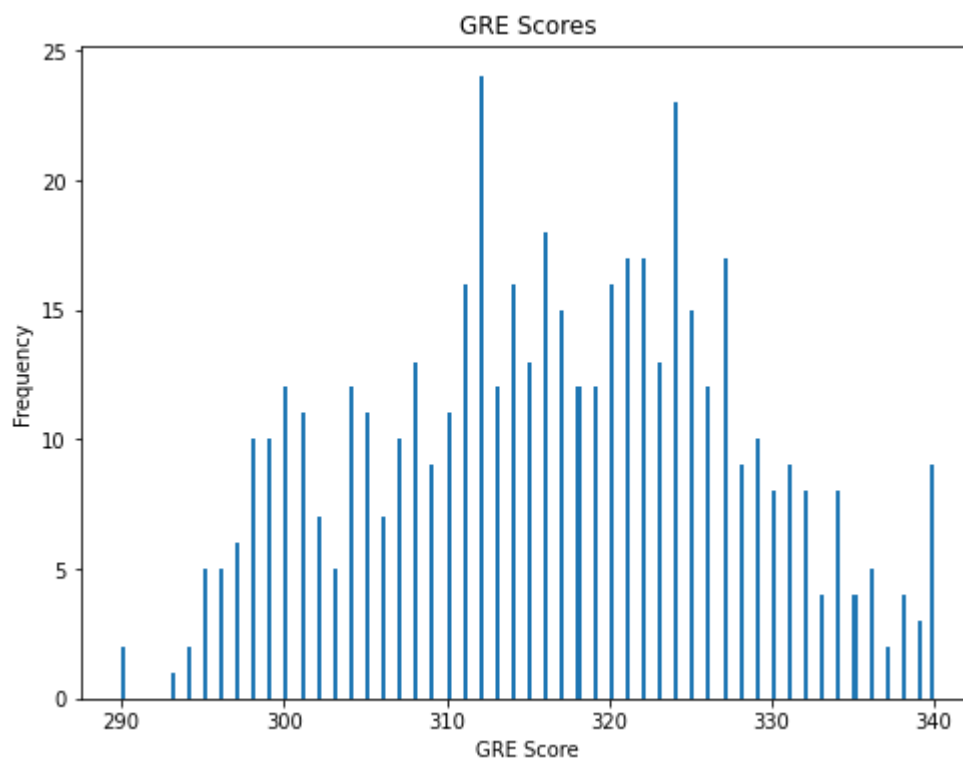
500
Text(0, 0.5, 'y')
```



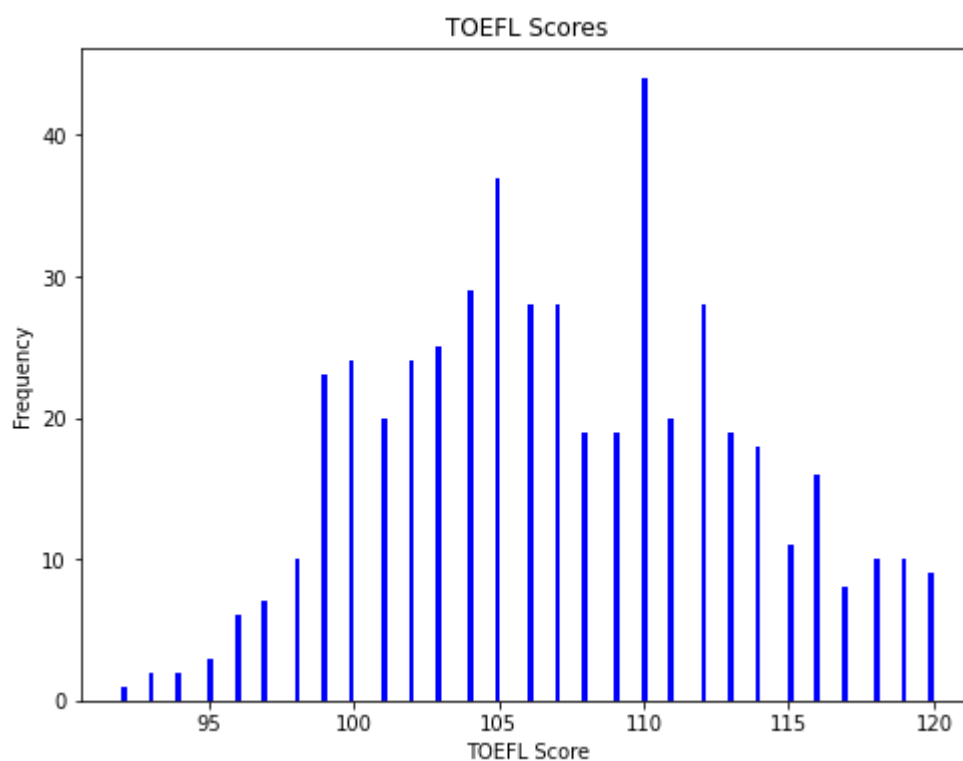
Analyzing each variable to Y - Chance of Admit

```
data["GRE Score"].plot(kind = 'hist',bins = 200,figsize = (8,6))
plt.title("GRE Scores")
plt.xlabel("GRE Score")
```

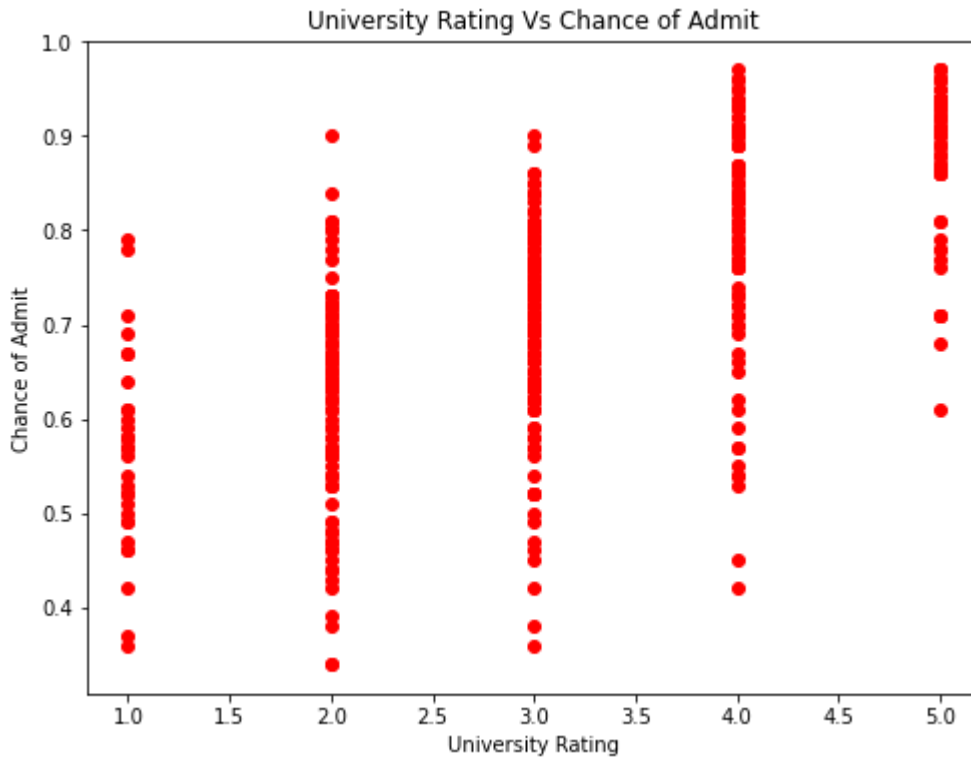
```
plt.ylabel("Frequency")  
plt.show()
```



```
data["TOEFL Score"].plot(kind = 'hist',bins = 150,figsize = (8,6), color='b')  
plt.title("TOEFL Scores")  
plt.xlabel("TOEFL Score")  
plt.ylabel("Frequency")  
plt.show()
```

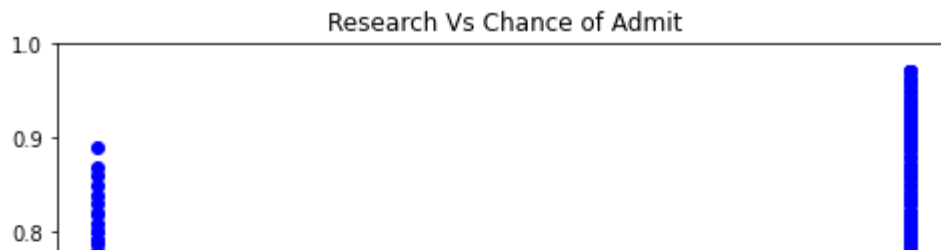


```
plt.figure(figsize=(8, 6))
plt.scatter(data["University Rating"],data["Chance of Admit "],color='r')
plt.title("University Rating Vs Chance of Admit")
plt.xlabel("University Rating")
plt.ylabel("Chance of Admit")
plt.show()
```

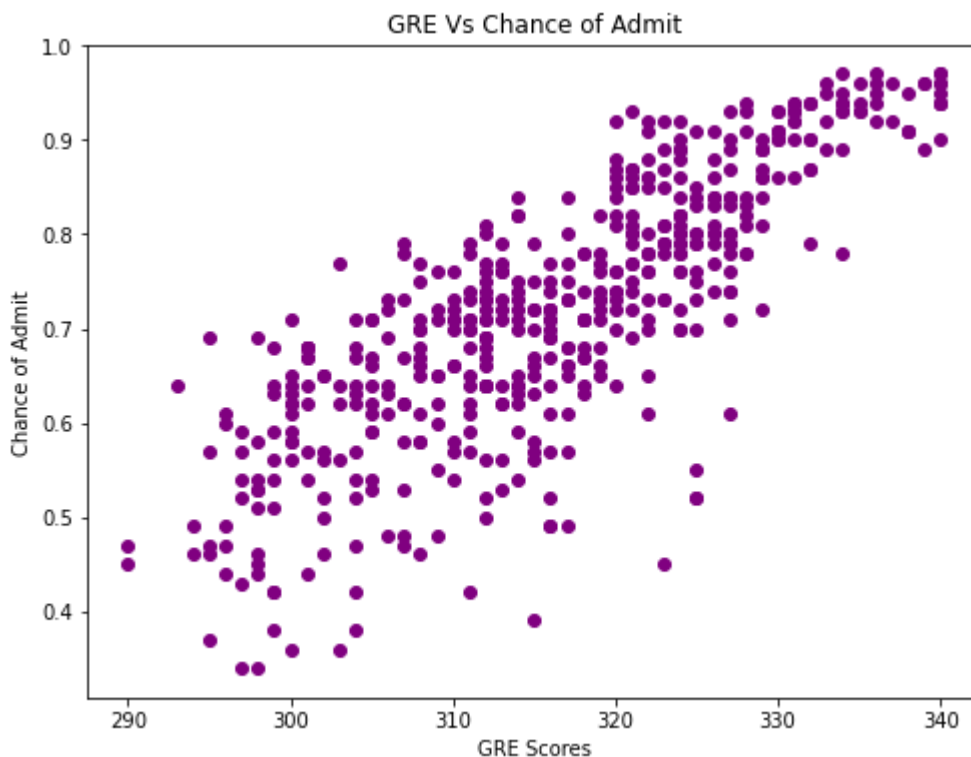


```
plt.figure(figsize=(8, 6))
plt.scatter(data["Research"],data["Chance of Admit "],color='b')
plt.title("Research Vs Chance of Admit")
plt.xlabel("Research")
plt.ylabel("Chance of Admit")
plt.show()
```





```
plt.figure(figsize=(8, 6))
plt.scatter(data["GRE Score"],data["Chance of Admit "],color='purple')
plt.title("GRE Vs Chance of Admit")
plt.xlabel("GRE Scores")
plt.ylabel("Chance of Admit")
plt.show()
```



Simple Linear Regression

```
X1 = data.drop(['Chance of Admit '], axis=1).values
y1 = data['Chance of Admit '].values
```

```
X_lm_train, X_lm_test, y_lm_train, y_lm_test = train_test_split(X1 , y1,test_size = 0.3, rand
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_lm_train = scaler.fit_transform(X_lm_train)
X_lm_test = scaler.fit_transform(X_lm_test)
```

```
from sklearn.linear_model import LinearRegression
model_slr = LinearRegression()
```

```
# Fir the Train data in Linear Model instance
model_slr.fit(X_lm_train, y_lm_train)
```

```
↳ LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
model_slr.coef_
```

```
↳ array([0.01299535, 0.02450052, 0.02363411, 0.00357782, 0.00416139,
        0.01225861, 0.06640091, 0.00948531])
```

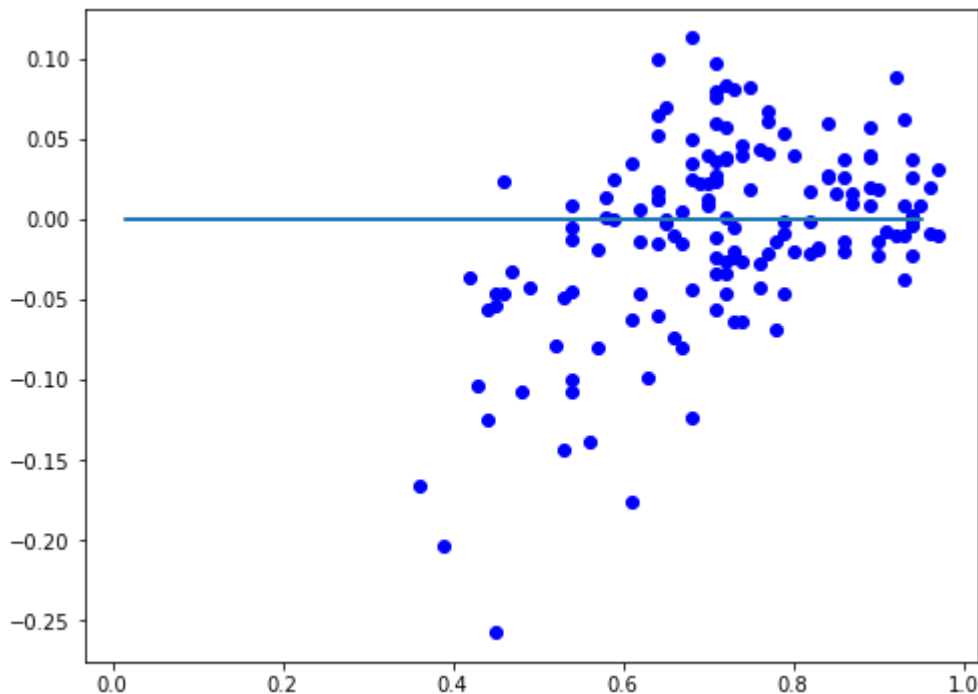
```
pred_y = model_slr.predict(X_lm_test)
```

```
from sklearn.metrics import mean_squared_error, classification_report, r2_score
print("Mean Squared error\t : ", mean_squared_error(y_lm_test, pred_y))
print("R_Square Score\t\t : ", r2_score(y_lm_test, pred_y))
```

```
↳ Mean Squared error      : 0.0035356482950009704
   R_Square Score          : 0.8303599587553676
```

```
plt.figure(figsize=(8,6))
plt.scatter(y_lm_test, y_lm_test-pred_y, color='b')
x = np.random.rand(30)
plt.plot(x, x*0 )
```

```
↳ [<matplotlib.lines.Line2D at 0x7f73bc6ee908>]
```



▼ Multiple Linear Regression

In this example, we illustrate how to 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_0 (intercept) are the regression coefficients.

Given the input dataset, the following steps are performed:

1. Split the input data into their respective training and test sets.
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

↳ 1 cell hidden

► Step 2: Fit Regression Model to Training Set

↳ 1 cell hidden

► Step 3: Apply Model to the Test Set

↳ 1 cell hidden

► Step 4: Evaluate Model Performance on Test Set

↳ 1 cell hidden

► Step 5: Postprocessing

↳ 1 cell hidden

▼ Effect of Correlated Attributes

In this example, we illustrate how the presence of correlated attributes can affect the performance of regression models. Specifically, we create 4 additional variables, X2, X3, X4, and X5 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. We then fit y against the predictor variables and compare their training and test set errors.

First, we 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='blue')
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)

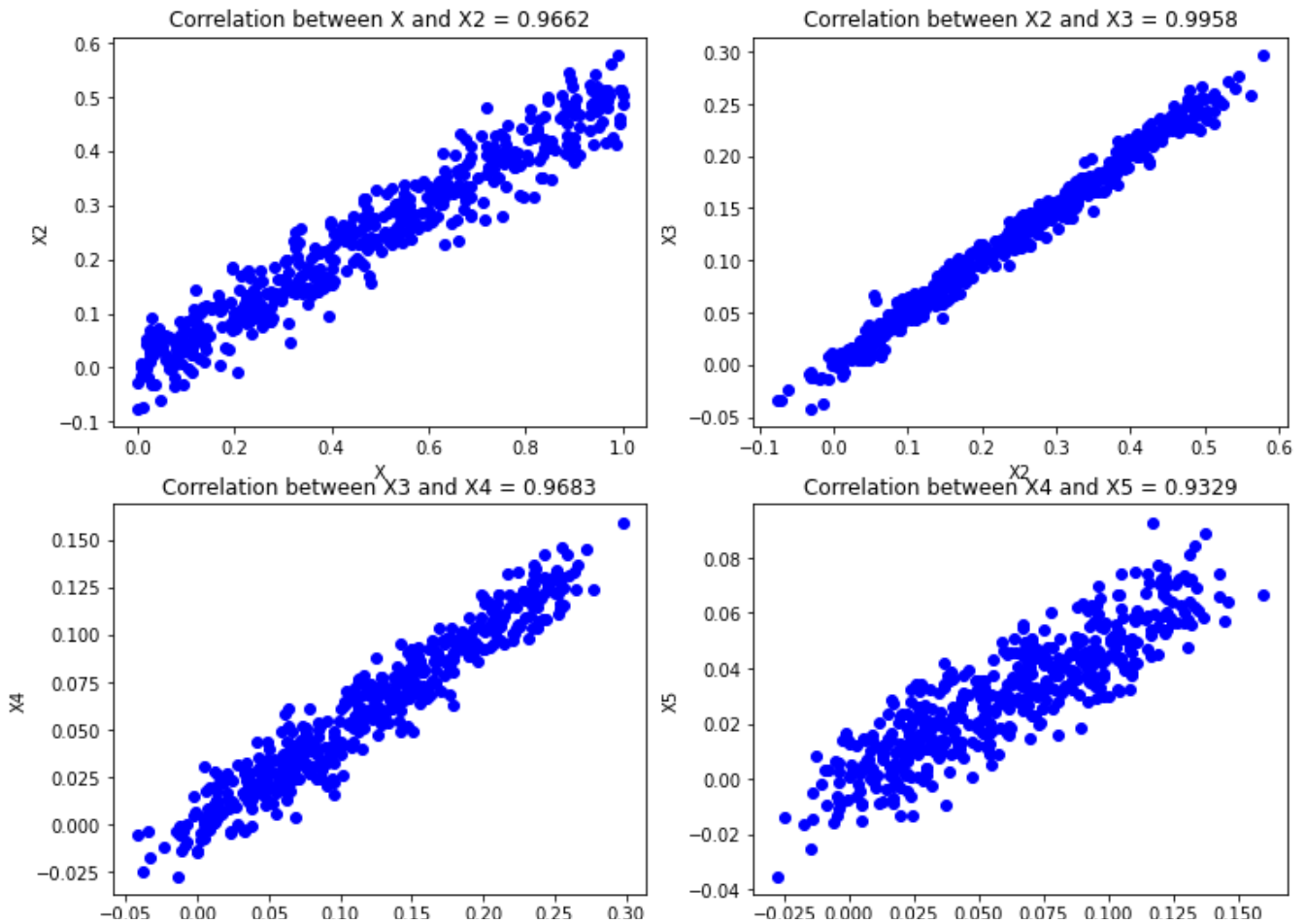
ax2.scatter(X2, X3, color='blue')
ax2.set_xlabel('X2')
ax2.set_ylabel('X3')
c = np.corrcoef(np.column_stack((X2[:-numTest],X3[:-numTest]))).T
titlestr = 'Correlation between X2 and X3 = %.4f' % (c[0,1])
ax2.set_title(titlestr)

ax3.scatter(X3, X4, color='blue')
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='blue')
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 = 0.9329')
```



Next, we 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 `X3`. 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:])))
```

Below, we train 4 new regression models based on the 4 versions of training and test data created in 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)

```



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

columns = ['Model', 'Train error', 'Test error', 'Sum of Absolute Weights']
model1 = "%.2f X + %.2f" % (regr.coef_[0][0], regr.intercept_[0])
values1 = [ model1, np.sqrt(mean_squared_error(y_train, y_pred_train)),
            np.sqrt(mean_squared_error(y_test, y_pred_test)),
            np.absolute(regr.coef_[0]).sum() + np.absolute(regr.intercept_[0])]

model2 = "%.2f X + %.2f X2 + %.2f" % (regr2.coef_[0][0], regr2.coef_[0][1], regr2.intercept_[0])
values2 = [ model2, np.sqrt(mean_squared_error(y_train, y_pred_train2)),
            np.sqrt(mean_squared_error(y_test, y_pred_test2)),
            np.absolute(regr2.coef_[0]).sum() + np.absolute(regr2.intercept_[0])]

model3 = "%.2f X + %.2f X2 + %.2f X3 + %.2f" % (regr3.coef_[0][0], regr3.coef_[0][1],
                                                regr3.coef_[0][2], regr3.intercept_[0])
values3 = [ model3, np.sqrt(mean_squared_error(y_train, y_pred_train3)),
            np.sqrt(mean_squared_error(y_test, y_pred_test3)),
            np.absolute(regr3.coef_[0]).sum() + np.absolute(regr3.intercept_[0])]

model4 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f" % (regr4.coef_[0][0], regr4.coef_[0][1],

```

```

regr4.coef_[0][2], regr4.coef_[0][3], regr4.intercept_
values4 = [ model4, np.sqrt(mean_squared_error(y_train, y_pred_train4)),
            np.sqrt(mean_squared_error(y_test, y_pred_test4)),
            np.absolute(regr4.coef_[0]).sum() + np.absolute(regr4.intercept_[0])]

model5 = "%.2f X + %.2f X2 + %.2f X3 + %.2f X4 + %.2f X5 + %.2f" % (regr5.coef_[0][0],
                                                                    regr5.coef_[0][1], regr5.coef_[0][2],
                                                                    regr5.coef_[0][3], regr5.coef_[0][4], regr5.intercept_
values5 = [ model5, np.sqrt(mean_squared_error(y_train, y_pred_train5)),
            np.sqrt(mean_squared_error(y_test, y_pred_test5)),
            np.absolute(regr5.coef_[0]).sum() + np.absolute(regr5.intercept_[0])]

results = pd.DataFrame([values1, values2, values3, values4, values5], columns=columns)

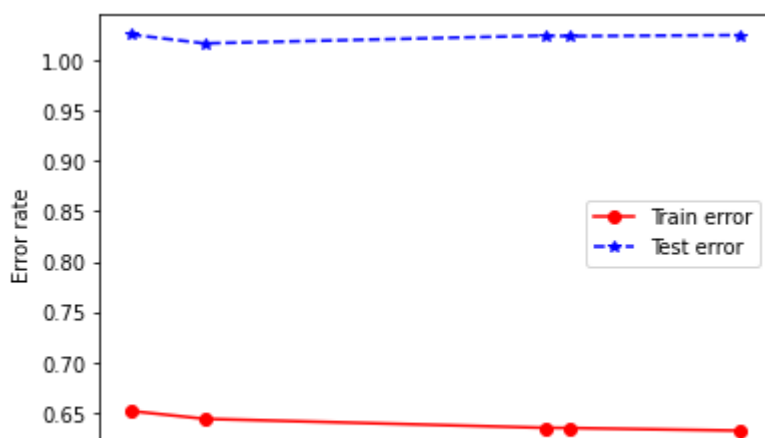
plt.plot(results['Sum of Absolute Weights'], results['Train error'], 'ro-')
plt.plot(results['Sum of Absolute Weights'], results['Test error'], 'b*--')
plt.legend(['Train error', 'Test error'])
plt.xlabel('Sum of Absolute Weights')
plt.ylabel('Error rate')

```

results



| | Model | Train error | Test error | Sum of Absolute Weights |
|---|---|-------------|------------|-------------------------|
| 0 | -2.95 X + 0.58 | 0.651375 | 1.025975 | 3.526421 |
| 1 | -4.03 X + 2.38 X2 + 0.54 | 0.643577 | 1.017197 | 6.954598 |
| 2 | -4.03 X + -4.47 X2 + 13.66 X3 + 0.56 | 0.634641 | 1.024930 | 22.715747 |
| 3 | -3.99 X + -4.69 X2 + 13.15 X3 + 1.48 X4 + 0.57 | 0.634435 | 1.024380 | 23.872902 |
| 4 | -4.12 X + -4.47 X2 + 13.82 X3 + -2.49 X4 + 6.3... | 0.631819 | 1.025395 | 31.760121 |



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 on 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.

▼ Ridge Regression

Ridge regression is a variant of MLR designed to fit a linear model to the dataset by minimizing the following regularized least-square loss function:

$$L_{\text{ridge}}(y, f(X, w)) = \sum_{i=1}^N \|y_i - X_i w - w_0\|^2 + \alpha \left[\|w\|^2 + w_0^2 \right],$$

where α is the hyperparameter for ridge regression. Note that the ridge regression model reduces to MLR when $\alpha = 0$. 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_
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])
```



| | Model | Train error | Test error | Sum of Absolute Weights |
|---|--------------------------|-------------|------------|-------------------------|
| 0 | -2.95 X + 0.58 | 0.651375 | 1.025975 | 3.526421 |
| 1 | -4.03 X + 2.38 X2 + 0.54 | 0.643577 | 1.017197 | 6.954598 |

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.

4 + 6.3 0.651819 1.025595 51.700121

▼ 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:

$$L_{\text{lasso}}(y, f(X, w)) = \sum_{i=1}^N \|y_i - X_i w - w_0\|^2 + \alpha \left[\|w\|_1 + |w_0| \right]$$

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])
```



| | Model | Train error | Test error | Sum of Absolute Weights |
|---|--|-------------|------------|-------------------------|
| 0 | -2.95 X + 0.58 | 0.651375 | 1.025975 | 3.526421 |
| 1 | -4.03 X + 2.38 X ₂ + 0.54 | 0.643577 | 1.017197 | 6.954598 |
| 2 | -4.03 X + -4.47 X ₂ + 13.66 X ₃ + 0.56 | 0.634641 | 1.024930 | 22.715747 |
| 3 | -3.99 X + -4.69 X ₂ + 13.15 X ₃ + 1.48 X ₄ + _ _ _ | 0.634435 | 1.024380 | 23.872902 |

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.

▼ Regression in Real Dataset

▼ Useful functions

```
#Function to normalize columns
def normalize_numeric_minmax(df, name):
    df[name] = ((df[name] - df[name].min()) / (df[name].max() - df[name].min())).astype(n

# Convert a Pandas dataframe to the x,y inputs that TensorFlow needs
import collections
def to_xy(df, target):
    result = []
    for x in df.columns:
        if x != target:
            result.append(x)
    # find out the type of the target column.
    target_type = df[target].dtypes
    target_type = target_type[0] if isinstance(target_type, collections.Sequence) else target
    # Encode to int for classification, float otherwise. TensorFlow likes 32 bits.
    if target_type in (np.int64, np.int32):
        # Classification
        dummies = pd.get_dummies(df[target])
        return df[result].values.astype(np.float32), dummies.values.astype(np.float32)
    else:
        # Regression
        return df[result].values.astype(np.float32), df[target].values.astype(np.float32)

# Regression chart.
def chart_regression(pred,y,sort=True):
    t = pd.DataFrame({'pred' : pred, 'y' : y.flatten()})
```

```

1+ sort:
    t.sort_values(by=['y'],inplace=True)
a = plt.plot(t['y'].tolist(),label='expected')
b = plt.plot(t['pred'].tolist(),label='prediction')
plt.ylabel('output')
plt.legend()
plt.show()


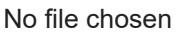
```

▼ Data Pre-Processing

```

from google.colab import files
uploaded = files.upload()

```




 Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

Saving Admission Predict .csv.csv to Admission Predict .csv.csv

```

admission_df = pd.read_csv(io.StringIO(uploaded['Admission_Predict_.csv.csv'].decode('utf-8')))
admission_df[0:5]

```




| | Serial No. | GRE Score | TOEFL Score | University Rating | SOP | LOR | CGPA | Research | Chance of Admit |
|---|------------|-----------|-------------|-------------------|-----|-----|------|----------|-----------------|
| 0 | 1 | 337 | 118 | 4 | 4.5 | 4.5 | 9.65 | 1 | 0.92 |
| 1 | 2 | 324 | 107 | 4 | 4.0 | 4.5 | 8.87 | 1 | 0.76 |
| 2 | 3 | 316 | 104 | 3 | 3.0 | 3.5 | 8.00 | 1 | 0.72 |
| 3 | 4 | 322 | 110 | 3 | 3.5 | 2.5 | 8.67 | 1 | 0.80 |
| 4 | 5 | 314 | 103 | 2 | 2.0 | 3.0 | 8.21 | 0 | 0.65 |

```

admission_df = admission_df.drop(['Serial No.', 'Chance of Admit '], axis = 1)
admission_df[0:5]

```



| | GRE Score | TOEFL Score | University Rating | SOP | LOR | CGPA | Research |
|---|-----------|-------------|-------------------|-----|-----|------|----------|
| 0 | 337 | 118 | 4 | 4.5 | 4.5 | 9.65 | 1 |
| 1 | 324 | 107 | 4 | 4.0 | 4.5 | 8.87 | 1 |
| 2 | 316 | 104 | 3 | 3.0 | 3.5 | 8.00 | 1 |
| 3 | 322 | 110 | 3 | 3.5 | 2.5 | 8.67 | 1 |
| 4 | 314 | 103 | 2 | 2.0 | 3.0 | 8.21 | 0 |

```
# Normalize the input columns
```

```
normalize_numeric_minmax(admission_df,"GRE Score")
```

```

normalize_numeric_minmax(admission_df,"TOEFL Score")
normalize_numeric_minmax(admission_df,"University Rating")
normalize_numeric_minmax(admission_df,"SOP")
stock_df[0:5]

```

| | GRE Score | TOEFL Score | University Rating | SOP | LOR | CGPA | Research |
|---|-----------|-------------|-------------------|-----|-----|------|----------|
| 0 | 337 | 118 | 4 | 4.5 | 4.5 | 9.65 | 1 |
| 1 | 324 | 107 | 4 | 4.0 | 4.5 | 8.87 | 1 |
| 2 | 316 | 104 | 3 | 3.0 | 3.5 | 8.00 | 1 |
| 3 | 322 | 110 | 3 | 3.5 | 2.5 | 8.67 | 1 |
| 4 | 314 | 103 | 2 | 2.0 | 3.0 | 8.21 | 0 |

```

# to xy to convert pandas to tensor flow
x,y=to_xy(admission_df,"TOEFL Score")

```

```

#Split for train and test
from sklearn.model_selection import train_test_split

```

```

x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.3, random_state=42)

```

```

x_train.shape

```

```

(350, 6)

```

```

x_test.shape

```

```

(150, 6)

```

```

y_train.shape

```

```

(350,)

```

```

y_test.shape

```

```

(150,)

```

▼ Train Model - SKLearn

```

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

```

```

# Create linear regression object

```



```
adm_regr = linear_model.LinearRegression()

# Fit regression model to the training set
adm_regr.fit(x_train, y_train)

↳ LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

▼ Prediction and Model Evaluation

```
y_pred = adm_regr.predict(x_test)

for i in range(10):
    print("True price : ",y_test[i],"Prediction : ", y_pred[i])

↳ True price : 0.85714287 Prediction : 0.8341271
   True price : 0.5714286 Prediction : 0.5927926
   True price : 0.4642857 Prediction : 0.40457386
   True price : 0.60714287 Prediction : 0.5085656
   True price : 0.71428573 Prediction : 0.6837569
   True price : 0.6785714 Prediction : 0.7628655
   True price : 0.2857143 Prediction : 0.121112525
   True price : 0.5 Prediction : 0.3771537
   True price : 0.60714287 Prediction : 0.6150289
   True price : 0.71428573 Prediction : 0.5993108

print(adm_regr.coef_, adm_regr.intercept_)

↳ [ 0.5055065  0.05055182  0.09207092 -0.00432891  0.09695275 -0.01276234] -0.61470133

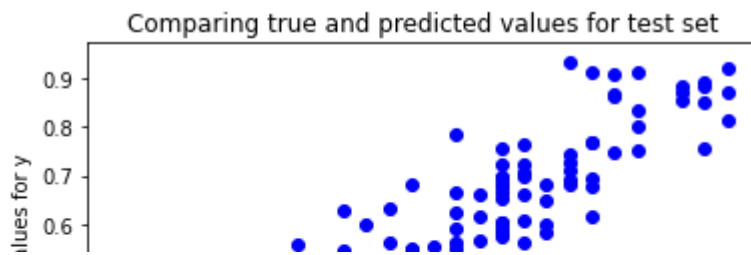
# Comparing true versus predicted values
plt.scatter(y_test, y_pred, color='blue')
plt.title('Comparing true and predicted values for test set')
plt.xlabel('True values for y')
plt.ylabel('Predicted values for y')

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

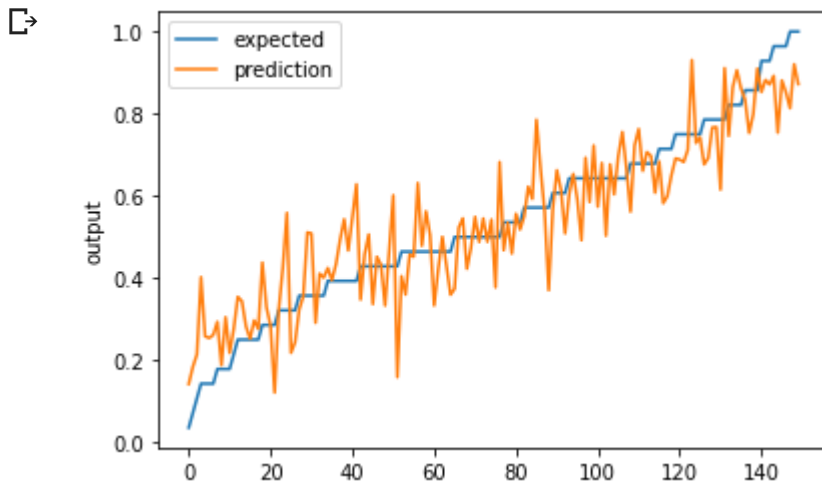
↳
```

Root mean squared error = 0.0957

R-squared = 0.8088



chart_regression(y_pred,y_test)



▼ Train Model - Neural Network

Imports

```
from keras import Sequential
from keras.layers.core import Dense
```

↳ Using TensorFlow backend.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
tf.__version__
tf.test.gpu_device_name()
```

↳ '/device:GPU:0'

```
from tensorflow.python.keras.layers import Dense
from tensorflow.python.keras import Sequential
```

```
# Build network
model_relu = Sequential()
model_relu.add(Dense(60, input_dim=x_train.shape[1], activation='relu'))
model_relu.add(Dense(30, activation='relu')) # Hidden 2
model_relu.add(Dense(1)) # Output
model_relu.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
model_relu.fit(x_train,y_train,verbose=1,epochs=100)
```



```
Epoch 1/100
11/11 [=====] - 0s 3ms/step - loss: 0.0408 - accuracy: 0.0200
Epoch 2/100
11/11 [=====] - 0s 3ms/step - loss: 0.0371 - accuracy: 0.0200
Epoch 3/100
11/11 [=====] - 0s 3ms/step - loss: 0.0300 - accuracy: 0.0200
Epoch 4/100
11/11 [=====] - 0s 2ms/step - loss: 0.0258 - accuracy: 0.0171
Epoch 5/100
11/11 [=====] - 0s 3ms/step - loss: 0.0253 - accuracy: 0.0200
Epoch 6/100
11/11 [=====] - 0s 3ms/step - loss: 0.0228 - accuracy: 0.0200
Epoch 7/100
11/11 [=====] - 0s 2ms/step - loss: 0.0216 - accuracy: 0.0200
Epoch 8/100
11/11 [=====] - 0s 3ms/step - loss: 0.0199 - accuracy: 0.0200
Epoch 9/100
11/11 [=====] - 0s 2ms/step - loss: 0.0190 - accuracy: 0.0200
Epoch 10/100
11/11 [=====] - 0s 3ms/step - loss: 0.0188 - accuracy: 0.0200
Epoch 11/100
11/11 [=====] - 0s 3ms/step - loss: 0.0185 - accuracy: 0.0200
Epoch 12/100
11/11 [=====] - 0s 3ms/step - loss: 0.0165 - accuracy: 0.0200
Epoch 13/100
11/11 [=====] - 0s 2ms/step - loss: 0.0156 - accuracy: 0.0200
Epoch 14/100
11/11 [=====] - 0s 3ms/step - loss: 0.0154 - accuracy: 0.0200
Epoch 15/100
11/11 [=====] - 0s 3ms/step - loss: 0.0152 - accuracy: 0.0200
Epoch 16/100
11/11 [=====] - 0s 3ms/step - loss: 0.0152 - accuracy: 0.0200
Epoch 17/100
11/11 [=====] - 0s 3ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 18/100
11/11 [=====] - 0s 3ms/step - loss: 0.0146 - accuracy: 0.0200
Epoch 19/100
11/11 [=====] - 0s 3ms/step - loss: 0.0157 - accuracy: 0.0200
Epoch 20/100
11/11 [=====] - 0s 3ms/step - loss: 0.0154 - accuracy: 0.0200
Epoch 21/100
11/11 [=====] - 0s 3ms/step - loss: 0.0148 - accuracy: 0.0200
Epoch 22/100
11/11 [=====] - 0s 2ms/step - loss: 0.0159 - accuracy: 0.0200
Epoch 23/100
11/11 [=====] - 0s 4ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 24/100
11/11 [=====] - 0s 3ms/step - loss: 0.0141 - accuracy: 0.0200
Epoch 25/100
11/11 [=====] - 0s 3ms/step - loss: 0.0144 - accuracy: 0.0200
Epoch 26/100
11/11 [=====] - 0s 3ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 27/100
11/11 [=====] - 0s 3ms/step - loss: 0.0140 - accuracy: 0.0200
Epoch 28/100
11/11 [=====] - 0s 3ms/step - loss: 0.0143 - accuracy: 0.0200
Epoch 29/100
```

```
11/11 [=====] - 0s 3ms/step - loss: 0.0139 - accuracy: 0.0200
Epoch 30/100
11/11 [=====] - 0s 3ms/step - loss: 0.0138 - accuracy: 0.0200
Epoch 31/100
11/11 [=====] - 0s 3ms/step - loss: 0.0139 - accuracy: 0.0200
Epoch 32/100
11/11 [=====] - 0s 3ms/step - loss: 0.0143 - accuracy: 0.0200
Epoch 33/100
11/11 [=====] - 0s 3ms/step - loss: 0.0141 - accuracy: 0.0200
Epoch 34/100
11/11 [=====] - 0s 3ms/step - loss: 0.0144 - accuracy: 0.0200
Epoch 35/100
11/11 [=====] - 0s 3ms/step - loss: 0.0146 - accuracy: 0.0200
Epoch 36/100
11/11 [=====] - 0s 3ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 37/100
11/11 [=====] - 0s 3ms/step - loss: 0.0136 - accuracy: 0.0200
Epoch 38/100
11/11 [=====] - 0s 3ms/step - loss: 0.0139 - accuracy: 0.0200
Epoch 39/100
11/11 [=====] - 0s 3ms/step - loss: 0.0138 - accuracy: 0.0200
Epoch 40/100
11/11 [=====] - 0s 3ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 41/100
11/11 [=====] - 0s 3ms/step - loss: 0.0149 - accuracy: 0.0200
Epoch 42/100
11/11 [=====] - 0s 3ms/step - loss: 0.0152 - accuracy: 0.0200
Epoch 43/100
11/11 [=====] - 0s 3ms/step - loss: 0.0147 - accuracy: 0.0200
Epoch 44/100
11/11 [=====] - 0s 3ms/step - loss: 0.0138 - accuracy: 0.0200
Epoch 45/100
11/11 [=====] - 0s 3ms/step - loss: 0.0148 - accuracy: 0.0200
Epoch 46/100
11/11 [=====] - 0s 3ms/step - loss: 0.0154 - accuracy: 0.0200
Epoch 47/100
11/11 [=====] - 0s 3ms/step - loss: 0.0137 - accuracy: 0.0200
Epoch 48/100
11/11 [=====] - 0s 3ms/step - loss: 0.0143 - accuracy: 0.0200
Epoch 49/100
11/11 [=====] - 0s 3ms/step - loss: 0.0144 - accuracy: 0.0200
Epoch 50/100
11/11 [=====] - 0s 3ms/step - loss: 0.0135 - accuracy: 0.0200
Epoch 51/100
11/11 [=====] - 0s 3ms/step - loss: 0.0136 - accuracy: 0.0200
Epoch 52/100
11/11 [=====] - 0s 3ms/step - loss: 0.0140 - accuracy: 0.0200
Epoch 53/100
11/11 [=====] - 0s 3ms/step - loss: 0.0136 - accuracy: 0.0200
Epoch 54/100
11/11 [=====] - 0s 3ms/step - loss: 0.0136 - accuracy: 0.0200
Epoch 55/100
11/11 [=====] - 0s 3ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 56/100
11/11 [=====] - 0s 2ms/step - loss: 0.0145 - accuracy: 0.0200
Epoch 57/100
11/11 [=====] - 0s 2ms/step - loss: 0.0136 - accuracy: 0.0200
Epoch 58/100
```