Table of Contents

[Reshaping Data 1](#_Toc61284737)

[Ridge Regression 3](#_Toc61284738)

[Ridge Classifier 7](#_Toc61284739)

[Saving and Re-Using Trained Models 11](#_Toc61284740)

## Reshaping Data

Many algorithms will require a transformation to adjust how X and y values are arranged. The need to transform, scale and then revert back to original formats is magnified when working with machine learning algorithms. The need to switch data formats can be frustrating but the problem will become less troubling with practice and experience. A common error to watch for is:

**Reshape your data either using array.reshape(-1, 1)**

Example : Resolving the Reshape Error

This example demonstrates the reshape error. Initially the data is presented as:

642 5

679 5

473 5

390 8

1096 6

When trying to scale the data, the following error displays:

**Reshape your data either using array.reshape(-1, 1)**

The code needs to be transformed into a set of vertical arrays.

[[5]

[5]

[5]

...

[6]

[6]

[5]]

Here is the code which leads to the error while scaling:

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.metrics import mean\_squared\_error  from sklearn.model\_selection import train\_test\_split  import statsmodels.api as sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "winequality.csv"  dataset = pd.read\_csv(PATH + CSV\_DATA)  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Include only statistically significant columns.  X = dataset[['volatile acidity', 'chlorides', 'total sulfur dioxide',  'pH', 'sulphates','alcohol']]  y = dataset['quality']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)  # Stochastic gradient descent models are sensitive to differences  # in scale so a MinMax is usually used.  from sklearn.preprocessing import MinMaxScaler  scalerX = MinMaxScaler()  scalerX.fit(X\_train)  print(y\_train)  # Build scaler for y.  scalerY = MinMaxScaler()  scalerY.fit(y\_train) |

To fix the error we use:

y\_train = np.array(y\_train).reshape(-1, 1)

To transform the y values to the following format of vertical arrays:

Modified y\_train

[[5]

[5]

[5]

...

[6]

[6]

[5]]

To build this example, replace the code that performs the scaling in Example 1 with the following:

|  |
| --- |
| import numpy as np  # Create array of y-values and reshape.  y\_train = np.array(y\_train).reshape(-1, 1)  print("\nModified y\_train")  print(y\_train)  # Build scaler for y.  scalerY = MinMaxScaler()  scalerY.fit(y\_train)  y\_scaled = scalerY.transform(y\_train)  print("\nScaled y")  print(y\_scaled) |

## Ridge Regression

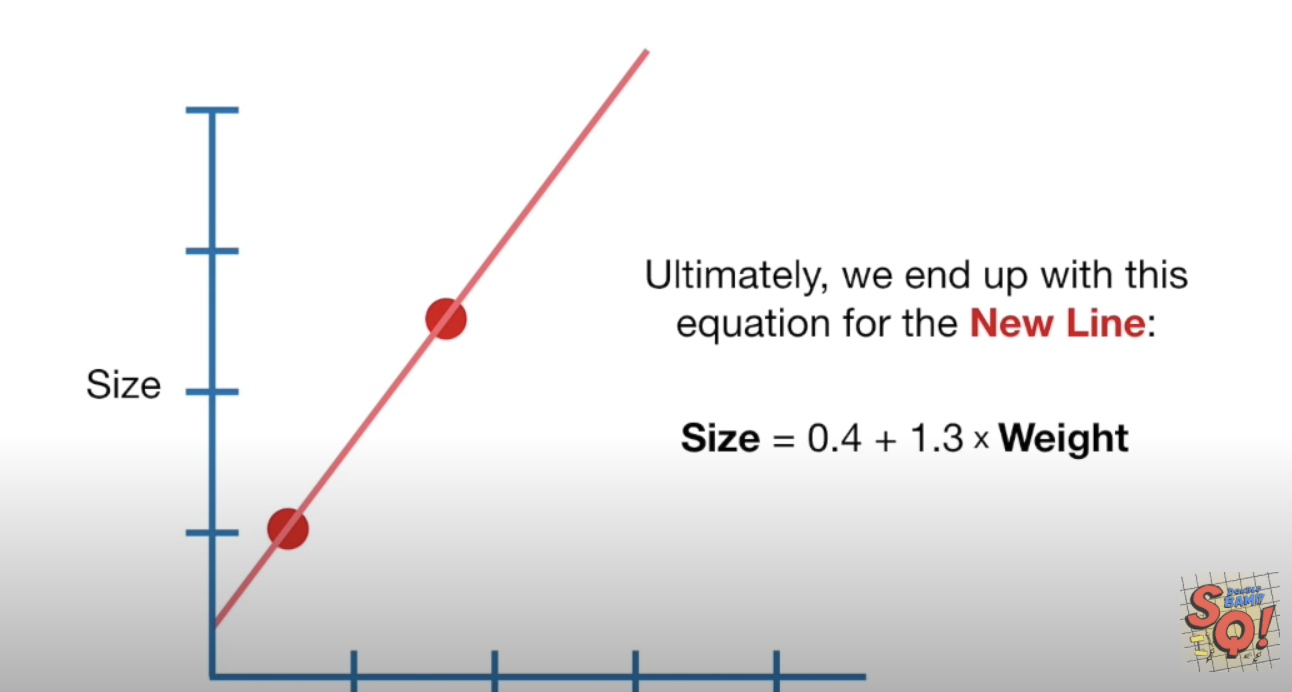
Ridge regression is an algorithm that builds on gradient descent to find an optimum regression. Ridge regression adds a penalty measure to enable a more flexible best-fit line. This video by Josh Stormer explains nicely how the ridge regression works.

<https://www.youtube.com/watch?v=Q81RR3yKn30>

Example : Ridge Regression Introduction

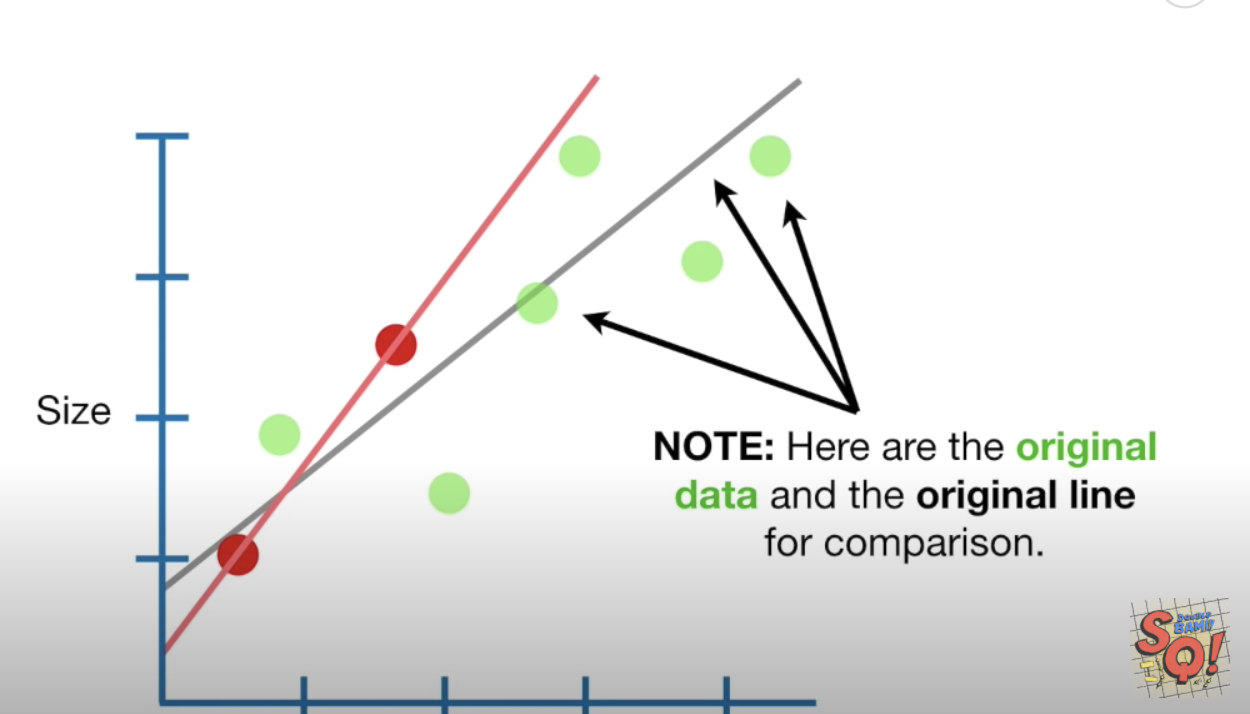
This example will summarize the highlights from the video. We can use a best-fit line for training data with an algorithm like ordinary least squares like the one in Figure 1.

Figure : Best Fit Line with our Sample Data



However, if our training set is in red and the test set is in green (refer to Figure 2) then the least squares fit actually is actually quite poor.

Figure : Poor fitting regression line

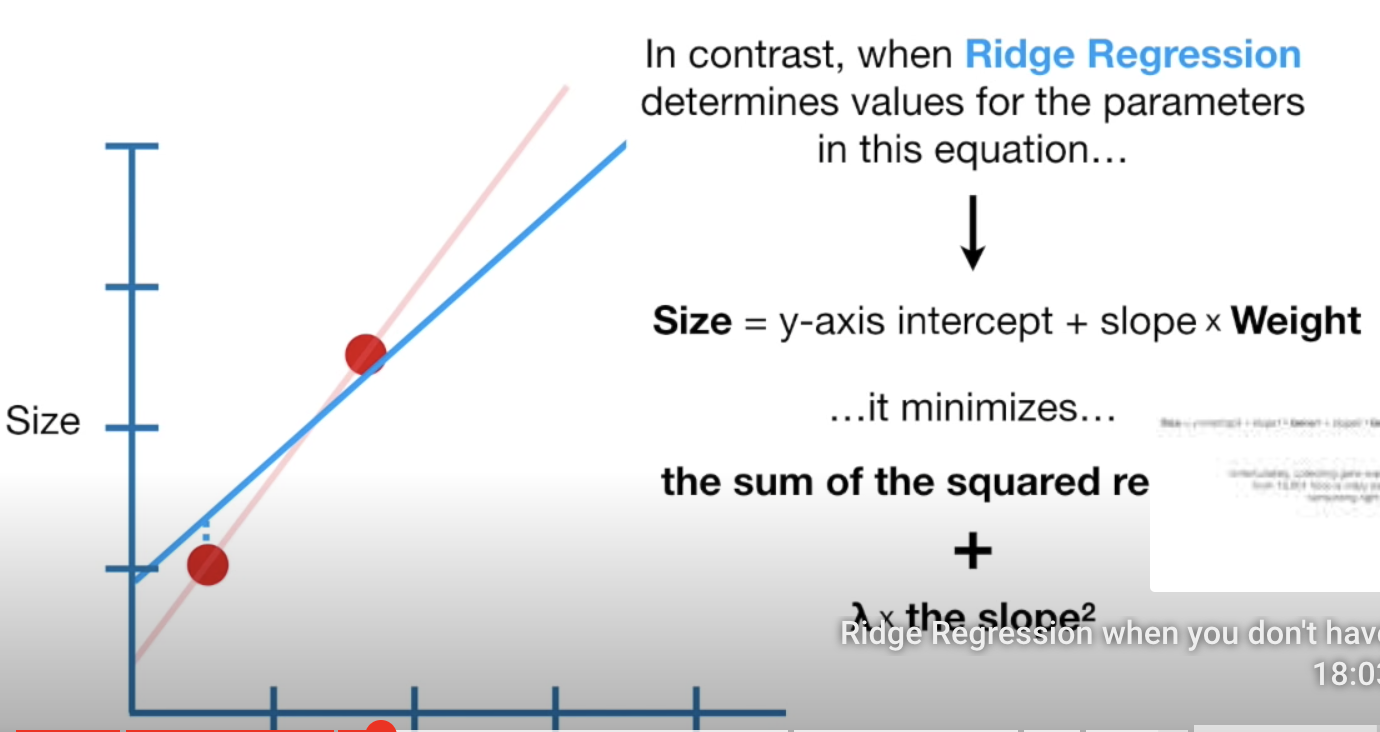
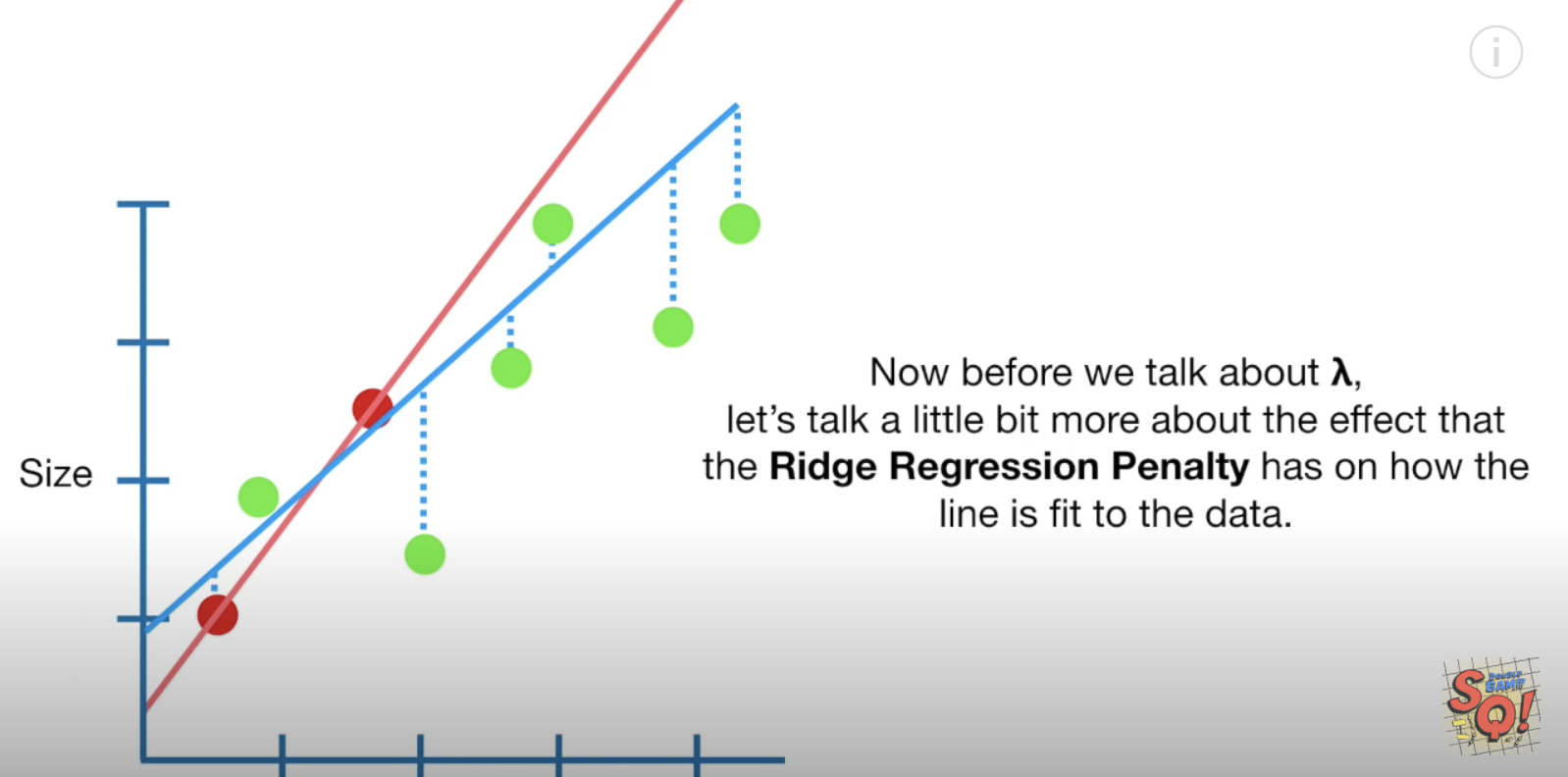


Ridge regression uses a penalty measure to avoid this problem of over-fitting on the training data. Ridge regression combines the least squares algorithm with a penalty that equals . determines how severe the penalty is. The equation for ridge regression becomes:

y-axis-intercept + slopeweight +

Notice in Figure 3 how the slope is slightly reduced with the penalty. The best-fit line does not fit the training data as well but it does have a better fit with the test data. Cross fold validation is used to find the optimum penalty.

Figure : Effect on fit with penalty

Example : Ridge Regression

This example shows how to use ridge regression to predict wine quality. If you compare the predictions of wine quality for OLS, Ridge and Stochastic Gradient Descent regression the results are not very dramatic in this case.

SGD (Calculated in the lab)

Root Mean Squared Error: 0.6223729015084277

OLS Regression

Root Mean Squared Error: 0.6206983401233142

Ridge Regression

Root Mean Squared Error: 0.6199800116556096

The code for the ridge regression is highlighted in yellow.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.metrics import mean\_squared\_error  from sklearn.model\_selection import train\_test\_split  import statsmodels.api as sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "winequality.csv"  dataset = pd.read\_csv(PATH + CSV\_DATA,  skiprows=1, # Don't include header row as part of data.  encoding = "ISO-8859-1", sep=',',  names=('fixed acidity', 'volatile acidity', 'citric acid',  'residual sugar', 'chlorides', 'free sulfur dioxide',  'total sulfur dioxide', 'density', 'pH', 'sulphates',  'alcohol', 'quality'))  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  print(dataset.head())  print(dataset.describe())  # Include only statistically significant columns.  X = dataset[['volatile acidity',  'chlorides', 'total sulfur dioxide',  'pH', 'sulphates','alcohol']]  y = dataset['quality']  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)  # Stochastic gradient descent models are sensitive to differences  # in scale so a MinMax is usually used.  from sklearn.preprocessing import MinMaxScaler  scalerX = MinMaxScaler()  scalerX.fit(X\_train)  # Build scaler for y.  scalerY = MinMaxScaler()  reshapedYtrain = np.array(y\_train).reshape(-1,1)  scalerY.fit(reshapedYtrain)  # Scale X\_train, X\_test and y\_train.  X\_trainScaled = scalerX.transform(X\_train)  X\_testScaled = scalerX.transform(X\_test)  y\_trainScaled = scalerY.transform(reshapedYtrain)  # Add constant to scaled data.  X\_trainScaled = sm.add\_constant(X\_trainScaled)  X\_testScaled = sm.add\_constant(X\_testScaled)  #---------------------------------------------------------------  # Perform OLS regression.  model = sm.OLS(y\_trainScaled, X\_trainScaled).fit()  predictions = model.predict(X\_testScaled) # make the predictions by the model  print(model.summary())  # Convert predictions to unscaled predicitons and compare with y\_test.  unscaledPredictionsOLS = scalerY.inverse\_transform(predictions.reshape(-1,1))  print('Root Mean Squared Error:',  np.sqrt(mean\_squared\_error(y\_test, unscaledPredictionsOLS)))  #---------------------------------------------------------------  # Perform Ridge regression.  print("\nRidge Regression")  from sklearn.linear\_model import Ridge  ridge\_reg = Ridge(solver='auto')  ridge\_reg.fit(X\_trainScaled, y\_trainScaled)  predictions = ridge\_reg.predict(X\_testScaled)  # Convert predictions to unscaled predicitons and compare with y\_test.  unscaledPredictionsRidge = scalerY.inverse\_transform(predictions.reshape(-1,1))  print('Root Mean Squared Error:',  np.sqrt(mean\_squared\_error(y\_test, unscaledPredictionsRidge))) |

Example : OLS Using the USA Housing Data Set

Here is code which performs OLS regression on the USA housing data set to predict price. It will be used to answer questions that follow.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn import metrics  from sklearn.model\_selection import train\_test\_split  import statsmodels.api as sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "USA\_Housing.csv"  df = pd.read\_csv(PATH + CSV\_DATA)  # Show all columns.  pd.set\_option('display.max\_columns', None)  # Increase number of columns that display on one line.  pd.set\_option('display.width', 1000)  print(df.head())  print(df.describe())  X = df[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms', 'Area Population']].values  # Adding an intercept \*\*\* This is required \*\*\*. Don't forget this step.  # The intercept centers the error residuals around zero  # which helps to avoid over-fitting.  X = sm.add\_constant(X)  y = df['Price'].values  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)  model = sm.OLS(y\_train, X\_train).fit()  predictions = model.predict(X\_test) # make the predictions by the model  print(model.summary())  print('Root Mean Squared Error:',  np.sqrt(metrics.mean\_squared\_error(y\_test, predictions))) |

Exercise (1 mark)

What is the RMSE value for the regression that is shown Example 4?

|  |
| --- |
| In example 4, the RMSE is 102671.054. (Seems really high) |

Exercise (4 marks)

Scale the X and y data in Example 4 using a MinMaxScaler. Perform an inverse\_transform() on the predictions. Then calculate the RMSE for the unscaled predictions and original y\_test data. Show your code here:

|  |
| --- |
| *import* pandas *as* pd  *import* numpy *as* np  *from* sklearn.metrics *import* mean\_squared\_error  *from* sklearn.model\_selection *import* train\_test\_split  *import* statsmodels.api       *as* sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "USA\_Housing.csv"  df  = pd.read\_csv(CSV\_DATA)  *# Show all columns.*  pd.set\_option('display.max\_columns', None)  *# Increase number of columns that display on one line.*  pd.set\_option('display.width', 1000)  print(df.head())  print(df.describe())  X = df[['Avg. Area Income', 'Avg. Area House Age',  'Avg. Area Number of Rooms', 'Area Population']].values  *# Adding an intercept \*\*\* This is required \*\*\*. Don't forget this step.*  *# The intercept centers the error residuals around zero*  *# which helps to avoid over-fitting.*  X = sm.add\_constant(X)  y = df['Price'].values  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, *test\_size*=0.2, *random\_state*=0)  *from* sklearn.preprocessing *import* MinMaxScaler  scalerX = MinMaxScaler()  scalerX.fit(X\_train)  *# Build scaler for y.*  scalerY = MinMaxScaler()  reshapedYtrain = np.array(y\_train).reshape(-1,1)  scalerY.fit(reshapedYtrain)  *# Scale X\_train, X\_test and y\_train.*  X\_trainScaled = scalerX.transform(X\_train)  X\_testScaled  = scalerX.transform(X\_test)  y\_trainScaled = scalerY.transform(reshapedYtrain)  *# Add constant to scaled data.*  X\_trainScaled = sm.add\_constant(X\_trainScaled)  X\_testScaled  = sm.add\_constant(X\_testScaled)  *# Perform OLS regression.*  model       = sm.OLS(y\_trainScaled, X\_trainScaled).fit()  predictions = model.predict(X\_testScaled) *# make the predictions by the model*  print(model.summary())  *# Convert predictions to unscaled predicitons and compare with y\_test.*  unscaledPredictionsOLS = scalerY.inverse\_transform(predictions.reshape(-1,1))  print('Root Mean Squared Error:',        np.sqrt(mean\_squared\_error(y\_test, unscaledPredictionsOLS))) |

Show your RMSE here:

|  |
| --- |
| Root Mean Squared Error: 102671.0543. (It’s the same as unscaled, seems strange) |

Exercise (4 marks)

Perform Ridge regression to predict housing prices with the data presented in Example 4. Ensure the data is scaled with a MinMaxScaler object. Show your code here:

|  |
| --- |
| *import* pandas *as* pd  *import* numpy *as* np  *from* sklearn.metrics *import* mean\_squared\_error  *from* sklearn.model\_selection *import* train\_test\_split  *import* statsmodels.api       *as* sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "USA\_Housing.csv"  df  = pd.read\_csv(CSV\_DATA)  *# Show all columns.*  pd.set\_option('display.max\_columns', None)  *# Increase number of columns that display on one line.*  pd.set\_option('display.width', 1000)  print(df.head())  print(df.describe())  X = df[['Avg. Area Income', 'Avg. Area House Age',  'Avg. Area Number of Rooms', 'Area Population']].values  *# Adding an intercept \*\*\* This is required \*\*\*. Don't forget this step.*  *# The intercept centers the error residuals around zero*  *# which helps to avoid over-fitting.*  X = sm.add\_constant(X)  y = df['Price'].values  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, *test\_size*=0.2, *random\_state*=0)  *from* sklearn.preprocessing *import* MinMaxScaler  scalerX = MinMaxScaler()  scalerX.fit(X\_train)  *# Build scaler for y.*  scalerY = MinMaxScaler()  reshapedYtrain = np.array(y\_train).reshape(-1,1)  scalerY.fit(reshapedYtrain)  *# Scale X\_train, X\_test and y\_train.*  X\_trainScaled = scalerX.transform(X\_train)  X\_testScaled  = scalerX.transform(X\_test)  y\_trainScaled = scalerY.transform(reshapedYtrain)  *# Add constant to scaled data.*  X\_trainScaled = sm.add\_constant(X\_trainScaled)  X\_testScaled  = sm.add\_constant(X\_testScaled)  *#---------------------------------------------------------------*  *# Perform OLS regression.*  model       = sm.OLS(y\_trainScaled, X\_trainScaled).fit()  predictions = model.predict(X\_testScaled) *# make the predictions by the model*  print(model.summary())  *# Convert predictions to unscaled predicitons and compare with y\_test.*  unscaledPredictionsOLS = scalerY.inverse\_transform(predictions.reshape(-1,1))  print('Root Mean Squared Error:',        np.sqrt(mean\_squared\_error(y\_test, unscaledPredictionsOLS)))  *#---------------------------------------------------------------*  *# Perform Ridge regression.*  print("\nRidge Regression")  *from* sklearn.linear\_model *import*  Ridge  ridge\_reg   = Ridge(*solver*='auto')  ridge\_reg.fit(X\_trainScaled, y\_trainScaled)  predictions = ridge\_reg.predict(X\_testScaled)  unscaledPredictionsRidge = scalerY.inverse\_transform(predictions.reshape(-1,1))  print('Root Mean Squared Error:',        np.sqrt(mean\_squared\_error(y\_test, unscaledPredictionsRidge))) |

Show your RMSE here:

|  |
| --- |
| Root Mean Squared Error: 102552.70189365739 |

Exercise (1 mark)

Compare the RMSE’s for the OLS model and Ridge regression for the USA Housing set. Explain which model performs better.

|  |
| --- |
| The Ridge regression yields lower AIC and BIC values as well as a lower RMSE (indicating better fit). Based on these qualities, I would say that Ridge regression performs better overall. |

## Ridge Classifier

Example : Ridge Classification

As a reference, here is code to perform the same college admissions classification that was done earlier in class with a Ridge classifier. Once again, the results of Ridge regression with scaling exhibit perfect accuracy.

Accuracy: 1.0

Confusion Matrix

Predicted 0 1

Actual

0 5 0

1 0 5

The ridge regression code is highlighted in yellow below.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn import metrics  # Setup data.  candidates = {'gmat': [780,750,690,710,680,730,690,720,  740,690,610,690,710,680,770,610,580,650,540,590,620,  600,550,550,570,670,660,580,650,660,640,620,660,660,  680,650,670,580,590,690],  'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,  3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,  3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,  3.3,3.3,2.3,2.7,3.3,1.7,3.7],  'work\_experience': [3,4,3,5,4,6,1,4,5,  1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,  5,1,2,1,4,5],  'admitted': [1,1,1,1,1,1,0,1,1,0,0,1,  1,1,1,0,0,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,  0,0,1]}  df = pd.DataFrame(candidates,columns= ['gmat', 'gpa',  'work\_experience','admitted'])  print(df)  # Separate into x and y values.  X = df[['gmat', 'gpa','work\_experience']]  y = df['admitted']  # Import the necessary libraries first  from sklearn.feature\_selection import SelectKBest  from sklearn.feature\_selection import chi2  # Show chi-square scores for each feature.  # There is 1-degree freedom since 1 predictor during feature evaluation.  # Generally, >=3.8 is good)  test = SelectKBest(score\_func=chi2, k=3)  chiScores = test.fit(X, y) # Summarize scores  np.set\_printoptions(precision=3)  print("\nPredictor Chi-Square Scores: " + str(chiScores.scores\_))  # Re-assign X with significant columns only after chi-square test.  X = df[['gmat', 'work\_experience']]  # Split data.  X\_train,X\_test,y\_train,y\_test = train\_test\_split(  X, y, test\_size=0.25,random\_state=0)  # Perform logistic regression.  logisticModel = LogisticRegression(fit\_intercept=True, random\_state = 0,  solver='liblinear')  # Stochastic gradient descent models are sensitive to differences  # in scale so a StandardScaler is usually used.  from sklearn.preprocessing import MinMaxScaler  scaler = MinMaxScaler()  scaler.fit(X\_train)  X\_trainScaled = scaler.transform(X\_train)  X\_testScaled = scaler.transform(X\_test)  print("\nRidge Classifier")  from sklearn.linear\_model import RidgeClassifier  clf = RidgeClassifier(solver='auto')  clf.fit(X\_trainScaled, y\_train)  y\_pred = clf.predict(X\_testScaled)  print(y\_pred)  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(y\_test, y\_pred,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix")  print(confusion\_matrix) |

Exercise (1 mark)

Why isn’t the y value in Example 5 scaled?

|  |
| --- |
| The y value in example 5 isn’t scaled because it is an all or nothing field (ie. Admitted or not admitted). The data points would not need to be scaled in this case. |

Exercise (3 marks)

The following code uses logistic regression to predict whether a patient has heart disease.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.model\_selection import train\_test\_split  from sklearn.linear\_model import LogisticRegression  from sklearn import metrics  import statsmodels.api as sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE = 'heart\_disease.csv'  df = pd.read\_csv(PATH + FILE)  print(df)  # Separate into x and y values.  X = df[['age',  'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang',  'oldpeak', 'slope', 'ca', 'thal']]  y = df['target']  # Import the necessary libraries first  from sklearn.feature\_selection import SelectKBest  from sklearn.feature\_selection import chi2  # Show chi-square scores for each feature.  # There is 1-degree freedom since 1 predictor during feature evaluation.  # Generally, >=3.8 is good)  test = SelectKBest(score\_func=chi2, k=3)  chiScores = test.fit(X, y) # Summarize scores  np.set\_printoptions(precision=3)  print("\nPredictor Chi-Square Scores: " + str(chiScores.scores\_))  # Re-assign X with significant columns only after chi-square test.  X = df[['age',  'sex', 'cp', 'trestbps', 'chol', 'fbs', 'thalach', 'exang',  'oldpeak', 'slope', 'ca', 'thal']]  # Split data.  X\_train,X\_test,y\_train,y\_test = train\_test\_split(  X, y, test\_size=0.25,random\_state=0)  # Stochastic gradient descent models are sensitive to differences  # in scale so a StandardScaler is usually used.  from sklearn.preprocessing import MinMaxScaler  scaler = MinMaxScaler()  scaler.fit(X\_train)  X\_trainScaled = scaler.transform(X\_train)  X\_testScaled = scaler.transform(X\_test)  X\_trainScaled = sm.add\_constant(X\_trainScaled)  X\_testScaled = sm.add\_constant(X\_testScaled)  print("\nLogistic Regression")  # Perform logistic regression.  logisticModel = LogisticRegression(fit\_intercept=True, random\_state = 0,  solver='liblinear')  logisticModel.fit(X\_trainScaled, y\_train)  y\_pred = logisticModel.predict(X\_testScaled)  # Show confusion matrix and accuracy scores.  confusion\_matrix = pd.crosstab(y\_test, y\_pred,  rownames=['Actual'],  colnames=['Predicted'])  print('\nAccuracy: ', metrics.accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix")  print(confusion\_matrix) |

Modify the code to use the RidgeClassifier to predict if a patient has heart disease. Show the new confusion matrix and accuracy score here (It is possible that the accuracy may not be as good as the Logistic regression classifier). Compare this output to the confusion matrix and accuracy score for logistic regression:

|  |
| --- |
| Logistic Regression:    RidgeClassifier: |

Show your code here:

|  |
| --- |
| *import* pandas  *as* pd  *import* numpy   *as* np  *from*   sklearn.model\_selection *import* train\_test\_split  *from*   sklearn.linear\_model    *import* LogisticRegression  *from*   sklearn                 *import* metrics  *import* statsmodels.api        *as* sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE  = 'heart\_disease.csv'  df = pd.read\_csv(FILE)  print(df)  *# Separate into x and y values.*  X = df[['age',          'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang',          'oldpeak', 'slope', 'ca', 'thal']]  y = df['target']  *# Import the necessary libraries first*  *from* sklearn.feature\_selection *import* SelectKBest  *from* sklearn.feature\_selection *import* chi2  *# Show chi-square scores for each feature.*  *# There is 1-degree freedom since 1 predictor during feature evaluation.*  *# Generally, >=3.8 is good)*  test      = SelectKBest(*score\_func*=chi2, *k*=3)  chiScores = test.fit(X, y) *# Summarize scores*  np.set\_printoptions(*precision*=3)  print("\nPredictor Chi-Square Scores: " + str(chiScores.scores\_))  *# Re-assign X with significant columns only after chi-square test.*  X = df[['age',          'sex', 'cp', 'trestbps', 'chol', 'fbs',  'thalach', 'exang',          'oldpeak', 'slope', 'ca', 'thal']]  *# Split data.*  X\_train,X\_test,y\_train,y\_test = train\_test\_split(          X, y, *test\_size*=0.25,*random\_state*=0)  *# Stochastic gradient descent models are sensitive to differences*  *# in scale so a StandardScaler is usually used.*  *from* sklearn.preprocessing *import* MinMaxScaler  scaler = MinMaxScaler()  scaler.fit(X\_train)  X\_trainScaled = scaler.transform(X\_train)  X\_testScaled  = scaler.transform(X\_test)  X\_trainScaled = sm.add\_constant(X\_trainScaled)  X\_testScaled  = sm.add\_constant(X\_testScaled)  print("\nLogistic  Regression")  *# Perform logistic regression.*  logisticModel = LogisticRegression(*fit\_intercept*=True, *random\_state* = 0,  *solver*='liblinear')  logisticModel.fit(X\_trainScaled, y\_train)  y\_pred = logisticModel.predict(X\_testScaled)  print("\nRidge Classifier")  *from* sklearn.linear\_model *import* RidgeClassifier  clf = RidgeClassifier(*solver*='auto')  clf.fit(X\_trainScaled, y\_train)  y\_pred = clf.predict(X\_testScaled)  print(y\_pred)  *# Show confusion matrix and accuracy scores.*  confusion\_matrix = pd.crosstab(y\_test, y\_pred,  *rownames*=['Actual'],  *colnames*=['Predicted'])  print('\nAccuracy: ',metrics.accuracy\_score(y\_test, y\_pred))  print("\nConfusion Matrix")  print(confusion\_matrix) |

## Saving and Re-Using Trained Models

Most machine learning models can be saved as a binary file after they have been trained. You can then use the saved model to make predictions without training later. Using a binary model will save you an enormous amount of time as your training sets grow in size.

Example : Saving a Ridge Regression Model

This example shows how you could save and then re-use the saved pre-trained ridge regression model to make predictions. To build this example, add this code to Example 5.

|  |
| --- |
| import pickle  # save model to file  pickle.dump(clf, open("myRidgeModel.dat", "wb"))  # load model from file  loaded\_model = pickle.load(open("myRidgeModel.dat", "rb"))  ridgePredictions = loaded\_model.predict(X\_testScaled)  print(ridgePredictions) |

Exercise (2 marks)

Adjust your program in Exercise 6 to also save your model for logistic regression. Then, load it and use it to make predictions for the test set. Show your code here:

|  |
| --- |
| *import* pandas  *as* pd  *import* numpy   *as* np  *from*   sklearn.model\_selection *import* train\_test\_split  *from*   sklearn.linear\_model    *import* LogisticRegression  *from*   sklearn                 *import* metrics  *import* statsmodels.api        *as* sm  PATH = "/Users/pm/Desktop/DayDocs/data/"  FILE  = 'heart\_disease.csv'  df = pd.read\_csv(FILE)  *# print(df)*  *# Separate into x and y values.*  X = df[['age',          'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang',          'oldpeak', 'slope', 'ca', 'thal']]  y = df['target']  *# Import the necessary libraries first*  *from* sklearn.feature\_selection *import* SelectKBest  *from* sklearn.feature\_selection *import* chi2  *# Show chi-square scores for each feature.*  *# There is 1-degree freedom since 1 predictor during feature evaluation.*  *# Generally, >=3.8 is good)*  test      = SelectKBest(*score\_func*=chi2, *k*=3)  chiScores = test.fit(X, y) *# Summarize scores*  np.set\_printoptions(*precision*=3)  *# print("\nPredictor Chi-Square Scores: " + str(chiScores.scores\_))*  *# Re-assign X with significant columns only after chi-square test.*  X = df[['age',          'sex', 'cp', 'trestbps', 'chol', 'fbs',  'thalach', 'exang',          'oldpeak', 'slope', 'ca', 'thal']]  *# Split data.*  X\_train,X\_test,y\_train,y\_test = train\_test\_split(          X, y, *test\_size*=0.25,*random\_state*=0)  *# Stochastic gradient descent models are sensitive to differences*  *# in scale so a StandardScaler is usually used.*  *from* sklearn.preprocessing *import* MinMaxScaler  scaler = MinMaxScaler()  scaler.fit(X\_train)  X\_trainScaled = scaler.transform(X\_train)  X\_testScaled  = scaler.transform(X\_test)  X\_trainScaled = sm.add\_constant(X\_trainScaled)  X\_testScaled  = sm.add\_constant(X\_testScaled)  logisticModel = LogisticRegression(*fit\_intercept*=True, *random\_state* = 0,  *solver*='liblinear')  logisticModel.fit(X\_trainScaled, y\_train)  y\_pred = logisticModel.predict(X\_testScaled)  *# print("\nRidge Classifier")*  *from* sklearn.linear\_model *import* RidgeClassifier  clf = RidgeClassifier(*solver*='auto')  clf.fit(X\_trainScaled, y\_train)  y\_pred = clf.predict(X\_testScaled)  print("\n ### Original Predictions ### \n")  print(y\_pred)  *# Show confusion matrix and accuracy scores.*  confusion\_matrix = pd.crosstab(y\_test, y\_pred,  *rownames*=['Actual'],  *colnames*=['Predicted'])  *import* pickle  *# save model to file*  pickle.dump(clf, open("myRidgeModel.dat", "wb"))  *# load model from file*  loaded\_model = pickle.load(open("myRidgeModel.dat", "rb"))  ridgePredictions = loaded\_model.predict(X\_testScaled)  print("\n### Ridge Predictions ###\n")  print(ridgePredictions) |