Homework4

November 11, 2024

[2]:
```
import numpy as np
import pandas as pd


from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
df = pd.read_csv("Real estate.csv")
df.rename({'Ouse_price_of_unit_area': 'House_price_of_unit_area'}, axis = 1,␣
 ↪inplace = True)
df.drop("No", axis = 1, inplace = True)
column_maping = {}
for i in df.columns:
    new_column = i[3:].capitalize().replace(' ', '_')
    column_maping[i] = new_column
# Now we will rename the column using the dictionary
df.rename(columns = column_maping, inplace = True)
df.rename({'Ouse_price_of_unit_area': 'House_price_of_unit_area'}, axis = 1,␣
 ↪inplace = True)
X  = df.drop(['Transaction_date', "House_price_of_unit_area"], axis = 1)
y = df['House_price_of_unit_area']
```

[4]:
```
# Do not change the code below
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
# Do not change the code Above

# Add a column of ones to X_train and X_test to account for the bias term␣
 ↪(intercept)
X_train_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]
```

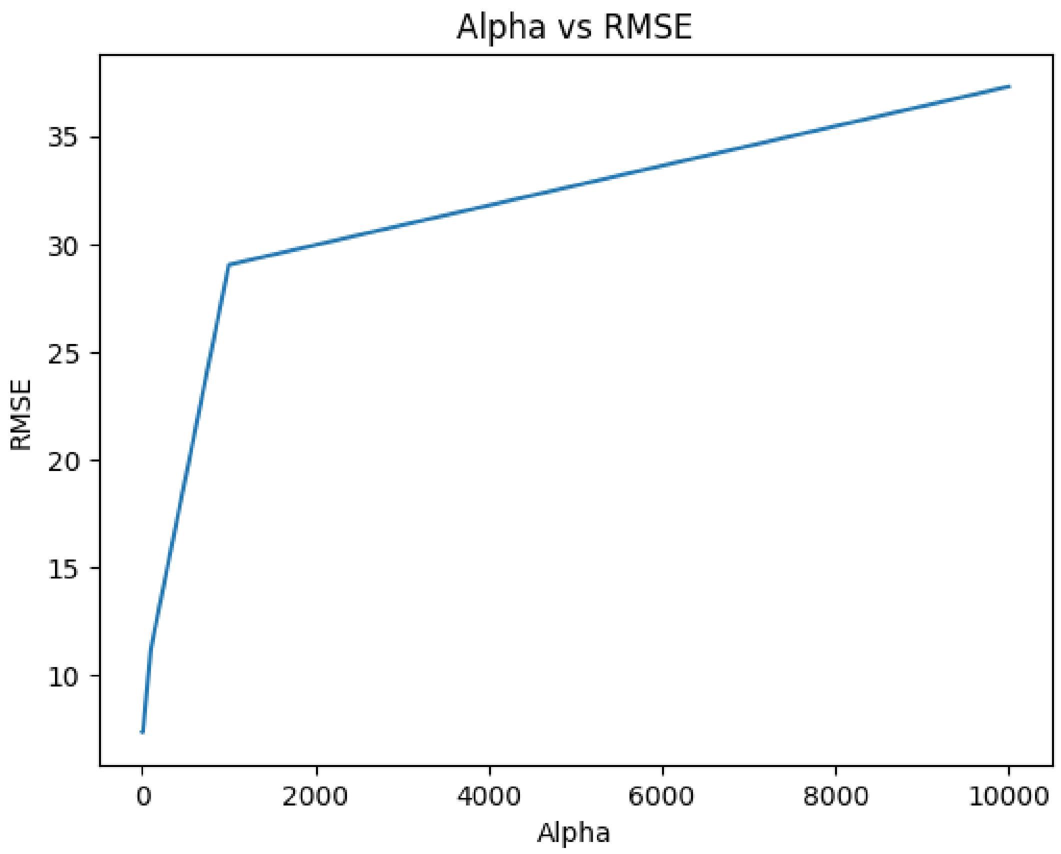
```
X_test_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]

# Perform SVD decomposition on the training data
U, s, VT = np.linalg.svd(X_train_b, full_matrices=False)

# Create diagonal matrix for Sigma
S_diag = np.diag(s)

# Compute the pseudo-inverse of the training data
X_train_pinv = VT.T @ np.linalg.inv(S_diag) @ U.T

# Calculate the weights (regression coefficients), including the bias term␣
 ↪(intercept)
w = X_train_pinv @ y_train

# Make predictions using the testing set
y_pred = X_test_b @ w

se = (y_pred-y_test) ** 2
mse = se.mean()
rmse = mse**0.5
print(f"Root Mean Squre Error using SVD {rmse}")
```

```
Root Mean Squre Error using SVD 7.387891796775459
```

[11]:
```
import matplotlib.pyplot as plt
class Model(object):
    """
     Ridge Regression.
    """

    def fit(self, X, y, alpha=0):
        """
        Fits the ridge regression model to the training data.

        Arguments
        ----------
        X: nxp matrix of n examples with p independent variables
        y: response variable vector for n examples
        alpha: regularization parameter.
        """

        intercept = np.ones((len(X),1))
        X_b = np.c_[intercept,X]

        I = np.identity(X_b.shape[1])
        betha_optim = np.linalg.inv(X_b.T.dot(X_b) + alpha*I).dot(X_b.T).dot(y)
        self.betas = betha_optim
        return betha_optim

    def predict(self, X):
        """
        Predicts the dependent variable of new data using the model.

        Arguments
        ----------
        X: nxp matrix of n examples with p covariates

        Returns
        ----------
        response variable vector for n examples
        """
                # Your code here
        X_predictor = np.c_[np.ones((X.shape[0], 1)), X]
        self.predictions = X_predictor.dot(self.betas)
        return self.predictions

    def rmse(self, X, y):
        """
        Returns the RMSE(Root Mean Squared Error) when the model is validated.

        Arguments
        ----------
        X: nxp matrix of n examples with p covariates
        y: response variable vector for n examples

        Returns
        ----------
        RMSE when model is used to predict y
        """
        y_predict = self.predict(X=X)
        se = (y_predict-y) ** 2
        mse = se.mean()
        rmse = mse**0.5
        return rmse

# Do not change the code below
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
```

```
my_model = Model()
#! TODO: USE DIFFERENT SETS OF ALPHA VALUES TO FIND THE BEST ALPHA
alphas = [0.01, 0.1, 1, 10, 100, 1000, 10000]
RMSES = []
for alpha in alphas:
    my_model.fit(X=X_train, y=y_train, alpha=alpha)
    rmse = my_model.rmse(X=X_test, y=y_test)
    RMSES.append(rmse)
    print(f"Root Mean Squre Error Ridge for alpha: {alpha} is {rmse}")
    print(my_model.rmse(X=X_test, y=y_test))

# plot different alpha values against RMSE
plt.plot(alphas, RMSES)
plt.xlabel('Alpha')
plt.ylabel('RMSE')
plt.title('Alpha vs RMSE')
plt.show()
```

```
Root Mean Squre Error Ridge for alpha: 0.01 is 7.387798950430715
7.387798950430715
Root Mean Squre Error Ridge for alpha: 0.1 is 7.38697328534338
7.38697328534338
Root Mean Squre Error Ridge for alpha: 1 is 7.379693013437837
7.379693013437837
Root Mean Squre Error Ridge for alpha: 10 is 7.39620540310947
7.39620540310947
Root Mean Squre Error Ridge for alpha: 100 is 11.238394958870039
11.238394958870039
Root Mean Squre Error Ridge for alpha: 1000 is 29.033427420684916
29.033427420684916
Root Mean Squre Error Ridge for alpha: 10000 is 37.290048013604675
37.290048013604675
```



[6]:
```python
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Mean Squared Error loss
def mse_loss(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean()

def mse_loss_derivative(y_true, y_pred):
    return -2 * (y_true - y_pred)/ y_true.size

# Forward pass
def forward_pass(x, W_h1, b_h1, W_o, b_o):
    #TODO what is the output of the hidden layer
    a_h1 =  np.dot(x, W_h1) + b_h1
    #TODO apply activation to the output of the hidden layer
    z_h1 = sigmoid(a_h1)
    #TODO use ouput of activation as input to the output layer (it is just␣
 ↪similar to first layer but we don't apply activation)
    y_pred = np.dot(z_h1, W_o) + b_o
    return y_pred, z_h1, a_h1

# Backward pass
def backward_pass(x, y_true, y_pred, z_h1, a_h1, W_h1, W_o):
    # Derivative of loss with respect to y_pred
    #TODO What type of loss function are we using, what is it's derivative ?
    dL_dy_pred = mse_loss_derivative(y_true, y_pred)


    # Gradients for output layer
    #TODO remember the output layer is just a dense layer
    dL_dW_o = np.dot(z_h1.T, dL_dy_pred)

    dL_db_o = np.sum(dL_dy_pred, axis=0, keepdims=True)


    # Derivative of loss with respect to z_h1
    #TODO derivative with respect to the output of activation layer

    dL_dz_h1 = np.dot(dL_dy_pred, W_o.T)


    # Derivative of loss with respect to a_h1
    #TODO derivative with respect to the activation layer
    dL_da_h1 = dL_dz_h1 * sigmoid_derivative(a_h1)

    # Gradients for hidden layer
    dL_dW_h1 = np.dot(x.T, dL_da_h1)
    dL_db_h1 = np.sum(dL_da_h1, axis=0, keepdims=True)


    return dL_dW_h1, dL_db_h1, dL_dW_o, dL_db_o



scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
↪2, random_state = 42)


y_train = y_train.reshape((y_train.shape[0],1))


# Network architecture
input_size = 5 # Number of features
hidden_layer_size = 20 # Number of neurons in layer
output_size = 1 # predicted variable

# Initial random weights and biases for each layer
W_h1 = np.random.randn(input_size, hidden_layer_size) * 0.001
b_h1 = np.zeros((1, hidden_layer_size))
W_o = np.random.randn(hidden_layer_size, output_size) * 0.001
b_o = np.zeros((1, output_size))

learning_rate = 0.2

#To save the weights which give the lowest loss
lowest_loss = float('inf')
best_weights = None


for i in range(100):
    # Forward pass to get predictions
    y_pred, z_h1, a_h1 = forward_pass(X_train, W_h1, b_h1, W_o, b_o)
    loss = mse_loss(y_train, y_pred)

    if loss < lowest_loss:
        lowest_loss = loss
        # Save the best weights and biases
        best_weights = (W_h1.copy(), b_h1.copy(), W_o.copy(), b_o.copy())

    # Backward pass to get gradients

    dL_dW_h1, dL_db_h1, dL_dW_o, dL_db_o = backward_pass(X_train, y_train,␣
↪y_pred, z_h1, a_h1, W_h1, W_o)

    # Now you would use the gradients to update the weights and biases
    W_h1 -= learning_rate * dL_dW_h1
    b_h1 -= learning_rate * dL_db_h1
    W_o -= learning_rate * dL_dW_o
    b_o -= learning_rate * dL_db_o
```

```
W_h1_best, b_h1_best, W_o_best, b_o_best = best_weights
y_pred_test, _, _ = forward_pass(X_test, W_h1_best, b_h1_best, W_o_best,␣
  ↪b_o_best)
se = (y_pred_test-y_test) ** 2
mse = se.mean()
rmse = mse**0.5
print(f"Root Mean Squre Error 1 Layer MLP {rmse}")

#(331, 1)
#Root Mean Squre Error 1 Layer MLP 15.69657819926082
```

```
Root Mean Squre Error 1 Layer MLP 13.113716211105924
```

Bonus Lets add one more hidden layer (10 Points)

0.0.1 You must write down all gradients and complete the code below to get full bonus points

[7]:
```
# Forward pass
def forward_pass(x, W_h1, b_h1, W_h2, b_h2, W_o, b_o):
    #TODO compute ouput of first hidden layer
    a_h1 = np.dot(x, W_h1) + b_h1
    z_h1 = sigmoid(a_h1) # apply activation to ouputs of first hidden layer

    #TODO compute ouput of second hidden layer
    a_h2 = np.dot(z_h1, W_h2) + b_h2
    z_h2 = sigmoid(a_h2) #apply activation to ouputs of first hidden layer

    # compute ouput of output layer, why don't we apply activation ?
    y_pred = np.dot(z_h2, W_o) + b_o

    return y_pred, z_h1, a_h1, z_h2, a_h2

# Backward pass
def backward_pass(x, y_true, y_pred, z_h1, a_h1, z_h2, a_h2, W_h1, W_h2, W_o):

    # Derivative of loss with respect to y_pred, what kind of loss are we using␣
  ↪?
    #TODO
    dL_dy_pred = mse_loss_derivative(y_true, y_pred)

    # Gradients for output layer
    #TODO
    dL_dW_o = np.dot(z_h2.T, dL_dy_pred)
    #TODO
    dL_db_o = np.sum(dL_dy_pred, axis=0, keepdims=True)
    # Derivative of loss with respect to z_h2
    #TODO
    dL_dz_h2 = np.dot(dL_dy_pred, W_o.T)

    # Derivative of loss with respect to a_h2
    #TODO
    dL_da_h2 =  dL_dz_h2 * sigmoid_derivative(a_h2)

    # Gradients for second hidden layer
    #TODO
    dL_dW_h2 =  np.dot(z_h1.T, dL_da_h2)
    #TODO
    dL_db_h2 =  np.sum(dL_da_h2, axis=0, keepdims=True)

    # Derivative of loss with respect to z_h1
    #TODO
    dL_dz_h1 =  np.dot(dL_da_h2, W_h2.T)

    # Derivative of loss with respect to a_h1
    #TODO
    dL_da_h1 =  dL_dz_h1 * sigmoid_derivative(a_h1)

    # Gradients for first hidden layer
    #TODO
    dL_dW_h1 =  np.dot(x.T, dL_da_h1)
    #TODO
    dL_db_h1 =  np.sum(dL_da_h1, axis=0, keepdims=True)

    return dL_dW_h1, dL_db_h1, dL_dW_h2, dL_db_h2, dL_dW_o, dL_db_o



# Random input and true output (modify these according to your dataset)
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
y_train = y_train.reshape((y_train.shape[0],1))

# Network architecture
input_size = 5 # Number of features
hidden_layer1_size = 100
hidden_layer2_size = 20
output_size = 1
```

```
# Initial random weights and biases for each layer
W_h1 = np.random.randn(input_size, hidden_layer1_size) * 0.001
b_h1 = np.zeros((1, hidden_layer1_size))
W_h2 = np.random.randn(hidden_layer1_size, hidden_layer2_size) * 0.001
b_h2 = np.zeros((1, hidden_layer2_size))
W_o = np.random.randn(hidden_layer2_size, output_size) * 0.001
b_o = np.zeros((1, output_size))


learning_rate = 0.1

#Training loop
for i in range(200):
    # Forward pass to get predictions
    y_pred, z_h1, a_h1, z_h2, a_h2 = forward_pass(X_train, W_h1, b_h1, W_h2,␣
 ↪b_h2, W_o, b_o)
    #TODO Compute the loss
    # Backward pass to get gradients
    dL_dW_h1, dL_db_h1, dL_dW_h2, dL_db_h2, dL_dW_o, dL_db_o =␣
 ↪backward_pass(X_train, y_train, y_pred, z_h1, a_h1, z_h2, a_h2, W_h1, W_h2,␣
 ↪W_o)

    # Now you would use the gradients to update the weights and biases for each␣
 ↪layer
    #TODO
    W_h1 -= learning_rate * dL_dW_h1
    #TODO
    b_h1 -= learning_rate * dL_db_h1

    #TODO
    W_h2 -= learning_rate * dL_dW_h2
    #TODO
    b_h2 -= learning_rate * dL_db_h2
    #TODO
    W_o -= learning_rate * dL_dW_o
    #TODO
    b_o -= learning_rate * dL_db_o

y_pred, _, _, _, _ = forward_pass(X_test, W_h1, b_h1, W_h2, b_h2, W_o, b_o)
# print(y_pred)
se = (y_pred-y_test) ** 2
mse = se.mean()
rmse = mse**0.5
print(f"Root Mean Squre Error 2 Layer MLP {rmse}")
```

```
Root Mean Squre Error 2 Layer MLP 13.1137162111749
```

```
[8]: %pip install keras
     %pip install tensorflow

     from keras.models import Sequential
     from keras.layers import Dense, Dropout
     from keras.callbacks import EarlyStopping


     model = Sequential()


     model.add(Dense(400, input_dim = 5, kernel_initializer = 'he_uniform', ␣
      ↪activation = 'relu')) #
     model.add(Dropout(0.2))


     model.add(Dense(400, input_dim = 5, kernel_initializer = 'he_uniform',␣
      ↪activation = 'relu')) #
     model.add(Dropout(0.2))

     model.add(Dense(400, kernel_initializer = 'he_uniform',activation = 'relu')) #
     model.add(Dropout(0.2))

     model.add(Dense(1, activation = 'linear'))

     model.compile(loss = 'mean_squared_error', optimizer = 'adam')
     model.summary()
```

```
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages
(3.4.1)
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-
packages (from keras) (1.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from keras) (1.26.4)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages
(from keras) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages
(from keras) (0.0.8)
Requirement already satisfied: h5py in /usr/local/lib/python3.10/dist-packages
(from keras) (3.12.1)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages
(from keras) (0.13.0)
Requirement already satisfied: ml-dtypes in /usr/local/lib/python3.10/dist-
packages (from keras) (0.4.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from keras) (24.1)
Requirement already satisfied: typing-extensions>=4.5.0 in
/usr/local/lib/python3.10/dist-packages (from optree->keras) (4.12.2)
```

```
Requirement already satisfied: markdown-it-py>=2.2.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras) (2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-
packages (from markdown-it-py>=2.2.0->rich->keras) (0.1.2)
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-
packages (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: h5py>=3.10.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (3.12.1)
Requirement already satisfied: libclang>=13.0.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.4.1)
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (24.1)
Requirement already satisfied:
protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3
in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.20.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (75.1.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.5.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (4.12.2)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.16.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.64.1)
Requirement already satisfied: tensorboard<2.18,>=2.17 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.17.0)
Requirement already satisfied: keras>=3.2.0 in /usr/local/lib/python3.10/dist-
```

```
packages (from tensorflow) (3.4.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.37.1)
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.26.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from astunparse>=1.6.0->tensorflow)
(0.44.0)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->tensorflow) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->tensorflow) (0.13.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorflow)
(3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests<3,>=2.21.0->tensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorflow)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorflow)
(2024.8.30)
Requirement already satisfied: markdown>=2.6.8 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.7)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.18,>=2.17->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.1.2)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/usr/local/lib/python3.10/dist-packages (from
werkzeug>=1.0.1->tensorboard<2.18,>=2.17->tensorflow) (3.0.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras>=3.2.0->tensorflow)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras>=3.2.0->tensorflow)
(2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-
packages (from markdown-it-py>=2.2.0->rich->keras>=3.2.0->tensorflow) (0.1.2)
```

```
Model: "sequential"
```

```
Layer (type)                          Output Shape                              ␣
↪Param #

dense (Dense)                         (None, 400)                                 ␣
↪2,400

dropout (Dropout)                     (None, 400)                                   ␣
↪  0

dense_1 (Dense)                       (None, 400)                               ␣
↪160,400

dropout_1 (Dropout)                   (None, 400)                                   ␣
↪  0

dense_2 (Dense)                       (None, 400)                               ␣
↪160,400

dropout_2 (Dropout)                   (None, 400)                                   ␣
↪  0

dense_3 (Dense)                       (None, 1)                                     ␣
↪401
```

```
Total params: 323,601 (1.23 MB)

Trainable params: 323,601 (1.23 MB)

Non-trainable params: 0 (0.00 B)
```

[9]:
```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
y_train = y_train.reshape((y_train.shape[0],1))
history = model.fit(X_train, y_train,  epochs = 200, validation_data=(X_test,␣
 ↪y_test),
                    callbacks = EarlyStopping(monitor = 'val_loss',patience =␣
↪40))
```

```
Epoch 1/200
11/11             4s 83ms/step -
loss: 986.4611 - val_loss: 187.3347
Epoch 2/200
11/11             0s 34ms/step -
loss: 179.3348 - val_loss: 156.9577
Epoch 3/200
11/11             1s 26ms/step -
loss: 128.8178 - val_loss: 97.9532
Epoch 4/200
11/11             1s 20ms/step -
loss: 113.4353 - val_loss: 100.7919
Epoch 5/200
11/11             0s 24ms/step -
loss: 98.3579 - val_loss: 89.4237
Epoch 6/200
11/11             1s 20ms/step -
loss: 139.2811 - val_loss: 83.5790
Epoch 7/200
11/11             0s 22ms/step -
loss: 95.1077 - val_loss: 73.6563
Epoch 8/200
11/11             0s 23ms/step -
loss: 102.2080 - val_loss: 79.5526
Epoch 9/200
11/11             0s 19ms/step -
loss: 88.3762 - val_loss: 65.0579
Epoch 10/200
11/11             0s 21ms/step -
loss: 73.7353 - val_loss: 61.5901
Epoch 11/200
11/11             0s 16ms/step -
loss: 102.1125 - val_loss: 59.2914
Epoch 12/200
11/11             0s 30ms/step -
loss: 91.2132 - val_loss: 64.8738
Epoch 13/200
11/11             1s 24ms/step -
loss: 97.1009 - val_loss: 59.5561
Epoch 14/200
11/11             0s 21ms/step -
loss: 93.7803 - val_loss: 59.5480
Epoch 15/200
11/11             0s 17ms/step -
loss: 93.1076 - val_loss: 58.1397
```

```
Epoch 16/200
11/11            0s 18ms/step -
loss: 88.8036 - val_loss: 54.3965
Epoch 17/200
11/11            0s 17ms/step -
loss: 70.2638 - val_loss: 49.3561
Epoch 18/200
11/11            1s 38ms/step -
loss: 84.1058 - val_loss: 50.6575
Epoch 19/200
11/11            1s 33ms/step -
loss: 70.0297 - val_loss: 54.6827
Epoch 20/200
11/11            1s 21ms/step -
loss: 88.8656 - val_loss: 55.3178
Epoch 21/200
11/11            1s 28ms/step -
loss: 67.8646 - val_loss: 51.2793
Epoch 22/200
11/11            0s 18ms/step -
loss: 76.7303 - val_loss: 61.0252
Epoch 23/200
11/11            0s 23ms/step -
loss: 89.9349 - val_loss: 53.9564
Epoch 24/200
11/11            0s 18ms/step -
loss: 77.2778 - val_loss: 43.7928
Epoch 25/200
11/11            0s 19ms/step -
loss: 87.9341 - val_loss: 51.7264
Epoch 26/200
11/11            1s 23ms/step -
loss: 73.4521 - val_loss: 50.7750
Epoch 27/200
11/11            1s 34ms/step -
loss: 85.6545 - val_loss: 51.1736
Epoch 28/200
11/11            1s 31ms/step -
loss: 89.2620 - val_loss: 68.4901
Epoch 29/200
11/11            1s 37ms/step -
loss: 74.7465 - val_loss: 51.0189
Epoch 30/200
11/11            0s 25ms/step -
loss: 68.6750 - val_loss: 44.9507
Epoch 31/200
11/11            0s 16ms/step -
loss: 71.4699 - val_loss: 50.2313
```

```
Epoch 32/200
11/11              0s 20ms/step -
loss: 88.5188 - val_loss: 44.1377
Epoch 33/200
11/11              0s 18ms/step -
loss: 67.7506 - val_loss: 46.9963
Epoch 34/200
11/11              0s 16ms/step -
loss: 72.2658 - val_loss: 45.3241
Epoch 35/200
11/11              0s 18ms/step -
loss: 89.9286 - val_loss: 44.4716
Epoch 36/200
11/11              0s 17ms/step -
loss: 80.5026 - val_loss: 51.0675
Epoch 37/200
11/11              0s 18ms/step -
loss: 63.0110 - val_loss: 42.5683
Epoch 38/200
11/11              0s 17ms/step -
loss: 57.0131 - val_loss: 41.4556
Epoch 39/200
11/11              0s 12ms/step -
loss: 81.6651 - val_loss: 41.3081
Epoch 40/200
11/11              0s 14ms/step -
loss: 74.9821 - val_loss: 45.5565
Epoch 41/200
11/11              0s 10ms/step -
loss: 73.2874 - val_loss: 61.6684
Epoch 42/200
11/11              0s 11ms/step -
loss: 64.2432 - val_loss: 44.2818
Epoch 43/200
11/11              0s 10ms/step -
loss: 90.6332 - val_loss: 42.1117
Epoch 44/200
11/11              0s 11ms/step -
loss: 70.9702 - val_loss: 50.3262
Epoch 45/200
11/11              0s 11ms/step -
loss: 79.0764 - val_loss: 57.3779
Epoch 46/200
11/11              0s 11ms/step -
loss: 97.1907 - val_loss: 43.1374
Epoch 47/200
11/11              0s 12ms/step -
loss: 104.1230 - val_loss: 41.1444
```

```
Epoch 48/200
11/11            0s 12ms/step -
loss: 72.1089 - val_loss: 39.1258
Epoch 49/200
11/11            0s 12ms/step -
loss: 76.6023 - val_loss: 67.6951
Epoch 50/200
11/11            0s 13ms/step -
loss: 75.2273 - val_loss: 38.8589
Epoch 51/200
11/11            0s 10ms/step -
loss: 95.9648 - val_loss: 42.3573
Epoch 52/200
11/11            0s 11ms/step -
loss: 61.2024 - val_loss: 44.3708
Epoch 53/200
11/11            0s 11ms/step -
loss: 77.4434 - val_loss: 48.9275
Epoch 54/200
11/11            0s 10ms/step -
loss: 72.5472 - val_loss: 47.1935
Epoch 55/200
11/11            0s 13ms/step -
loss: 71.6391 - val_loss: 50.4937
Epoch 56/200
11/11            0s 13ms/step -
loss: 72.6620 - val_loss: 48.8558
Epoch 57/200
11/11            0s 10ms/step -
loss: 78.5944 - val_loss: 55.0177
Epoch 58/200
11/11            0s 12ms/step -
loss: 68.6633 - val_loss: 42.5117
Epoch 59/200
11/11            0s 12ms/step -
loss: 56.7422 - val_loss: 37.6094
Epoch 60/200
11/11            0s 10ms/step -
loss: 58.6207 - val_loss: 38.5807
Epoch 61/200
11/11            0s 11ms/step -
loss: 97.7858 - val_loss: 56.3075
Epoch 62/200
11/11            0s 11ms/step -
loss: 97.2284 - val_loss: 48.2464
Epoch 63/200
11/11            0s 10ms/step -
loss: 59.8639 - val_loss: 39.2761
```

```
Epoch 64/200
11/11            0s 12ms/step -
loss: 67.8466 - val_loss: 50.1880
Epoch 65/200
11/11            0s 10ms/step -
loss: 71.4863 - val_loss: 40.2775
Epoch 66/200
11/11            0s 10ms/step -
loss: 83.0523 - val_loss: 45.2619
Epoch 67/200
11/11            0s 12ms/step -
loss: 58.9747 - val_loss: 49.9192
Epoch 68/200
11/11            0s 10ms/step -
loss: 65.9682 - val_loss: 39.6778
Epoch 69/200
11/11            0s 12ms/step -
loss: 99.1024 - val_loss: 40.7259
Epoch 70/200
11/11            0s 12ms/step -
loss: 69.9960 - val_loss: 47.4773
Epoch 71/200
11/11            0s 13ms/step -
loss: 74.9860 - val_loss: 51.0106
Epoch 72/200
11/11            0s 10ms/step -
loss: 55.5364 - val_loss: 37.8628
Epoch 73/200
11/11            0s 12ms/step -
loss: 57.7738 - val_loss: 41.2249
Epoch 74/200
11/11            0s 11ms/step -
loss: 55.6683 - val_loss: 38.5737
Epoch 75/200
11/11            0s 11ms/step -
loss: 85.8466 - val_loss: 73.6150
Epoch 76/200
11/11            0s 13ms/step -
loss: 101.9437 - val_loss: 47.3916
Epoch 77/200
11/11            0s 12ms/step -
loss: 59.1887 - val_loss: 42.2810
Epoch 78/200
11/11            0s 10ms/step -
loss: 79.0101 - val_loss: 42.5591
Epoch 79/200
11/11            0s 11ms/step -
loss: 60.5573 - val_loss: 39.3539
```

```
Epoch 80/200
11/11            0s 12ms/step -
loss: 85.0743 - val_loss: 47.3171
Epoch 81/200
11/11            0s 12ms/step -
loss: 52.8557 - val_loss: 41.1603
Epoch 82/200
11/11            0s 10ms/step -
loss: 74.5153 - val_loss: 36.3660
Epoch 83/200
11/11            0s 10ms/step -
loss: 76.8667 - val_loss: 38.8608
Epoch 84/200
11/11            0s 11ms/step -
loss: 85.8755 - val_loss: 39.9939
Epoch 85/200
11/11            0s 10ms/step -
loss: 59.3341 - val_loss: 39.6148
Epoch 86/200
11/11            0s 11ms/step -
loss: 77.3023 - val_loss: 42.0261
Epoch 87/200
11/11            0s 14ms/step -
loss: 56.8682 - val_loss: 38.1092
Epoch 88/200
11/11            0s 10ms/step -
loss: 55.7162 - val_loss: 41.7898
Epoch 89/200
11/11            0s 10ms/step -
loss: 62.8504 - val_loss: 42.8936
Epoch 90/200
11/11            0s 11ms/step -
loss: 54.4851 - val_loss: 41.7834
Epoch 91/200
11/11            0s 14ms/step -
loss: 56.8465 - val_loss: 39.0139
Epoch 92/200
11/11            0s 17ms/step -
loss: 65.4451 - val_loss: 38.8984
Epoch 93/200
11/11            0s 29ms/step -
loss: 60.2444 - val_loss: 41.7889
Epoch 94/200
11/11            1s 27ms/step -
loss: 79.2747 - val_loss: 46.0970
Epoch 95/200
11/11            1s 24ms/step -
loss: 56.7410 - val_loss: 41.3795
```

```
Epoch 96/200
11/11            0s 16ms/step -
loss: 55.5673 - val_loss: 37.1161
Epoch 97/200
11/11            0s 19ms/step -
loss: 66.8238 - val_loss: 40.3603
Epoch 98/200
11/11            0s 17ms/step -
loss: 88.2185 - val_loss: 54.7229
Epoch 99/200
11/11            0s 19ms/step -
loss: 73.9865 - val_loss: 66.6440
Epoch 100/200
11/11            0s 20ms/step -
loss: 88.4238 - val_loss: 44.2323
Epoch 101/200
11/11            0s 11ms/step -
loss: 61.8064 - val_loss: 41.4330
Epoch 102/200
11/11            0s 12ms/step -
loss: 59.4315 - val_loss: 42.0228
Epoch 103/200
11/11            0s 10ms/step -
loss: 82.1133 - val_loss: 45.4302
Epoch 104/200
11/11            0s 12ms/step -
loss: 55.7618 - val_loss: 36.3542
Epoch 105/200
11/11            0s 11ms/step -
loss: 49.7134 - val_loss: 42.4889
Epoch 106/200
11/11            0s 10ms/step -
loss: 59.3687 - val_loss: 45.5686
Epoch 107/200
11/11            0s 12ms/step -
loss: 78.7495 - val_loss: 42.4134
Epoch 108/200
11/11            0s 15ms/step -
loss: 50.0286 - val_loss: 49.7534
Epoch 109/200
11/11            0s 12ms/step -
loss: 67.2207 - val_loss: 43.6824
Epoch 110/200
11/11            0s 11ms/step -
loss: 69.1416 - val_loss: 37.8418
Epoch 111/200
11/11            0s 13ms/step -
loss: 79.1478 - val_loss: 38.2905
```

```
Epoch 112/200
11/11            0s 23ms/step -
loss: 74.2002 - val_loss: 41.7023
Epoch 113/200
11/11            1s 28ms/step -
loss: 57.7071 - val_loss: 43.4487
Epoch 114/200
11/11            1s 19ms/step -
loss: 49.6566 - val_loss: 37.9704
Epoch 115/200
11/11            0s 12ms/step -
loss: 55.2487 - val_loss: 38.8944
Epoch 116/200
11/11            0s 10ms/step -
loss: 65.2462 - val_loss: 43.7662
Epoch 117/200
11/11            0s 19ms/step -
loss: 58.9636 - val_loss: 41.4765
Epoch 118/200
11/11            0s 21ms/step -
loss: 53.6188 - val_loss: 41.2831
Epoch 119/200
11/11            1s 19ms/step -
loss: 68.3268 - val_loss: 38.9324
Epoch 120/200
11/11            0s 29ms/step -
loss: 65.8823 - val_loss: 39.5982
Epoch 121/200
11/11            0s 20ms/step -
loss: 89.0370 - val_loss: 41.0503
Epoch 122/200
11/11            0s 22ms/step -
loss: 51.6565 - val_loss: 41.2813
Epoch 123/200
11/11            0s 22ms/step -
loss: 50.5487 - val_loss: 43.7615
Epoch 124/200
11/11            0s 25ms/step -
loss: 53.0840 - val_loss: 41.4637
Epoch 125/200
11/11            1s 31ms/step -
loss: 47.4496 - val_loss: 36.3940
Epoch 126/200
11/11            0s 18ms/step -
loss: 94.7648 - val_loss: 55.7338
Epoch 127/200
11/11            0s 19ms/step -
loss: 69.1148 - val_loss: 65.4991
```

```
Epoch 128/200
11/11             0s 17ms/step -
loss: 63.3496 - val_loss: 37.4291
Epoch 129/200
11/11             0s 17ms/step -
loss: 63.0614 - val_loss: 37.4612
Epoch 130/200
11/11             0s 18ms/step -
loss: 67.0856 - val_loss: 38.8843
Epoch 131/200
11/11             0s 18ms/step -
loss: 80.2918 - val_loss: 43.9271
Epoch 132/200
11/11             0s 22ms/step -
loss: 79.5727 - val_loss: 55.0062
Epoch 133/200
11/11             0s 21ms/step -
loss: 70.1385 - val_loss: 38.4630
Epoch 134/200
11/11             0s 32ms/step -
loss: 50.2648 - val_loss: 44.9514
Epoch 135/200
11/11             1s 28ms/step -
loss: 70.8141 - val_loss: 45.0127
Epoch 136/200
11/11             0s 26ms/step -
loss: 54.2321 - val_loss: 42.3256
Epoch 137/200
11/11             1s 32ms/step -
loss: 60.5990 - val_loss: 41.5957
Epoch 138/200
11/11             1s 39ms/step -
loss: 69.2141 - val_loss: 42.1470
Epoch 139/200
11/11             1s 38ms/step -
loss: 50.8823 - val_loss: 51.2459
Epoch 140/200
11/11             0s 21ms/step -
loss: 58.0639 - val_loss: 38.5995
Epoch 141/200
11/11             1s 19ms/step -
loss: 55.5084 - val_loss: 38.1223
Epoch 142/200
11/11             0s 25ms/step -
loss: 42.5869 - val_loss: 42.7461
Epoch 143/200
11/11             0s 20ms/step -
loss: 75.5971 - val_loss: 43.5264
```

```
Epoch 144/200
11/11              0s 26ms/step -
loss: 53.2123 - val_loss: 52.3444
```

[10]:
```
print(f"RMSE error using deeper neural network {model.evaluate(X_test,␣
 ↪y_test)**0.5}")
```

```
3/3              0s 6ms/step - loss:
60.4568
RMSE error using deeper neural network 7.234945254168172
```