hw2_linear_regression

November 6, 2023

1 Homework2: Linear Regression

In this assignment, we will start with utilizing scikit-learn to implement a linear regression model. Afterwards, we will be dropping scikit-learn and implementing these algorithms from scratch without the use of machine learning libraries. While you would likely never have to implement your own linear regression algorithm from scratch in practice, such a skill is valuable to have as you progress further into the field and find many scenarios where you actually may need to perform such implementations manually. Additionally, implementing algorithms from scratch will help you better understand the underlying mathematics behind each model.

1.1 Import Libraries

We will be using the following libraries for this homework assignment. For the questions requiring manual implementation, the pre-existing implementations from scikit-learn should *not* be used.

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import operator
%matplotlib inline

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
```

1.2 Preparing Data

The file named **dataset1.csv** includes data that was generated from an n-degree polynomial with some gaussian noise. The data has 2 columns - first column is the feature (input) and the second column is its label (output). The first step is to load the data and split them into training, validation, and test sets. A reminder that the purpose of each of the splitted sets are as follows:

- **Training Set**: The sample of data used to fit the model
- **Validation Set**: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters.

- **Test Set**: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

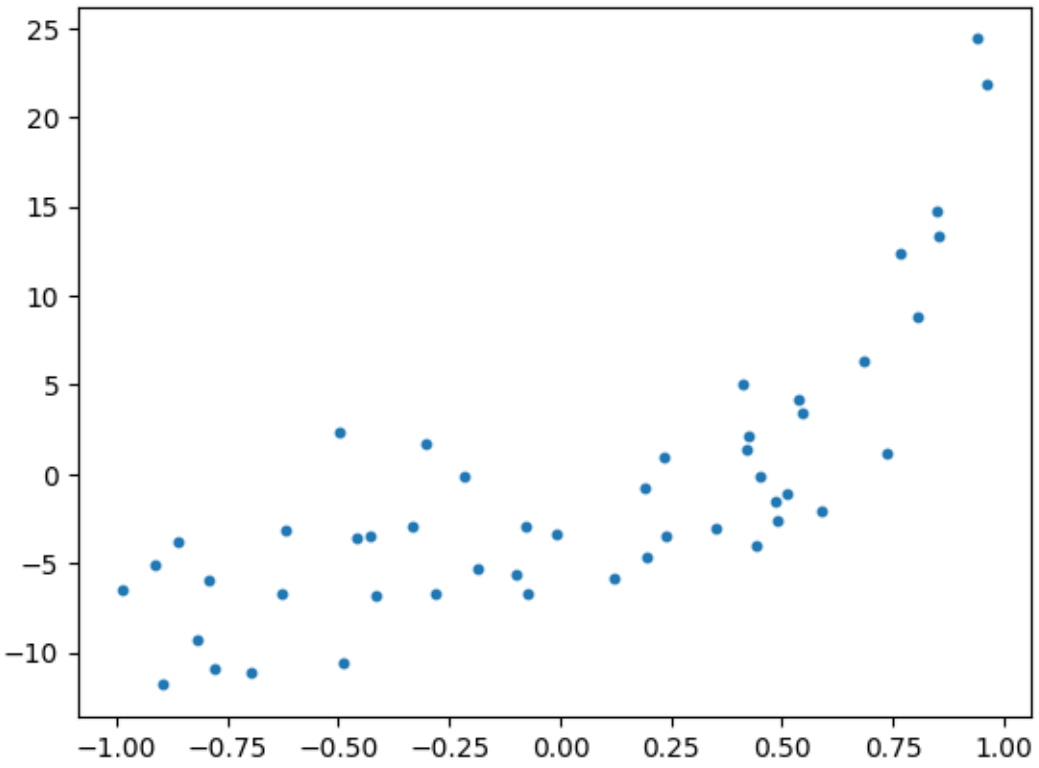
In the section below, we load the csv file and split the data randomnly into 3 equal sets.

Note that in practice, we usually aim for around a 70-20-10 split for train, valid, and test respectively, but due to limited data in our case, we will do an even split in order to have sufficient data for evaluation

[ ]:
```
# Load the data and split into 3 equal sets
data = pd.read_csv('./datasets/dataset1.csv', header=None)
data = data.iloc[:, :-1]
train, valid, test = np.split(data, [int(.33*len(data)), int(.66*len(data))])

# We sort the data in order for plotting purposes later
train.sort_values(by=[0], inplace=True)
valid.sort_values(by=[0], inplace=True)
test.sort_values(by=[0], inplace=True)
```

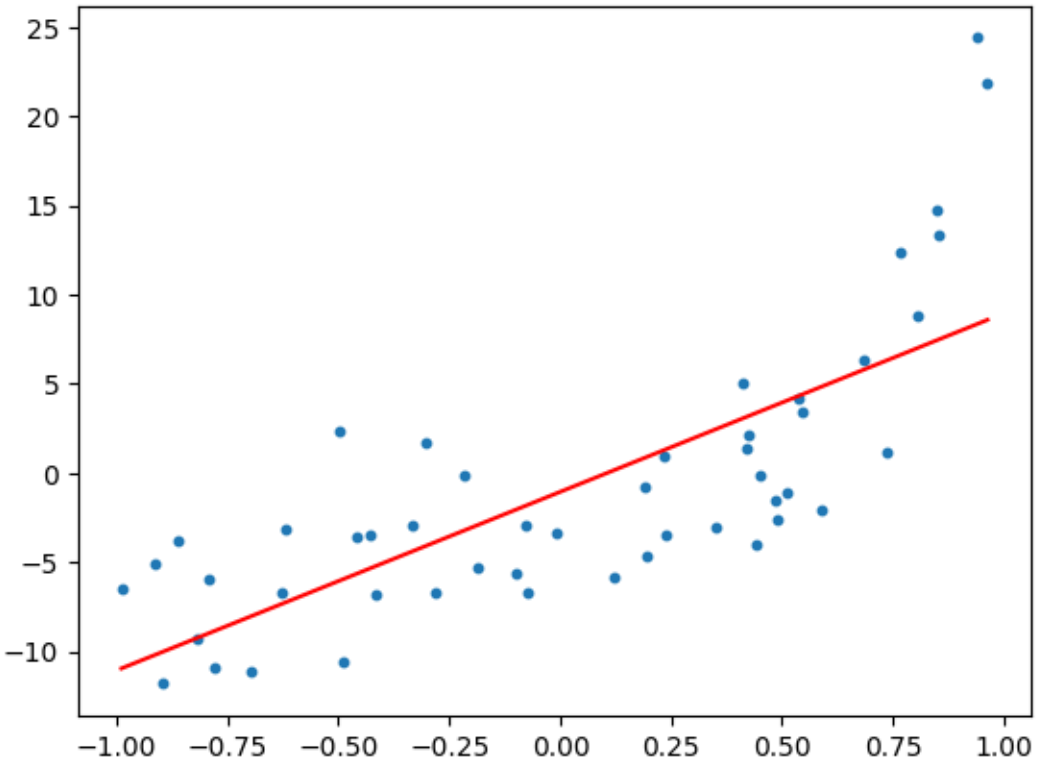
Let's take a look at what our data looks like

[ ]:
```
plt.scatter(train[0], train[1], s=10)
plt.show()
```



Let's apply a linear regression model using scikit-learn and see what the results look like.

[ ]:
```
# Reshape arrays since scikit-learn only takes in 2D arrays
train_x = np.array(train[0])
train_y = np.array(train[1])
valid_x = np.array(valid[0])
valid_y = np.array(valid[1])

train_x = train_x.reshape(-1,1)
train_y = train_y.reshape(-1,1)
valid_x = valid_x.reshape(-1,1)
valid_y = valid_y.reshape(-1,1)

# Apply linear regression model
model = LinearRegression()
model.fit(train_x, train_y)
y_pred = model.predict(train_x)

# Plot the results
plt.scatter(train_x, train_y, s=10)
plt.plot(train_x, y_pred, color='r')
plt.show()
```



By analyzing the line of best fit above, we can see that a straight line is unable to capture the patterns of the data. This is an example of underfitting. As seen in the latest lecture, we can generate a higher order equation by adding powers of the original features as new features.

The linear model:

$$y(x) = w_1 x + w_0$$

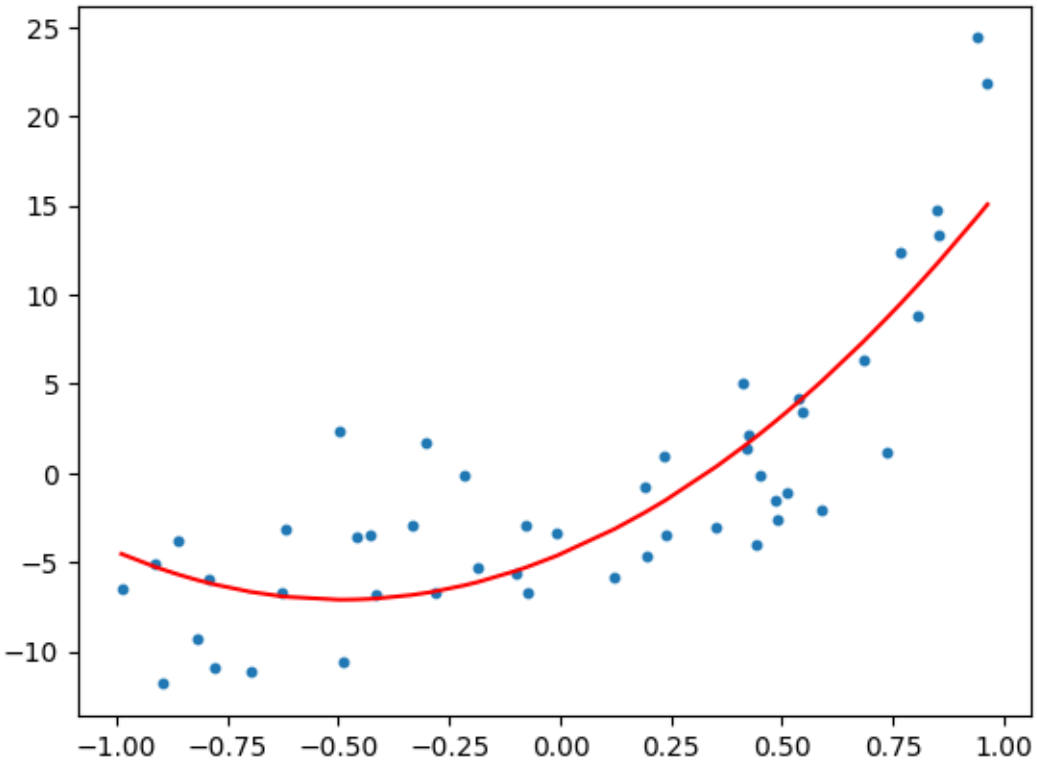
can be transformed to a polynomial model such as:

$$y(x) = w_2 x^2 + w_1 x + w_0$$

Note that this is still considered to be linear model as the coefficients/weights associated with the features are still linear. x^2 is only a feature. However the curve that we would be fitting in this case is quadratic in nature.

Below we show an example of a quadratic curve being fit to the data

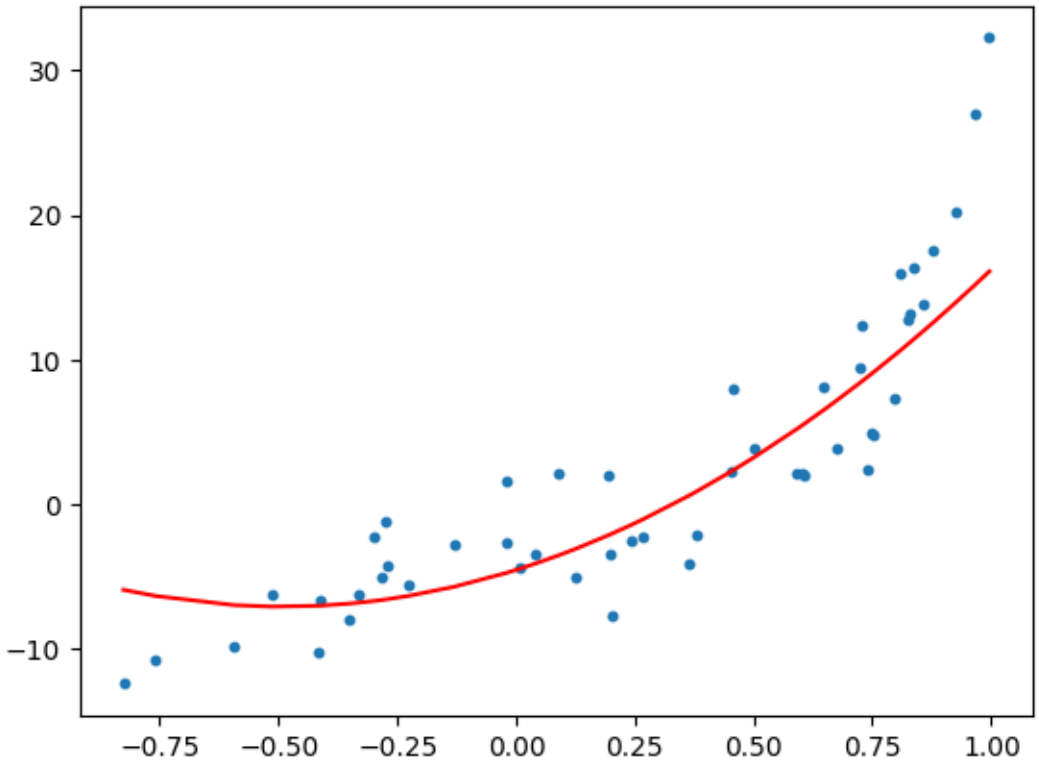
[ ]:
```
# Create polynomial features with degree 2
polynomial_features = PolynomialFeatures(degree=2)
x_poly = polynomial_features.fit_transform(train_x)

# Apply linear regression
model = LinearRegression()
model.fit(x_poly, train_y)
y_poly_pred = model.predict(x_poly)

# Plot the results
plt.scatter(train_x, train_y, s=10)
plt.plot(train_x, y_poly_pred, color='r')
plt.show()
```



As you can see, we get a slightly better fit with a quadratic curve. Let's use the model to make predictions on our validation set and compute the mean squared error, which is the error which we wish to minimize.

[ ]:
```
# Make predictions using pretrained model
valid_y_poly_pred = model.predict(polynomial_features.fit_transform(valid_x))

# Calculate mean squared error
mse = mean_squared_error(valid_y, valid_y_poly_pred)
print("Mean Squared Error: {}".format(mse))

# Plot the prediction results
plt.scatter(valid_x, valid_y, s=10)
plt.plot(valid_x, valid_y_poly_pred, color='r')
plt.show()
```

```
Mean Squared Error: 20.485214511024232
```



1.3 Question 1: Polynomial Regression Using Scikit-learn [10pts]

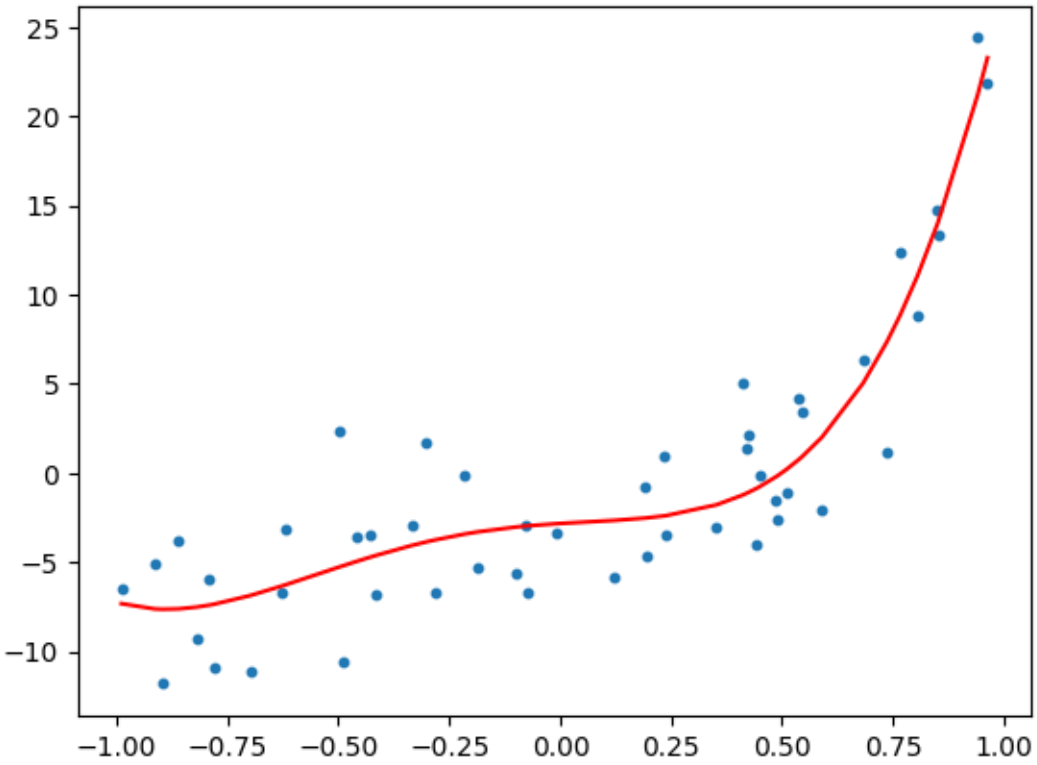
Now it is your turn! Following the same format as above, implement a 5-degree polynomial regression model on the training data and plot your results. Use your model to predict the output of the validation set and calculate the root mean square error. Report and plot the results.

Grading policy:

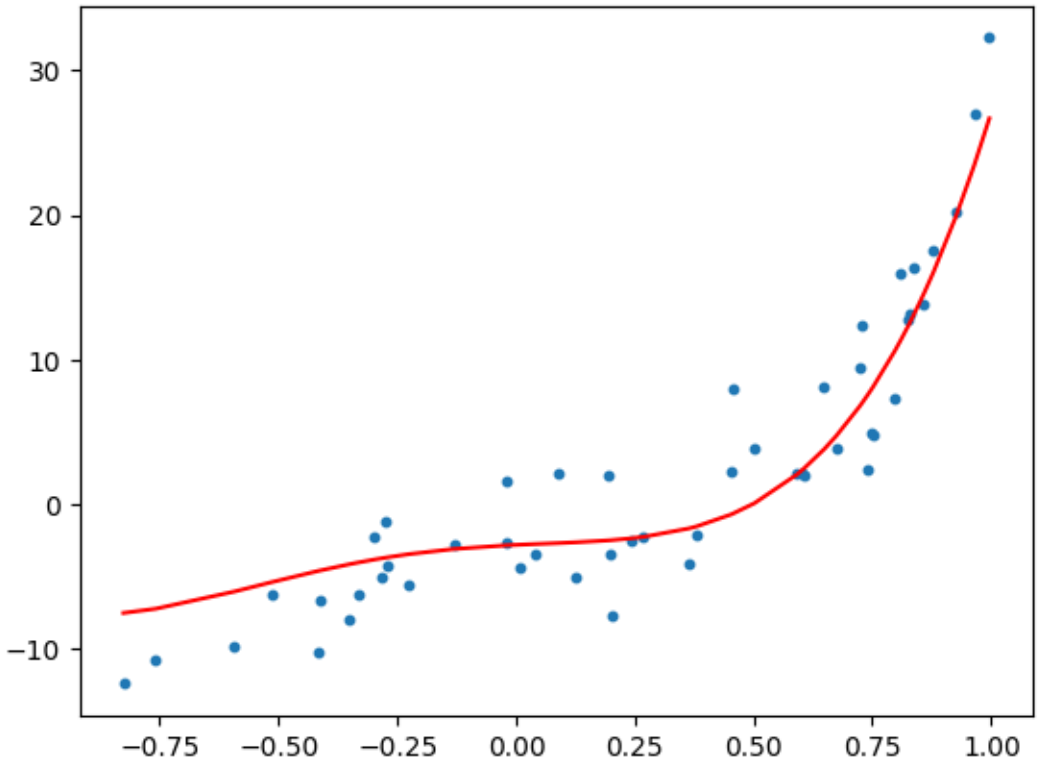
- Q1.1 [4pts]
  - Fit a 5-degree polynomial using scikit-learn [1pts]
  - Use model to predict output of validation set [1pts]
  - Calculate and report the MSE [1pt]
  - Plot curves on the training set and the validation set [1pt]
- Q1.2 [1pts]
- Q1.3 [4pts]
  - Fit a 10-degree polynomial using scikit-learn [1pts]
  - Use model to predict output of validation set [1pts]
  - Calculate and report the MSE [1pts]
  - Plot curves on the training set the validation set [1pt]
- Q1.4 [1pts]

1.4 Q1.1

[ ]:
```
### YOUR CODE HERE - Fit a 5-degree polynomial using scikit-learn
# Create polynomial features with degree 5
polynomial_features = PolynomialFeatures(degree=5)
x_poly = polynomial_features.fit_transform(train_x)

# Apply linear regression
model = LinearRegression()
model.fit(x_poly, train_y)
y_poly_pred = model.predict(x_poly)

### YOUR CODE HERE - Plot your the curve on the training data set
# Plot the training data set
plt.scatter(train_x, train_y, s=10)
plt.plot(train_x, y_poly_pred, color='r')
plt.show()

### YOUR CODE HERE - Use model to predict output of validation set
valid_y_poly_pred = model.predict(polynomial_features.fit_transform(valid_x))

### YOUR CODE HERE - Calculate the MSE. Report and plot the curve on the␣
 ↪validation set.
# Calculate mean squared error
mse = mean_squared_error(valid_y, valid_y_poly_pred)
print("Mean Squared Error: {}".format(mse))

# Plot the prediction results on the validation set
plt.scatter(valid_x, valid_y, s=10)
plt.plot(valid_x, valid_y_poly_pred, color='r')
plt.show()
```




```
Mean Squared Error: 10.363005107697845
```



1.5 Q1.2 Did the mean squared error go up or down as compared to the 2-degree polynomial curve? Why do you think this is the case?

——— **ANSWER HERE** ———

The MSE of the 5-degree polynomial curve is 10.363005107697845.

And the MSE of the 2-degree polynomial curve is 20.485214511024232.

The the mean squared error goes down.

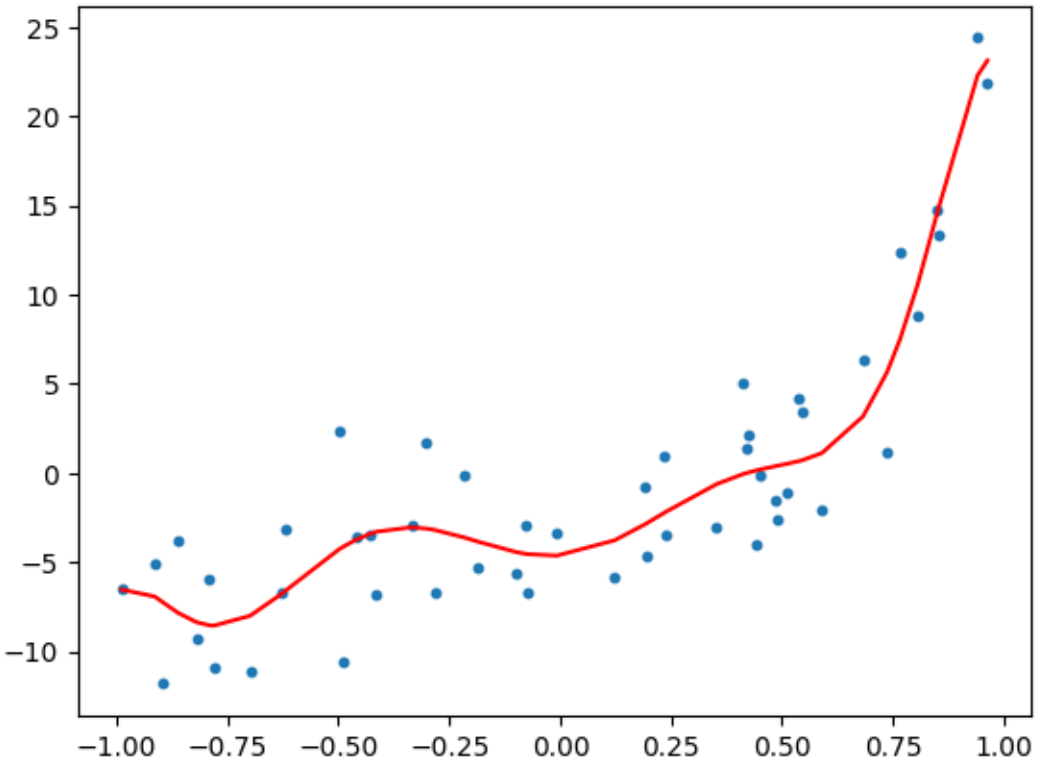
The probably reason of this in the case is that 2-degree polynomial curve is somehow underfit the data, so it has a worse effect on the validation dataset. So 5-degree polynomial curve fits the data better.

1.6 Q1.3

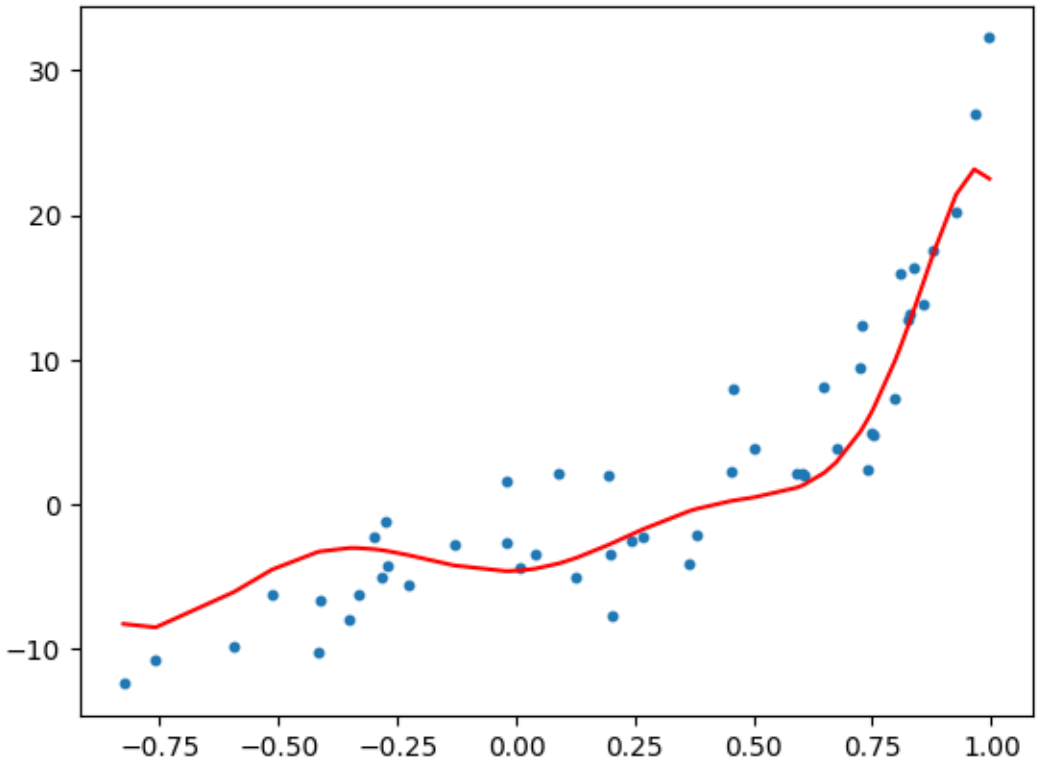
Now repeat the above for a 10-degree polynomial regression model.

[ ]:
```
### YOUR CODE HERE - Fit a 10-degree polynomial using scikit-learn
# Create polynomial features with degree 10
polynomial_features = PolynomialFeatures(degree=10)
x_poly = polynomial_features.fit_transform(train_x)
```

```
# Apply linear regression
model = LinearRegression()
model.fit(x_poly, train_y)
y_poly_pred = model.predict(x_poly)

### YOUR CODE HERE - Plot your the curve on the training data set
# Plot the training data set
plt.scatter(train_x, train_y, s=10)
plt.plot(train_x, y_poly_pred, color='r')
plt.show()

### YOUR CODE HERE - Use model to predict output of validation set
valid_y_poly_pred = model.predict(polynomial_features.fit_transform(valid_x))

### YOUR CODE HERE - Calculate the MSE. Report and plot the curve on the␣
 ↪validation set.
# Calculate mean squared error
mse = mean_squared_error(valid_y, valid_y_poly_pred)
print("Mean Squared Error: {}".format(mse))

# Plot the prediction results on the validation set
plt.scatter(valid_x, valid_y, s=10)
plt.plot(valid_x, valid_y_poly_pred, color='r')
plt.show()
```




```
Mean Squared Error: 13.115190778263702
```



1.7 Q1.4 How does the mean square error compare to the polynomial regression with degree 5? Why do you think this is the case?

——— **ANSWER HERE** ———

The MSE of the 10-degree polynomial curve is 13.115190778263702.

And the MSE of the 5-degree polynomial curve is 10.363005107697845.

The the mean squared error goes up.

The probably reason of this in the case is that 10-degree polynomial curve is somehow overfit on the training data, so it has a worse effect on the validation dataset. So 5-degree polynomial curve fits the data better.

1.8 Question 2: Manual Implementation [10pts]

Now it's time to appreciate the hard work that open source developers have put, in order to allow you to implemenent machine learning models without doing any math! No more scikit-learn (or any other libraries like Tensorflow, Pytorch, etc) for the rest of this assignment!

Your first step is to fit a **10-degree polynomial** to the dataset we have been used above. Then using your results, calculate the mean squared error on both the training and validation set.

A reminder that in polynomial regression, we are looking for a solution for the equation:

$$Y(X) = W^\top * \phi(X),$$

where

$$\phi(X) = [1, X, X^2, X^3, ..., X^n]^\top.$$

Let $\phi(\mathbf{X}) = [\phi(X_1)^\top; \ldots; \phi(X_n)^\top]$ denote the data matrix after polynomail transformation and $\mathbf{Y}$ denote the target vector. Recall the the closed-form solution for linear regression is given by normal equation, which is:

$$W = (\phi(\mathbf{X})^T \phi(\mathbf{X}))^{-1} \phi(\mathbf{X})^\top \mathbf{Y}.$$

Make sure to review the slides, do some research, and/or ask for clarification if this doesn't make sense. You must understand the underlying math before being able to implement this properly.

Suggestion - Use the original pandas dataframes variables named train, valid, and test instead of the reshaped arrays that were used specifically for scikit-learn. It will make your computations cleaner and more inuitive.

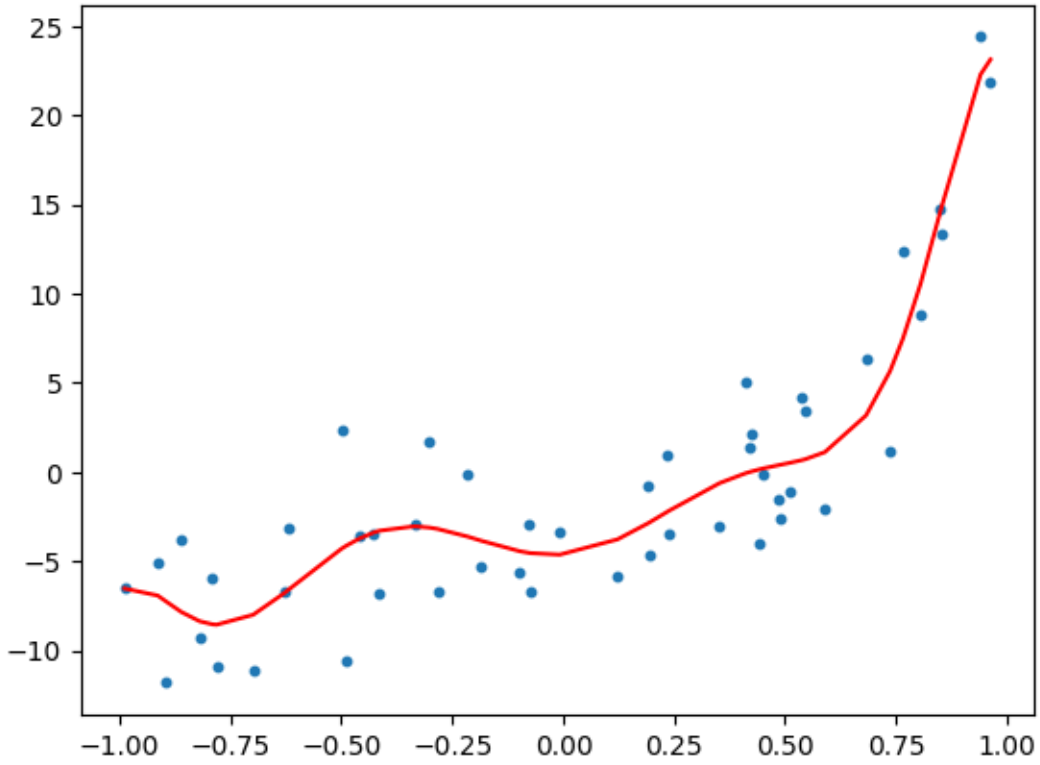
Grading policy: - Q2.1 [4pts] - Create the polynomial matrix $\phi(X)$ [1pts] - Find the weighted matrix W by normal equation [1pts] - Make predictions on the training set, calculate and report the mean squared error [1pts] - Make predictions on the validation set, calculate and report the mean squared error [1pts] - Q2.2 [6pts] - Implement gradient decent manually [6pts] - Correctness of the gradient [2pts] - Correctness of the update in each iteration [2pts] - Convergence of the algorithm [1pts] - Calculate and report the mean squared error on both training and validation set [1pts]

1.9 Q2.1

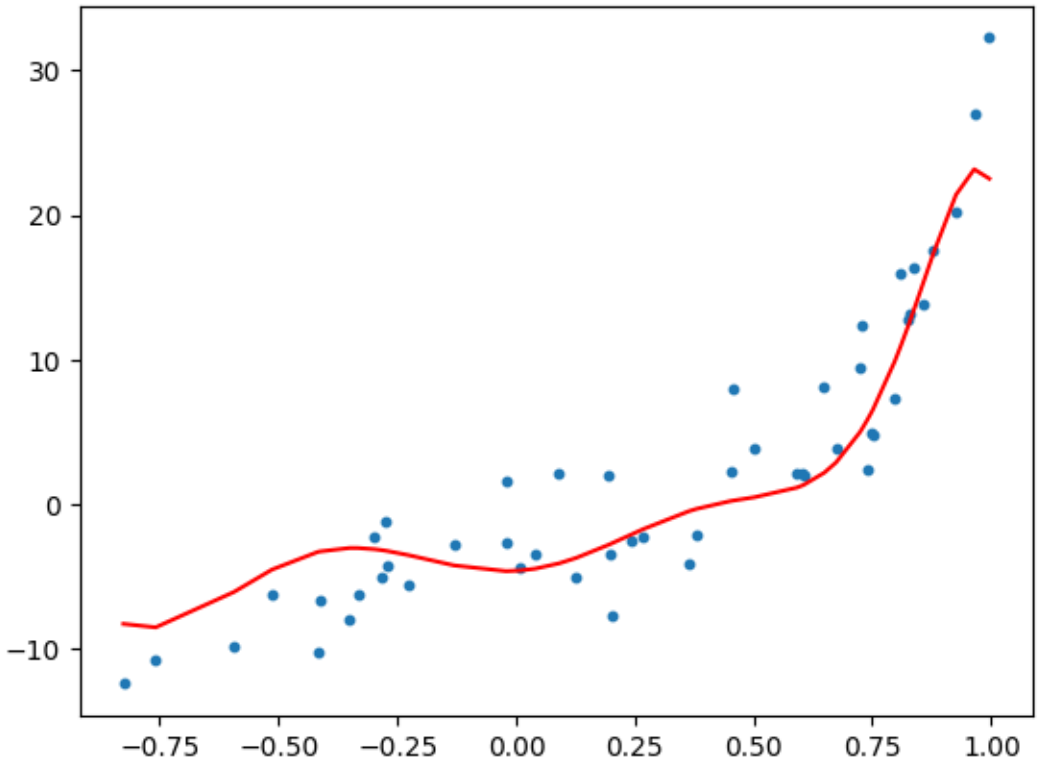
[ ]:
```
# Use the original pandas dataframes variables named train, valid, and test␣
↪instead of the reshaped arrays
train_x = np.array(train[0])
train_y = np.array(train[1])
valid_x = np.array(valid[0])
valid_y = np.array(valid[1])

### YOUR CODE HERE - Create the polynomial matrix \phi(X), which is a numpy␣
↪array

def generate_phi_x(X) -> np.array:
    degree = 10
    return np.array([X ** i for i in range(degree + 1)]).T

### YOUR CODE HERE - Find the weighted matrix W by normal equation
```

```python
def generate_W(phi_x: np.array, y: np.array) -> np.array:
    # a @ b  <=> np.dot(a, b)
    return np.linalg.inv(phi_x.T @ phi_x) @ phi_x.T @ y

### YOUR CODE HERE - Make predictions on the training set and calculate the␣
 ↪mean squared error.

phi_x_train = generate_phi_x(train_x)
W = generate_W(phi_x_train, train_y)

# phi_x_train : 49 * 11
# train_X : 49 * 1
# train_Y : 49 * 1
# W : 11 * 1
# so why W.T @ phi_x_train, instead of W.T @ phi_x_train.T ???

pred_y_train = W.T @ phi_x_train.T

mse_trian = 0.0
for i in range(len(train_x)):
    mse_trian += (pred_y_train[i] - train_y[i]) ** 2
mse_trian /= len(train_x)

# ==============================

print("Mean Squared Error (Training): {}".format(mse_trian))
plt.scatter(train_x, train_y, s=10)
plt.plot(train_x,pred_y_train,color='r')
plt.show()

### YOUR CODE HERE - Make predictions on the validation set and calculate the␣
 ↪mean squared error.
phi_x_valid = generate_phi_x(valid_x)
pred_y_valid = W.T @ phi_x_valid.T # same W

mse_valid = 0.0
for i in range(len(valid_x)):
    mse_valid += (pred_y_valid[i] - valid_y[i]) ** 2
mse_valid /= len(valid_x)

# ==============================

print("Mean Squared Error (Validation): {}".format(mse_valid))
plt.scatter(valid_x, valid_y, s=10)
plt.plot(valid_x,pred_y_valid,color='r')
plt.show()
```

```
Mean Squared Error (Training): 8.600664341757437
```




```
Mean Squared Error (Validation): 13.115190778000148
```



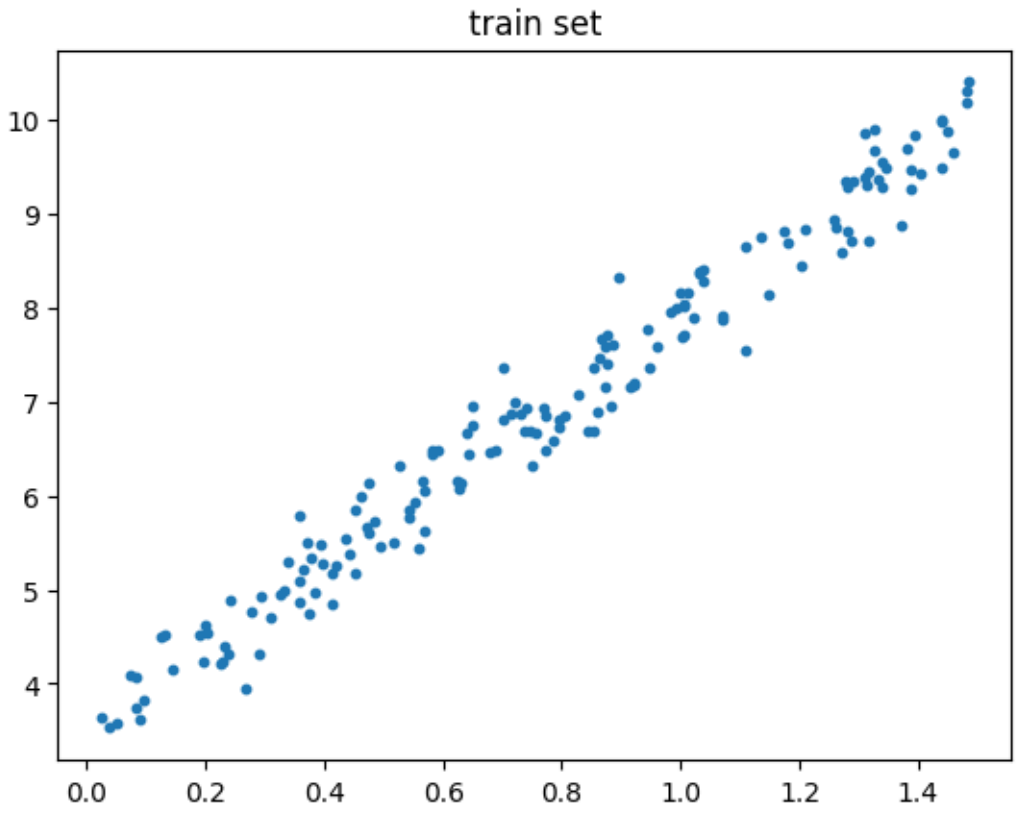
For the rest of the assignment, we will use the other dataset named **dataset2.csv**. First load the csv and split the model into train, valid, and test sets as shown earlier in the assignment.

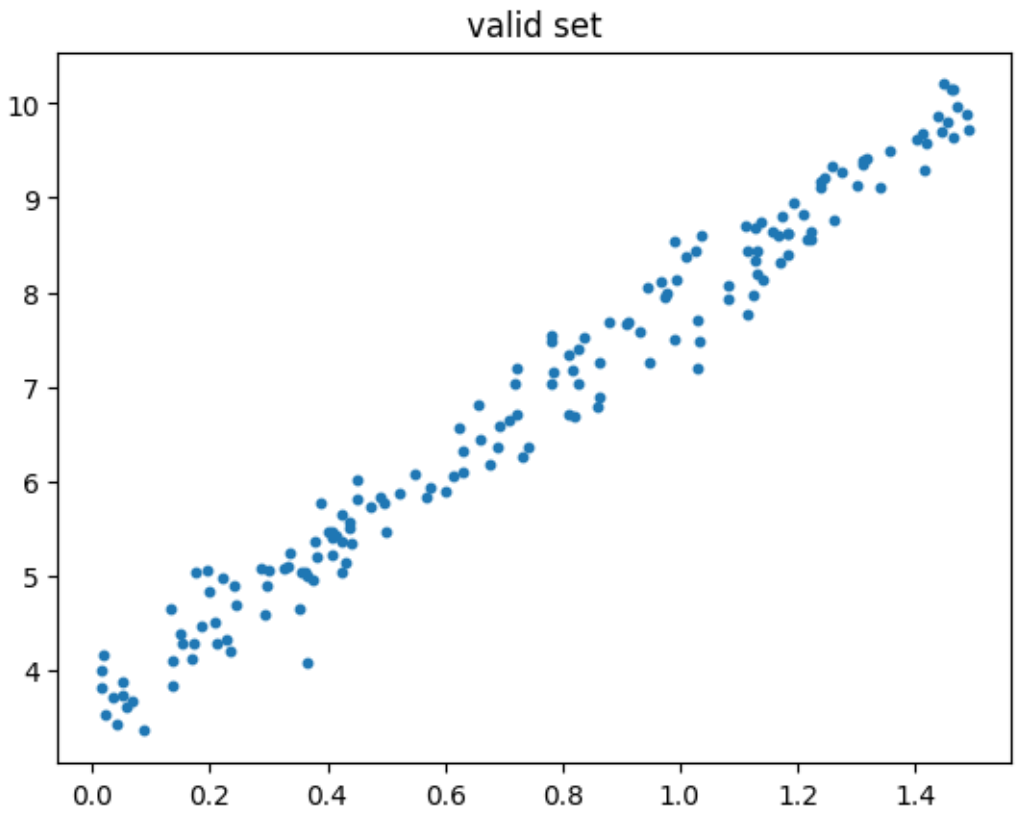
[ ]:
```
# Load dataset2.csv and split into 3 equal sets
data = pd.read_csv('./datasets/dataset2.csv', header=None)
data = data.iloc[:, :-1]
train, valid, test = np.split(data, [int(.33*len(data)), int(.66*len(data))])
# Sort the data in order for plotting purposes later
train.sort_values(by=[0], inplace=True)
valid.sort_values(by=[0], inplace=True)
test.sort_values(by=[0], inplace=True)

train_x = np.array(train[0])
train_y = np.array(train[1])
valid_x = np.array(valid[0])
valid_y = np.array(valid[1])
test_x = np.array(test[0])
test_y = np.array(test[1])
train_x = train_x.reshape(-1, 1)
train_y = train_y.reshape(-1, 1)
valid_x = valid_x.reshape(-1, 1)
valid_y = valid_y.reshape(-1, 1)
```

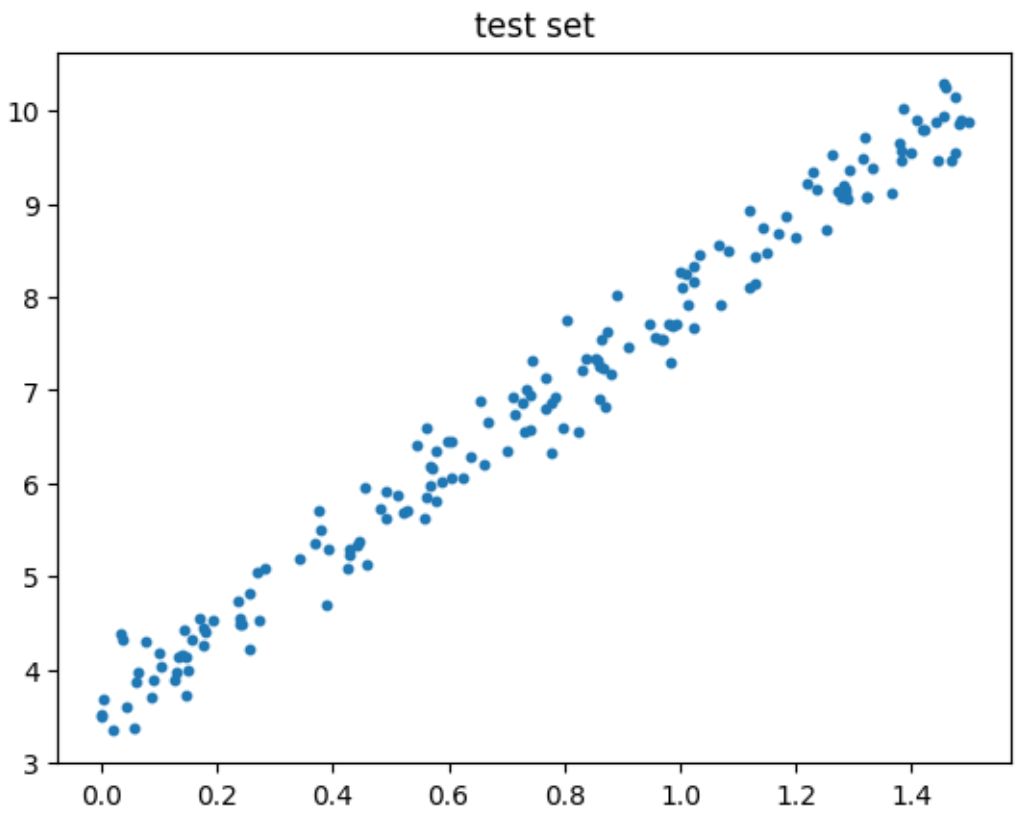
```
test_x = test_x.reshape(-1, 1)
test_y = test_y.reshape(-1, 1)
```

Plot the data below to see what it looks like

[ ]:
```
plt.title('train set')
plt.scatter(train_x, train_y, s=10)
plt.show()
plt.title('valid set')
plt.scatter(valid_x, valid_y, s=10)
plt.show()
plt.title('test set')
plt.scatter(test_x, test_y, s=10)
plt.show()
```




valid set
10
9
8
7
6
5
4
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4



1.10 Q2.2

If done properly, you should see that the points fall under a relatively straight line with minor deviations. Looks like a perfect example to implement a linear regression model using the **gradient descent** method ….. without the use of any machine learning libraries!

Since the data falls along a straight line, we can assume the solution follows the form:

$$y(x) = wx + b$$

A reminder that in gradient descent, we essentially want to iteratively get closer to the minimum of our objective function (the mean squared error), such that:

$$MSE(w_0) > MSE(w_1) > MSE(w_2) > ...$$

The algorithm is as follows:

1) Pick initial w_0 randomnly.

2) For $k = 1, 2..$ $\Rightarrow$ w_{k+1} = w_k - α $g(w_k)$ where $\alpha > 0$ is the learning rate and $g(w_k)$ is the gradient.

End when | w_{k+1} - w_k | $< \epsilon$

There are many resources online for gradient descent. You must understand the underlying math before being able to implement this properly.

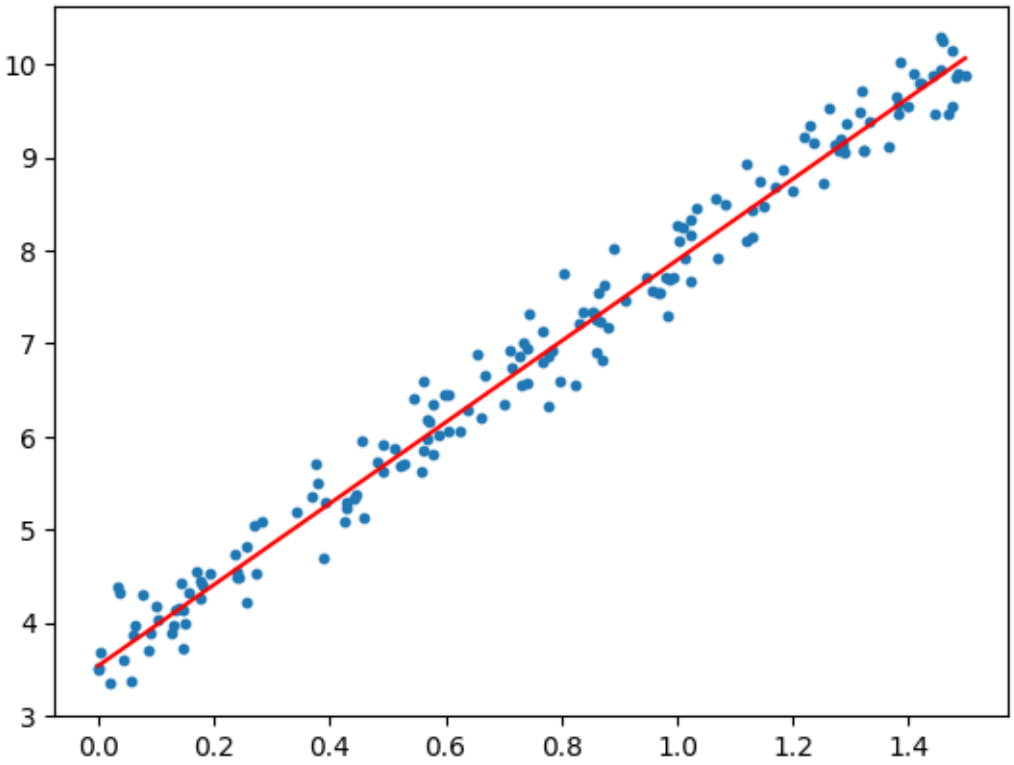
Now once you understand, it is time to implement the gradient descent below. You may set the learning rate to 1e-6 or whatever value you think is best. As usual, calculate the mean squared error and plot your results. This time, training should be done using the training and validation sets, while the final mean squared error should be computed using the testing set.

Feel Free to modify the existing code, or just ignore it and provide your own implementation!

[ ]:
```
### Implement gradient descent
w = 1  # initial w (slope)
b = 1  # initial b (intercept)
temp_w = 0
temp_b = 0
lr = 1e-6
epsilon = 1e-9
epsilon_valid = 1e-12
lr_decay = 0.9
epsilon_lr = 0.2

def get_mse(w,b,x,y):
    return np.sum(np.multiply((y - w * x - b),(y - w * x - b))) / len(x)

# ======== YOUR CODE HERE ========

def gradient_descent(w, b, lr, epsilon, epsilon_valid, lr_decay, epsilon_lr):
    iter = 0
    pre_loss = get_mse(w, b, valid_x, valid_y)
    while True:
        iter += 1

        # update w
        gradient_w = np.sum(np.multiply((train_y - w * train_x - b), train_x))␣
 ↪* (-2) / len(train_x)
        temp_w = w - lr * gradient_w

        # update b
        gradient_b = np.sum(train_y - w * train_x - b) * (-2) / len(train_x)
        temp_b = b - lr * gradient_b

        w, temp_w = temp_w, w
        b, temp_b = temp_b, b

        # validation loss
        loss = get_mse(w, b, valid_x, valid_y)
```

```
        # check whether to stop
        if abs(w - temp_w) < epsilon and abs(b - temp_b) < epsilon and abs(loss␣
↪- pre_loss) < epsilon_valid:
            break

        # validation set is used to control the learning rate
        if abs(loss - pre_loss) > epsilon_lr:
            lr *= lr_decay

        pre_loss = loss

        # if iter % 10000000 == 0:
            # print(f"w = {w}, b = {b}, current_mse = {get_mse(w, b, train_x,␣
↪train_y)}")

    # print('iter = ', iter)
    return w, b

final_w, final_b = gradient_descent(w, b, lr, epsilon, epsilon_valid, lr_decay,␣
↪epsilon_lr)

# ==============================

### Calculate the mean squared error on both training and validation set and␣
↪plot the results.
print("Mean Squared Error (Training): {}".
↪format(get_mse(final_w,final_b,train_x,train_y)))
print("Mean Squared Error (Testing): {}".
↪format(get_mse(final_w,final_b,test_x,test_y)))
pred_y_test = final_w * test_x + final_b
plt.scatter(test_x, test_y, s=10)
plt.plot(test_x,pred_y_test,color='r')
plt.show()
```

```
Mean Squared Error (Training): 0.09349003119972489
Mean Squared Error (Testing): 0.07559024924738163
```


10
9
8
7
6
5
4
3
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4