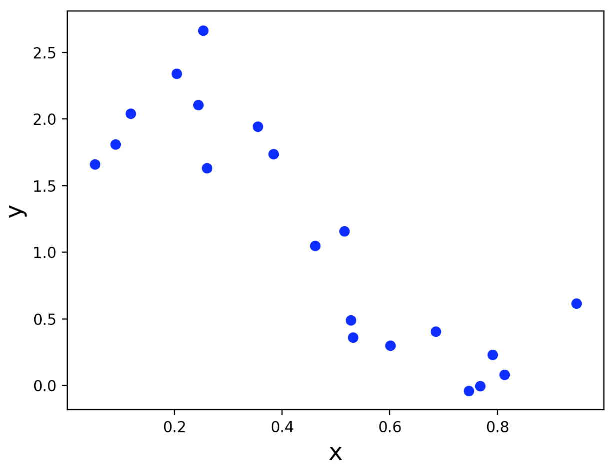
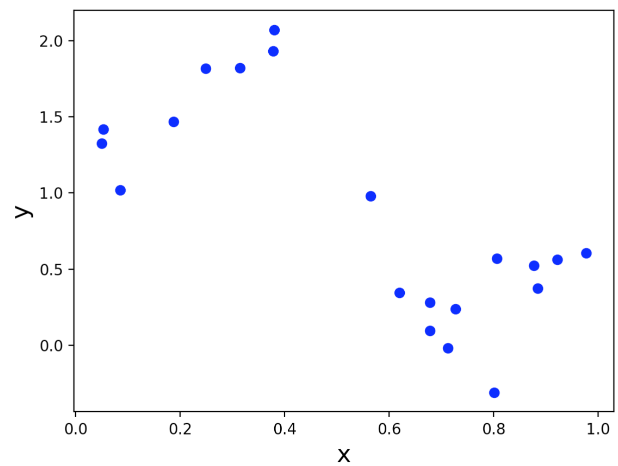
Problem 4:

a.)

Training Data



Test Data



Linear regression will fit the training data well since the data points seem to have an almost linear trend. It will also do decent on the test data set; however since the data point are slightly spread out towards the lower and higher x's there might be a slightly larger error.

b.) Completed this portion with the following code:

```
# part g: modify to create matrix for polynomial mode
X = np.append(np.ones((n, d)), X, axis=1)
```

c.) Completed this portion by using the following code:

```
y = np.dot(X, self.coef_)
```

d.)

(i) Completed this portion by using the following code:

```
# part d: compute J(theta)
n, d = X.shape
cost = 0
temp = np.dot(X, self.coef_) - y
temp *= temp
cost = temp.sum()
```

(ii) Completed this portion with the following code:

```
# hint: you cannot use self.predict(...) to make the predictions
y_pred = np.dot(self.coef_, X.T)
self.coef_ = self.coef_ - (2*eta*(np.dot(X.T, y_pred-y)))
err_list[t] = np.sum(np.power(y - y_pred, 2)) / float(n)
```

(iii)

Learning Rate	Number of iterations	Value of Objective Function
0.0001	10000	4.0863970367957645
0.001	7021	3.9125764057919437
0.01	765	3.912576405791486
0.0407	10000	2.710916520014198e+39

The first 3 end up converging to about the same value for the objective function, but 0.01 converges much faster than 0.0001. So, 0.01 was a good size in that it was small enough to coverage, but big enough to cause only a few iterations. The step size of 0.0407 was too large, and made it so the algorithm did not converge, and the value of the objective function was very large.

e.)

```
self.coef_ = np.dot(np.dot(np.linalg.pinv(np.dot(X.T, X)), X.T), y)
```

Closed Form Solution:

w = [2.44640709 -2.81635359]
cost = 3.9125764057914645
time = 0.003287792205810547 seconds

Gradient Descent:

w = [2.44640703 -2.81635347] with gradient descent, learning rate = 0.01.
cost = 3.912576405791486
time = 0.012884140014648438 seconds

Both solutions and costs are very similar; however gradient descent takes a lot longer to converge than the closed for solution.

f.)

time = 0.029319286346435547 seconds
Number of iterations: 1679
Value of objective function: 3.912576405792008
w = [2.44640678 -2.81635296]

g.)

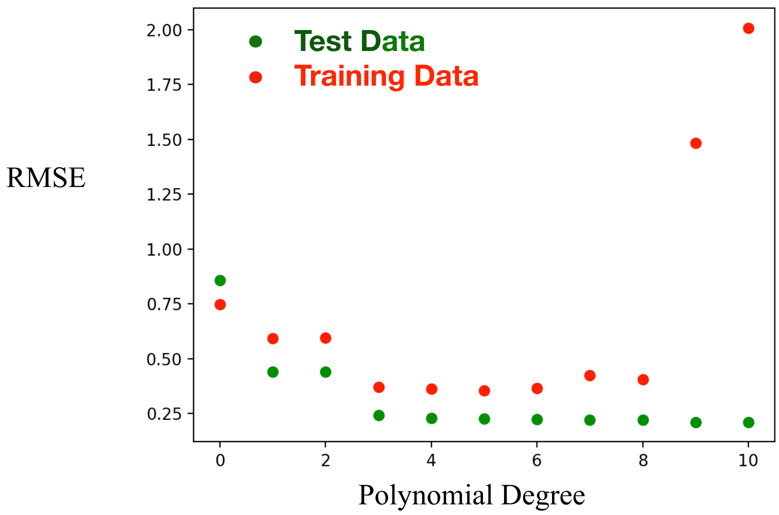
```
# part g: modify to create matrix for polynomia
n = len(X)
X = X.reshape(n, 1)
Phi = np.ones((n, 1))
m = self.m_
for i in range (1, m+1):
    Phi = np.append(Phi, X**i, axis=1)
```

h.)

```
# part h: compute RMSE
n, d = X.shape
error = 0
cost = self.cost(X, y)
error = (cost/n)**(1/2)
```

We prefer RMS, because it accounts for how many data points are in the set and normalizes based on that. J(w) will increase as the number of points in the data set increase even if the fit is better.

i.)



I would say a polynomial of degree 5 best fits the data, because it has a low training and test root mean square error. There is evidence of under-fitting with degrees of less than 4, because both training and test error are high. There is evidence of overfitting in degrees 9 and 10, because training root mean square error is very low and testing root mean square error is very high.