Karim_ManishaMini_Project1CAP_6673_004

Problem 1:

a.

A system has generated outputs marked by vector 'd' in response to inputs which are listed in vector 'x':

d = [6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7972, 11.6371] ;

x = [1 , 1.7143, 2.4286 , 3.1429, 3.8571, 4.5714, 5.4857, 6] ;

Use the theory of regression to fit a line to this data. Measure the cost function defined as the mean of squared errors. Plot your data points and the line that models the system's function.

b.

Use the same data but this time fit a second order polynomial to these data points. What's the value of your cost function? Plot the second order curve.

c.

Increase the order of the polynomial to 6 and fit the curve. How much error do you measure this time? Plot the 6th order curve.

d.

Remove one data point (d: 12.7772, x: 5.2857) from your set. Once again fit the 6th order polynomial. After finding the polynomial bring that data point back and once again measure the cost function for all data points. How much the value of the cost function changed compared to the 6th order polynomial where all data points were used for curve fitting? Is the 6th order polynomial a case of over fitting? Plot the new 6th order curve.

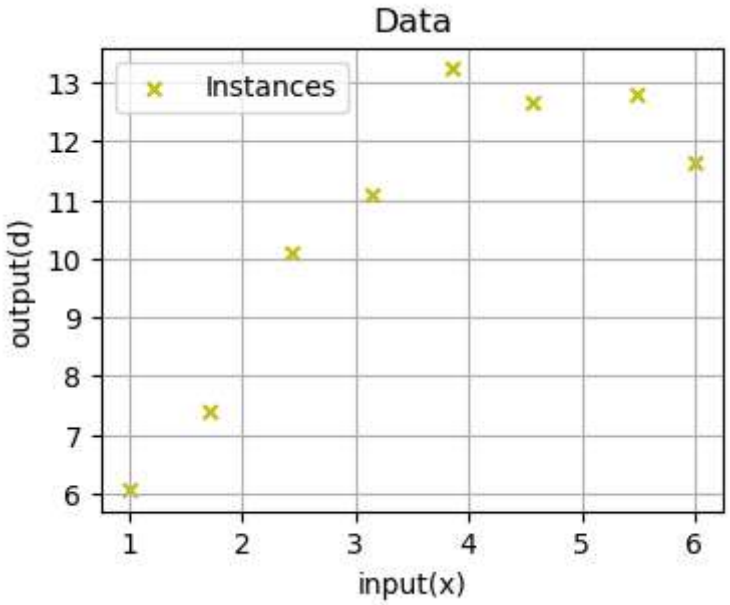
e.

Plot the value of the cost function as a function of the polynomial order (from 1 to 10) using all data points. Based on this curve, which order is suitable for this dataset to avoid over or under fitting?

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
d = np.array([6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7972, 11.6371 ])
x = np.array([1 , 1.7143, 2.4286 , 3.1429, 3.8571, 4.5714, 5.4857, 6 ])
```

In [3]:
```
plt.figure(figsize = (4,3))

plt.scatter(x, d, color = 'y', marker = 'x', label = 'Instances', s = 25)
plt.xlabel('input(x)')
plt.ylabel('output(d)')
plt.title('Data')

plt.grid()
plt.legend()
plt.show()
```



Solution Description

Regression Line

A regression line is a straight line that shows the relationship between changes in an explanatory variable (x) and response variable (y).

The problems described in **a., b., and c.** required fitting a regression line of first, second, and sixth order, respectively. The most effective method to resolve this is to develop a function for the regression line plotting.

Algorithm for the function:

First, let's look at the regression line y = mx + c. In this case, m and c are coefficients.

Let's use Ax = y as an example to determine the coefficients.

Here, A is the input data x is the co-efficients of the regression line y is the desired output.

The formula to solve this:

```
x = (A^T . A)^-1 . b
```

To obtain the regression line, enter the acquired coefficients into the line's equation.

We use the mean squared of errors to construct the cost function.

Underfitting and overfitting:

These two factors are mostly responsible for machine learning algorithms' subpar performance. *Underfitting* occurs when a model is too simplistic to accurately represent the intricacies of the data. In contrast, when a machine learning model fails to produce reliable predictions on test data, it is considered *overfitted*.

The easiest way to tell if a model is overfitted or underfitted is to add or delete an instance from the data. The model is overfitted if the removal of a datapoint results in a significant change in the cost function.

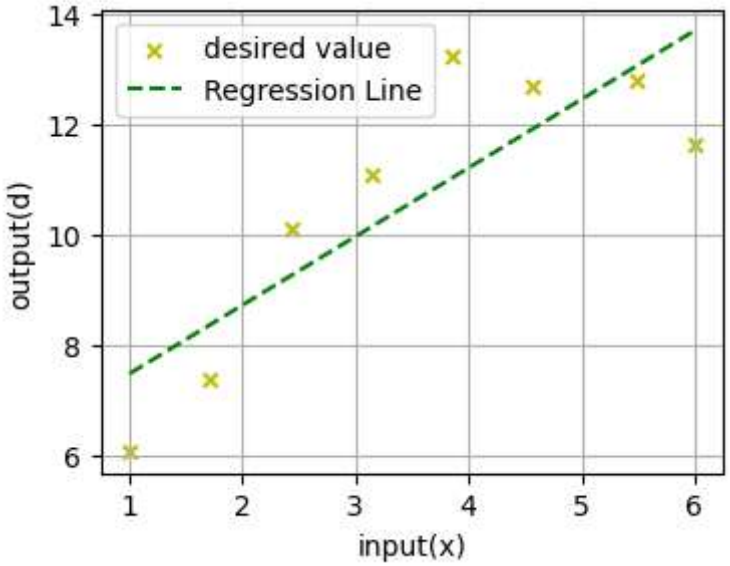
> Function for regression line

In [4]:
```
def beta(A,d):
    A = A
    a1 = np.linalg.inv(A.T @ A)
    a_dagger = a1 @ A.T
    global var
    var = a_dagger @ d



def plot_regression_line(x, y, b, eqn):

    # Predicted output value
    global d_pred
    d_pred = eqn

    plt.figure(figsize = (4,3))
    plt.xlabel('input(x)')
    plt.ylabel('output(d)')

    # Plotting the actual points as scatter plot
    plt.scatter(x, y, color = "y", label = 'desired value', marker = "x", s = 25)

    # Plotting the regression line
    plt.plot(x, d_pred, color = "g", label = "Regression Line", linestyle="--")

    plt.grid()
    plt.legend()
    plt.show()

    global mse
    mse = np.square(d_pred - y)
    mse = np.sum(mse)/len(x)
```

```
    print('Cost Function defined as mse: ', mse)
```

> **(a)**

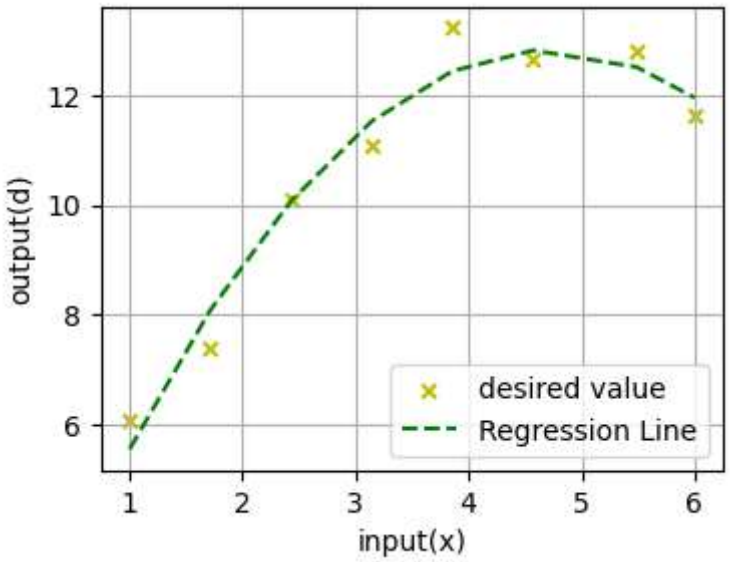
In [5]:
```
A = np.c_[np.ones(len(x)), x]
beta(A,d)

eqn = var[0] + var[1]*x
plot_regression_line(x, d, var, eqn)

mse_1st_order = mse
```




```
Cost Function defined as mse:  1.7872722572247457
```

> **(b)**

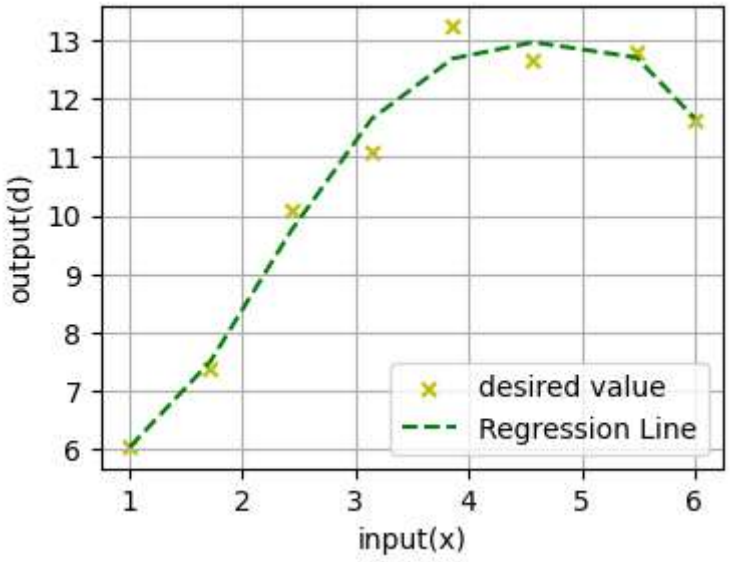
In [6]:
```
A = np.c_[np.ones(len(x)), x, x**2]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2
plot_regression_line(x, d, var, eqn)

mse_2nd_order = mse
```




```
Cost Function defined as mse:  0.2208480705312497
```

(c)

In [7]:
```
A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
plot_regression_line(x, d, var, eqn)

mse_6th_order = mse
```

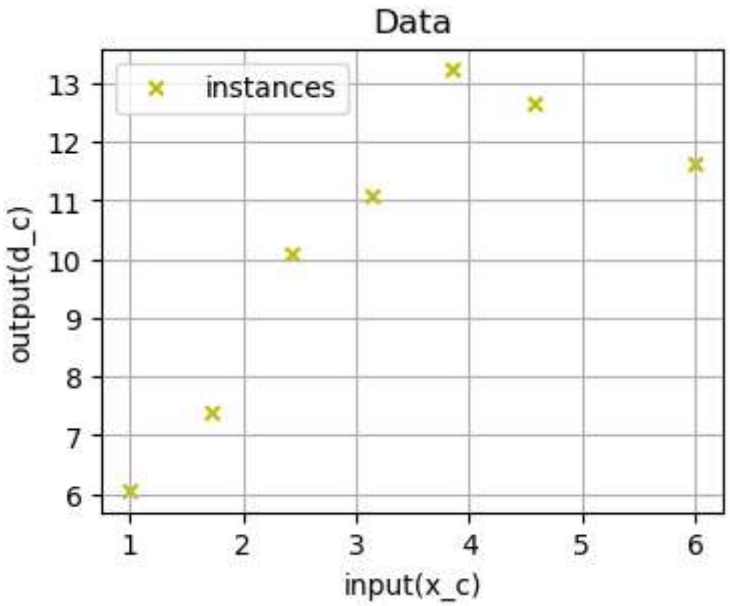



```
Cost Function defined as mse:  0.10800458701017875
```

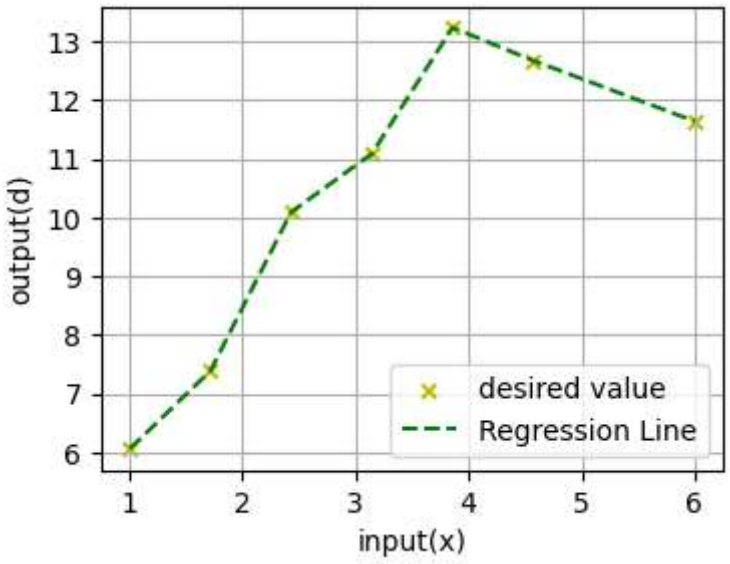
(d)

In [8]:
```
d_c = np.array([6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 11.6371 ])
x_c = np.array([1 , 1.7143, 2.4286 , 3.1429, 3.8571, 4.5714,  6 ])

plt.figure(figsize = (4,3))
```

```
plt.scatter(x_c, d_c, color = 'y', label = 'instances', marker = 'x', s = 25)
plt.xlabel('input(x_c)')
plt.ylabel('output(d_c)')
plt.title('Data')

plt.grid()
plt.legend()
plt.show()
```



In [9]:
```
A = np.c_[np.ones(len(x_c)), x_c, x_c**2, x_c**3, x_c**4, x_c**5, x_c**6]
beta(A,d_c)

eqn = var[0] + var[1]*x_c + var[2]*x_c**2 + var[3]*x_c**3 + var[4]*x_c**4 + var[5] * x
plot_regression_line(x_c, d_c, var, eqn)

mse_6th_order_dropped = mse
```




```
Cost Function defined as mse:  5.722935015206403e-13
```

In [10]:
```
print('The cost function for all instances: ', mse_6th_order)
print('The cost function after dropping an instance: ', mse_6th_order_dropped)
```

```
The cost function for all instances:  0.10800458701017875
The cost function after dropping an instance:  5.722935015206403e-13
```

In [11]:
```
Change_Cost_Function = mse_6th_order - mse_6th_order_dropped
Change_Cost_Function
```

Out[11]:
```
0.10800458700960645
```

The elimination of an instance from the data results in a significant alteration to the Cost Function. Thus, the model is being overfitted by the sixth order polynomial.

> **e**

As previously indicated, the simplest method to assess whether the model is overfitted or underfitted is to observe whether the cost function significantly alters when an instance of the data is eliminated.

In [12]:
```
def beta(A,d):

    A = A
    a1 = np.linalg.inv(A.T @ A)
    a_dagger = a1 @ A.T
    global var
    var = a_dagger @ d


def cost_function(x, y, b, eqn):

    # Predicted output value
    global d_pred
    d_pred = eqn

    global mse
    mse = np.square(d_pred - y)
    mse = np.sum(mse)/len(x)
```

Calculate cost function with all instances

In [13]:
```
#1st Order has been calculated in 1a
# 2nd Order has been calculted in 1b

#3rd Oder
A = np.c_[np.ones(len(x)), x, x**2, x**3]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3
cost_function(x, d, var, eqn)

mse_3rd_order = mse

#4th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4]
```

```
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4
cost_function(x, d, var, eqn)

mse_4th_order = mse

#5th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5
cost_function(x, d, var, eqn)

mse_5th_order = mse

#6th Order has been calculated in 6c

#7th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
cost_function(x, d, var, eqn)

mse_7th_order = mse

#8th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7, x**8]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
cost_function(x, d, var, eqn)

mse_8th_order = mse

#9th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7, x**8, x**9]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
+ var[9] * x**9
cost_function(x, d, var, eqn)

mse_9th_order = mse

#10th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7, x**8, x**9, x**10]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
+ var[9] * x**9 + var[10] * x**10
cost_function(x, d, var, eqn)

mse_10th_order = mse
```

Calculate Cost Function after dropping an instance (same as problem 1d)

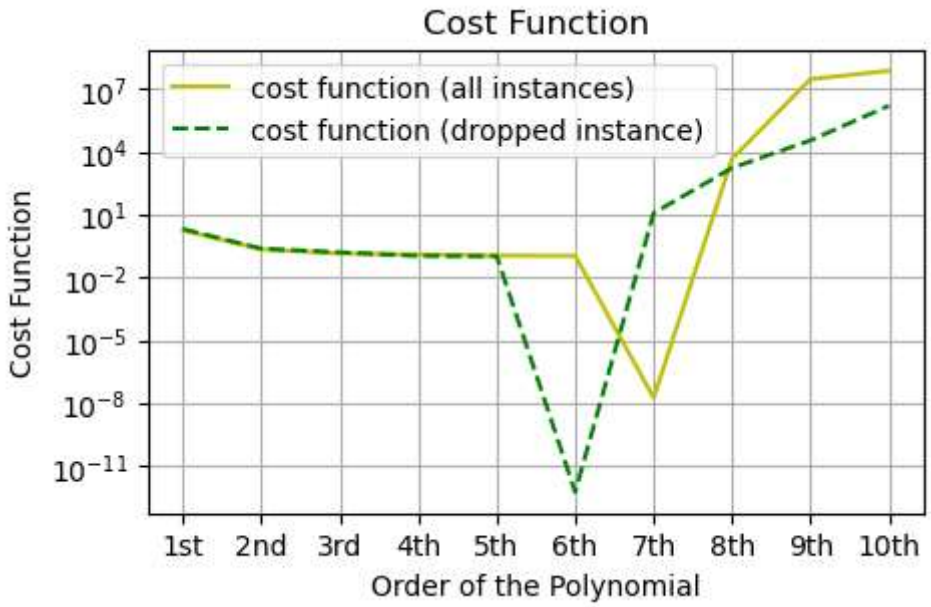
In [14]:
```
x = x_c
d = d_c

#1st Order

A = np.c_[np.ones(len(x)), x]
beta(A,d)

eqn = var[0] + var[1]*x
cost_function(x, d, var, eqn)

mse_1st_order_dropped = mse

#2nd Order

A = np.c_[np.ones(len(x)), x, x**2]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2
cost_function(x, d, var, eqn)

mse_2nd_order_dropped = mse

#3rd Order

A = np.c_[np.ones(len(x)), x, x**2, x**3]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]* x**3
cost_function(x, d, var, eqn)

mse_3rd_order_dropped = mse


#4th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4
cost_function(x, d, var, eqn)

mse_4th_order_dropped = mse

#5th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5
cost_function(x, d, var, eqn)

mse_5th_order_dropped = mse

#6th Order has been calculated in 1d
```

```
#7th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
cost_function(x, d, var, eqn)

mse_7th_order_dropped = mse

#8th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7, x**8]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
cost_function(x, d, var, eqn)

mse_8th_order_dropped = mse

#9th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7, x**8, x**9]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
+ var[9] * x**9
cost_function(x, d, var, eqn)

mse_9th_order_dropped = mse

#10th Order

A = np.c_[np.ones(len(x)), x, x**2, x**3, x**4, x**5, x**6, x**7, x**8, x**9, x**10]
beta(A,d)

eqn = var[0] + var[1]*x + var[2]*x**2 + var[3]*x**3 + + var[4]*x**4 + var[5] * x**5 +
+ var[9] * x**9 + var[10] * x**10
cost_function(x, d, var, eqn)

mse_10th_order_dropped = mse
```

In [15]:
```
cost_function_mse = np.array([mse_1st_order, mse_2nd_order, mse_3rd_order, mse_4th_orc
                              mse_6th_order, mse_7th_order, mse_8th_order, mse_9th_orc

cost_function_mse_dropped = np.array([mse_1st_order_dropped, mse_2nd_order_dropped, ms
                               mse_5th_order_dropped, mse_6th_order_dropped, mse_7th_or
                               mse_9th_order_dropped, mse_10th_order_dropped])
polynomial_order = np.array(['1st', '2nd', '3rd', '4th', '5th', '6th', '7th', '8th', '
```

In [16]:
```
plt.figure(figsize = (5,3))

plt.semilogy(polynomial_order, cost_function_mse, color = 'y', label = 'cost function
plt.semilogy(polynomial_order, cost_function_mse_dropped, color = 'g', label = 'cost f

plt.xlabel('Order of the Polynomial')
plt.ylabel('Cost Function')
plt.title('Cost Function')
```

```
plt.legend()
plt.grid()
plt.show()
```



Given that the model in problem 1d is overfit at the sixth order polynomial, it follows that any order higher will also overfit the data. That notion is also supported by the plot above. Out of the first five polynomial orders, the first order regression line has the largest cost function. The cost function is fairly similar in the next four polynomial orders, decreasing slightly with each order increase. In this case, all of these will work incredibly well, but the 5th order polynomial will suit the data the best.

Problem 2

Consider a 2-dimensional classification dataset with the given desired values for each point.

x1 = [0.5, 0.8, 0.9, 1.0, 1.1, 2.0, 2.2, 2.5, 2.8, 3.0];

x2 = [0.5, 0.2, 0.9, 0.8, 0.3, 2.5, 3.5, 1.8, 2.1, 3.2];

d = [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0];

Here for (x1=0.5, x2=0.5) the desired value is 0.0 and for (x1=3.0, x2=3.2) the desired value is 1.0.

a.

Use the logistic regression algorithm to design a supervised classifier that can perfectly separate these two sets. Plot these data points and your decision line.

b.

Increase the order of the decision making curve to 2 and once again use the logistic regression to design the classifier. How do you compare these two classifiers?

In [17]:
```
x1 = np.array([0.5, 0.8, 0.9, 1.0, 1.1, 2.0, 2.2, 2.5, 2.8, 3.0])
x2 = np.array([0.5, 0.2, 0.9, 0.8, 0.3, 2.5, 3.5, 1.8, 2.1, 3.2])
d =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0])
```

In [18]:
```
X = np.stack((np.ones(len(x1)),x1, x2), axis=0)
X
```

Out[18]:
```
array([[1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. , 1. ],
       [0.5, 0.8, 0.9, 1. , 1.1, 2. , 2.2, 2.5, 2.8, 3. ],
       [0.5, 0.2, 0.9, 0.8, 0.3, 2.5, 3.5, 1.8, 2.1, 3.2]])
```

Solution Description

Logistic regression is a straightforward and more efficient approach to binary and linear classification issues. It utilises the logit function to perform prediction.

The logit function:

f(x) = 1/ (1+ e^-x)

If f(x)> 0.5, the predicted value is class 1 else it is class 0.

For instances [x1,x2,.....xn] within sensor X, the output y can be written as:

y = 1/ (1 + e^-(w0+w1x))

Here, **Gradient Descent** can be used to determine the value of of w0 and w1. A local minimum of a given function can be found using gradient descent, an iterative first-order optimization technique. This approach is frequently used to minimize a cost function in deep learning and machine learning.

The boundary or separator line, w0+w1x = 0, can be plotted using the w0 and w1 that were acquired by gradient descent. Anything categorized as class 0 is below this line, and anything categorized as class 1 is above.

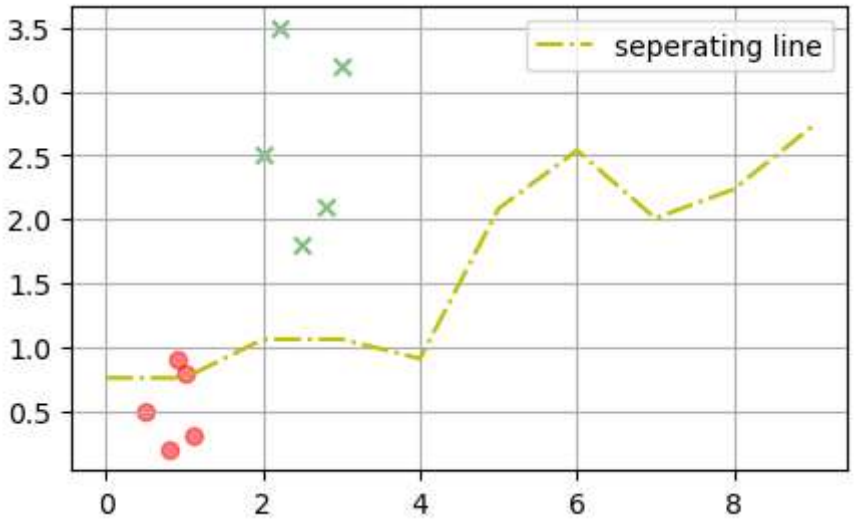
> a

In [19]:
```
def sigmoid(x):
        return (1/ (1 + np.exp(-x)))

class Logistic_Regression():

    def __init__(self, lr , n_iters ):

        self.lr = lr
        self.n_iters = n_iters


    def gradient_descent(self, X, d):
```

```
        w = np.ones([X.shape[0],1])
        for j in range(self.n_iters):
            w = w + self.lr * (np.sum(np.dot (X,( d - sigmoid((np.dot(X.T, w))))))) /2

        global coef
        coef = w
```

In [20]:
```
clf = Logistic_Regression(0.01, 10000)
clf.gradient_descent(X, d)
```

In [21]:
```
seperating_line = coef[0] + coef[1] * x1 + coef[2] *x2

plt.figure(figsize = (5,3))

for i in range(len(d)):
    if d[i] == 1:
        plt.scatter(x = x1[i], y = x2[i], marker='x', c='g', alpha = 0.5)
    else:
        plt.scatter(x = x1[i], y = x2[i], marker='o', c='r', alpha = 0.5)


plt.plot(seperating_line, c = 'y', label = 'seperating line', linestyle = '-.')

plt.grid()
plt.legend()
plt.show()
```



2b

For a decision making curve of order 2, the equation for the seperating line is: **w0 + w1 x1 + w2 x2 + w3 x1^2 + w4 x2^2**

In [22]:
```
x3 = x1 ** 2
x4 = x2 ** 2
```

In [23]:
```
X2 = np.stack((np.ones(len(x1)),x1, x2, x3, x4), axis=0)
X2
```

Out[23]:
```
array([[ 1.  ,  1.  ,  1.  ,  1.  ,  1.  ,  1.  ,  1.  ,  1.  ,  1.  ,
         1.  ],
       [ 0.5 ,  0.8 ,  0.9 ,  1.  ,  1.1 ,  2.  ,  2.2 ,  2.5 ,  2.8 ,
         3.  ],
       [ 0.5 ,  0.2 ,  0.9 ,  0.8 ,  0.3 ,  2.5 ,  3.5 ,  1.8 ,  2.1 ,
         3.2 ],
       [ 0.25,  0.64,  0.81,  1.  ,  1.21,  4.  ,  4.84,  6.25,  7.84,
         9.  ],
       [ 0.25,  0.04,  0.81,  0.64,  0.09,  6.25, 12.25,  3.24,  4.41,
        10.24]])
```

In [24]:
```
w = np.ones([5,1])
```

In [25]:
```
def sigmoid(x):
        return (1/ (1 + np.exp(-x)))

class Logistic_Regression():

    def __init__(self, lr , n_iters ):

        self.lr = lr
        self.n_iters = n_iters


    def gradient_descent(self, X, d):

        w = np.ones([X.shape[0],1])
        for j in range(self.n_iters):
            w = w + self.lr * (np.sum(np.dot (X,( d - sigmoid((np.dot(X.T, w))))))) /2

        global coef
        coef = w
```

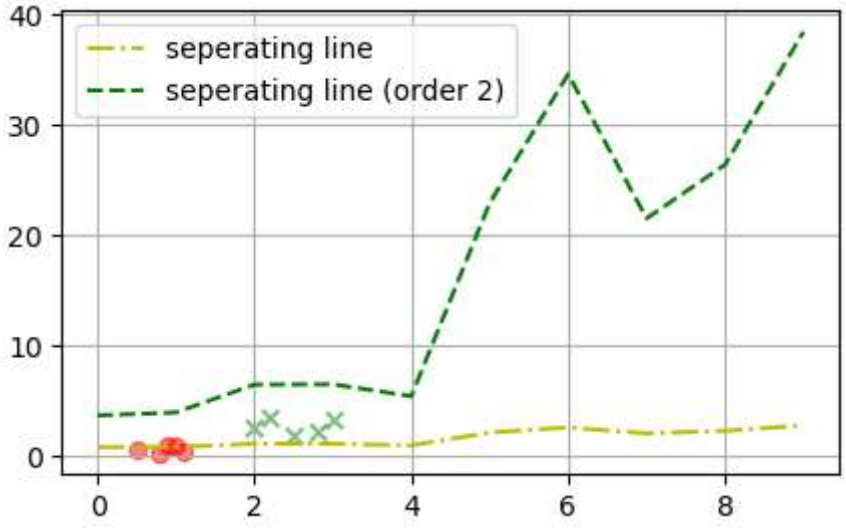
In [26]:
```
X = X2
y = d

clf = Logistic_Regression(0.01, 10000)
clf.gradient_descent(X, y)
```

In [27]:
```
coef_2 = coef
coef_2
```

Out[27]:
```
array([[1.45453849],
       [1.45453849],
       [1.45453849],
       [1.45453849],
       [1.45453849]])
```

In [28]:
```
seperating_line2 = coef_2[0] + coef_2[1] * x1 + coef_2[2] *x2 + x3 * coef_2[3] + x4 *

plt.figure(figsize = (5,3))

for i in range(len(y)):
    if y[i] == 1:
        plt.scatter(x = x1[i], y = x2[i], marker='x', c='g', alpha = 0.5)
    else:
        plt.scatter(x = x1[i], y = x2[i], marker='o', c='r', alpha = 0.5)
```

```
plt.plot(seperating_line, c = 'y', label = 'seperating line', linestyle = '-.')
plt.plot(seperating_line2, c = 'g', label = 'seperating line (order 2)', linestyle = '

plt.grid()
plt.legend()
plt.show()
```



The majority of the datapoints are properly classified by the separating line of order 1, however only half of the datapoints are correctly predicted by the separating line of order 2. Therefore, compared to the second classifier, **the first is more efficient.**