ECE M146 - HW 2

John Rapp Farnes | 405461225

1.

We want to prove

$$\underbrace{A^TA \text{ nonsingular}}_{S_1} \Leftrightarrow \underbrace{A \text{ has linearly independent columns}}_{S_2}$$

By definition:

$$(Ax = 0 \Leftrightarrow x = 0) \Leftrightarrow A \text{ has linearly independent columns}$$
$$\therefore \exists z \neq 0 \text{ s.t. } Az = 0 \Leftrightarrow A \text{ has linearly dependent columns}$$

We also have the following properties of (non)singular matrices:

$$A \text{ nonsingular} \Leftrightarrow (Ax = 0 \Leftrightarrow x = 0)$$
$$\therefore A \text{ singular} \Leftrightarrow \exists z \neq 0 \text{ s.t. } Az = 0$$

Logically, $(S_1 \Leftrightarrow S_2) \Leftrightarrow (\neg S_1 \Leftrightarrow \neg S_2) \Leftrightarrow (A^TA \text{ singular} \Leftrightarrow A \text{ has linearly dependent columns})$, which we will try to prove by showing that $\neg S_1 \implies \neg S_2$ as well as $\neg S_2 \implies \neg S_1$.

$\neg S_1 \implies \neg S_2$:

$$A^TA \text{ singular} \implies \exists z \neq 0 \text{ s.t. } A^TAz = 0$$
$$\implies 0 = z^TA^TAz = (Az)^TAz = ||Az||^2 = 0 \implies Az = 0$$
$$\therefore \exists z \neq 0 \text{ s.t. } Az = 0 \implies A \text{ has linearly dependent columns}$$

$\neg S_2 \implies \neg S_1$:

$$A \text{ has linearly dependent columns} \implies \exists z \neq 0 \text{ s.t. } Az = 0$$
$$\implies A^TAz = A^T0 = 0$$
$$\therefore \exists z \neq 0 \text{ s.t. } A^TAz = 0 \implies A^TA \text{ singular}$$

2.

(a)

By definition:

$$H \text{ symmetric} \Leftrightarrow H^T = H$$

Where the transpose has the properties $(A^T)^T = A$, $(AB)^T = B^T A^T$ as well as $(A^{-1})^T = (A^T)^{-1}$, and $A^{-1}A = I$.

We have:

$$H^T = (X(X^TX)^{-1}X^T)^T = X \underbrace{((X^TX)^{-1})^T}_{=((X^TX)^T)^{-1}=(X^TX)^{-1}} X^T = X(X^TX)^{-1}X^T = H$$
$$\implies H \text{ symmetric}$$

(b)

We have $H^1 = H$, and for $k = 2$:

$$H^2 = HH = X(X^TX)^{-1}\underbrace{X^TX(X^TX)^{-1}}_{=I}X^T = X(X^TX)^{-1}X^T = H$$

For $k > 2$ we have

$$H^k = \underbrace{\underbrace{H \cdot H}_{=H} \cdots H \cdot H}_{k \text{ times}} = \underbrace{H \cdot H \cdots H \cdot H}_{k-1 \text{ times}} = H^{k-1}$$

which can be repeated until the exponent is 2. As such $H^k = H$

(c)

We have $(I - H)^1 = I - H$, and for $k = 2$:

$$(I-H)^2 = (I-H)(I-H) = \underbrace{II}_{=I} - \underbrace{IH}_{=H} - \underbrace{HI}_{=H} + \underbrace{HH}_{=H} = I - H$$

As such, the argument in (b) can be applied to the matrix $I - H$, proving $(I - H)^K = I - H$

(d)

Since the value of $Trace$ is independent of the order of multiplication, we can rearrange as such:

$$Trace(H) = Trace(X(X^TX)^{-1}X^T) = Trace(\underbrace{X^TX}_{\text{Dimension } M\times M} \underbrace{(X^TX)^{-1}}_{\text{Dimension } M\times M}) = Trace(I_M) = M$$

3.

We have

$$\begin{aligned}
J(w_0, w_1) \quad &= \sum_{n=1}^{N} \alpha_n (w_0 + w_1 x_n - y_n)^2 \implies \\
\frac{dJ}{dw_0} \quad &= \sum_{n=1}^{N} 2\alpha_n (w_0 + w_1 x_n - y_n) \\
&= 2[w_0 \sum_{n=1}^{N} \alpha_n + w_1 \sum_{n=1}^{N} \alpha_n x_n - \sum_{n=1}^{N} \alpha_n y_n] \\
\frac{dJ}{dw_1} \quad &= \sum_{n=1}^{N} 2\alpha_n x_n (w_0 + w_1 x_n - y_n) \\
&= 2[w_0 \sum_{n=1}^{N} \alpha_n x_n + w_1 \sum_{n=1}^{N} \alpha_n x_n^2 - \sum_{n=1}^{N} \alpha_n x_n y_n] \\
\therefore \nabla J \quad &= \begin{bmatrix} 2[w_0 \sum_{n=1}^{N} \alpha_n (1 + w_1 x_n - y_n)] \\ 2[w_0 \sum_{n=1}^{N} \alpha_n (x_n + w_1 x_n^2 - x_n y_n)] \end{bmatrix}
\end{aligned}$$

We can see that the α_i factors weight each of the terms in the sum of the gradient. This means that if $\alpha_j = 0$ for some j, this obeservation will not be added in the calculation of neither the loss function nor the gradient, and the result will be the same as it was not included. If $\alpha_j >> \alpha_i \; \forall i$ then this term will dominate the sum in the gradient, and the gradient decent algorithm will "point" in the direction that fits that point well more than the others, i.e. dowards the minimum of the loss function in terms of that point.

4.

(a)

We have

$$J(w) = -\sum_{i\in\mathbb{M}} y_i w^T x_i \implies$$
$$\nabla J(w) = -\sum_{i\in\mathbb{M}} y_i x_i$$

By the $\frac{d(w^T b)}{dw} = b$ rule, and linearity of the gradient operator.

(b)

For $z_i = -y_i w^T x_i$ we have that $z_i > 0$ if i is missclassified (since y_i and $w^T x_i$ have opposite signs), and $z_i < 0$ if i is correctly classified (assuming no $z_i = 0$). As such

$$\max(0, z_i) = \begin{cases} -y_i w^T x_i & \text{if the } i\text{th point is missclassified} \\ 0 & \text{else} \end{cases}$$

So, if we let the set $\mathbb{M}$ include the missclassified points, we get that $\sum_{i=1}^{M} \max(0, z_i) = -\sum_{i\in\mathbb{M}} y_i w^T x_i$ as the rest of the terms are 0, and the equation is equivalent to (a) and therefore has the same solution.

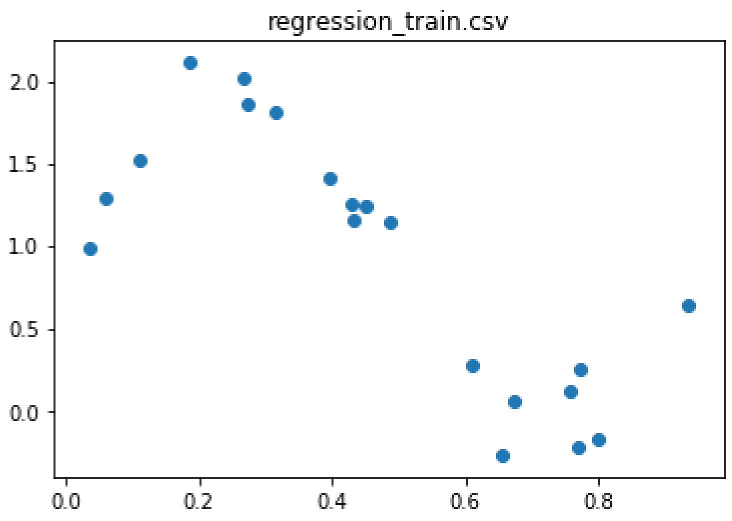
(c)

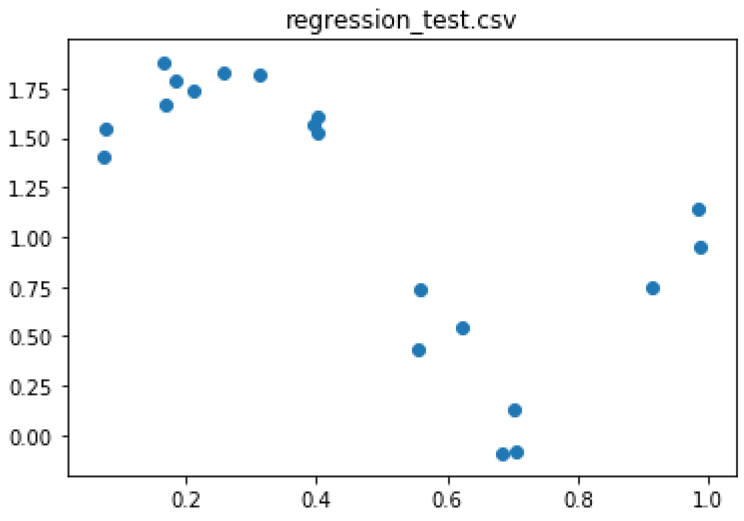
Yes they are equivalent, see (b). Performing the SGD algorithm is equivalent to taking the gradient at only one point i at a time, so the gradient in that point becomes $\nabla J(w) = -y_i x_i$ for the missclassified points and 0 for the correctly classified points. For SGD, we have the update rule $w_{k+1} = w_k - \eta\nabla J(w_k) = w_k + \eta y_i x_i$. With $\eta = 1$, we add $y_i x_i$ to the current w for every missclassified point, which is equivalent to the perceptron algorithm.

5.

(a)

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

filenames = ["regression_train.csv", "regression_test.csv"]

datasets = list(map(lambda f : pd.read_csv(f, header=None, names = ["x", "y"
]), filenames))

for i in range(len(datasets)):
    plt.figure(i)
    plt.scatter(datasets[i].x, datasets[i].y)
    plt.title(filenames[i])
```



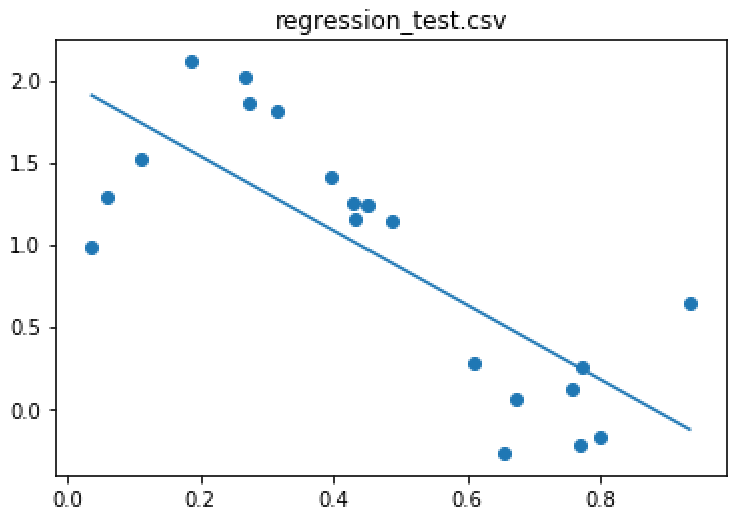


Both files have a downward trend, with added noise. The trend looks linear in the interval $x \leq 0.85$, with some outliers thereafter. If looked as linear data with noise, the noise has a quite big magnitude in relation to the signal, and the algorithm may not perform very well, especially for points $x > 0.85$. However, as the data looks similar to a cubic polynomial, with added noise, linear regression fit with polynomial terms may fit better.

(b)

In [2]:
```
train, test = datasets

def H(X):
    return np.matmul(np.linalg.inv(np.matmul(X.transpose(), X)), X.transpose
())

def get_X(x, m = 1):
    return np.c_[list(map(lambda k : x**k, range(0, m + 1)))].transpose()

def lin_reg_closed_form(x, y, m = 1):
    X = get_X(x, m)
    w = H(X).dot(y)
    return w, X

def plot_lin_reg(x, y, w):
    plt.scatter(x, y)
    xl = [min(x), max(x)]
    yl = list(map(lambda x : w.dot([1, x]), xl))
    plt.plot(xl, yl)

def cost(X, y, w):
    z = np.matmul(X, w) - y
    return z.transpose().dot(z)

data = train
w, X = lin_reg_closed_form(data.x, data.y)

plt.figure(i)
plt.title(filenames[i])
plot_lin_reg(data.x, data.y, w)

print(f'w = {w}')
print(f'J = {cost(X, data.y, w)}')
```

```
w = [ 1.99196163 -2.27048372]
J = 4.603634906406956
```



(c)

In [3]:
```
def grad(X, y, w):
    return (np.matmul(X, w) - y).dot(X)

J_tol = 0.0001
def lin_reg_grad(x, y, rate, MAX_ITERATIONS = 10000):
    X = np.c_[np.ones(x.shape[0]), x]
    w = np.array([0, 0])
    Js = [cost(X, y, w)]
    ws = [w]
    for iters in range(MAX_ITERATIONS):
        w = w - rate * grad(X, y, w)
        ws.append(w)
        J = cost(X, y, w)
        if (abs(J - Js[-1]) < J_tol): break
        Js.append(J)
    return w, X, Js, ws, (iters + 1)

for i in range(len(datasets)):
    data = datasets[i]
    print(f"------ {filenames[i]} ------")

    for rate in [0.03, 0.05, 0.001, 0.0001, 0.00001]:
        print(f"eta = {rate}")
        w, X, Js, ws, iters = lin_reg_grad(data.x, data.y, rate)
        print(f'\tFinal J = {cost(X, data.y, w)}')
        print(f'\tIterations = {iters}')
```

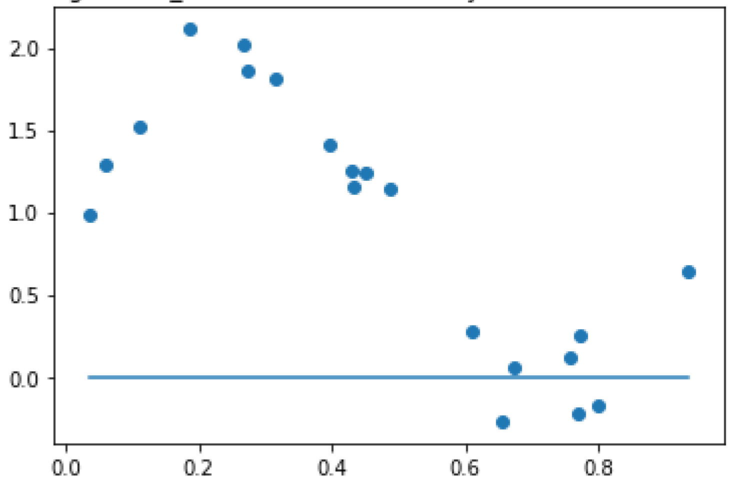
```
------ regression_train.csv ------
eta = 0.03
        Final J = 4.605039456284985
        Iterations = 131
eta = 0.05
        Final J = 4.6044056136703855
        Iterations = 83
eta = 0.001
        Final J = 4.648852786068524
        Iterations = 2420
eta = 0.0001
        Final J = 5.640772178144738
        Iterations = 10000
eta = 1e-05
        Final J = 12.24936726844026
        Iterations = 10000
------ regression_test.csv ------
eta = 0.03
        Final J = 4.558134044503394
        Iterations = 106
eta = 0.05
        Final J = 4.557614122344283
        Iterations = 67
eta = 0.001
        Final J = 4.594362523428096
        Iterations = 1944
eta = 0.0001
        Final J = 5.02267628490104
        Iterations = 10000
eta = 1e-05
        Final J = 9.866646876417967
        Iterations = 10000
```

As seen above, too high of a value of η, e.g. 0.03 in this case, causes the algorithm to "overshoot", and the minumum takes longer to find. Too low of a value makes the algorithm "too slow", and the minimum may not be found in as few iterations, in this case not at all given the 10000 max iterations. A such, there is an "optimal" value of η where the algorithm performs the best, in this case 0.05 performed the best of the tried ones.

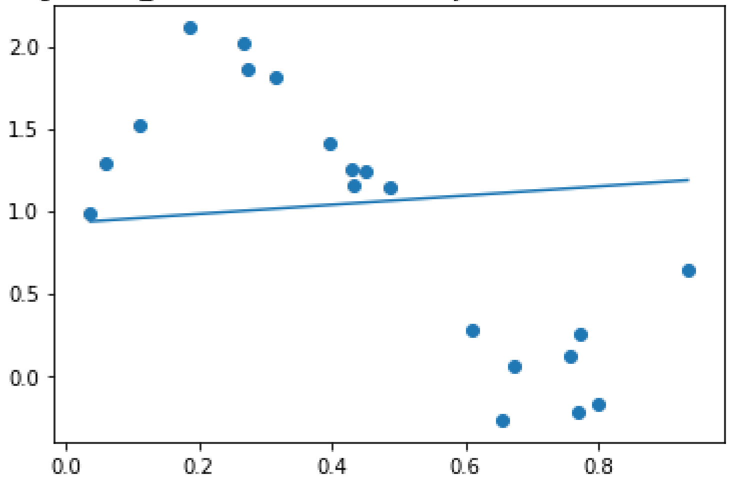
(d)

In [4]:
```
for i in range(len(datasets)):
    data = datasets[i]
    w, X, Js, ws, iters = lin_reg_grad(data.x, data.y, 0.05, 40)

    indices = [0, 10, 20, 30, 40]
    relevant_ws = np.array(ws)[indices]

    for j in range(len(relevant_ws)):
        w = ws[j]
        plt.figure(i * 10 + j)
        plt.title(f'{filenames[i]}, iteration = {indices[j]}, J = {cost(X, dat
a.y, w)}')
        plot_lin_reg(data.x, data.y, w)

    plt.figure(i * 10 + 100)
    plt.plot(Js)
    plt.title(f'Lostt for {filenames[i]}')
    plt.xlabel('Iteration')
```

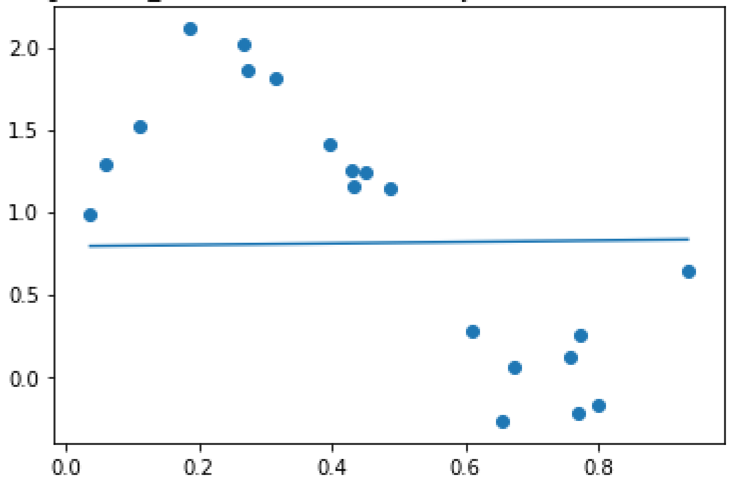
regression_train.csv, iteration = 0, J = 28.655524112125

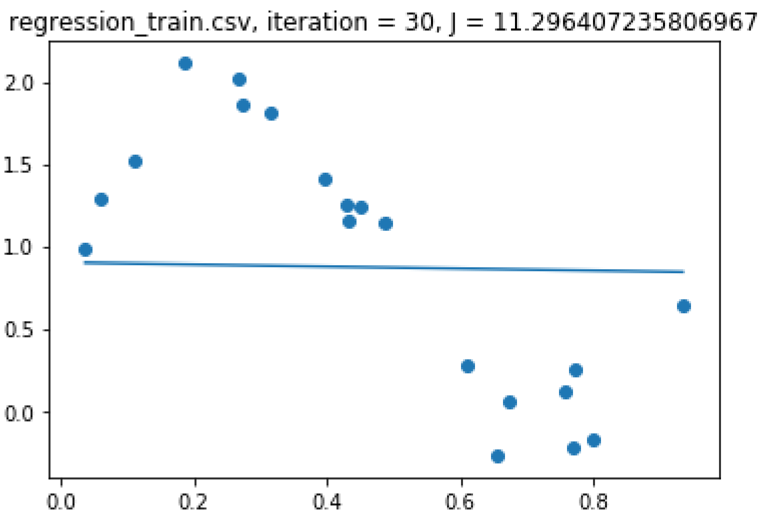


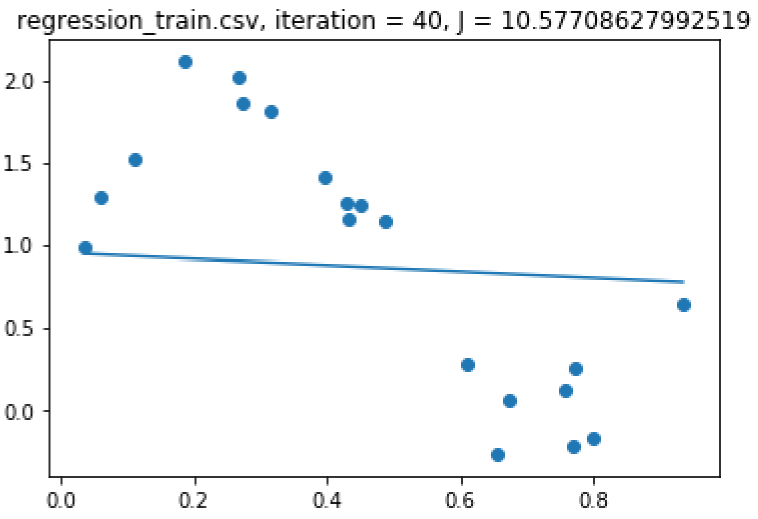
regression_train.csv, iteration = 10, J = 13.802881238906233

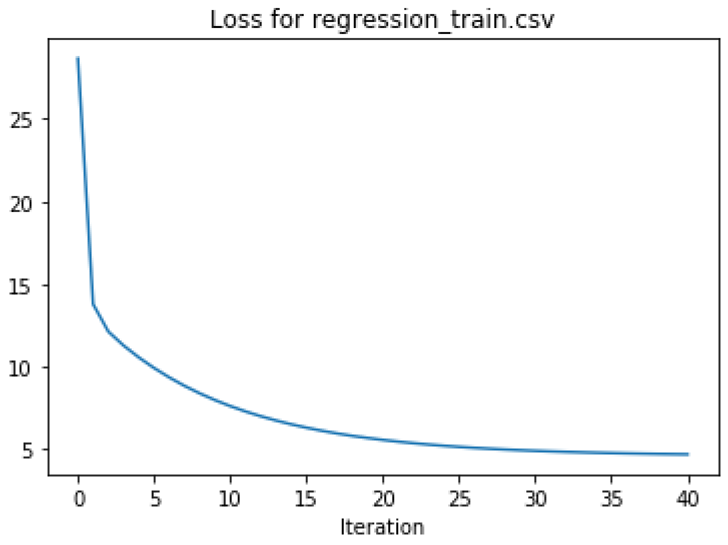


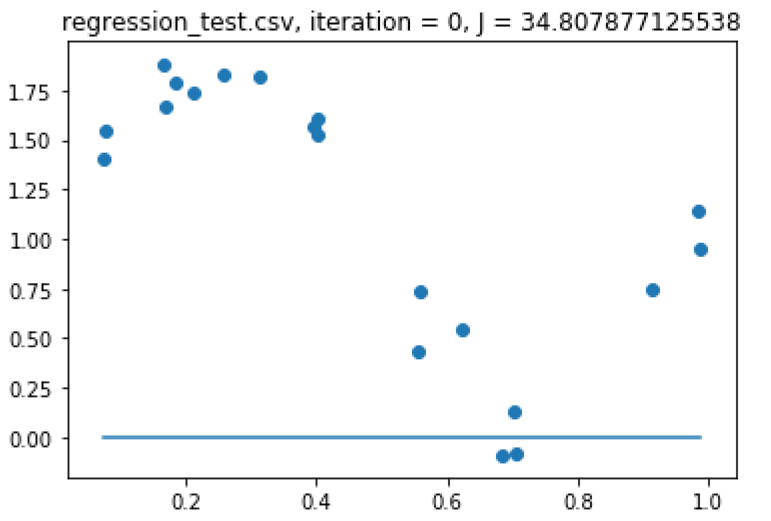
regression_train.csv, iteration = 20, J = 12.14142355273447

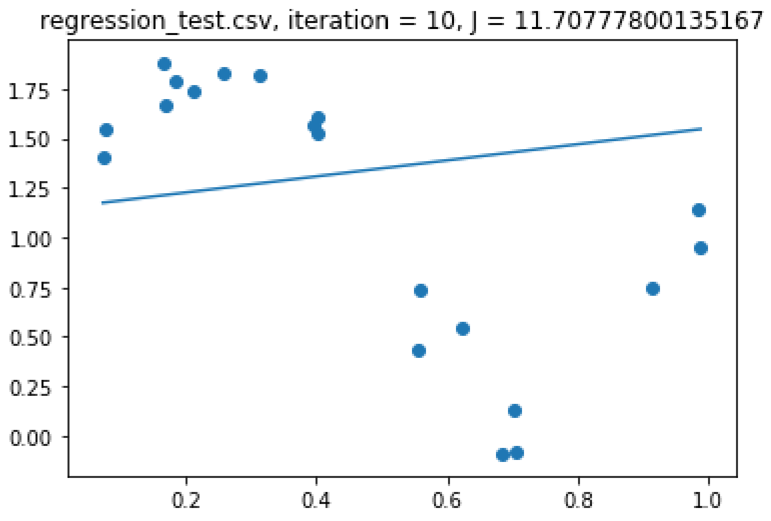


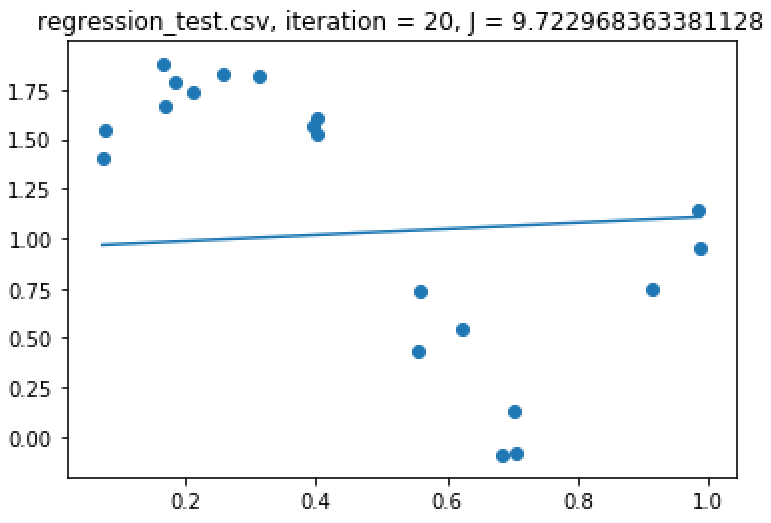

regression_train.csv, iteration = 30, J = 11.296407235806967
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8

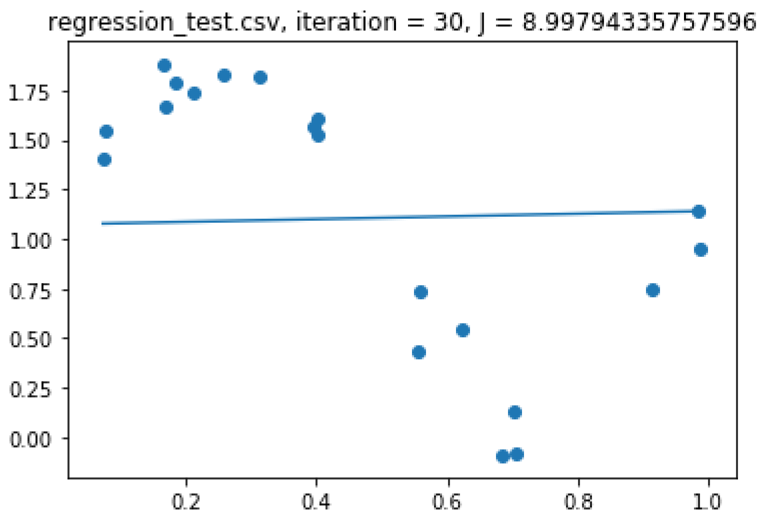

regression_train.csv, iteration = 40, J = 10.57708627992519
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8

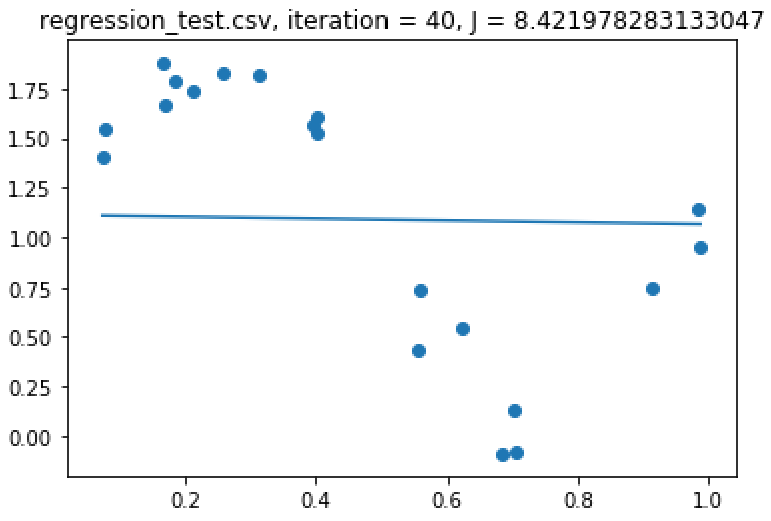

Loss for regression_train.csv
25
20
15
10
5
0
5
10
15
20
25
30
35
40
Iteration

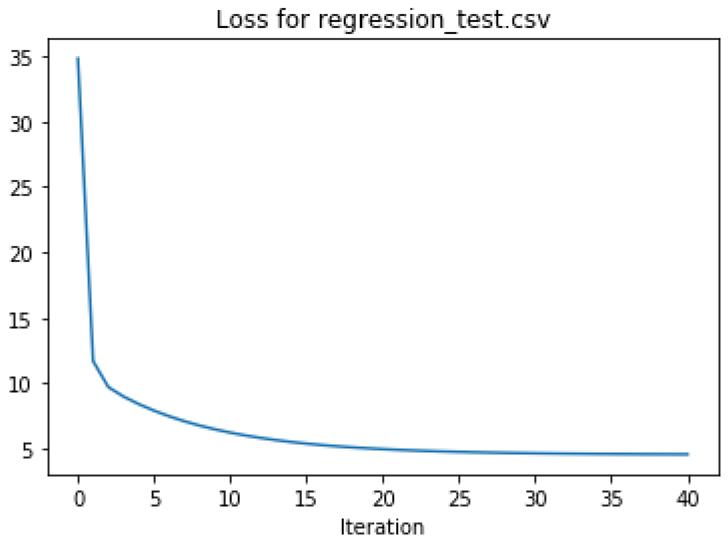

regression_test.csv, iteration = 0, J = 34.807877125538
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.2
0.4
0.6
0.8
1.0


regression_test.csv, iteration = 10, J = 11.70777800135167
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.2
0.4
0.6
0.8
1.0


regression_test.csv, iteration = 20, J = 9.722968363381128
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.2
0.4
0.6
0.8
1.0

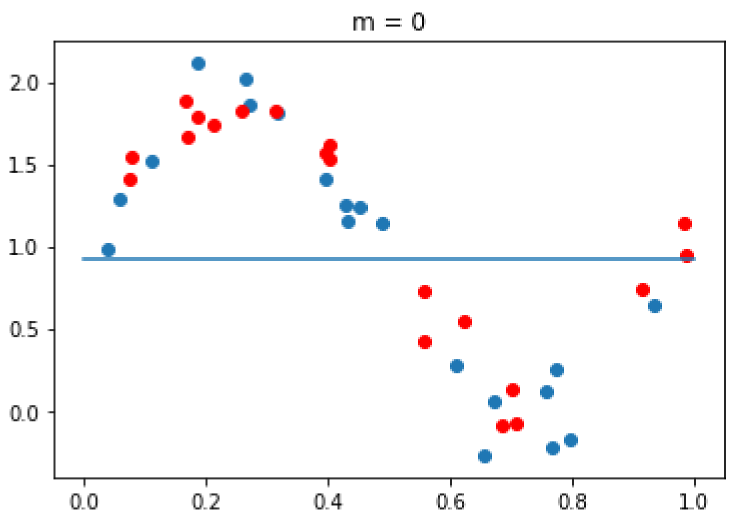

regression_test.csv, iteration = 30, J = 8.99794335757596
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.2
0.4
0.6
0.8
1.0

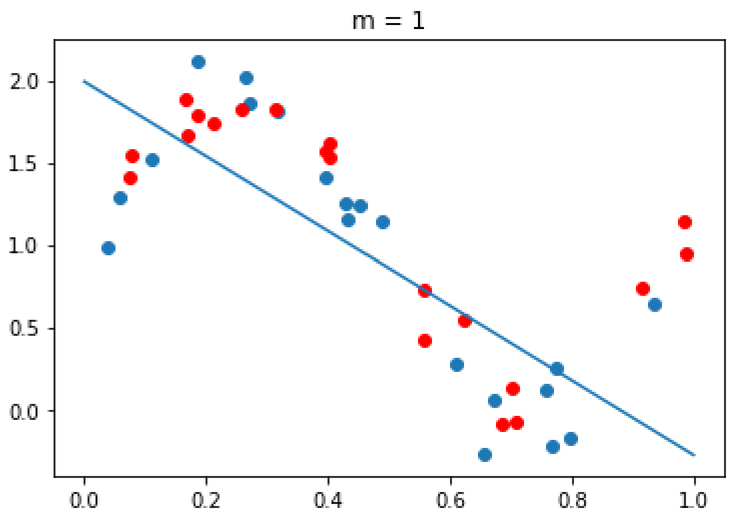

regression_test.csv, iteration = 40, J = 8.421978283133047
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.2
0.4
0.6
0.8
1.0

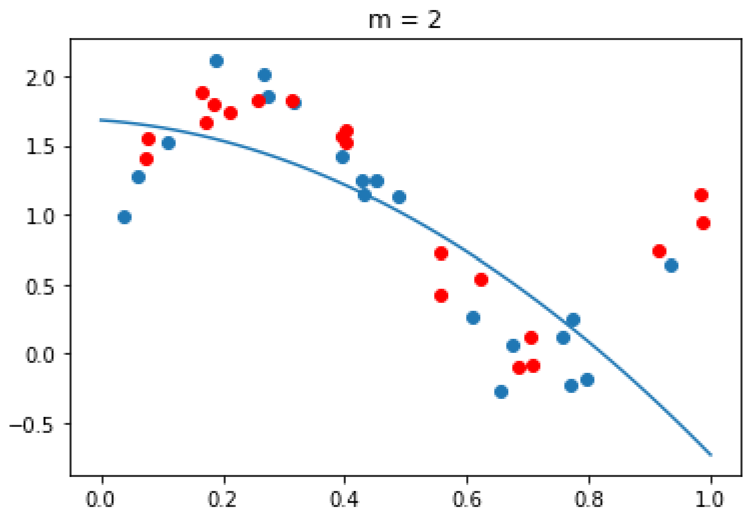

Loss for regression_test.csv
35
30
25
20
15
10
5
0
5
10
15
20
25
30
35
40
Iteration

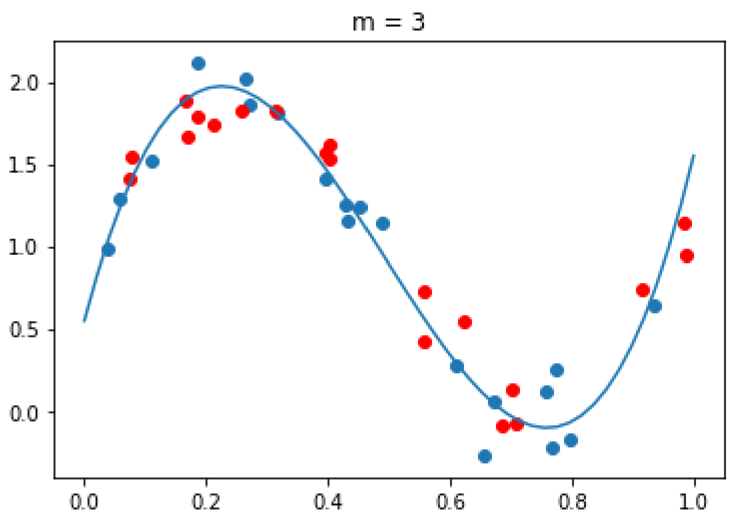
From the printouts and plots, it is clear that the loss function decreases, and that the fitted line aligns more with with the data, as the number of iterations increase. Therefore, the estimate gets better and better for every iteration. Compared to (b), the loss after 40 iterations is lower, which means that the algorithm did not find as "good" a solution, this can also be seen in the fitted line, which does not describe the data as well as the one in (b).

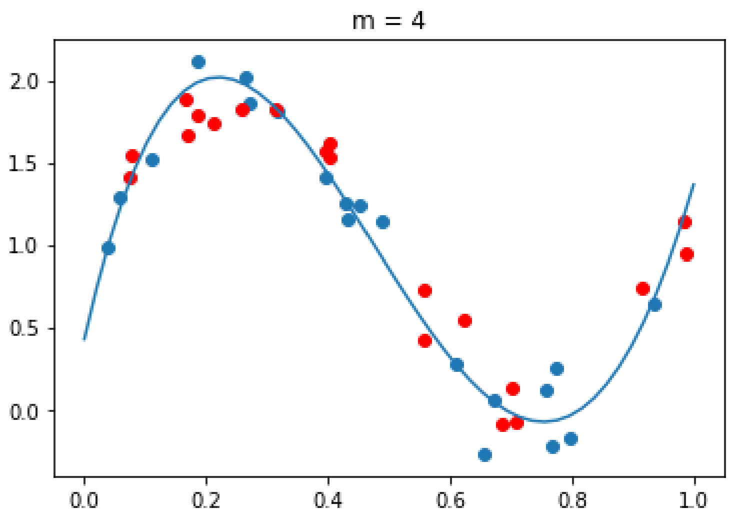
In [5]:
```
J_tol = 0.0001

def E_rms(J, N):
    return np.sqrt(J / N)

ms = []
train_Es = []
test_Es = []

etas = [0.05, 0.05, 0.05, 0.005, 0.005, 0.0005, 0.0005, 0.00005, 0.00005, 0.00
005, 0.00005]
for m in range(11):
    w, X = lin_reg_closed_form(train.x, train.y, m)

    ms.append(m)
    train_Es.append(E_rms(cost(X, train.y, w), train.x.shape[0]))
    test_Es.append(E_rms(cost(get_X(test.x, m), test.y, w), test.x.shape[0]))

    plt.figure(m)
    plt.title(f'm = {m}')
    plt.scatter(train.x, train.y)
    plt.scatter(test.x, test.y, color="red")
    xx = np.linspace(0, 1)
    plt.plot(xx, get_X(xx, m).dot(w))

plt.figure(100)
p_train = plt.scatter(ms, train_Es, color = "red", label="Training")
p_test = plt.scatter(ms, test_Es, color = "green", label="Testing")

plt.title("Polynomial regression")
plt.xlabel("m")
plt.ylabel("E_rms")
_ = plt.legend(handles=[p_train, p_test], loc='upper left')
```

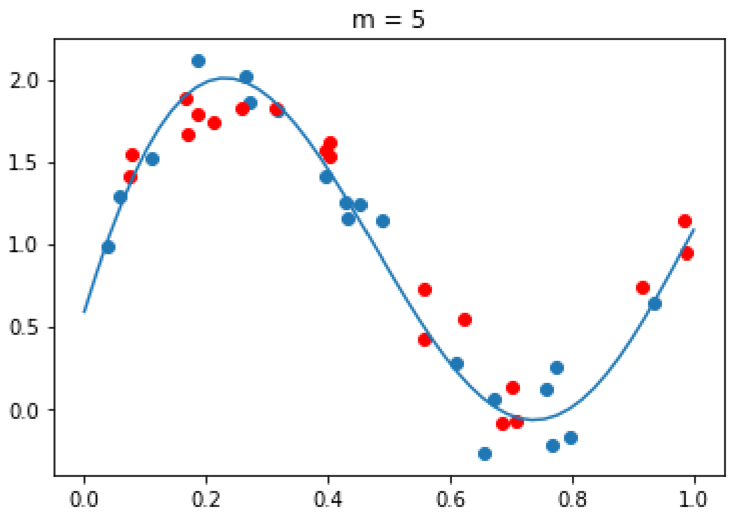

m = 0
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0

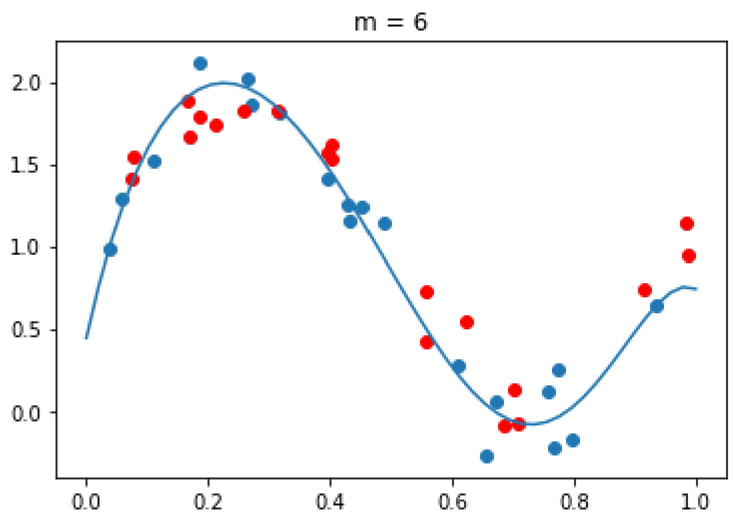

m = 1
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0

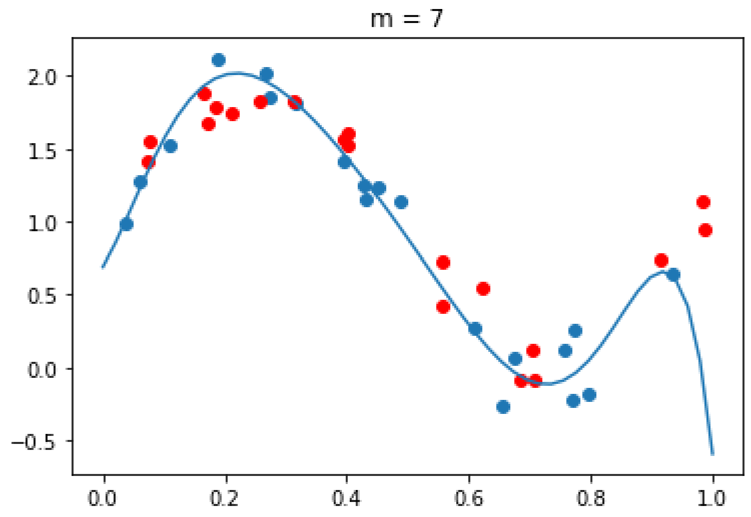

m = 2
2.0
1.5
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0

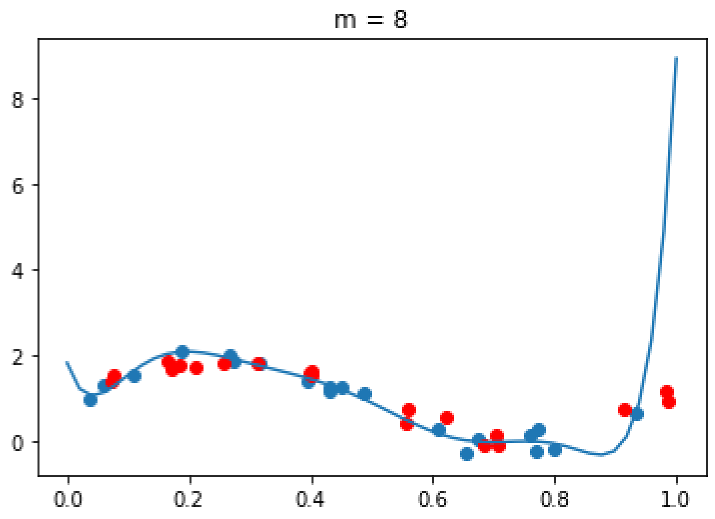

m = 3
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0

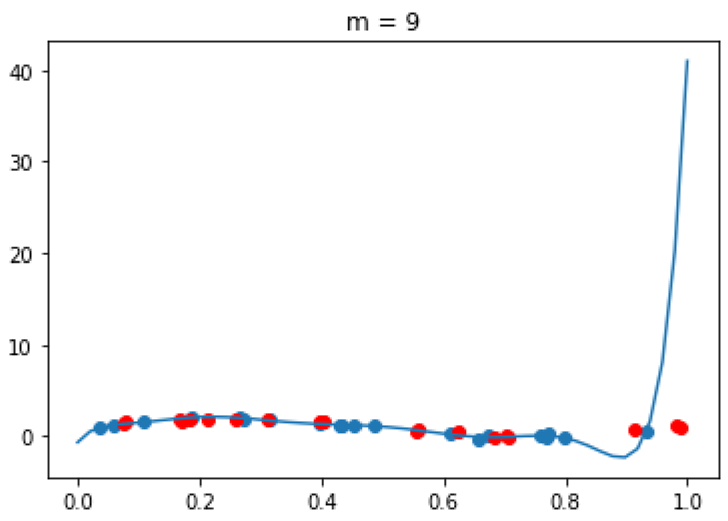

m = 4
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0

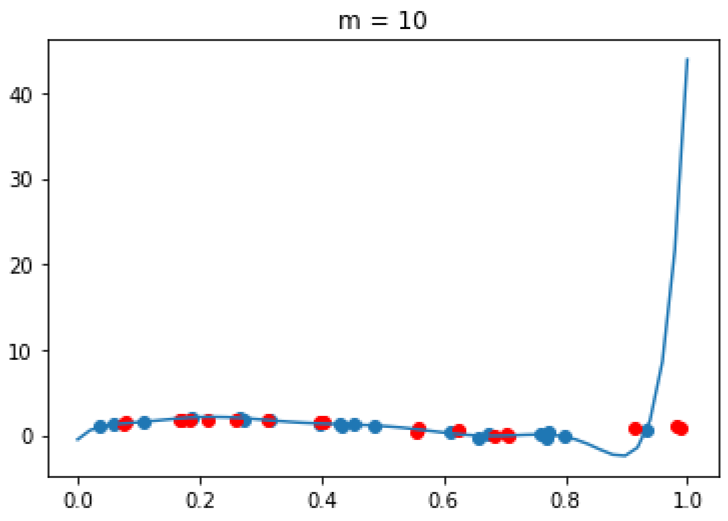

m = 5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0

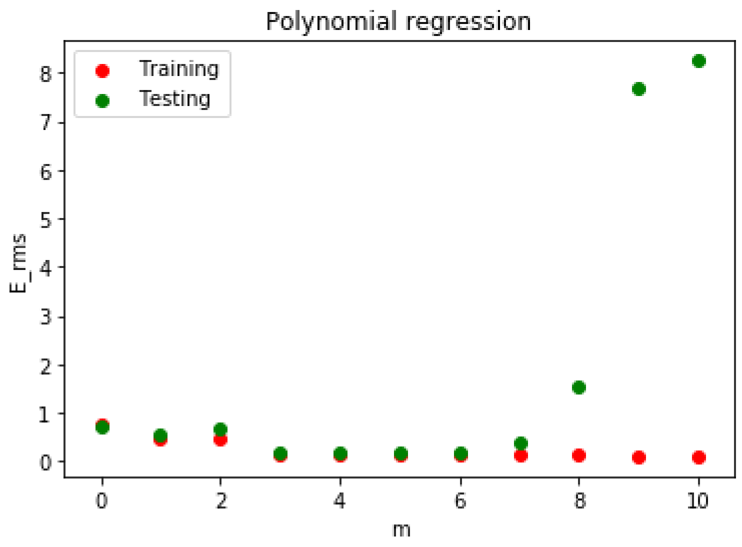

m = 6
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0


m = 7
2.0
1.5
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


m = 8
8
6
4
2
0
0.0
0.2
0.4
0.6
0.8
1.0


m = 9
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0


m = 10
40
30
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0


Polynomial regression
Training
Testing
E_rms
m
0
1
2
3
4
5
6
7
8
0
2
4
6
8
10

As can be seen in the final plot, the E_{rms} decreases for both datasets for $0 \leq m \leq 3$, then stays approximately the same, until the testing E_{rms} starts increasing for $m \geq 7$. This increasing of E_{rms} is evidence of overfitting, as the fitted polynomial very closely follows the points in the training sets for high values of m, however it does not generalize to values between or around the values. This can be seen in the plots of the fits, where the fit is very intricate for the training set when $m = 10$, but shoots far out after the last point and does not predict the testing data very accurately there. At the same time, the decreasing for low values of m is evidence of underfitting, where the model is not complex enough to accurately describe the data.

As all values of $3 \leq m \leq 6$ have approximately the same error, I would pick the least complex model, $m = 3$ as it has the lowest risk of overfitting, and will probably generalize better to unseen data. This also aligns with the "look" of the data, as mentioned in the beginning of this section.