Statistical Machine Learning: Assignment 1

Joris van Vugt, s4279859
Luc Nies, s4136748

September 27, 2016

1

1.1

```
def f(x):
    return 1 + np.sin(6 * (x - 2))

def noisy_f(x):
    noise = np.random.normal(0, 0.3)
    return noise + f(x)

# Generate data
D = [noisy_f(x) for x in np.linspace(0, 1, 10)]
T = [noisy_f(x) for x in np.linspace(0, 1, 100)]
```

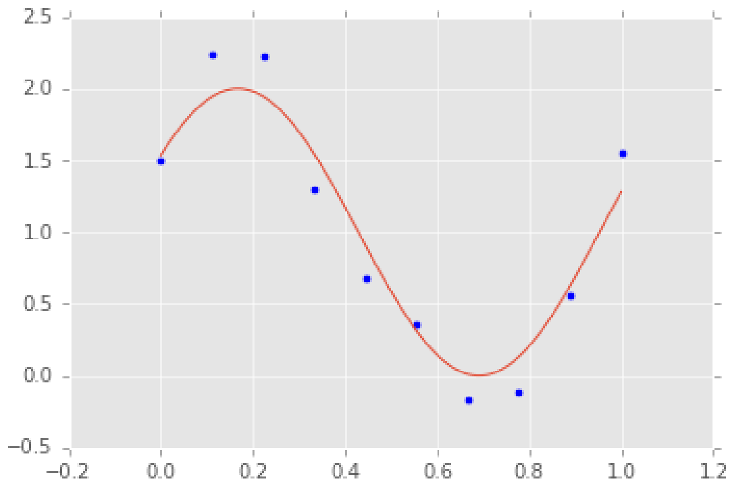

Figure 1: Plot of $f(x)$ and the 10 noisy observations in the training data

1.2

We can use `np.linalg.solve` to solve a set of linear equations in Python/NumPy. We use this function to solve

$$\sum_{j=0}^{M} A_{ij} w_j = T_i$$

with

$$A_{ij} = \sum_{n=1}^{N} x_n^{i+j} \qquad T_i = \sum_{n=1}^{N} t_n x_i^n$$

Which gives the optimal weights for a dataset given the order of the polynomial M. Note that our code is for solving a polynomial of order $M - 1$.

```
def pol_cur_fit(D, M):
    x = D[0, :]
    t = D[1, :]
    A = np.zeros((M, M))
    for i in range(M):
        for j in range(M):
            A[i, j] = np.sum(x ** (i+j))
    T = np.zeros(M)
    for i in range(M):
        T[i] = np.sum(t * x**i)
    w = np.linalg.solve(A, T)
    return w
```

1.3

```
def polynomial(X, w):
    return np.polyval(list(reversed(w)), X)

def RMSE(observed, target):
    error = 0.5 * np.sum((observed - target)**2)
    return np.sqrt(2*error / len(observed))
```

Note that `np.polyeval` expects the weights in reversed order from what we computed with `pol_cur_fit`.

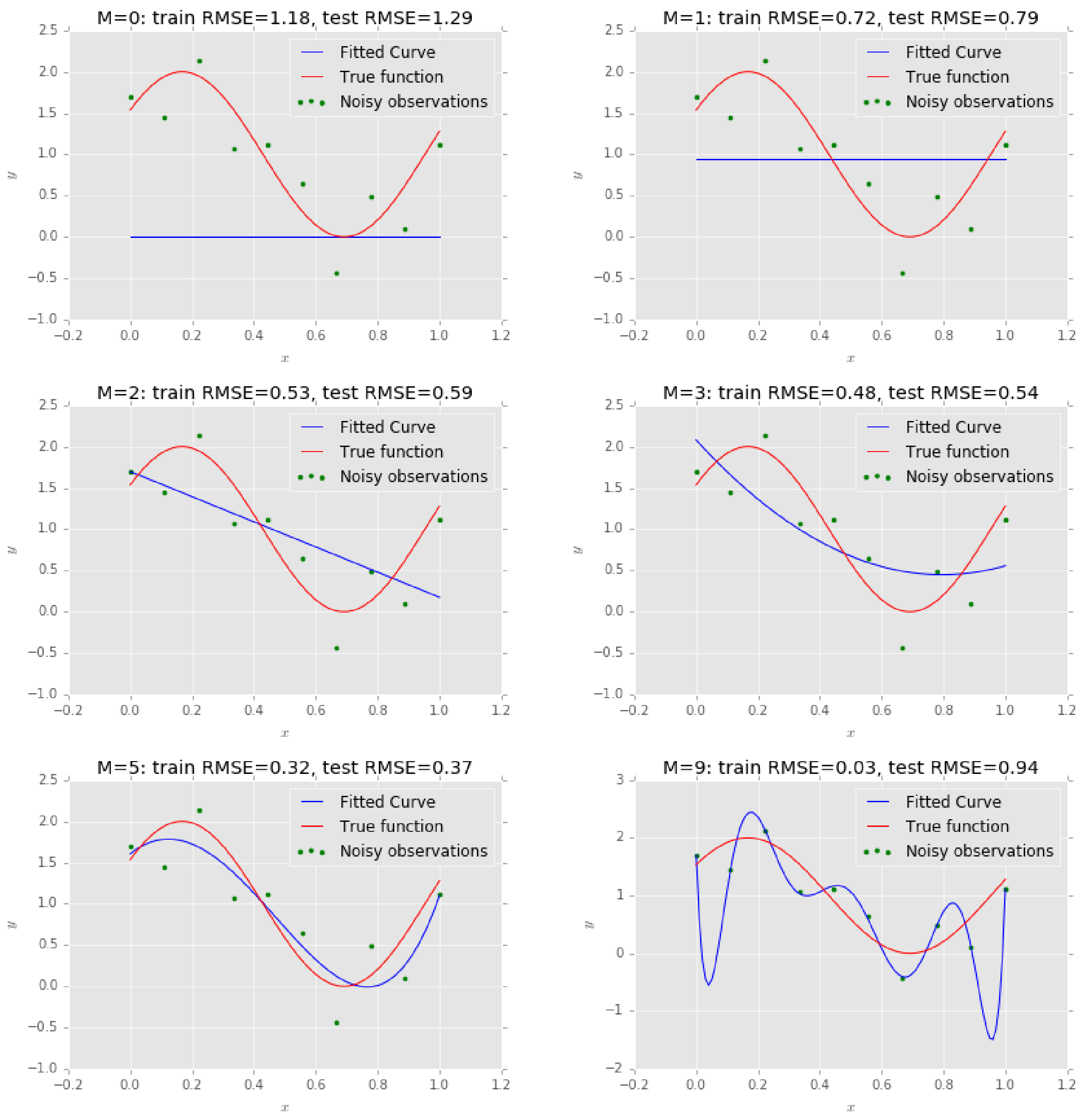

Figure 2: Fitted curves for different values of M

When $M = 2$, the polynomial is linear. From $M = 4$ to $M = 8$, the polynomial fits the underlying sine wave quite well. When $M = 9$, the polynomial is clearly overfitted on the training data. This can also be concluded by comparing the root mean squared errors on the training and test sets.

1.4

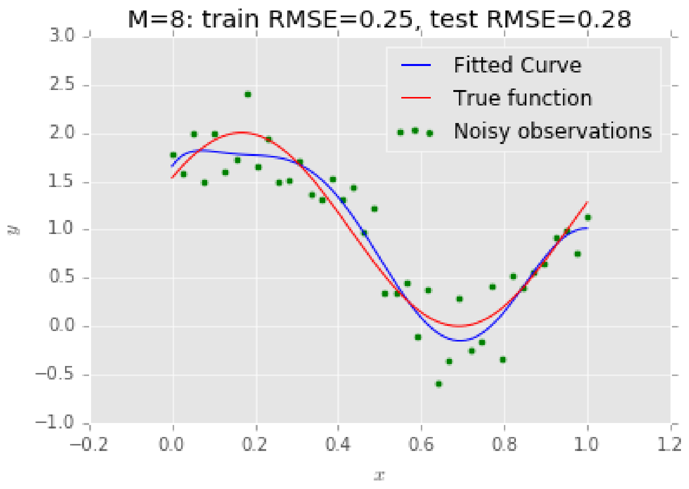

Figure 3: Fitted curve with $M = 8$ and $N = 40$

The fitted curves show less signs of overfitting overall and have lower errors. However for $M \geq 8$ there is still some overfitting to the noise, but since $N > M + 1$, the curve does not have enough degrees of freedom to fit all points exactly.

1.5

$\mathbf{w}$	$\lambda = 0$	$\lambda = 0.1$
w_0^*	1.37	1.78
w_1^*	61.59	-1.19
w_2^*	-1022.49	-1.29
w_3^*	7096.50	-0.62
w_4^*	-25604.26	-0.05
w_5^*	51562.88	0.34
w_6^*	-58476.55	0.60
w_7^*	34924.10	0.77
w_8^*	-8541.82	0.88

Table 1: Optimal weights for $M = 9$ and $N = 10$ with $(\lambda = 0.1)$and without $(\lambda = 0)$ regularization

The weigths with regularization are a lot smaller than the unregularized weights. There is also a smaller difference between the error on the test and the training set when using regularization.

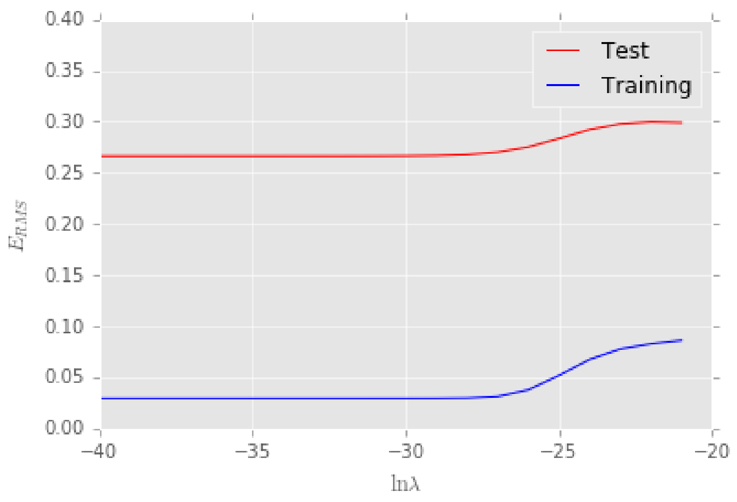

Figure 4: The error on the training and test set versus the regularization strength

The reproduction of Figure 1.8 in Bishop looks slightly different for small values of $\ln\lambda$. We assume that the figure in Bishop is not on scale for $\ln\lambda < 35$ for educational purposes. We also noticed that this figure can vary a lot depending on the training and test set.

Exercise 2

2.1

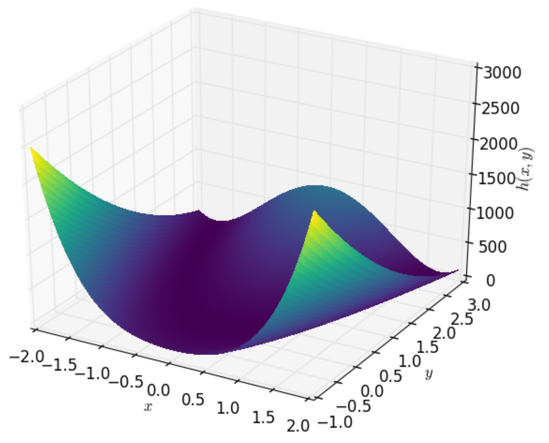

Figure 5: Surface plot of $h(x, y)$

```
def h(x, y):
    return 100 * (y - x**2)**2 + (1 - x)**2
```

We think that using gradient descent for this function will be slow, because there is a large 'valley' (dark-blue area in the plot). The function also grows very rapidly outside the valley, which might make gradient descent more difficult because it will be difficult to find the right learning rate.

2.2

We first calculate the gradient of the functions with respect to x and y.

$$\begin{aligned} h(x,y) &= 100(y-x^2)^2 + (1-x)^2 && (1)\\ \frac{\partial h}{\partial y} &= 200(y-x^2) && (2)\\ \frac{\partial h}{\partial x} &= -400x(y-x^2) - 2 + 2x && (3)\\ & && (4)\end{aligned}$$

We now set $\frac{\partial h}{\partial y} = 0$ to find the minimum.

$$\begin{aligned} 200(y-x^2) &= 0 && (5)\\ 200y &= 200x^2 && (6)\\ y &= x^2 && (7)\end{aligned}$$

Substitute the result in $\frac{\partial h}{\partial x}$ to find the x-coordinate of the minimum.

$$\begin{aligned} -400x(y-x^2) - 2 + 2x &= 0 && (8)\\ -400x(x^2-x^2) - 2 + 2x &= 0 && (9)\\ -2 + 2x &= 0 && (10)\\ 2x &= 2 && (11)\\ x &= 1 && (12)\end{aligned}$$

Lastly, we can use that to find the y-coordinate of the minimum.

$$y = 1^2 = 1 \qquad (13)$$

So the minimum of h is indeed at $(1,1)$.

2.3

We fill in the partial derivates of h into $\mathbf{x}_{n+1} = \mathbf{x}_n - \eta\nabla E(\mathbf{x}_n)$

$$\begin{aligned} x_{n+1} &= x_n - \eta(-400x_n(y_n - x_n^2) - 2 + 2x_n) && (14)\\ y_{n+1} &= y_n - \eta(200(y_n - x_n^2)) && (15)\end{aligned}$$

2.4

For small η values ($\eta \leq 0.001$ in the tests we ran) gradient descent converges, but for larger values it quickly overshoots and never manages to converge. This is due to the rapid increase of h. It is hard to find a learning rate that scales well for the entire range of h. So when gradient descent overshoot, it will overcompensate and quickly diverge. The smaller the η the more steps are needed.

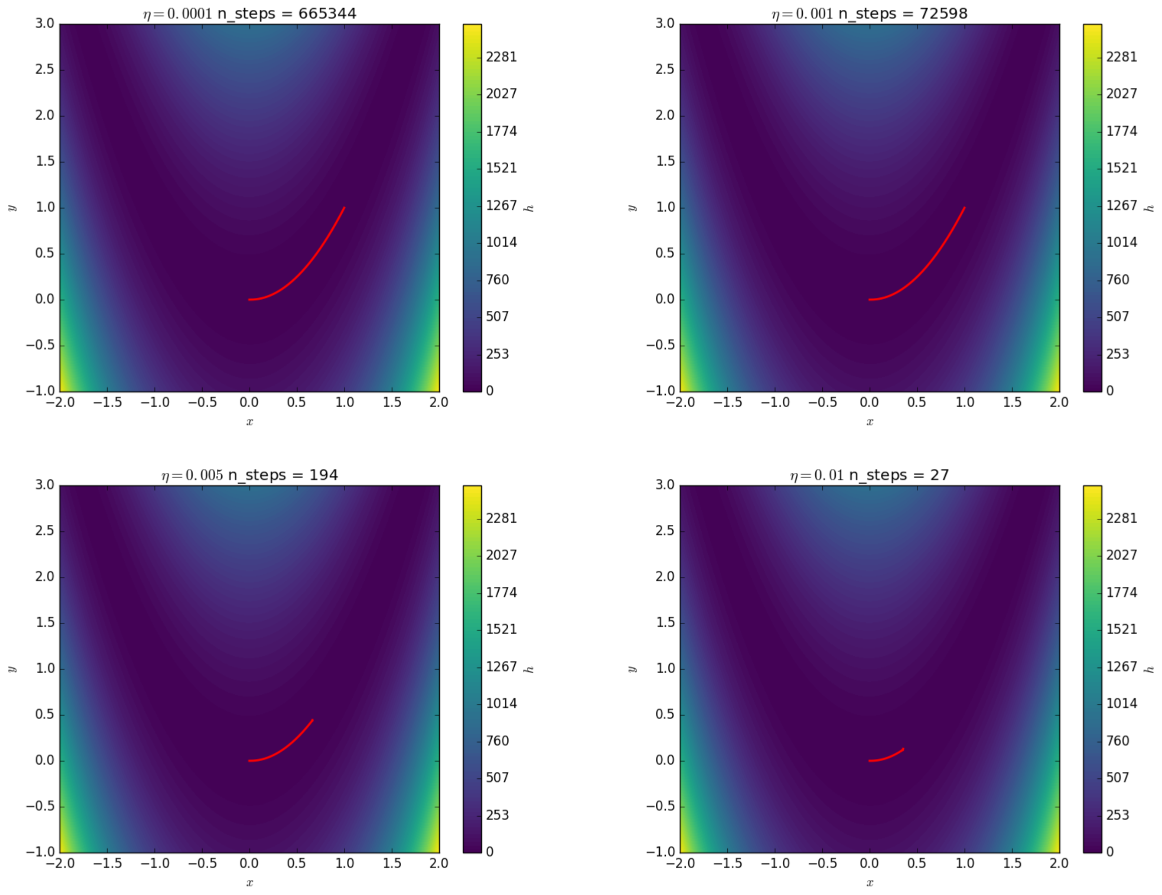

Figure 6: Gradient descent with different values or η, with start values $x = 0$ and $y = 0$

```python
def gradient_descent(eta = 0.005, start_x = 0, start_y = 0, max_iter = 500):
    x_list, y_list, h_list  = [start_x], [start_y], [h(start_x, start_y)]
    x = start_x
    y = start_y
    for _ in range(max_iter):
        x_temp = x - eta * (400 * x * (x**2-y) - 2 + 2 * x)
        y = y - eta * (200 * (y - x**2))
        x = x_temp
        h_new = h(x,y)
        if h_new >= h_list[-1]:
            break # Stop if h doesn't decrease anymore

        x_list.append(x)
        y_list.append(y)
        h_list.append(h(x,y))


    return x_list, y_list, h_list
```

3

3.1

$$B = \{1, 2\} \tag{16}$$
$$F = \{A, G\} \tag{17}$$

We want to calculate: $P(F_1 = A|F_2 = G)$. To this end, we first calculate the probability of picking a grapefruit from each box.

$$P(F = A|B = 1) = \frac{2}{3} \tag{18}$$
$$P(F = A|B = 2) = \frac{5}{6} \tag{19}$$
$$P(F = G) = P(F = G|B = 1)P(B = 1) + P(F = G|B = 2)P(B = 2) \tag{20}$$
$$= \frac{1}{3} \times \frac{1}{2} + \frac{1}{6} \times \frac{1}{2} = \frac{1}{4} \tag{21}$$
$$P(B = 1|F = G) = \frac{P(F = G|B = 1)P(B = 1)}{P(F = G)} \tag{22}$$
$$= \frac{\frac{1}{3} \times \frac{1}{2}}{\frac{1}{4}} = \frac{2}{3} \tag{23}$$

$$P(B = 2|F = G) = 1 - P(B = 1|F = G) \tag{24}$$
$$= 1 - \frac{2}{3} = \frac{1}{3} \tag{25}$$

Intuitively, these are now the probabilities of each of the boxes.

$$P(F_1 = A, |F_2 = G) = P(F = A|B = 1)P(B = 1|F = G) + P(F = A|B = 2)P(B = 2|F = G) \tag{26}$$
$$= \frac{2}{3} \times \frac{2}{3} + \frac{1}{3} \times \frac{5}{6} \tag{27}$$
$$= \frac{4}{9} + \frac{5}{18} \tag{28}$$
$$= \frac{13}{18} \tag{29}$$

So the probablity that the first fruit is an apple given that the second fruit is a grapefruit is $\frac{13}{18}$. It is less likely that a fruit picked from box 1 is an apple than when it was picked from box 2. If the second fruit is a grapefruit, it is more likely that it is picked from box 1, making it less likely that the first fruit is an apple.

3.2

The new proboality of picking a grapefruit from box 1 is:

$$P(F = G|B = 1) = \frac{4}{24} = \frac{1}{6} = P(F = G|B = 2) \tag{30}$$

So from this, it follows that $P(B = 1|F = G) = P(B = 2|F = G) = \frac{1}{2}$. Since the probabilities of picking a grapefruit from both boxes are equal, it does not tell us anything about which box is now more likely.
The two picks are still dependent, as the other two fruits have different probablities per box. So if the first pick is an orange, it still tells us something about the second pick.