Statistical Machine Learning
Assignment 1

Inez Wijnands (s4149696) & Guido Zuidhof (s4160703)

Radboud University Nijmegen

29/09/2015

The entire code listing is in a separate file. The listings shown here are merely code snippets.

Exercise 1

1. The function $t = f(x)$ we have created is $f(x) = 1 + sin(8x + 1)$. This function is neither even nor odd. An even function is a function such as $cos(x)$, and an odd function is a function such as $sin(x)$, since we changed the phase and intercept of $sin(x)$ our function is neither. See Figure 1 for the function $f(x)$ and observations $\mathcal{D}$.

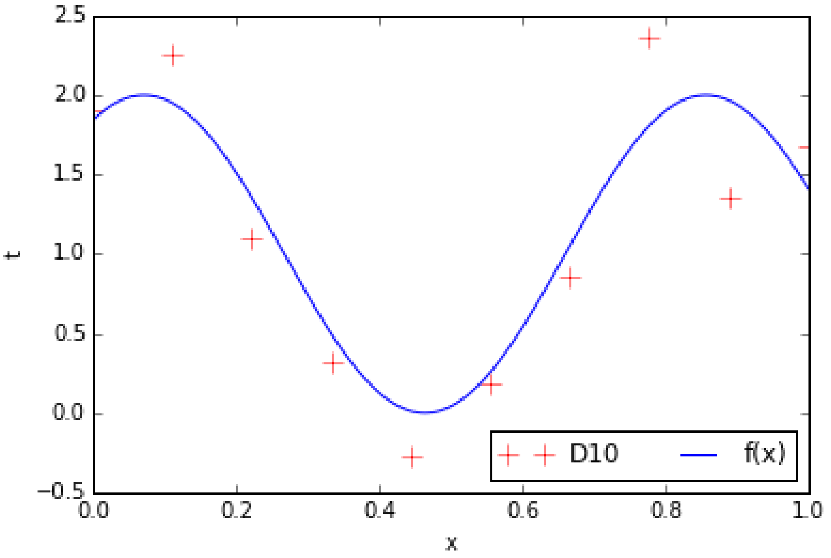

Figure 1: The function $f(x)$ and observations $\mathcal{D}$ are plotted (similar to Bishop, Fig.1.2)

2. See Listing 1 for the function $\boldsymbol{w} = PolCurFit(\mathcal{D}, M)$.

   Listing 1: Python code for function PolCurFit – Input are the observations $\mathcal{D}$ and the results t of function $f(x)$ and the order of the polynomial M. The functions calculates the A-matrix and T-vector and solves this equation to find the weights.

```python
def PolCurFit(D,M):
    x = D[0]
    t = D[1]
    M = M + 1
    A = np.array([[Aij(i,j,x) for j in xrange(M)] for i in xrange(M)])
    T = np.array([Ti(i,t,x) for i in xrange(M)])
    return np.linalg.solve(A,T)
```

```

# A-matrix
def Aij(i,j,x):
    return np.sum(x[n]**(i+j) for n in xrange(len(x)))
# T-vector
def Ti(i,t,x):
    return np.sum(t[n]*(x[n]**i) for n in xrange(len(x)))
```

3. See Figure 2 & 3

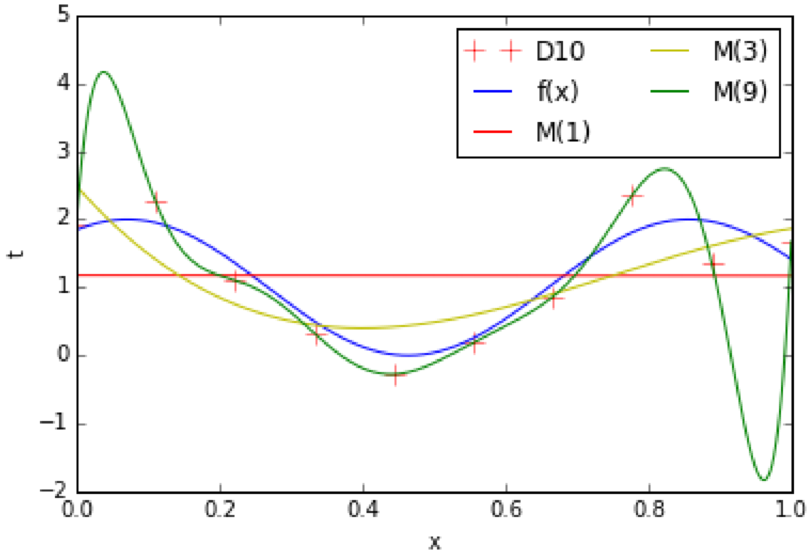

Figure 2: In this figure, the observations $\mathcal{D}$, the function $f(x)$ and polynomials of different orders of M are shown.

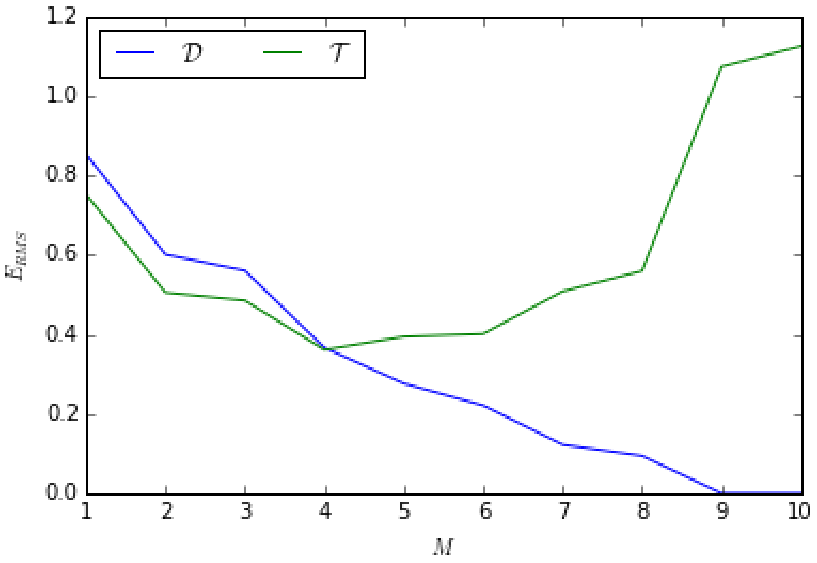

Figure 3: For the various orders $M = [1, \dots, 10]$ and for each solution $\boldsymbol{w}^*$ the root-mean-quare error $E_{RMS} = \sqrt{2E(\boldsymbol{w}^*)/N}$ is computed of the corresponding polynomial, evaluated on both the training set $\mathcal{D}$ and the testset $\mathcal{T}$ (similar to Bishop, Fig.1.5)

4. See Figure 4 & 5. Since there are now 40 observations in $\mathcal{D}$, larger order polynomials work better.

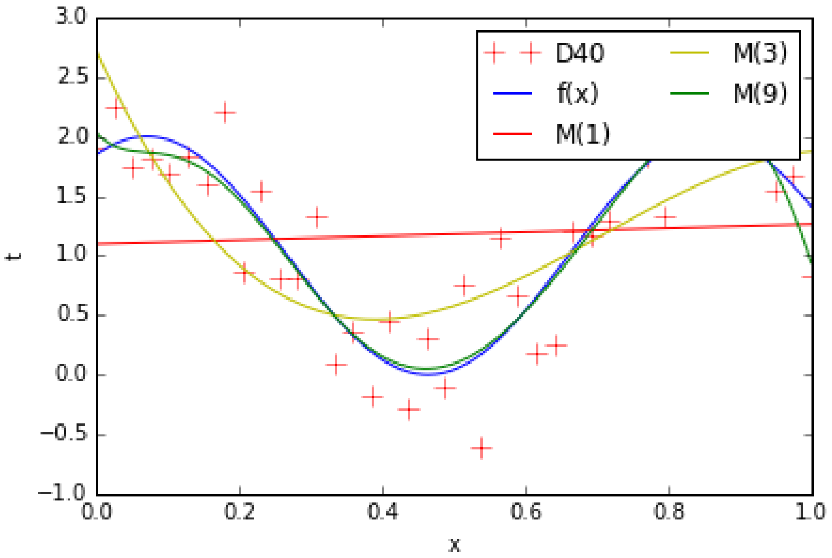

Figure 4: In this figure, the observations $\mathcal{D}$, the function $f(x)$ and polynomials of different orders of M are shown.

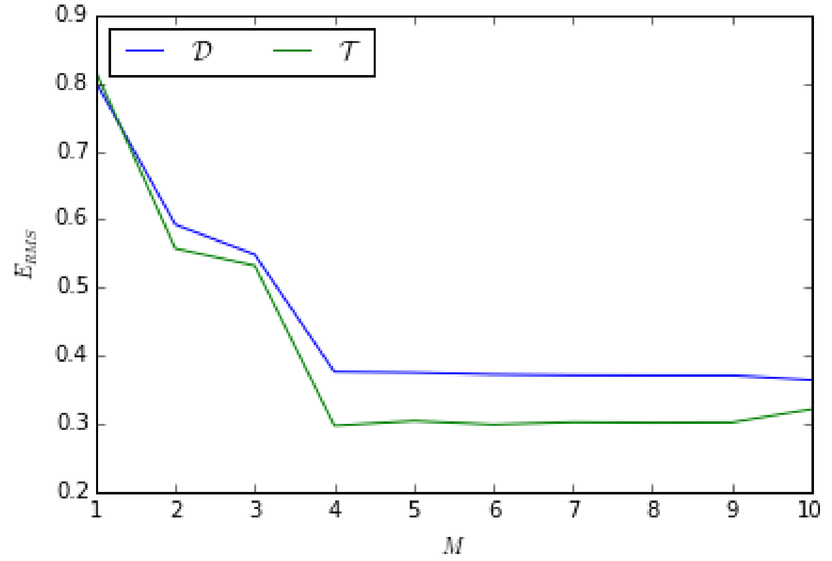

Figure 5: For the various orders $M = [1, \ldots, 10]$ and for each solution $\boldsymbol{w}^*$ the root-mean-quare error $E_{RMS} = \sqrt{2E(\boldsymbol{w}^*)/N}$ is computed of the corresponding polynomial, evaluated on both the training set $\mathcal{D}$ and the testset $\mathcal{T}$. (similar to Bishop, Fig.1.5)

For 10 observations, a polynomial of the 10th order will create a function that goes straight through every observation, as you can see in Figure 2, but these observations are noisy, thus the function $f(x)$ will not be found, and adding more observations will cause the error to increase dramatically since these will not follow the curves of that polynomial. With more observations, the 9th- and 10th-order polynomials do not match every observation, but it will be a far better approximation of $f(x)$. In Figure 3 and 5, you can see that when the order is the same as the amount of observations the error can become zero, but a change in the order will increase the error drastically. With more observations in the training set, the error of the training set and testset are more similar.

5. We included the λ parameter in the A-matrix. Since the gradient of $\tilde{E}$ with respect to $\boldsymbol{w}$ is $A\boldsymbol{w} -$

$\boldsymbol{T}+\lambda\boldsymbol{w}$, we can rewrite this to $(A+\lambda I)\boldsymbol{w}-\boldsymbol{T}$[1] See Listing 2 for the altered function $PolCurFit()$. See Figure 6 and 7 for the results of the modified $PolCurFit()$.

Listing 2: Python code for function PolCurFit – Penalty added.

```python
def PolCurFit(D,M, _lambda=0):
    x = D[0]
    t = D[1]
    M = M + 1

    #Construct A matrix by calling Aij function for every entry
    A = np.array([[Aij(i,j,x) for j in xrange(M)] for i in xrange(M)])

    #Ridge Regression
    A = A + _lambda * np.identity(len(A))

    #Construct T vector
    T = np.array([Ti(i,t,x) for i in xrange(M)])
    return np.linalg.solve(A,T)
```

[1] See Bishop, pp.144-145 for the used theory behind the regularization of the least-squares, using λ with the identity matrix.

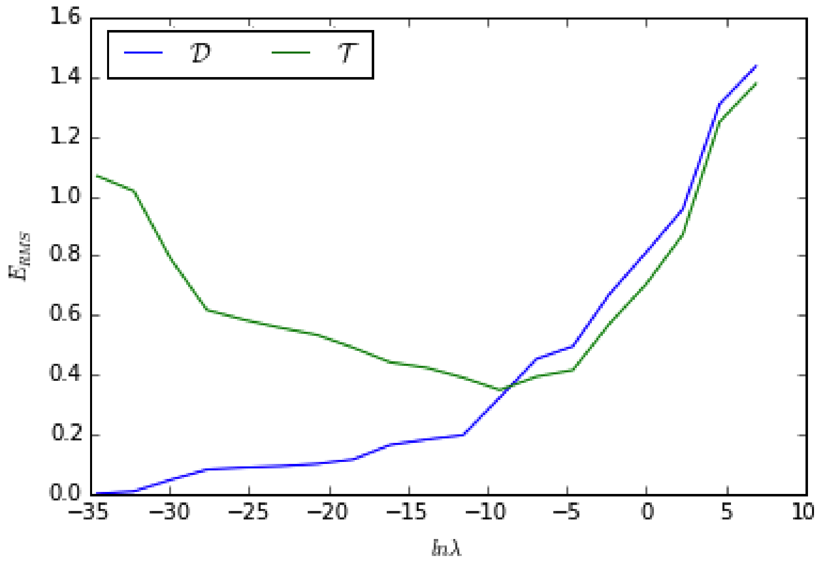

Figure 6: For $M = 9$, we calculated the root-mean-squared error for the training set $\mathcal{D}$ and the testset $\mathcal{T}$. You can see that the root-mean-squared error increases for larger λ for $\mathcal{D}$ and first decreases and later increases for $\mathcal{T}$. You try to minimize both the error and the sum of the weights, and these weights are regularized by λ. If your λ is very high, the weights become smaller to minimize the sum of the error plus the λ times the weights. For a large order polynomial (such as $M = 9$ as in this plot), the weights have to be quite high to pass all the data points in the training set as close as possible. This is however noisy data, with few observations. But for larger λ, you lower the weights, and thus the error will increase on the training set. (similar to Bishop, Fig.1.8).

For the error on the test set you can see that it first decreases, this is due to a reduction of overfit. Later, it increases again as the weights are too low to capture the original function.

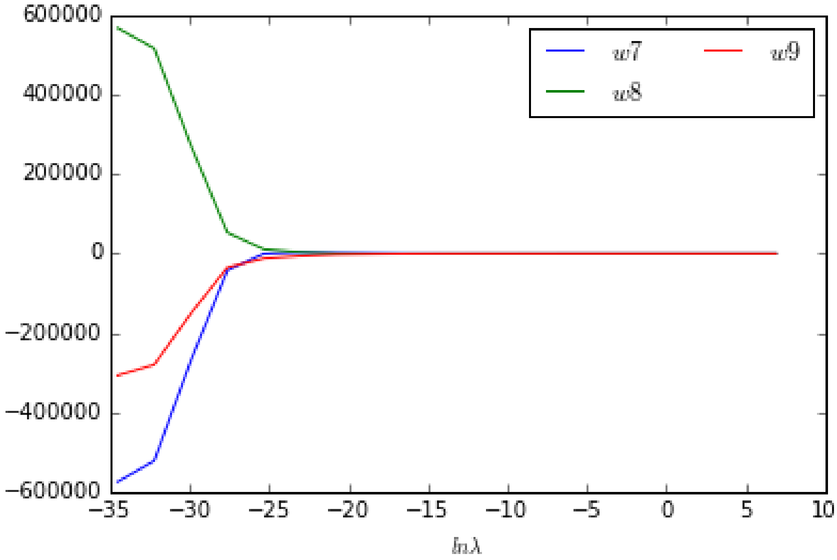

Figure 7: The value of the weights of the 7th-, 8th- and 9th-order polynomials are plotted against the regularizer λ. You can see that the weights are driven to zero.

Exercise 2

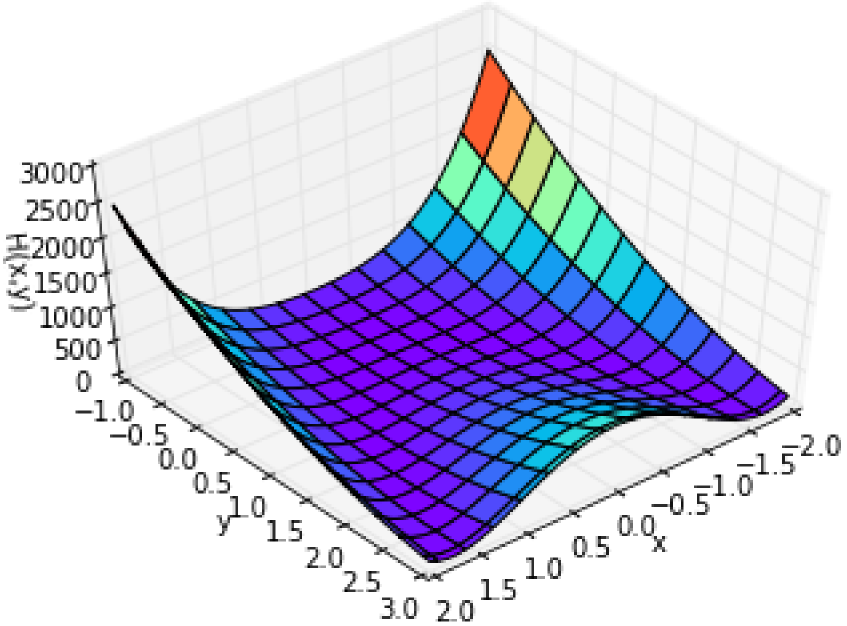

Figure 8: Surface plot of $h(x, y)$ over the interval $[-2 \leq x \leq 2] \times [-1 \leq y \leq 3]$.

1. At first glance the function seems symmetrical, but it's not. The gradient is very small in the *valley* visible in the plot, it does however only have 1 minimum and no local minima (at least in this interval). We expect convergence to be fairly slow due to this low gradient.

2. First we rewrite $h(x, y)$:
   $h(x, y) = 100(y - x^2)^2 + (1 - x)^2 = 100(y^2 - 2x^2y + x^4) + x^2 - 2x + 1 =$
   $100y^2 - 200x^2y + 100x^4 + x^2 - 2x + 1$

   We compute the gradient of $h(x, y)$ with respect to x and with respect to y:
   $\frac{dh(x,y)}{dx} = -400xy + 400x^3 + 2x - 2$
   $\frac{dh(x,y)}{dy} = 200y - 200x^2$

   Set derivative of $h(x, y)$ equal to zero to find minimum:
   $200y - 200x^2 = 0 \rightarrow 200y = 200x^2 \rightarrow y = x^2$
   $-400xy + 400x^3 + 2x - 2 = 0 \rightarrow$ use $y = x^2 \rightarrow 400x^3 = 400x^3 + 2x - 2 \rightarrow 2x = 2 \rightarrow x = 1$
   $y = x^2$ and $x = 1 \rightarrow y = 1^2 = 1$
   (1,1) is the minimum of $h(x, y)$

3. Gradient descent iteration rule:
   $x_{n+1} = x_n - \eta \nabla E(x_n)$
   $E(x_n) = h(x, y)$

   $x_{n+1} = x_n - \eta \frac{dh(x_n,y_n)}{dx_n} = x_n - \eta(-400x_ny + 400x_n^3 + 2x_n - 2$

$= (1 + \eta 400y) * x_n - \eta 400x_n^3 - \eta 2x_n + \eta 2$
$= (\eta 400y - 1)x_n - \eta 400x_n^3 + \eta 2$

$y_{n+1} = y_n - \eta \frac{dh(x_n,y_n)}{dy_n} = y_n - \eta(200y_n - 200x^2)$
$= (1 - \eta 200)y_n - \eta 200x^2$

4. We investigated $\eta \in \{0.01, 0.001, 0.0001, 0.00001\}$, as starting point we took $(-2, 2)$. $\eta = 0.01$ did not converge, as the steps it took were large enough to end up higher on the opposite side of the valley. It quickly moved out of bounds. The other values of η did converge, see Table 1 for the number of iterations for the specific amount of iterations required. A smaller value of η leads to a higher amount of iterations required, as was expected. The η parameter can not be too high, as it would quickly go out of bounds when started on the steep gradient found at the edge of the interval.

   The very low gradient and the relatively low η required makes for very slow numerical minimization with gradient descent for this function. See below for the trajectory plots.

Table 1: Convergence of GD for different values of η

η	Iterations until convergence
0.01	Did not converge
0.001	26390
0.0001	267391
0.00001	2674138

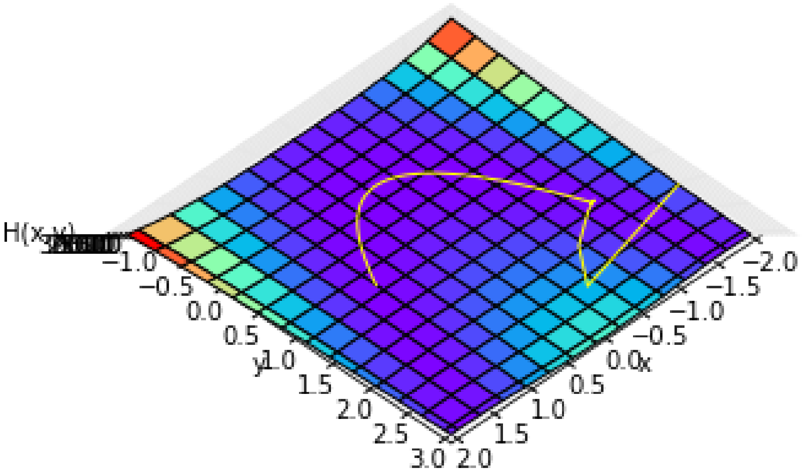

Figure 9: Trajectory of gradient descent with $\eta = 0.001$.

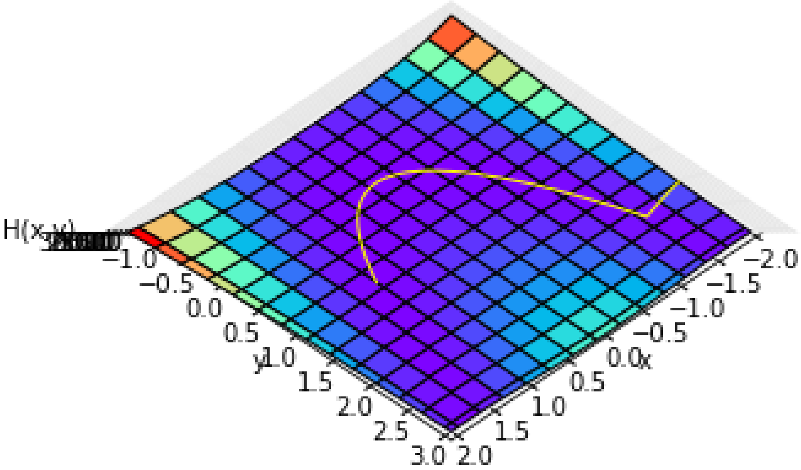

Figure 10: Trajectory of gradient descent with $\eta = 0.0001$.

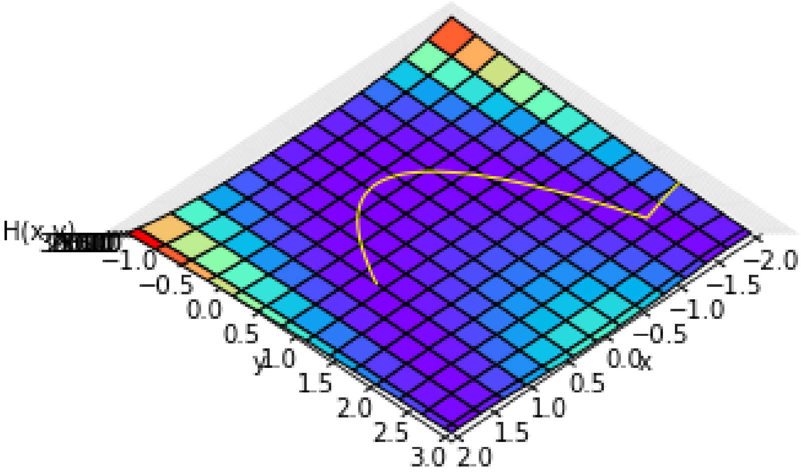

Figure 11: Trajectory of gradient descent with $\eta = 0.00001$.

Exercise 3

1. Initial probability of grabbing an apple:

   $P(F_1 == Apple) =$

   $P(B == 1) * P(F_1 == Apple|B == 1) + P(B == 2) * P(F_1 == Apple|B == 2)$

   $= \frac{1}{2} * \frac{8}{12} + \frac{1}{2} * \frac{15}{18}$

   $= \frac{4}{12} + \frac{15}{36} = \frac{27}{36} = \frac{3}{4}$

   Initial probability of grabbing a grapefruit:

   $P(F_2 == Grapefruit) =$

   $P(B == 1) * P(F_2 == Grapefruit|B == 1) + P(B == 2) * P(F_2 == Grapefruit|B == 2)$

   $= \frac{1}{2} * \frac{4}{12} + \frac{1}{2} * \frac{3}{18}$

   $= \frac{2}{12} + \frac{3}{36} = \frac{1}{4}$

Probability of having grabbed from box 1 given that you grabbed a grapefruit:
$P(B == 1|F_2 == Grapefruit) = \frac{P(F_2 == Grapefruit|B == 1) * P(B == 1)}{P(F_2 == Grapefruit)}$
$= \frac{\frac{4}{12} * \frac{1}{2}}{\frac{1}{4}}$
$= \frac{\frac{2}{12}}{\frac{1}{4}} = \frac{8}{12} = \frac{2}{3}$

Probability of having grabbed from box 2 given that you grabbed a grapefruit:
$P(B == 2|F_2 == Grapefruit) = \frac{P(F_2 == Grapefruit|B == 2) * P(B == 2)}{P(F_2 == Grapefruit)}$
$= \frac{\frac{3}{18} * \frac{1}{2}}{\frac{1}{4}}$
$= \frac{\frac{3}{36}}{\frac{1}{4}} = \frac{12}{36} = \frac{1}{3}$

Probability of grabbing an apple but with updated probabilities for the boxes since these are given having grabbed a grapefruit:
$P(F_1 == Apple) =$
$P(B == 1) * P(F_1 == Apple|B == 1) + P(B == 2) * P(F_1 == Apple|B == 2)$
$= \frac{2}{3} * \frac{8}{12} + \frac{1}{3} * \frac{15}{18}$
$= \frac{16}{36} + \frac{5}{18} = \frac{13}{18}$

The probability for grabbing an apple is different when you know that you grabbed a grapefruit from the same box, since knowing you grabbed a grapefruit changes the probability of the box you grabbed from, this is not 50-50 anymore. Since the probabilities of grabbing an apple are different for the two boxes, knowing more about which box you grabbed from, changes the probability of grabbing an apple.

2. Initial probability of grabbing a grapefruit:
$P(F_2 == Grapefruit) =$
$P(B == 1) * P(F_2 == Grapefruit|B == 1) + P(B == 2) * P(F_2 == Grapefruit|B == 2)$
$= \frac{1}{2} * \frac{1}{6} + \frac{1}{2} * \frac{1}{6} = \frac{1}{6}$

Probabilities of having grabbed from a certain box, given that you grabbed a grapefruit:
$P(B == 1|F_2 == Grapefruit) = \frac{P(F_2 == Grapefruit|B == 1) * P(B == 1)}{P(F_2 == Grapefruit)} = \frac{\frac{1}{6} * \frac{1}{2}}{\frac{1}{6}} = \frac{1}{2}$
$P(B == 2|F_2 == Grapefruit) = P(B == 1|F_2 == Grapefruit) = \frac{1}{2}$

Since again the probability for the boxes is both $\frac{1}{2}$, the probability for grabbing an apple remains the same as the initial probability:
$P(F_1 == Apple) = P(F_1 == Apple)$ initially $= \frac{1}{4}$

The probability of grabbing an apple is not different when you know that you have grabbed a grapefruit, because having grabbed a grapefruit does not give you more knowledge about which box you grabbed from, both are still $P = 0.5$. When you do not have more knowledge about which box you grabbed from, you cannot use the probability for grabbing an apple, given that you know the box, so it is the same as the prior probability for grabbing an apple.

However, this does not mean that the first and second piece of fruit you grab are independent. In the case of grabbing a grapefruit, it seems independent, because in box 1 there are 24 pieces of fruit, of which 4 grapefruits, and thus $P = \frac{1}{6}$, and in box 2 there are 18 pieces of fruit, of which 3 grapefruits, thus also $P = \frac{1}{6}$. But, when you grabbed an orange instead of a grapefruit, you know with 100% certainty that you grabbed from box 1 since there are no oranges in box 2. When you grabbed an apple, this also makes one box more likely than the other, because the distribution over the boxes is also different for apples. Thus, the two picks are still dependent.