MTU

Ollscoil Teicneolaíochta na Mumhan

Munster Technological University

Machine Learning

Lecture 19: Parameter estimation (example)

Parameter estimation

- Last time we looked at an iterative algorithm for estimating the parameters θ for a known model function

$$y = f[x; \theta]$$

- From a training set of pairs of model inputs and outputs

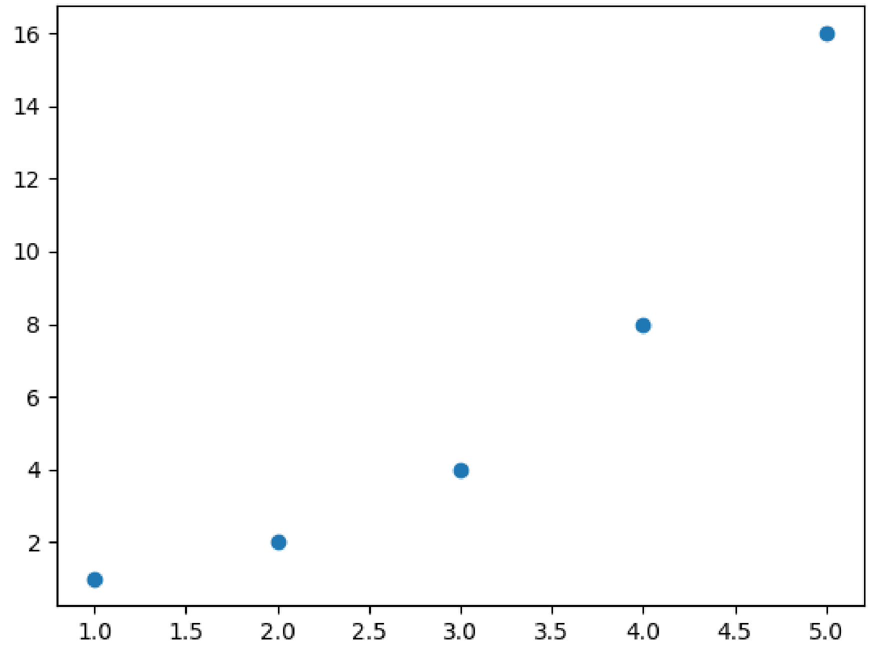
$$\{x_1 \rightarrow y_1, \dots, x_n \rightarrow y_n\}$$

Training data

- In the first example we will work with the training data

$$\{1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 4, 4 \rightarrow 8, 5 \rightarrow 16\}$$

```
data = np.array([1,2,3,4,5])
target = np.array([1,2,4,8,16])
```



Model function

- As model function we choose to use polynomials, i.e.

$$f[x;\theta] = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^2$$

- This model function can be implemented in Python as

```
def calculate_model_function(data, p):
    result = np.zeros(len(data))
    for i in range(len(p)):
        result += p[i]*(data**i)
    return result
```

The degree of the polynomial can be determined from the length of the parameter vector

For example:
- data=[1 2 3 4 5]
- p=[3,2,1]
- result=
  [6. 11. 18. 27. 38.]

Linearization

- Although we could linearize the simple polynomials analytically, we use the black-box linearization procedure

$$\frac{\partial f}{\partial \theta_k} = \lim_{\epsilon \to 0} \frac{f[x;\theta_1,\dots,\theta_k+\epsilon,\dots,\theta_m] - f[x;\theta]}{\epsilon}$$

- This looks in Python as follows

```
def linearize(data, p0):
    f0 = calculate_model_function(data,p0)
    J = np.zeros((len(f0), len(p0)))
    epsilon = 1e-6
    for i in range(len(p0)):
        p0[i] += epsilon
        fi = calculate_model_function(data,p0)
        p0[i] -= epsilon
        di = (fi - f0)/epsilon
        J[:,i] = di
    return f0,J
```

First we calculate the model function at the linearization point p0

We iterate through all components of the parameter p0

And calculate the model function again, adding a small perturbation to the component of the parameter vector

The element of the Jacobian matrix is the difference divided by the perturbation

Linearization

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]
- i=0

```
def linearize(data, p0):
    f0 = calculate_model_function(data,p0)
    J = np.zeros((len(f0), len(p0)))
    epsilon = 1e-6
    for i in range(len(p0)):
        p0[i] += epsilon
        fi = calculate_model_function(data,p0)
        p0[i] -= epsilon
        di = (fi - f0)/epsilon
        J[:,i] = di
    return f0,J
```

```
f0 = [0. 0. 0. 0. 0.]
J =
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
p0 = [1.e-06 0.e+00 0.e+00]
```

```
fi = [1.e-06 1.e-06 1.e-06 1.e-06 1.e-06]
```

```
di = [1.e-06 1.e-06 1.e-06 1.e-06 1.e-06]
```

```
J =
 [[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]]
```

Linearization

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]
- i=1

```
def linearize(data, p0):
    f0 = calculate_model_function(data,p0)
    J = np.zeros((len(f0), len(p0)))
    epsilon = 1e-6
    for i in range(len(p0)):
        p0[i] += epsilon
        fi = calculate_model_function(data,p0)
        p0[i] -= epsilon
        di = (fi - f0)/epsilon
        J[:,i] = di
    return f0,J
```

```
f0 = [0. 0. 0. 0. 0.]
J =
 [[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]]
```

```
p0 = [0.e+00 1.e-06 0.e+00]
```

```
fi = [1.e-06 2.e-06 3.e-06 4.e-06 5.e-06]
```

```
di = [1.e-06 2.e-06 3.e-06 4.e-06 5.e-06]
```

```
J =
 [[1. 1. 0.]
 [1. 2. 0.]
 [1. 3. 0.]
 [1. 4. 0.]
 [1. 5. 0.]]
```

Linearization

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]
- i=2

```
def linearize(data, p0):
    f0 = calculate_model_function(data,p0)
    J = np.zeros((len(f0), len(p0)))
    epsilon = 1e-6
    for i in range(len(p0)):
        p0[i] += epsilon
        fi = calculate_model_function(data,p0)
        p0[i] -= epsilon
        di = (fi - f0)/epsilon
        J[:,i] = di
    return f0,J
```

```
f0 = [0. 0. 0. 0. 0.]
J =
 [[1. 1. 0.]
 [1. 2. 0.]
 [1. 3. 0.]
 [1. 4. 0.]
 [1. 5. 0.]]
```

```
p0 = [0.e+00 0.e+00 1.e-06]
```

```
fi = [1.e-06 2.e-06 3.e-06 4.e-06 5.e-06]
```

```
di = [1.0e-06 4.0e-06 9.0e-06 1.6e-05 2.5e-05]
```

```
J =
 [[ 1.  1.  1.]
 [ 1.  2.  4.]
 [ 1.  3.  9.]
 [ 1.  4. 16.]
 [ 1.  5. 25.]]
```

Linearization

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]
- The Jacobian J collects all derivatives of f w.r.t. θ for all values of x

```
def linearize(data, p0):
    f0 = calculate_model_function(data,p0)
    J = np.zeros((len(f0), len(p0)))
    epsilon = 1e-6
    for i in range(len(p0)):
        p0[i] += epsilon
        fi = calculate_model_function(data,p0)
        p0[i] -= epsilon
        di = (fi - f0)/epsilon
        J[:,i] = di
    return f0,J
```

```
J =
 [[ 1.  1.  1.]
 [ 1.  2.  4.]
 [ 1.  3.  9.]
 [ 1.  4. 16.]
 [ 1.  5. 25.]]
```

$$f[x;\theta] = \theta_0 + \theta_1 x + \theta_2 x^2$$

$$\frac{\partial f[x;\theta]}{\partial \theta_1} = 1$$

$$\frac{\partial f[x;\theta]}{\partial \theta_2} = x$$

$$\frac{\partial f[x;\theta]}{\partial \theta_3} = x^2$$

x	$\frac{\partial f}{\partial \theta_1}$	$\frac{\partial f}{\partial \theta_2}$	$\frac{\partial f}{\partial \theta_3}$
1	1	1	1
2	1	2	4
3	1	3	9
4	1	4	16
5	1	5	25

Update step

- The update step was to calculate

$$\Delta\theta = \left(\sum_i J_i^T J_i + \lambda I\right)^{-1} \sum_i J_i^T (y_i - f_{0_i})$$

- This can be implemented in Python as

```
def calculate_update(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    n = np.matmul(J.T,r)
    dp = np.linalg.solve(N,n)
    return dp
```

We multiply $J^T J$

Then we add the regularisation matrix λI, which needs to have the same size a $J^T J$

The residual is r $= y - f\_0$

We create the right-hand-side of the normal equation system $n = J^T r$

Finally, we solve for $\Delta\theta$ (Solve is faster than matrix inversion)

Update step

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]

```
def calculate_update(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    n = np.matmul(J.T,r)
    dp = np.linalg.solve(N,n)
    return dp
```

```
J =
 [[ 1.  1.  1.]
 [ 1.  2.  4.]
 [ 1.  3.  9.]
 [ 1.  4. 16.]
 [ 1.  5. 25.]]
```

```
J.T =
 [[ 1.  1.  1.  1.  1.]
 [ 1.  2.  3.  4.  5.]
 [ 1.  4.  9. 16. 25.]]
```

```
np.matmul(J.T,J) =
 [[  5.  15.  55.]
 [ 15.  55. 225.]
 [ 55. 225. 979.]]
```

The dimensions are the number of parameters (3) and the number of data points (5):

$$J: \quad 3 \times 5$$
$$J^T: \quad 5 \times 3$$

The size of the result only depends on the number of parameters (3):

$$J^TJ: \ 3 \times 3$$

Update step

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]

```
def calculate_update(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    n = np.matmul(J.T,r)
    dp = np.linalg.solve(N,n)
    return dp
```

```
J =
 [[ 1.  1.  1.]
 [ 1.  2.  4.]
 [ 1.  3.  9.]
 [ 1.  4. 16.]
 [ 1.  5. 25.]]
```

```
l*np.eye(J.shape[1])
=
 [[0.01 0.   0.  ]
 [0.   0.01 0.  ]
 [0.   0.   0.01]]
```

Each diagonal element of the regularisation matrix is the weight of the constraint for that particular component to stay unchanged

Update step

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]

```
def calculate_update(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    n = np.matmul(J.T,r)
    dp = np.linalg.solve(N,n)
    return dp
```

```
y = [ 1  2  4  8 16]
f0 = [0. 0. 0. 0. 0.]
r = [ 1.  2.  4.  8. 16.]
```

The size of the residual vector is the same as the model function output and the target vector

It contains for every training sample how it does not comply with the model function

Update step

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]

```
def calculate_update(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    n = np.matmul(J.T,r)
    dp = np.linalg.solve(N,n)
    return dp
```

```
J.T =
 [[ 1.  1.  1.  1.  1.]
 [ 1.  2.  3.  4.  5.]
 [ 1.  4.  9. 16. 25.]]
```

```
r = [ 1.  2.  4.  8. 16.]
```

```
n =  [ 31. 129. 573.]
```

- The size of the residual vector is the number of data points
- The size of the n-vector is the number of parameters

- The multiplication with J^T aggregates the contributions of each residual to the parameter vector update, so its size is 3

Update step

- We run this for data=[1 2 3 4 5] and p0=[0,0,0]

```
def calculate_update(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    n = np.matmul(J.T,r)
    dp = np.linalg.solve(N,n)
    return dp
```

```
N =
[[  5.01  15.    55.  ]
 [ 15.    55.01 225.  ]
 [ 55.   225.   979.01]]
```

```
n =  [ 31. 129. 573.]
```

```
dp =  [ 3.14841094 -3.06655374  1.11317759]
```

The final step is the solution of the normal equation system for determining the parameter update

Obviously, its size is the size of the parameter vector (3)

Complete procedure

- The complete estimation procedure is to start from an initial parameter vector θ_0 and iteratively update the parameter vector
$$\theta_0' = \theta_0 + \Delta\theta$$

- This can be implemented in Python as follows

```
max_iter = 10
deg = 2
p0 = np.zeros(deg+1)
for i in range(max_iter):
    f0,J = linearize(data, p0)
    dp = calculate_update(target,f0,J)
    p0 += dp
```

The desired degree of the model polynomial determines the length of the parameter vector

For lack of a better guess, we start with $\theta_0 = 0$

We compute the model function and the Jacobian at the current linearization point θ_0

This enables the computation of the parameter update $\Delta\theta$

Finally we update the current linearization point θ_0 and repeat

Complete procedure

- We run this procedure and see how the parameter vector evolved

```
max_iter = 10
deg = 2
p0 = np.zeros(deg+1)
for i in range(max_iter):
    f0,J = linearize(data, p0)
    dp = calculate_update(target,f0,J)
    p0 += dp
```

```
p0 = [0. 0. 0.]

dp = [ 3.14841094 -3.06655374  1.11317759]
p0 = [ 3.14841094 -3.06655374  1.11317759]

dp = [ 0.23477772 -0.17796834  0.02772314]
p0 = [ 3.38318865 -3.24452208  1.14090073]

dp = [ 0.01569165 -0.01178042  0.00182616]
p0 = [ 3.3988803  -3.2563025   1.14272689]

dp = [ 0.00104513 -0.00078439  0.00012158]
p0 = [ 3.39992543 -3.25708689  1.14284847]

dp = [ 6.95994095e-05 -5.22352798e-05  8.09612384e-06]
p0 = [ 3.39999503 -3.25713913  1.14285656]
```

The first update is the largest

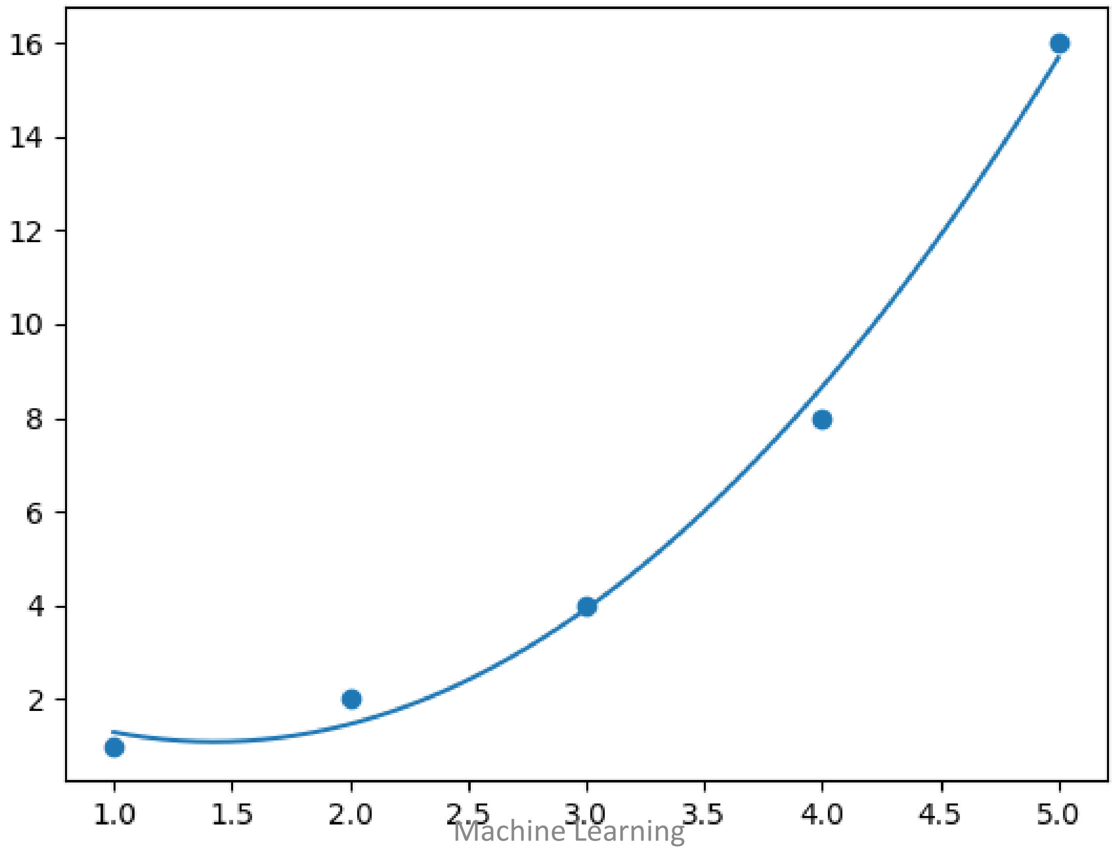
Subsequent updates should get smaller, depending on the search along the non-linear model function surface

When a local minimum is reached, updates will become very small

At the end we converege to the solution vector

Final result

- The result was $\theta = [3.4, -3.3, 1.1]$
- We can plot the model function $f[x] = 3.4 - 3.3x + 1.1x^2$



Accuracy

- To calculate the accuracy of the result we need to first estimate the variance factor

$$\hat{\sigma}_0^2 = \frac{1}{N-U}\sum_i r_i^T r_i$$

- Then we can calculate the covariance matrix

$$\hat{\Sigma}_{\hat{\theta}\hat{\theta}} = \hat{\sigma}_0^2 \left(\sum_i J_i^T J_i\right)^{-1}$$

- This can be implemented in Python as follows:

```
def calculate_covariance(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    sigma0_squared = np.matmul(r.T,r)/(J.shape[0]-J.shape[1])
    cov = sigma0_squared * np.linalg.inv(N)
    return cov
```

First we calculate the normal equation matrix and residuals as before (we can also use the results from the last iteration)

The variance factor is calculated from the squared residuals

The covariance matrix of the parameter vector is proportional to the inverse of N

Accuracy

- The final result was $\theta = [3.4, -3.3, 1.1]$, so we calculate the covariance matrix

```
def calculate_covariance(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    sigma0_squared = np.matmul(r.T,r)/(J.shape[0]-J.shape[1])
    cov = sigma0_squared * np.linalg.inv(N)
    return cov
```

```
y = [ 1  2  4  8 16]
f0 = [ 1.28571428  1.45714286  3.91428571  8.65714286 15.68571429]
r = [-0.28571428  0.54285714  0.08571429 -0.65714286  0.31428571]
```

```
J.shape[0] = 5
J.shape[1] = 3
```

```
np.matmul(r.T,r) = 0.9142857142857128
```

```
sigma0_squared = 0.4571428571428564
```

```
cov =
 [[ 1.96502892 -1.40527357  0.21257185]
 [-1.40527357  1.14351432 -0.18385989]
 [ 0.21257185 -0.18385989  0.03078025]]
```

Accuracy

- The final result was $\theta = [3.4, -3.3, 1.1]$, so we calculate the covariance matrix

The square-roots of the diagonal elements indicate how certain we are with the parameters:

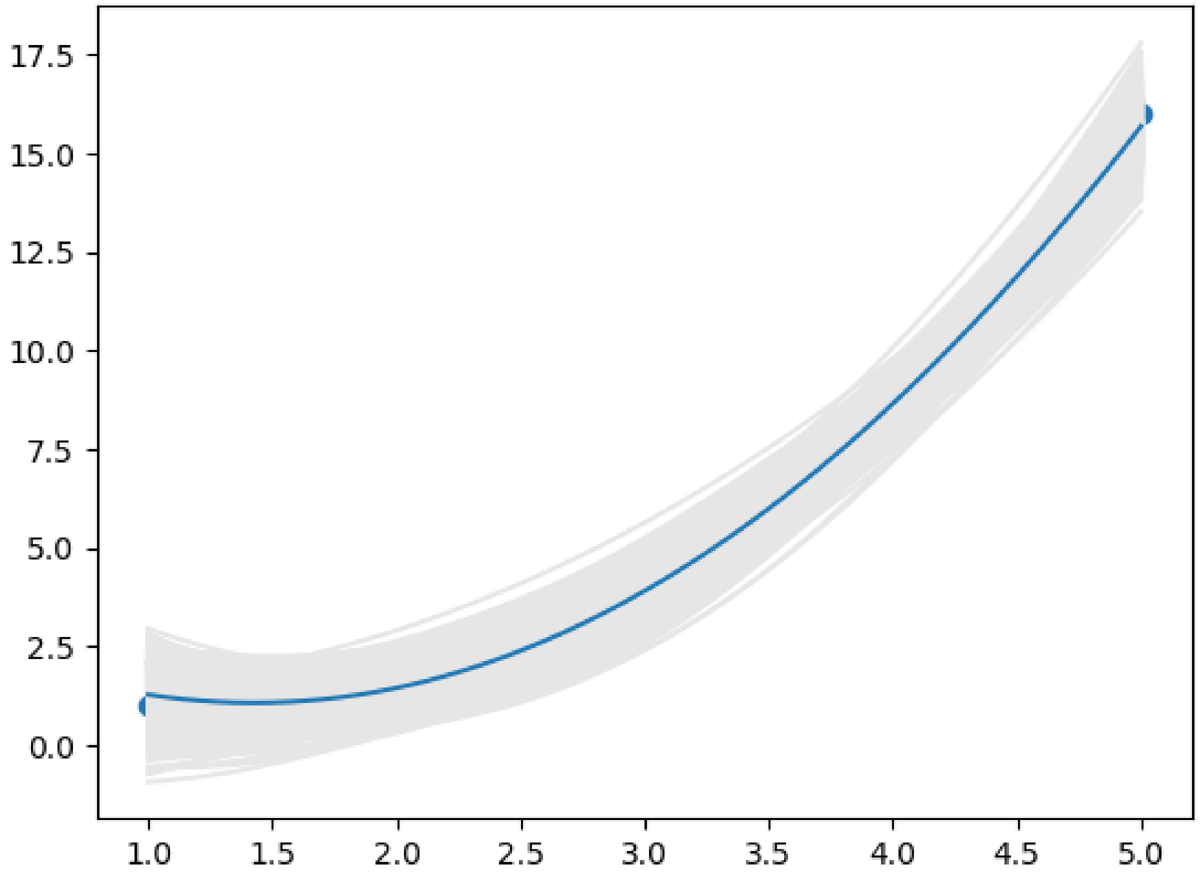
```
p[0]= 3.4000000035266265  ± 1.4017948921754533
p[1]= -3.25714285991827  ± 1.0693522891938485
p[2]= 1.1428571432966015  ± 0.17544299648781383
```

```python
def calculate_covariance(y,f0,J):
    l=1e-2
    N = np.matmul(J.T,J) + l*np.eye(J.shape[1])
    r = y-f0
    sigma0_squared = np.matmul(r.T,r)/(J.shape[0]-J.shape[1])
    cov = sigma0_squared * np.linalg.inv(N)
    return cov
```

```
cov =
 [[ 1.96502892 -1.40527357  0.21257185]
 [-1.40527357  1.14351432 -0.18385989]
 [ 0.21257185 -0.18385989  0.03078025]]
```

Accuracy visualisation

- We can visualise the accuracy by plotting different functions and adding noise to the parameters according to the covariance matrix we estimated



Polynomial model in 2D

- If our training data is 2-dimensional like this random points

```
n = 20
data = 10*np.random.rand(n,2)
target = np.random.rand(n)
```

- The only change required is the model function

$$f[x;\theta] = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2 + \cdots$$

```
p0 = np.zeros(int((deg+2)*(deg+1)/2))
def calculate_model_function(data, p):
    result = np.zeros(data.shape[0])
    deg = int(-(3./2.) + math.sqrt(3.*3./4 - 2 + 2*len(p)))
    k=0
    for n in range(deg+1):
        for i in range(n+1):
            result += p[k]*(data[:,0]**i)*(data[:,1]**(n-i))
            k+=1
    return result
```

Polynomial model in 2D

The parameter vector needs to be larger to accommodate all coefficients,

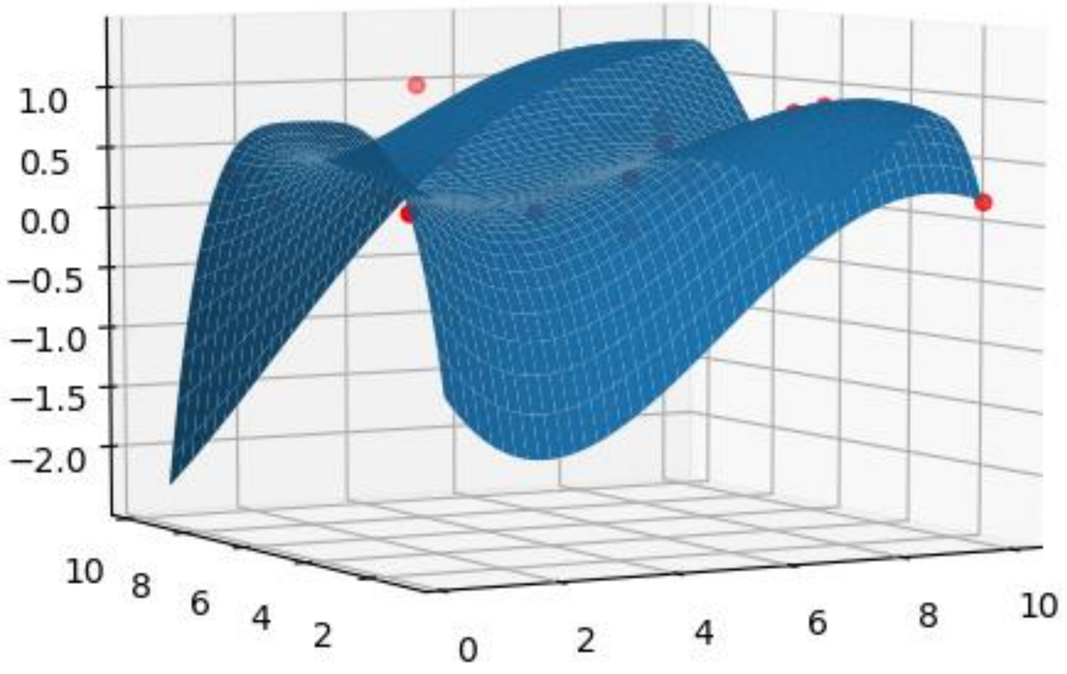
We can invert this formula, but we could also simply pass the degree of the polynomial

```
p0 = np.zeros(int((deg+2)*(deg+1)/2))
def calculate_model_function(data, p):
    result = np.zeros(data.shape[0])
    deg = int(-(3./2.) + math.sqrt(3.*3./4 - 2 + 2*len(p)))
    k=0
    for n in range(deg+1):
        for i in range(n+1):
            result += p[k]*(data[:,0]**i)*(data[:,1]**(n-i))
            k+=1
    return result
```

Instead of a one dimensional data vector, we need to process a 2d data vector

Final result for 2D random points

- The resulting function for 2d input data can still be plotted
- For higher dimensions this is no longer possible



Thank you for your attention