Python Programming for Finance

Marius A. Zoican, PhD.

Lecture 4:
Linear regression. Optimization. Symbolic math.

Outline

Motivation

1. Most of the times in finance, we do not know the DGP (Data Generating Process).
2. Many applications in finance involve “reverse engineering” patterns from data.
3. This is useful, for example to make predictions about the future dynamics of financial variables.
4. Two main techniques:
   - 4.1 Regression
   - 4.2 Interpolation

First, define a function (the DGP)..

We specifically choose a non-polynomial function (more difficult).

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.sin(x)+0.5*x
```

Next, generate data from the DGP

We generate 50 data-points from the DGP: $(x, f(x))$.

- Function `x=np.linspace(a, b, N)` returns an array of N numbers, equally spaced, from *a* to *b*.
- What does $f(x)$ return?

```
x=np.linspace(-2*np.pi, 2*np.pi, 50)

plt.plot(x, f(x), 'b')
plt.grid()
plt.xlabel('x', fontsize=18)
plt.ylabel('y', fontsize=18)
plt.show()
```

Regression

Theoretical framework:

1. You are given N points in a 2-D (can be 3-D, 4-D...) space: (x_j, y_j).
2. You choose K (base) functions of x_j, i.e., $b_i(x_j)$, such that you believe y_j can be written as a linear combination of these functions.
3. You select coefficients of said linear combinations, α_i by minimising the squared difference from the actual data.

$$\min_{\alpha_i} \frac{1}{N} \sum_{j=1}^{N} \left(y_j - \sum_{i=1}^{K} \alpha_i b_i(x_j) \right)^2 \tag{1}$$

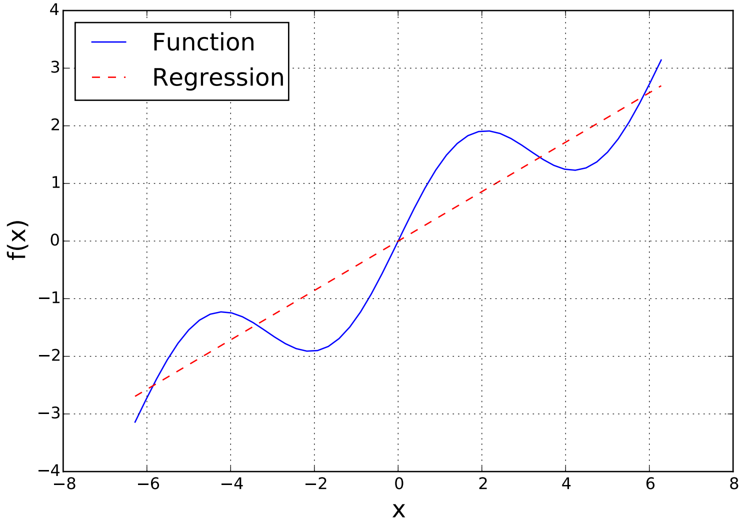
Polynomial regression

A simple case is to approximate y_j as a polynomial function of x_j.
That is, choose: $b_1 = 1$, $b_1 = x$, $b_2 = x^2$, ... , $b_k = x^k$.
Easy to implement in Python with `polyfit` (polynomial fit):

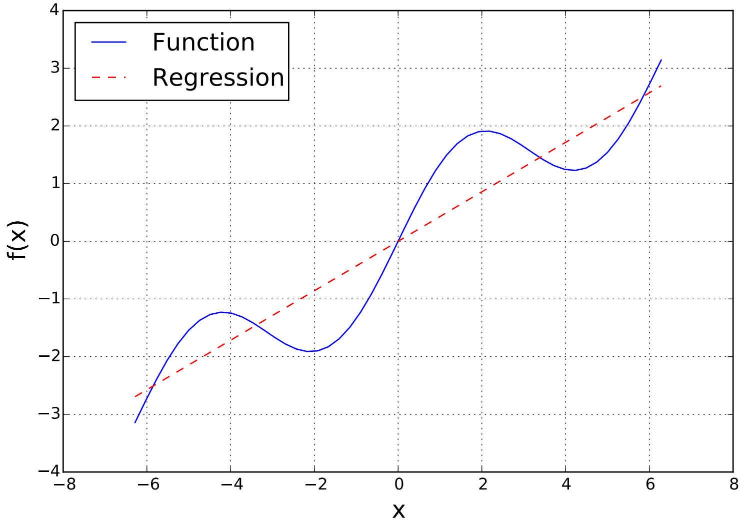
1. First, get the coefficient list using `polyfit`.
2. Next, get the fitted values from the coefficient list using `polyval`.

```
reg=np.polyfit(x, f(x), deg=k)
y_fit=np.polyval(reg,x)
```

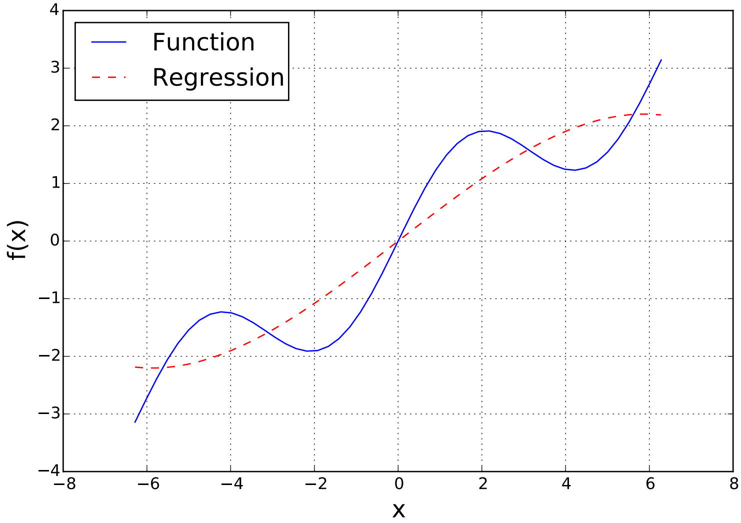
What happens if we vary the polynomial degree?

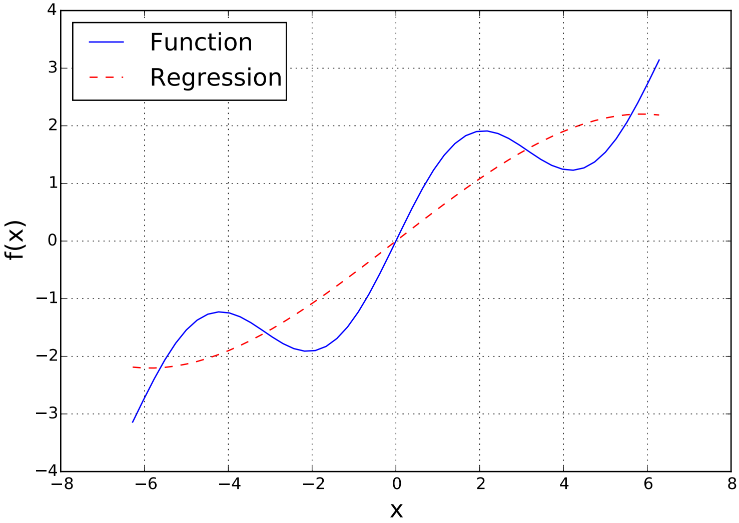


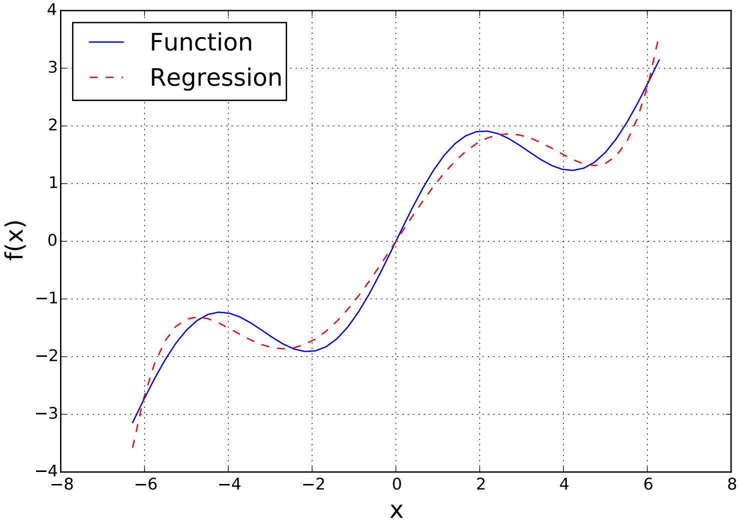
Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8

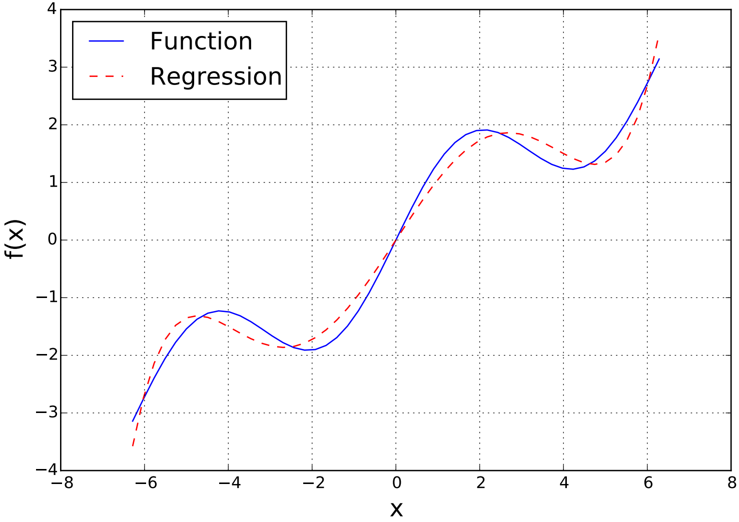


Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8

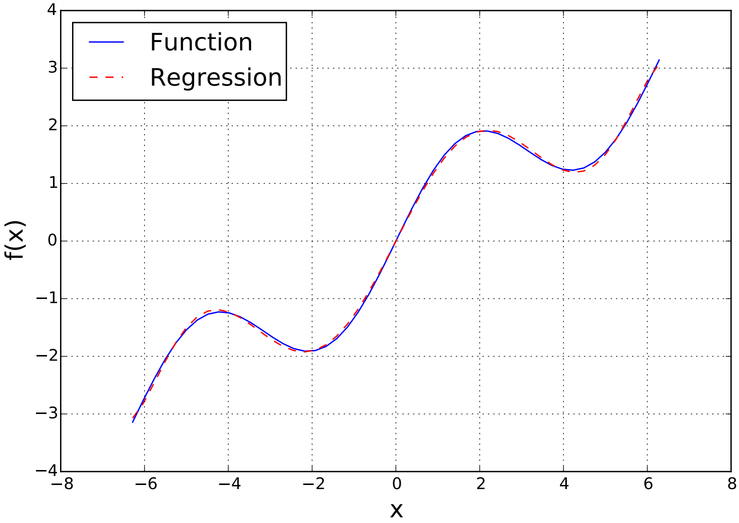

Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8


Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8


Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8



Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8



Function
Regression
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8

Beyond polynomials

- The mean squared error of our fit is not zero....rather 1.77×10^{-3}.
- Not surprising, since the original function was not a polynomial.
- How can we approximate it using other base functions, i.e., trigonometric?

1. Say we know (prior theoretical work) our function is a combination of a second order polynomial and sin/cos functions.
2. Let us define a matrix with values for 1, x, x^2, $\sin(x)$, $\cos(x)$

Formalization of the problem

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ .. \\ y_N \end{pmatrix} = \underbrace{\begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ .. \\ \alpha_N \end{pmatrix}}_{\text{Coefficients}} \underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 & \sin(x_1) & \cos(x_1) \\ 1 & x_2 & x_2^2 & \sin(x_2) & \cos(x_2) \\ 1 & x_3 & x_3^2 & \sin(x_3) & \cos(x_3) \\ .. & .. & .. & .. & .. \\ 1 & x_N & x_N^2 & \sin(x_N) & \cos(x_N) \end{pmatrix}}_{\text{Matrix M}} + \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ .. \\ u_N \end{pmatrix}$$

Solving the problem in Python

- Initialize the matrix M:

```
M=np.zeros((len(x),5))

```

- Fill in each column with a variable:

```
M[:,0]=1
M[:,1]=x
M[:,2]=x**2
M[:,3]=np.sin(x)
M[:,4]=np.cos(x)

```

Solving the problem in Python (2)

We use `numpy.linalg.lstsq` to minimise the sum of squared residuals.
Least-square coefficients are given by:

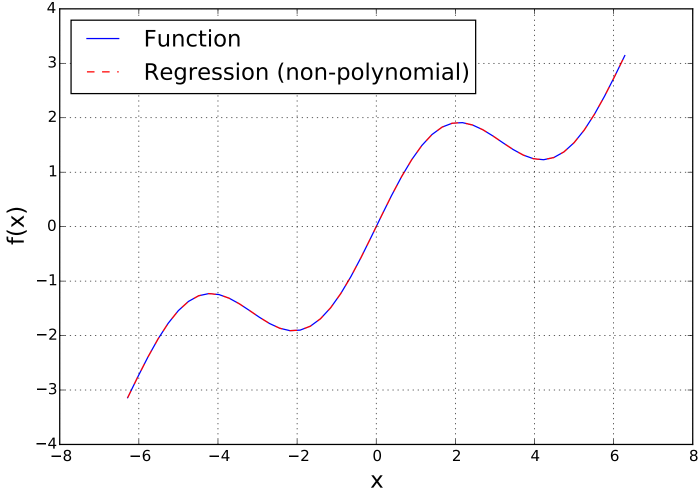
```
reg=np.linalg.lstsq(M, f(x))[0]
```

The fitted-values are computed as a dot-product between the coefficients vector (`reg`) and the matrix M:

```
y_fit2=np.dot(reg,M.T)
# we need to transpose the matrix
```

1. What are the coefficients in reg?
2. What it the MSE?

Output



Function
Regression (non-polynomial)
f(x)
x
4
3
2
1
0
−1
−2
−3
−4
−8
−6
−4
−2
0
2
4
6
8

General idea

1. With *regression*, one tries to identify a **unique** function $g(x)$ that is as close as possible to the "true", unknown function $f(x)$, i.e.,
$$\min \sum \left(g(x) - f(x)\right)^2$$
2. With *interpolation*, one fits more (generally polynomial) functions, one between **each pair of consecutive points**.
   - The fit is perfect, i.e., $\forall i$, $g_i(x_i) = f(x_i)$.
   - The function is not unique, which is mathematically involved.
   - The function is constrained to be continuous, $g_i(x_i) = g_{i+1}(x_i)$.
   - Some additional constraint is needed, i.e., second derivatives are continuous.
3. One needs ordered data in interpolation (unlike regression).
4. Procedure takes more time and is less parsimonious (more coefficients in the end) – but generally more accurate.

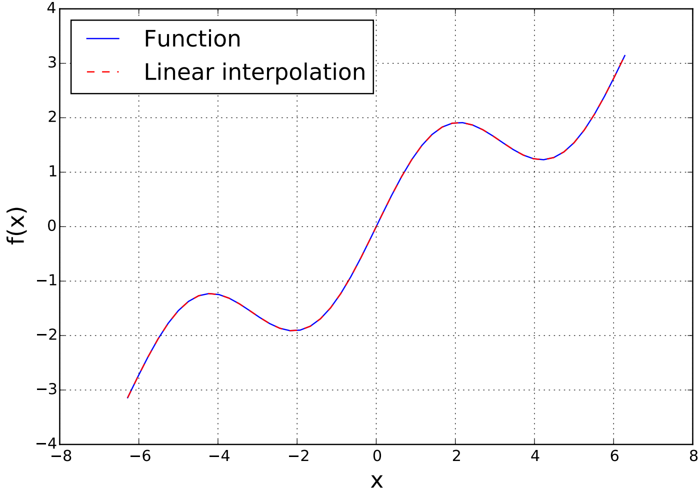
Implementation

The interpolation package is in the Scientific Python library (`scipy`). The parameter k defines the degree of the polinomial ($k = 1$ is a linear spline, $k = 3$ a cubic spline...)

```python
import scipy.interpolate as spi
interp=spi.splrep(x,f(x), k=1)
y_interp=spi.splev(x,interp)
```

1. What type of object is `interp` relative to `reg`? Why?
2. How good is the linear interpolation?

Interpolation output



Outline

Main idea

We want to minimize a function $f\left(x_1, x_2, x_3, \ldots x_n\right)$:

$$\min_{x_i} f\left(x_1, x_2, x_3, \ldots x_n\right) \tag{2}$$

All local extrema satisfy

$$\frac{\partial f}{\partial x_i} = 0, \forall i \in \left\{1, 2, \ldots n\right\}. \tag{3}$$

The global minimum/maximum (if it exists and/or is unique) is either one of the local extrema or one of the domain end-points (see whiteboard).

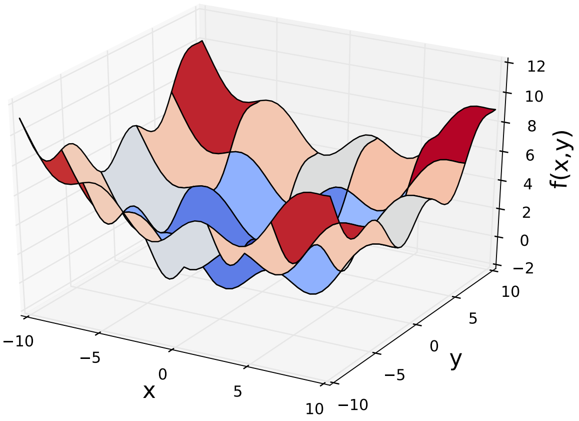
More?

The Weierstrass (extreme value) theorem guarantees the existence of a maximum and minimum on closed and bounded intervals.

A two dimensional function

First, we define a function to minimize

```
def fm((x,y)):
    return np.sin(x)+1/20.0*x**2
      +np.sin(y)+1/20.0*y**2
```


12
10
8
6
4
2
0
−2
f(x,y)
−10
−5
0
5
10
x
10
5
0
−5
−10
y

Brute force optimization (the "caveman" approach)

```
import scipy.optimize as spo
```

Define a range and step to search for minimum:

```
search_area=(-10,10.01,5)
```

Change the function to print all iterations and output:

```
def fm((x,y)):
    z=np.sin(x)+1/20.0*x**2
      +np.sin(y)+1/20.0*y**2
    print '%8.4f␣%8.4f␣%8.4f' %(x,y,z)
    return z
```

Run the function brute (force) to find the minimum:

```
min_1=spo.brute(fm, (search_area,search_area),
    finish=None)
```

1. What is the minimum found by this method?
2. How can we improve the accuracy? What is the drawback?

The brute force method, while limited, can serve to provide starting values for more sophisticated algorithms.
One such function, working with numerical gradients, is `fmin`.

Optimization with `fmin`

General structure:

```
[xopt, fopt]=spo.fmin(function, start_values,
  xtol=, ftol=, maxiter=, maxfun=,)
```

1. xtol : Relative error in argument acceptable for convergence.
2. ftol: Relative error in function acceptable for convergence.
3. maxiter : Maximum number of iterations to perform.
4. maxfun : Maximum number of function evaluations to make.

We can use the global optimization results as starting values:

```
min_2=spo.fmin(fm, min_1, xtol=0.001, ftol=0.001)
```

Caveats

- Local optimization routines can get stuck in local extrema...
- ... or they may never converge.
- It is a good idea to perform a global optimization first to pinpoint the neighbourhood of global minimum.
- What happens if we start `fmin` with $(2, 2)$ as starting values?

Constrained optimization

Most of the time, we look for optimal values of a function **under a set of constraints**.

Problem

There are two securities, A and B: Both cost 10 today. Tomorrow there are two equally likely states of the world: g or b. In state g, $A = 15$ and $B = 5$. In state b, $A = 5$ and $B = 12$. Assume an investor has 100 units of cash today and utility $u(w) = \sqrt{w}$. What is his optimal investment?

Formalization of the problem

$$\max_{a,b} \mathbb{E} u\left(w_1\right) = \max_{a,b} \frac{1}{2}\sqrt{15a+5b} + \frac{1}{2}\sqrt{5a+12b}, \tag{4}$$

subject to:

$$10a + 10b \leq 100. \tag{5}$$

Python implementation

First, define the function. Note: we want to **maximize** expected utility!

```
def ExpU((s,b)):
    return -(0.5*np.sqrt(s*15+b*5)+
    0.5*np.sqrt(s*5+b*12))
```

Second, define the constraint as a `dict` variable and an implicit function.Inequality sign is always implicitly "≥ 0".

```
cons=({'type':'ineq', 'fun':
   lambda (s,b): 100-s*10-b*10})
```

Third, choose starting values:

```
startval=[5,5]
```

Fourth, run the `minimize` function from the optimization package:

```
result=spo.minimize(ExpU, startval,
   method='SLSQP', constraints=cons)
```

Notes

- `method` stands for optimization algorithm. SLSQP (Sequential Least SQuares Programming) allows one to introduce constraints.
- One can specify Jacobian (`jac`) or Hessian matrix (`hess`) directly.
- In addition, bounds for the argument can be specified by `bounds`.

Output methods:

1. `result.fun` returns the optimum function values.
2. `result.x` returns the arguments corresponding to the optimum.
3. `result.success` returns True if optimization complete.

Numerical integration

Numerical integration is done via the `scipy.integrate` package.

```
import scipy.integrate as integr
```

There are several methods to numerically integrate a function (say $f(x) = \sin x + \frac{x}{2}$); fixed Gaussian quadrature, adaptive quadrature, Romberg integration....
All are approximations of the same thing, though...

```
integr.fixed_quad(f, lmin, lmax)[0]
integr.quad(f, lmin, lmax)[0]
integr.romberg(f, lmin, lmax)[0]
```

Outline

Basics

1. Import the Symbolic Python `sympy` library.

```python
import sympy as sy
```

2. Create new “symbol” type objects:

```python
x=sy.Symbol('x')
y=sy.Symbol('y')
```

3. Just like `numpy`, `sympy` has a number of available functions: `sy.log, sy.sin, sy.cos, sy.sqrt....`
4. One can easily simplify functions:

```python
f=x**2+0.5*x**2+3-2
sy.simplify(f)
```

Why is it useful?

Solving equations (including imaginary solutions)

```
sy.solve(x**2-4)
sy.solve(x**2-y**2)
sy.solve(x**3+3*x**2+3*x+1)
sy.solve({x-y=1, x+y=2})
```

Symbolic integration

```
sy.integrate(sy.log(x)+sy.sin(x))
```

Symbolic differentiation

```
f=x**3+x-sy.exp(y)
sy.diff(f,x)=?
sy.diff(f,y)=?
```

Why is it useful? (2)

One can combine all of the above to, say, solve a first-order condition:

```
x_zero=sy.solve(sy.diff(f,x))
```

One can numerically evaluate functions at certain parameters using `evalf`:

```
f.subs({x:2, y:0}).evalf()
```

Application

Optimize the function

$$\frac{x^2}{2} + x\,(y - 1) + y \tag{6}$$

both numerically and solving the first order conditions in `sympy`.