1D_dataset_report

Creating testing dataset

Here we are going to implement **moving-least squares algorithm (MLS)** to fit testing dataset (described by function $f1(x)$ below) in one dimension. Later, we try to compare its result with **ordinary-least-squares (OLS)** fit.

$$f_1(x) = \sin(2\pi x) + 0.5\cos\left(6\pi x + \frac{\pi}{4}\right), \quad x \in [0,1],$$

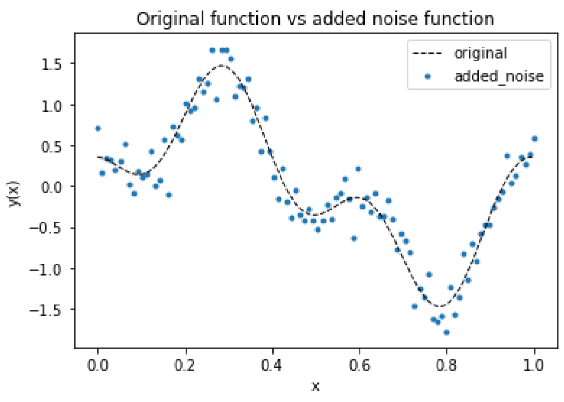
First step would be creating data, that will provide "learning material" for our model to estimate coefficients in later steps. In following block of code, noise is added to underlying function to simulate measuring our observations.

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


#original fucntion -> N values
x = np.linspace(0, 1, 100) # "observations" [1 x N] ----> our NODES
y = np.sin(2 * np.pi * x) + 0.5 * np.cos(6 * np.pi * x + (np.pi / 4)) #[Nx1]

# noise implementation
noise = np.random.normal(0,0.2,100) #[1 x N]
noisy_y = y + noise #[1 x N]

#plot function vs function with noise
plt.plot(x, y, color='k', linestyle='--', linewidth='1', label='original')
plt.scatter(x, noisy_y, marker='.', label='added_noise')
plt.title('Original function vs added noise function')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.legend()
plt.show()
```



OLS approximation

Fitting a curve in this part is provided by OLS algorithm. Such process can be theoretically explained by estimating appropriate coefficients β, used for solving system of linear equations.

$$X\boldsymbol{\beta} = \mathbf{y},$$

where

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{bmatrix}, \qquad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

p … represents order of used polynomial base (including "all ones" column)

n … number of observations

The vector of approximated coefficients $\hat{\beta}$ would be computed according to following formula:

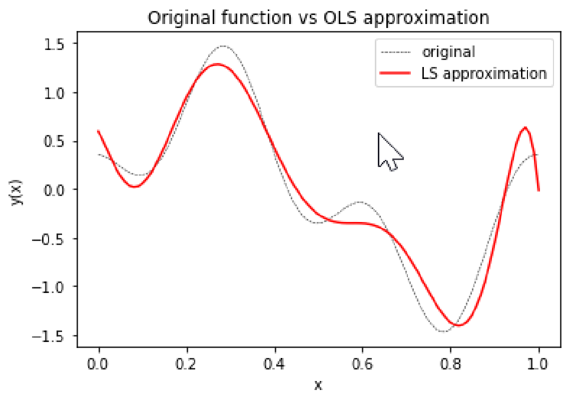
$$\hat{\boldsymbol{\beta}} = (\mathrm{X}^{\mathsf{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathsf{T}}\mathbf{y}.$$

However, for OLS fitting is much simpler to use tools from numpy library: built-in function polyfit/polyval (running on least-square principles) proved itself convenient for our purposes. Last argument in the polyfit function sets the order of used polynomial. In example below, 8th order polynomial was required to get reasonable fit of underlying function.

Formula above will be useful later in custom MLS algorithm.

```
#fit using polynomial basis -> 2nd parameter of polyfit function describes the Mth order of used polynome
#(8th order in this example)
# this represents calculated coefficients "beta"
ols_fit = np.polyfit(x.reshape(1, -1)[0], noisy_y.reshape(1, -1)[0], 8) #[1 x (M + 1)]

#plot original vs OLS approximation
plt.plot(x, y, color='k', linestyle='--', linewidth='0.5', label='original')
plt.plot(x, np.polyval(ols_fit, x), 'r', label='LS approximation')
plt.legend()
plt.title('Original function vs OLS approximation')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.legend()
plt.show()
```



MLS approximation

Mathematical part:

- In oppose to OLS method, coefficients $\hat{\beta}$ are not constant. They vary with each particular value of x vector of unknowns.

- We need to implement weight matrix in equation used for estimating unknown coefficient vector in calculation each particular approximation $\hat{y}(x)$. P corresponds to design matrix X from above.

$$\hat{\beta} = \left[P^T W P\right]^{-1} P^T W y$$

- approximated value $\hat{y}(x)$ is then computed as:

$$\hat{y} = p^T \left[P^T W P\right]^{-1} P^T W y = \Phi(x) y$$

  where p is basis function for particular unknown x, $\Phi(x)$ is known as MLS shape function

- **weighting function** - very important element in MLS algorithm. We specify "support domain" of particular size. Weighting function is defined inside this domain (gaussian function is commonly used for such task) and returns 0 otherwise. This allows us to make an estimate of particular point based only it's close neighbors and thus, make kind of "local" estimation.

  In the block of code with MLS algorithm, a diameter of the support domain is represented by parameter dmi

Practical part:

First step of this task will be creation of design matrix (matrix X in formula for obtaining β vector of approximations). For this purpose, I defined custom function:

```
def create_design_matrix(x):
    '''

    IN - vector of independent variable values (where y(x) is known); [1 x N]

    OUT - "design" matrix [N x (M+1)]; each row represents one particular observation; each column corresponds to one
          "element" of selected basis
    '''

    ones_col = np.ones(len(x)).reshape(-1,1) # [N x 1]
    bases_to_P = (ones_col,x.reshape(-1,1),x.reshape(-1,1)**2) # the base is defined here

    return np.hstack(bases_to_P)
```

Now we create a list of values , so we can estimate their value functional value. Here I came up with same vector as the initial vector of nodes, but the algorithm of course works for distinct ones.

In the next step, I set weight function's parameters that provided good results. By tuning of these parameters, it is possible to get quite different outcomes. More on that will be discussed later.

```
#choose values where we want to get y(x) approximation
unknowns = np.linspace(0, 1, 100) #[1 x N]
```

```
#setting parameters for MLS algorithm
h = 0.01 ## interval between observations
dmi = 20 * h ## weighting diameter
gauss_coef = 3 ## gauss function coefficient
```

The block of code below describes the heart of MLS algorithm. It can be described in following steps:

1. Given nodes , create matrix P (this is our design matrix) with respect to chosen basis
2. Loop over every unknown point x and form the shape function, then make estimation:
   - calculate weight function $w(x)$ for N nodes and form diagonal weight matrix W
   - calculate shape function $\Phi(x) = p^T[P^T W P]^{-1} P^T W$
   - Get the approximation $\hat{y} = \Phi(x)y$

```
# create design matrix
P = create_design_matrix(x) #[N x (M+1)]

# initialize empty array = placeholder for approximated y(x) vector
mls_fit = np.array([]) #[]

#loop over unknowns points (y(unk) will be approximated)
for unk in unknowns:

    # calculate relative distances between unknown point and each x (or "observation"); # [1 x N]
    r = np.abs(x - unk) / dmi

    # calculate "weight for each x" [1 x N]; gaussian function used
    w = (np.where(r <= 1, (np.exp((-r ** 2) * (gauss_coef** 2)) - np.exp(-gauss_coef** 2))
                  / (1 - np.exp(-gauss_coef** 2)), 0))

    # put weights into diagonal matrix [N x N]
    W = np.diag(w)

    #computation of matrices used for coefficients calculation
    A = np.dot(np.dot(P.T, W), P) #[M x M]
    A_inv = np.linalg.inv(A) #[M x M]
    B = np.dot(P.T, W) #[M x N]

    # here we need to use selected base
    p = np.hstack([1, unk, unk ** 2]) #[1 x M]

    #create shape function
    shape_func = np.dot(np.dot(p, A_inv), B) #[1 x N]

    #append approximate value to final vector
    mls_fit = np.append(mls_fit, shape_func.dot(y)) #[1 x N]
```

Results

The figure below shows comparison of the original function (+ noise) and "fitted" functions. We can see markable benefits of using MLS algorithm and its more local-based approximation.

By tuning of parameters dmi and gaussian coefficient of the weight function, we can affect the result of MLS fit. Generally, by choosing higher value of dmi, "smoother" approximation is achieved, on the other hand there might be higher residuals between fitted and original function. Lower value will provide fit closer to original, but smoothness of the approximation curve would be reduced.

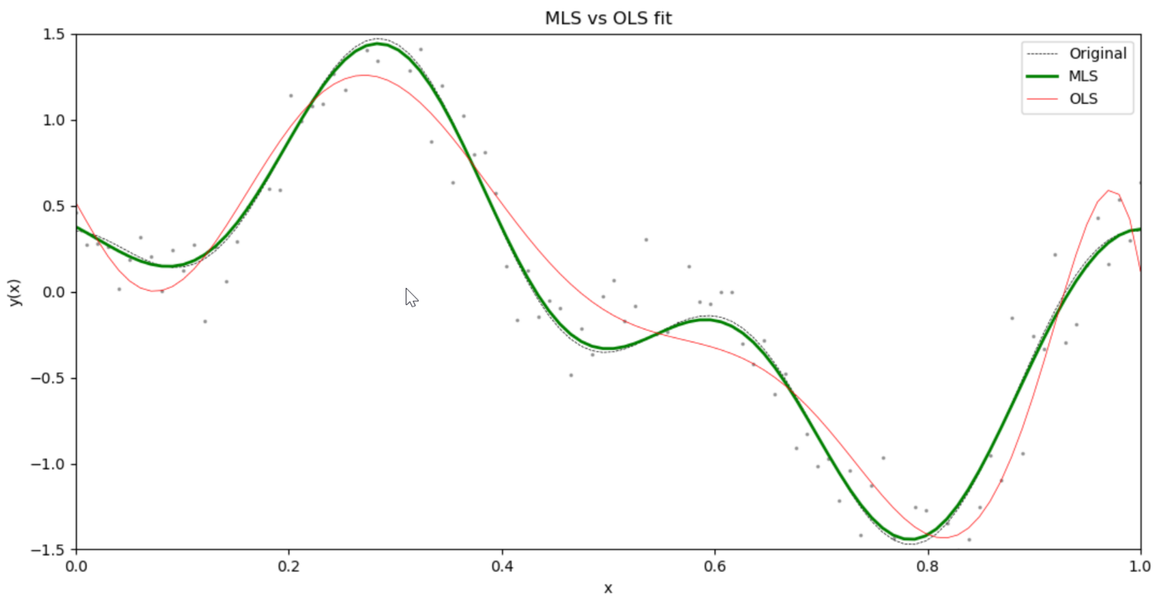
The value of gauss parameter works in similar way.

MLS proved itself to be much more flexible than OLS algorithm - it can be tuned by parameters above and we can get good results with using lower order of polynomial bases.

```
plt.scatter(x, noisy_y, marker='.', s=10, color='k', alpha=0.3)
plt.plot(x, y, 'k--', label='Original', linewidth=0.5)
plt.plot(unknowns, mls_fit, 'g', linewidth=2, label='MLS')
plt.plot(x, np.polyval(ols_fit, x), 'r', label='OLS', linewidth=0.5)

fig = plt.gcf()
fig.set_size_inches(10, 8)


plt.xlim(0,1)
plt.ylim(-1.5,1.5)
plt.title('MLS vs OLS fit')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.legend()
plt.show()
```


MLS vs OLS fit
Original
MLS
OLS
y(x)
x