Report Robotique et Apprentissage - Lab 1

Nicolas Zuo Julien Willaime-Angonin

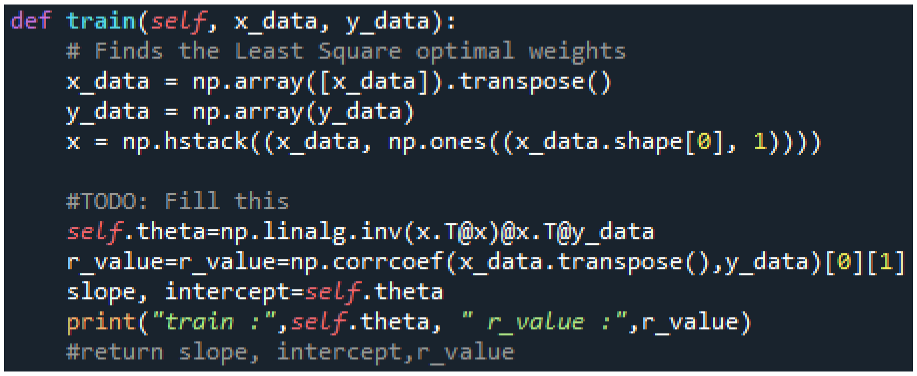
11 February 2021

Professor : Olivier Sigaud

1 Code question 1 :

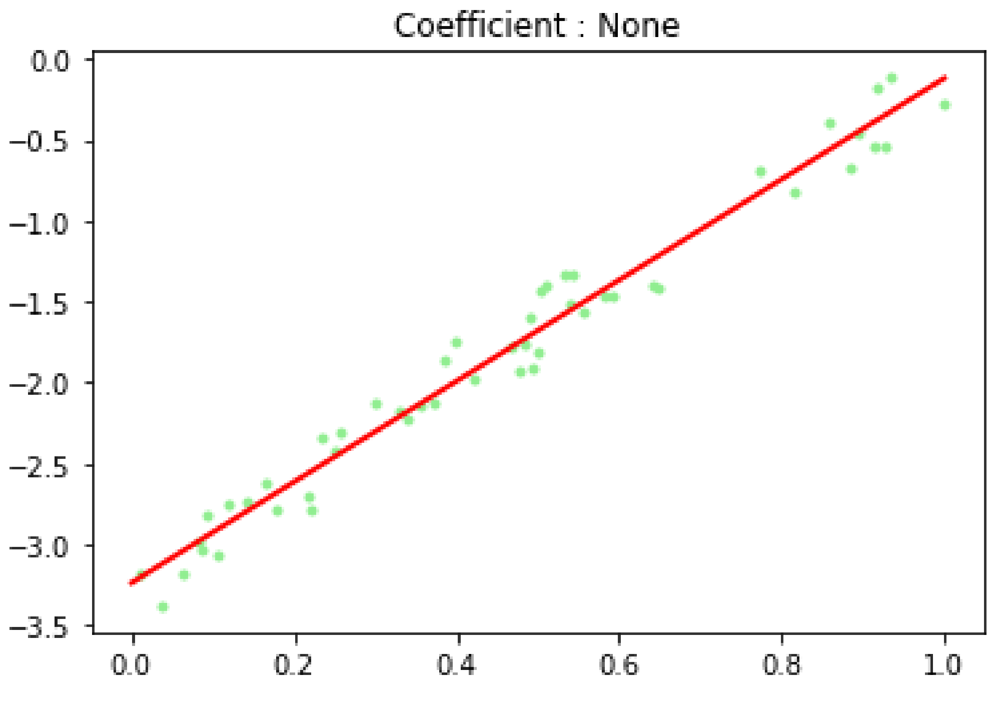
The values returned by ”train” are almost the same as the ones from ”train_from_stats” but ”train from_stats” gives values with more decimals than ”train” and the r_value differs slightly (around the 14Th or 15Th decimal) between the two methods. The difference between the two results comes from the differences between the calculation methods, which probably round off slightly different.This difference can be explained by the fact that ”train” calculates the precise result and that train_from_stats calculates a very approximate value with the data given.

Code of the function train :

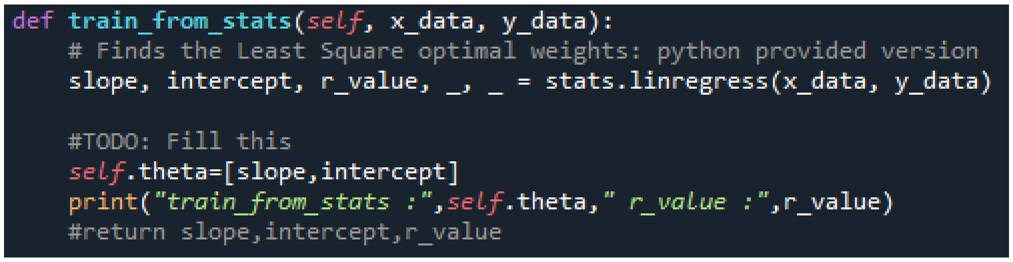


```
def train(self, x_data, y_data):
    # Finds the Least Square optimal weights
    x_data = np.array([x_data]).transpose()
    y_data = np.array(y_data)
    x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

    #TODO: Fill this
    self.theta=np.linalg.inv(x.T@x)@x.T@y_data
    r_value=r_value=np.corrcoef(x_data.transpose(),y_data)[0][1]
    slope, intercept=self.theta
    print("train :",self.theta, " r_value :",r_value)
    #return slope, intercept,r_value
```

Graph result of train :

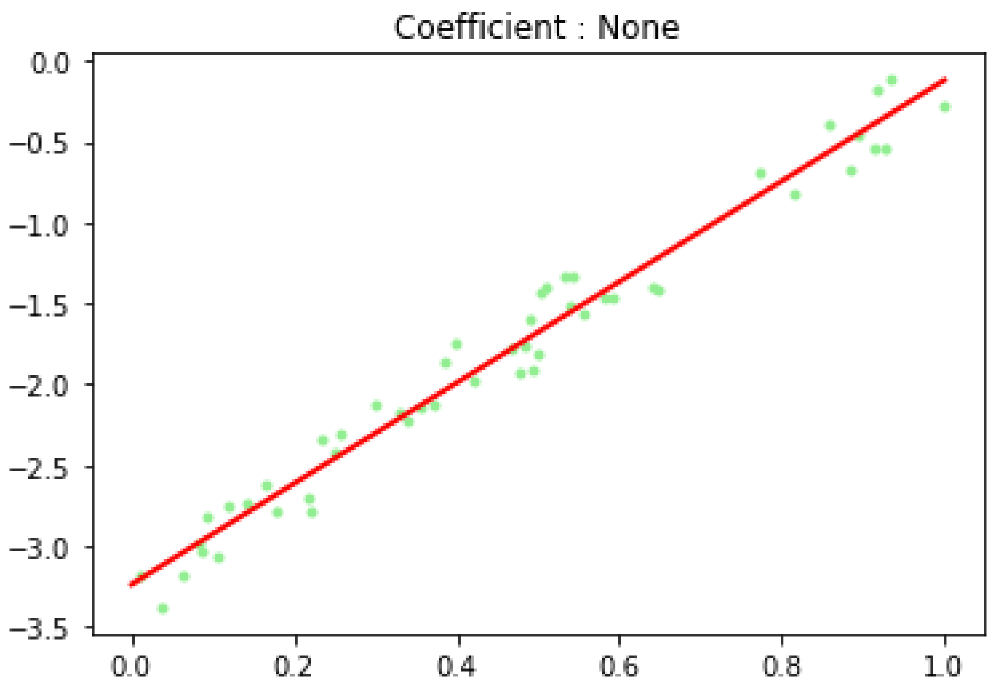


Code of the function train_from_stats :

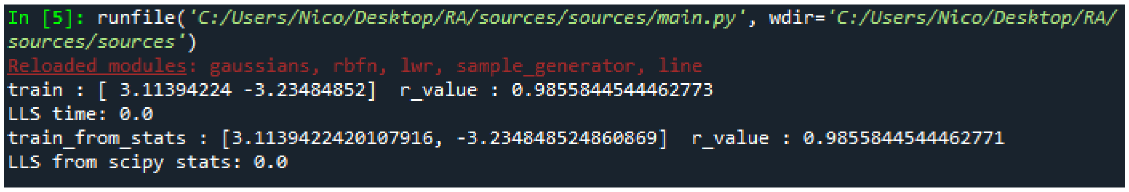


```
def train_from_stats(self, x_data, y_data):
    # Finds the Least Square optimal weights: python provided version
    slope, intercept, r_value, _, _ = stats.linregress(x_data, y_data)

    #TODO: Fill this
    self.theta=[slope,intercept]
    print("train_from_stats :",self.theta," r_value :",r_value)
    #return slope,intercept,r_value
```

Graph result of train_from_stats :



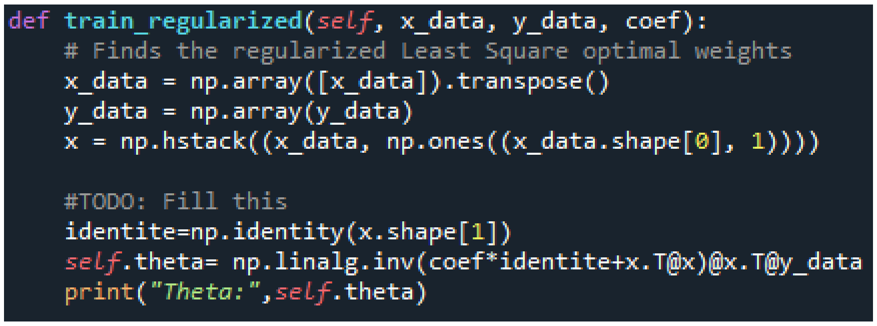
Result on console :



```
In [5]: runfile('C:/Users/Nico/Desktop/RA/sources/sources/main.py', wdir='C:/Users/Nico/Desktop/RA/
sources/sources')
Reloaded modules: gaussians, rbfn, lwr, sample_generator, line
train : [ 3.11394224 -3.23484852]  r_value : 0.9855844544462773
LLS time: 0.0
train_from_stats : [3.1139422420107916, -3.234848524860869]  r_value : 0.9855844544462771
LLS from scipy stats: 0.0
```

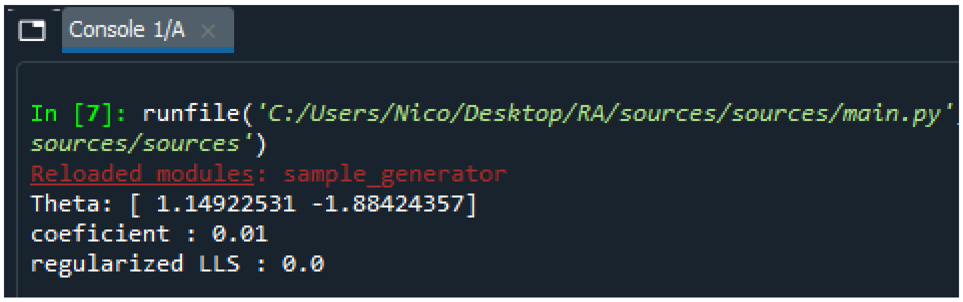
2 Code question 2 :

Code of the function :



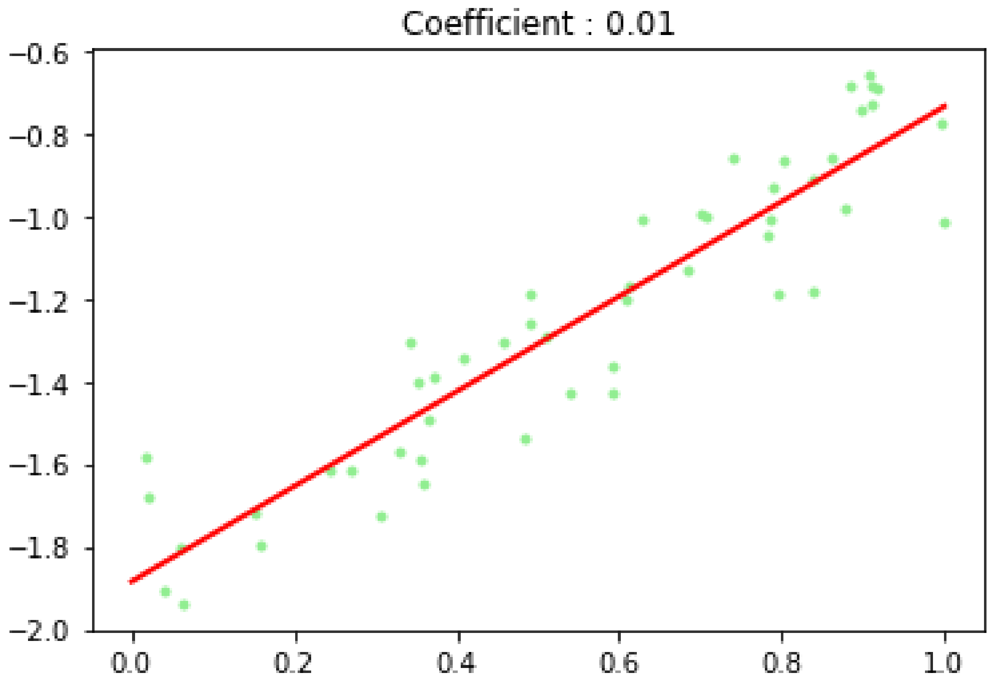
```python
def train_regularized(self, x_data, y_data, coef):
    # Finds the regularized Least Square optimal weights
    x_data = np.array([x_data]).transpose()
    y_data = np.array(y_data)
    x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

    #TODO: Fill this
    identite=np.identity(x.shape[1])
    self.theta= np.linalg.inv(coef*identite+x.T@x)@x.T@y_data
    print("Theta:",self.theta)
```

Result on console :



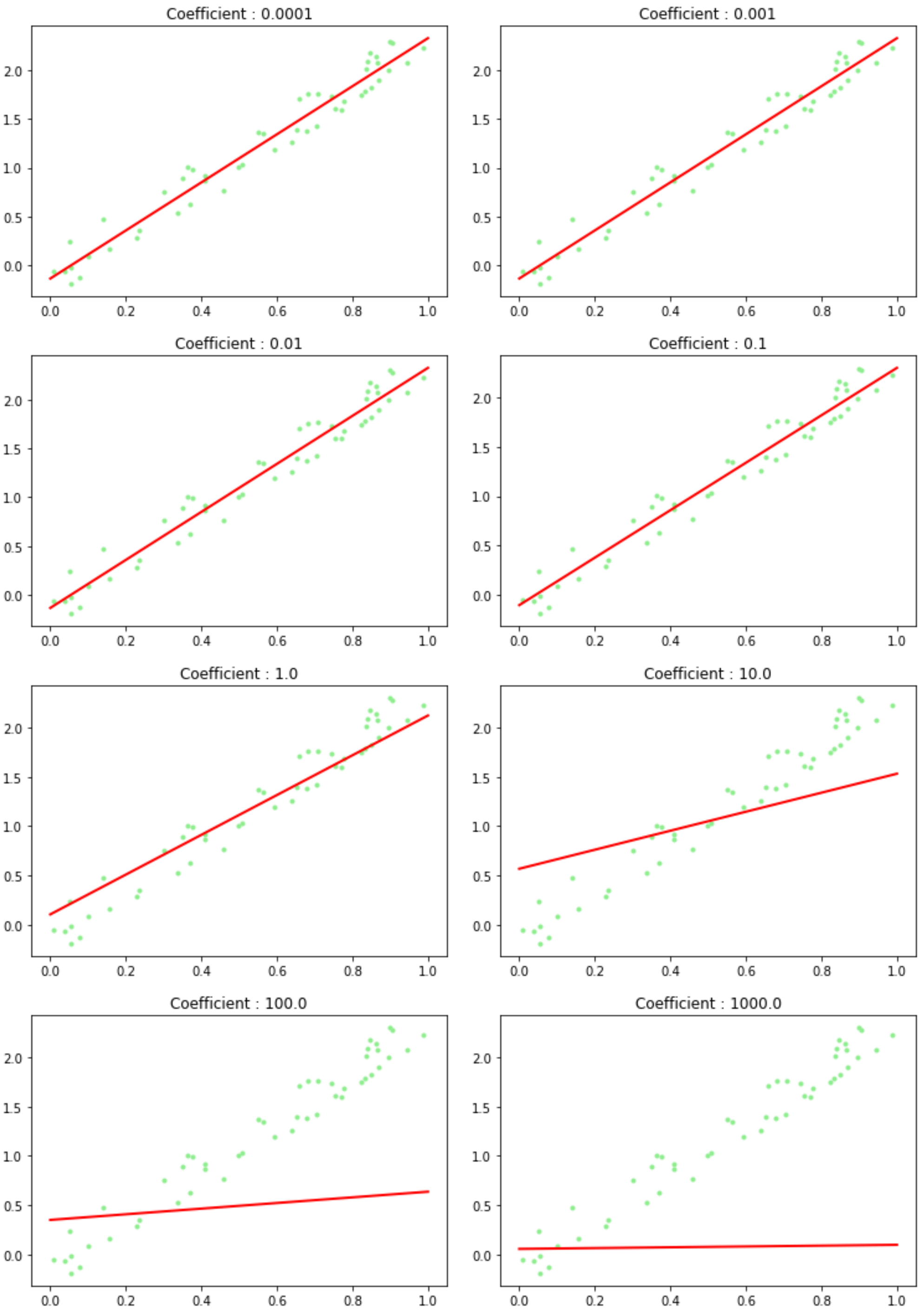
```
In [7]: runfile('C:/Users/Nico/Desktop/RA/sources/sources/main.py'
sources/sources')
Reloaded modules: sample_generator
Theta: [ 1.14922531 -1.88424357]
coeficient : 0.01
regularized LLS : 0.0
```

Graph result :



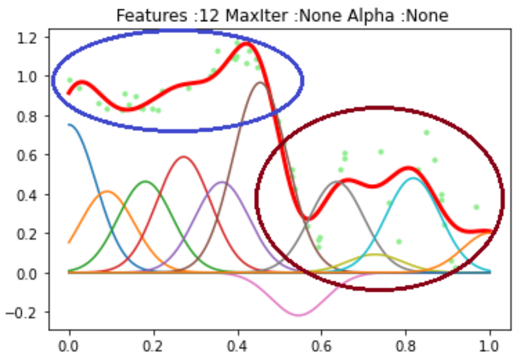
3 Study question 3 :

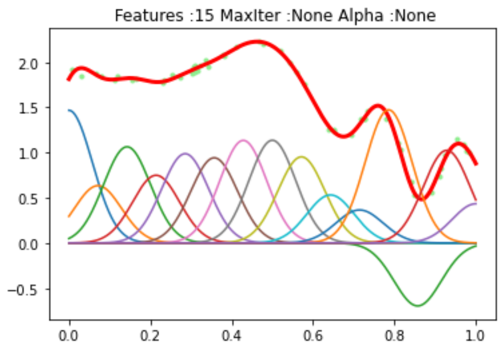
With a batch of 50 points, when the coefficient is smaller than 0.01 there is close to no difference in the result. When the coefficient is 0.1 the slope of line becomes slightly less accurate . When the coefficient is greater than 1 the difference becomes much more noticeable and when it is greater than 10 the line is almost to meaningless.

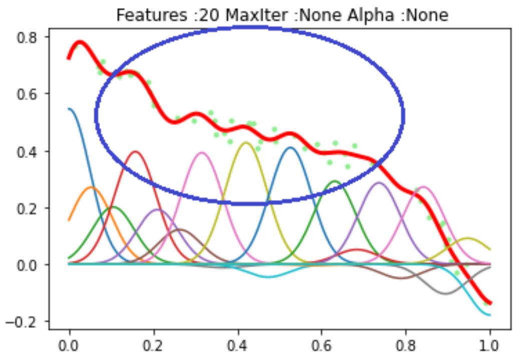

Coefficient : 0.0001
Coefficient : 0.001
Coefficient : 0.01
Coefficient : 0.1
Coefficient : 1.0
Coefficient : 10.0
Coefficient : 100.0
Coefficient : 1000.0

4 Code question 4 :

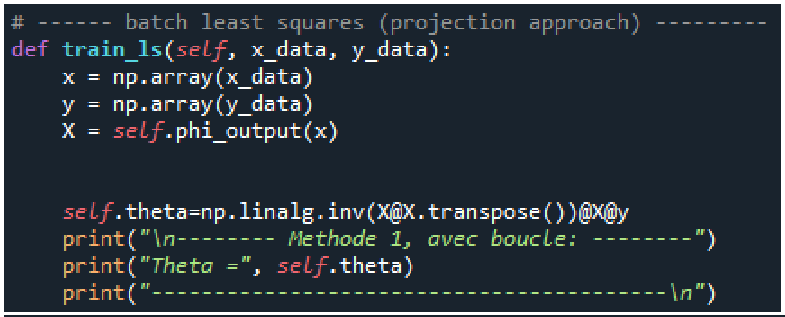
By assigning a number of features between 10-12, sometimes there is a very small start of overfitting (see the first image the blue frame) but also when you have areas with a lot of waviness it's not enough anymore (see the first image the brown frame). When you increase to 20 features the precision seems perfect, but when you have a lot of points in the same area you have overfitting (see the third image with the blue frame) and so 15 features seems to be the right compromise between underfitting and overfitting.



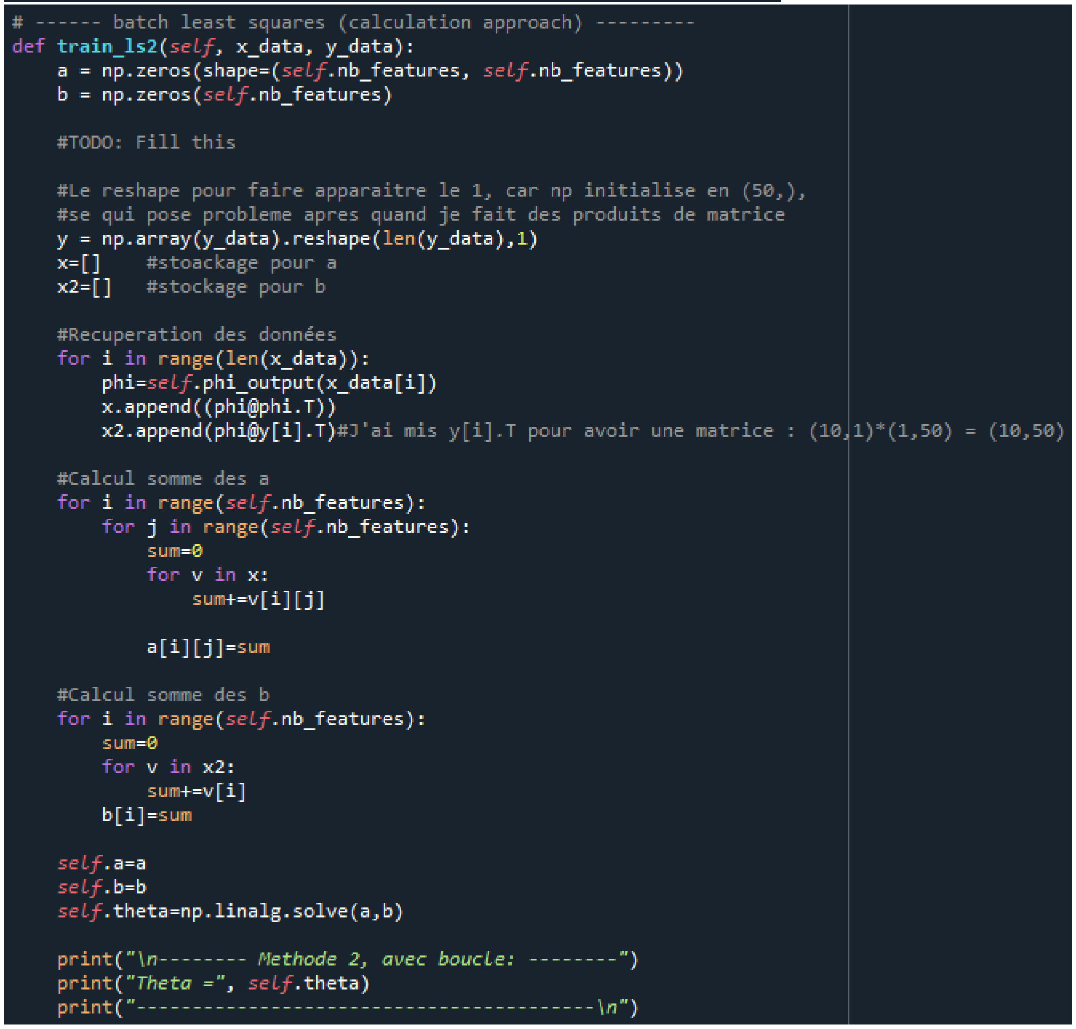




Code of the function ls and ls2 :



```python
# ------ batch least squares (projection approach) ---------
def train_ls(self, x_data, y_data):
    x = np.array(x_data)
    y = np.array(y_data)
    X = self.phi_output(x)


    self.theta=np.linalg.inv(X@X.transpose())@X@y
    print("\n-------- Methode 1, avec boucle: --------")
    print("Theta =", self.theta)
    print("-----------------------------------------\n")
```

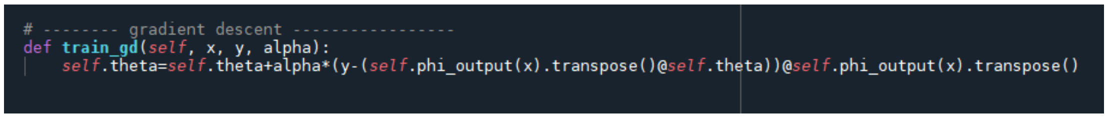


```python
# ------ batch least squares (calculation approach) ---------
def train_ls2(self, x_data, y_data):
    a = np.zeros(shape=(self.nb_features, self.nb_features))
    b = np.zeros(self.nb_features)

    #TODO: Fill this

    #Le reshape pour faire apparaitre le 1, car np initialise en (50,),
    #se qui pose probleme apres quand je fait des produits de matrice
    y = np.array(y_data).reshape(len(y_data),1)
    x=[]    #stoackage pour a
    x2=[]   #stockage pour b

    #Recuperation des données
    for i in range(len(x_data)):
        phi=self.phi_output(x_data[i])
        x.append((phi@phi.T))
        x2.append(phi@y[i].T)#J'ai mis y[i].T pour avoir une matrice : (10,1)*(1,50) = (10,50)

    #Calcul somme des a
    for i in range(self.nb_features):
        for j in range(self.nb_features):
            sum=0
            for v in x:
                sum+=v[i][j]

            a[i][j]=sum

    #Calcul somme des b
    for i in range(self.nb_features):
        sum=0
        for v in x2:
            sum+=v[i]
        b[i]=sum

    self.a=a
    self.b=b
    self.theta=np.linalg.solve(a,b)

    print("\n-------- Methode 2, avec boucle: --------")
    print("Theta =", self.theta)
    print("-----------------------------------------\n")
```

5 Code question 5 :

The values of maxIter we chose is : 50000, the value of nbFeatures is 20 and the value of alpha is 0.1.If the number of features is too great, there can be some over-fitting, in areas with little waviness. The value of the learning rate alpha determines how big each steps will be for each iteration of traingd, if the learning rate is too big, there is a risk of missing the optimum, and if the learning rate is too small, there is a chance of not getting to the optimum at all, the smaller the alpha is the bigger the number of iteration must be. When the graph doesn't have many curves a smaller number of features can be better but when the graph has many curves, a small number of features isn't enough to learn the function properly because it can cause some underfitting in areas with a lot of curves.

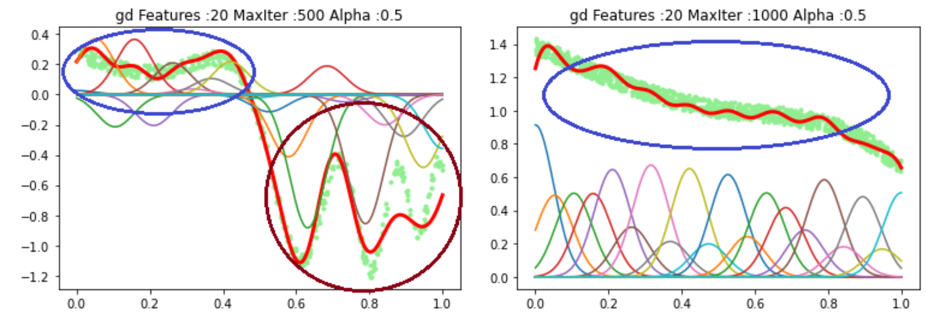
Code of traingd :



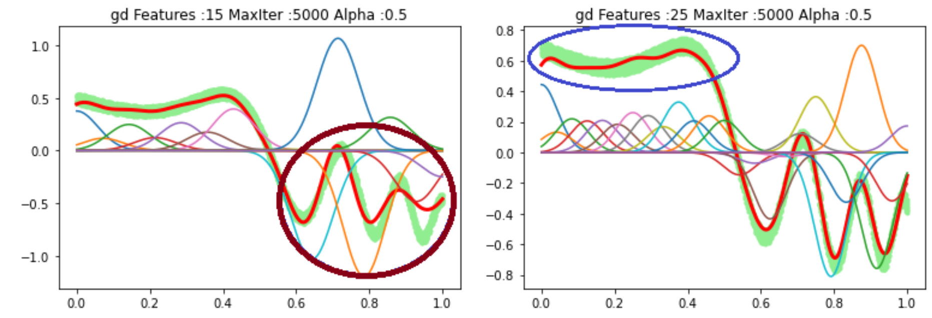
```
# -------- gradient descent -----------------
def train_gd(self, x, y, alpha):
    self.theta=self.theta+alpha*(y-(self.phi_output(x).transpose()@self.theta))@self.phi_output(x).transpose()
```

Graphs obtained by using the function:

We have observed that by increasing the number of points we can settle most of the cases in underfitting :

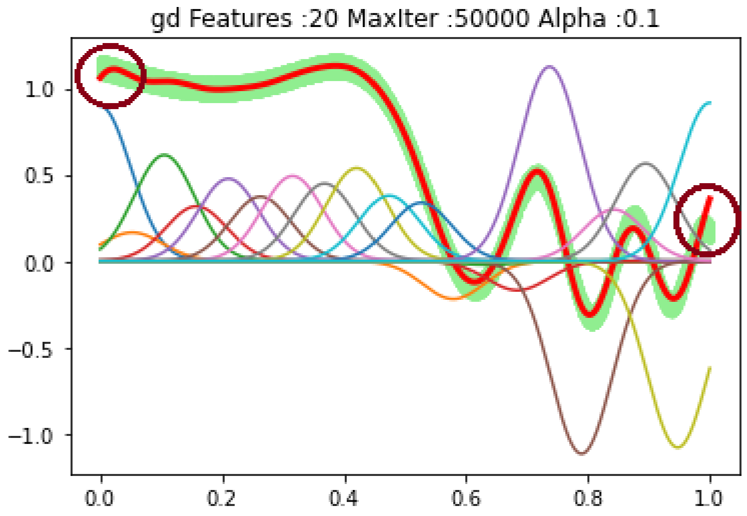


We have observed that with 15 features we often have underfitting and with 25 overfitting, for 5000 iterations and alpha at 0.5.



We finally understood that in order to have a good learning, we have to be as many points/iteration as possible in our case, but also to avoid an over learning we can lower the steps from alpha to 0.1, but as we have a lot of iteration to learn avoids a little bit the problem of under learning due to the too small step of alpha. And we took as feature equal to 20, because it's a good compromise between overfitting and underfitting in our case.

***Note** : On the extremities there is always a strange curve, here circled in brown, because there is no Gaussian at the beginning and at the end to smooth this area.*

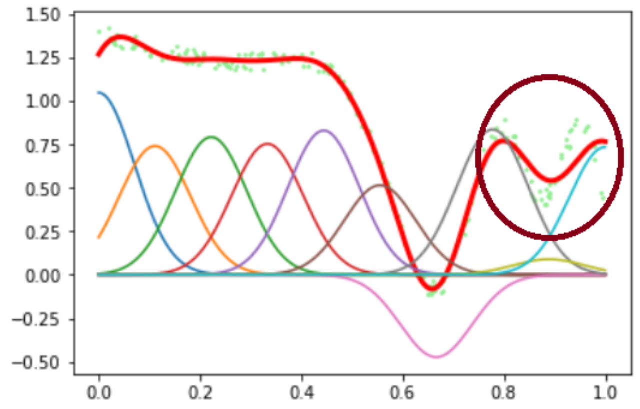

gd Features :20 MaxIter :50000 Alpha :0.1
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

6 Study Question 6 :

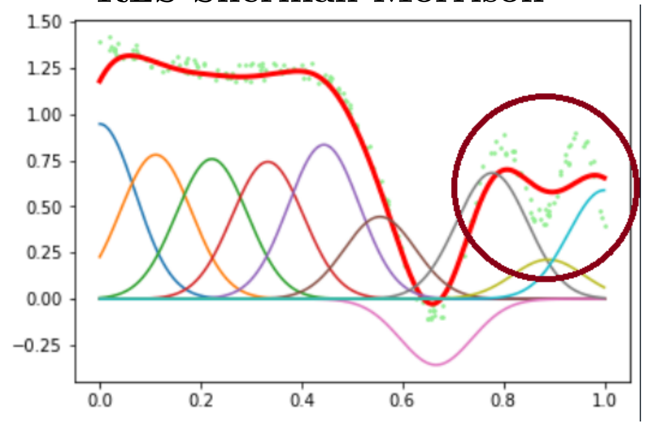
Both methods seem to be precise with 200 iteration, the difference is in the number of features needed to get good results. RLS needs about 15 features and RLS with Sherman-Morrison needs about 25.

With only **10 features** both methods seem to generate underfitting.

RLS

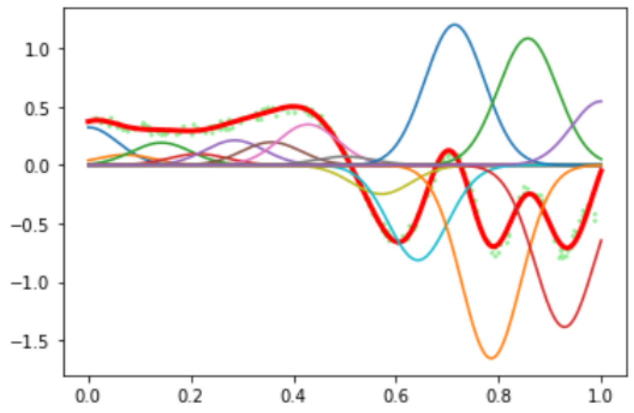


RLS Sherman-Morrison

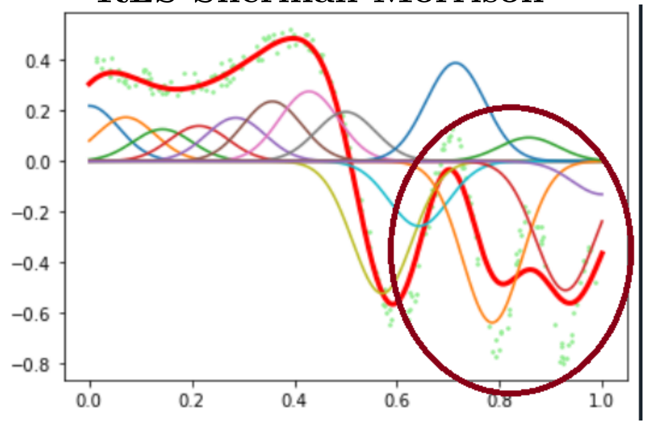


With **15 features** the RLS methods without Sherman-Morrison seem to give good result while the one with Sherman-Morrison seem to still be underfitting.

RLS

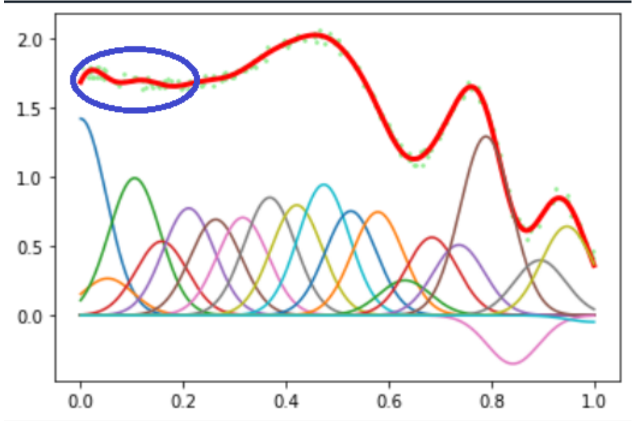


RLS Sherman-Morrison

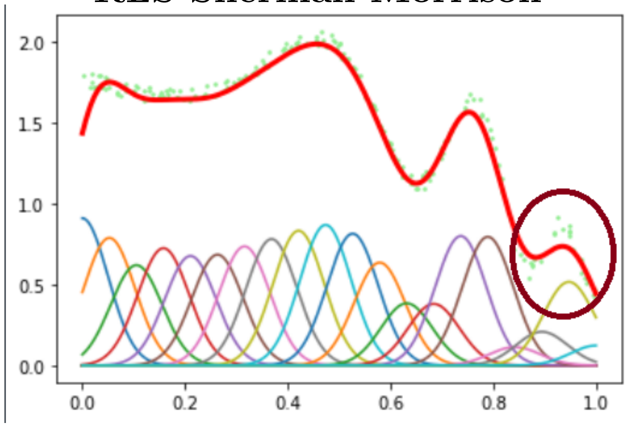


With **20 features** the RLS method is beginning to overfit while the RLS with Sherman-Morrison one is still underfitting.

RLS

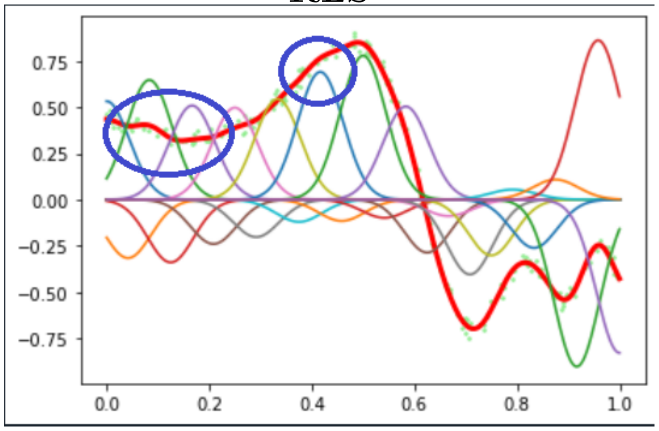


RLS Sherman-Morrison

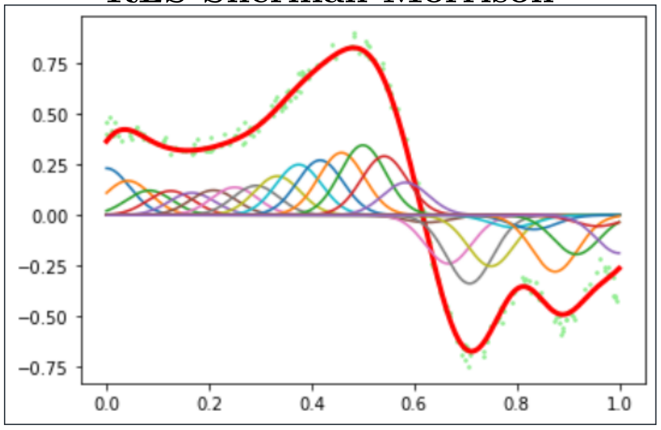


Finally with **25 features** the RLS with Sherman-Morrison method is giving good results and RLS is over-fitting.

RLS



RLS Sherman-Morrison



7 Study Question 7 :

The RLS method (without sherman-morrison) is more precise but has more of a risk of overfitting than the RLS method (with sherman-morrison), which has less ”noise” when the number of features is high, and is more precise than the gradient descent method. The RLS method(with sherman-morrison) takes about twice the computation time the gradient descent need with the same parameters , RLS(without sherman-morrison) needs thrice this time: for **15 features, alpha= 0.5 and 200 iterations**:

- GD=0.007s;
- RLS(sherman-morrison) = 0,013s;
- RLS=0,022s

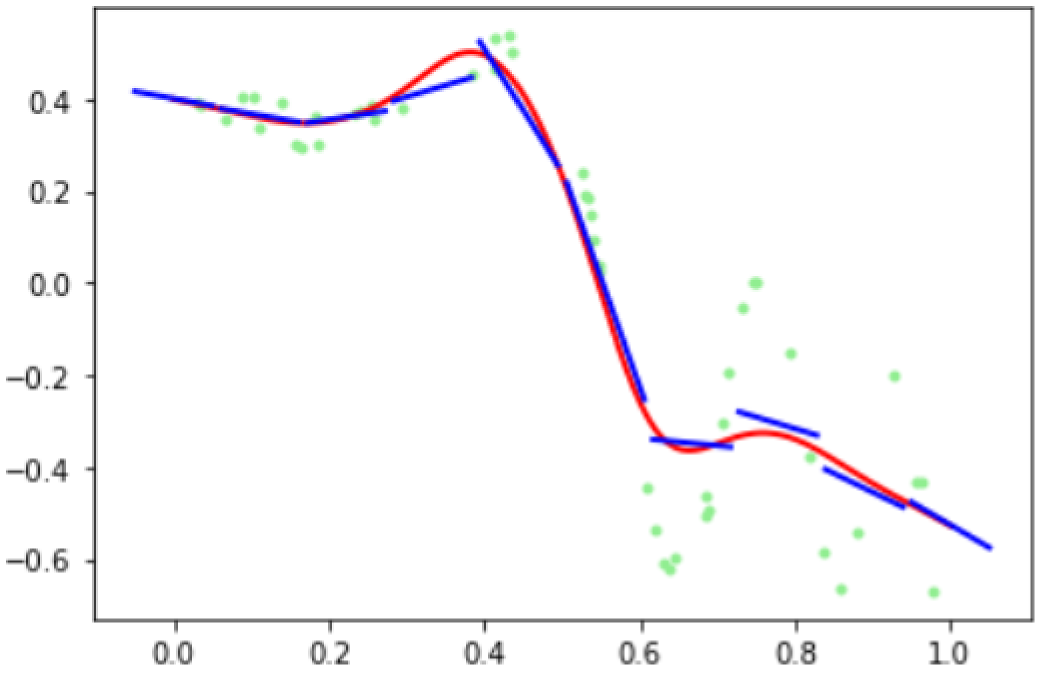
Note : These times are an average on 1000 iterations with the same parameters.
RLS methods are slower than gradient descent because they have to inverse a matrix.

8 Study Question 8 :

The main difference between the batch mode and the incremental mode is the fact that the batch mode is very precise, as the name says it learns about the whole batch of data, whereas the incremental method is less precise and gives us approximate values therefore less accurate. The advantages of the batch mode is that it is very accurate but the computation time is exponential with the number of features (due to a matrix inversion), which makes it impossible to use this method which is very accurate but unusable when we have a large number of features. The incremental method is certainly less precise, because it is an approximate value, but in terms of complexity it is much better and therefore it is often the chosen method, because we often work in large databases. More generally we could use the batch method when we have few features to have more precise learning, and use the incremental method when we have a large number of features.

9 Code Question 9 :

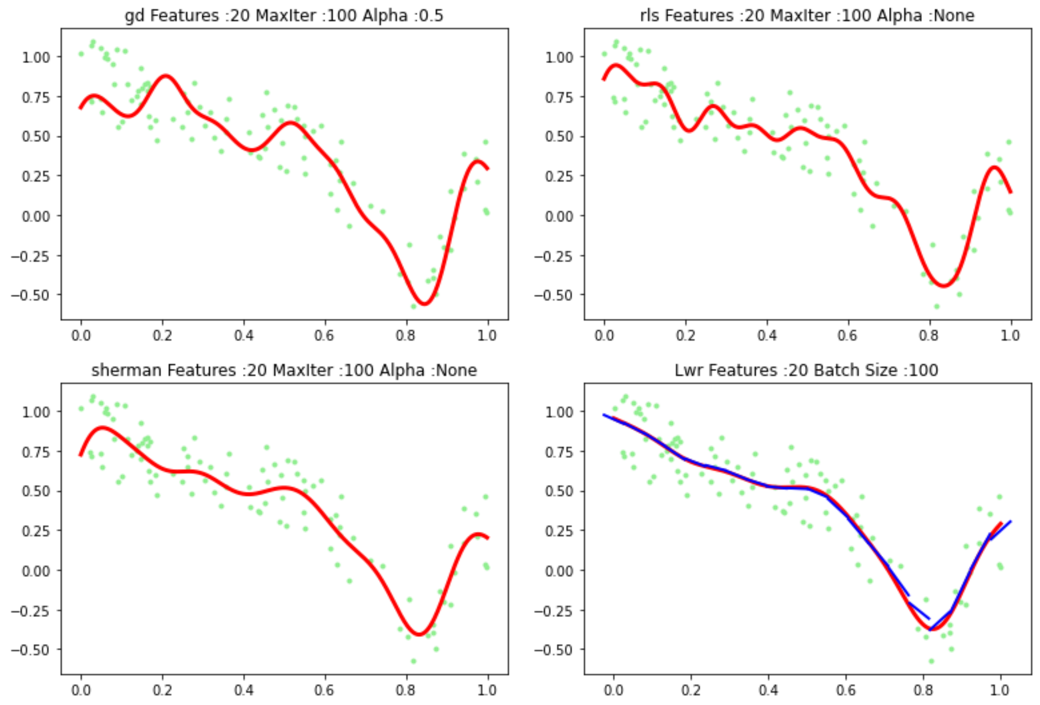
We can see that 10 features is not enough to learn the more complex functions.



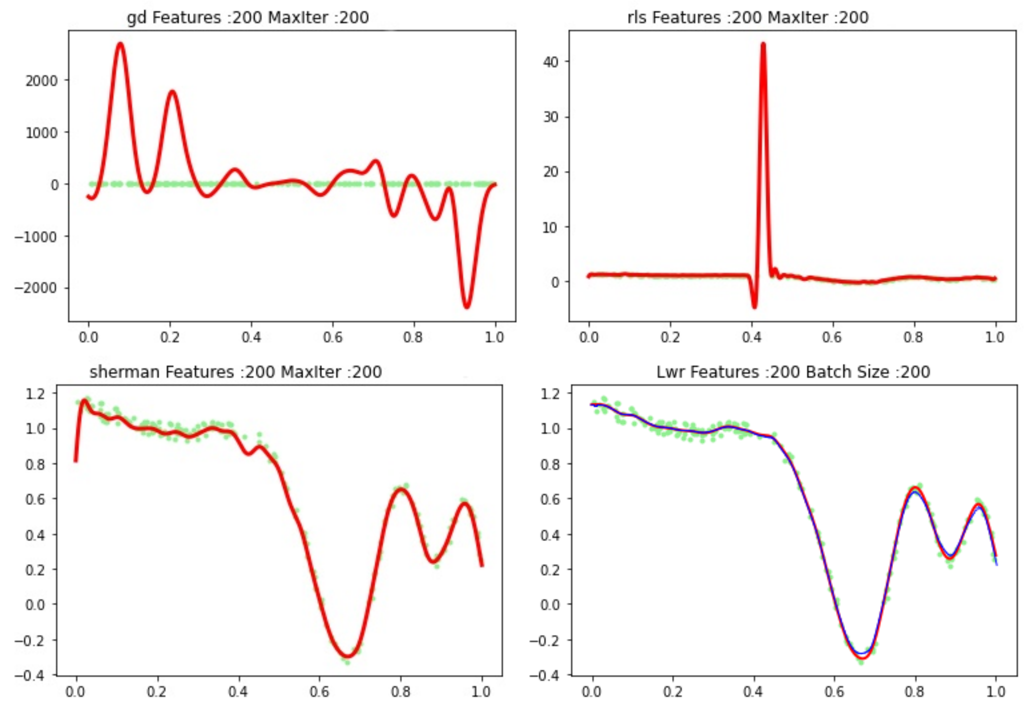
10 Study Question 10 :

For the same parameters RBFN methods are faster and more precise when the data noise is small, whereas LWR is more resistant to data noise and less precise when the number of features is small but if this number is big, LWR is less prone to overfitting than the other methods.

We can see on these four images below, with **20 features, 200 data and some noise (from the sample, sigma equal to 0.5)**, that the RBFN method had a hard time learning well, but for the Locally Weighted Least Squares method, it managed to learn well from data with high noise.



Then we tested on a large number of features to see the difference between RBFNs and LWRs, so with **200 features, 200 data and a bit of noise (from the sample, sigma equal to 0.1)**, we can see that only Sherman Morrisson's method, still shows us a curve (red curve) that is correct in RBFNs methods, because the "gd" and "rls" methods, manage much less well the overfitting, which gives us curves that don't correspond to anything anymore, but we can see that Sherman Morrisson's result has a lot of overfitting areas compared to the Locally Weighted Least Squares method.



LWR is a lot slower than all the RBFN as it takes about 0.13s (versus GD=0.007s, RLS(Sherman-Morrison) = 0,013s and RLS=0,022s) to compute in the same con-

ditions as the RBFNs (15 features, 200 data points).
We also tested the "ls" method, the batch mode of the RBFN with 100 features and 10 000 points, the "ls" method is much faster than the lwr method, which seems to be normal, because "lwr" makes a succession of batch methods on local areas.

Note : These are exactly the same parameters used in speed tests with RBFN methods. (see above question 7)

Overall, the LWR method seems to give better results

In conclusion, Locally Weighted Least Squares give us better results in most cases, this better result can be explained by the fact that RBFNs are successive concatenations of Gaussian which can facilitate the appearance of overfitting in certain areas with undulations (humps), or the fact that when there is a strong noise (sigma equal to 0.5) and when we don't have many points, RBFNs have much more difficulty to learn them, because we concatenate Gaussian while with Locally Weighted Least Squares we calculate a succession of locally weighted degrees of slope which avoids strong variation as with RBFNs that create waves(humps). **But above all, Locally Weighted Least Squares is more accurate because it does a succession of local batch methods, which is precise but resource consuming.**
The main differences between these two methods is when we have a high noise (data sample, sigma), the Locally Weighted Least Squares method is much more adapted in this situation than RBFNs, or when we have a lot

of features, but the only drawback is that the execution time compared to RBFNs is about ten times higher.