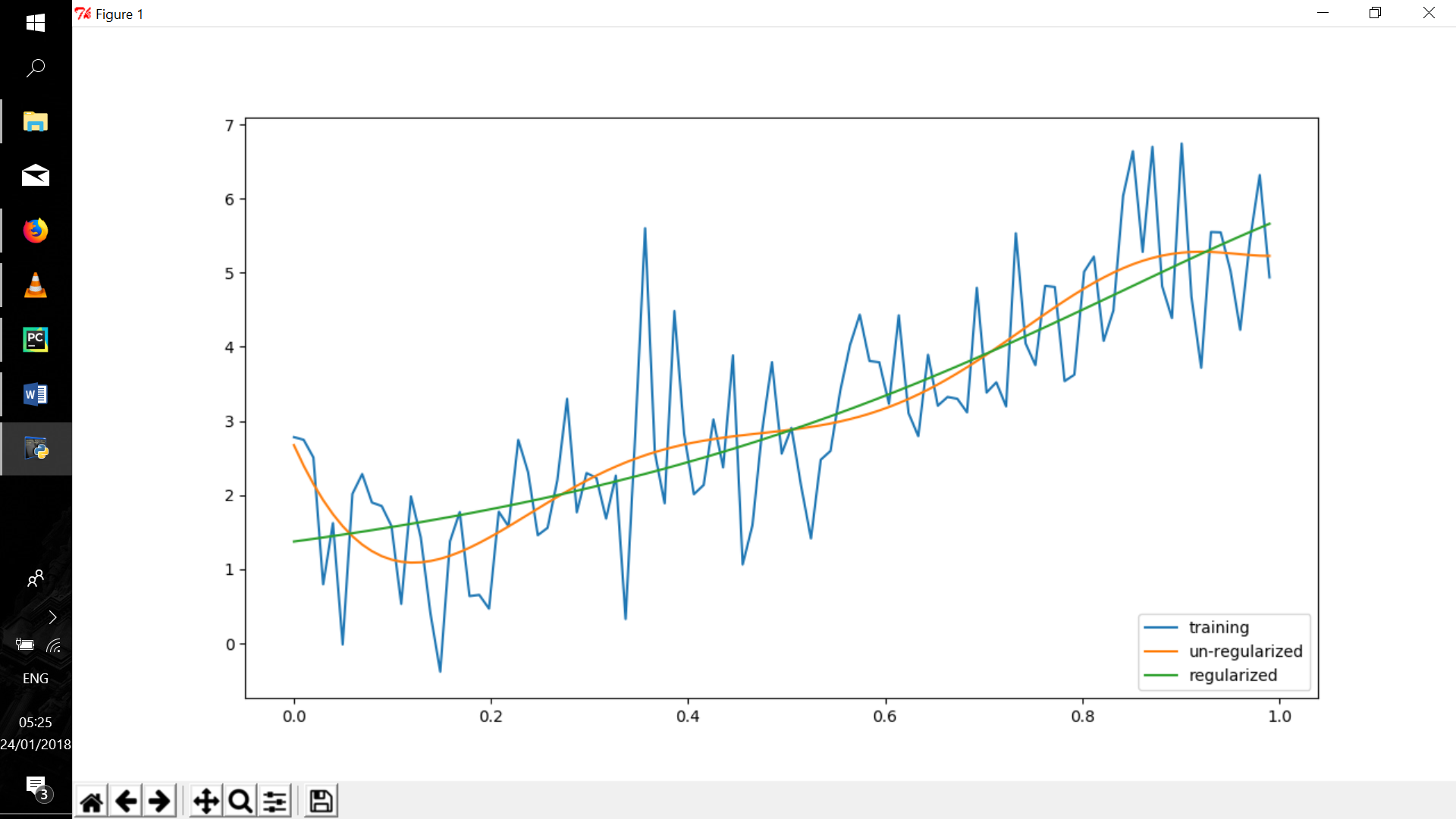
Hit and trial method to choose the regularization value can turn out to be costly.

Fine adjustments in regularized graph without having to compute through all the training data can be very handy.

This is a method of fine tuning regularized graph.

Idea : -

Fourier’s idea was to define graph in terms of trigonometric functions for various transformation. A graph can be split into various smaller segments of waves. In our case we shall split the regularized graph into various segments. It can be assumed that regularized graph meets non-regularized graph at various points.

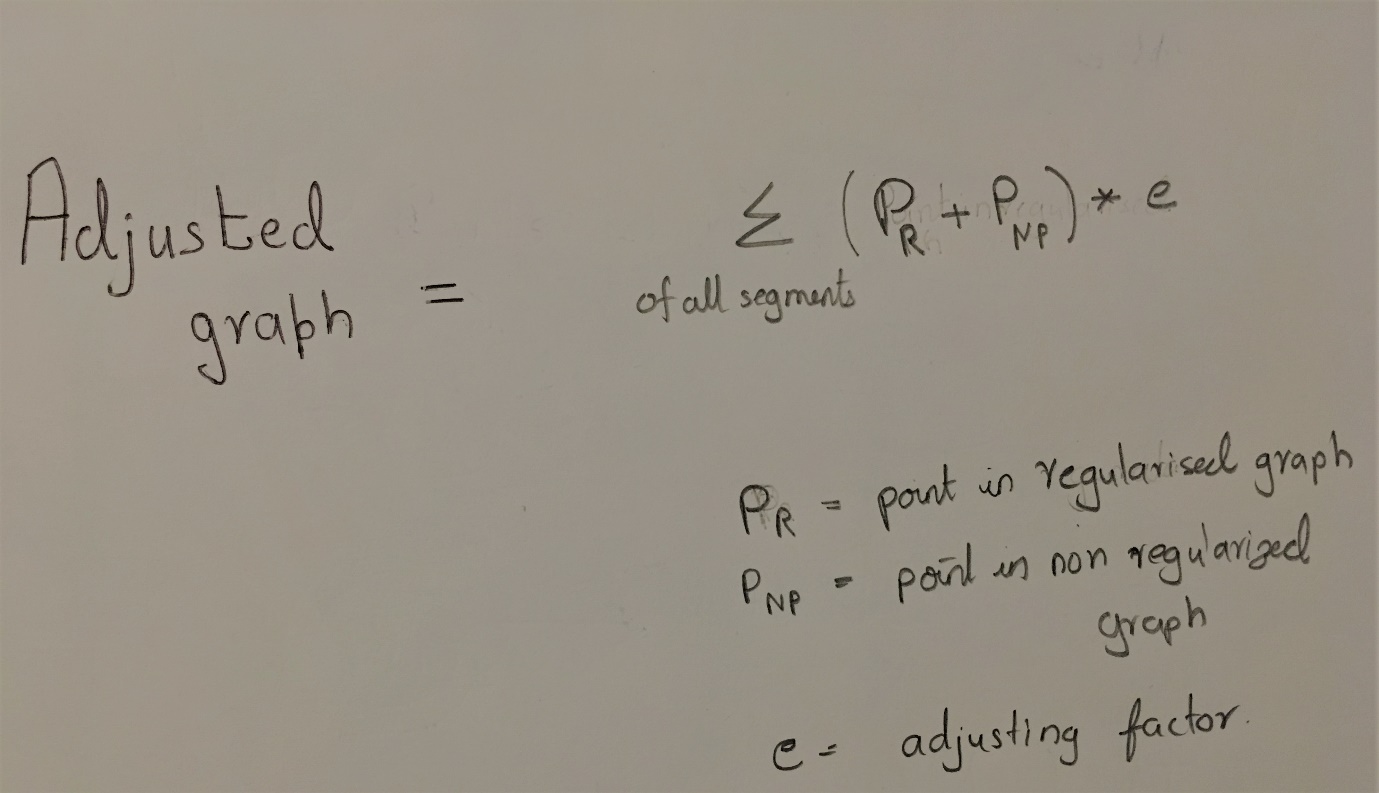


These meeting points can be used to segment regularized graph.

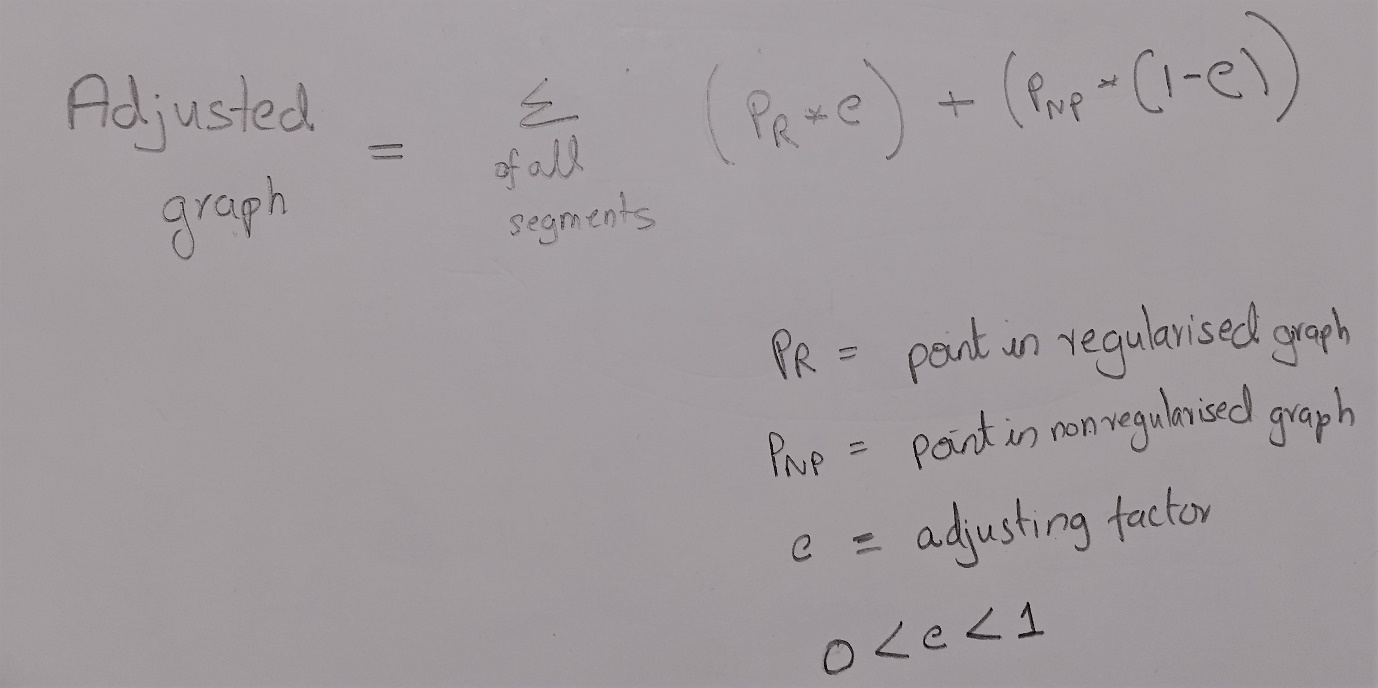
Adjustments can be performed on these segments to obtain a satisfying graph. This would save computation in some cases.

Constant e; is the adjusting factor.(new term that I introduced). Setting the right ‘e’ value will help us adjust the regularization without having to change regularization constant (or having to compute the graph all over again).

‘e’ can be seen as ratio between closeness to regularized graph and the distance from the non regularized graph.



When the adjusted graph has to be within limits bounded by non-regularized and regularized graph ; we can use the below equation:



Implementation :

In this implementation, we use a training data randomly generated and the model is erroneously chosen to be a 7th order polynomial. This implementation illustrates the point that a regularized model can be fine-tuned to some extend.

Python code used for implementation. (Modified code from <https://github.com/masinoa/machine_learning> . Code has been modified to include the new concept)

**import** numpy **as** np  
**from** matplotlib **import** pyplot **as** plt  
*#in order to compare between examples, set a seed in random*seed = 123456789  
np.random.seed(seed)  
**def** y(x,m,b,mu=0,sigma=1): **return** m\*x + b + np.random.normal(mu,sigma,1)[0]  
**def** el\_pow(x,pow):  
 temp = x  
 **for** i **in** range(pow-1):  
 temp = temp \* x  
 **return** temp  
**def** prediction(params, x):  
 pred = 0  
 **for** i **in** range(len(params)):pred += params[i] \* np.math.pow(x, i)  
 **return** pred  
*#training data, with N data points*N = 101  
M = 8  
t = np.empty(N)  
domain = np.empty(N)  
domain\_bound = 1.0/N  
**for** i **in** range(N): domain[i] = i\*domain\_bound  
**for** i **in** range(N): t[i] = y(x=domain[i],m=4.89,b=0.57)  
*#find the solution without using regularization  
#design matrix, phi, N X M*phi = np.array([np.ones(N),domain, el\_pow(domain,2),el\_pow(domain,3),el\_pow(domain,4),el\_pow(domain,5),el\_pow(domain,6),el\_pow(domain,7)]).T  
temp1 = np.linalg.inv(np.dot(phi.T,phi)) *#inverse of phi.T X phi*temp2 = np.dot(temp1, phi.T)  
w1 = np.dot(temp2,t) *#solution***print 'w1='**,w1  
predicted\_t = [prediction(w1,x) **for** x **in** domain]  
  
  
*#find the regularized solution*lam = .5  
temp1 = np.linalg.inv(lam\*np.eye(M)+np.dot(phi.T,phi))  
temp2 = np.dot(temp1,phi.T)  
w2 = np.dot(temp2,t)  
**print 'w2='**,w2  
predicted\_t\_reg = [prediction(w2,x) **for** x **in** domain]  
  
  
**def** regularization\_adjust(args1,args2,times):  
 k=.5  
 **for** x **in** range(1, times):  
 result = (args1 + args2) \* k  
 args1=result  
 **return** args1  
  
*#when e = 3*e=3  
reg\_adjusted\_1 =[regularization\_adjust(prediction(w2,x),prediction(w1,x),e) **for** x **in** domain]  
 *# when e = 2*e=2  
reg\_adjusted\_2 =[regularization\_adjust(prediction(w2,x),prediction(w1,x),e) **for** x **in** domain]  
  
  
*#add some plots*plt.plot(domain,t)  
plt.plot(domain,predicted\_t)  
plt.plot(domain,predicted\_t\_reg)  
plt.plot(domain,reg\_adjusted\_1)  
plt.plot(domain,reg\_adjusted\_2)  
plt.legend((**"training"**,**"un-regularized"**,**"regularized"**,**"reg\_adjusted\_@e=3"**,**"reg\_adjusted\_@e=2"**), loc=**'lower right'**)  
plt.show()

Results:

