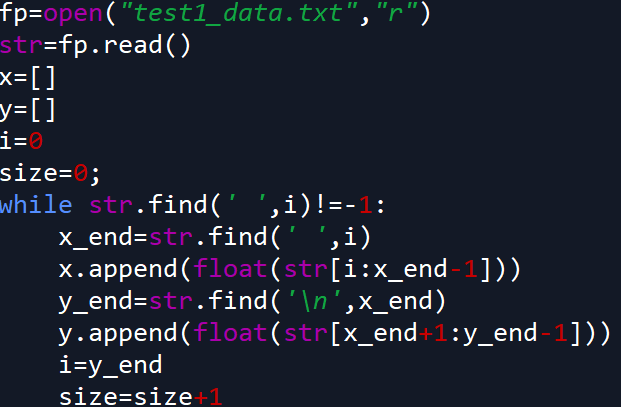
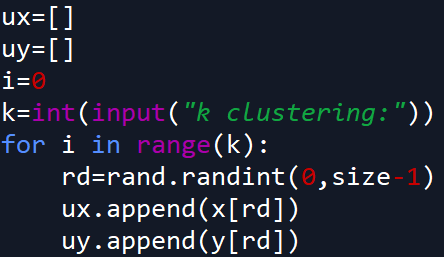
0410024余東儒

1. Kmeans

(1)開檔 將data point分別存入x和y,size存#data points



(2)ux,uy為u的初始值，這邊的初始方法為隨機挑選k個資料點



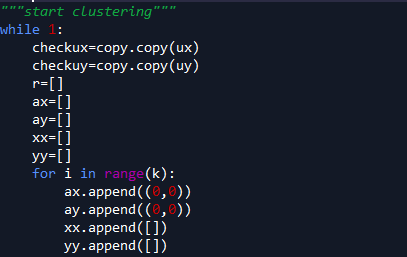
(3)開始kmeans clustering

r為每個資料點的判斷分類

ax,ay為幫助最後計算u的暫存,分別(Σrnkxn,Σrnk)

xx,yy存分類後的資料點,xx[k],yy[k]存分類k的資料點

checkux,checkuy為計算收斂,當checku跟u一樣時代表收斂,結束clustering



(4)開始計算每個data point到u的distance

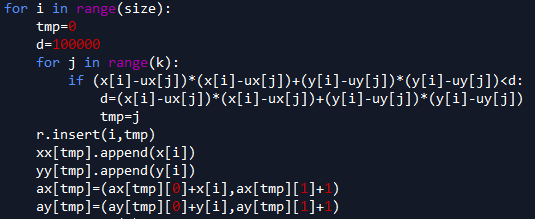
tmp紀錄每個data point離最近u的分類

d為distance初始值設成非常大

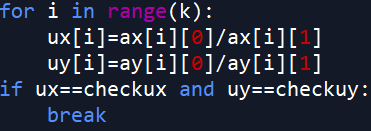
算完之後,r存最後的tmp,

xx.yy將資料點存進第tmp個list,

ax.ay更新



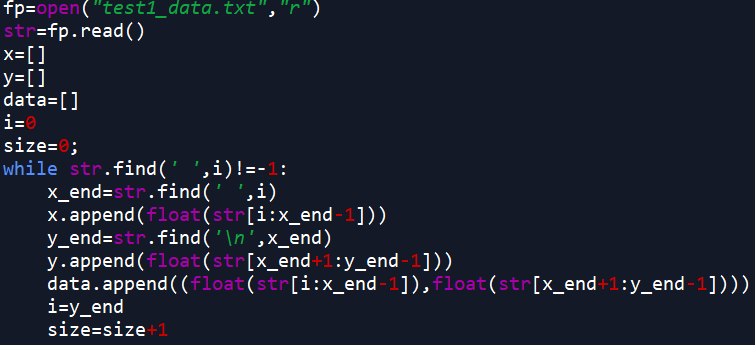
(5)做完step(4)之後,用ax,ay算最新的u

再來判斷是否跟check一樣,如果一樣就結束,不一樣就繼續執行程式 

Result:

|  |  |  |
| --- | --- | --- |
|  | Test1 | Test2 |
| K=2 |  |  |
| K=3 |  |  |
| K=4 |  |  |
| K=8 |  |  |

1. kernel kmeans
2. 跟kmeans一樣先讀檔,存進data list



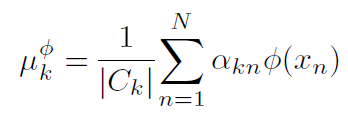
1. Initial

這裡用的初始方法為隨機把所有data points指定成k類

xx和yy一樣是存分類後的data point

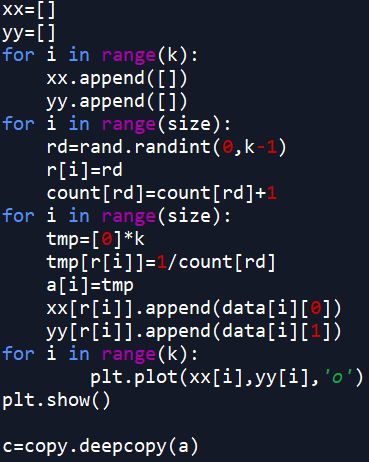
r則是存分類的類別

count紀錄每個類別的數量

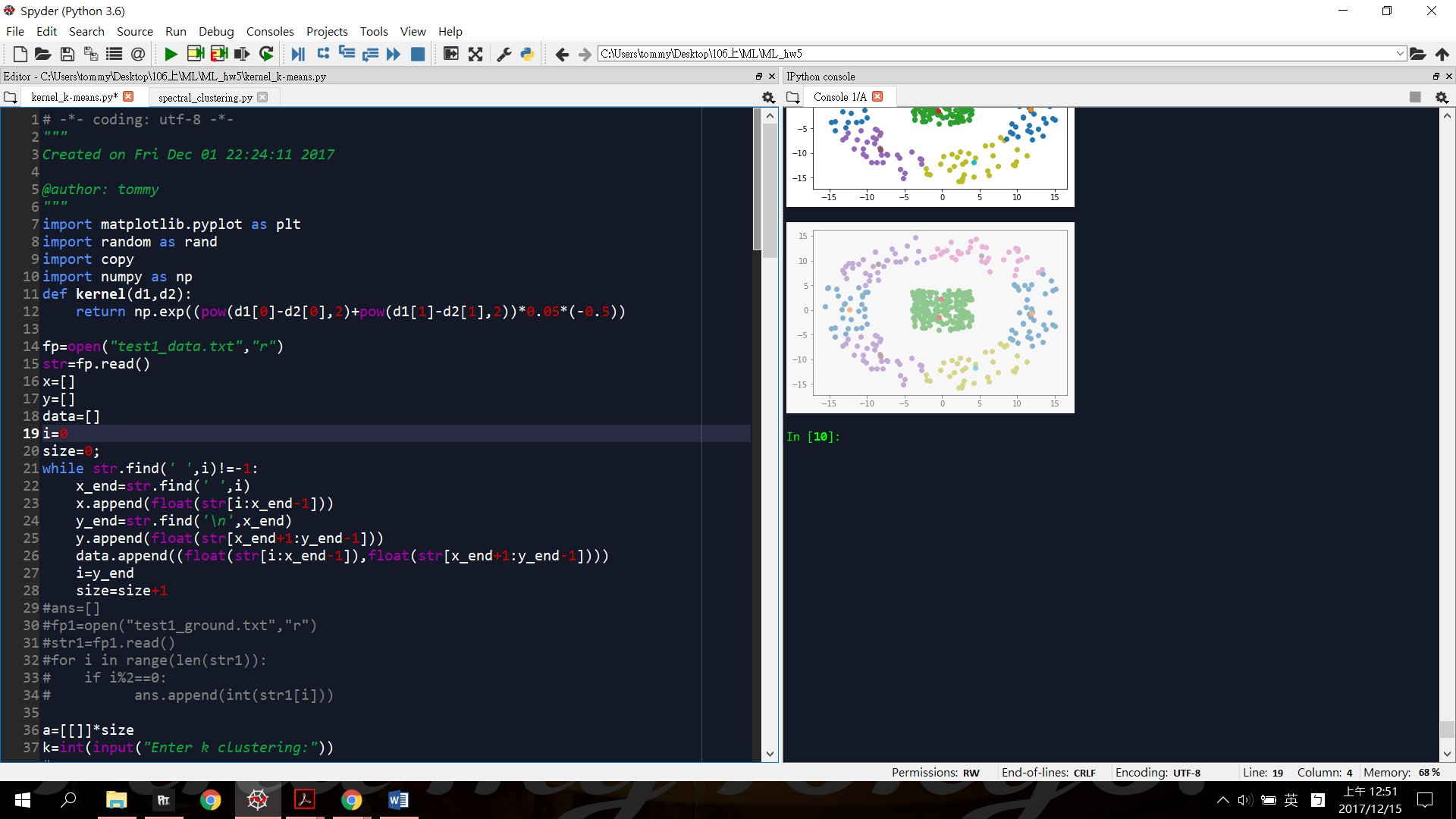


a則是計算u時用的參數,在這裡直接將akn跟ck結合

c則是check的功用,當a和c一樣時則代表收斂



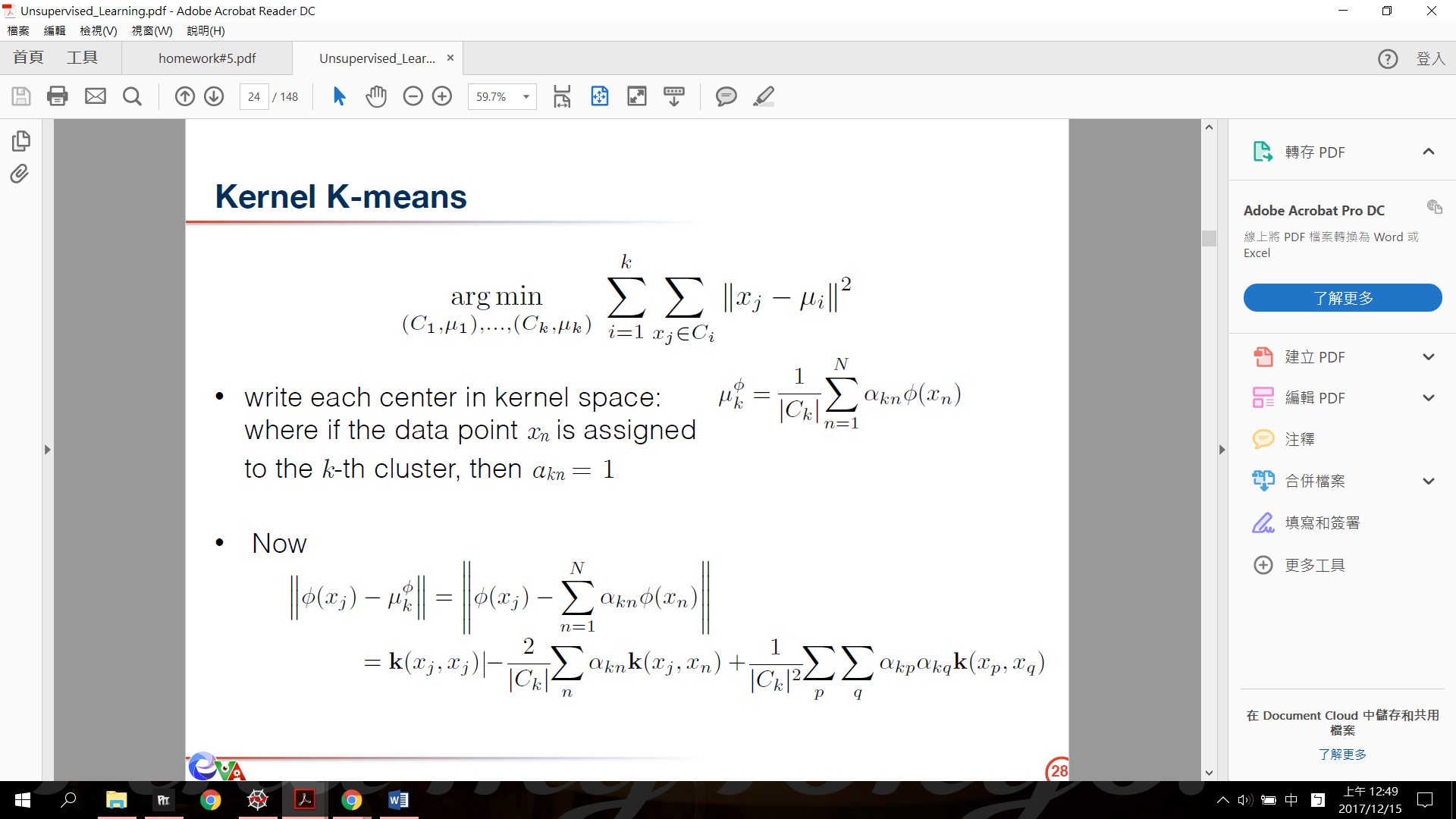
(3) RBF kernel function:d1.d2為兩資料點,

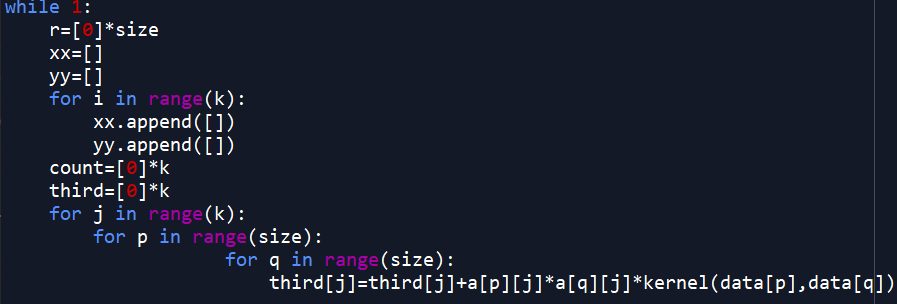


(4)開始進入loop

r,xx,yy每經過一次iteration都要歸零,只有a要留著上次iteraion的結果

third為計算以下式子

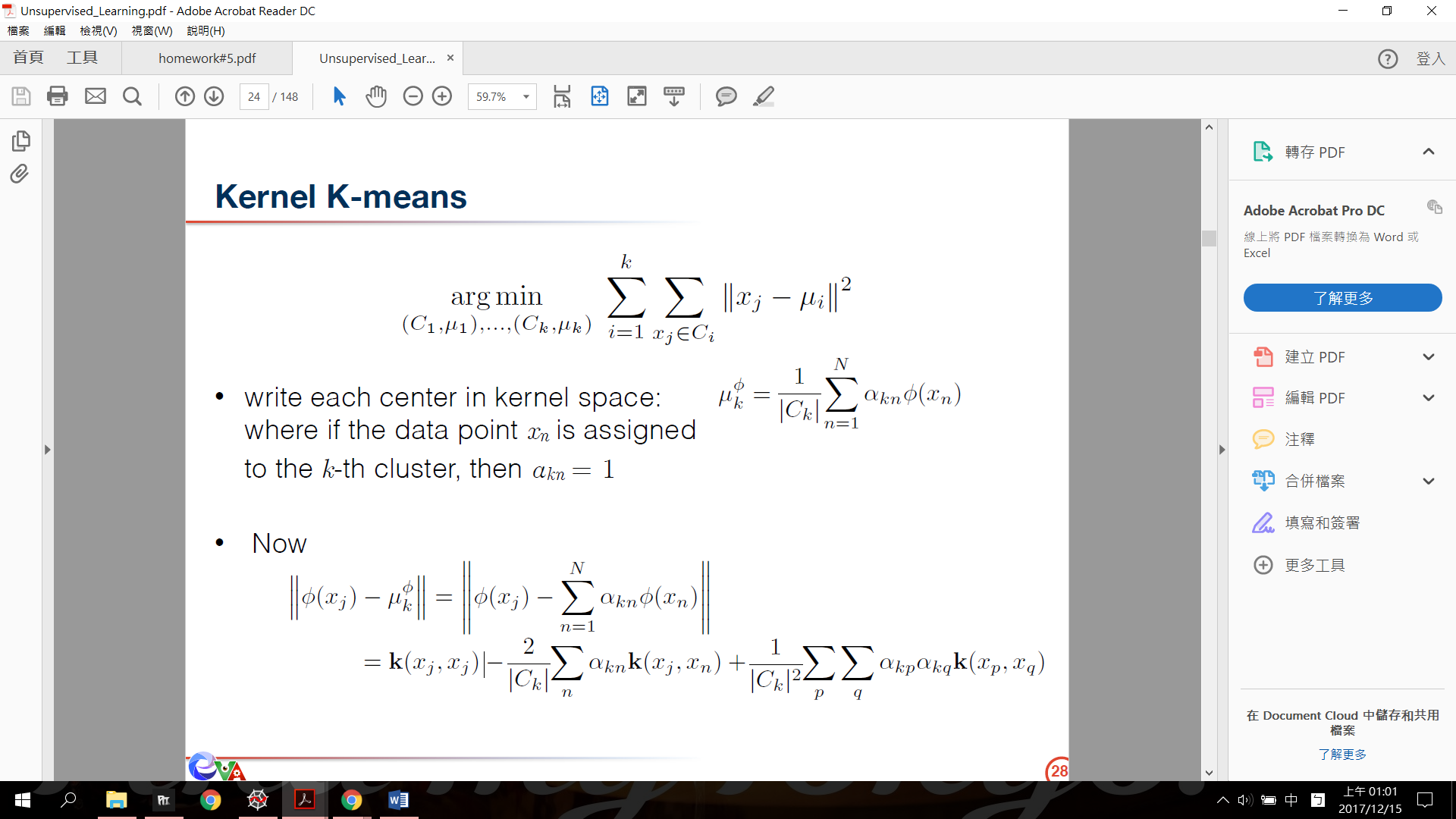




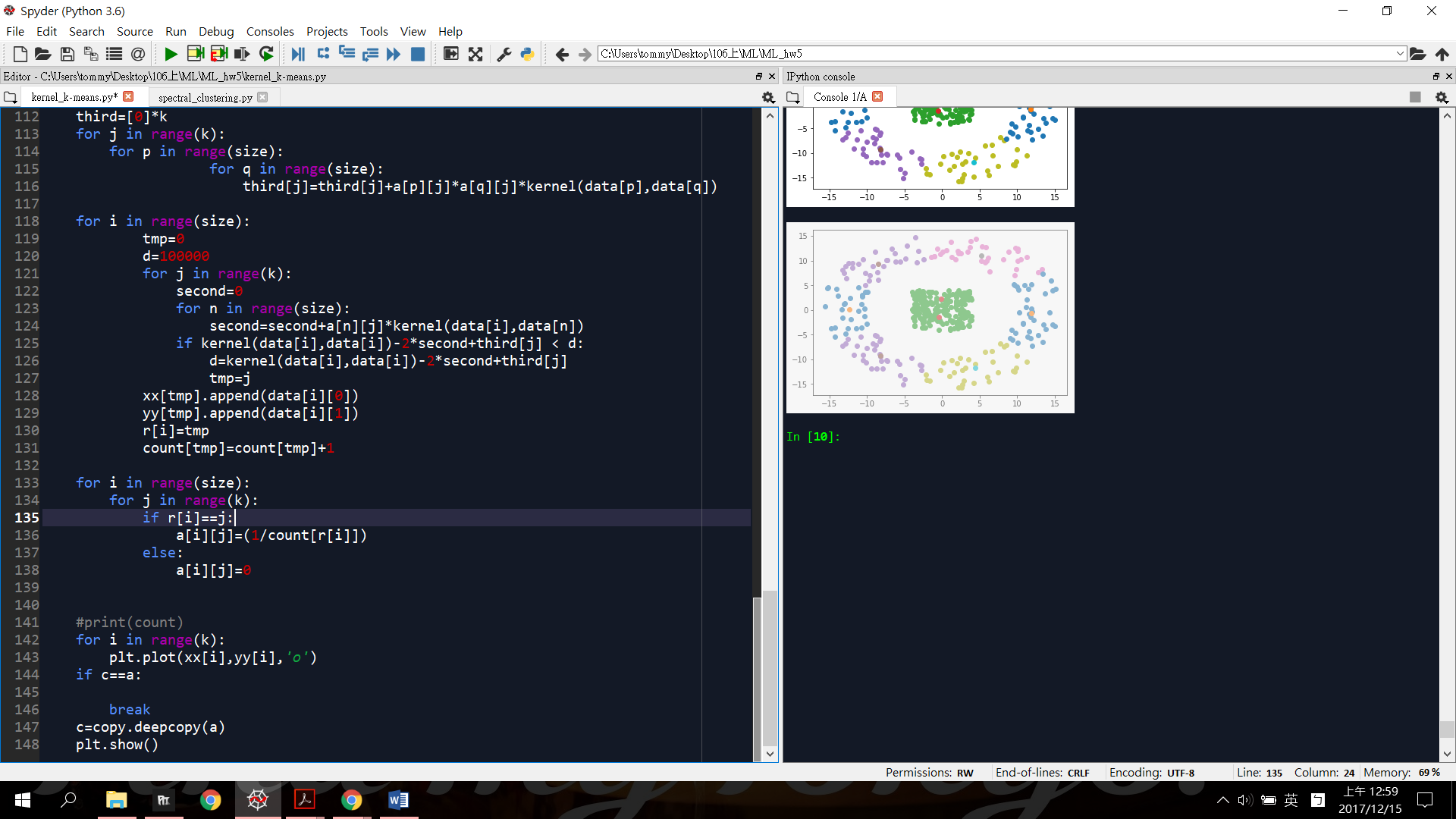
(5)

tmp和d的作用一樣是存分類跟紀錄distance:

second則是



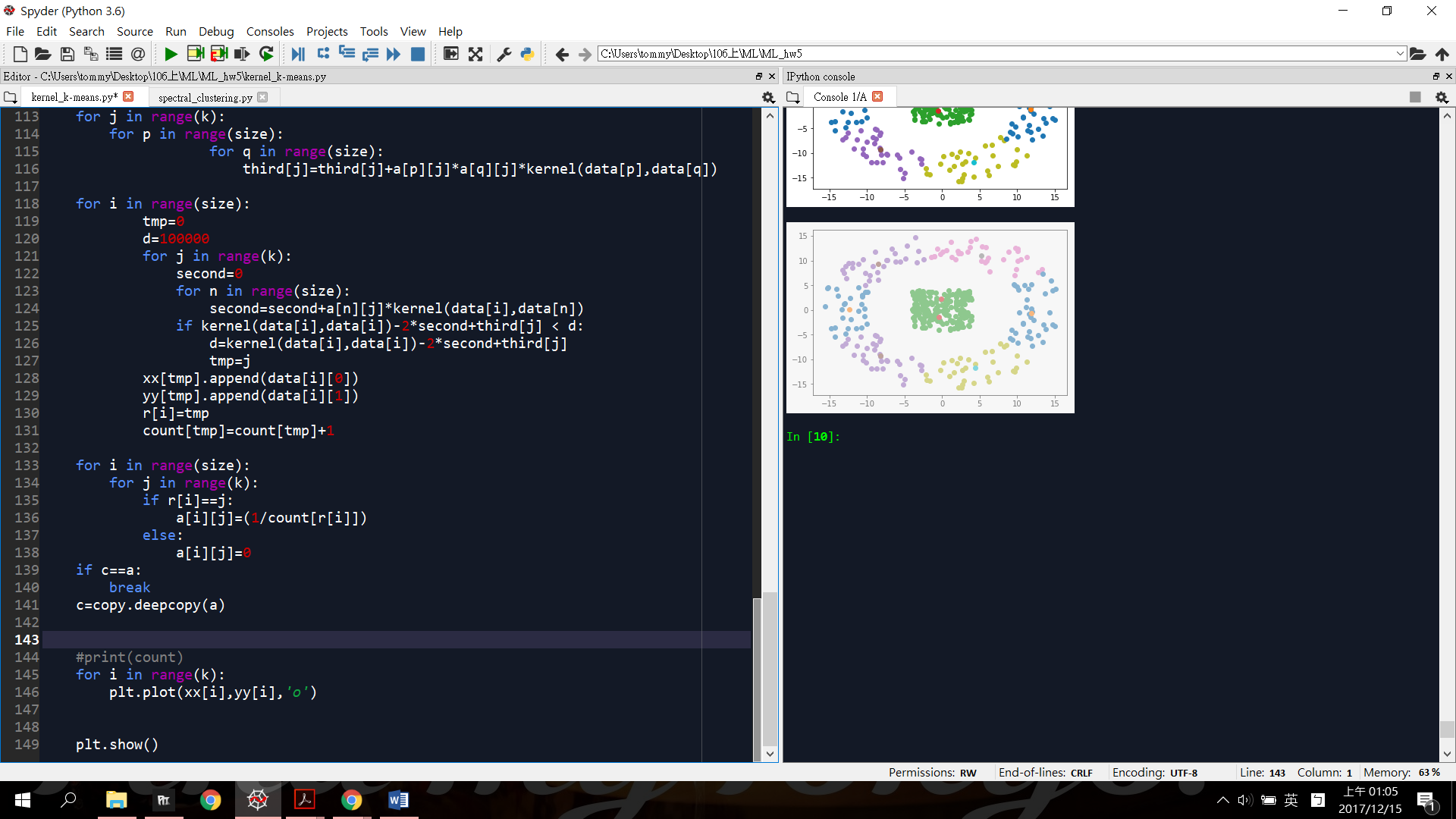
隨後將data point紀錄進xx和yy裡,更新r和count



(6)

用count的結果更新a

判斷c和a是否一樣，如果一樣則代表收斂，不一樣則繼續執行下個iteration

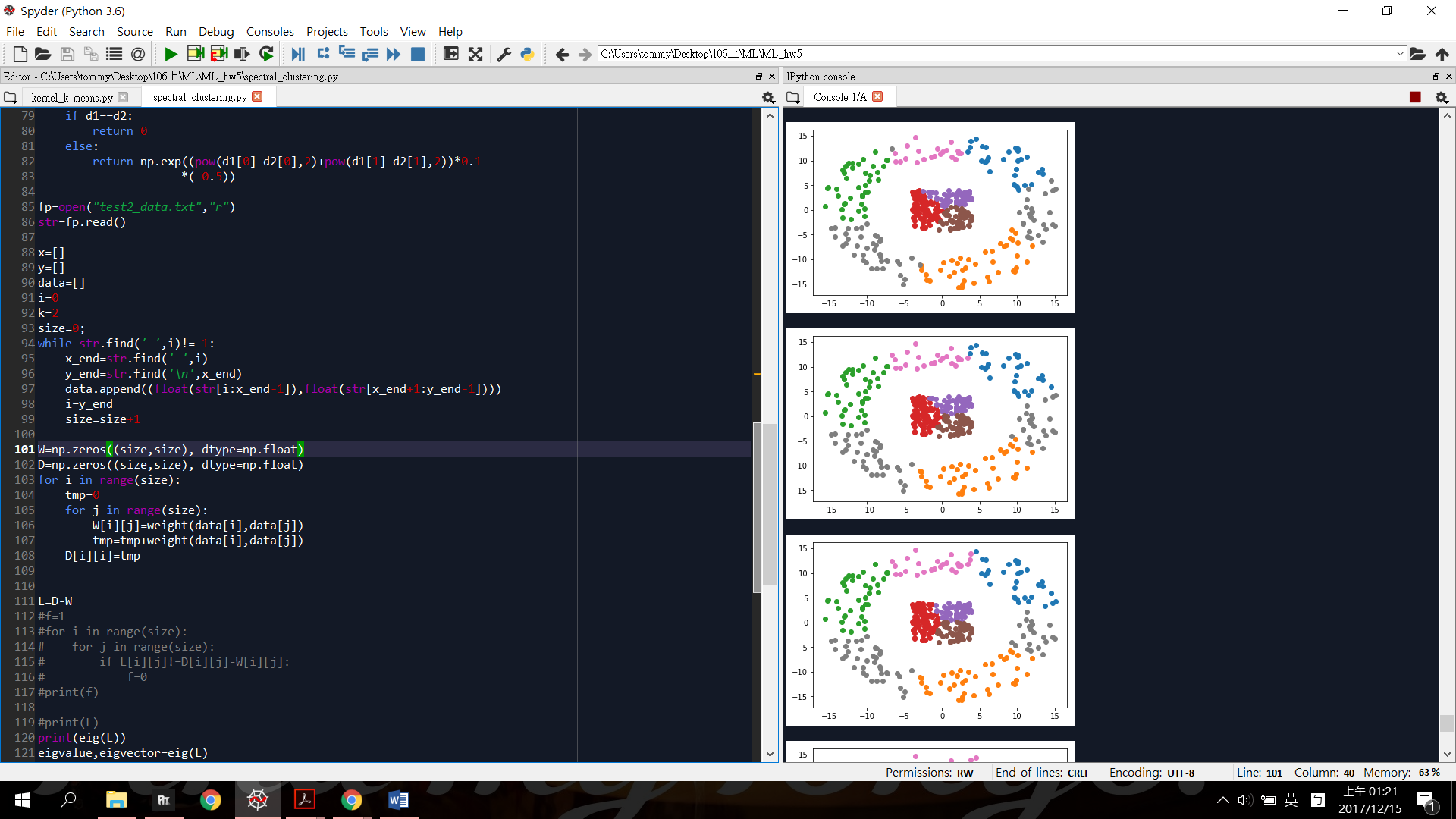


Result:

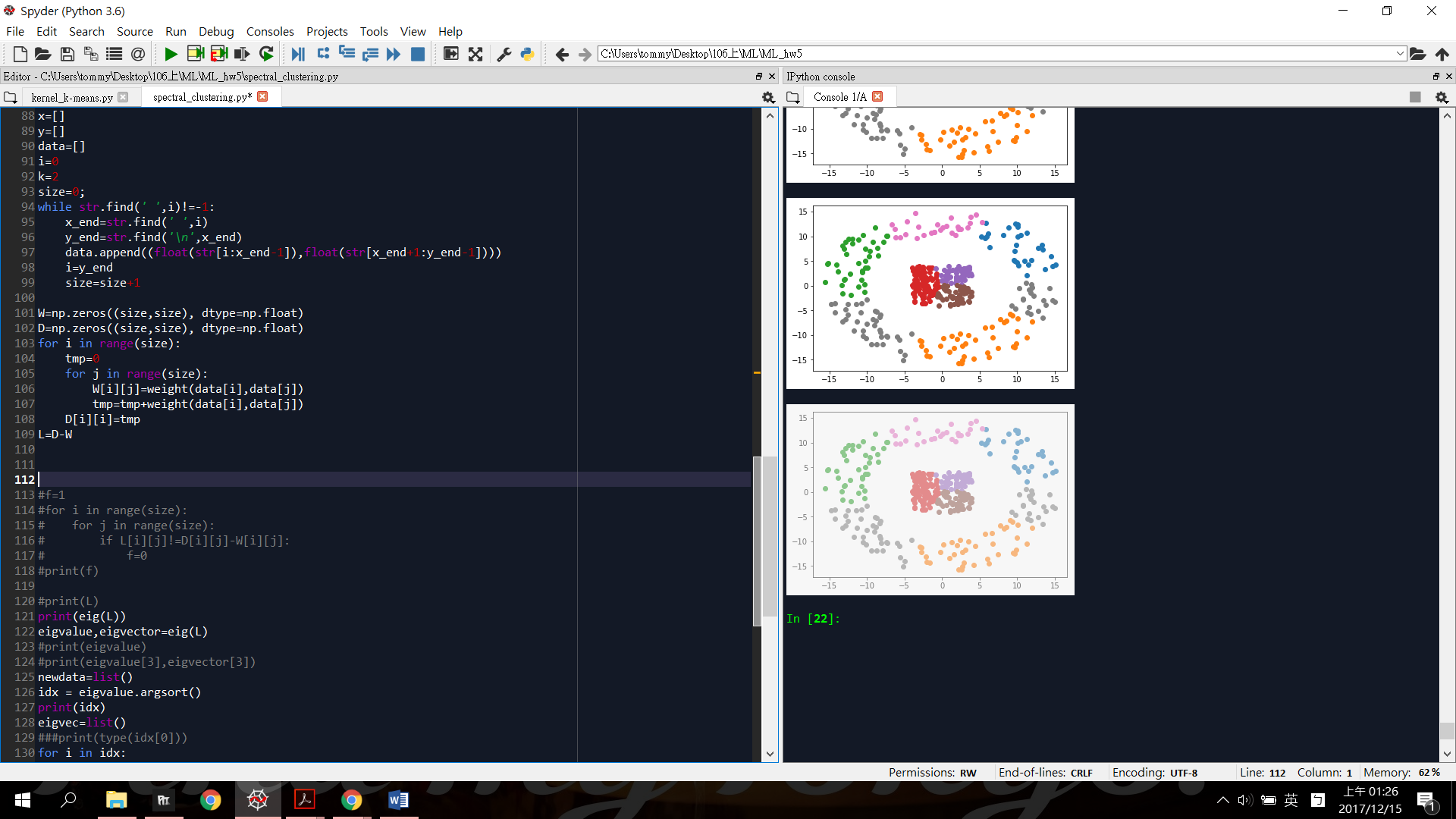
|  |  |  |
| --- | --- | --- |
|  | Test1 | Test2 |
| K=2 |  |  |
| K=3 |  |  |
| K=4 |  |  |
| K=8 |  |  |

1. spectral clustering

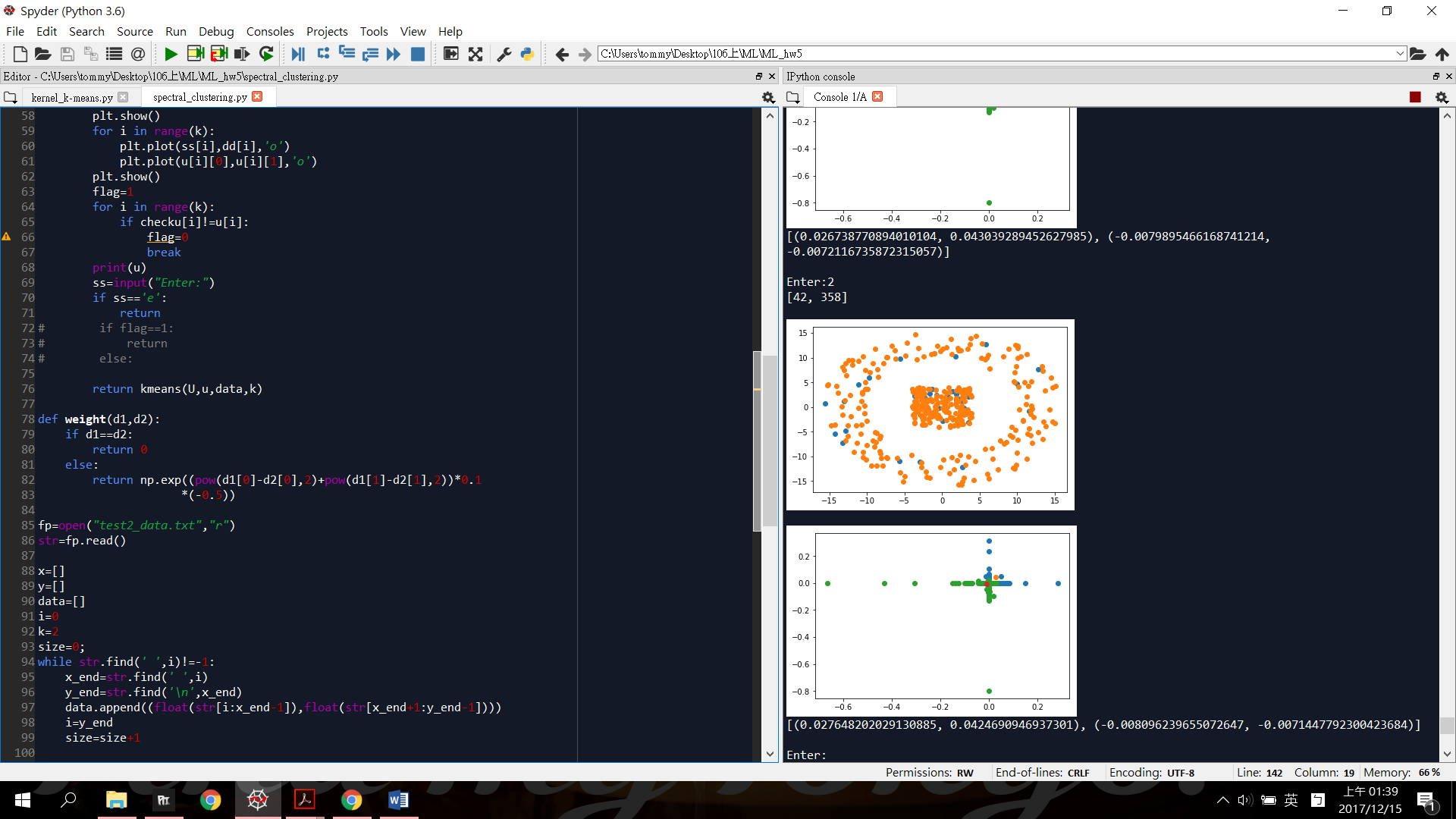
(1)開檔



(2)計算L,D,W矩陣(皆為(size\*size)



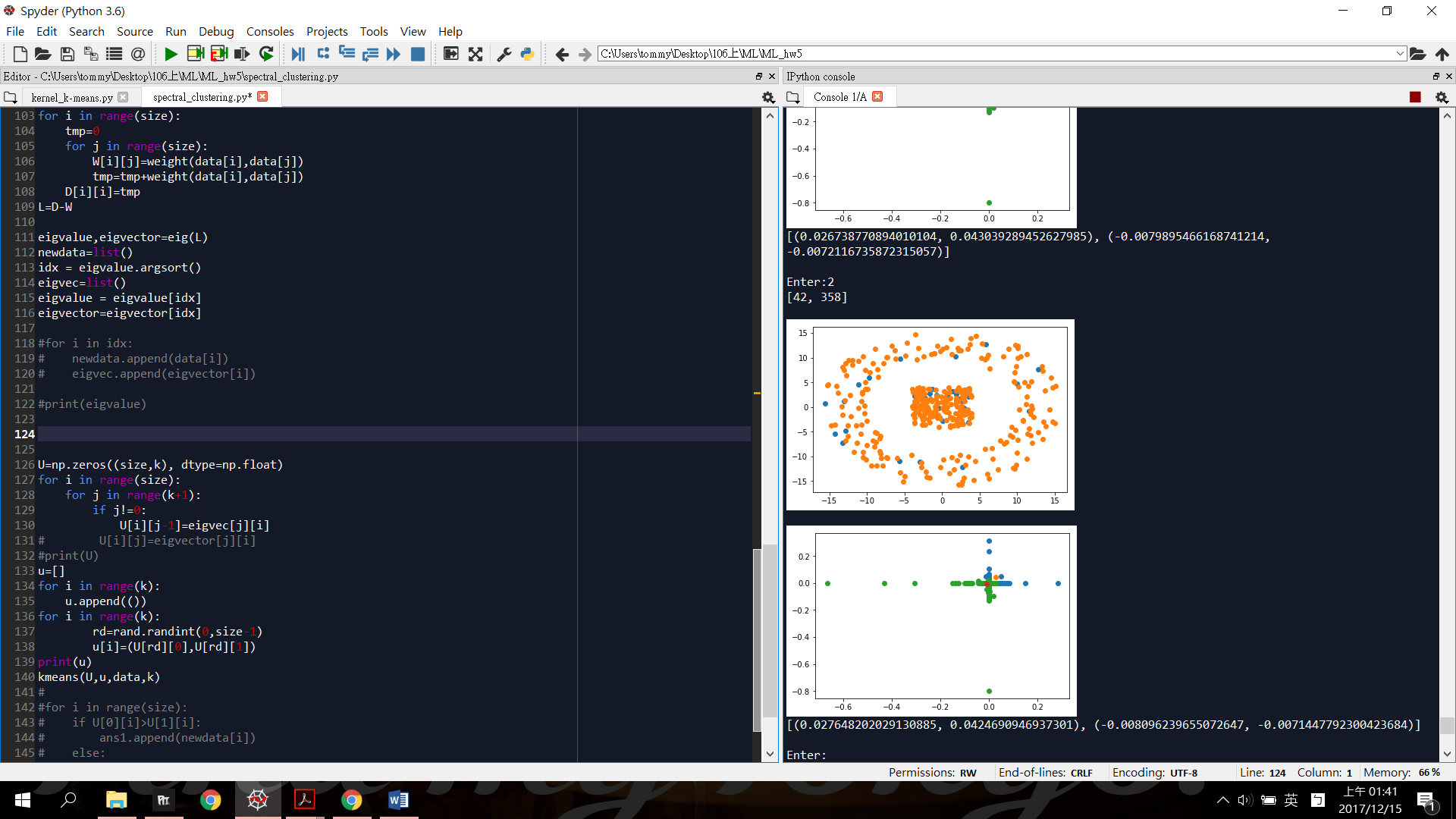
Weight函式一樣是用RBF kernel



1. solve eigen problem

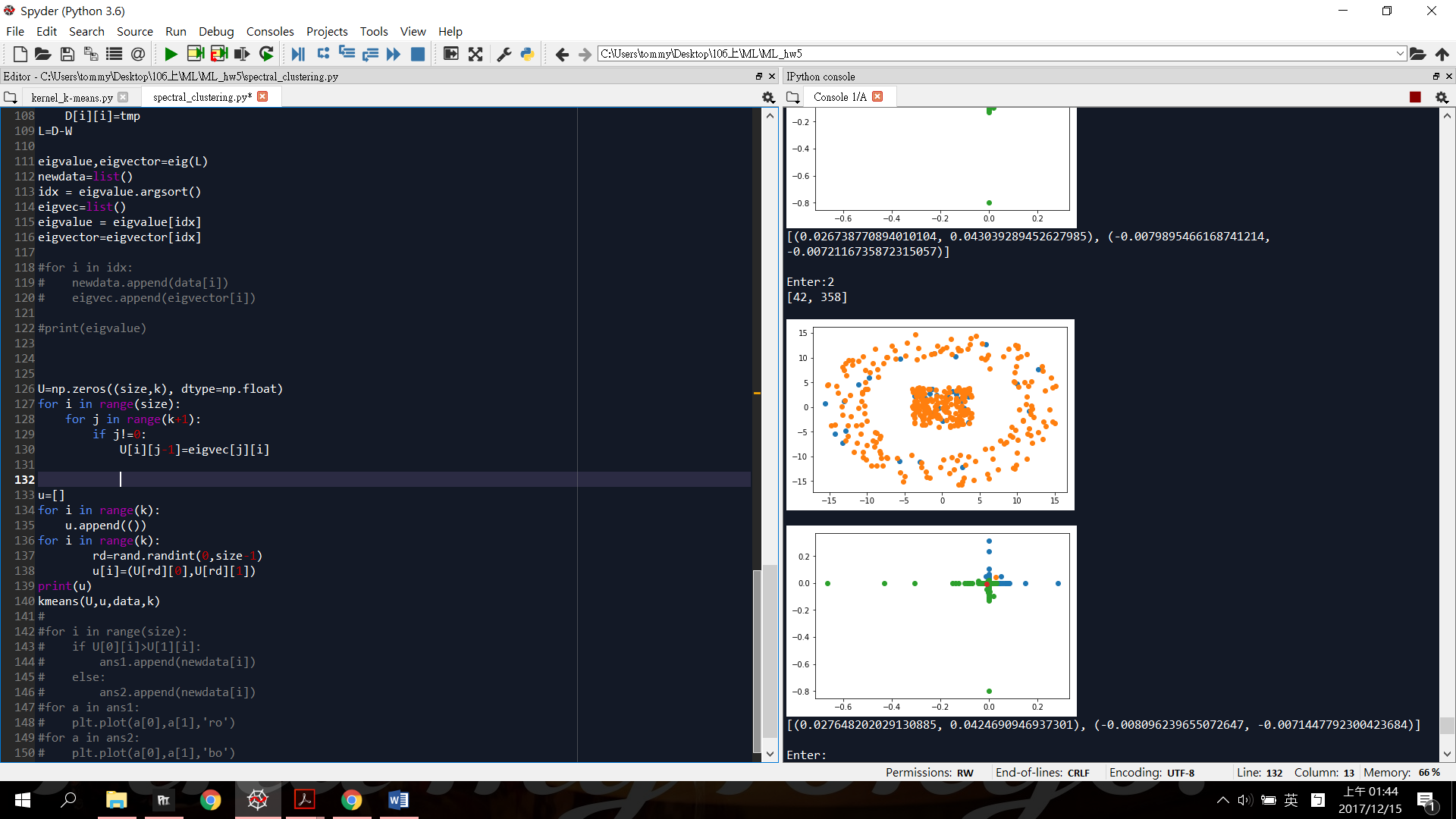
用eig function找eigenvalue和eigenvector

用argsort funtion排序eigen value大小並且其相對應的eigen vector



1. U矩陣(size\*k)

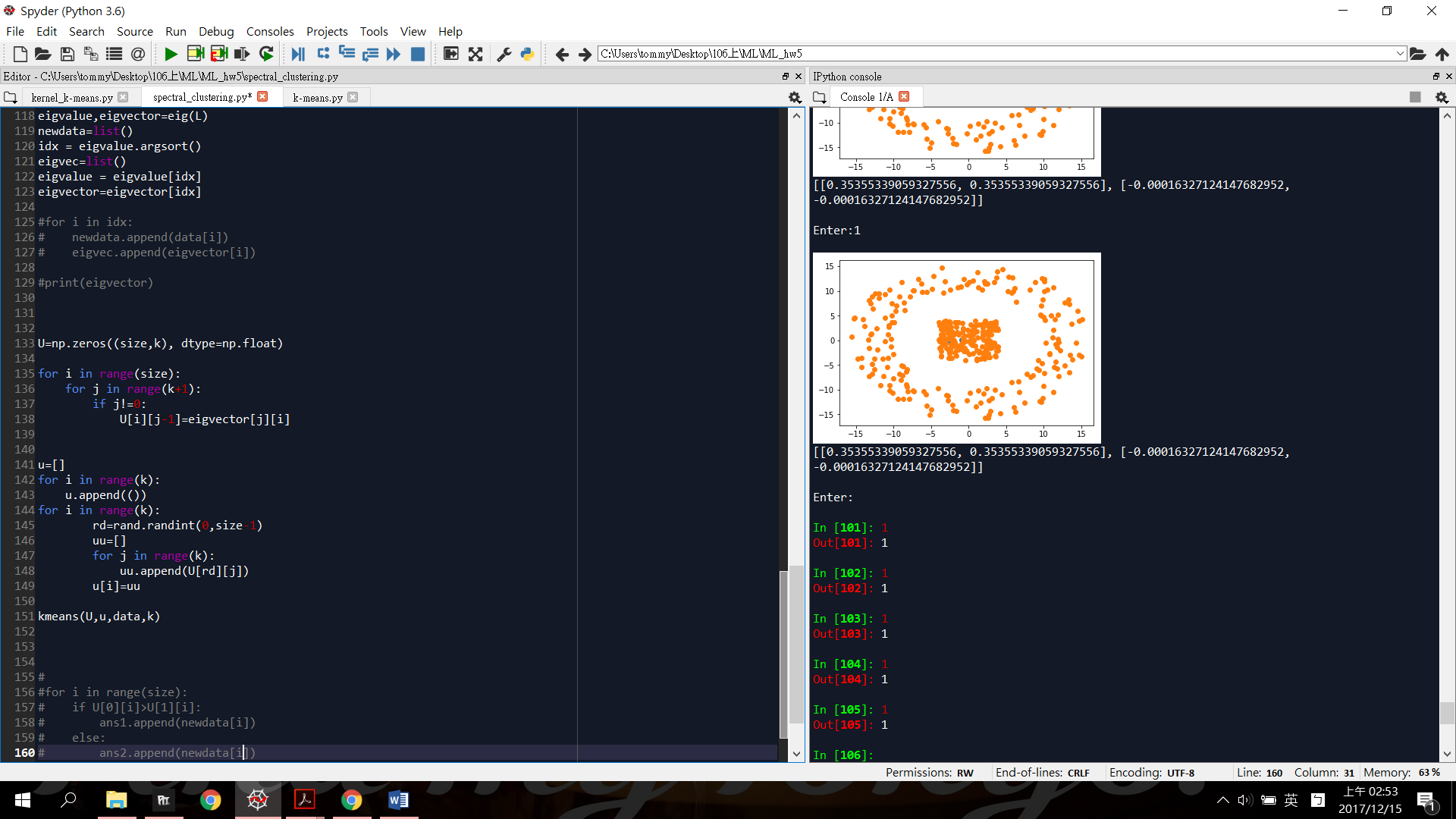
將前k個non-null eigenvalue對應的eigen vector存入U



1. 做kmeans clustering

u為k個分類的初始中心點 u:k\*k

以隨機的方式初始

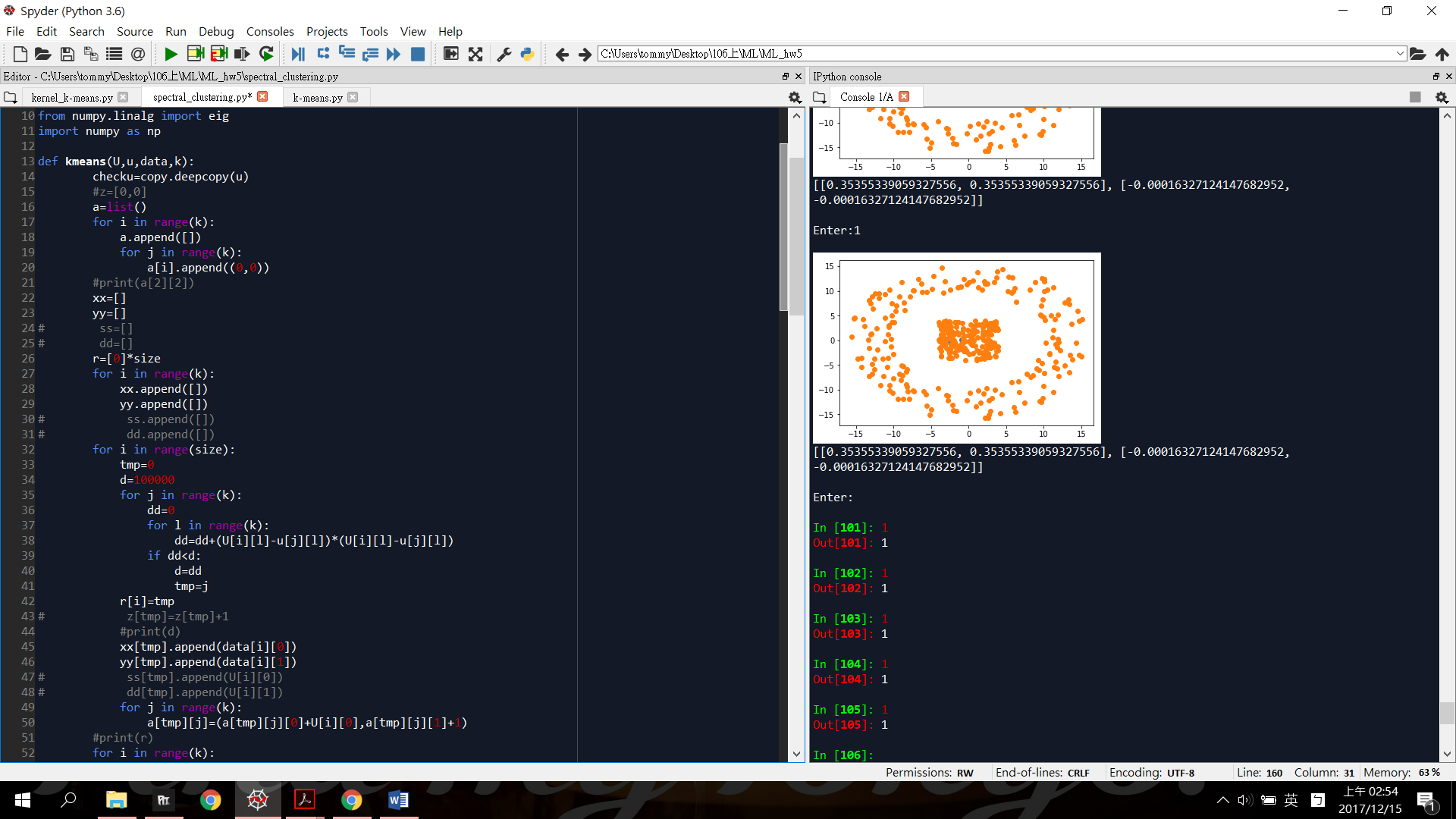


Kmeans function

基本上做法跟第一題一模一樣

只是a的維度變成k\*k\*2

a[分類][axis][計算u的分子和分母]



u一樣是k分類的中心點

flag判斷是否收斂，若收斂則結束，否則繼續kmeans

