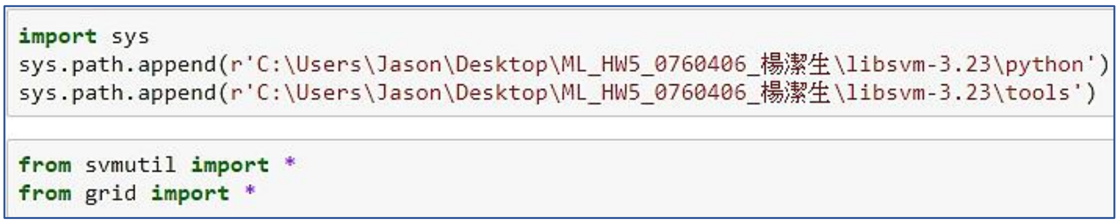
ML_HW05_0760406_楊潔生

SVM on MNIST dataset (60% in total):

Download LIBSVM package and import Libery

If you run my code, please change the path by yourself.

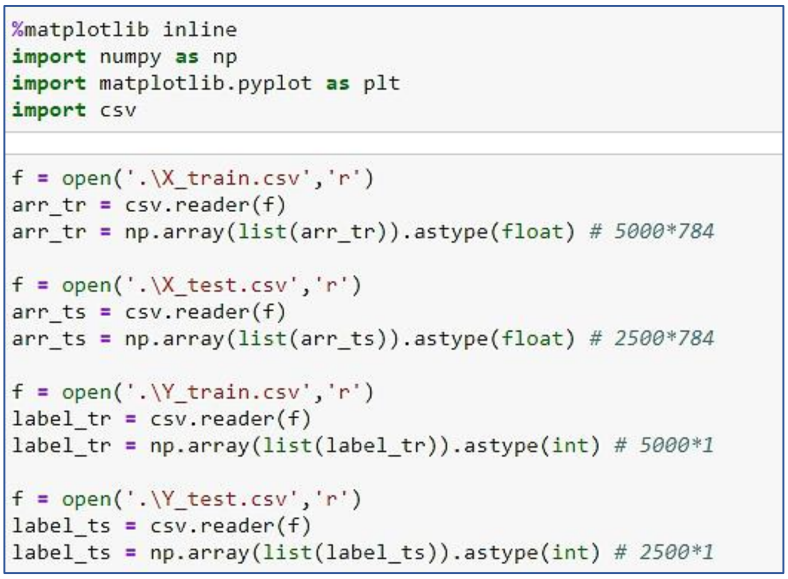


```
import sys
sys.path.append(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\libsvm-3.23\python')
sys.path.append(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\libsvm-3.23\tools')
```

```
from svmutil import *
from grid import *
```

Prepare data in specified format(.libsvm)

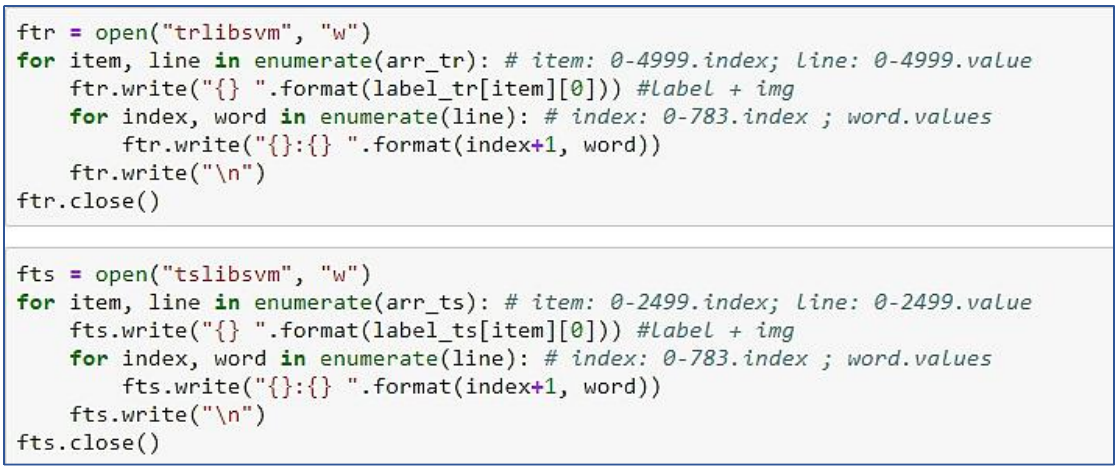
Read csv dataset.



```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import csv
```

```
f = open('.\X_train.csv','r')
arr_tr = csv.reader(f)
arr_tr = np.array(list(arr_tr)).astype(float) # 5000*784

f = open('.\X_test.csv','r')
arr_ts = csv.reader(f)
arr_ts = np.array(list(arr_ts)).astype(float) # 2500*784

f = open('.\Y_train.csv','r')
label_tr = csv.reader(f)
label_tr = np.array(list(label_tr)).astype(int) # 5000*1

f = open('.\Y_test.csv','r')
label_ts = csv.reader(f)
label_ts = np.array(list(label_ts)).astype(int) # 2500*1
```

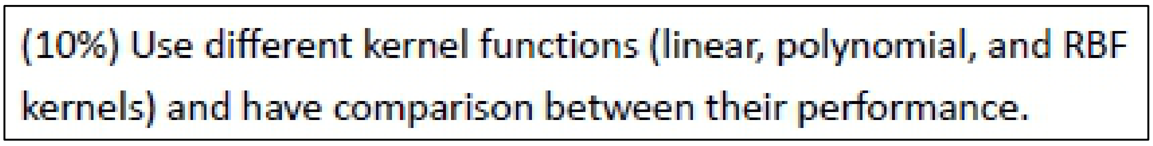
Use *enumerate* function, put train label in to 1st column of train image and add an index to each value.



```
ftr = open("trlibsvm", "w")
for item, line in enumerate(arr_tr): # item: 0-4999.index; line: 0-4999.value
    ftr.write("{} ".format(label_tr[item][0])) #label + img
    for index, word in enumerate(line): # index: 0-783.index ; word.values
        ftr.write("{}:{} ".format(index+1, word))
    ftr.write("\n")
ftr.close()
```

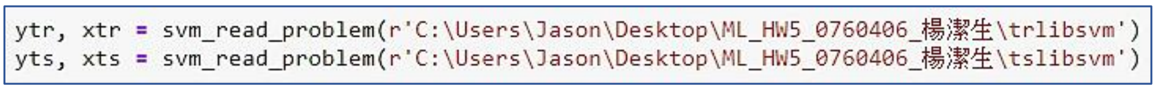
```
fts = open("tslibsvm", "w")
for item, line in enumerate(arr_ts): # item: 0-2499.index; line: 0-2499.value
    fts.write("{} ".format(label_ts[item][0])) #label + img
    for index, word in enumerate(line): # index: 0-783.index ; word.values
        fts.write("{}:{} ".format(index+1, word))
    fts.write("\n")
fts.close()
```

Question 1：


(10%) Use different kernel functions (linear, polynomial, and RBF kernels) and have comparison between their performance.

First step: Read dataset

Read training dataset and testing dataset in libsvm format.
Using command
svm_read_problem() : read the data from a LIBSVM-format file.


```
ytr, xtr = svm_read_problem(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\trlibsvm')
yts, xts = svm_read_problem(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\tslibsvm')
```

Second step: Select parameter

- -s svm_type：set type of SVM (default 0)
  - 0 -- C-SVC
  - 1 -- nu- SVC
  - 2 -- one-class SVM
  - 3 -- epsilon-SVR
  - 4 -- nu-SVR
- -t kernel_type : set type of kernel function (default 2)
  - 0 -- linear: $u' * v$
  - 1 -- polynomial: $(\text{gamma} * u' * \text{v} + \text{coef0})\text{^degree}$
  - 2 -- radial basis function: $\exp(-\text{gamma} * |\text{u} - \text{v}|\text{^}2$
  - 3 -- sigmoid: $\tanh(\text{gamma} * u' * \text{v} + \text{coef0})$
  - 4 -- precomputed kernel (kernel values in training_set_file)

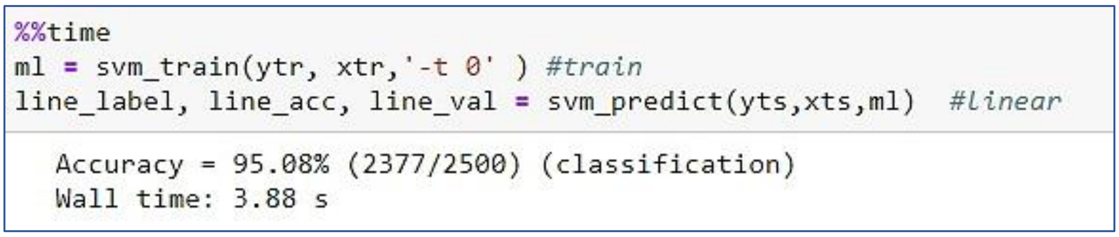
Third step: Using command and variable

Command:
svm_train() : train an SVM model.
svm_predict() : predict testing data.

Variable*:*
(Kernel)_labels: a list of predicted labels.
(Kernel)_acc: a tuple including accuracy (for classification), mean squared error, and squared correlation coefficient (for regression).
(Kernel)_vals: a list of decision values or probability estimates. (if '-b 1' is specified).

For only using default parameter.

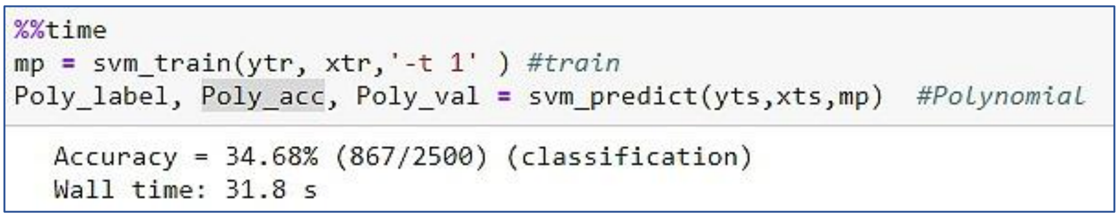
Kernel function – linear



```
%%time
ml = svm_train(ytr, xtr,'-t 0' ) #train
line_label, line_acc, line_val = svm_predict(yts,xts,ml)  #Linear
```

```
  Accuracy = 95.08% (2377/2500) (classification)
  Wall time: 3.88 s
```

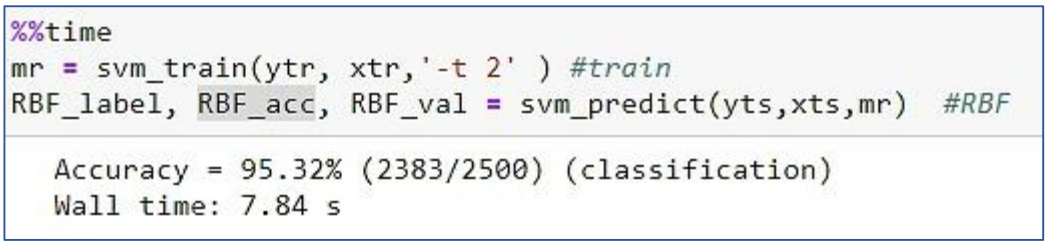
Kernel function – *polynomial*



```
%%time
mp = svm_train(ytr, xtr,'-t 1' ) #train
Poly_label, Poly_acc, Poly_val = svm_predict(yts,xts,mp)  #Polynomial
```

```
  Accuracy = 34.68% (867/2500) (classification)
  Wall time: 31.8 s
```

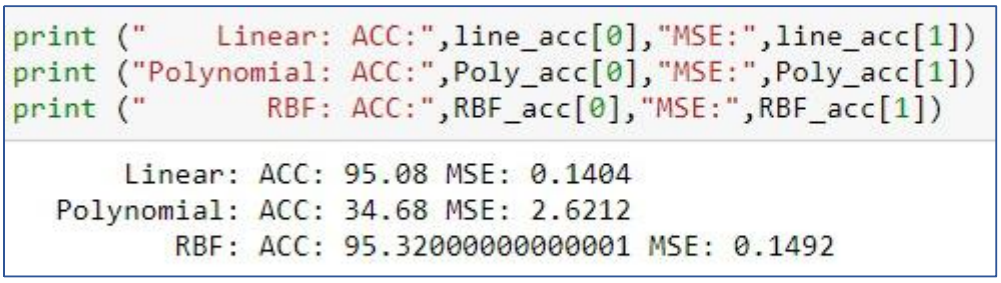
Kernel function – *RBF*



```
%%time
mr = svm_train(ytr, xtr,'-t 2' ) #train
RBF_label, RBF_acc, RBF_val = svm_predict(yts,xts,mr)  #RBF
```

```
  Accuracy = 95.32% (2383/2500) (classification)
  Wall time: 7.84 s
```

3 kinds of kernel's Acc (Accuracy) 、MSE(Mean Square Error)



```
print ("    Linear: ACC:",line_acc[0],"MSE:",line_acc[1])
print ("Polynomial: ACC:",Poly_acc[0],"MSE:",Poly_acc[1])
print ("       RBF: ACC:",RBF_acc[0],"MSE:",RBF_acc[1])
```

```
      Linear: ACC: 95.08 MSE: 0.1404
  Polynomial: ACC: 34.68 MSE: 2.6212
         RBF: ACC: 95.32000000000001 MSE: 0.1492
```

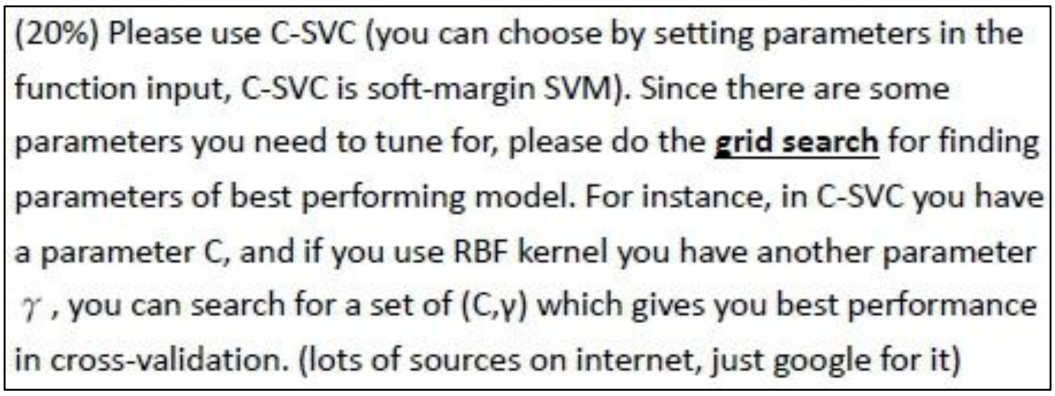
Discussion:

If we choice "*polynomial*", it takes the most time and has lowest ACC. In this case, RBF model has the best perform under the premise of using default parameters

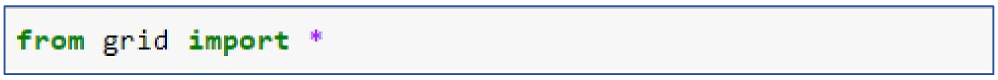
Table 1 Compare different kernel function with default parameter

Kernel function	*Linear*	*Polynomial*	*RBF*
Acc (%)	95.08	34.68	95.32
MSE (%)	0.1404	2.6212	0.1492
Time (s)	3.88	31.8	7.84

Question 2：


(20%) Please use C-SVC (you can choose by setting parameters in the function input, C-SVC is soft-margin SVM). Since there are some parameters you need to tune for, please do the **grid search** for finding parameters of best performing model. For instance, in C-SVC you have a parameter C, and if you use RBF kernel you have another parameter γ , you can search for a set of (C,γ) which gives you best performance in cross-validation. (lots of sources on internet, just google for it)

First step: Import grid function

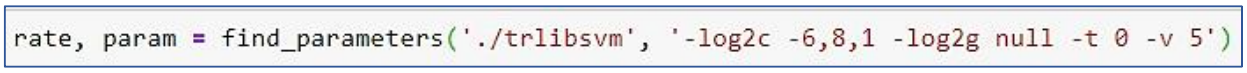

```
from grid import *
```

Second step: Grid options

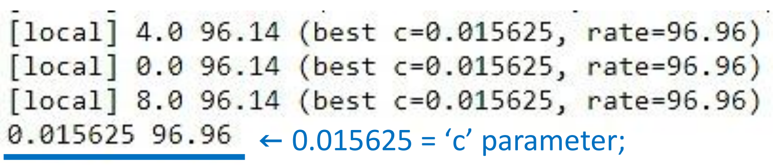
- -log2c {begin, end, step | "null"}: set the range of c (default -5,15,2)
  - begin, end, step – c range = 2^ {begin, ..., begin + k* step, ..., end}
  - "null" – do not grid with c
- -log2g {begin, end, step | "null"}: set the range of g (default 3, -15, -2)
  - begin, end, step – g range = 2^ {begin, ..., begin + k*step, ..., end}
  - "null" – do not grid with g
- -v n: n-fold cross validation (default 5)
- -svm train pathname: set SVM executable path and name

Third step: Using *find_parameters* function

Kernel function – *linear*

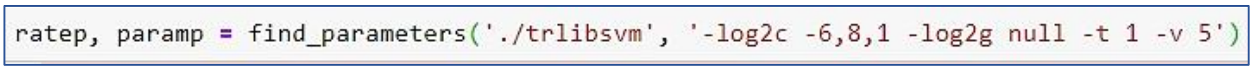

```
rate, param = find_parameters('./trlibsvm', '-log2c -6,8,1 -log2g null -t 0 -v 5')
```

Output & Parameter:


```
[local] 4.0 96.14 (best c=0.015625, rate=96.96)
[local] 0.0 96.14 (best c=0.015625, rate=96.96)
[local] 8.0 96.14 (best c=0.015625, rate=96.96)
0.015625 96.96
```

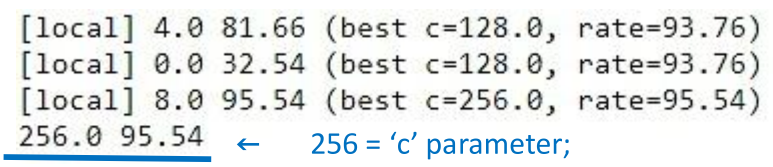
← 0.015625 = ‘c’ parameter;
96.96 = accuracy (%)

Kernel function – *polynomial*



```
ratep, paramp = find_parameters('./trlibsvm', '-log2c -6,8,1 -log2g null -t 1 -v 5')
```

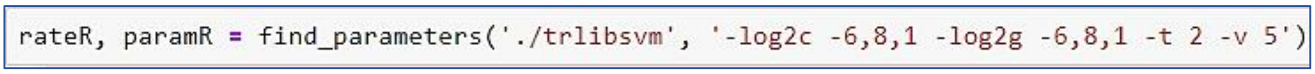
Output & Parameter:



```
[local] 4.0 81.66 (best c=128.0, rate=93.76)
[local] 0.0 32.54 (best c=128.0, rate=93.76)
[local] 8.0 95.54 (best c=256.0, rate=95.54)
256.0 95.54
```

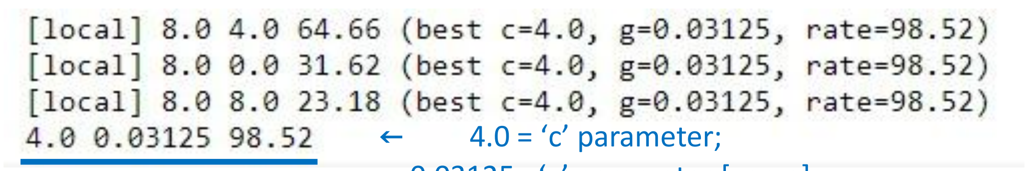
← 256 = 'c' parameter;
95.54 = accuracy (%)

Kernel function – *RBF*



```
rateR, paramR = find_parameters('./trlibsvm', '-log2c -6,8,1 -log2g -6,8,1 -t 2 -v 5')
```

Output & Parameter:



```
[local] 8.0 4.0 64.66 (best c=4.0, g=0.03125, rate=98.52)
[local] 8.0 0.0 31.62 (best c=4.0, g=0.03125, rate=98.52)
[local] 8.0 8.0 23.18 (best c=4.0, g=0.03125, rate=98.52)
4.0 0.03125 98.52
```

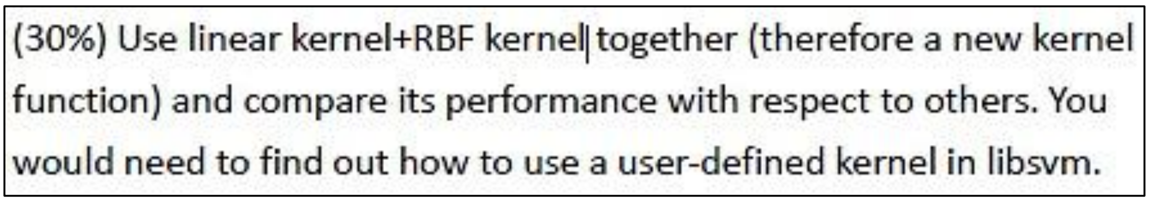
← 4.0 = 'c' parameter;
0.03125= 'g' parameter [gama]
98.52 = accuracy (%)

Discussion:

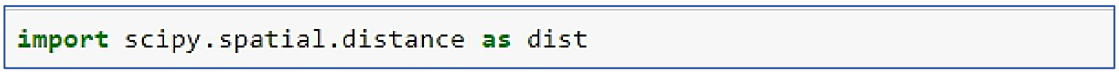
When we using the best parameter, it can greatly increase the original low accuracy such as "*polynomial*", which accuracy increased from 31.8% to 95.54%.

According to the literature, you will only modify two important arguments when you are using training with data: *gamma (-g)* and *cost (-c)*. And cross validation (-v) is usually set to 5. Then how do we know what value to choose as arguments? Just by trial and error.

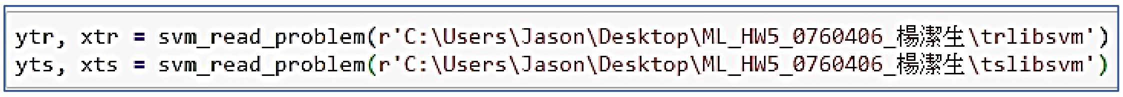
Question 3：


(30%) Use linear kernel+RBF kernel together (therefore a new kernel function) and compare its performance with respect to others. You would need to find out how to use a user-defined kernel in libsvm.

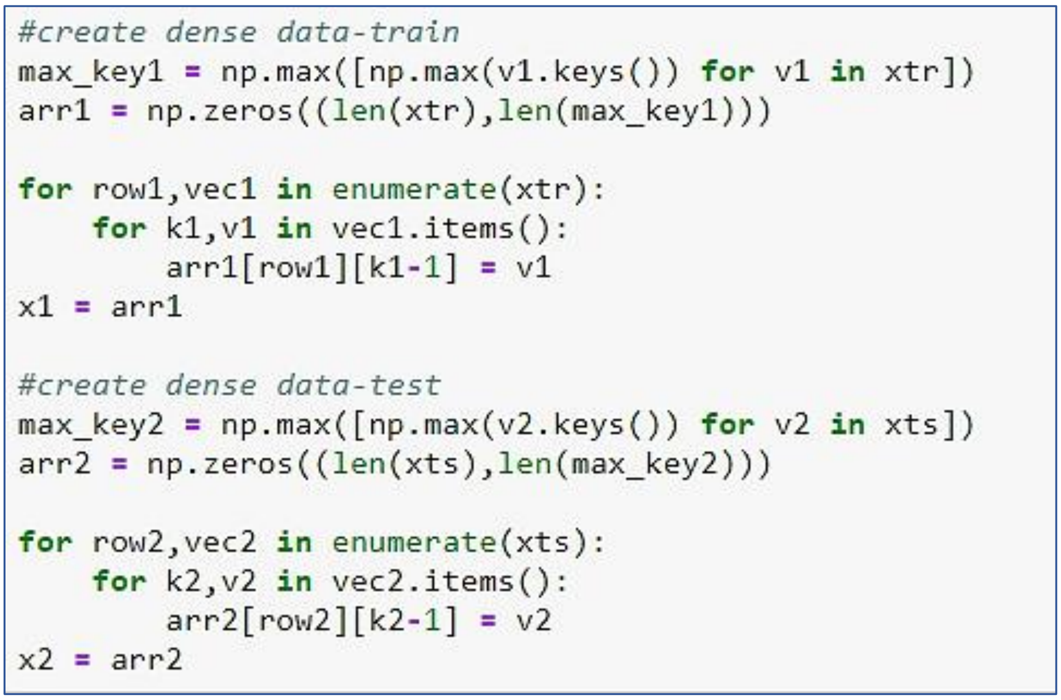
First step: Import distance function


```
import scipy.spatial.distance as dist
```

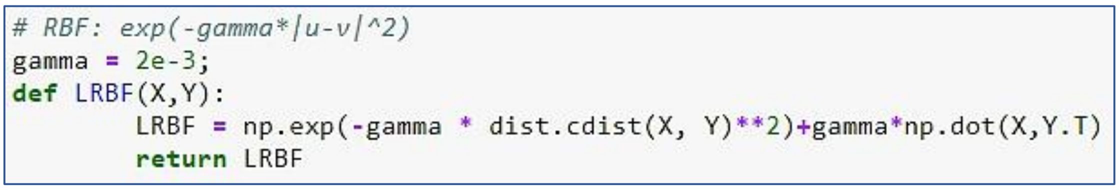
Second step: Read dataset (format: libsvm)


```
ytr, xtr = svm_read_problem(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\trlibsvm')
yts, xts = svm_read_problem(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\tslibsvm')
```

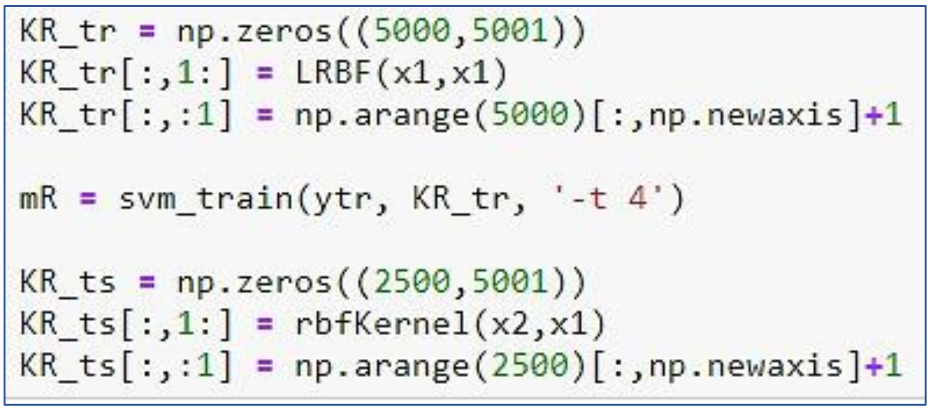
Third step: Create dense dataset


```
#create dense data-train
max_key1 = np.max([np.max(v1.keys()) for v1 in xtr])
arr1 = np.zeros((len(xtr),len(max_key1)))

for row1,vec1 in enumerate(xtr):
    for k1,v1 in vec1.items():
        arr1[row1][k1-1] = v1
x1 = arr1

#create dense data-test
max_key2 = np.max([np.max(v2.keys()) for v2 in xts])
arr2 = np.zeros((len(xts),len(max_key2)))

for row2,vec2 in enumerate(xts):
    for k2,v2 in vec2.items():
        arr2[row2][k2-1] = v2
x2 = arr2
```

Fourth step: Define Linear + RBF Kernel function

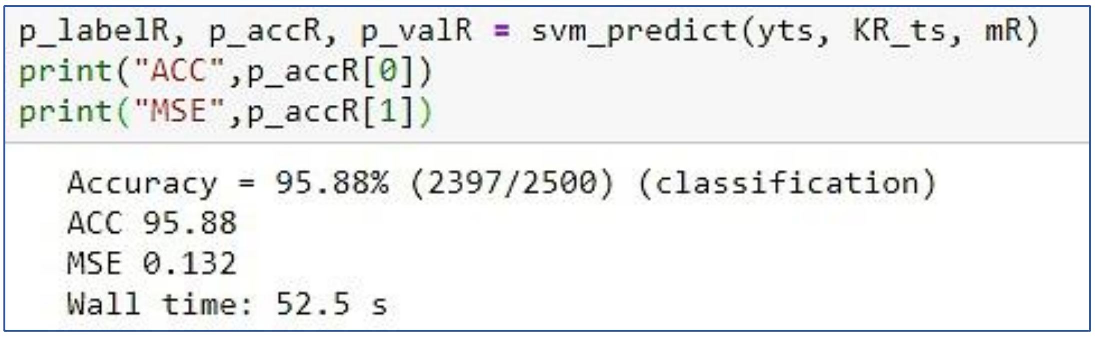

```
# RBF: exp(-gamma*|u-v|^2)
gamma = 2e-3;
def LRBF(X,Y):
        LRBF = np.exp(-gamma * dist.cdist(X, Y)**2)+gamma*np.dot(X,Y.T)
        return LRBF
```

Fourth step: Create a Linear + RBF kernel matrix



```
KR_tr = np.zeros((5000,5001))
KR_tr[:,1:] = LRBF(x1,x1)
KR_tr[:,:1] = np.arange(5000)[:,np.newaxis]+1

mR = svm_train(ytr, KR_tr, '-t 4')

KR_ts = np.zeros((2500,5001))
KR_ts[:,1:] = rbfKernel(x2,x1)
KR_ts[:,:1] = np.arange(2500)[:,np.newaxis]+1
```

Fifth step: Create a Linear + RBF kernel matrix



```
p_labelR, p_accR, p_valR = svm_predict(yts, KR_ts, mR)
print("ACC",p_accR[0])
print("MSE",p_accR[1])
```

```
Accuracy = 95.88% (2397/2500) (classification)
ACC 95.88
MSE 0.132
Wall time: 52.5 s
```

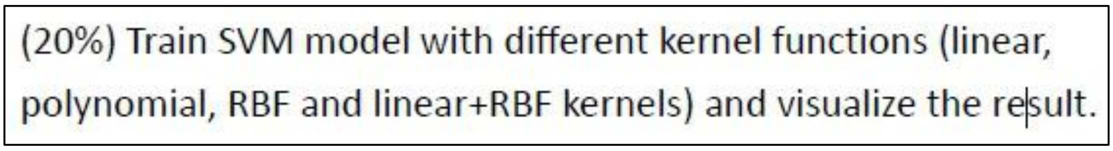
Discussion:

When every kernel type is using default parameter, precomputed kernel- "*Linear+ RBF*" has the best performance. But relative, it takes about 13.5 times longer than the linear Kernel.

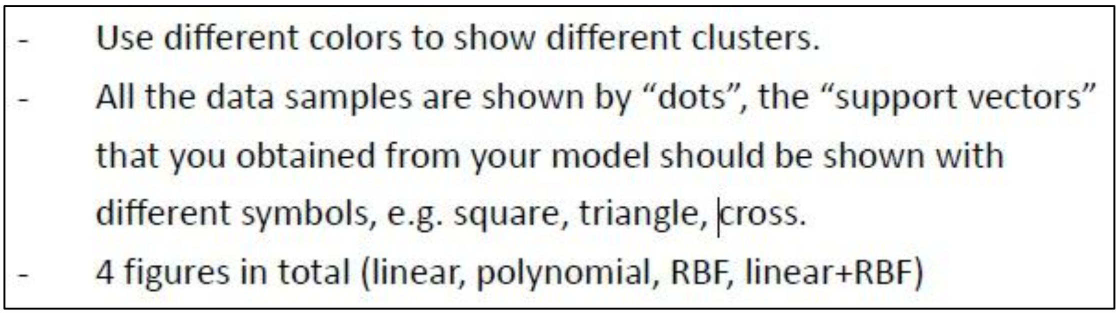
Table 2 Compare different kernel function

Kernel function	*Linear*	*Polynomial*	*RBF*	*Linear+ RBF*
Acc (%)	95.08	34.68	95.32	95.88
MSE (%)	0.1404	2.6212	0.1492	0.132
Time (s)	3.88	31.8	7.84	52.5

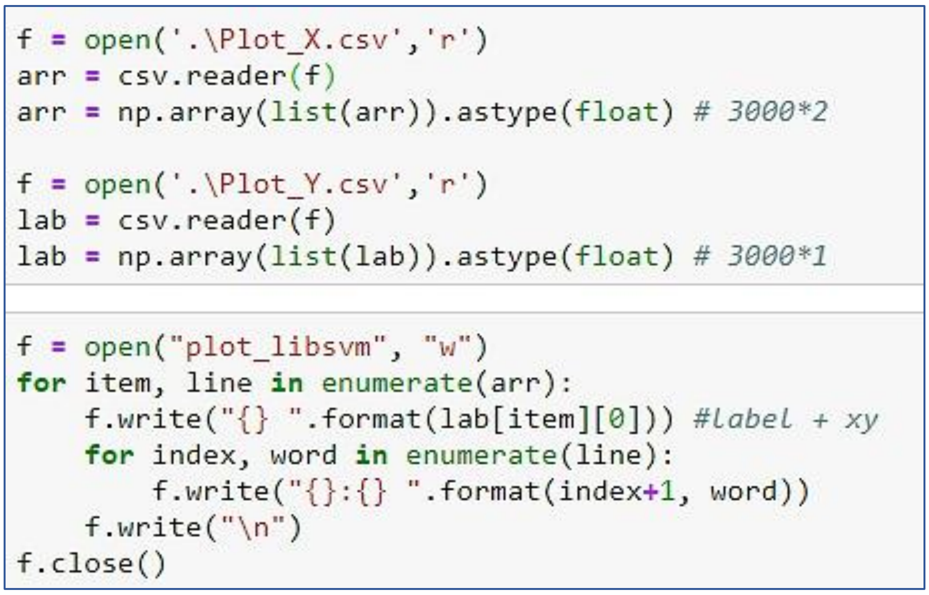
Find out support vectors (20% in total):


(20%) Train SVM model with different kernel functions (linear, polynomial, RBF and linear+RBF kernels) and visualize the result.

Rule：

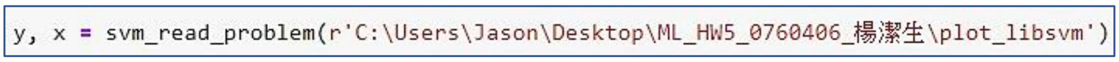

- Use different colors to show different clusters.
- All the data samples are shown by “dots”, the “support vectors” that you obtained from your model should be shown with different symbols, e.g. square, triangle, cross.
- 4 figures in total (linear, polynomial, RBF, linear+RBF)

First step: Prepare data in specified format(.libsvm)


```
f = open('.\Plot_X.csv','r')
arr = csv.reader(f)
arr = np.array(list(arr)).astype(float) # 3000*2

f = open('.\Plot_Y.csv','r')
lab = csv.reader(f)
lab = np.array(list(lab)).astype(float) # 3000*1
```

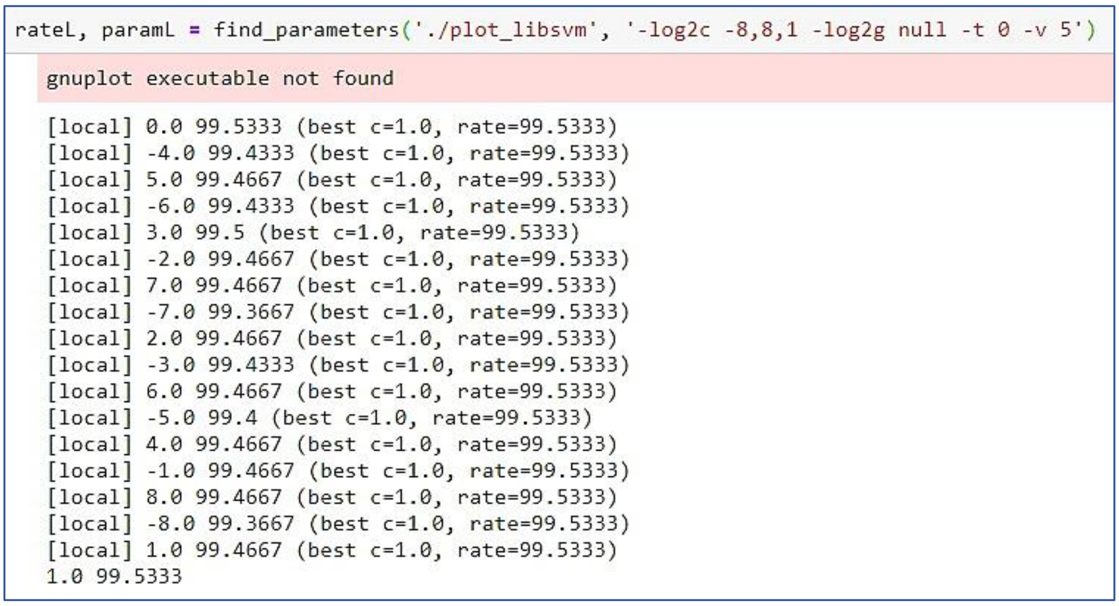
```
f = open("plot_libsvm", "w")
for item, line in enumerate(arr):
    f.write("{} ".format(lab[item][0])) #label + xy
    for index, word in enumerate(line):
        f.write("{}:{} ".format(index+1, word))
    f.write("\n")
f.close()
```

Second step: Read dateset


```
y, x = svm_read_problem(r'C:\Users\Jason\Desktop\ML_HW5_0760406_楊潔生\plot_libsvm')
```

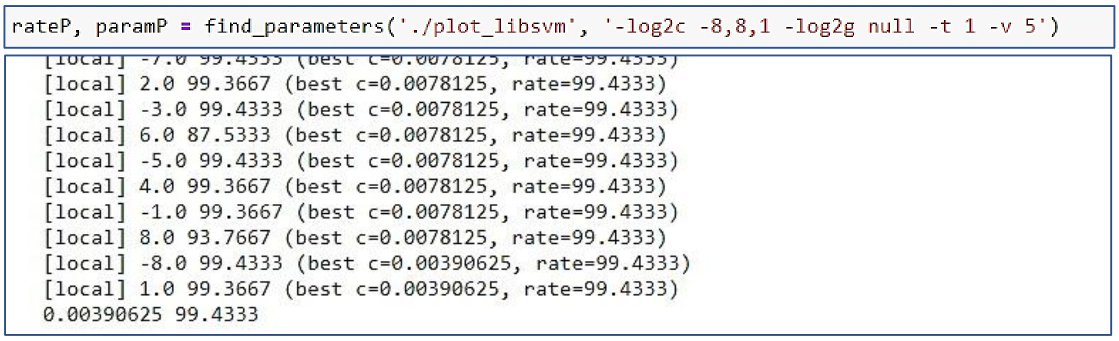
Third step: Find the best parameter

Kernel function – *linear*



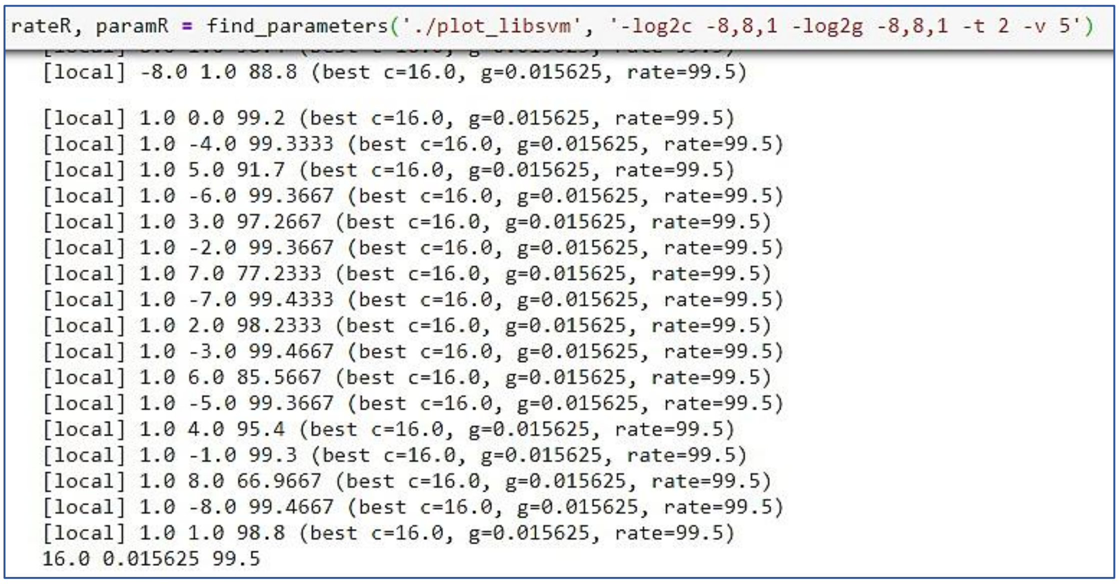
```
rateL, paramL = find_parameters('./plot_libsvm', '-log2c -8,8,1 -log2g null -t 0 -v 5')

    gnuplot executable not found

    [local] 0.0 99.5333 (best c=1.0, rate=99.5333)
    [local] -4.0 99.4333 (best c=1.0, rate=99.5333)
    [local] 5.0 99.4667 (best c=1.0, rate=99.5333)
    [local] -6.0 99.4333 (best c=1.0, rate=99.5333)
    [local] 3.0 99.5 (best c=1.0, rate=99.5333)
    [local] -2.0 99.4667 (best c=1.0, rate=99.5333)
    [local] 7.0 99.4667 (best c=1.0, rate=99.5333)
    [local] -7.0 99.3667 (best c=1.0, rate=99.5333)
    [local] 2.0 99.4667 (best c=1.0, rate=99.5333)
    [local] -3.0 99.4333 (best c=1.0, rate=99.5333)
    [local] 6.0 99.4667 (best c=1.0, rate=99.5333)
    [local] -5.0 99.4 (best c=1.0, rate=99.5333)
    [local] 4.0 99.4667 (best c=1.0, rate=99.5333)
    [local] -1.0 99.4667 (best c=1.0, rate=99.5333)
    [local] 8.0 99.4667 (best c=1.0, rate=99.5333)
    [local] -8.0 99.3667 (best c=1.0, rate=99.5333)
    [local] 1.0 99.4667 (best c=1.0, rate=99.5333)
    1.0 99.5333
```

Kernel function – *polynomial*



```
rateP, paramP = find_parameters('./plot_libsvm', '-log2c -8,8,1 -log2g null -t 1 -v 5')

    [local] -7.0 99.4333 (best c=0.0078125, rate=99.4333)
    [local] 2.0 99.3667 (best c=0.0078125, rate=99.4333)
    [local] -3.0 99.4333 (best c=0.0078125, rate=99.4333)
    [local] 6.0 87.5333 (best c=0.0078125, rate=99.4333)
    [local] -5.0 99.4333 (best c=0.0078125, rate=99.4333)
    [local] 4.0 99.3667 (best c=0.0078125, rate=99.4333)
    [local] -1.0 99.3667 (best c=0.0078125, rate=99.4333)
    [local] 8.0 93.7667 (best c=0.0078125, rate=99.4333)
    [local] -8.0 99.4333 (best c=0.00390625, rate=99.4333)
    [local] 1.0 99.3667 (best c=0.00390625, rate=99.4333)
    0.00390625 99.4333
```

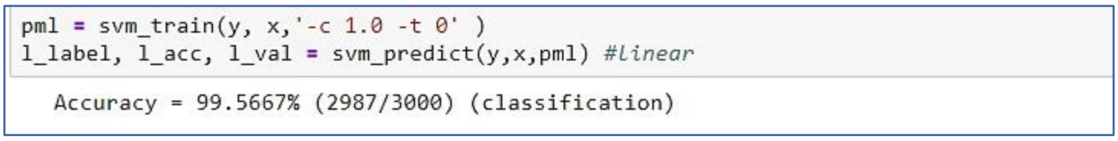
Kernel function – *RBF*



```
rateR, paramR = find_parameters('./plot_libsvm', '-log2c -8,8,1 -log2g -8,8,1 -t 2 -v 5')
    [illegible]
    [local] -8.0 1.0 88.8 (best c=16.0, g=0.015625, rate=99.5)

    [local] 1.0 0.0 99.2 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -4.0 99.3333 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 5.0 91.7 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -6.0 99.3667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 3.0 97.2667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -2.0 99.3667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 7.0 77.2333 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -7.0 99.4333 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 2.0 98.2333 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -3.0 99.4667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 6.0 85.5667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -5.0 99.3667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 4.0 95.4 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -1.0 99.3 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 8.0 66.9667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 -8.0 99.4667 (best c=16.0, g=0.015625, rate=99.5)
    [local] 1.0 1.0 98.8 (best c=16.0, g=0.015625, rate=99.5)
    16.0 0.015625 99.5
```

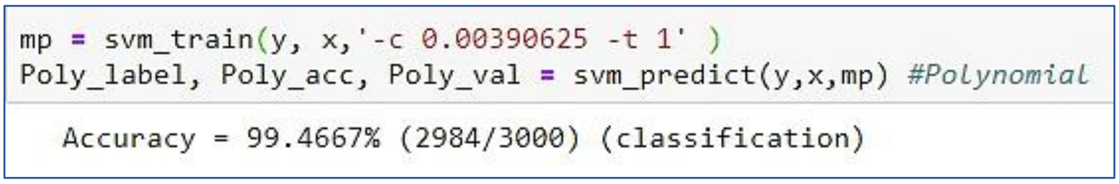
Fourth step: Traning data with diffrernt kernel type

Kernel function – *Linear*



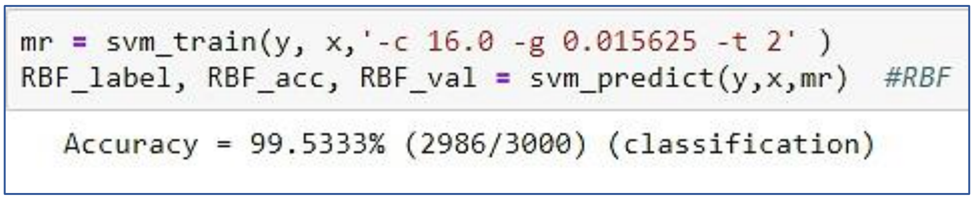
```
pml = svm_train(y, x,'-c 1.0 -t 0' )
l_label, l_acc, l_val = svm_predict(y,x,pml) #Linear

  Accuracy = 99.5667% (2987/3000) (classification)
```

Kernel function – *Polynomial*



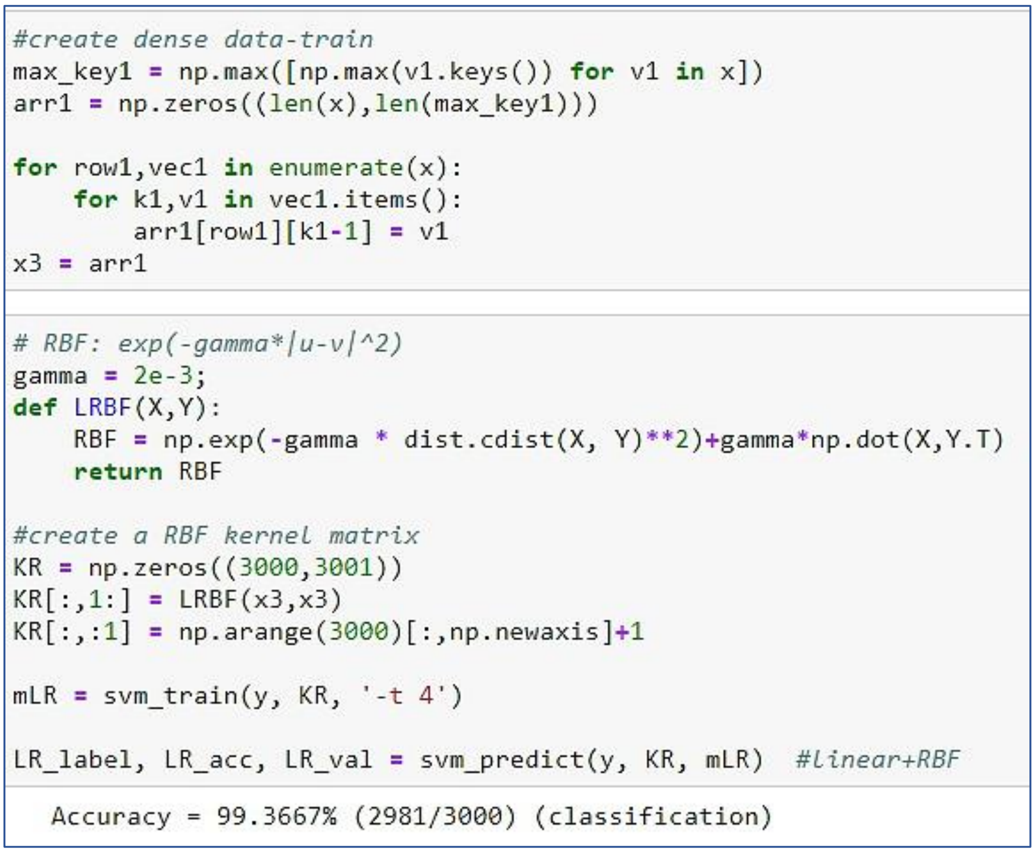
```
mp = svm_train(y, x,'-c 0.00390625 -t 1' )
Poly_label, Poly_acc, Poly_val = svm_predict(y,x,mp) #Polynomial

  Accuracy = 99.4667% (2984/3000) (classification)
```

Kernel function – *RBF*



```
mr = svm_train(y, x,'-c 16.0 -g 0.015625 -t 2' )
RBF_label, RBF_acc, RBF_val = svm_predict(y,x,mr)  #RBF

  Accuracy = 99.5333% (2986/3000) (classification)
```

Kernel function – *Linear +RBF*



```
#create dense data-train
max_key1 = np.max([np.max(v1.keys()) for v1 in x])
arr1 = np.zeros((len(x),len(max_key1)))

for row1,vec1 in enumerate(x):
    for k1,v1 in vec1.items():
        arr1[row1][k1-1] = v1
x3 = arr1

# RBF: exp(-gamma*|u-v|^2)
gamma = 2e-3;
def LRBF(X,Y):
    RBF = np.exp(-gamma * dist.cdist(X, Y)**2)+gamma*np.dot(X,Y.T)
    return RBF

#create a RBF kernel matrix
KR = np.zeros((3000,3001))
KR[:,1:] = LRBF(x3,x3)
KR[:,:1] = np.arange(3000)[:,np.newaxis]+1

mLR = svm_train(y, KR, '-t 4')

LR_label, LR_acc, LR_val = svm_predict(y, KR, mLR)  #Linear+RBF

  Accuracy = 99.3667% (2981/3000) (classification)
```

FIFTH step: Draw the result

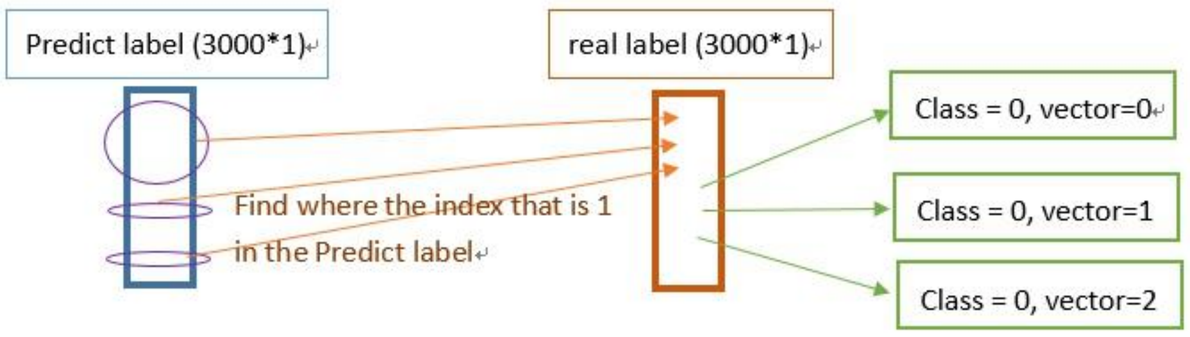
Prepare a title list:

```
title = ['Linear','Polynomial','RBF','Linear+RBF']
```

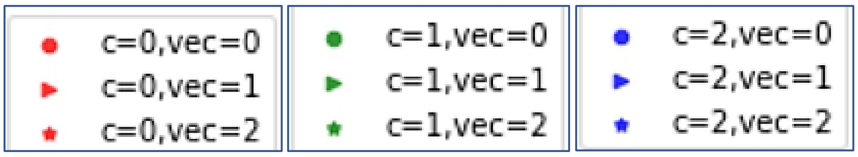
Classify the three categories of forecast results with the real three categories.

```
dot=[]
l_label=np.asarray(l_label)
index0=np.where(l_label == 0.0)
index1=np.where(l_label == 1.0)
index2=np.where(l_label == 2.0)
dot.append(index0[0][np.where(index0[0] < 1000)])
dot.append(index0[0][np.where((index0[0] >=1000)&(index0[0] <2000))])
dot.append(index0[0][np.where(index0[0] >= 2000)])
dot.append(index1[0][np.where(index1[0] < 1000)])
dot.append(index1[0][np.where((index1[0] >=1000)&(index1[0] <2000))])
dot.append(index1[0][np.where(index1[0] >= 2000)])
dot.append(index2[0][np.where(index2[2000:3000] == 0.0)])
dot.append(index2[0][np.where((index2[0] >=1000)&(index2[0] <2000))])
dot.append(index2[0][np.where(index2[0] >= 2000)])

plt.scatter(arr[dot[0],0], arr[dot[0],1], s=20,c='r',marker="o", alpha=0.8, label='c=0,vec=0')
plt.scatter(arr[dot[1],0], arr[dot[1],1], s=20,c='r',marker=">", alpha=0.8, label='c=0,vec=1')
plt.scatter(arr[dot[2],0], arr[dot[2],1], s=20,c='r',marker=(5, 1), alpha=0.8, label='c=0,vec=2')
plt.scatter(arr[dot[3],0], arr[dot[3],1], s=20,c='g',marker="o", alpha=0.8, label='c=1,vec=0')
plt.scatter(arr[dot[4],0], arr[dot[4],1], s=20,c='g',marker=">", alpha=0.8, label='c=1,vec=1')
plt.scatter(arr[dot[5],0], arr[dot[5],1], s=20,c='g',marker=(5, 1), alpha=0.8, label='c=1,vec=2')
plt.scatter(arr[dot[6],0], arr[dot[6],1], s=20,c='b',marker="o", alpha=0.8, label='c=2,vec=0')
plt.scatter(arr[dot[7],0], arr[dot[7],1], s=20,c='b',marker=">", alpha=0.8, label='c=2,vec=1')
plt.scatter(arr[dot[8],0], arr[dot[8],1], s=20,c='b',marker=(5, 1), alpha=0.8, label='c=2,vec=2')

plt.title(title[0])
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()
```

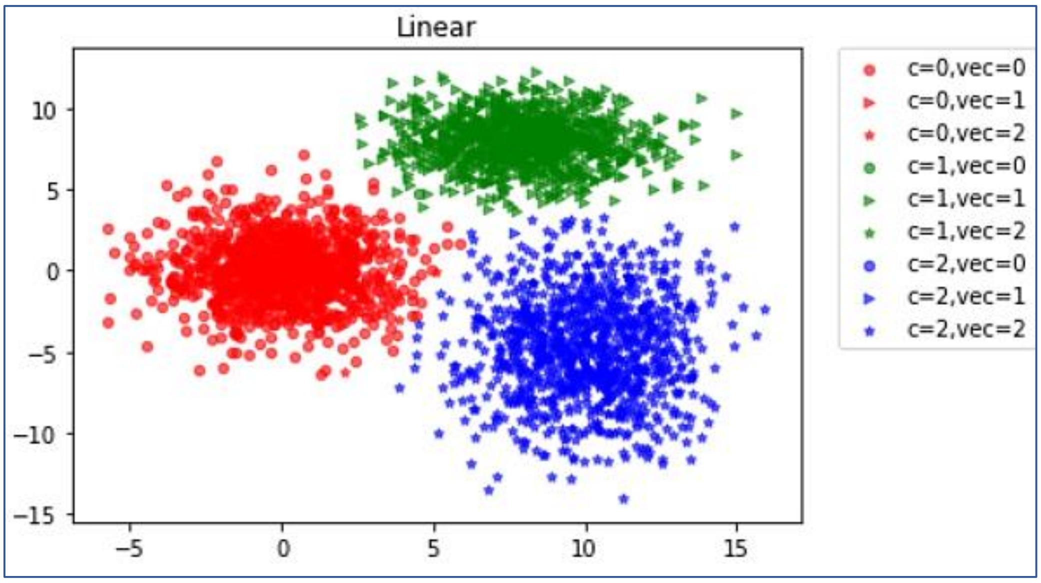
The classification concept is as below:



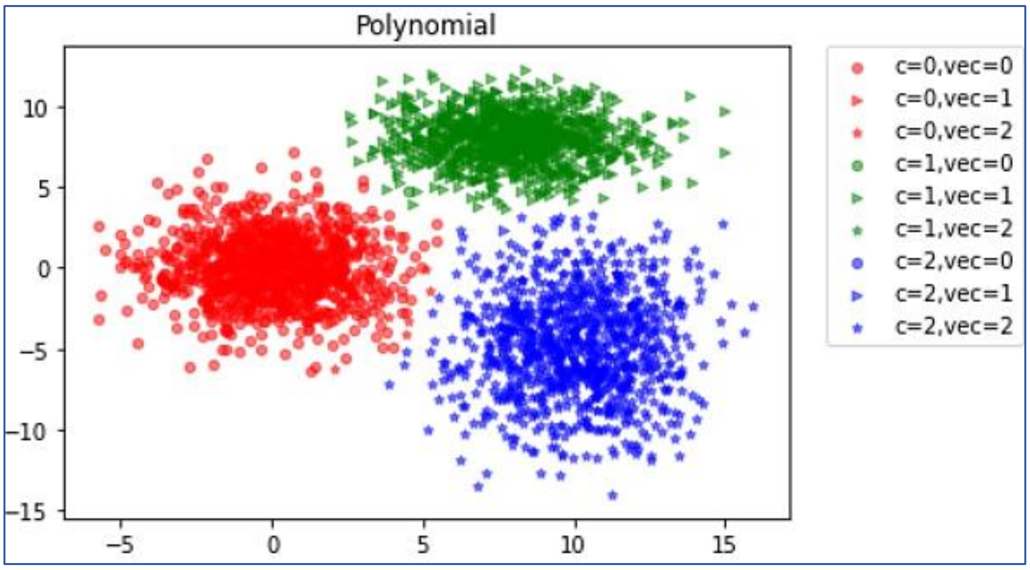
Therefore, our drawing classification should be as shown below:



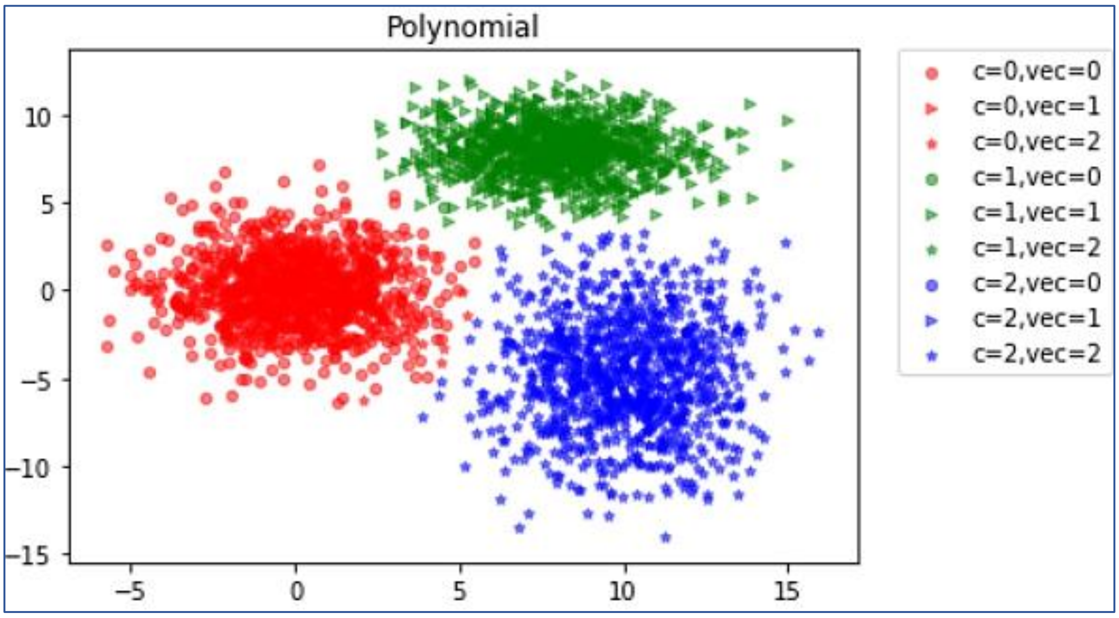
Kernel function – *Linear*



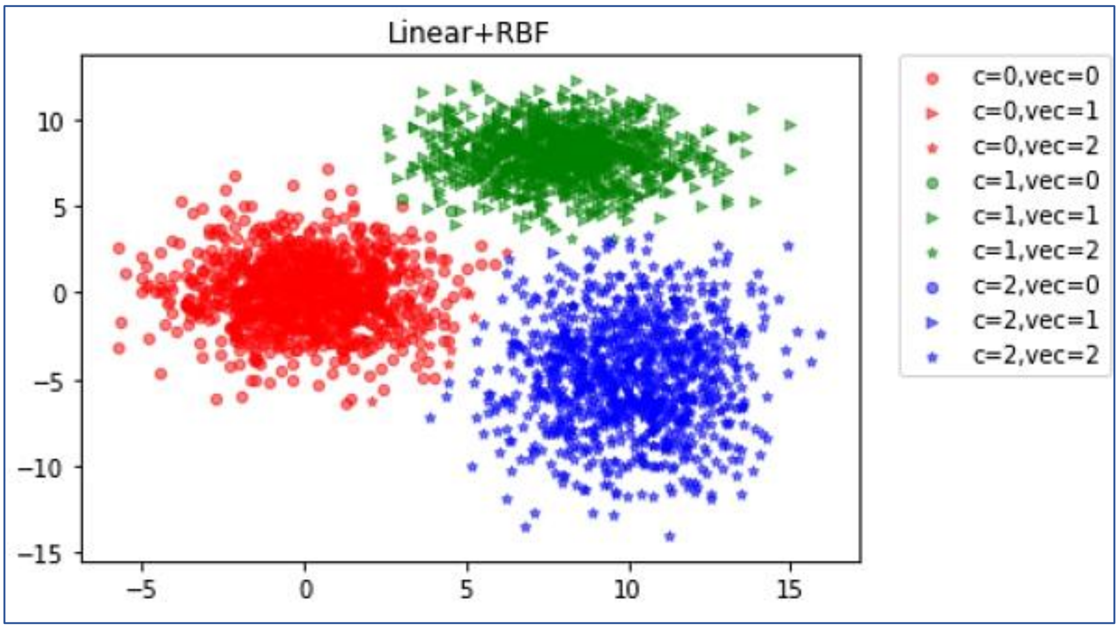
Kernel function – *Polynomial*



Kernel function – *RBF*



Kernel function – *Linear +RBF*



Discussion:

In this case, *"Linear"* kernel type gets the best performance, but other types are not bad either. I guess because the boundaries of the three categories are distinct, so they all have good performance.