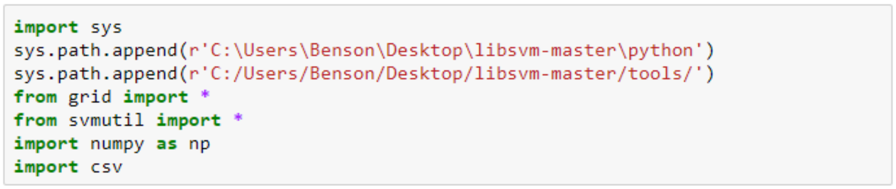
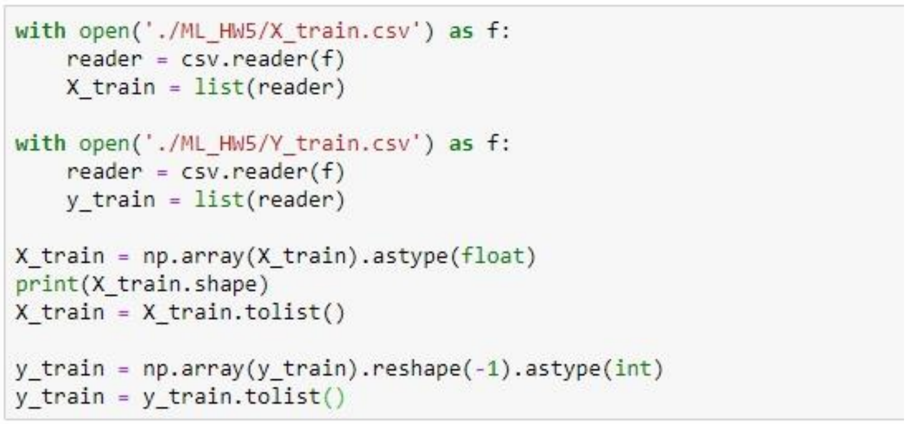
First, we need to let the computer know we have some necessary python file [*grid.py and svmutil.py*] download from libsvm. Thus, we also need to import numpy for some calculate and csv to load the file easily.

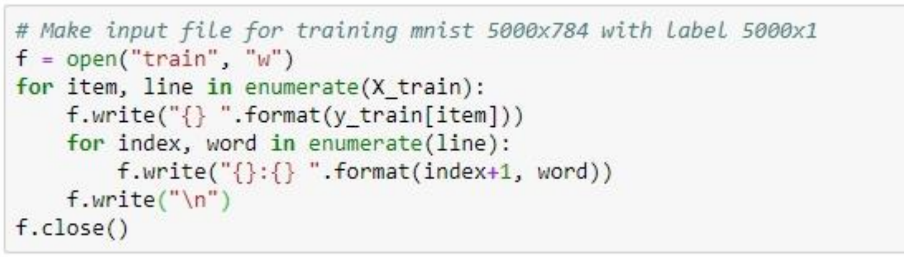


```
import sys
sys.path.append(r'C:\Users\Benson\Desktop\libsvm-master\python')
sys.path.append(r'C:/Users/Benson/Desktop/libsvm-master/tools/')
from grid import *
from svmutil import *
import numpy as np
import csv
```

The code below loading two files [*X_train.csv and Y_train.csv*] which is unzip in our ML_HW5 file. After that, we transfer data into the format svm function can accept and save the training data into specific format. The format looks like this: <label> <index>:<value> <index>:<value>…

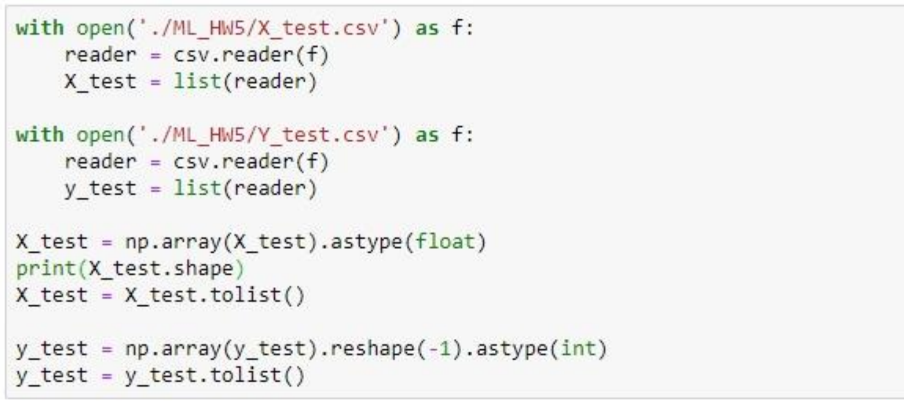


```
with open('./ML_HW5/X_train.csv') as f:
    reader = csv.reader(f)
    X_train = list(reader)

with open('./ML_HW5/Y_train.csv') as f:
    reader = csv.reader(f)
    y_train = list(reader)

X_train = np.array(X_train).astype(float)
print(X_train.shape)
X_train = X_train.tolist()

y_train = np.array(y_train).reshape(-1).astype(int)
y_train = y_train.tolist()
```

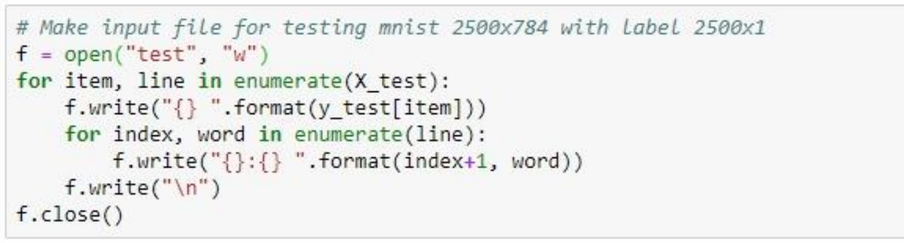


```
# Make input file for training mnist 5000x784 with label 5000x1
f = open("train", "w")
for item, line in enumerate(X_train):
    f.write("{} ".format(y_train[item]))
    for index, word in enumerate(line):
        f.write("{}:{} ".format(index+1, word))
    f.write("\n")
f.close()
```

Do the same things to testing data.

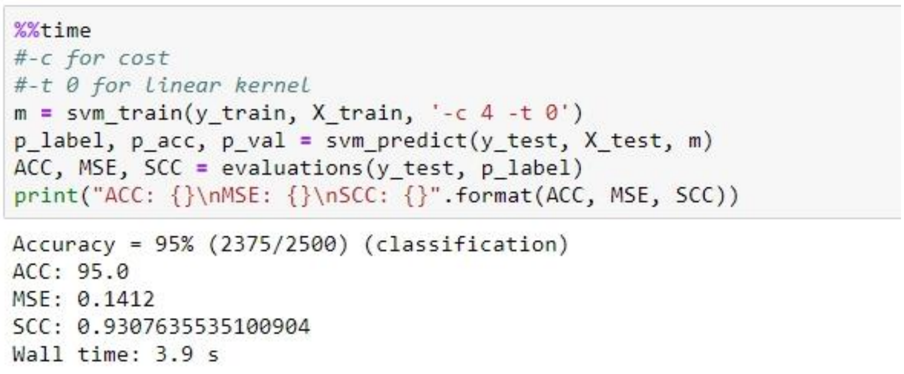


```
with open('./ML_HW5/X_test.csv') as f:
    reader = csv.reader(f)
    X_test = list(reader)

with open('./ML_HW5/Y_test.csv') as f:
    reader = csv.reader(f)
    y_test = list(reader)

X_test = np.array(X_test).astype(float)
print(X_test.shape)
X_test = X_test.tolist()

y_test = np.array(y_test).reshape(-1).astype(int)
y_test = y_test.tolist()
```



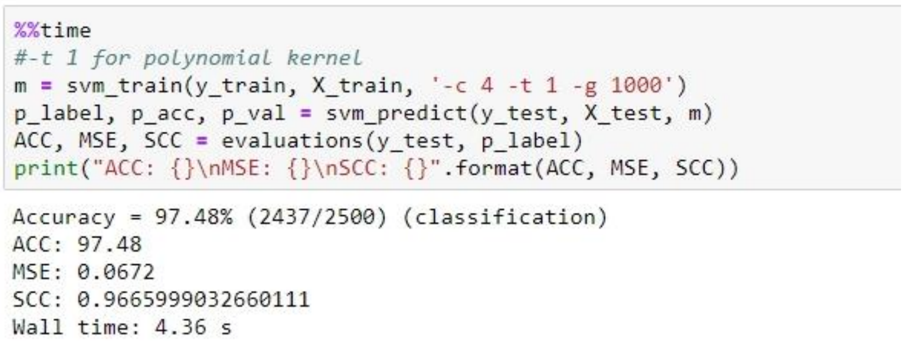
```
# Make input file for testing mnist 2500x784 with label 2500x1
f = open("test", "w")
for item, line in enumerate(X_test):
    f.write("{} ".format(y_test[item]))
    for index, word in enumerate(line):
        f.write("{}:{} ".format(index+1, word))
    f.write("\n")
f.close()
```

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



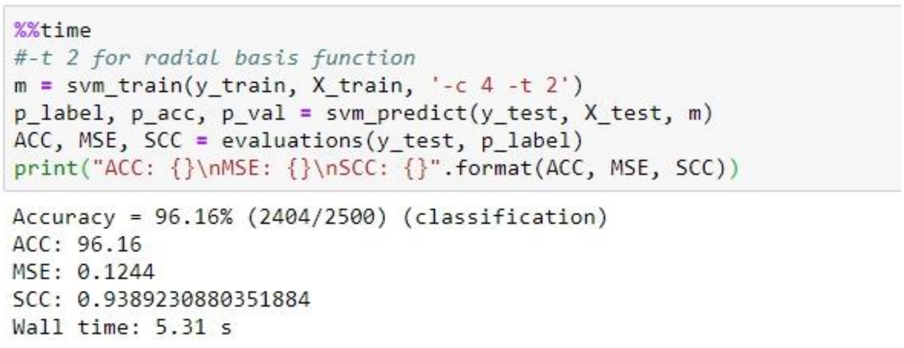
```
%%time
#-c for cost
#-t 0 for linear kernel
m = svm_train(y_train, X_train, '-c 4 -t 0')
p_label, p_acc, p_val = svm_predict(y_test, X_test, m)
ACC, MSE, SCC = evaluations(y_test, p_label)
print("ACC: {}\nMSE: {}\nSCC: {}".format(ACC, MSE, SCC))
```

```
Accuracy = 95% (2375/2500) (classification)
ACC: 95.0
MSE: 0.1412
SCC: 0.9307635535100904
Wall time: 3.9 s
```



```
%%time
#-t 1 for polynomial kernel
m = svm_train(y_train, X_train, '-c 4 -t 1 -g 1000')
p_label, p_acc, p_val = svm_predict(y_test, X_test, m)
ACC, MSE, SCC = evaluations(y_test, p_label)
print("ACC: {}\nMSE: {}\nSCC: {}".format(ACC, MSE, SCC))
```

```
Accuracy = 97.48% (2437/2500) (classification)
ACC: 97.48
MSE: 0.0672
SCC: 0.9665999032660111
Wall time: 4.36 s
```



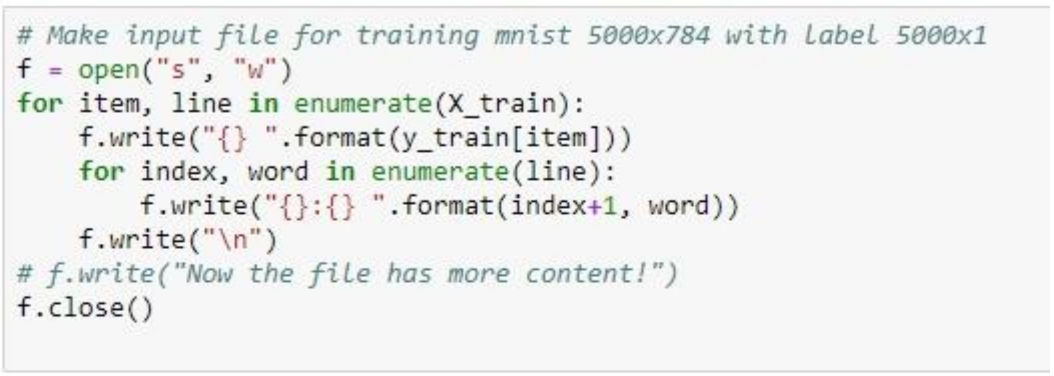
```
%%time
#-t 2 for radial basis function
m = svm_train(y_train, X_train, '-c 4 -t 2')
p_label, p_acc, p_val = svm_predict(y_test, X_test, m)
ACC, MSE, SCC = evaluations(y_test, p_label)
print("ACC: {}\nMSE: {}\nSCC: {}".format(ACC, MSE, SCC))
```

```
Accuracy = 96.16% (2404/2500) (classification)
ACC: 96.16
MSE: 0.1244
SCC: 0.9389230880351884
Wall time: 5.31 s
```

%%time can help calculate the time for each classifier, the parameter *‘-c 4 –t 0’* in svm_train means that, we use cost (C = 4) and use linear kernel (-t 0) to classify. From the libsvm’s github there also other kernels (-t 1 for polynomial, -t 2 for RBF kernel -t 3 for sigmoid kernel) . Additionally, there also one important parameter gamma (-g) for RBF kernel. We use svm_train to generate model (m) and use *svm_predict* to predict out X_test, then we can get p_label which classifier predict. In the end, we use evaluations to test each of kernel’s performance. We found that RBF kernel cost much time than others, and polynomial kernel got better performance. Due to the random cost and gamma we gave, we didn’t get the best parameters for each classifier model.

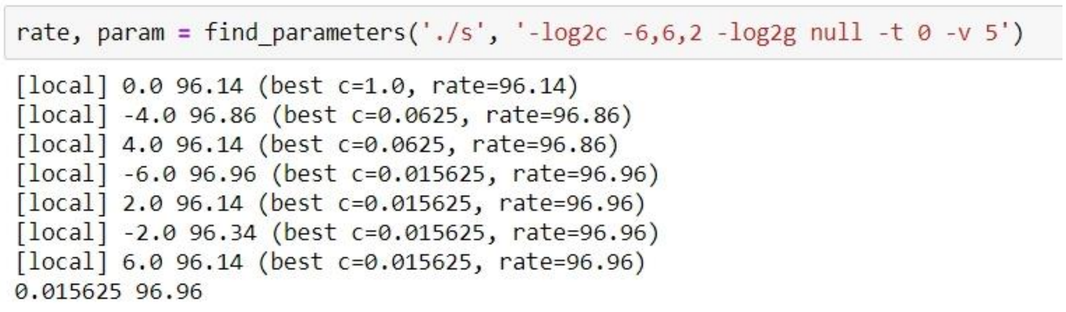
2. Please use C-SVC (you can choose by setting parameters in the function input, C-SVC is soft-margin SVM). Since there are some parameters you need to tune for, please do the grid search for finding parameters of best performing model. For instance, in C-SVC you have a parameter C, and if you use RBF 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 of all we transfer the input data from ML_HW5 (X_train.csv and Y_train.csv) to specific format like this: <label> <index>:<value> <index>:<value>…



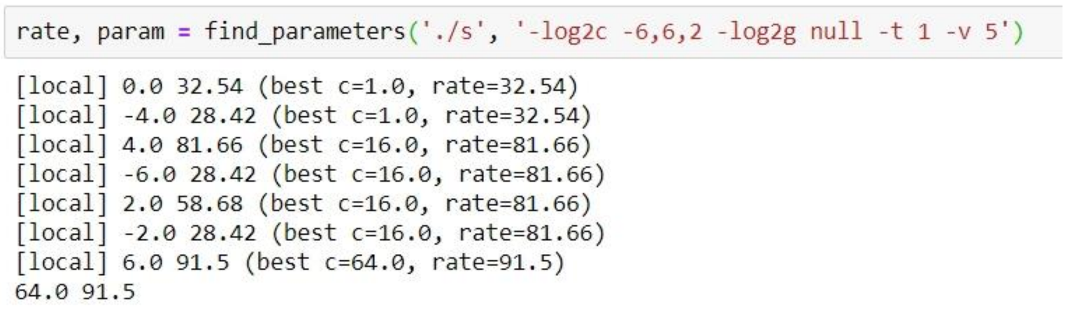
```python
# Make input file for training mnist 5000x784 with label 5000x1
f = open("s", "w")
for item, line in enumerate(X_train):
    f.write("{} ".format(y_train[item]))
    for index, word in enumerate(line):
        f.write("{}:{} ".format(index+1, word))
    f.write("\n")
# f.write("Now the file has more content!")
f.close()
```

After that we can use function *find_parameters* to grid search the best parameters for each model.



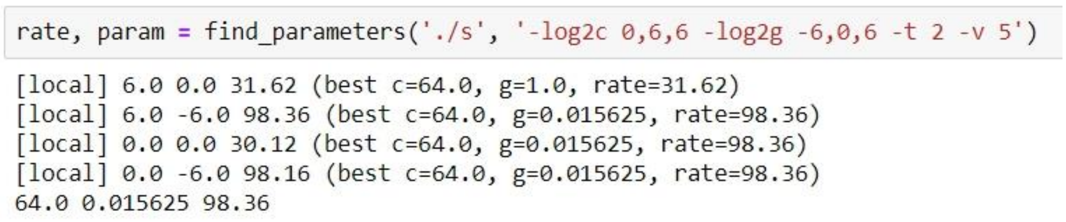
```
rate, param = find_parameters('./s', '-log2c -6,6,2 -log2g null -t 0 -v 5')
```

```
[local] 0.0 96.14 (best c=1.0, rate=96.14)
[local] -4.0 96.86 (best c=0.0625, rate=96.86)
[local] 4.0 96.14 (best c=0.0625, rate=96.86)
[local] -6.0 96.96 (best c=0.015625, rate=96.96)
[local] 2.0 96.14 (best c=0.015625, rate=96.96)
[local] -2.0 96.34 (best c=0.015625, rate=96.96)
[local] 6.0 96.14 (best c=0.015625, rate=96.96)
0.015625 96.96
```



```
rate, param = find_parameters('./s', '-log2c -6,6,2 -log2g null -t 1 -v 5')
```

```
[local] 0.0 32.54 (best c=1.0, rate=32.54)
[local] -4.0 28.42 (best c=1.0, rate=32.54)
[local] 4.0 81.66 (best c=16.0, rate=81.66)
[local] -6.0 28.42 (best c=16.0, rate=81.66)
[local] 2.0 58.68 (best c=16.0, rate=81.66)
[local] -2.0 28.42 (best c=16.0, rate=81.66)
[local] 6.0 91.5 (best c=64.0, rate=91.5)
64.0 91.5
```



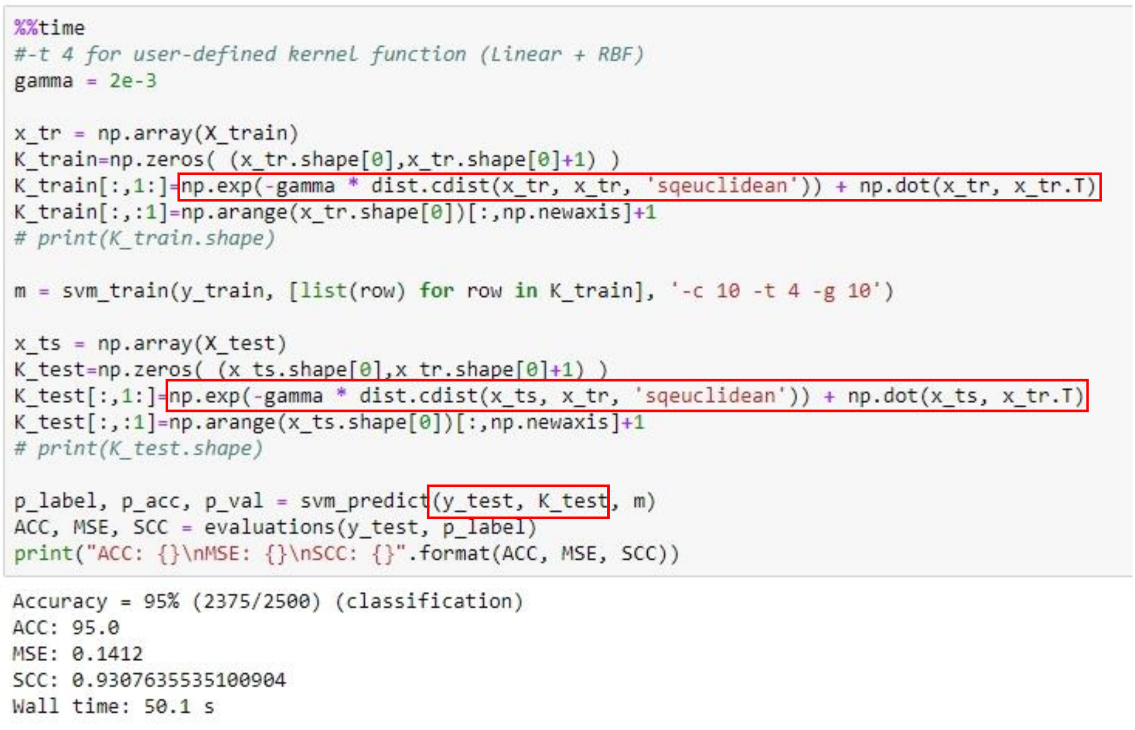
```
rate, param = find_parameters('./s', '-log2c 0,6,6 -log2g -6,0,6 -t 2 -v 5')
```

```
[local] 6.0 0.0 31.62 (best c=64.0, g=1.0, rate=31.62)
[local] 6.0 -6.0 98.36 (best c=64.0, g=0.015625, rate=98.36)
[local] 0.0 0.0 30.12 (best c=64.0, g=0.015625, rate=98.36)
[local] 0.0 -6.0 98.16 (best c=64.0, g=0.015625, rate=98.36)
64.0 0.015625 98.36
```

In this part, we try to find out which parameters may let the classifier more powerful. In the grid.py we can use the function *find_parameters* to find best parameters by grid search with 5-cross validation (-v 5). The first argument in *find_parameters* is the filename, the second arguments *'-log2c 0,6,6 –log2g -6,0,6 –t 2 –v 5'* can be separated into [grid options] + [svm_options], It means that we calculate C in range 2^{0, 6} and g in range 2^{-6, 0}. After calling find_parameters it will return the rate (aka accuracy) and param (aka best fit parameters), as we can see , RBF kernel looks well in the given range with c and gamma. Cause C-SVC is the default argument (-s 0), we don’t need to put it on.The definition about find_parameters is followed below.

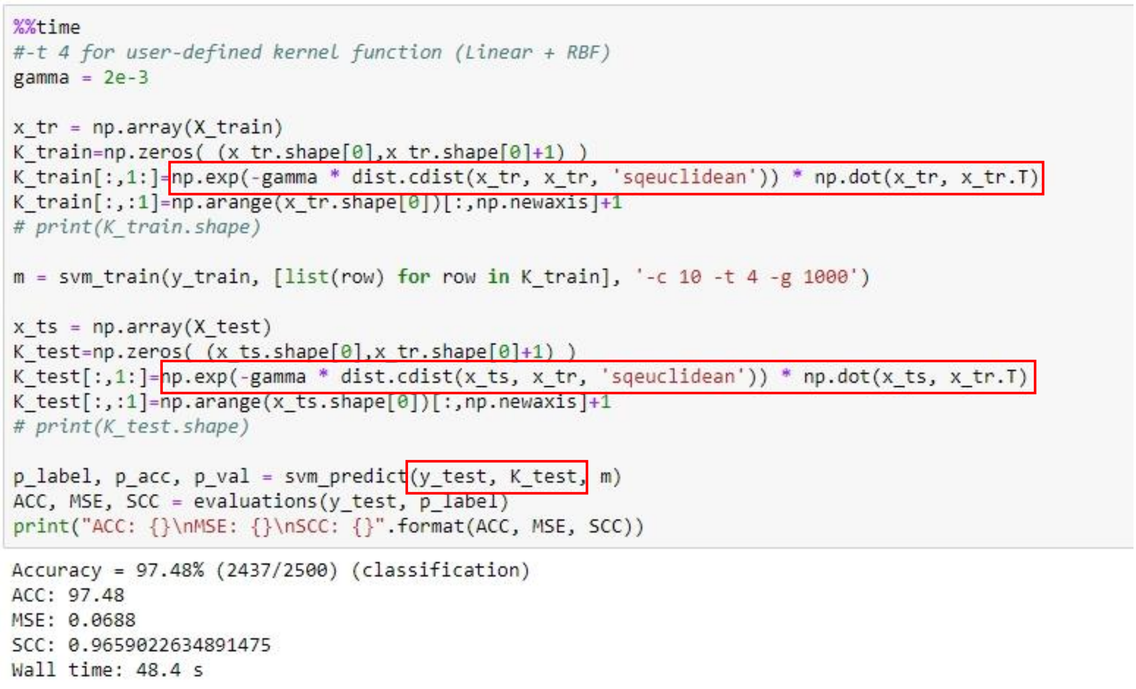
```
Grid_options :
-log2c {begin,end,step | "null"} : set the range of c (default -5,15,2)
    begin,end,step -- c_range = 2^{begin,...,begin+k*step,...,end}
    "null"        -- do not grid with c
-log2g {begin,end,step | "null"} : set the range of g (default 3,-15,-2)
    begin,end,step -- g_range = 2^{begin,...,begin+k*step,...,end}
    "null"        -- do not grid with g
-v n : n-fold cross validation (default 5)

 svm options:
-s svm_type : set type of SVM (default 0)
    0 -- C-SVC                  (multi-class classification)
    1 -- nu-SVC                 (multi-class classification)
    2 -- one-class SVM
    3 -- epsilon-SVR   (regression)
    4 -- nu-SVR                 (regression)
-t kernel_type : set type of kernel function (default 2)
    0 -- linear: u'*v
    1 -- polynomial: (gamma*u'*v + coef0)^degree
    2 -- radial basis function: exp(-gamma*|u-v|^2)
    3 -- sigmoid: tanh(gamma*u'*v + coef0)
    4 -- precomputed kernel (kernel values in training_set_file)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
```

3. 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.



```
%%time
#-t 4 for user-defined kernel function (Linear + RBF)
gamma = 2e-3

x_tr = np.array(X_train)
K_train=np.zeros( (x_tr.shape[0],x_tr.shape[0]+1) )
K_train[:,1:]=np.exp(-gamma * dist.cdist(x_tr, x_tr, 'sqeuclidean')) + np.dot(x_tr, x_tr.T)
K_train[:,:1]=np.arange(x_tr.shape[0])[:,np.newaxis]+1
# print(K_train.shape)

m = svm_train(y_train, [list(row) for row in K_train], '-c 10 -t 4 -g 10')

x_ts = np.array(X_test)
K_test=np.zeros( (x_ts.shape[0],x_tr.shape[0]+1) )
K_test[:,1:]=np.exp(-gamma * dist.cdist(x_ts, x_tr, 'sqeuclidean')) + np.dot(x_ts, x_tr.T)
K_test[:,:1]=np.arange(x_ts.shape[0])[:,np.newaxis]+1
# print(K_test.shape)

p_label, p_acc, p_val = svm_predict(y_test, K_test, m)
ACC, MSE, SCC = evaluations(y_test, p_label)
print("ACC: {}\nMSE: {}\nSCC: {}".format(ACC, MSE, SCC))
```

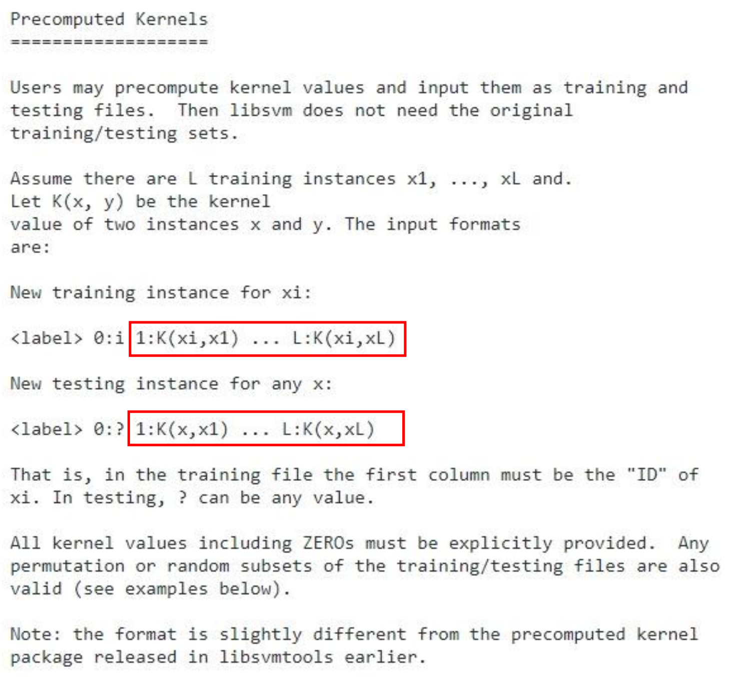
```
Accuracy = 95% (2375/2500) (classification)
ACC: 95.0
MSE: 0.1412
SCC: 0.9307635535100904
Wall time: 50.1 s
```



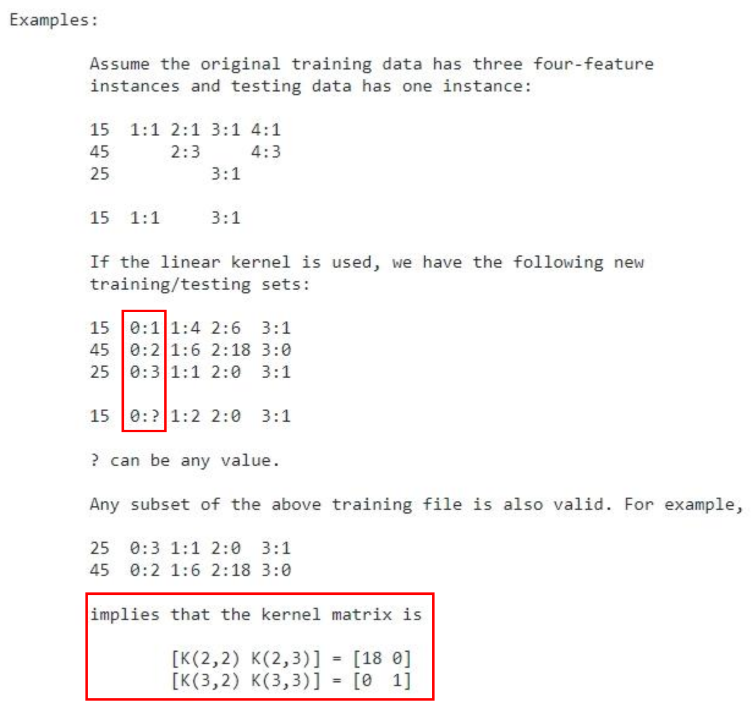
```
%%time
#-t 4 for user-defined kernel function (Linear + RBF)
gamma = 2e-3

x_tr = np.array(X_train)
K_train=np.zeros( (x_tr.shape[0],x_tr.shape[0]+1) )
K_train[:,1:]=np.exp(-gamma * dist.cdist(x_tr, x_tr, 'sqeuclidean')) * np.dot(x_tr, x_tr.T)
K_train[:,:1]=np.arange(x_tr.shape[0])[:,np.newaxis]+1
# print(K_train.shape)

m = svm_train(y_train, [list(row) for row in K_train], '-c 10 -t 4 -g 1000')

x_ts = np.array(X_test)
K_test=np.zeros( (x_ts.shape[0],x_tr.shape[0]+1) )
K_test[:,1:]=np.exp(-gamma * dist.cdist(x_ts, x_tr, 'sqeuclidean')) * np.dot(x_ts, x_tr.T)
K_test[:,:1]=np.arange(x_ts.shape[0])[:,np.newaxis]+1
# print(K_test.shape)

p_label, p_acc, p_val = svm_predict(y_test, K_test, m)
ACC, MSE, SCC = evaluations(y_test, p_label)
print("ACC: {}\nMSE: {}\nSCC: {}".format(ACC, MSE, SCC))
```

```
Accuracy = 97.48% (2437/2500) (classification)
ACC: 97.48
MSE: 0.0688
SCC: 0.9659022634891475
Wall time: 48.4 s
```

According to the description on libsvm's github, and stackoverflow, [https://stackoverflow.com/questions/10978261/libsvm-precomputed-kernels], I follow the step and create the linear + RBF kernel with the code [np.exp(-gamma * dist.cdist(x_tr, x_tr, 'sqeuclidean')) + np.dot(x_tr,x_tr.T)]. Red color represent RBF kernel function: *exp(-gamma*|u-v|^2)*, Green color represent linear kernel function: *u'*v*. Because we use user-defined kernel (-t 4), the format should add column for index 0, so that we can use svm_train function. The describe below for the detail:



```
Precomputed Kernels
===================

Users may precompute kernel values and input them as training and
testing files.  Then libsvm does not need the original
training/testing sets.

Assume there are L training instances x1, ..., xL and.
Let K(x, y) be the kernel
value of two instances x and y. The input formats
are:

New training instance for xi:

<label> 0:i 1:K(xi,x1) ... L:K(xi,xL)

New testing instance for any x:

<label> 0:? 1:K(x,x1) ... L:K(x,xL)

That is, in the training file the first column must be the "ID" of
xi. In testing, ? can be any value.

All kernel values including ZEROs must be explicitly provided.  Any
permutation or random subsets of the training/testing files are also
valid (see examples below).

Note: the format is slightly different from the precomputed kernel
package released in libsvmtools earlier.
```



```
Examples:

        Assume the original training data has three four-feature
        instances and testing data has one instance:

        15  1:1 2:1 3:1 4:1
        45      2:3     4:3
        25          3:1

        15  1:1     3:1

        If the linear kernel is used, we have the following new
        training/testing sets:

        15  0:1 1:4 2:6  3:1
        45  0:2 1:6 2:18 3:0
        25  0:3 1:1 2:0  3:1

        15  0:? 1:2 2:0  3:1

        ? can be any value.

        Any subset of the above training file is also valid. For example,

        25  0:3 1:1 2:0  3:1
        45  0:2 1:6 2:18 3:0

        implies that the kernel matrix is

                [K(2,2) K(2,3)] = [18 0]
                [K(3,2) K(3,3)] = [0  1]
```

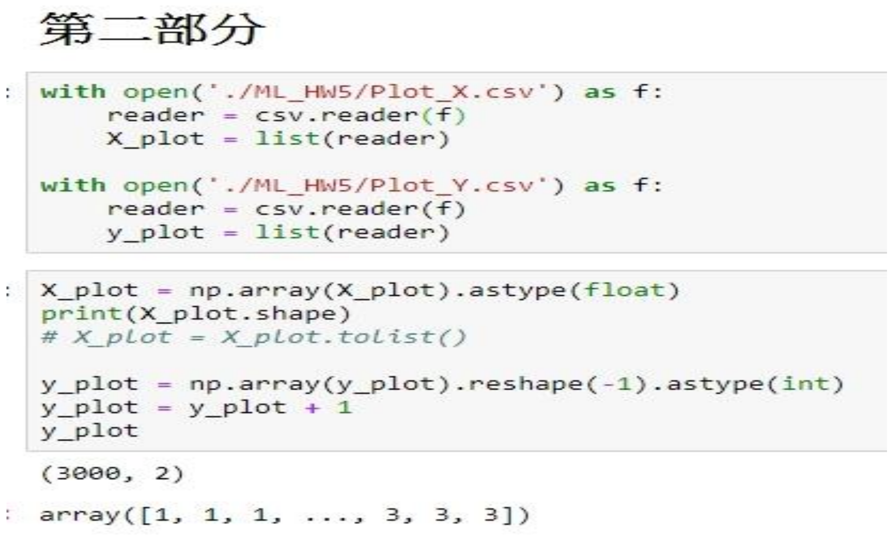
dist.cdist(x_tr, x_tr, 'sqeuclidean') get the euclidean distance without taking root => |x-v|^2

Compare with other kernel like linear , poly or RBF , my kernel function got well performance. I also try my kernel function (Linear + RBF) with two ways. One is Linear kernel + RBF kernel, the other is Linear * RBF kernel, and I found that Linear * RBF kernel get better performance (Accuracy: 97.48%), Linear + RBF kernel get (Accuracy: 95%)

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

   Detail of the visualization:

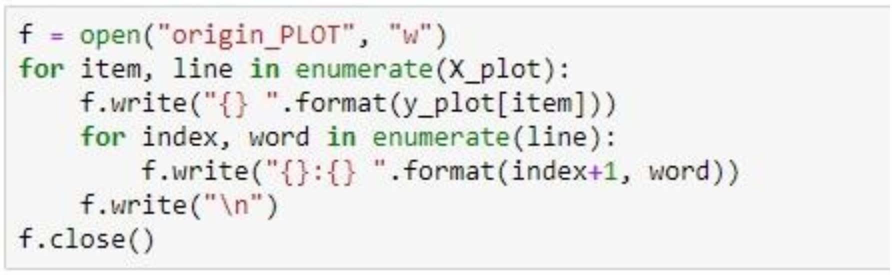
   - Use different colors to show different clusters.
   - All the data samples are shown by "dots", the "support vectors" that you obtained from your model should be shown with different symbols, e.g. square, triangle, cross.
   - 4 figures in total (linear, polynomial, RBF, linear+RBF)



第二部分

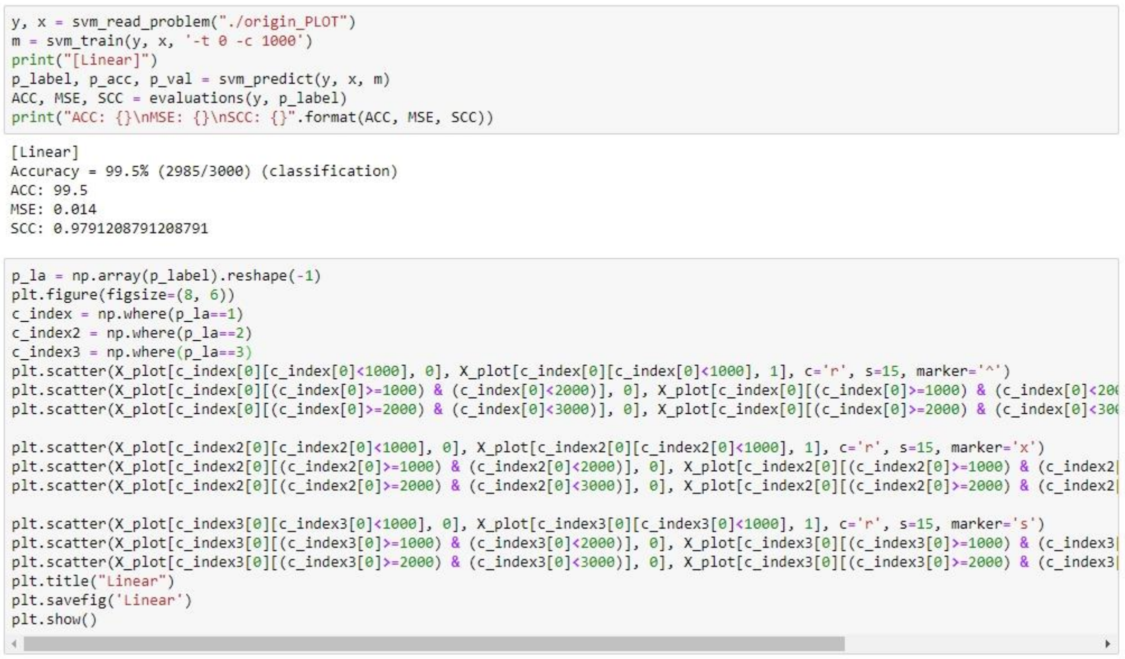
```
:  with open('./ML_HW5/Plot_X.csv') as f:
       reader = csv.reader(f)
       X_plot = list(reader)

   with open('./ML_HW5/Plot_Y.csv') as f:
       reader = csv.reader(f)
       y_plot = list(reader)
```

```
:  X_plot = np.array(X_plot).astype(float)
   print(X_plot.shape)
   # X_plot = X_plot.tolist()

   y_plot = np.array(y_plot).reshape(-1).astype(int)
   y_plot = y_plot + 1
   y_plot

   (3000, 2)

:  array([1, 1, 1, ..., 3, 3, 3])
```



```
f = open("origin_PLOT", "w")
for item, line in enumerate(X_plot):
    f.write("{} ".format(y_plot[item]))
    for index, word in enumerate(line):
        f.write("{}:{} ".format(index+1, word))
    f.write("\n")
f.close()
```

For each of the kernel, I follow the same code as the picture above. First, read the "origin_PLOT", which is the file combine with Plot_X.csv and Plot_Y.csv. Second, draw the picture with matplot.

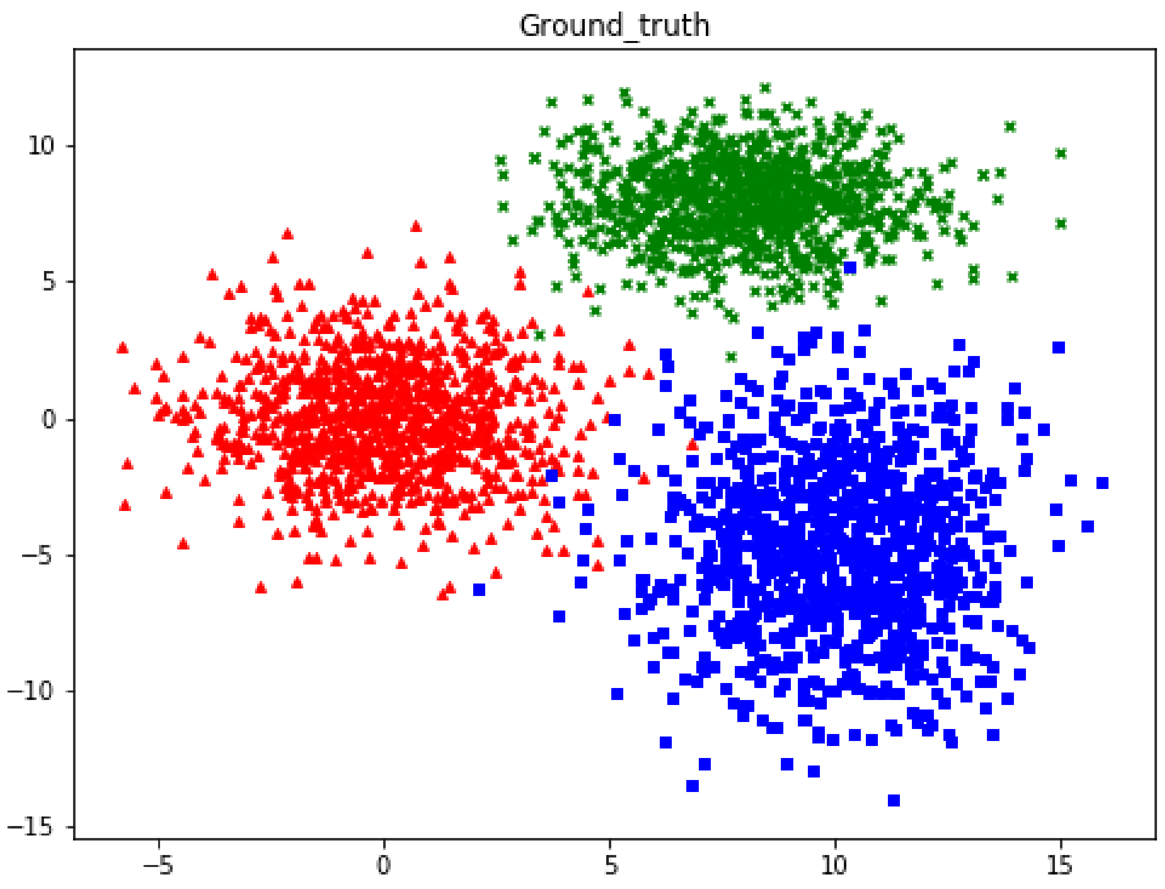


```
y, x = svm_read_problem("./origin_PLOT")
m = svm_train(y, x, '-t 0 -c 1000')
print("[Linear]")
p_label, p_acc, p_val = svm_predict(y, x, m)
ACC, MSE, SCC = evaluations(y, p_label)
print("ACC: {}\nMSE: {}\nSCC: {}".format(ACC, MSE, SCC))
```

```
[Linear]
Accuracy = 99.5% (2985/3000) (classification)
ACC: 99.5
MSE: 0.014
SCC: 0.9791208791208791
```

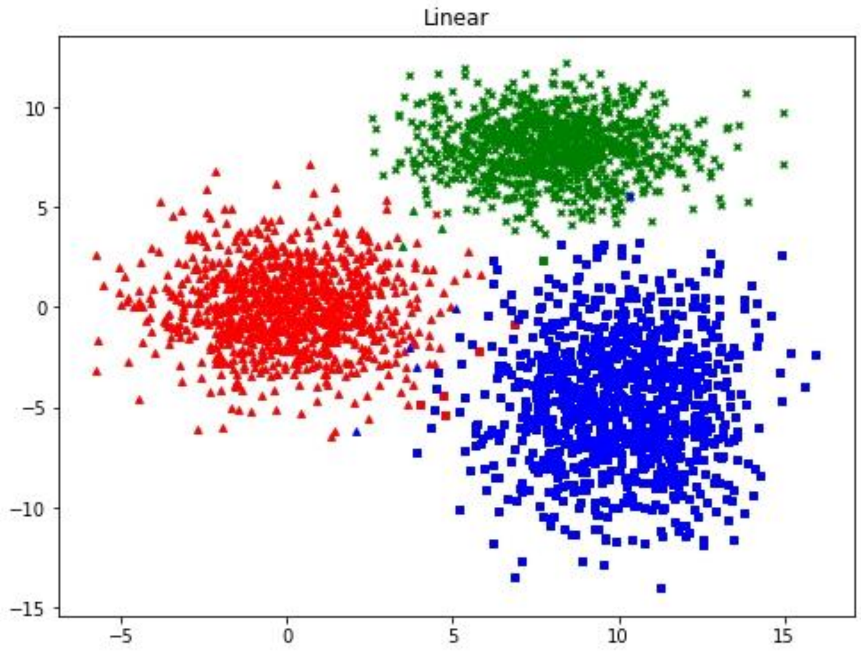
```
p_la = np.array(p_label).reshape(-1)
plt.figure(figsize=(8, 6))
c_index = np.where(p_la==1)
c_index2 = np.where(p_la==2)
c_index3 = np.where(p_la==3)
plt.scatter(X_plot[c_index[0][c_index[0]<1000], 0], X_plot[c_index[0][c_index[0]<1000], 1], c='r', s=15, marker='^')
plt.scatter(X_plot[c_index[0][(c_index[0]>=1000) & (c_index[0]<2000)], 0], X_plot[c_index[0][(c_index[0]>=1000) & (c_index[0]<20
plt.scatter(X_plot[c_index[0][(c_index[0]>=2000) & (c_index[0]<3000)], 0], X_plot[c_index[0][(c_index[0]>=2000) & (c_index[0]<30

plt.scatter(X_plot[c_index2[0][c_index2[0]<1000], 0], X_plot[c_index2[0][c_index2[0]<1000], 1], c='r', s=15, marker='x')
plt.scatter(X_plot[c_index2[0][(c_index2[0]>=1000) & (c_index2[0]<2000)], 0], X_plot[c_index2[0][(c_index2[0]>=1000) & (c_index2
plt.scatter(X_plot[c_index2[0][(c_index2[0]>=2000) & (c_index2[0]<3000)], 0], X_plot[c_index2[0][(c_index2[0]>=2000) & (c_index2

plt.scatter(X_plot[c_index3[0][c_index3[0]<1000], 0], X_plot[c_index3[0][c_index3[0]<1000], 1], c='r', s=15, marker='s')
plt.scatter(X_plot[c_index3[0][(c_index3[0]>=1000) & (c_index3[0]<2000)], 0], X_plot[c_index3[0][(c_index3[0]>=1000) & (c_index3
plt.scatter(X_plot[c_index3[0][(c_index3[0]>=2000) & (c_index3[0]<3000)], 0], X_plot[c_index3[0][(c_index3[0]>=2000) & (c_index3
plt.title("Linear")
plt.savefig('Linear')
plt.show()
```

From the Plot_Y.csv, we know three clusters followed by (1-1000, 1001-2000, 2001-3000), and we have p_label which is the predict label from the *svm_predict* function. According to the p_label , I separate it into three index_group, each index_group belongs to one cluster the classifier predict, I use trangle marker to show that it belongs to cluster 1, cross marker to show that it belongs to cluster 2, square marker to show that it belongs to cluster 3. All the prediction will be shown below, and you can see that there are some mistake happened in the boundary between each cluster.

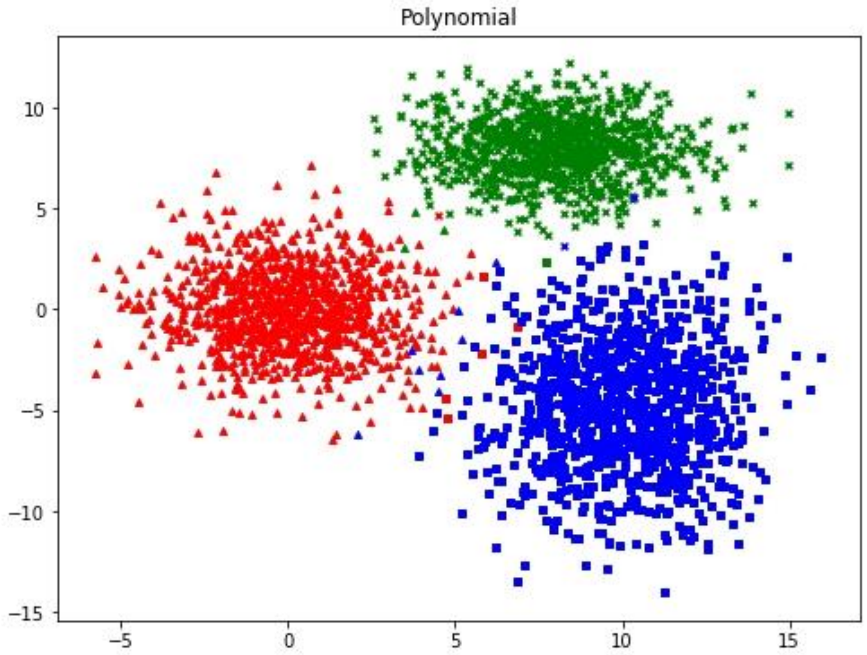


The picture above is the original data, and I use three for 3 clustering (Red, Green, Blue),after that I try to use four different ways to predict each data to the clustering. The pictures below are the result of different kernel and its performance.

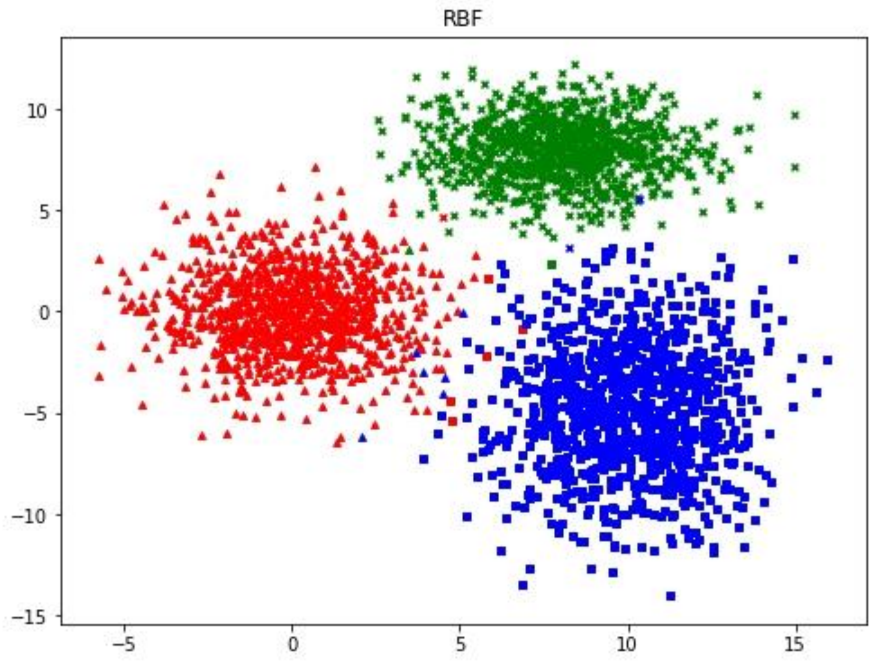
```
[Linear]
Accuracy = 99.5% (2985/3000) (classification)
ACC: 99.5
MSE: 0.014
SCC: 0.9791208791208791
```



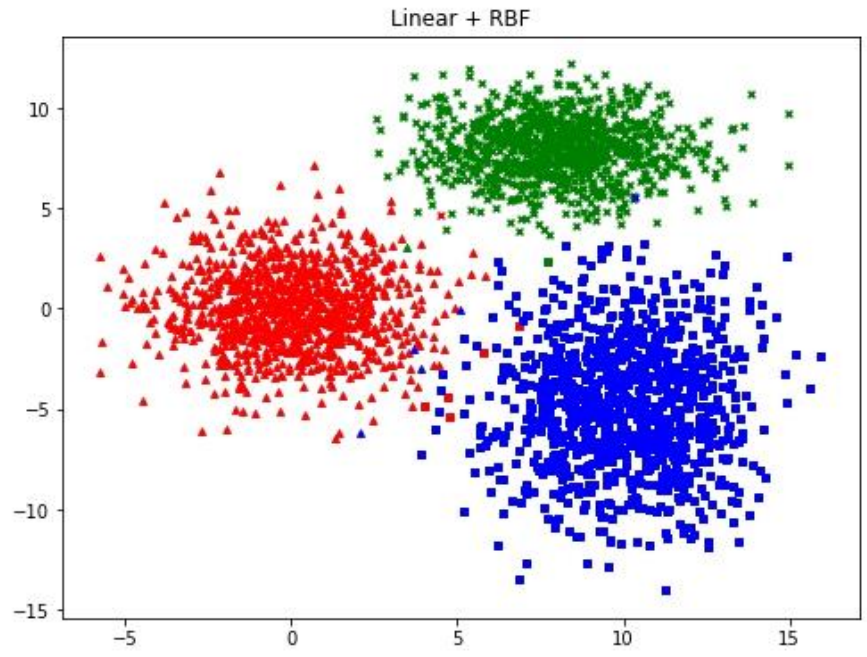
```
[Poly]
Accuracy = 99.3333% (2980/3000) (classification)
ACC: 99.33333333333333
MSE: 0.019666666666666666
SCC: 0.9707379859698256
```



```
[RBF]
Accuracy = 99.4667% (2984/3000) (classification)
ACC: 99.46666666666667
MSE: 0.01633333333333333
SCC: 0.9756455362364226
```



```
[Linear + RBF]
Accuracy = 99.5667% (2987/3000) (classification)
ACC: 99.56666666666666
MSE: 0.013333333333333334
SCC: 0.9801006534004356
```


Linear + RBF