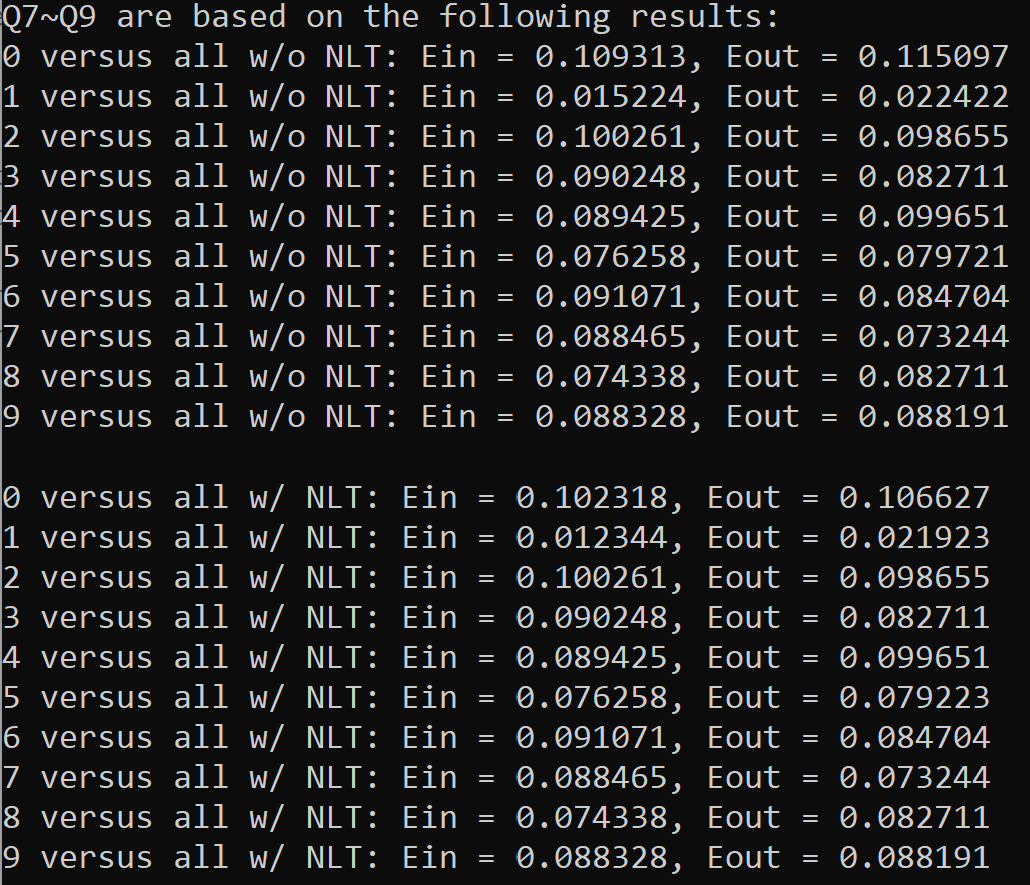
7. [d]

8. [b]

9. [e]



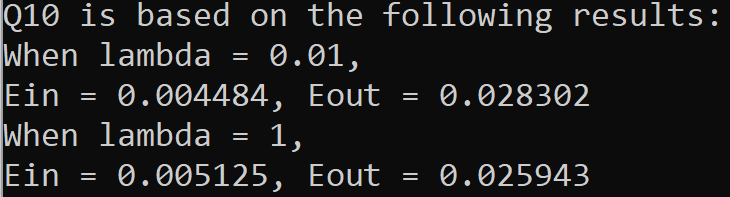
For ‘1 versus all’ with and without nonlinear transform, Eout is small (no overfitting observed, 9[a] wrong);

For ‘9 versus all’ with and without nonlinear transform, Eout is of the same, 9[b] wrong;

For ‘0 versus all’ before and after nonlinear transform, Eout becomes smaller, 9[c][d] wrong;

For ‘5 versus all’, Eout w/o NLT =0.079721, Eout w/ NLT = 0.079223, improvement = 0.6%, 9[e] correct.

10. [a]



Q7~Q10 are based on the following codes:

import numpy as np

class RegularizedLinearRegression:

def \_\_init\_\_(self, dim, lamda):

self.weights = np.zeros((dim + 1,1))

self.lamda = lamda

def X\_new(self,X):

examples = X.shape[0]

X\_new = np.c\_[np.ones(examples), X]

return X\_new

def train\_regularization(self, X,Y): # training with regularization: (ZT\*Z + lambda\*I)^-1 \* ZT\*y

X\_new = self.X\_new(X)

xTx = np.dot(X\_new.T, X\_new)

lI = np.multiply(self.lamda, np.identity(xTx.shape[0])) # lambda\*I

inv\_X = np.linalg.inv(np.add(xTx, lI))

self.weights = np.dot(inv\_X, np.dot(X\_new.T, Y))

def predict(self,X):

X\_new = self.X\_new(X)

h = np.matmul(X\_new, self.weights)

return h

def calc\_error(self, X,Y):

examples = X.shape[0]

predict = np.sign(self.predict(X))

num\_error = np.sum(np.not\_equal(predict, np.sign(Y)))

error = float(num\_error)/float(examples)

return error

class NonlinearTrans(RegularizedLinearRegression):

def \_\_init\_\_(self, dim, lamda):

self.dim = (2\*dim + 1) # 1, x1, x2, x1^2, x2^2, x1\*x2

self.weights = np.zeros((self.dim + 1, 1))

self.lamda = lamda

def change\_lambda(self, lamda):

self.lamda = lamda

def X\_new(self,X):

examples = X.shape[0]

X\_multiply = np.prod(X, axis=1) # x1\*x2

X\_subtract = np.c\_[X[:,0],-X[:,1]] # x1 -x2

X\_new = np.c\_[np.ones(examples), X, X\_multiply, np.square(X)]

return X\_new

def set\_labels\_all(x, Y): # x = desired digit in 'x versus all'

Y\_labeled = np.array([])

for i in np.arange(Y.shape[0]):

if Y[i] == x:

Y\_labeled = np.append(Y\_labeled, 1)

else:

Y\_labeled = np.append(Y\_labeled, -1)

return Y\_labeled

def set\_labels\_one(x1, x2, X, Y): # x1, x2 = desired digits in 'x1 versus x2'

X\_labeled = np.array([])

Y\_labeled = np.array([])

for i in np.arange(Y.shape[0]):

if Y[i] == x1:

X\_labeled = np.concatenate((X\_labeled, X[i,:]))

Y\_labeled = np.append(Y\_labeled, 1)

if Y[i] == x2:

X\_labeled = np.concatenate((X\_labeled, X[i,:]))

Y\_labeled = np.append(Y\_labeled, -1)

X\_labeled = X\_labeled.reshape((Y\_labeled.shape[0], X.shape[1]))

return X\_labeled, Y\_labeled

def load\_data(filename\_train, filename\_test):

data\_train = np.loadtxt(filename\_train)

X\_train = data\_train[:, 1:] # delete 1st digit term (Y)

Y\_train = data\_train[:, :1] # extract 1st digit term (Y)

data\_test = np.loadtxt(filename\_test)

X\_test = data\_test[:, 1:] # delete 1st digit term (Y)

Y\_test = data\_test[:, :1] # extract 1st digit term (Y)

# data = np.r\_[data\_train, data\_test]

return X\_train, Y\_train, X\_test, Y\_test

def main():

X\_train, Y\_train, X\_test, Y\_test = load\_data('features.train','features.test')

print("Q7~Q9 are based on the following results:")

RegLR = RegularizedLinearRegression(X\_train.shape[1], 1) # lambda = 1

for x in range(10):

Y\_train\_all = set\_labels\_all(x, Y\_train)

Y\_test\_all = set\_labels\_all(x, Y\_test)

RegLR.train\_regularization(X\_train, Y\_train\_all)

print("%d versus all w/o NLT: Ein = %f, Eout = %f" % (x, RegLR.calc\_error(X\_train, Y\_train\_all), RegLR.calc\_error(X\_test, Y\_test\_all)))

print("")

NLT = NonlinearTrans(X\_train.shape[1], 1) # lambda = 1

for x in range(10):

Y\_train\_all = set\_labels\_all(x, Y\_train)

Y\_test\_all = set\_labels\_all(x, Y\_test)

NLT.train\_regularization(X\_train, Y\_train\_all)

print("%d versus all w/ NLT: Ein = %f, Eout = %f" % (x, NLT.calc\_error(X\_train, Y\_train\_all), NLT.calc\_error(X\_test, Y\_test\_all)))

print("")

print("Q10 is based on the following results:")

X\_train\_one, Y\_train\_one = set\_labels\_one(1, 5, X\_train, Y\_train)

X\_test\_one, Y\_test\_one = set\_labels\_one(1, 5, X\_test, Y\_test)

print("When lambda = 0.01,")

NLT.change\_lambda(0.01) # lambda = 0.01

NLT.train\_regularization(X\_train\_one, Y\_train\_one)

print("Ein = %f, Eout = %f" % (NLT.calc\_error(X\_train\_one, Y\_train\_one), NLT.calc\_error(X\_test\_one, Y\_test\_one)))

print("When lambda = 1,")

NLT.change\_lambda(1) # lambda = 0.01

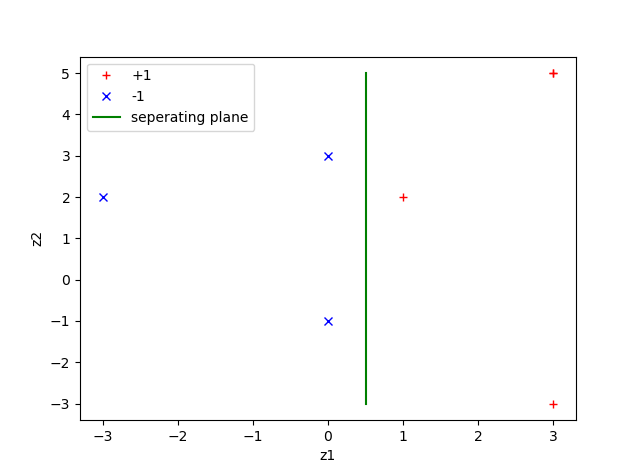
NLT.train\_regularization(X\_train\_one, Y\_train\_one)

print("Ein = %f, Eout = %f" % (NLT.calc\_error(X\_train\_one, Y\_train\_one), NLT.calc\_error(X\_test\_one, Y\_test\_one)))

if \_\_name\_\_== "\_\_main\_\_":

main()

11. [c]



From the figure we can see that z1 = 0.5 is the separating plane in Z space.

In this case, let w1 = 1, w2 = 0. Then b = -0.5. [c] correct

12. [c]

Number of support vectors = 5.

Q11~Q12 are based on the following codes:

import numpy as np

from sklearn import svm

import matplotlib.pyplot as plt

X\_train = np.array([[1,0],[0,1], [0,-1], [-1,0], [0,2],[0,-2], [-2,0]])

Y\_train = np.array([-1,-1,-1,1,1,1,1])

x1 = X\_train[:, :1]

x2 = X\_train[:, 1:]

z1 = pow(x2,2.0) - 2 \* x1 - 1

z2 = pow(x1,2.0) - 2 \* x2 + 1

z1p = np.array([z1[i] for i in np.where(Y\_train == 1)]).flatten()

z2p = np.array([z2[i] for i in np.where(Y\_train == 1)]).flatten()

z1n = np.array([z1[i] for i in np.where(Y\_train == -1)]).flatten()

z2n = np.array([z2[i] for i in np.where(Y\_train == -1)]).flatten()

plt.plot(z1p, z2p, 'r+', label='+1')

plt.plot(z1n, z2n, 'bx', label='-1')

plt.plot([0.5, 0.5], [-3.0, 5.0], 'g-', label='seperating plane')

plt.xlabel('z1')

plt.ylabel('z2')

plt.legend()

plt.show()

clf = svm.SVC(C = np.inf, kernel = 'poly', degree = 2, coef0 = 1, gamma = 1) # C = infinite for hard-margin SVM with 2nd order polynomial kernel with intercept coef0 = 1

# For simplicity, take gamma = 1, same in Lecture 15 page 7

Z = np.c\_[z1, z2]

clf.fit(Z, Y\_train)

print("number of support vectors: ", sum(clf.n\_support\_))

13. [a]

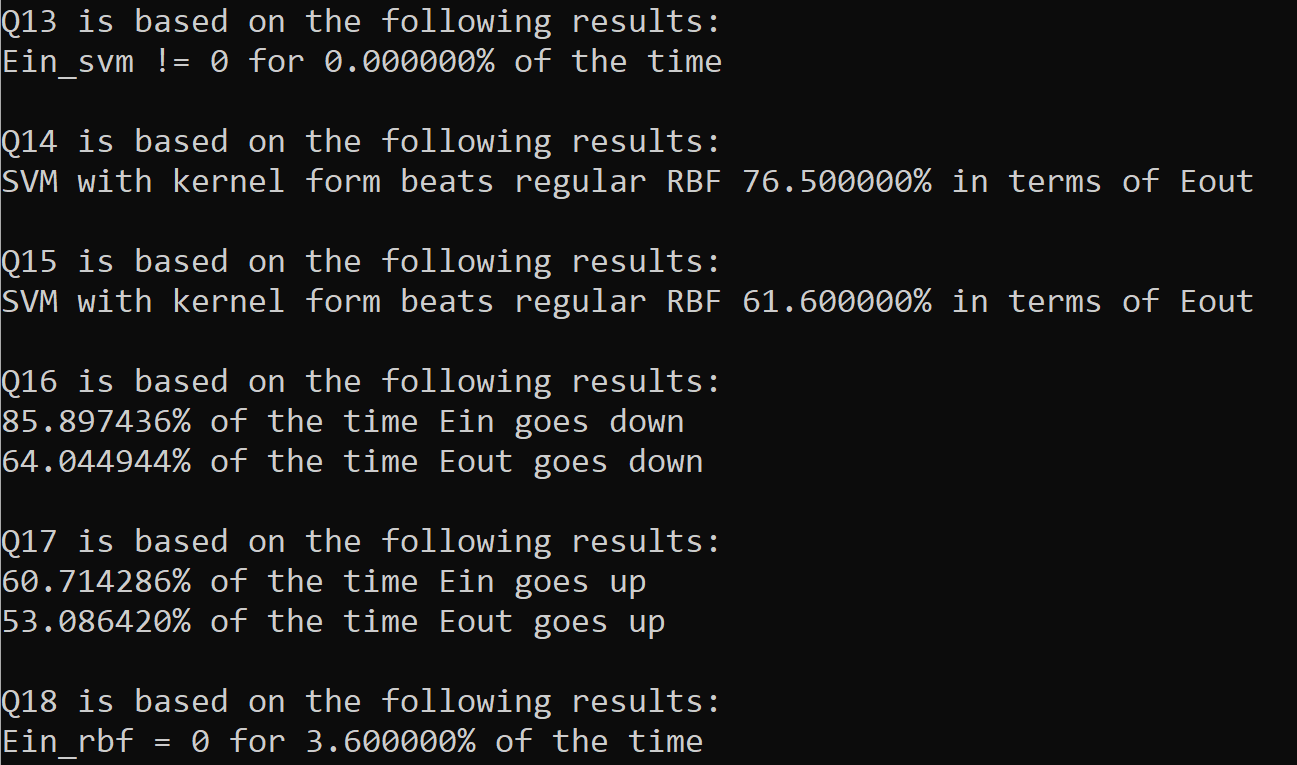
14. [e]

15. [d]

16. [d]

17. [c]

18. [a]



Q13~Q18 are based on the following codes:

import numpy as np

from sklearn import svm

def rb\_func(x, mu, gamma):

return np.exp(-gamma \* np.linalg.norm(x - mu) \*\* 2)

def rb\_matrix(X, mu, gamma, row, column):

rb\_matrix = np.zeros((row, column))

for i in range(row):

for k in range(column):

rb\_matrix[i,k] = rb\_func(X[i], mu[k], gamma)

return rb\_matrix

class kmeans\_RBF:

def \_\_init\_\_(self, num\_clusters = None, gamma = None):

self.K = num\_clusters # K = number of clusters

self.mu = None

self.cluster\_x = None # for each point x the cluster it belongs to

self.weights = None

self.gamma = gamma

def k\_means(self, N\_train, X\_train):

centers = np.random.uniform(-1,1,(self.K,2)) # initial random cluster centers

cluster\_empty = False

cluster\_of\_x = -1 \* np.ones(N\_train)

previous\_cluster = -1 \* np.ones(N\_train) # differentiate no change from iteration to iteration

for i in range(100000): # Lloyd, take maximum iterations as 100000

S = [[] for \_ in range(self.K)] # initialize clusters

for index, x in enumerate(X\_train): # for a given cluster center, assign all points to cluster

min\_dist = np.inf

min\_clst = -1

for mu\_index, mu in enumerate(centers):

dist = np.linalg.norm(x - mu)

if dist < min\_dist:

min\_dist = dist

min\_clst = mu\_index

S[min\_clst].append(x)

cluster\_of\_x[index] = min\_clst

for cluster in S:

if not cluster:

cluster\_empty = True

if cluster\_empty or (cluster\_of\_x == previous\_cluster).all():

break

previous\_cluster = cluster\_of\_x

# [cluster\_index for cluster\_index in cluster\_of\_x]

for index, cluster in enumerate(S):

mu = sum(cluster) / len(cluster) # for a given cluster, assign new cluster centers

centers[index] = mu

return centers, cluster\_of\_x, cluster\_empty

def fit(self, X\_train, y):

while True:

N\_train = X\_train.shape[0]

centers, cluster\_of\_x, cluster\_empty = self.k\_means(N\_train, X\_train)

if (cluster\_empty == True): # if cluster empty, repeat iterations

continue

self.mu = centers

self.cluster\_x = cluster\_of\_x

phi = rb\_matrix(X\_train, self.mu, self.gamma, N\_train, self.K)

phi\_new = np.c\_[np.ones(N\_train), phi] # add 1st column for bias

pinv\_phi = np.linalg.pinv(phi\_new) # pseudo inverse

self.weights = np.dot(pinv\_phi,y)

break

def predict(self, X\_test): #Takes points X, Returns predicted y

N\_test = X\_test.shape[0]

phi = rb\_matrix(X\_test, self.mu, self.gamma, N\_test, self.K)

phi\_new = np.c\_[np.ones(N\_test), phi] # add 1st column for bias

y\_predicted = np.sign(np.dot(phi\_new, self.weights))

return y\_predicted

def f(x1, x2):

return np.sign(x2 - x1 + 0.25 \* np.sin(np.pi \* x1)) # target function

def data\_set(num\_pts):

x1 = np.random.uniform(-1,1,num\_pts)

x2 = np.random.uniform(-1,1,num\_pts)

X\_train = np.c\_[x1, x2]

Y\_train = f(x1, x2)

return X\_train, Y\_train

def compare\_svm\_with\_rbf(K, gammaa):

count = 0

for num\_run in np.arange(1000): # run 1000 times to make sure stable

X\_train, Y\_train = data\_set(100) # each time generate 100 points

X\_test, Y\_test = data\_set(100) # each time generate 100 points

clf = svm.SVC(C = np.inf, kernel = 'rbf', gamma = gammaa)

clf.fit(X\_train, Y\_train)

Ein\_svm = sum(clf.predict(X\_train) != Y\_train) / 100

if Ein\_svm > 0:

continue

Eout\_svm = sum(clf.predict(X\_test) != Y\_test) / 100

rbf = kmeans\_RBF(num\_clusters = K, gamma = gammaa)

rbf.fit(X\_train, Y\_train)

Eout\_rbf = sum(rbf.predict(X\_test) != Y\_test) / 100

if Eout\_svm < Eout\_rbf:

count += 1

print("SVM with kernel form beats regular RBF %f%% in terms of Eout" % (count / 1000 \* 100))

def compare\_regular\_rbf(K, gammaa):

Ein\_down = 0

Ein\_up = 0

Eout\_down = 0

Eout\_up = 0

for num\_run in np.arange(100): # run 100 times

X\_train, Y\_train = data\_set(100) # each time generate 100 points

X\_test, Y\_test = data\_set(100) # each time generate 100 points

Ein = [None, None] #compare regular RBF with K = 9 and K = 12

Eout = [None, None]

for i in np.arange(2):

rbf = kmeans\_RBF(num\_clusters = K[i], gamma = gammaa[i])

rbf.fit(X\_train, Y\_train)

Ein\_rbf = sum(rbf.predict(X\_train) != Y\_train) / 100

Eout\_rbf = sum(rbf.predict(X\_test) != Y\_test) / 100

Ein[i] = Ein\_rbf

Eout[i] = Eout\_rbf

if (Ein[0] > Ein[1]):

Ein\_down += 1

if (Ein[0] < Ein[1]):

Ein\_up += 1

if (Eout[0] > Eout[1]):

Eout\_down += 1

if (Eout[0] < Eout[1]):

Eout\_up += 1

if (Ein\_down > Ein\_up):

print("%f%% of the time Ein goes down" % (Ein\_down / 100 \* 100))

elif (Ein\_down < Ein\_up):

print("%f%% of the time Ein goes up" % (Ein\_up / 100 \* 100))

else:

print("Ein remains the same")

if (Eout\_down > Eout\_up):

print("%f%% of the time Eout goes down" % (Eout\_down / 100 \* 100))

elif (Eout\_down < Eout\_up):

print("%f%% of the time Eout goes up" % (Eout\_up / 100 \* 100))

else:

print("Eout remains the same")

print("")

def main():

print("Q13 is based on the following results:")

Ein = np.array([])

for num\_run in np.arange(1000): # run 1000 times to make sure stable

X\_train, Y\_train = data\_set(100) # each time generate 100 points

clf = svm.SVC(C = np.inf, kernel = 'rbf', gamma = 1.5)

clf.fit(X\_train, Y\_train)

Ein = np.append(Ein, sum(clf.predict(X\_train) != Y\_train) / 100)

Ein\_svm = sum(Ein[i] != 0 for i in np.arange(1000)) / 100

print("Ein\_svm != 0 for %f%% of the time" % (Ein\_svm \* 100))

print("")

print("Q14 is based on the following results:")

compare\_svm\_with\_rbf(9, 1.5)

print("")

print("Q15 is based on the following results:")

compare\_svm\_with\_rbf(12, 1.5)

print("")

print("Q16 is based on the following results:")

compare\_regular\_rbf([9, 12], [1.5, 1.5])

print("Q17 is based on the following results:")

compare\_regular\_rbf([9, 9], [1.5, 2])

print("Q18 is based on the following results:")

count = 0

for num\_run in np.arange(1000): # run 1000 times to make sure stable

X\_train, Y\_train = data\_set(100) # each time generate 100 points

X\_test, Y\_test = data\_set(100) # each time generate 100 points

rbf = kmeans\_RBF(num\_clusters = 9, gamma = 1.5)

rbf.fit(X\_train, Y\_train)

if sum(rbf.predict(X\_train) != Y\_train) / 100 == 0:

count += 1

print("Ein\_rbf = 0 for %f%% of the time" % (count / 1000 \* 100))

if \_\_name\_\_== "\_\_main\_\_":

main()