CS/CNS/EE 156a
Final Exam

Sung Hoon Choi
(Last name: Choi)

1. Answer: [e]
Note that we don't count zeroth coordinate (constants).
1st order terms: $x_1,\ x_2$ (2 terms)
2nd order terms: $x_1x_2,\ {x_1}^2,\ {x_2}^2$ (3 terms)
3rd order terms: ${x_1}^2x_2,\ x_1{x_2}^2,\ {x_1}^3,\ {x_2}^3$ (4 terms)
4th order terms: ${x_1}^2{x_2}^2,\ {x_1}^3x_2,\ x_1{x_2}^3,\ {x_1}^4,\ {x_2}^4$ (5 terms)
5th order terms: …. (6 terms)
…
By the pattern we observe above, we can deduce that for 10th order polynomial transform, the dimensionality of Z space would be:

$$2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 = 65$$

Thus, the answer is 65, and none of the given choices matches the answer.
Therefore, the answer is [e].

2. Answer: [d]
When H has one hypothesis, it is obvious that the average is the hypothesis itself. When H is the set of all constant, the average of hypotheses are also a constant. Thus, the average hypothesis is again within H. When H is the linear regression model, $\mathrm{h(x)} = w^T x$ and since the hypotheses are summations of $\mathrm{w} * \mathrm{x}$, their average would also take the form of $\sum w\, x$. Thus, it's again within H. However, for logistic regression, the hypotheses are nonlinear($\mathrm{h(x)} = \frac{1}{1+e^{-W^T x}}$), and their average may not result in the same sigmoid form. Therefore, the answer is [d].

3. Answer: [d]
[a],[b],[c],[e] are true because the reason why overfitting happens is choosing a misleading hypothesis solely based on smaller E_{in} while the hypothesis with smaller E_{in} turns out to be overfitted and has larger E_{out}. Besides, "overfit measure =E_{out}(h1)-E_{out}(h2)". Thus, we need at least two hypotheses of different E_{in} and E_{out} to discuss overfitting. Also, when overfitting happens, E_{out}-E_{in} gets larger. We need at least two hypotheses of different E_{out}-E_{in} to determine overfitting. However, for [d], we cannot actually guarantee that there is overfitting by simply comparing E_{out}-E_{in}. For example, let's say that there two hypotheses with equal E_{out}-E_{in}. However, the first hypothesis has E_{out} and E_{in} that are higher than the second hypothesis's E_{out} and E_{in}. In this case, we can say that maybe the first hypothesis was just poorly trained since it has a high E_{in}. The second hypothesis with lower E_{in} has a higher possibility of being overfitted. As such, we cannot guarantee if there is overfitting by simply comparing the difference, E_{out}-E_{in}. Therefore, the answer is [d].

4. Answer: [d]
Deterministic noise is a byproduct of having a hypothesis that cannot fully capture the target. For example, if the target function is way too complex to be fully captured by our hypotheses, the noise (limit to capture everything) is determined. This is the deterministic noise. Thus, deterministic noise depends on both the hypothesis and target function. Meanwhile, stochastic noise is a random noise which is often a byproduct of measurement errors or natural fluctuations. Thus, although stochastic noise does not depend on the

hypothesis set, it does depend on the target distribution. As mentioned, stochastic noise is generated when there are random errors in the data (often from measurement errors or environmental fluctuations) when forming the target function. Therefore, stochastic noise depends on the target distribution. If the target is complex and measuring noises exist, deterministic noise and stochastic noise can occur together. Therefore, the only true statement is [d].

5. Answer: [a]
The linear regression solution w_{lin} is obtained by minimizing $\frac{1}{N}\sum_{n=1}^{N}(w^T x_n - y_n)^2$. Meanwhile, the problem states that w_{lin} already satisfies the given constraint: $w_{lin}\Gamma^T\Gamma w_{lin} \le C$. Therefore, the regularized weight w_{reg} is equal to w_{lin}. Therefore, the answer is [a].

6. Answer: [b]
Solving for the regularized w_{reg} with soft-order constraints (Minimize $E_{in}(w) = \frac{1}{N}(Zw - y)^T(Zw - y)$ subject to $w^T w \le C$) is translated into minimizing $E_{aug}(w) = E_{in}(w) + \frac{\lambda}{N} w^T w$ unconditionally. Therefore, the answer is [b]. Choice [a],[c],[d] are incorrect. (Soft-order constraints cannot be written as hard-order constraints since hard constraints simply remove the coefficients above a certain order.(For example, w_q=0 for q>5) Also, VC dimension does not determine soft-order constraints. Lastly, soft-order constraints are used for regularization, not for decreasing both E_{in} and E_{out}).

7. Answer: [d]
The E_{in} values I got were:
5 vs all: 0.076258
6 vs all: 0.091071
7 vs all: 0.088465
8 vs all: 0.074338
9 vs all: 0.088328
Please refer to the code below for the derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 7
import numpy as np

def extract_data(filename):
    data_array = []
    for line in open(filename):
        data=[]
        data_entries = line.split('  ')
        data_row = [float(data_entries[1]),float(data_entries[2]),float(data_entries[3].rstrip("\n"))]
        data_array.append(data_row)
    return np.array(data_array)

def one_vs_all_label(data_array, digit):
    labelled_data_array = []
    for i in range (0, len(data_array)):
        if data_array[i,0] == digit:
            labelled_data_array.append(1)
        else:
            labelled_data_array.append(-1)
    return np.array(labelled_data_array)

def calculate_binary_error(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count/len(g_x)

def calculate_regularized_weight(x, y, Lambda):
    reg_weight_inter1 = np.dot(np.transpose(x),x)+Lambda*np.identity(3)
    reg_weight_inter2 = np.dot(np.linalg.inv(reg_weight_inter1),np.transpose((x)))
    reg_weight = np.dot(reg_weight_inter2,y)
    return reg_weight

extracted = extract_data("features.train.txt")
transformed_x = np.array(extracted[:,0:3])
transformed_x[:,0] = 1

for n in range(5,10):
    y = one_vs_all_label(extracted, n)
    weights = calculate_regularized_weight(transformed_x,y,1)
    weights = np.array([weights])
    g_x = np.squeeze(np.dot(weights,transformed_x.T))
    sign_g_x = np.sign(g_x)
    print("%d vs all: %f " %(n, calculate_binary_error(sign_g_x,y.T)))
```

8. Answer: [b]

The simulation output is as following.

```
0 vs all: 0.121574
1 vs all: 0.024415
2 vs all: 0.098655
3 vs all: 0.082711
4 vs all: 0.099651
```

Therefore, the lowest E_{out} is 1 versus all.

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 8
import numpy as np

def extract_data(filename):
    data_array = []
    for line in open(filename):
        data=[]
        data_entries = line.split('  ')
        data_row = [float(data_entries[1]),float(data_entries[2]),float(data_entries[3].rstrip("\n"))]
        data_array.append(data_row)
    return np.array(data_array)

def one_vs_all_label(data_array, digit):
    labelled_data_array = []
    for i in range (0, len(data_array)):
        if data_array[i,0] == digit:
            labelled_data_array.append(1)
        else:
            labelled_data_array.append(-1)
    return np.array(labelled_data_array)

def calculate_binary_error(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count/len(g_x)

def calculate_regularized_weight(x, y, Lambda):
    reg_weight_inter1 = np.dot(np.transpose(x),x)+Lambda*np.identity(3)
    reg_weight_inter2 = np.dot(np.linalg.inv(reg_weight_inter1),np.transpose((x)))
    reg_weight = np.dot(reg_weight_inter2,y)
    return reg_weight

def nonlinear_trans(x,lineNum):
    nonlin_transformed_data = []
    for i in range(0,lineNum):
        nonlin_transformed_row = []
        nonlin_transformed_row.append(1)
        nonlin_transformed_row.append(x[i][1])
        nonlin_transformed_row.append(x[i][2])
        nonlin_transformed_row.append(x[i][1]*x[i][2])
        nonlin_transformed_row.append(x[i][1]**2)
        nonlin_transformed_row.append((x[i][2]**2))
        nonlin_transformed_data.append(nonlin_transformed_row)
    return nonlin_transformed_data

extracted_in = extract_data("features.train.txt")
transformed_x_in = np.array(extracted_in[:,0:3])
transformed_x_in[:,0] = 1
nonlinear_transformed_x_in = nonlinear_trans(transformed_x_in,len(transformed_x_in))

extracted_out = extract_data("features.test.txt")
transformed_x_out = np.array(extracted_out[:,0:3])
transformed_x_out[:,0] = 1
nonlinear_transformed_x_out = nonlinear_trans(transformed_x_out,len(transformed_x_out))

for n in range(0,5):
    y_out = one_vs_all_label(extracted_out, n)
    weights = calculate_regularized_weight(transformed_x_out,y_out,1)
    weights = np.array([weights])
    g_x = np.squeeze(np.dot(weights,transformed_x_out.T))
    sign_g_x = np.sign(g_x)
    print("%d vs all: %f " %(n, calculate_binary_error(sign_g_x,y_out.T)))
```

9. Answer: [e]

The simulation output(simulation result) is as follows:

```
------------------------------
Ein Without Transform: 0 vs all: 0.109313
Ein With Transform: 0 vs all: 0.102318
Eout Without Transform: 0 vs all: 0.115097
Eout With Transform: 0 vs all: 0.106627
------------------------------
Ein Without Transform: 1 vs all: 0.015224
Ein With Transform: 1 vs all: 0.012344
Eout Without Transform: 1 vs all: 0.022422
Eout With Transform: 1 vs all: 0.021923
------------------------------
Ein Without Transform: 2 vs all: 0.100261
Ein With Transform: 2 vs all: 0.100261
Eout Without Transform: 2 vs all: 0.098655
Eout With Transform: 2 vs all: 0.098655
------------------------------
Ein Without Transform: 3 vs all: 0.090248
Ein With Transform: 3 vs all: 0.090248
Eout Without Transform: 3 vs all: 0.082711
Eout With Transform: 3 vs all: 0.082711
------------------------------
Ein Without Transform: 4 vs all: 0.089425
Ein With Transform: 4 vs all: 0.089425
Eout Without Transform: 4 vs all: 0.099651
Eout With Transform: 4 vs all: 0.099651
------------------------------
Ein Without Transform: 5 vs all: 0.076258
Ein With Transform: 5 vs all: 0.076258
Eout Without Transform: 5 vs all: 0.079721
Eout With Transform: 5 vs all: 0.079223 (Eout is improved by less than 5%)
------------------------------
Ein Without Transform: 6 vs all: 0.091071
Ein With Transform: 6 vs all: 0.091071
Eout Without Transform: 6 vs all: 0.084704
Eout With Transform: 6 vs all: 0.084704
-------------------------------
Ein Without Transform: 7 vs all: 0.088465
Ein With Transform: 7 vs all: 0.088465
Eout Without Transform: 7 vs all: 0.073244
Eout With Transform: 7 vs all: 0.073244
-------------------------------
Ein Without Transform: 8 vs all: 0.074338
Ein With Transform: 8 vs all: 0.074338
Eout Without Transform: 8 vs all: 0.082711
Eout With Transform: 8 vs all: 0.082711
-------------------------------
Ein Without Transform: 9 vs all: 0.088328
Ein With Transform: 9 vs all: 0.088328
Eout Without Transform: 9 vs all: 0.088191
Eout With Transform: 9 vs all: 0.088191
-------------------------------
```

The result shows us that for 5 vs all, E_{out} without transform is 0.079721 while E_{out} with transform is 0.079223. Thus, the transform improved out-of-sample performance of 5 vs all by $\frac{0.079721-0.079223}{0.079721} * 100(\%) = 0.62(\%)$, which is less than 5%. Therefore, the correct answer is [e]. Other choices are all incorrect. We can see from the simulation result that in most cases (except 0 vs all, 1 vs all, and 5 vs all), the transform do not affect the E_{in} and E_{out}. Thus, it is incorrect to say that overfitting always occur when we use the transform, or to say that the transform always improves/worsens E_{out}. Since E_{out} actually gets improved for 5 vs all case, it is also incorrect to say that the transform does not make any change in out-of-sample error. Therefore, the only correct answer is [e].

Please refer to the code below for derivation.

```python
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 9
import numpy as np

def extract_data(filename):
    data_array = []
    for line in open(filename):
        data=[]
        data_entries = line.split('  ')
        data_row = [float(data_entries[1]),float(data_entries[2]),float(data_entries[3].rstrip("\n"))]
        data_array.append(data_row)
    return np.array(data_array)

def one_vs_all_label(data_array, digit):
    labelled_data_array = []
    for i in range (0, len(data_array)):
        if data_array[i,0] == digit:
            labelled_data_array.append(1)
        else:
            labelled_data_array.append(-1)
    return np.array(labelled_data_array)

def calculate_binary_error(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count/len(g_x)

def calculate_regularized_weight(x, y, Lambda, z_param_num):
    reg_weight_inter1 = np.dot(np.transpose(x),x)+Lambda*np.identity(z_param_num)
    reg_weight_inter2 = np.dot(np.linalg.inv(reg_weight_inter1),np.transpose((x)))
    reg_weight = np.dot(reg_weight_inter2,y)
    return reg_weight

def nonlinear_trans(x,lineNum):
    nonlin_transformed_data = []
    for i in range(0,lineNum):
        nonlin_transformed_row = []
        nonlin_transformed_row.append(1)
        nonlin_transformed_row.append(x[i][1])
        nonlin_transformed_row.append(x[i][2])
        nonlin_transformed_row.append(x[i][1]*x[i][2])
        nonlin_transformed_row.append(x[i][1]**2)
        nonlin_transformed_row.append((x[i][2]**2))
        nonlin_transformed_data.append(nonlin_transformed_row)
    return np.array(nonlin_transformed_data)

extracted_in = extract_data("features.train.txt")
transformed_x_in = np.array(extracted_in[:,0:3])
transformed_x_in[:,0] = 1
nonlinear_transformed_x_in = nonlinear_trans(transformed_x_in,len(transformed_x_in))

extracted_out = extract_data("features.test.txt")
transformed_x_out = np.array(extracted_out[:,0:3])
transformed_x_out[:,0] = 1
nonlinear_transformed_x_out = nonlinear_trans(transformed_x_out,len(transformed_x_out))

print("-------------------------------")
for n in range(0,10):
    y_in = one_vs_all_label(extracted_in, n)
    y_out = one_vs_all_label(extracted_out, n)

    weights = calculate_regularized_weight(transformed_x_in,y_in,1,3)
    weights = np.array([weights])

    weights_nonlinear = calculate_regularized_weight(nonlinear_transformed_x_in,y_in,1,6)
    weights_nonlinear = np.array([weights_nonlinear])

    g_x_in = np.squeeze(np.dot(weights,transformed_x_in.T))
    sign_g_x_in = np.sign(g_x_in)
    g_x_out = np.squeeze(np.dot(weights,transformed_x_out.T))
    sign_g_x_out = np.sign(g_x_out)
    g_x_nonlinear_in = np.squeeze(np.dot(weights_nonlinear,nonlinear_transformed_x_in.T))
    sign_g_x_nonlinear_in = np.sign(g_x_nonlinear_in)
    g_x_nonlinear_out = np.squeeze(np.dot(weights_nonlinear,nonlinear_transformed_x_out.T))
    sign_g_x_nonlinear_out = np.sign(g_x_nonlinear_out)
    print("Ein Without Transform: %d vs all: %f" %(n,calculate_binary_error(sign_g_x_in,y_in.T)))
    print("Ein With Transform: %d vs all: %f " %(n, calculate_binary_error(sign_g_x_nonlinear_in,y_in.T)))
    print("Eout Without Transform: %d vs all: %f" %(n,calculate_binary_error(sign_g_x_out,y_out.T)))
    print("Eout With Transform: %d vs all: %f " %(n, calculate_binary_error(sign_g_x_nonlinear_out,y_out.T)))
    print("-------------------------------")
```

10. Answer: [a]
The simulation output(result) is as follows.

```
Lambda = 1.00--------------------------
Ein With Transform: 5 vs all: 0.005125
Eout With Transform: 5 vs all: 0.025943
Lambda = 0.01--------------------------
Ein With Transform: 5 vs all: 0.004484
Eout With Transform: 5 vs all: 0.028302
---------------------------------------------
```

As we can see from the simulation result, as from λ=1 to λ=0.01, E_{in} decreases while E_{out} increases. Thus, overfitting has occurred. The simulation output demonstrates that all other choices [b]~[e] are incorrect. Therefore, the only correct answer is [a].
Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 10
```

```python
import numpy as np

def extract_data(filename):
    data_array = []
    for line in open(filename):
        data=[]
        data_entries = line.split('  ')
        data_row = [float(data_entries[1]),float(data_entries[2]),float(data_entries[3].rstrip("\n"))]
        data_array.append(data_row)
    return np.array(data_array)

def one_vs_one_label(data_array, digit1, digit2):
    labelled_data_array = []
    for i in range (0, len(data_array)):
        if data_array[i,0] == digit1:
            labelled_data_array.append(1)
        elif data_array[i,0] == digit2:
            labelled_data_array.append(-1)
    return np.array(labelled_data_array)

def filter_rest_of_digits(data_array, digit1, digit2):
    filtered_data_array = []
    for i in range(0, len(data_array)):
        if data_array[i,0] == digit1:
            filtered_data_array.append(data_array[i])
        elif data_array[i,0] == digit2:
            filtered_data_array.append(data_array[i])
    return np.array(filtered_data_array)

def calculate_binary_error(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count/len(g_x)

def calculate_regularized_weight(x, y, Lambda, z_param_num):
    reg_weight_inter1 = np.dot(np.transpose(x),x)+Lambda*np.identity(z_param_num)
    reg_weight_inter2 = np.dot(np.linalg.inv(reg_weight_inter1),np.transpose((x)))
    reg_weight = np.dot(reg_weight_inter2,y)
    return reg_weight

def nonlinear_trans(x,lineNum):
    nonlin_transformed_data = []
    for i in range(0,lineNum):
        nonlin_transformed_row = []
        nonlin_transformed_row.append(1)
        nonlin_transformed_row.append(x[i][1])
        nonlin_transformed_row.append(x[i][2])
        nonlin_transformed_row.append(x[i][1]*x[i][2])
        nonlin_transformed_row.append(x[i][1]**2)
        nonlin_transformed_row.append((x[i][2]**2))
        nonlin_transformed_data.append(nonlin_transformed_row)
    return np.array(nonlin_transformed_data)

extracted_in = filter_rest_of_digits(extract_data("features.train.txt"),1,5)
transformed_x_in = np.array(extracted_in[:,0:3])
transformed_x_in[:,0] = 1
nonlinear_transformed_x_in = nonlinear_trans(transformed_x_in,len(transformed_x_in))

extracted_out = filter_rest_of_digits(extract_data("features.test.txt"),1,5)
transformed_x_out = np.array(extracted_out[:,0:3])
transformed_x_out[:,0] = 1
nonlinear_transformed_x_out = nonlinear_trans(transformed_x_out,len(transformed_x_out))

n = 5
y_in = one_vs_one_label(extracted_in, 1, 5)
y_out = one_vs_one_label(extracted_out,1, 5)

for Lambda in [1,0.01]:
    print("Lambda = %.2f---------------------------" %Lambda)
    weights_nonlinear = calculate_regularized_weight(nonlinear_transformed_x_in,y_in,Lambda,6)
    weights_nonlinear = np.array([weights_nonlinear])
    g_x_nonlinear_in = np.squeeze(np.dot(weights_nonlinear,nonlinear_transformed_x_in.T))
    sign_g_x_nonlinear_in = np.sign(g_x_nonlinear_in)
    g_x_nonlinear_out = np.squeeze(np.dot(weights_nonlinear,nonlinear_transformed_x_out.T))
    sign_g_x_nonlinear_out = np.sign(g_x_nonlinear_out)
    print("Ein With Transform: %d vs all: %f " %(n, calculate_binary_error(sign_g_x_nonlinear_in,y_in.T)))
    print("Eout With Transform: %d vs all: %f " %(n, calculate_binary_error(sign_g_x_nonlinear_out,y_out.T)))
print("----------------------------------------------")
```

11. Answer: [c]

Once we do the transform, we get the new z-coordinates and y pairs as following:

(z_1,z_2), y

(-3,2), -1

(0,-1), -1

(0,3), -1

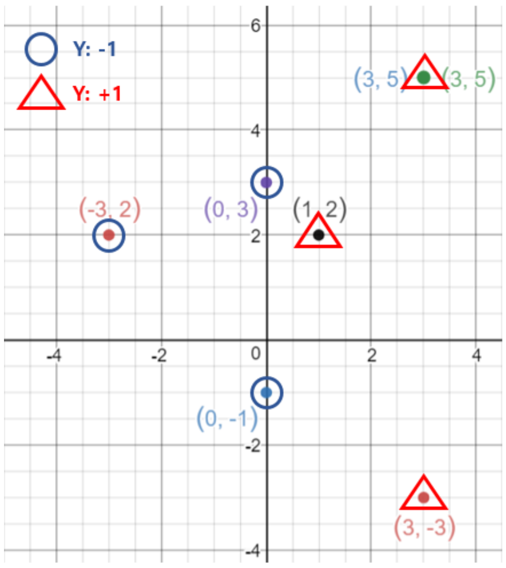
(1,2), +1

(3,-3), +1

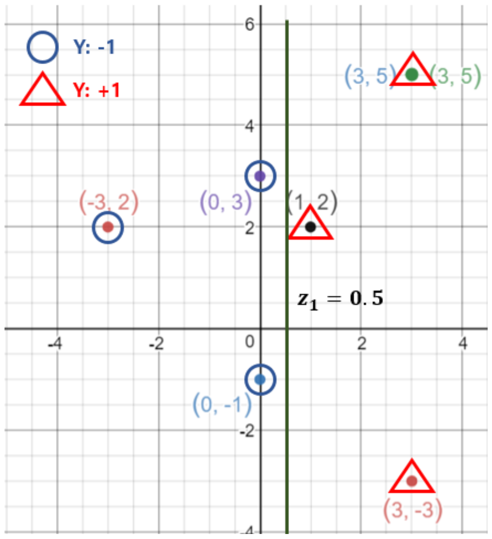
(3,5), +1

(3,5) +1

If we plot these on a graph, we get:



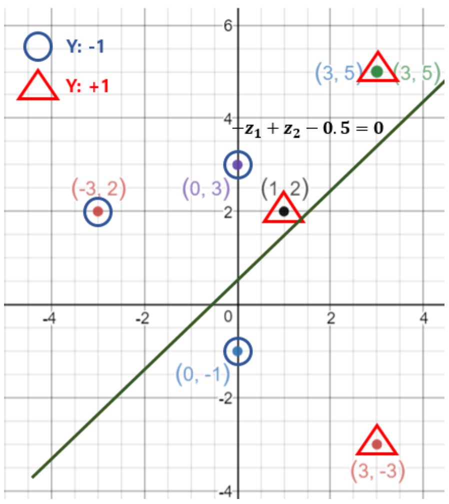
As we can see from the plot, the labels(y) only depend on the horizontal coordinate. (The left side is -1 while the right side is +1.) Thus, we can successfully separate the data points by using a vertical line between x=0 and x=1. The line passing through z_1=0.5 would take its place right between the data points(0,3), (0,-1) and (1,2), successfully separating the data points with a large margin.



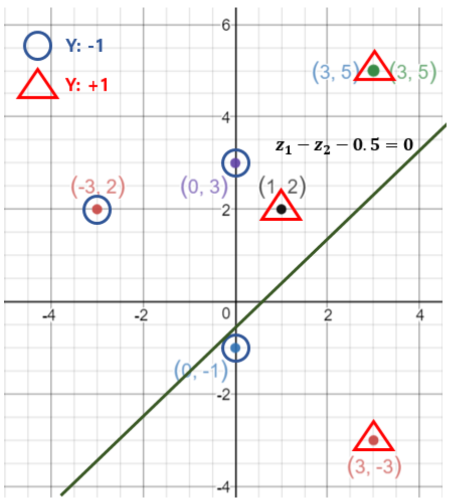
Choice [c] ($z_1 - 0.5 = 0$)

Therefore, since z_1=0.5 can be translated into $1 * z_1 + 0 * z_2 = -0.5$, the answer is [c]. For [a],[b], and [d], you will notice that the given equations fail to correctly separate the data points.

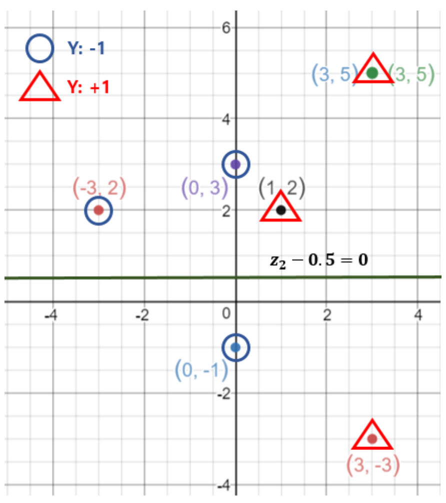
Please refer to the plots below (on the next page) to see how they actually fail to label the data correctly.



Choice [a] $(-z_1 + z_2 - 0.5 = 0)$



Choice [b] $(z_1 - z_2 - 0.5 = 0)$



Choice [d] $(z_2 - 0.5 = 0)$

12. Answer: [c]
The simulation output shows that the number of support vectors is 5.
Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 12

import numpy as np
from sklearn.svm import SVC   #Used for implementing SVM.

x = (((1,0),(0,1),(0,-1),(-1,0),(0,2),(0,-2),(-2,0)))
y = ((-1),(-1),(-1),(1),(1),(1),(1))

clf = SVC(C=1e12, kernel='poly', degree=2, coef0=1.0, gamma= 'auto') #kernel definition
clf.fit(x,y) #fit the SVM
print("vectors: %d" %len(clf.support_)) #number of support vectors
```

13. Answer: [a]
The simulation output was
Total Run: 10000
Not separable data sets: 0

As the result shows, the number of inseparable data set was *zero*. (The experiment was run 10000 times)
Therefore, the answer is [a].
Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 13

import numpy as np
from sklearn.svm import SVC   #Used for implementing SVM.

def generate_random_point():  # generate random data point's coordinate in [-1,1]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

def label_data(x):
    f_x = np.sign(x[1]-x[0]+0.25*np.sin(np.pi*x[0]))
    return f_x

def calculate_binary_error(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count/len(g_x)
```

```
run_num = 10000
not_separable = 0
for run in range(0,run_num):
    x = generate_random_point() #First stack of x coordinate is built manually.
    y = label_data(x)          #First stack of y label is built manually.
    for i in range (0,99):      #Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y,y_new_stack))

    x = np.squeeze(x)
    y = np.squeeze(y)

    clf = SVC(C=1e12, kernel='rbf', degree=2, gamma=1.5)
    clf.fit(x,y)
    g_x = np.array(clf.predict(x))

    if(calculate_binary_error(g_x,y) != 0):
        not_separable = not_separable + 1
print("Total Run: %d \nNot separable data sets: %d" %(run_num,not_separable))
```

14. Answer: [e]

I ran the experiment 100 times for K=9, and it turned out that the kernel method has beaten the regular RBF (Lloyd method) for 86% of time in terms of E$_{out}$.

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 14

import numpy as np
import math
from sklearn.svm import SVC   #Used for implementing SVM.

def generate_random_point():  # generate random data point's coordinate in [-1,1]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

def label_data(x):
    f_x = np.sign(x[1]-x[0]+0.25*np.sin(np.pi*x[0]))
    return f_x

def calculate_distance(x1,x2):
    return math.sqrt((x1[0]-x2[0])**2 + (x1[1]-x2[1])**2)

def calculate_binary_error_abs(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count

not_separable = 0
kernel_beats_regular = 0
K=9 #Problem 14
NUM_DATA = 100
gamma = 1.5
total_run = 100

for run in range(0,total_run):
    ################################TRAINING BEGINS####################################
    check_empty_cluster = 1     #Flag to check if there is an empty cluster.

    #Generate Train data.
    x = generate_random_point()          # First stack of x coordinate is built manually.
    y = label_data(x)                   # First stack of y label is built manually.
    for i in range(0, NUM_DATA-1):       # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))
```

```
x = np.squeeze(x)
y = np.squeeze(y)

#Kernel Form Method
clf = SVC(C=1e12, kernel='rbf', degree=2, gamma=1.5)
clf.fit(x,y)
g_x = np.array(clf.predict(x))
kernel_error = calculate_binary_error_abs(g_x,y)

if(kernel_error != 0):
    not_separable = not_separable + 1

#Regular Form(Lloyd method)
centroid = np.zeros((K,2))
for i in range(0,K):
    centroid[i] = generate_random_point()

#Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
#Tracked by corresponding indices.
dist = np.zeros((1, K))
x_cluster = []
for i in range (0,NUM_DATA):
    for j in range(0, K):
        dist[j] = calculate_distance(x[i],centroid[j])
        dist = np.squeeze(dist)
    x_cluster.append(np.argmin(dist))

for value in range(0,K):
    if value in x_cluster:
        check_empty_cluster = check_empty_cluster*1
    else:
        check_empty_cluster = 0

if(check_empty_cluster!=0 and not_separable == 0 ):
    new_centroid = np.zeros((K, 2))
    while(1): #Run until the clusters converge.
        #Find the new mu k. (Find the new centroid coordinates)
        for i in range(0,K):
            sum = np.zeros((K, 2))
            num_cluster_data = 0
            for j in range (0,NUM_DATA):
                if(x_cluster[j] == i):
                    sum[i] = sum[i] + x[j]
                    num_cluster_data = num_cluster_data + 1
            new_centroid[i] = sum[i]/num_cluster_data

        prev_cluster = x_cluster
        new_dist = np.zeros((1, K))
        new_x_cluster = []
        #Find the new S k. (Label the data points according to the new centroids)
        for i in range (0,NUM_DATA):
            for j in range(0, K):
                new_dist[j] = calculate_distance(x[i],new_centroid[j])
                new_dist = np.squeeze(new_dist)
            new_x_cluster.append(np.argmin(new_dist))
        x_cluster = new_x_cluster
        if(np.array_equal(prev_cluster,new_x_cluster)): #If converged.
            break

    #Calculate phi
    phi = np.zeros((NUM_DATA,K))
    for i in range(0,K):
        for j in range(0,NUM_DATA):
            phi[j,i] = math.exp(-gamma*(calculate_distance(x[j],new_centroid[i]))**2)

    weights = np.matmul(np.linalg.pinv(phi),y)
    ###Now we have the weights. Let's test the hypothesis by using the test data###
    ################################TEST BEGINS#####################################
    #Generate Test data
    x = generate_random_point()      # First stack of x coordinate is built manually.
    y = label_data(x)               # First stack of y label is built manually.
    for i in range(0, NUM_DATA - 1):  # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))
        g_x = np.array(clf.predict(x))
        kernel_error = calculate_binary_error_abs(g_x, y)

        # Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
        # Tracked by indices.
        dist = np.zeros((1, K))
        x_cluster = []
        for i in range(0, NUM_DATA):
            for j in range(0, K):
                dist[j] = calculate_distance(x[i], new_centroid[j])
                dist = np.squeeze(dist)
            x_cluster.append(np.argmin(dist))

        for value in range(0, K):
            if value in x_cluster:
                check_empty_cluster = check_empty_cluster * 1
            else:
                check_empty_cluster = 0

        if (check_empty_cluster != 0 and not_separable == 0):
            new_centroid = np.zeros((K, 2))
            for mini_run in range(1, 50):
                # print("x_cluster", x_cluster)
                # Find the new mu k. (Find the new centroid coordinates)
                for i in range(0, K):
                    sum = np.zeros((K, 2))
                    num_cluster_data = 0
                    for j in range(0, NUM_DATA):
                        if (x_cluster[j] == i):
                            sum[i] = sum[i] + x[j]
                            num_cluster_data = num_cluster_data + 1
                    new_centroid[i] = sum[i] / num_cluster_data

                prev_cluster = x_cluster
                new_dist = np.zeros((1, K))
                new_x_cluster = []
                for i in range(0, NUM_DATA):
                    for j in range(0, K):
                        new_dist[j] = calculate_distance(x[i], new_centroid[j])
                        new_dist = np.squeeze(new_dist)
                    new_x_cluster.append(np.argmin(new_dist))
                x_cluster = new_x_cluster
                if (np.array_equal(prev_cluster, new_x_cluster)):  # If converged.
                    break

            phi = np.zeros((NUM_DATA, K))
            for i in range(0, K):
                for j in range(0, NUM_DATA):
                    phi[j, i] = math.exp(-gamma * (calculate_distance(x[j], new_centroid[i])) ** 2)

            h = np.sign(np.matmul(phi,weights))

            regular_error = 0
            for i in range(0,NUM_DATA):
                if(h[i] != y[i]):
                    regular_error = regular_error+ 1
            print("regular error: ", regular_error)
            print("kernel_error: ", kernel_error)
            #print("regular(lloyd) error: ", regular_error)

            if (kernel_error < regular_error):
                kernel_beats_regular = kernel_beats_regular + 1

print("kernel beats regular: %.2f percent" % ((kernel_beats_regular / total_run) * 100))
```

15. Answer: [d]

I ran the experiment 100 times for K=12, and it turned out that the kernel method has beaten the regular RBF (Lloyd method) for 71% of time in terms of E_{out}.

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 15
```

```
import numpy as np
import math
from sklearn.svm import SVC   #Used for implementing SVM.

def generate_random_point():  # generate random data point's coordinate in [-1,1]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

def label_data(x):
    f_x = np.sign(x[1]-x[0]+0.25*np.sin(np.pi*x[0]))
    return f_x

def calculate_distance(x1,x2):
    return math.sqrt((x1[0]-x2[0])**2 + (x1[1]-x2[1])**2)

def calculate_binary_error_abs(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count

not_separable = 0
kernel_beats_regular = 0
K=12 #Problem 15
NUM_DATA = 100
gamma = 1.5
total_run = 100

for run in range(0,total_run):
    ################################TRAINING BEGINS#####################################
    check_empty_cluster = 1     #Flag to check if there is an empty cluster.

    #Generate Train data.
    x = generate_random_point()         # First stack of x coordinate is built manually.
    y = label_data(x)                  # First stack of y label is built manually.
    for i in range(0, NUM_DATA-1):      # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))

    x = np.squeeze(x)
    y = np.squeeze(y)

    #Kernel Form Method
    clf = SVC(C=1e12, kernel='rbf', degree=2, gamma=1.5)
    clf.fit(x,y)
    g_x = np.array(clf.predict(x))
    kernel_error = calculate_binary_error_abs(g_x,y)

    if(kernel_error != 0):
        not_separable = not_separable + 1

    #Regular Form(Lloyd method)
    centroid = np.zeros((K,2))
    for i in range(0,K):
        centroid[i] = generate_random_point()

    #Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
    #Tracked by corresponding indices.
    dist = np.zeros((1, K))
    x_cluster = []
    for i in range (0,NUM_DATA):
        for j in range(0, K):
            dist[j] = calculate_distance(x[i],centroid[j])
            dist = np.squeeze(dist)
        x_cluster.append(np.argmin(dist))

    for value in range(0,K):
        if value in x_cluster:
            check_empty_cluster = check_empty_cluster*1
        else:
            check_empty_cluster = 0
```

```
if(check_empty_cluster!=0 and not_separable == 0 ):
    new_centroid = np.zeros((K, 2))
    while(1): #Run until the clusters converge.
        #Find the new mu k. (Find the new centroid coordinates)
        for i in range(0,K):
            sum = np.zeros((K, 2))
            num_cluster_data = 0
            for j in range (0,NUM_DATA):
                if(x_cluster[j] == i):
                    sum[i] = sum[i] + x[j]
                    num_cluster_data = num_cluster_data + 1
            new_centroid[i] = sum[i]/num_cluster_data

        prev_cluster = x_cluster
        new_dist = np.zeros((1, K))
        new_x_cluster = []
        #Find the new S k. (Label the data points according to the new centroids)
        for i in range (0,NUM_DATA):
            for j in range(0, K):
                new_dist[j] = calculate_distance(x[i],new_centroid[j])
                new_dist = np.squeeze(new_dist)
            new_x_cluster.append(np.argmin(new_dist))
        x_cluster = new_x_cluster
        if(np.array_equal(prev_cluster,new_x_cluster)): #If converged.
            break

    #Calculate phi
    phi = np.zeros((NUM_DATA,K))
    for i in range(0,K):
        for j in range(0,NUM_DATA):
            phi[j,i] = math.exp(-gamma*(calculate_distance(x[j],new_centroid[i]))**2)

    weights = np.matmul(np.linalg.pinv(phi),y)
    ###Now we have the weights. Let's test the hypothesis by using the test data###
    ################################TEST BEGINS#####################################
    #Generate Test data
    x = generate_random_point()      # First stack of x coordinate is built manually.
    y = label_data(x)               # First stack of y label is built manually.
    for i in range(0, NUM_DATA - 1):  # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))

    g_x = np.array(clf.predict(x))
    kernel_error = calculate_binary_error_abs(g_x, y)

    # Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
    # Tracked by indices.
    dist = np.zeros((1, K))
    x_cluster = []
    for i in range(0, NUM_DATA):
        for j in range(0, K):
            dist[j] = calculate_distance(x[i], new_centroid[j])
            dist = np.squeeze(dist)
        x_cluster.append(np.argmin(dist))

    for value in range(0, K):
        if value in x_cluster:
            check_empty_cluster = check_empty_cluster * 1
        else:
            check_empty_cluster = 0

    if (check_empty_cluster != 0 and not_separable == 0):
        new_centroid = np.zeros((K, 2))
        for mini_run in range(1, 50):
            # print("x_cluster", x_cluster)
            # Find the new mu k. (Find the new centroid coordinates)
            for i in range(0, K):
                sum = np.zeros((K, 2))
                num_cluster_data = 0
                for j in range(0, NUM_DATA):
                    if (x_cluster[j] == i):
                        sum[i] = sum[i] + x[j]
                        num_cluster_data = num_cluster_data + 1
                new_centroid[i] = sum[i] / num_cluster_data

            prev_cluster = x_cluster
            new_dist = np.zeros((1, K))
            new_x_cluster = []
            for i in range(0, NUM_DATA):
                for j in range(0, K):
                    new_dist[j] = calculate_distance(x[i], new_centroid[j])
                    new_dist = np.squeeze(new_dist)
                new_x_cluster.append(np.argmin(new_dist))
            x_cluster = new_x_cluster
            if (np.array_equal(prev_cluster, new_x_cluster)):  # If converged.
                break

        phi = np.zeros((NUM_DATA, K))
        for i in range(0, K):
            for j in range(0, NUM_DATA):
                phi[j, i] = math.exp(-gamma * (calculate_distance(x[j], new_centroid[i])) ** 2)

        h = np.sign(np.matmul(phi,weights))

        regular_error = 0
        for i in range(0,NUM_DATA):
            if(h[i] != y[i]):
                regular_error = regular_error+ 1
        print("regular error: ", regular_error)
        print("kernel_error: ", kernel_error)
        #print("regular(lloyd) error: ", regular_error)

        if (kernel_error < regular_error):
            kernel_beats_regular = kernel_beats_regular + 1

print("kernel beats regular: %.2f percent" % ((kernel_beats_regular / total_run) * 100))
```

16. Answer: [a] or [d]

The simulation shows that as K=9→K=12, E_{in} clearly goes down for most of times.

However, the behavior of E_{out} turned out to be inconsistent: For some runs it went down while for some runs it went up. Thus, it is hard to tell about E_{out}.
Since I'm sure that E_{in} definitely goes down most often but not sure about E_{out}'s behavior, I would hedge the choices: [a] and [d].
Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 16

import numpy as np
import math

def generate_random_point():  # generate random data point's coordinate in [-1,1]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

def label_data(x):
    f_x = np.sign(x[1]-x[0]+0.25*np.sin(np.pi*x[0]))
    return f_x

def calculate_distance(x1,x2):
    return math.sqrt((x1[0]-x2[0])**2 + (x1[1]-x2[1])**2)

def calculate_binary_error_abs(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count

not_separable = 0
kernel_beats_regular = 0
K=9
```

```
NUM_DATA = 100
gamma = 1.5
total_run = 20

for run in range(0,total_run):
    ################################TRAINING BEGINS####################################
    check_empty_cluster = 1     #Flag to check if there is an empty cluster.

    #Generate Train data.
    x = generate_random_point()         # First stack of x coordinate is built manually.
    y = label_data(x)                  # First stack of y label is built manually.
    for i in range(0, NUM_DATA-1):      # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))

    x = np.squeeze(x)
    y = np.squeeze(y)

    for K in [9,12]: #Problem 16
        print("K = %.1f" %(K))
        #Regular Form(Lloyd method)
        centroid = np.zeros((K,2))
        for i in range(0,K):
            centroid[i] = generate_random_point()

        #Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
        #Tracked by corresponding indices.
        dist = np.zeros((1, K))
        x_cluster = []
        for i in range (0,NUM_DATA):
            for j in range(0, K):
                dist[j] = calculate_distance(x[i],centroid[j])
                dist = np.squeeze(dist)
            x_cluster.append(np.argmin(dist))

        for value in range(0,K):
            if value in x_cluster:
                check_empty_cluster = check_empty_cluster*1
            else:
                check_empty_cluster = 0

        if(check_empty_cluster!=0):
            new_centroid = np.zeros((K, 2))
            while(1): #Run until the clusters converge.
                #Find the new mu k. (Find the new centroid coordinates)
                for i in range(0,K):
                    sum = np.zeros((K, 2))
                    num_cluster_data = 0
                    for j in range (0,NUM_DATA):
                        if(x_cluster[j] == i):
                            sum[i] = sum[i] + x[j]
                            num_cluster_data = num_cluster_data + 1
                    new_centroid[i] = sum[i]/num_cluster_data

                prev_cluster = x_cluster
                new_dist = np.zeros((1, K))
                new_x_cluster = []
                #Find the new S k. (Label the data points according to the new centroids)
                for i in range (0,NUM_DATA):
                    for j in range(0, K):
                        new_dist[j] = calculate_distance(x[i],new_centroid[j])
                        new_dist = np.squeeze(new_dist)
                    new_x_cluster.append(np.argmin(new_dist))
                x_cluster = new_x_cluster
                if(np.array_equal(prev_cluster,new_x_cluster)): #If converged.
                    break

            #Calculate phi
            phi = np.zeros((NUM_DATA,K))
            for i in range(0,K):
                for j in range(0,NUM_DATA):
                    phi[j,i] = math.exp(-gamma*(calculate_distance(x[j],new_centroid[i]))**2)

            weights = np.matmul(np.linalg.pinv(phi),y)
            h = np.sign(np.matmul(phi, weights))

            regular_in_error = 0
            for i in range(0, NUM_DATA):
                if (h[i] != y[i]):
                    regular_in_error = regular_in_error + 1
            print("E_in error: ", regular_in_error)
            ###Now we have the weights. Let's test the hypothesis by using the test data###
            ################################TEST BEGINS####################################
            #Generate Test data
            x = generate_random_point()      # First stack of x coordinate is built manually.
            y = label_data(x)               # First stack of y label is built manually.
            for i in range(0, NUM_DATA - 1):  # Remaining 99 stacks are built by for loop.
                x_new_stack = generate_random_point()
                y_new_stack = label_data(x_new_stack)
                x = np.vstack((x, x_new_stack))
                y = np.vstack((y, y_new_stack))


            # Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
            # Tracked by indices.
            dist = np.zeros((1, K))
            x_cluster = []
            for i in range(0, NUM_DATA):
                for j in range(0, K):
                    dist[j] = calculate_distance(x[i], new_centroid[j])
                    dist = np.squeeze(dist)
                x_cluster.append(np.argmin(dist))

            for value in range(0, K):
                if value in x_cluster:
                    check_empty_cluster = check_empty_cluster * 1
                else:
                    check_empty_cluster = 0

            if (check_empty_cluster != 0 and not_separable == 0):
                new_centroid = np.zeros((K, 2))
                for mini_run in range(1, 50):
                    # print("x_cluster", x_cluster)
                    # Find the new mu k. (Find the new centroid coordinates)
                    for i in range(0, K):
                        sum = np.zeros((K, 2))
                        num_cluster_data = 0
                        for j in range(0, NUM_DATA):
                            if (x_cluster[j] == i):
                                sum[i] = sum[i] + x[j]
                                num_cluster_data = num_cluster_data + 1
                        new_centroid[i] = sum[i] / num_cluster_data

                    prev_cluster = x_cluster
                    new_dist = np.zeros((1, K))
                    new_x_cluster = []
                    for i in range(0, NUM_DATA):
                        for j in range(0, K):
                            new_dist[j] = calculate_distance(x[i], new_centroid[j])
                            new_dist = np.squeeze(new_dist)
                        new_x_cluster.append(np.argmin(new_dist))
                    x_cluster = new_x_cluster
                    if (np.array_equal(prev_cluster, new_x_cluster)):  # If converged.
                        break

                phi = np.zeros((NUM_DATA, K))
                for i in range(0, K):
                    for j in range(0, NUM_DATA):
                        phi[j, i] = math.exp(-gamma * (calculate_distance(x[j], new_centroid[i])) ** 2)

                h = np.sign(np.matmul(phi,weights))

                regular_out_error = 0
                for i in range(0,NUM_DATA):
                    if(h[i] != y[i]):
                        regular_out_error = regular_out_error+ 1
                print("E_out error: ", regular_out_error)
    print("--------------------------------------------------")
```

17. Answer: [b] or [c]

E_{in}'s behavior was clear: E_{in} increases for most of times as we go from $\gamma=1.5 \rightarrow \gamma=2$. However, E_{out}'s behavior was inconsistent as I repeated the experiments. Therefore, I would hedge the choices: [b] an [c].

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 17

import numpy as np
import math

def generate_random_point():  # generate random data point's coordinate in [-1,1]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

def label_data(x):
    f_x = np.sign(x[1]-x[0]+0.25*np.sin(np.pi*x[0]))
    return f_x

def calculate_distance(x1,x2):
    return math.sqrt((x1[0]-x2[0])**2 + (x1[1]-x2[1])**2)

def calculate_binary_error_abs(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count

not_separable = 0
kernel_beats_regular = 0
K=9
NUM_DATA = 100
gamma = 1.5
total_run = 20


for run in range(0,total_run):
    ################################TRAINING BEGINS####################################
    check_empty_cluster = 1    #Flag to check if there is an empty cluster.

    #Generate Train data.
    x = generate_random_point()         # First stack of x coordinate is built manually.
    y = label_data(x)                  # First stack of y label is built manually.
    for i in range(0, NUM_DATA-1):      # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))

    x = np.squeeze(x)
    y = np.squeeze(y)

    for gamma in [1.5,2]: #Problem 17
        print("gamma = %.1f" %(gamma))
        #Regular Form(Lloyd method)
        centroid = np.zeros((K,2))
        for i in range(0,K):
            centroid[i] = generate_random_point()

        #Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
        #Tracked by corresponding indices.
        dist = np.zeros((1, K))
        x_cluster = []
        for i in range (0,NUM_DATA):
            for j in range(0, K):
                dist[j] = calculate_distance(x[i],centroid[j])
                dist = np.squeeze(dist)
            x_cluster.append(np.argmin(dist))

        for value in range(0,K):
            if value in x_cluster:
                check_empty_cluster = check_empty_cluster*1
            else:
                check_empty_cluster = 0

        if(check_empty_cluster!=0):
            new_centroid = np.zeros((K, 2))
            while(1): #Run until the clusters converge.
                #Find the new mu k. (Find the new centroid coordinates)
                for i in range(0,K):
                    sum = np.zeros((K, 2))
                    num_cluster_data = 0
                    for j in range (0,NUM_DATA):
                        if(x_cluster[j] == i):
                            sum[i] = sum[i] + x[j]
                            num_cluster_data = num_cluster_data + 1
                    new_centroid[i] = sum[i]/num_cluster_data

                prev_cluster = x_cluster
                new_dist = np.zeros((1, K))
                new_x_cluster = []
                #Find the new S k. (Label the data points according to the new centroids)
                for i in range (0,NUM_DATA):
                    for j in range(0, K):
                        new_dist[j] = calculate_distance(x[i],new_centroid[j])
                        new_dist = np.squeeze(new_dist)
                    new_x_cluster.append(np.argmin(new_dist))
                x_cluster = new_x_cluster
                if(np.array_equal(prev_cluster,new_x_cluster)): #If converged.
                    break

            #Calculate phi
            phi = np.zeros((NUM_DATA,K))
            for i in range(0,K):
                for j in range(0,NUM_DATA):
                    phi[j,i] = math.exp(-gamma*(calculate_distance(x[j],new_centroid[i]))**2)

            weights = np.matmul(np.linalg.pinv(phi),y)
            h = np.sign(np.matmul(phi, weights))

            regular_in_error = 0
            for i in range(0, NUM_DATA):
                if (h[i] != y[i]):
                    regular_in_error = regular_in_error + 1
            print("E_in error: ", regular_in_error)
            ###Now we have the weights. Let's test the hypothesis by using the test data###
            ################################TEST BEGINS#####################################
            #Generate Test data
            x = generate_random_point()       # First stack of x coordinate is built manually.
            y = label_data(x)                # First stack of y label is built manually.
            for i in range(0, NUM_DATA - 1):  # Remaining 99 stacks are built by for loop.
                x_new_stack = generate_random_point()
                y_new_stack = label_data(x_new_stack)
                x = np.vstack((x, x_new_stack))
                y = np.vstack((y, y_new_stack))


            # Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
            # Tracked by indices.
            dist = np.zeros((1, K))
            x_cluster = []
            for i in range(0, NUM_DATA):
                for j in range(0, K):
                    dist[j] = calculate_distance(x[i], new_centroid[j])
                    dist = np.squeeze(dist)
                x_cluster.append(np.argmin(dist))

            for value in range(0, K):
                if value in x_cluster:
                    check_empty_cluster = check_empty_cluster * 1
                else:
                    check_empty_cluster = 0

            if (check_empty_cluster != 0 and not_separable == 0):
                new_centroid = np.zeros((K, 2))
                for mini_run in range(1, 50):
                    # print("x_cluster", x_cluster)
                    # Find the new mu k. (Find the new centroid coordinates)
                    for i in range(0, K):
                        sum = np.zeros((K, 2))
                        num_cluster_data = 0
                        for j in range(0, NUM_DATA):
                            if (x_cluster[j] == i):
                                sum[i] = sum[i] + x[j]
                                num_cluster_data = num_cluster_data + 1
                        new_centroid[i] = sum[i] / num_cluster_data

                    prev_cluster = x_cluster
                    new_dist = np.zeros((1, K))
                    new_x_cluster = []
                    for i in range(0, NUM_DATA):
                        for j in range(0, K):
                            new_dist[j] = calculate_distance(x[i], new_centroid[j])
                            new_dist = np.squeeze(new_dist)
                        new_x_cluster.append(np.argmin(new_dist))
                    x_cluster = new_x_cluster
                    if (np.array_equal(prev_cluster, new_x_cluster)):  # If converged.
                        break

                phi = np.zeros((NUM_DATA, K))
                for i in range(0, K):
                    for j in range(0, NUM_DATA):
                        phi[j, i] = math.exp(-gamma * (calculate_distance(x[j], new_centroid[i])) ** 2)

                h = np.sign(np.matmul(phi,weights))

                regular_out_error = 0
                for i in range(0,NUM_DATA):
                    if(h[i] != y[i]):
                        regular_out_error = regular_out_error+ 1
                print("E_out error: ", regular_out_error)
            print("-------------------------------------------------")
```

18. Answer: [a]

The simulation result shows that the regular RBF achieves E_{in}=0 for 1~2 percent of times.(Experiment run: total 100 times) with K=9 and γ=1.5. Thus, the answer is [a].

Please refer to the code below for derivation.

```
#Sung Hoon Choi
#CS/CNS/EE156a FINAL Problem 18

import numpy as np
import math

def generate_random_point():  # generate random data point's coordinate in [-1,1]
    x = np.zeros(2)
    x[0] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    x[1] = (1 if np.random.rand(1) < 0.5 else -1) * np.random.rand(1)
    return x

def label_data(x):
    f_x = np.sign(x[1]-x[0]+0.25*np.sin(np.pi*x[0]))
    return f_x

def calculate_distance(x1,x2):
    return math.sqrt((x1[0]-x2[0])**2 + (x1[1]-x2[1])**2)

def calculate_binary_error(g_x, f_x):
    error_count = 0
    for i in range (0,len(g_x)):
        if(g_x[i] != f_x[i]):
            error_count = error_count + 1
    return error_count/len(g_x)

kernel_beats_regular = 0
K=9
NUM_DATA = 100
gamma = 1.5
total_run = 100
Ein_is_zero = 0
for run in range(0,total_run):
    check_empty_cluster = 1
    x = generate_random_point()         # First stack of x coordinate is built manually.
    y = label_data(x)                  # First stack of y label is built manually.
    for i in range(0, NUM_DATA-1):      # Remaining 99 stacks are built by for loop.
        x_new_stack = generate_random_point()
        y_new_stack = label_data(x_new_stack)
        x = np.vstack((x, x_new_stack))
        y = np.vstack((y, y_new_stack))

    x = np.squeeze(x)
    y = np.squeeze(y)

    centroid = np.zeros((K,2))
    for i in range(0,K):
        centroid[i] = generate_random_point()

    #Assign clusters to the data points. (Save all the cluster labels in a separate x_cluster)
    #Tracked by indices.
    dist = np.zeros((1, K))
    x_cluster = []
    for i in range (0,NUM_DATA):
        for j in range(0, K):
            dist[j] = calculate_distance(x[i],centroid[j])
            dist = np.squeeze(dist)
        x_cluster.append(np.argmin(dist))


    for value in range(0,K):
        if value in x_cluster:
            check_empty_cluster = check_empty_cluster*1
        else:
            check_empty_cluster = 0

    if(check_empty_cluster!=0): #Run the code only if there's no empty cluster.
        new_centroid = np.zeros((K, 2))
        for mini_run in range(1,50):
            #print("x_cluster", x_cluster)
            #Find the new mu k. (Find the new centroid coordinates)
            for i in range(0,K):
                sum = np.zeros((K, 2))
                num_cluster_data = 0
                for j in range (0,NUM_DATA):
                    if(x_cluster[j] == i):
                        sum[i] = sum[i] + x[j]
                        num_cluster_data = num_cluster_data + 1
                new_centroid[i] = sum[i]/num_cluster_data

            prev_cluster = x_cluster
            new_dist = np.zeros((1, K))
            new_x_cluster = []
            for i in range (0,NUM_DATA):
                for j in range(0, K):
                    new_dist[j] = calculate_distance(x[i],new_centroid[j])
                    new_dist = np.squeeze(new_dist)
                new_x_cluster.append(np.argmin(new_dist))
            x_cluster = new_x_cluster
            if(np.array_equal(prev_cluster,new_x_cluster)): #If converged.
                break

        kernel = np.zeros((NUM_DATA,K))
        for i in range(0,K):
            for j in range(0,NUM_DATA):
                kernel[j,i] = math.exp(-gamma*(calculate_distance(x[j],new_centroid[i]))**2)

        weights = np.matmul(np.linalg.pinv(kernel),y)
        h = np.sign(np.matmul(kernel,weights))

        error = 0
        for i in range(0,NUM_DATA):
            if(h[i] != y[i]):
                error = error + 1

        #print("error_in: ", error)
        if(error == 0):
            Ein_is_zero = Ein_is_zero + 1
print("Ein is zero: %.2f percent" %((Ein_is_zero/total_run)*100))
```

19. Answer: [b]

$$\mathrm{P}(h=f|D)=\frac{P(D|h=f)P(h=f)}{P(D)}\propto \mathrm{P}(D|h=f)P(h=f)$$

It is given by the problem that P(h=f) is uniform in [0,1]. Thus, P(h=f|D=1) is proportional to P(D=1|h=f) while P(D=1|h=f) means "The probability that the selected person has a heart attack when we assumed that our hypothesis actually matches the target distribution". In other words, P(D=1|h=f) increases linearly as *h* increases. Therefore, by the Bayesian inference equation, P(h=f|D=1) increases linearly over [0,1]. Thus, the answer is [b].

20. Answer: [c]

$$\boldsymbol{E_{out}(g)}=\frac{1}{N}\sum_{n=1}^{N}\left(\frac{1}{2}\big(g_1(x_n)+g_2(x_n)\big)-y_n\right)^2$$

$$=\frac{1}{N}\sum_{n=1}^{N}\frac{1}{4}(g_1(x_n)+g_2(x_n))^2-\big(g_1(x_n)+g_2(x_n)\big)y_n+{y_n}^2$$

$$\textbf{Average of } \boldsymbol{E_{out}(g_1)} \textbf{ and } \boldsymbol{E_{out}(g_2)}=\frac{1}{2}\left(\frac{1}{N}\sum_{n=1}^{N}(g_1(x_n)-y_n)^2+\frac{1}{N}\sum_{n=1}^{N}(g_2(x_n)-y_n)^2\right)$$

$$=\frac{1}{N}\left(\frac{1}{2}\sum_{n=1}^{N}(g_1(x_n)-y_n)^2+\frac{1}{2}\sum_{n=1}^{N}(g_2(x_n)-y_n)^2\right)$$

$$=\frac{1}{N}\left(\frac{1}{2}\sum_{n=1}^{N}(g_1(x_n)-y_n)^2+(g_2(x_n)-y_n)^2\right)$$

$$=\frac{1}{N}\left(\frac{1}{2}\sum_{n=1}^{N}g_1(x_n)^2+g_2(x_n)^2-2g_1(x_n)y_n-2g_2(x_n)y_n+2{y_n}^2\right)$$

$$=\frac{1}{N}\left(\sum_{n=1}^{N}\frac{1}{2}g_1(x_n)^2+\frac{1}{2}g_2(x_n)^2-g_1(x_n)y_n-g_2(x_n)y_n+{y_n}^2\right)$$

$$=\frac{1}{N}\left(\sum_{n=1}^{N}\frac{1}{4}g_1(x_n)^2+\frac{1}{4}g_2(x_n)^2+\frac{1}{4}g_1(x_n)^2+\frac{1}{4}g_2(x_n)^2+\frac{1}{2}g_1(x_n)g_2(x_n)-\frac{1}{2}g_1(x_n)g_2(x_n)-g_1(x_n)y_n-g_2(x_n)y_n+{y_n}^2\right)$$

$$=\frac{1}{N}\left(\sum_{n=1}^{N}\frac{1}{4}(g_1(x_n)+g_2(x_n))^2-g_1(x_n)y_n-g_2(x_n)y_n+{y_n}^2+\frac{1}{4}g_1(x_n)^2+\frac{1}{4}g_2(x_n)^2-\frac{1}{2}g_1(x_n)g_2(x_n)\right)$$

$$=\frac{1}{N}\left(\sum_{n=1}^{N}\frac{1}{4}(g_1(x_n)+g_2(x_n))^2-\big(g_1(x_n)+g_2(x_n)\big)y_n+{y_n}^2+\frac{1}{4}g_1(x_n)^2+\frac{1}{4}g_2(x_n)^2-\frac{1}{2}g_1(x_n)g_2(x_n)\right)$$

$$=E_{out}(g)+\frac{1}{N}\sum_{n=1}^{N}\frac{1}{4}g_1(x_n)^2+\frac{1}{4}g_2(x_n)^2-\frac{1}{2}g_1(x_n)g_2(x_n)$$

$$=E_{out}(g)+\frac{1}{N}\sum_{n=1}^{N}\left(\frac{1}{2}g_1(x_n)-\frac{1}{2}g_2(x_n)\right)^2$$

$$=\boldsymbol{E_{out}(g)}+\frac{1}{N}\sum_{n=1}^{N}\frac{1}{4}\big(g_1(x_n)-g_2(x_n)\big)^2$$

Since (real number)$^2 \geq 0$, we know that $\frac{1}{N}\sum_{n=1}^{N}\frac{1}{4}\big(g_1(x_n)-g_2(x_n)\big)^2\geq 0$.

Therefore, $[\textbf{Average of } \boldsymbol{E_{out}(g_1)} \textbf{ and } \boldsymbol{E_{out}(g_2)}]=\boldsymbol{E_{out}(g)}+\frac{1}{N}\sum_{n=1}^{N}\frac{1}{4}\big(g_1(x_n)-g_2(x_n)\big)^2\geq \boldsymbol{E_{out}(g)}$

Thus, the answer is [c].