homework3

Two features are computed to produce the following ordered pairs for three different classes, A, B and C:

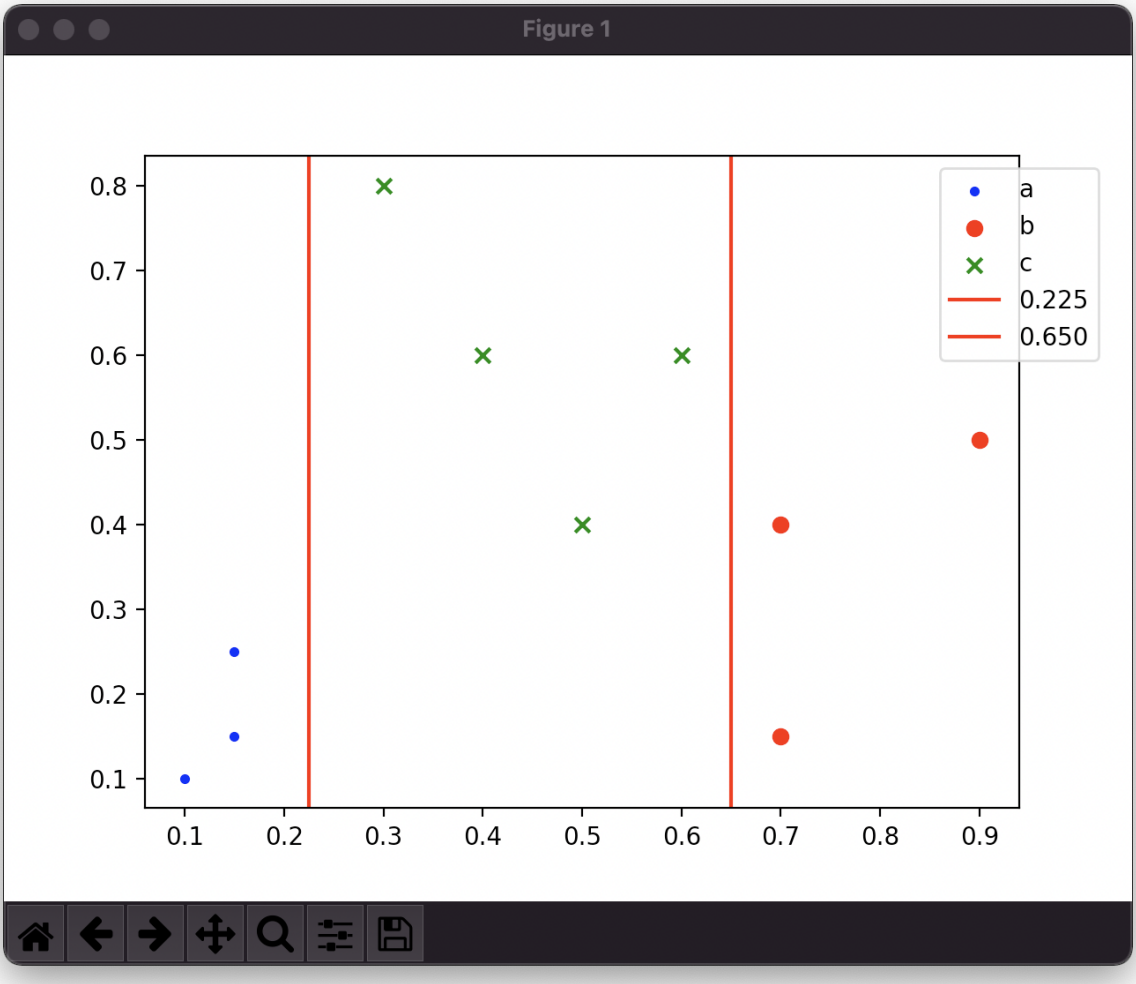
- Class A: `(.1,.1)`, `(.15,.15)`, `(.15,.25)`
- Class B: `(.7,.15)`, `(.7.4)`, `(.9,.5)`
- Class C: `(.3,.8)`, `(.4,.6)`, `(.5,.4)`, `(.6,.6)`

q1

Which of the two features is the better one to use and why? What would be the best decision thresholds to use?

answer q1

```
#! /usr/bin/env python3
import matplotlib.pyplot as plt
a = [(.1,.1),(.15,.15),(.15,.25)]
b = [(.7,.15),(.7,.4),(.9,.5)]
c = [(.3,.8),(.4,.6),(.5,.4),(.6,.6)]
plt.scatter(*zip(*a),marker='.',c='blue',label='a')
plt.scatter(*zip(*b),marker='o',c='red',label='b')
plt.scatter(*zip(*c),marker='x',c='green',label='c')
line1=(.15+.3)/2
line2=(.7+.6)/2
plt.axvline(x = line1, color = 'r', label=f'{line1:.3f}')
plt.axvline(x = line2, color = 'r', label=f'{line2:.3f}')
plt.legend(loc="upper center",bbox_to_anchor=(1, 1), ncol=1)
plt.show()
```



Feature 1 is better because you can linearly separate all 3 classes with that feature.

The decision thresholds would be found by finding the median point between the two closest points in each adjacent class using feature 1.

They would be at around `0.225` and `0.65`.

q2

Now suppose the following additional data points are additionally collected:
A: `(.1,.2)`, `(.3,.2)`;

B: `(.6,.2)`, `(.8,.3)`;

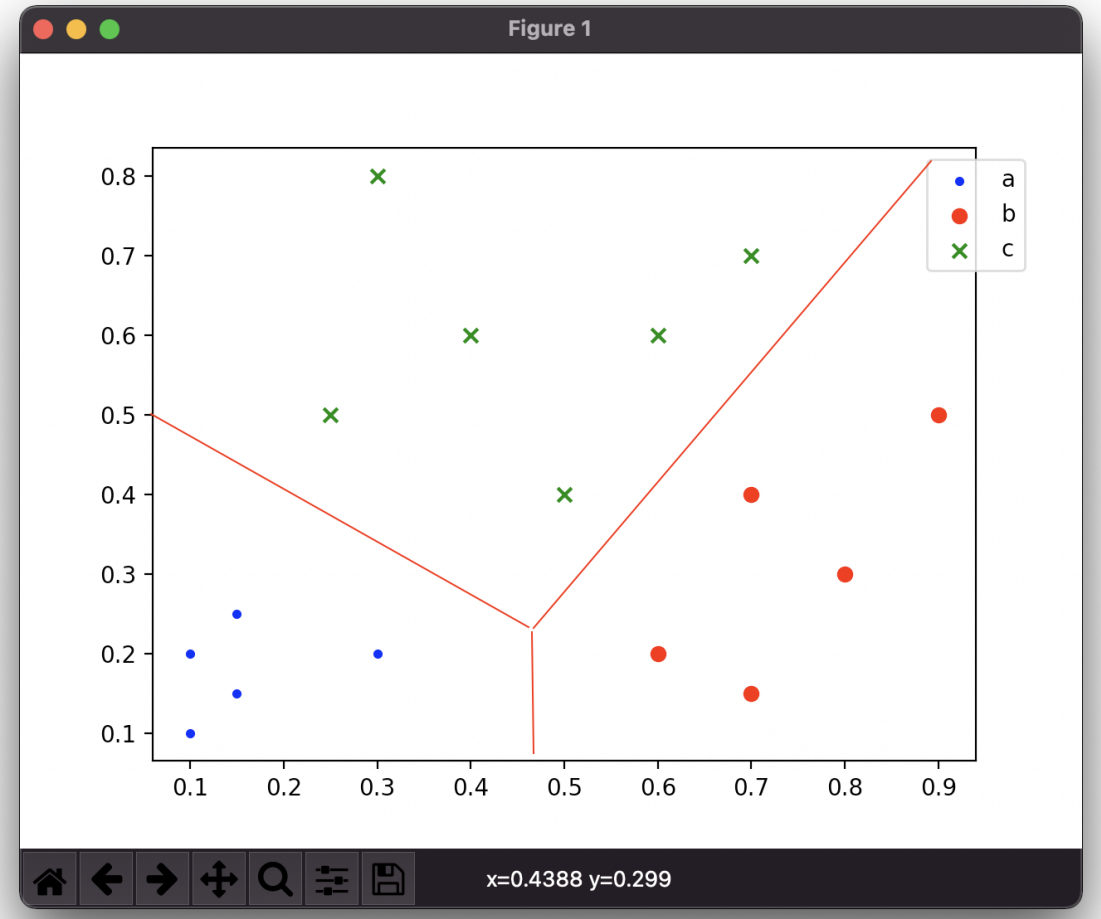
C: `(.25,.5)`, `(.7,.7)`;

Will any single feature separate the classes? Plot the data values and sketch plausible decision regions.

answer q2

No. No single feature can now separate these classes.

```
#! /usr/bin/env python3
import matplotlib.pyplot as plt
a = [(.1,.1),(.15,.15),(.15,.25),(.1,.2),(.3,.2)]
b = [(.7,.15),(.7,.4),(.9,.5),(.6,.2),(.8,.3)]
c = [(.3,.8),(.4,.6),(.5,.4),(.6,.6),(.25,.5),(.7,.7)]
plt.scatter(*zip(*a),marker='.',c='blue',label='a')
plt.scatter(*zip(*b),marker='o',c='red',label='b')
plt.scatter(*zip(*c),marker='x',c='green',label='c')
plt.legend(loc="upper center",bbox_to_anchor=(1, 1), ncol=1)
plt.show()
```



q3

The Lloyd algorithm, a version of the k-means algorithm, is an unsupervised learning method that separates data into clusters. It begins by defining a set of “bins” each consisting of an arbitrarily chosen cluster of points. The centroid of each bin is computed. Then for the next iteration, the squared distance of each data vector from each bin centroid is calculated, and the data vector is moved to the closest such centroid. This constitutes the new set of bins. The algorithm stops when no data vector changes bin membership. Suppose now we start with the following bins:

B1: `x<0.5`, `y<0.45`

B2: `x>0.5`, `y<0.45`

B3: `y>0.45`

Illustrate the operation of the algorithm and comment on whether it converges to the actual separation of the classes.

Notice that the heuristic behind the Lloyd algorithm is that data of the same class will be clustered closer together than to data of a different class, and that Euclidean distance is a good metric to use in the feature space. These assumptions are not necessarily true, and the convergence properties also depend on the initial choice of the set of bins. Having more prior information (e.g., what features separate the classes, typical values for thresholds) provides significant assistance. One can also generalize the algorithm to deal with situations where we do not know how many classes there are; we run the algorithm with different numbers of bins, and apply statistical measures of how tightly clustered the results are, penalizing solutions with more classes.

answer q3

naiieve implementation to walk through the math

```
#! /usr/bin/env python3
'''
0. Select k. (in this case k = 3).
1. Initialize k groups (e.g. random, or in this case a structured
initialization).
2. Compute the centroid for each group (k centroids).
3. For each point, compute the distance to each of the k centroids. Assign the
group of the point to the minimum distance centroid.
4. If the groups have changed membership, repeat step 2-3 until no points change
group membership.
'''

from math import sqrt
def dist(p1,p2):
    return sqrt((p1[1]-p2[1])**2+(p1[0]-p2[0])**2)

def centroid(l):
    x_tot = 0
    y_tot = 0
    for x,y in l:
        x_tot += x
        y_tot += y
    return (x_tot/len(l), y_tot/len(l))

def centroids(l1,l2,l3):
    return list(map(centroid,[l1,l2,l3]))

def iterate(points,cb1,cb2,cb3):
    nb1 = []
    nb2 = []
    nb3 = []
    for point in points:
        curMin = 10000000
        center = "unclassified"
        d1 = dist(point,cb1)
        d2 = dist(point,cb2)
        d3 = dist(point,cb3)
        if d1 < curMin:
            curMin = d1
            center = 'cb1'
        if d2 < curMin:
            curMin = d2
            center = 'cb2'
        if d3 < curMin:
            curMin = d3
            center = 'cb3'
        if center == 'cb1':
            nb1.append(point)
        elif center == 'cb2':
            nb2.append(point)
        elif center == 'cb3':
            nb3.append(point)
        else:
            exit(1)
    return nb1,nb2,nb3

points = [(.1,.1),(.15,.15),(.15,.25),(.1,.2),(.3,.2),(.7,.15),(.7,.4),(.9,.5),(.6,.2),(.8,.3),(.3,.8),(.4,.6),(.5,.4),(.6,.6),(.25,.5),(.7
```

```
b1 = []
b2 = []
b3 = []
unclassified = []

for x,y in points:
    if x < 0.5 and y < 0.45:
        b1.append((x,y))
    elif x >= 0.5 and y < 0.45:
        b2.append((x,y))
    elif y > 0.45:
        b3.append((x,y))
    else:
        unclassified.append((x,y))

print('b1:', sorted(b1))
print('b2:', sorted(b2))
print('b3:', sorted(b3))
print('unclassified:', unclassified)
cb1, cb2, cb3 = centroids(b1,b2,b3)
print('b1',cb1)
print('b2',cb2)
print('b3',cb3)

print()
print('iteration1')#iteration 1
b1_2, b2_2, b3_2 = iterate(points,cb1,cb2,cb3)
print('b1_2:',sorted(b1_2))
print('b2_2:',sorted(b2_2))
print('b3_2:',sorted(b3_2))
cb1_2,cb2_2,cb3_2 = centroids(b1_2,b2_2,b3_2)
print('centroids')
print('b1_2',cb1_2)
print('b2_2',cb2_2)
print('b3_2',cb3_2)

print()
print('iteration2')#iteration 2
b1_3, b2_3, b3_3 = iterate(points,cb1_2,cb2_2,cb3_2)
print('b1_3:',sorted(b1_3))
print('b2_3:',sorted(b2_3))
print('b3_3:',sorted(b3_3))
cb1_3, cb2_3, cb3_3 = centroids(b1_3,b2_3,b3_3)
print('centroids')
print('b1_2',cb1_2)
print('b2_2',cb2_2)
print('b3_2',cb3_2)
```

```
b1: [(0.1, 0.1), (0.1, 0.2), (0.15, 0.15), (0.15, 0.25), (0.3, 0.2)]
b2: [(0.5, 0.4), (0.6, 0.2), (0.7, 0.15), (0.7, 0.4), (0.8, 0.3)]
b3: [(0.25, 0.5), (0.3, 0.8), (0.4, 0.6), (0.6, 0.6), (0.7, 0.7), (0.9, 0.5)]
unclassified: []
b1 (0.16, 0.18)
b2 (0.6599999999999999, 0.29000000000000004)
b3 (0.525, 0.6166666666666667)

iteration1
b1_2: [(0.1, 0.1), (0.1, 0.2), (0.15, 0.15), (0.15, 0.25), (0.3, 0.2)]
b2_2: [(0.5, 0.4), (0.6, 0.2), (0.7, 0.15), (0.7, 0.4), (0.8, 0.3), (0.9, 0.5)]
b3_2: [(0.25, 0.5), (0.3, 0.8), (0.4, 0.6), (0.6, 0.6), (0.7, 0.7)]
centroids
b1_2 (0.16, 0.18)
b2_2 (0.7000000000000001, 0.325)
b3_2 (0.45, 0.64)

iteration2
b1_3: [(0.1, 0.1), (0.1, 0.2), (0.15, 0.15), (0.15, 0.25), (0.3, 0.2)]
b2_3: [(0.5, 0.4), (0.6, 0.2), (0.7, 0.15), (0.7, 0.4), (0.8, 0.3), (0.9, 0.5)]
b3_3: [(0.25, 0.5), (0.3, 0.8), (0.4, 0.6), (0.6, 0.6), (0.7, 0.7)]
centroids
b1_2 (0.16, 0.18)
b2_2 (0.7000000000000001, 0.325)
b3_2 (0.45, 0.64)
```

We can see that after 2 iterations the centroids did not move so we stop.

We did not reach the actual classification because `(0.5,0.4)` is incorrectly classified.