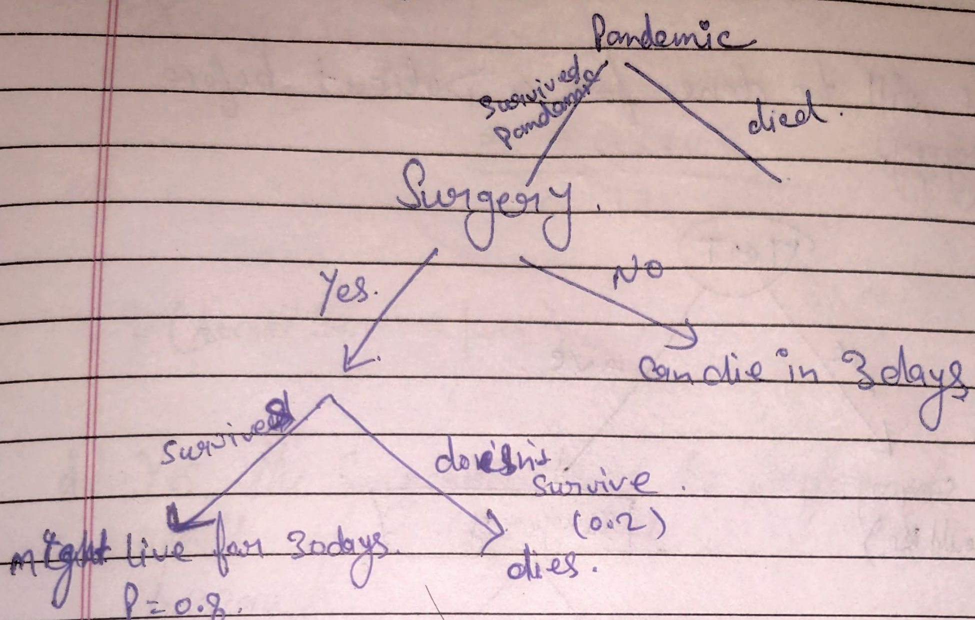
Assignment - 2

Sumit kumar.



$P(\text{Survive after surgery}) = 0.8$

$P(\text{not Survive after Surgery}) = 1-0.8 = 0.2$

b) living function. $L(x)$

$L(30) = 1.$ $\qquad L(x) = mx.$

$m = \frac{1}{30}$

$L(x) = \frac{x}{30}$

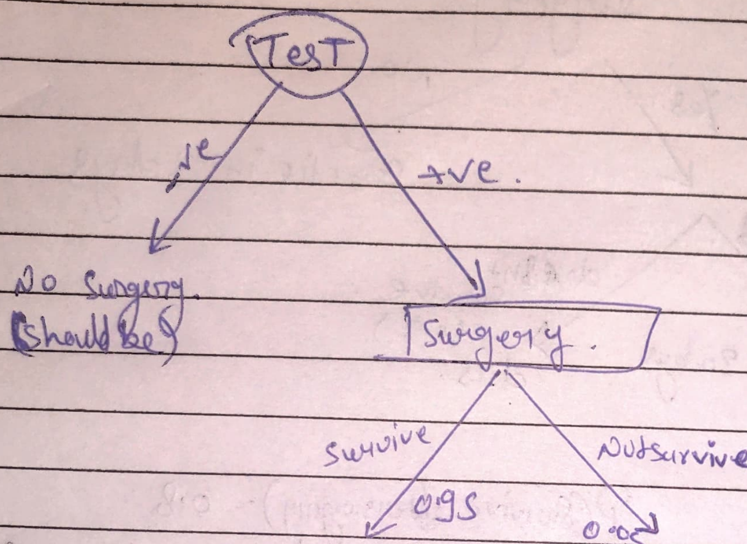
$L(30) = \frac{30}{30} = 1$

$L(0) = \frac{0}{30} = 0$

$L(3) = ?$ $\qquad L(3) = \frac{3}{30} = \boxed{0.1}$

c) A.T.Q.

Test will be done for a patient before Surgery.



~~P(+ve)~~

P(Survive Surgery given +ve) = 0.95.

~~P(not survive surgery given +ve)~~ =

~~P(Survive Surgery given -ve) = 0.05~~ (1-0.95)

P(not survive given positive) = 0.05

find.

$$P(\text{Survive} \mid +ve) = \frac{P(+ve \mid \text{survive S})\, P(\text{Survive Surgery})}{P(\cancel{pos}\ +ve)}$$

$P(+ve) = P(+ve \mid \text{survive}) + P(+ve \mid \text{not survive})$

$= 0.95 \times 0.8 + 0.05 \times 0.2$

$= 0.77$

~~P(+~~ $P(-ve) = 0.23$

~~P(survive|+ve~~

P(Survive Surgery | +ve)

$$= \frac{0.95 \times 0.8}{0.77} = \frac{0.76}{0.77}$$

$$= 0.987$$

~~P(doesn't survive | +ve)~~ =

d) Yes, the Surgery should be performed if the result of the test is positive because.

The true positive rate is high means the outcome of test is true given some Cases.

The false +ve Rate is too low.

⇒ The Surviving of the person is highly likely.

Contingency table.

TP FN = 1

FP TN = 1

1 1

0.95	0.05
0.05	0.95

$$P(\text{Survives} \mid -ve) = \frac{P(-ve \mid \text{Survives})\, P(\text{Survives})}{P(-ve)}$$

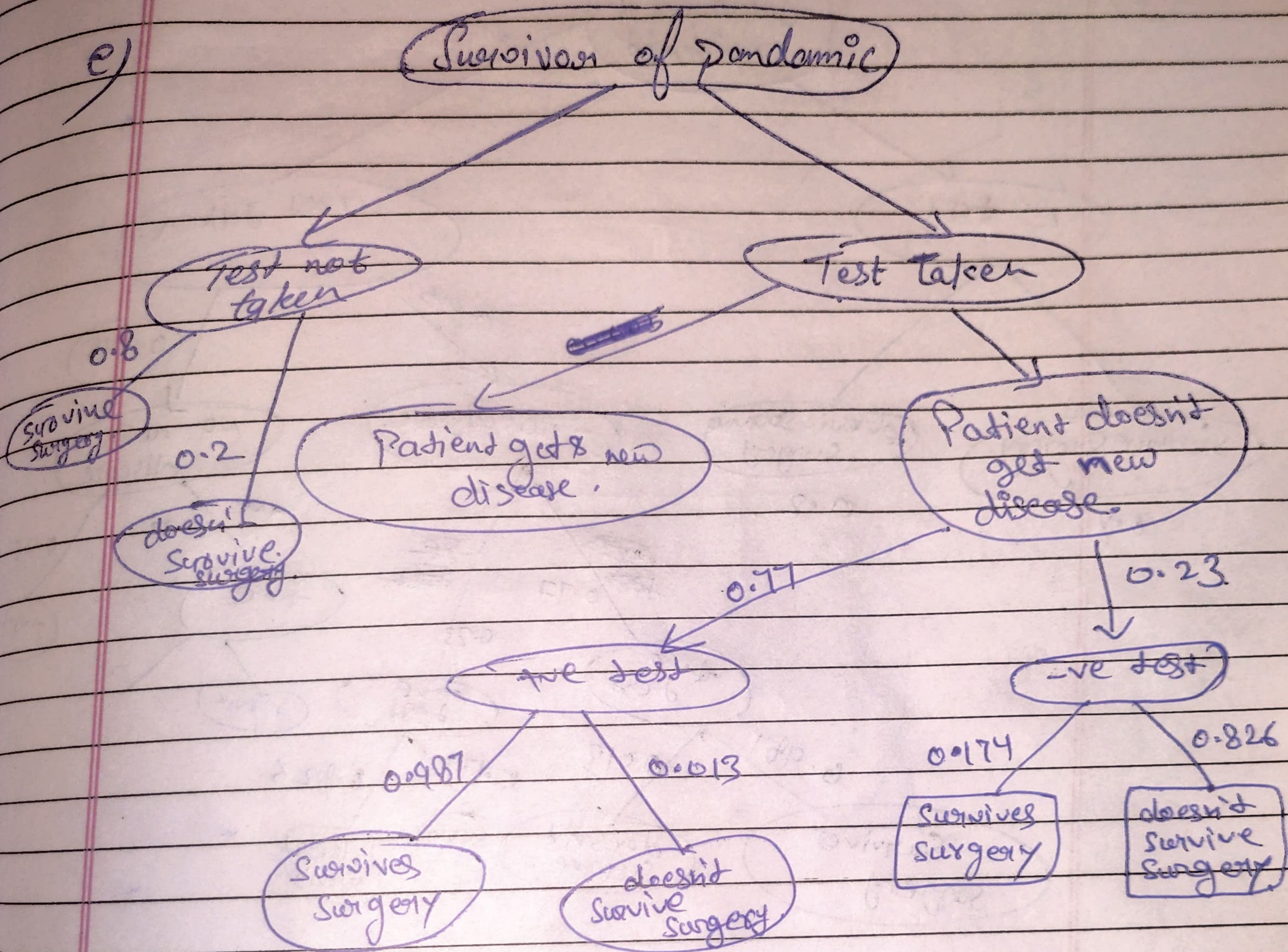
$$= \frac{0.05 \times 0.8}{0.23}$$

$$= 0.1739 \approx 0.174.$$

$$P(\text{not survive} \mid -ve) = 1 - 0.174$$

$$= 0.826$$

e)



f) Probablity of new disease on test. = 0.005

should take the test or not.

P[survive with no disease] $= 0.987 \times 0.77 \times 0.995 + 0.0174 \times 0.23 \times 0.995$

$= 0.796 < 0.8$

P[survive with no new disease after test taken] < P[Survive without test]

⇒ Person should not take the test.

In [ ]:

```
from utils import Dataset ,perceptron
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

In [ ]:

```
def plot1(inputs,weights):
    sns.scatterplot(data=inputs,x="X",y="Y",hue='Labels',s=2).set(title='plot with deci
    inputs=np.array(inputs)
    Ya_nn=[]
    ia_nn=[]
    for i in np.linspace(np.amin(inputs[:,:1]),np.amax(inputs[:,:1])):

            slope = -(weights[1])/(weights[2])
            intercept = -(weights[0]/weights[2])
            #y =mx+c, m is slope and c is intercept
            y = (slope*i) + intercept
            ia_nn.append(i)
            Ya_nn.append(y)
    sns.lineplot(x=ia_nn,y=Ya_nn,color='black',markersize=4)
    plt.legend()
    plt.show()
```

In [ ]:

```
def plot2(inputs,weights):
    print('All the points on/above the decision boundary belongs to class 1')
    sns.scatterplot(data=inputs,x="x1",y="x2",hue='out').set(title='plot with decision
    inputs=np.array(inputs)
    Ya_nn=[]
    ia_nn=[]
    for i in np.linspace(np.amin(inputs[:,:1]),np.amax(inputs[:,:1])):
            slope = 0
            intercept =0
            if(weights[2]!=0):
                slope = -(weights[1])/(weights[2])
            if(weights[2]!=0):
                intercept = -(weights[0]/weights[2])
            #y =mx+c, m is slope and c is intercept
            y = (slope*i) + intercept
            ia_nn.append(i)
            Ya_nn.append(y)
    sns.lineplot(x=ia_nn,y=Ya_nn,color='black',markersize=4)
    plt.legend()
    plt.show()
```

Part 1

In [ ]:

```
#without noise 10,000 samples
d=Dataset(10_000)
df=d.get()
df
```

Out[ ]:

	X	Y	Labels

	X	Y	Labels
0	0.898230	-0.439526	0
1	0.723509	2.309685	1
2	-0.822943	-0.568124	0
3	-0.281687	3.959506	1
4	0.281912	0.959440	0
...	...	...	...
9995	0.867456	2.502486	1
9996	0.992129	3.125221	1
9997	-0.242525	0.970145	0
9998	0.135728	2.009254	1
9999	0.860023	2.489745	1

10000 rows × 3 columns

In [ ]:

```
# d=Dataset(10000) with noise added
df1=d.get(True)
df1
```

Out[ ]:

	X	Y	Labels
0	0.561958	2.377508	1
1	-0.524846	0.868321	0
2	0.392301	0.718524	0
3	-0.027525	-1.074236	0
4	-0.127290	4.008247	1
...	...	...	...
9995	-0.457150	2.153987	1
9996	0.682129	3.870551	1
9997	-0.304008	-0.910514	0
9998	0.723689	-0.757239	0
9999	-0.120075	-0.982539	0

10000 rows × 3 columns

In [ ]:

```
df.shape
```

Out[ ]:

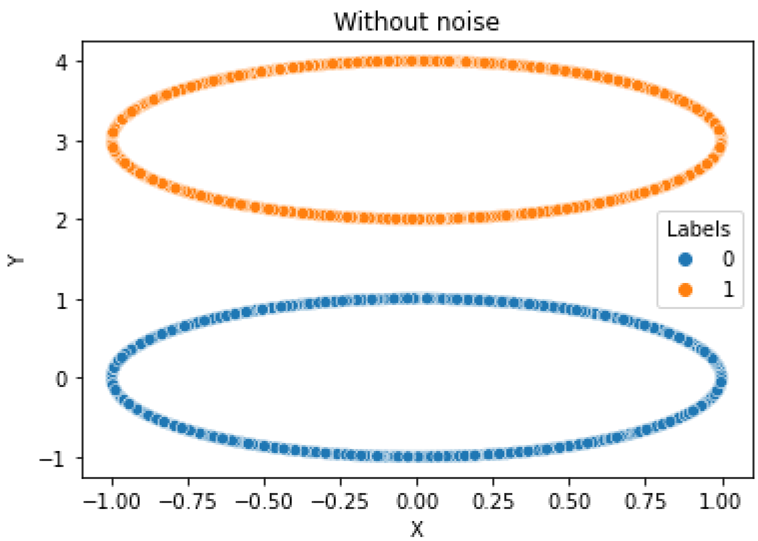
```
(10000, 3)
```

In [ ]:
```
df1.shape
```

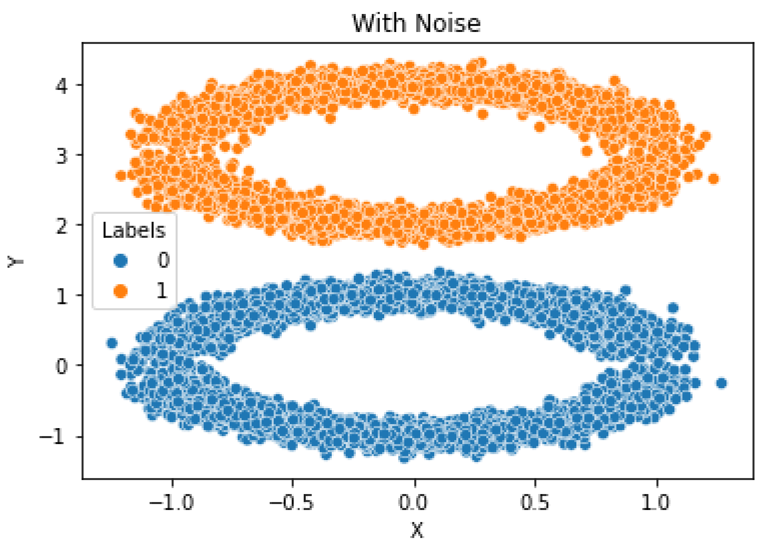
Out[ ]:
```
(10000, 3)
```

Part 2

In [ ]:
```
sns.scatterplot(data=df,x="X",y="Y",hue='Labels').set(title='Without noise')
plt.show()
```



In [ ]:
```
sns.scatterplot(data=df1,x="X",y="Y",hue='Labels').set(title='With Noise')
plt.show()
```



Part 3

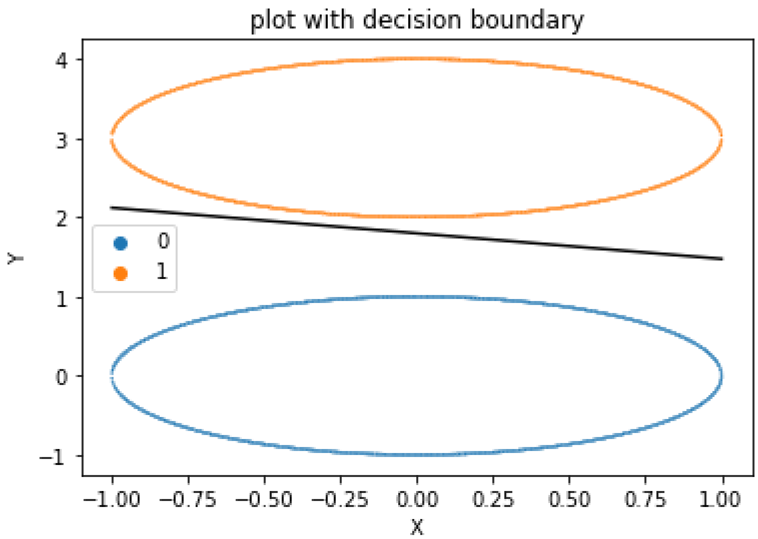
In [ ]:
```
p=perceptron()
# trainning the perceptron with learnable bias on without noise data
p.train_weights(np.array(df),5,1)
```

```
>epoch=0, error=6.000 [-4.0, 0.720376711074459, 2.228126731090508]
```

```
>epoch=1, error=0.000 [-4.0, 0.720376711074459, 2.228126731090508]
>epoch=2, error=0.000 [-4.0, 0.720376711074459, 2.228126731090508]
>epoch=3, error=0.000 [-4.0, 0.720376711074459, 2.228126731090508]
>epoch=4, error=0.000 [-4.0, 0.720376711074459, 2.228126731090508]
```

In [ ]:

```
plot1(df,p.weights)
```



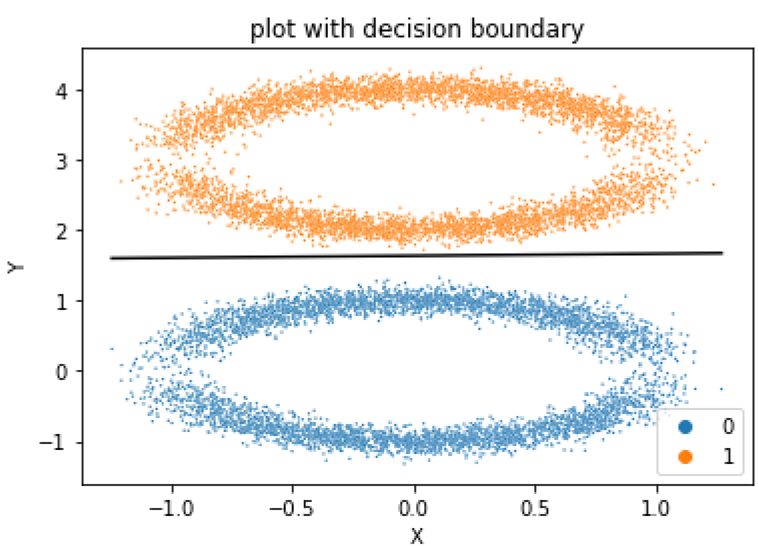
In [ ]:

```
p1=perceptron()
# the third parameter is a flag to make bias 0 or learnable
p1.train_weights(np.array(df1),5,1)
```

```
>epoch=0, error=13.000 [-5.0, -0.08963611587720852, 3.0705765156008695]
>epoch=1, error=0.000 [-5.0, -0.08963611587720852, 3.0705765156008695]
>epoch=2, error=0.000 [-5.0, -0.08963611587720852, 3.0705765156008695]
>epoch=3, error=0.000 [-5.0, -0.08963611587720852, 3.0705765156008695]
>epoch=4, error=0.000 [-5.0, -0.08963611587720852, 3.0705765156008695]
```

In [ ]:

```
plot1(df1,p1.weights)
```



The Decision boundary for both the datasets with and without noise exist because the data was linearly separable even after adding noise.

Part 4

In [ ]:

```
#classifier with bias set to constant 0 and data used = without noise data
p3=perceptron()
p3.train_weights(np.array(df),100,0)
```

```
>epoch=0, error=2966.000 [0.0, 0.7298420020772367, 0.357438305509001]
>epoch=1, error=2960.000 [0.0, 0.5832358101376358, 0.35825602366765097]
>epoch=2, error=2933.000 [0.0, 0.2057844099432271, 0.7686138975407416]
>epoch=3, error=2960.000 [0.0, 0.16148262079835995, 0.729842004188166]
>epoch=4, error=2969.000 [0.0, 0.5219670291163601, 0.33676161506953906]
>epoch=5, error=2986.000 [0.0, 0.4717218654541335, 0.704404884716293]
>epoch=6, error=2980.000 [0.0, 1.079825032968965, 0.9073725933009041]
>epoch=7, error=2976.000 [0.0, 0.2209896263871578, 0.7191236710494185]
>epoch=8, error=2969.000 [0.0, 0.43954112055514294, 0.3400822539368865]
>epoch=9, error=2962.000 [0.0, 1.015208100299604, 1.8071932011940732]
>epoch=10, error=3019.000 [0.0, 0.27536465020939405, 0.7640282261868007]
>epoch=11, error=2982.000 [0.0, 1.0515376689089853, 0.8808230634647499]
>epoch=12, error=2973.000 [0.0, 0.5526712300032313, 0.3441118716194903]
>epoch=13, error=2976.000 [0.0, 1.0479557213482988, 1.7845564166478538]
>epoch=14, error=2954.000 [0.0, 0.8895651222267846, 0.6777836946090053]
>epoch=15, error=2981.000 [0.0, 1.0238962330598933, 0.9260682819063845]
>epoch=16, error=2943.000 [0.0, 0.413556559985341, 0.27462716404239196]
>epoch=17, error=2962.000 [0.0, 0.8908658796434203, 0.7124451415946332]
>epoch=18, error=2931.000 [0.0, 0.23115779673824832, 0.740252831825445]
>epoch=19, error=2945.000 [0.0, 0.2235790460408258, 0.7242992172946875]
>epoch=20, error=2986.000 [0.0, 0.5234628793449905, 0.3871186492613027]
>epoch=21, error=2946.000 [0.0, 1.2920249004415536, 0.05901453359430808]
>epoch=22, error=2982.000 [0.0, 1.2204751962876135, 1.8716760371991037]
>epoch=23, error=2990.000 [0.0, 0.22422237420943558, 0.7234297371584014]
>epoch=24, error=2986.000 [0.0, 0.5241062075136003, 0.3862491691250166]
>epoch=25, error=2970.000 [0.0, 0.44639381683861723, 0.678922103955324]
>epoch=26, error=2970.000 [0.0, 0.26706168043483247, 0.7502170105397071]
>epoch=27, error=2977.000 [0.0, 0.21916295331120894, 0.6911493922734425]
>epoch=28, error=2930.000 [0.0, 0.21757398061945143, 0.7685420138586131]
>epoch=29, error=2968.000 [0.0, 1.0142076828134532, 1.8075247419397082]
>epoch=30, error=3019.000 [0.0, 0.2743642327232432, 0.7643597669324357]
>epoch=31, error=2952.000 [0.0, 1.1771151204575394, 1.8575551953991236]
>epoch=32, error=2946.000 [0.0, 0.36000097747622983, 0.29509140122480215]
>epoch=33, error=2964.000 [0.0, 0.9634727157056591, 0.09376357206955566]
>epoch=34, error=2978.000 [0.0, 0.8286671960717851, 0.17091159142503753]
>epoch=35, error=2978.000 [0.0, 1.0807771201850676, 0.9121824437372951]
>epoch=36, error=2969.000 [0.0, 0.41333223013522646, 0.2639591861289309]
>epoch=37, error=2937.000 [0.0, 1.0290007708943756, 1.8365090267355781]
>epoch=38, error=2969.000 [0.0, 0.4234426902239492, 0.2579431579500425]
>epoch=39, error=2936.000 [0.0, 1.0984844252118817, 0.9248185494052407]
>epoch=40, error=2938.000 [0.0, 0.40318690552436887, 1.015195217376354]
>epoch=41, error=2961.000 [0.0, 0.2249056046032074, 0.7175465413194148]
>epoch=42, error=2966.000 [0.0, 0.48820509362789766, 0.3726391945716837]
>epoch=43, error=2983.000 [0.0, 1.070524616034751, 0.14059532518057638]
>epoch=44, error=2992.000 [0.0, 1.1933755799043713, 1.8207417640622467]
>epoch=45, error=2980.000 [0.0, 0.8353787811192361, 0.6998728606801979]
>epoch=46, error=2957.000 [0.0, 0.5709799817256336, 0.36623613088986684]
>epoch=47, error=2934.000 [0.0, 0.7853815210235524, 0.11009130939192546]
>epoch=48, error=2984.000 [0.0, 0.7710810197975861, 0.6625578906055828]
>epoch=49, error=2983.000 [0.0, 1.0559619710752766, 1.7878318840814518]
>epoch=50, error=2985.000 [0.0, 0.38474941336508084, 0.3022871081049666]
>epoch=51, error=2949.000 [0.0, 1.306959398099452, 0.3311040116575696]
>epoch=52, error=2942.000 [0.0, 0.38228576669027503, 0.3017813355892739]
```

```
>epoch=53, error=2971.000 [0.0, 1.013526276108336, 0.01271917072614348]
>epoch=54, error=2973.000 [0.0, 0.6237100943362186, 0.3351875401664276]
>epoch=55, error=2959.000 [0.0, 0.2074676747499653, 0.7705033767694777]
>epoch=56, error=2955.000 [0.0, 0.8055852329601458, 0.6899159179378979]
>epoch=57, error=2949.000 [0.0, 0.45229981165421385, 0.71146885373078]
>epoch=58, error=2977.000 [0.0, 0.14208576312262844, 0.704971490343748]
>epoch=59, error=2965.000 [0.0, 0.4461986841854164, 0.9940558255217292]
>epoch=60, error=2939.000 [0.0, 0.5633765427537001, 0.35474479368297207]
>epoch=61, error=2987.000 [0.0, 0.5119312457203402, 0.394646613895442]
>epoch=62, error=2970.000 [0.0, 0.534745362493044, 0.3779592823018635]
>epoch=63, error=2947.000 [0.0, 1.0273500674541456, 0.9786792570226949]
>epoch=64, error=2951.000 [0.0, 1.0892495592794464, 0.922387617734958]
>epoch=65, error=2960.000 [0.0, 0.2162171657523444, 0.7701594051693813]
>epoch=66, error=2974.000 [0.0, 1.2346106013652909, 1.861068228691483]
>epoch=67, error=2987.000 [0.0, 0.5234085199478451, 0.38316788394450196]
>epoch=68, error=2947.000 [0.0, 0.5312481918168195, 0.3781791364588837]
>epoch=69, error=2957.000 [0.0, 0.18670376451299986, 0.715567940751555]
>epoch=70, error=2991.000 [0.0, 1.0692349887912083, 1.8215128855644127]
>epoch=71, error=2977.000 [0.0, 0.5484383132575446, 0.36547457444020026]
>epoch=72, error=2963.000 [0.0, 0.5307174625607014, 0.37668383309745856]
>epoch=73, error=2929.000 [0.0, 1.0888987756027788, 0.914777160787471]
>epoch=74, error=2959.000 [0.0, 0.4892079182042428, 0.3577475901042183]
>epoch=75, error=2980.000 [0.0, 0.3516770827282829, 0.7137366194037883]
>epoch=76, error=2952.000 [0.0, 0.5167642367549385, 0.32442246994427004]
>epoch=77, error=2995.000 [0.0, 0.3905955395355807, 0.30001945257603435]
>epoch=78, error=2963.000 [0.0, 1.2341992048468862, 0.21152734632781878]
>epoch=79, error=2978.000 [0.0, 1.357478140685172, 0.3222387287233023]
>epoch=80, error=2992.000 [0.0, 0.5982498622677015, 0.36817567160504894]
>epoch=81, error=2968.000 [0.0, 0.8630263116400407, 0.6791626659175686]
>epoch=82, error=2946.000 [0.0, 0.21564619698009468, 0.7665582619873074]
>epoch=83, error=2976.000 [0.0, 0.2254605698586607, 0.722054598538863]
>epoch=84, error=2929.000 [0.0, 0.43899393925054686, 1.0149527097663333]
>epoch=85, error=2961.000 [0.0, 1.094251549779135, 0.9230244344393049]
>epoch=86, error=2949.000 [0.0, -0.2327600096165423, 0.33110459579379425]
>epoch=87, error=2990.000 [0.0, 0.5270926753150253, 0.3836383258418171]
>epoch=88, error=2954.000 [0.0, 0.3945932807696495, 0.30412907566079117]
>epoch=89, error=2955.000 [0.0, 1.6046741234935753, 1.3940772544985056]
>epoch=90, error=2942.000 [0.0, 1.0546074237744407, 1.8206648641931085]
>epoch=91, error=2938.000 [0.0, 1.0717646100135783, 0.9109388556224572]
>epoch=92, error=2995.000 [0.0, 1.037873243028897, 1.8721538680239593]
>epoch=93, error=2976.000 [0.0, 0.45747633718115455, 0.9993320340648483]
>epoch=94, error=2932.000 [0.0, 1.0259644472285208, 1.800666461210835]
>epoch=95, error=2927.000 [0.0, 0.4048775487847811, 0.7628896522114547]
>epoch=96, error=2933.000 [0.0, 0.5754382951548034, 0.3551700715093872]
>epoch=97, error=2969.000 [0.0, 0.5103494190500921, 0.3519204291583412]
>epoch=98, error=2993.000 [0.0, 0.5474508945226755, 0.3843436913677226]
>epoch=99, error=2971.000 [0.0, 0.5483186593984457, 0.3843399300705964]
```

The decision boundary for this model won't exist because when we made the bias 0 we are forcing the decision boundry to pass through the origin and also separate the data , but the point (0,0) is the center for the circle having k=0 (label = 0) which makes the weights oscillate to the error value no matter the number of epochs.which results in a non existent decision boundary which passes through (0,0)

However this is not the case when we allow the bias to be learnable (can update with error) it can shift with the weights accordingly to finally converge to a proper decision boundary.

When bias was learnable it took only 2 epochs to find the correct weights but when we fix the bias the error is not reducing even after 100 epochs hence the perceptron will not converge if the bias is set to 0 for the given dataset

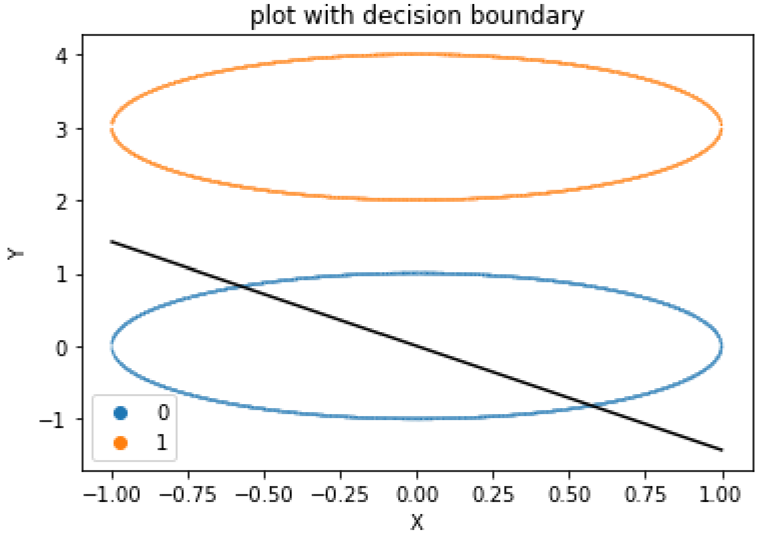
The plots can be seen below for both the cases.

In [ ]:

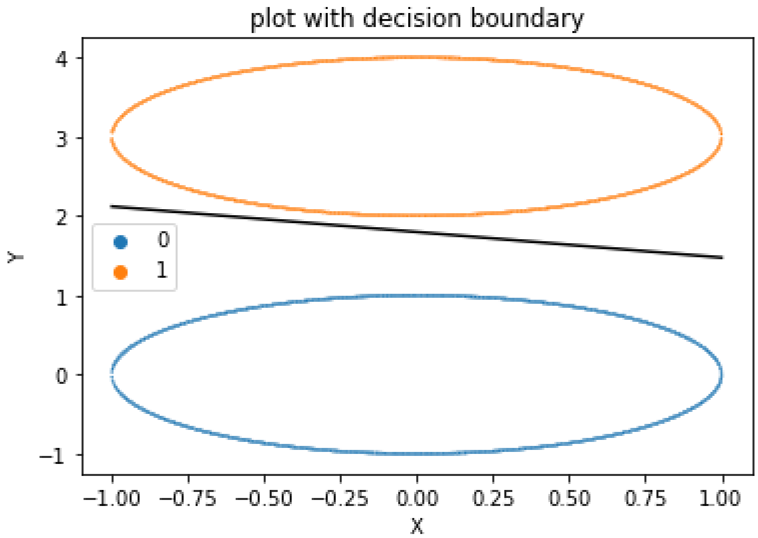
```
print('Forcing the boundary line to pass through origin')
```

```
plot1(df,p3.weights)
print('Bias can be adjusted with no restriction')
plot1(df,p.weights)
```

```
Forcing the boundary line to pass through origin
```




```
Bias can be adjusted with no restriction
```



Part 5

In [ ]:

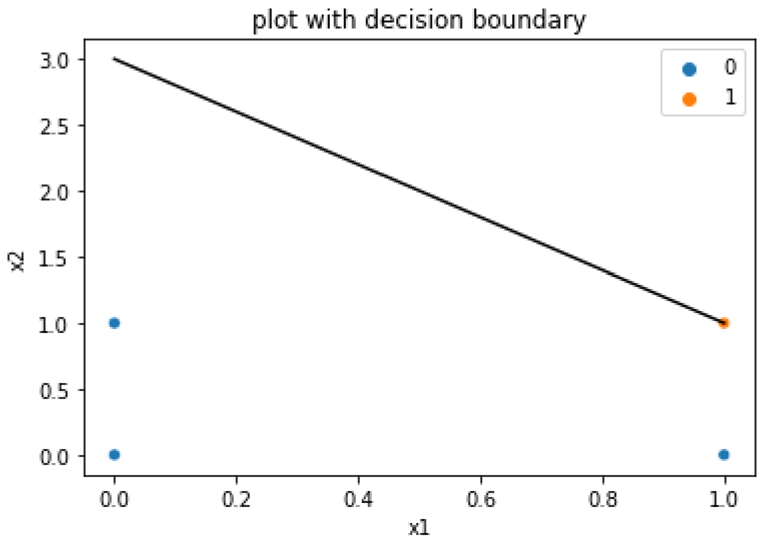
```
xor_d={'x1':[0,0,1,1],'x2':[0,1,0,1],'out':[0,1,1,0]}
and_d={'x1':[0,0,1,1],'x2':[0,1,0,1],'out':[0,0,0,1]}
or_d={'x1':[0,0,1,1],'x2':[0,1,0,1],'out':[0,1,1,1]}
```

In [ ]:

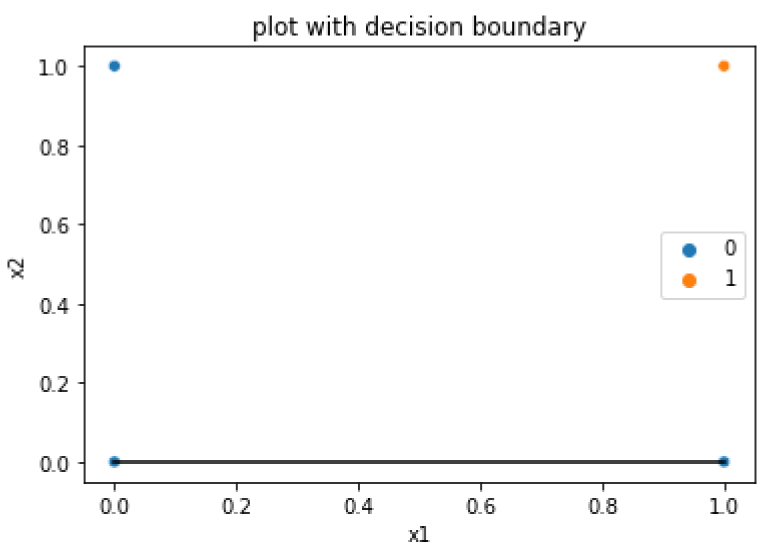
```
and_df=pd.DataFrame(and_d)
print(and_df)
p_and=perceptron()
p_and.train_weights(np.array(and_df),6,1)
print('With learnable bias')
plot2(and_df,p_and.weights)

p_and1=perceptron()
p_and1.train_weights(np.array(and_df),6,0)
```

```
print('With 0 bias')
plot2(and_df,p_and1.weights)
```

```
   x1  x2  out
0   0   0    0
1   0   1    0
2   1   0    0
3   1   1    1
>epoch=0, error=3.000 [-1.0, 2.0, 1.0]
>epoch=1, error=3.000 [-2.0, 2.0, 1.0]
>epoch=2, error=2.000 [-2.0, 2.0, 2.0]
>epoch=3, error=1.000 [-3.0, 2.0, 1.0]
>epoch=4, error=0.000 [-3.0, 2.0, 1.0]
>epoch=5, error=0.000 [-3.0, 2.0, 1.0]
With learnable bias
All the points on/above the decision boundary belongs to class 1
```

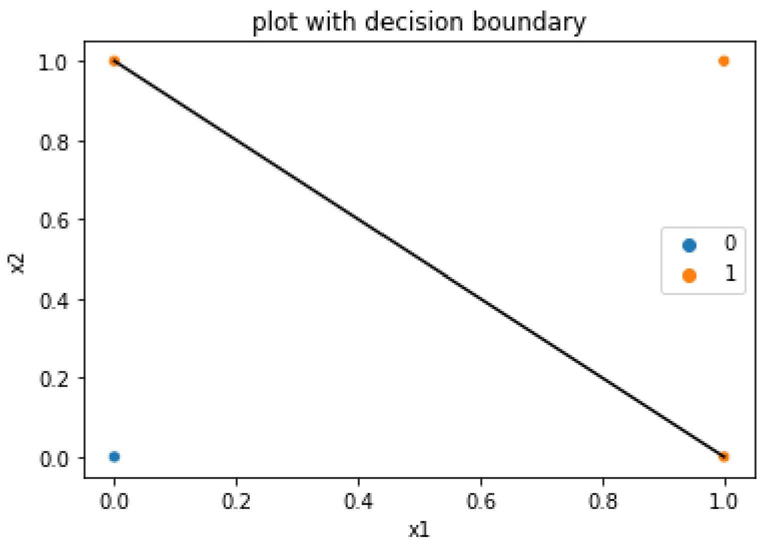



```
>epoch=0, error=3.000 [0.0, 0.0, 0.0]
>epoch=1, error=4.000 [0.0, 0.0, 0.0]
>epoch=2, error=4.000 [0.0, 0.0, 0.0]
>epoch=3, error=4.000 [0.0, 0.0, 0.0]
>epoch=4, error=4.000 [0.0, 0.0, 0.0]
>epoch=5, error=4.000 [0.0, 0.0, 0.0]
With 0 bias
All the points on/above the decision boundary belongs to class 1
```

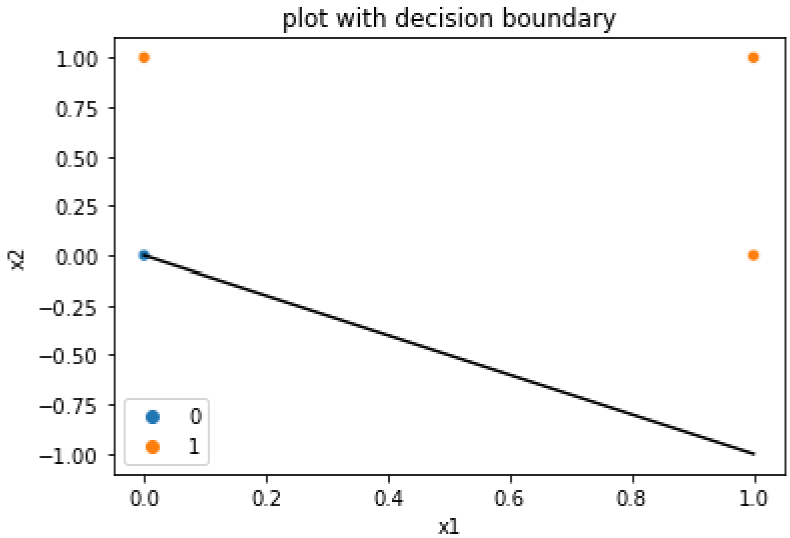


In [ ]:
```
or_df=pd.DataFrame(or_d)
print(or_df)
p_or=perceptron()
#leanable bias
p_or.train_weights(np.array(or_df),6,1)
plot2(or_df,p_or.weights)
p_or1=perceptron()
p_or1.train_weights(np.array(or_df),6,0)
plot2(or_df,p_or1.weights)
```

```
   x1  x2  out
0   0   0    0
1   0   1    1
2   1   0    1
3   1   1    1
>epoch=0, error=1.000 [-1.0, 1.0, 1.0]
>epoch=1, error=0.000 [-1.0, 1.0, 1.0]
>epoch=2, error=0.000 [-1.0, 1.0, 1.0]
>epoch=3, error=0.000 [-1.0, 1.0, 1.0]
>epoch=4, error=0.000 [-1.0, 1.0, 1.0]
>epoch=5, error=0.000 [-1.0, 1.0, 1.0]
All the points on/above the decision boundary belongs to class 1
```



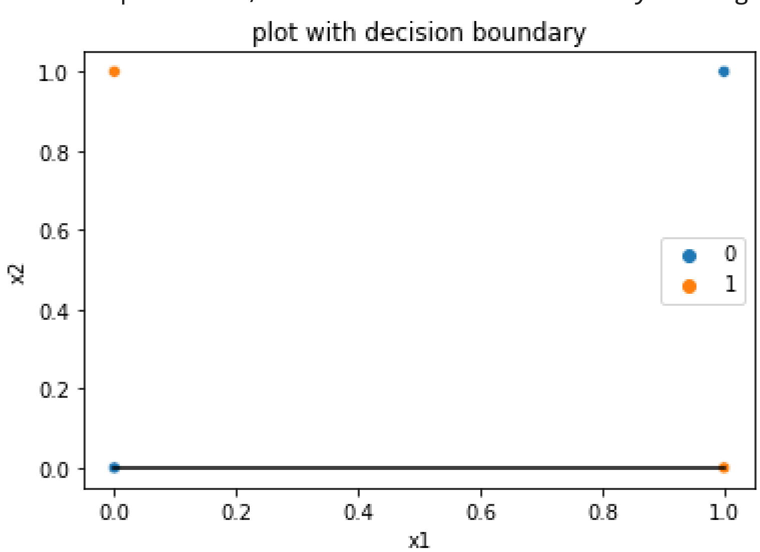

```
>epoch=0, error=1.000 [0.0, 1.0, 1.0]
>epoch=1, error=1.000 [0.0, 1.0, 1.0]
>epoch=2, error=1.000 [0.0, 1.0, 1.0]
>epoch=3, error=1.000 [0.0, 1.0, 1.0]
>epoch=4, error=1.000 [0.0, 1.0, 1.0]
>epoch=5, error=1.000 [0.0, 1.0, 1.0]
All the points on/above the decision boundary belongs to class 1
```



In [ ]:

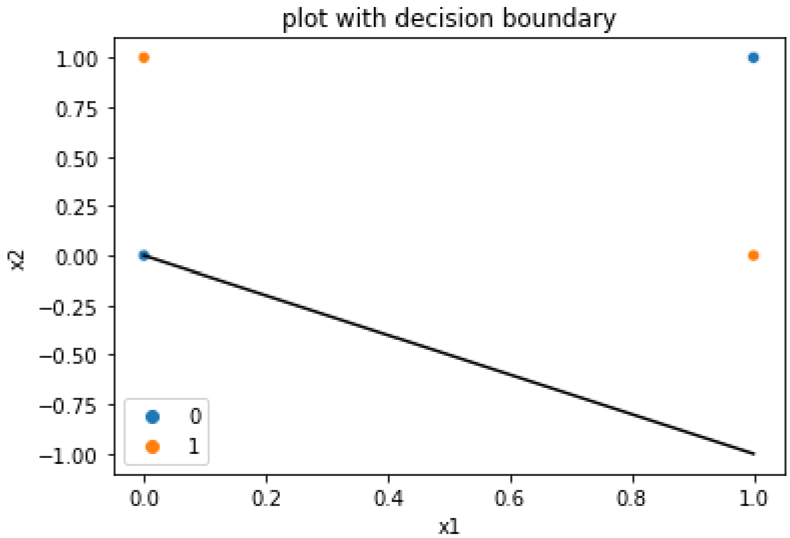
```
xor_df=pd.DataFrame(xor_d)
print(xor_df)
p_xor=perceptron()
p_xor.train_weights(np.array(xor_df),5,1)
plot2(xor_df,p_xor.weights)
p_xor1=perceptron()
p_xor1.train_weights(np.array(xor_df),5,0)
plot2(xor_df,p_xor1.weights)
```

```
   x1  x2  out
0   0   0    0
1   0   1    1
2   1   0    1
3   1   1    0
>epoch=0, error=2.000 [-2.0, 0.0, 0.0]
>epoch=1, error=3.000 [-1.0, 0.0, 0.0]
>epoch=2, error=2.000 [-1.0, -1.0, 0.0]
>epoch=3, error=3.000 [0.0, -1.0, 0.0]
>epoch=4, error=4.000 [0.0, -1.0, 0.0]
All the points on/above the decision boundary belongs to class 1
```




```
>epoch=0, error=2.000 [0.0, 0.0, 0.0]
>epoch=1, error=2.000 [0.0, -1.0, -1.0]
>epoch=2, error=4.000 [0.0, -1.0, -1.0]
```

```
>epoch=3, error=4.000 [0.0, -1.0, -1.0]
>epoch=4, error=4.000 [0.0, -1.0, -1.0]
All the points on/above the decision boundary belongs to class 1
```



Wrong decision boundary for XOR dataset because data can not be separated with one decision boundry

part 6

Given a hyperplane boundary and a point we can classify the point into class 0 or 1 by putting the point coordinates into the hyperplane equation by applying the sign(signum) function on the result(let's call it R)

Assumption :: sign function gives 1 when R is >=0 and 0 when R < 0

when the sign function gives 1 the class of the point is 1 else the class of the point is 0.

In [ ]:

In [ ]:
```
import pandas as pd
import seaborn as sb
import numpy as np
import matplotlib.pyplot as plt
import scipy as stats
from sklearn import preprocessing
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn import metrics #Import scikit-learn metrics module for accuracy calculatio
```

In [ ]:
```
df = pd.read_csv('BitcoinHeistData.csv')
```

In [ ]:
```
df
```

Out[ ]:

	address	year	day	length	weight	count	looped	neigh
0	111K8kZAEnJg245r2cM6y9zgJGHZtJPy6	2017	11	18	0.008333	1	0	
1	1123pJv8jzeFQaCV4w644pzQJzVWay2zcA	2016	132	44	0.000244	1	0	
2	112536im7hy6wtKbpH1qYDWtTyMRAcA2p7	2016	246	0	1.000000	1	0	
3	1126eDRw2wqSkWosjTCre8cjjQW8sSeWH7	2016	322	72	0.003906	1	0	
4	1129TSjKtx65E35GiUo4AYVeyo48twbrGX	2016	238	144	0.072848	456	0	
...	...	...	...	...	...	...	...	
2916692	12D3trgho1vJ4mGtWBRPyHdMJK96TRYSry	2018	330	0	0.111111	1	0	
2916693	1P7PputTcVkhXBmXBvSD9MJ3UYPsiou1u2	2018	330	0	1.000000	1	0	
2916694	1KYiKJEfdJtap9QX2v9BXJMpz2SfU4pgZw	2018	330	2	12.000000	6	6	
2916695	15iPUJsRNZQZHmZZVwmQ63srsmughCXV4a	2018	330	0	0.500000	1	0	
2916696	3LFFBxp15h9KSFtaw55np8eP5fv6kdK17e	2018	330	144	0.073972	6800	0	

2916697 rows × 10 columns

In [ ]:
```
# df.info()
df.isna().sum().sum()
```

Out[ ]:
```
0
```

No nan values found

In [ ]:
```
#still for precaution
df.dropna()
```

Out[ ]:

	address	year	day	length	weight	count	looped	neigh

	address	year	day	length	weight	count	looped	neigh
0	111K8kZAEnJg245r2cM6y9zgJGHZtJPy6	2017	11	18	0.008333	1	0	
1	1123pJv8jzeFQaCV4w644pzQJzVWay2zcA	2016	132	44	0.000244	1	0	
2	112536im7hy6wtKbpH1qYDWtTyMRAcA2p7	2016	246	0	1.000000	1	0	
3	1126eDRw2wqSkWosjTCre8cjjQW8sSeWH7	2016	322	72	0.003906	1	0	
4	1129TSjKtx65E35GiUo4AYVeyo48twbrGX	2016	238	144	0.072848	456	0	
...	...	...	...	...	...	...	...	
2916692	12D3trgho1vJ4mGtWBRPyHdMJK96TRYSry	2018	330	0	0.111111	1	0	
2916693	1P7PputTcVkhXBmXBvSD9MJ3UYPsiou1u2	2018	330	0	1.000000	1	0	
2916694	1KYiKJEfdJtap9QX2v9BXJMpz2SfU4pgZw	2018	330	2	12.000000	6	6	
2916695	15iPUJsRNZQZHmZZVwmQ63srsmughCXV4a	2018	330	0	0.500000	1	0	
2916696	3LFFBxp15h9KSFtaw55np8eP5fv6kdK17e	2018	330	144	0.073972	6800	0	

2916697 rows × 10 columns

In [ ]:

```
n = len(pd.unique(df['address']))
n
```

Out[ ]:

```
2631095
```

In [ ]:

```
df=df.drop(['address'],axis=1)
#dropped address column because two many unique values will only
#contribute to computation complexity when the address feature is not significant
df
```

Out[ ]:

	year	day	length	weight	count	looped	neighbors	income	label
0	2017	11	18	0.008333	1	0	2	1.000500e+08	princetonCerber
1	2016	132	44	0.000244	1	0	1	1.000000e+08	princetonLocky
2	2016	246	0	1.000000	1	0	2	2.000000e+08	princetonCerber
3	2016	322	72	0.003906	1	0	2	7.120000e+07	princetonCerber
4	2016	238	144	0.072848	456	0	1	2.000000e+08	princetonLocky
...	...	...	...	...	...	...	...	...	...
2916692	2018	330	0	0.111111	1	0	1	1.255809e+09	white
2916693	2018	330	0	1.000000	1	0	1	4.409699e+07	white
2916694	2018	330	2	12.000000	6	6	35	2.398267e+09	white
2916695	2018	330	0	0.500000	1	0	1	1.780427e+08	white
2916696	2018	330	144	0.073972	6800	0	2	1.123500e+08	white

2916697 rows × 9 columns

Part 1

In [ ]:
```
target=['label']
features=df.columns[:-1]
features
```

Out[ ]:
```
Index(['year', 'day', 'length', 'weight', 'count', 'looped', 'neighbors',
       'income'],
      dtype='object')
```

In [ ]:
```
X=df[features]
print(X)
Y=df[target]
print(Y)
```

```
         year  day  length     weight  count  looped  neighbors        income
0        2017   11      18   0.008333      1       0          2  1.000500e+08
1        2016  132      44   0.000244      1       0          1  1.000000e+08
2        2016  246       0   1.000000      1       0          2  2.000000e+08
3        2016  322      72   0.003906      1       0          2  7.120000e+07
4        2016  238     144   0.072848    456       0          1  2.000000e+08
...       ...  ...     ...        ...    ...     ...        ...           ...
2916692  2018  330       0   0.111111      1       0          1  1.255809e+09
2916693  2018  330       0   1.000000      1       0          1  4.409699e+07
2916694  2018  330       2  12.000000      6       6         35  2.398267e+09
2916695  2018  330       0   0.500000      1       0          1  1.780427e+08
2916696  2018  330     144   0.073972   6800       0          2  1.123500e+08

[2916697 rows x 8 columns]
                   label
0        princetonCerber
1         princetonLocky
2        princetonCerber
3        princetonCerber
4         princetonLocky
...                  ...
2916692            white
2916693            white
2916694            white
2916695            white
2916696            white

[2916697 rows x 1 columns]
```

In [ ]:
```
#encoding the target column
label_encoder = preprocessing.LabelEncoder()
df['label']= label_encoder.fit_transform(df['label'])
```

In [ ]:
```
df = df.sample(frac=1)
train_size = 0.70
test_size = 0.15
valid_size=0.15
```

```
train_index = int(len(df)*train_size)

df_train = df[0:train_index]
df_rem = df[train_index:]

valid_index = int(len(df)*valid_size)

df_valid = df[train_index:train_index+valid_index]
df_test = df[train_index+valid_index:]

X_train, y_train = df_train.drop(columns='label'), df_train['label']
X_valid, y_valid = df_valid.drop(columns='label'), df_valid['label']
X_test, y_test = df_test.drop(columns='label'), df_test['label']
trees_both=[]
```

In [ ]:

```
# Create Decision Tree classifer object with entropy
accuracy_entropy=[]
depth=[4,8,10,15,20]
for i in depth:
    clf = DecisionTreeClassifier(criterion="entropy", max_depth=i)
    # Train Decision Tree Classifer
    clf = clf.fit(X_train,y_train)
    #Predict the response for test dataset
    y_pred = clf.predict(X_valid)
    print("Accuracy for depth",i,":",metrics.accuracy_score(y_valid, y_pred))
    accuracy_entropy.append(metrics.accuracy_score(y_valid, y_pred))
```

```
Accuracy for depth 4 : 0.9855155610005851
Accuracy for depth 8 : 0.9857807014335869
Accuracy for depth 10 : 0.9870424041837331
Accuracy for depth 15 : 0.9879201104447045
Accuracy for depth 20 : 0.9860984128145114
```

In [ ]:

```
# Create Decision Tree classifer object with gini index
accuracy_gini=[]
for i in depth:
    clf = DecisionTreeClassifier(criterion="gini", max_depth=i)
    # Train Decision Tree Classifer
    clf = clf.fit(X_train,y_train)
    #Predict the response for test dataset
    y_pred = clf.predict(X_valid)
    print("Accuracy for depth",i,":",metrics.accuracy_score(y_valid, y_pred))
    accuracy_gini.append(metrics.accuracy_score(y_valid, y_pred))
```

```
Accuracy for depth 4 : 0.9855155610005851
Accuracy for depth 8 : 0.9862081260971328
Accuracy for depth 10 : 0.9867864065242832
Accuracy for depth 15 : 0.9878286827091867
Accuracy for depth 20 : 0.9863589818607372
```

In [ ]:

```
for i in range(len(depth)):
    print("Accuracy for depth",depth[i],"with entropy:",accuracy_entropy[i])
    print("Accuracy for depth",depth[i],"with gini:",accuracy_gini[i])
```

```
Accuracy for depth 4 with entropy: 0.9855155610005851
Accuracy for depth 4 with gini: 0.9855155610005851
Accuracy for depth 8 with entropy: 0.9857807014335869
```

```
Accuracy for depth 8 with gini: 0.9862081260971328
Accuracy for depth 10 with entropy: 0.9870424041837331
Accuracy for depth 10 with gini: 0.9867864065242832
Accuracy for depth 15 with entropy: 0.9879201104447045
Accuracy for depth 15 with gini: 0.9878286827091867
Accuracy for depth 20 with entropy: 0.9860984128145114
Accuracy for depth 20 with gini: 0.9863589818607372
```

The best accuracy for the ginni and entropy creteria is best with the max depth of 15

In [ ]:

```
clf = DecisionTreeClassifier(criterion="entropy", max_depth=15)
# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
#Predict the response for test dataset
y_pred = clf.predict(X_valid)
print("Accuracy for depth",15,":",metrics.accuracy_score(y_valid, y_pred))
accuracy_entropy.append(metrics.accuracy_score(y_valid, y_pred))
```

```
Accuracy for depth 15 : 0.9878766822703335
```

In [ ]:

```
clf = DecisionTreeClassifier(criterion="gini", max_depth=15)
# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
#Predict the response for test dataset
y_pred = clf.predict(X_test)
print("Accuracy for depth",15,":",metrics.accuracy_score(y_test, y_pred))
# accuracy_gini.append(metrics.accuracy_score(y_valid, y_pred))
```

```
Accuracy for depth 15 : 0.9881921619360649
```

Part 2

Ensembling is a method to combine multiple not-so-good models to get a better performing model. Create 100 different decision stumps (max depth 3). For each stump, train it on randomly selected 50% of the training data, i.e., select data for each stump separately. Now, predict the test samples' labels by taking a majority vote of the output of the stumps. How is the performance affected as compared to parts (a)

In [ ]:

```
def train_trees(X, Y, num_trees):
    indices = [i for i in range(X.shape[0])]

    # Here, note that we have set max_depth = 3 which
    # makes all the classifiers weak
    trees = [DecisionTreeClassifier(max_depth=3) for _ in range(num_trees)]
    for tree in trees:
        # Selecting n random samples with replacement from training set
        random_indices = np.random.choice(indices, X.shape[0])
        print(random_indices)
        # Bootstrap training data
        X_bootstrap= X.iloc[random_indices]
        Y_bootstrap= Y.iloc[random_indices]
        # X_bootstrap = X[random_indices]
        # Y_bootstrap = Y[random_indices]

        tree.fit(X_bootstrap, Y_bootstrap)
```

```
    return trees
```

In [ ]:
```
def predict(X, trees):
    predictions = []
    for tree in trees:
        Y_pred = tree.predict(X)
        predictions.append(Y_pred)

    predictions = np.array(predictions)

    # Aggregating all predictions to get final prediction
    # Since this is a classification problem, we use mode
    # i.e. the prediction that occurs the maximum number
    # of times. In case of regression problem, we use mean
    prediction = np.array(stats.stats.mode(predictions))
    return prediction[0, 0, :]
```

In [ ]:
```
# NUM_RANDOM_FEATURES =
# put 50% of data from the dataset df into x train and y train
df = df.sample(frac=1)
train_size = 0.50
# test_size = 0.50

train_index = int(len(df)*train_size)

df_train = df[0:train_index]
df_rem = df[train_index:]

# df_test = df[train_index+valid_index:]

X_train, y_train = df_train.drop(columns='label'), df_train['label']
# X_valid, y_valid = df_valid.drop(columns='label'), df_valid['label']
X_test, y_test = df_rem.drop(columns='label'), df_rem['label']
```

In [ ]:
```
# accuracies = []
trees = train_trees(X_train, y_train, 100)
prediction = predict(X_test, trees)
# accuracy = (prediction == y_test).sum() / prediction.shape[0]
# accuracies.append(accuracy)

# plt.plot(accuracies)
```

```
[ 487038 1007506  563817 ... 1368765 1150143   97461]
[1407639   39934  202282 ...   92420  423861  476474]
[ 790485 1354128  382393 ...  244555  422526  909608]
[ 300892  159636  801999 ... 1300258 1271287 1307805]
[ 812440 1316672  967794 ... 1256109  171250 1216143]
[380163 598884 413676 ... 458473 324453 856961]
[ 519269  548718 1156057 ...  183524   86952  243393]
[507258 233889 106903 ... 805654 701789 505699]
[ 546497  658392  698508 ...  786156 1368889  217902]
[1244210  739834 1293874 ...  594322  737519  569317]
[1038793  910238  966858 ...   81740  819588  153268]
[1240813 1078517  307703 ...  938405  924306  337887]
[492803 674715 517661 ... 408470 492369 776330]
```

```
[ 664939  735775  963678 ...    8297 1247053  459211]
[ 51247 636122 261997 ... 755708 521831 792985]
[993794 286242 565969 ... 659036 765881 641545]
[569821 769165 680852 ... 735125 779739 959945]
[ 605713 1404682  976885 ...  740845  944513  477987]
[  97579 1294435  467748 ...   45698 1303553  756960]
[ 664463   29629  719953 ... 1427001  529414 1419260]
[1287116  636249  427210 ... 1253818 1075314  715063]
[ 757228 1274626  334159 ...  279499 1424781 1428208]
[ 175747  930739  953735 ... 1161898 1116252  959795]
[ 983968  488997   46597 ... 1185829  930460 1208424]
[1361592   69999  591252 ... 1163116   49587  951567]
[  61837  975933  864424 ... 1397180 1001041  946954]
[ 974121  479586  255236 ... 1308838    9128  591585]
[ 716991  949564  717526 ...  490460  886779 1365359]
[ 493381  150331 1034749 ... 1358965  672295  985807]
[ 283845  859449 1447358 ... 1258167  233250  436222]
[ 982817  356944 1454024 ...  330816 1425310  181151]
[476331 978760 897728 ...  45593 499549 719541]
[ 677586  738497 1175960 ... 1066365  731885  378190]
[ 872939  857057  181534 ...  872671    1663 1402775]
[  45960  418015   50944 ... 1328696  836912  368057]
[ 443113 1183829  885177 ...  974229  614334 1415379]
[355546 845810 492055 ... 347048 112993 152460]
[ 487304  676929 1044852 ...  504399  208439 1213199]
[1428359  671319  643070 ... 1409111  357178   82470]
[  32192  860616  167497 ...  655095 1031646 1002204]
[ 766056 1146619  917254 ...  939197  676145 1310461]
[1238937 1098531  774903 ... 1053023   72965  418299]
[1242303  505996 1300313 ...  696197 1304215 1095193]
[  89426  184651  787849 ...  270194 1046648  745780]
[1160325  834564  353747 ... 1211148  510153  876046]
[452027 811146 805860 ... 825108 384110 322750]
[123501 411749 842365 ... 432345 195655 338327]
[1342797  361052  453262 ...  577745  290144 1302703]
[1368190 1277343  397451 ...  719887   66039  187308]
[656832 803070 281982 ... 763158 725581 227831]
[1156313  961013  313496 ...  216963  669786  220843]
[1380000  641153  952141 ...  587820 1351046  619124]
[ 494407  312887  820859 ... 1447779 1411868  276948]
[1108418  989178  389155 ...  700877 1230786  442667]
[  87023  249857 1161162 ... 1100196  250857  798845]
[ 677355  165861 1048584 ...  884577  211382  688622]
[1010041  438591  843989 ...  710285  253748  189112]
[ 579724 1299307  414920 ...  171022 1310999 1090755]
[1156825 1336039  162910 ...  858178  336273 1066958]
[ 824044 1373202   72400 ... 1394963 1004906 1087592]
[1216310  798048 1078414 ...  261018 1096595  802039]
[224611 316911 662692 ... 554168 699407 510108]
[ 238799  322937  579349 ...  845988 1024206  399991]
[ 394539  485123 1096147 ... 1073816 1415213  908968]
[ 598277 1035679 1311569 ...  268564  178296  615053]
[ 414048  886174 1096776 ... 1404711  679012 1249364]
[1133408 1195699  136425 ...  798810 1022772  987490]
[  66851  885814  294697 ... 1073227  620932  630341]
[ 975644  486284 1303351 ... 1394504  339615 1092543]
[926371 237732 580636 ... 899039 695894 105346]
[  47709  560631  794034 ... 1103234  721924 1151959]
[  95622  571195  648057 ... 1349376  349776  937730]
[1082161 1398378  360932 ...   34291 1013879  623918]
```

```
[ 352656 1282263   79299 ... 1323511  259815 1110811]
[1215300  998254  341342 ... 1099292 1215581  595246]
[ 443669  928346  999042 ... 1010714 1071909  190991]
[ 953301  773781  346109 ... 1453316  265343 1020354]
[ 410902 1289046 1429213 ...  284940 1269382 1184667]
[ 427767 1418616  454440 ...  462357  474576  671479]
[ 645545  804430 1324636 ...  428144  332429  762827]
[1064735 1251350  562385 ...  688578 1371566  414722]
[ 937302 1142144  570637 ...  844122  660801 1019414]
[1203832 1402864  646885 ...  908482  481109 1388782]
[622465 283471 706428 ... 202711 133764 220826]
[1175705 1208527 1193103 ...  296395  176641 1078478]
[ 774639  397155  812513 ... 1138326 1009635 1378101]
[742162  10341 897202 ... 984904 800333 799714]
[ 245072 1399541  856299 ...  200808  716042 1325927]
[ 873308  974438  452058 ... 1264068 1058405  346096]
[ 255207 1327077  587955 ...   91385  323130  639814]
[393722 403170 297684 ... 685136 650412 115588]
[ 586775  680916 1075386 ...  184380 1455744  318247]
[ 529183    4582  205242 ... 1002904  710015  157413]
[1361544  353017  321029 ...  425699 1334477  474951]
[ 770879 1390030  782581 ...   70825  590291  465473]
[1455195  424385  187844 ...  221484  341448  175547]
[ 366292  980816  260365 ... 1247176   19642  352156]
[1167245  111012   41229 ...  638733  778621 1264197]
[ 312166 1293266  253087 ... 1274547  222766  995451]
[ 372170  381616  133265 ...  323696  401585 1019053]
```

In [ ]:

```
accuracy = (prediction == y_test).sum() / prediction.shape[0]
accuracy
```

Out[ ]:

```
0.985858666204043
```

In [ ]:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

# Create adaboost classifer object
abc = AdaBoostClassifier(n_estimators=100, learning_rate=1, random_state=42)

# Train Adaboost Classifer
model1 = abc.fit(X_train, y_train)

#Predict the response for test dataset
y_pred_abc = model1.predict(X_test)
```

In [ ]:

```
accuracy = (y_pred_abc == y_test).sum() / y_pred_abc.shape[0]
accuracy
```

Out[ ]:

```
0.9858230094442414
```

In [ ]: