CS559- Homework Assignment 2

Problem 1

Solution

Step 1: Import libraries.

Pandas for data manipulation
Numpy for matrices calculation
StandardScaler for data normalization
LinearDiscriminantAnalysis for data classification
train_test_split for separating training data and testing data
classification_report for report generation

In [1]:
```
#import libraries
import pandas as pd
import numpy as np

from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

Step 2: Read data into Jypyter.

Read csv file to get data for later analysis. The data looks like in the table where columns 0 -3 are features and column 4 is labels.

In [2]:
```
#import data
path = 'iris.data'
data = pd.read_csv(path, header=None)
data.head()
```

Out[2]:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Step 3: Check if imported data is missing.

The data is completed, with 150 entries and non-null columns.

In [3]:
```
#visualize data
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   0       150 non-null    float64
 1   1       150 non-null    float64
 2   2       150 non-null    float64
 3   3       150 non-null    float64
 4   4       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

Step 4: Convert the label column into numbers.

Iris-setosa maps to 1, Iris-versicolor to 2, and Iris-virginica to 3.

In [4]:
```
#convert target column to class number using pandas
data.loc[data[4] == 'Iris-setosa', 4] = 1
data.loc[data[4] == 'Iris-versicolor', 4] = 2
data.loc[data[4] == 'Iris-virginica', 4] = 3
data
```

Out[4]:

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	1
1	4.9	3.0	1.4	0.2	1
2	4.7	3.2	1.3	0.2	1
3	4.6	3.1	1.5	0.2	1
4	5.0	3.6	1.4	0.2	1
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	3
146	6.3	2.5	5.0	1.9	3
147	6.5	3.0	5.2	2.0	3
148	6.2	3.4	5.4	2.3	3
149	5.9	3.0	5.1	1.8	3

150 rows × 5 columns

Step 5: Apply one-to-all classification to the label.

Make three copies of data to three classifiers, and each classifies one class against the others.

In [5]:
```
#select data for label 1 if label = 1 -> 1 else 0
data1 = data.copy()
data1.loc[(data1[4] == 2) | (data1[4] == 3), 4] = 0
print(data1[4].value_counts())
```

```
0    100
1     50
Name: 4, dtype: int64
```

In [6]:
```
#select data for label 2 if label = 2 -> 1 else 0
data2 = data.copy()
data2.loc[(data2[4] == 1) | (data2[4] == 3),4] = 0
data2.loc[(data2[4] == 2) ,4] = 1
print(data2[4].value_counts())
```

```
0    100
1     50
Name: 4, dtype: int64
```

In [7]:
```
#select data for label 3 if label = 3 -> 1 else 0
data3 = data.copy()
data3.loc[(data3[4] == 1) | (data3[4] == 2), 4] = 0
data3.loc[(data3[4] == 3), 4] = 1
print(data3[4].value_counts())
```

```
0    100
1     50
Name: 4, dtype: int64
```

Step 6: Normalize features.

Normalize features for 3 datasets and exclude labels from features.

In [8]:
```
#normalize data
def norm_data(data):
    norm_features = StandardScaler().fit_transform(data.loc[:,0:3])
    targets= data[4]
    targets = targets.astype(int)
    return norm_features, targets

norm_features1, targets1 = norm_data(data1) #C1
norm_features2, targets2 = norm_data(data2) #C2
norm_features3, targets3 = norm_data(data3) #C3
```

Step 7: Split training data from testing data.

In [9]:
```
#separate data into train/test sets
X_train1, X_test1, y_train1, y_test1 = train_test_split(norm_features1,targets1, test_size=0.25, random_state=42)
X_train2, X_test2, y_train2, y_test2 = train_test_split(norm_features2,targets2, test_size=0.25, random_state=42)
X_train3, X_test3, y_train3, y_test3 = train_test_split(norm_features3,targets3, test_size=0.25, random_state=42)
```

Step 8: Apply LDA from sklearn for three training data.

In [10]:
```
#apply LDA
lda1 = LinearDiscriminantAnalysis()
lda2 = LinearDiscriminantAnalysis()
lda3 = LinearDiscriminantAnalysis()
X_train1 = lda1.fit_transform(X_train1,y_train1)
X_train2 = lda2.fit_transform(X_train2,y_train2)
X_train3 = lda3.fit_transform(X_train3,y_train3)
```

Step 9: Predict the trained models using testing data features.

In [11]:
```
#test LDA
y_pred1 = lda1.predict(X_test1)
y_pred2 = lda2.predict(X_test2)
y_pred3 = lda3.predict(X_test3)
```

Step 10: Generate the classification reports for three models.

Report 1: Setosa versus all
Report 2: Versicolor versus all
Report 3: Veronica versus all

In [12]:
```
t_names1 = ['not setosa', 'setosa']
t_names2 = ['not versicolor', 'versicolor']
t_names3 = ['not verginica', 'verginica']

print(classification_report(y_test1,y_pred1, target_names=t_names1))
```

```
              precision    recall  f1-score   support

  not setosa       1.00      1.00      1.00        23
      setosa       1.00      1.00      1.00        15

    accuracy                           1.00        38
   macro avg       1.00      1.00      1.00        38
weighted avg       1.00      1.00      1.00        38
```

In [13]:
```
print(classification_report(y_test2,y_pred2, target_names=t_names2))
```

```
                precision    recall  f1-score   support

not versicolor       0.83      0.89      0.86        27
    versicolor       0.67      0.55      0.60        11

      accuracy                           0.79        38
     macro avg       0.75      0.72      0.73        38
  weighted avg       0.78      0.79      0.78        38
```

In [14]:
```
print(classification_report(y_test3,y_pred3, target_names=t_names3))
```

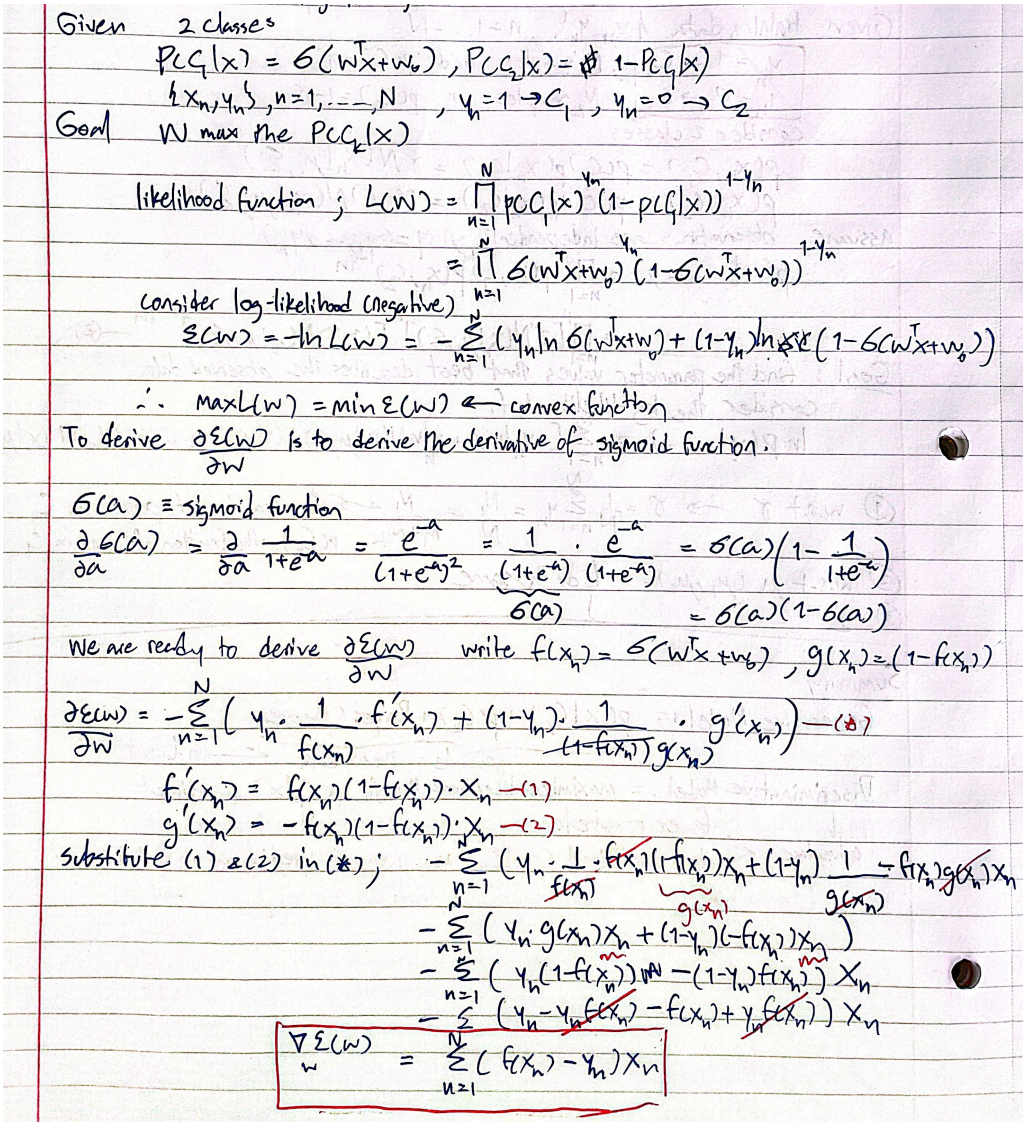
```
                precision    recall  f1-score   support

not verginica       1.00      0.92      0.96        26
    verginica       0.86      1.00      0.92        12

      accuracy                           0.95        38
     macro avg       0.93      0.96      0.94        38
  weighted avg       0.95      0.95      0.95        38
```

Problem 2

Part 1
Solution



Part 2
Solution

Step 1: Import the data.

In [2]:
```
path = 'wdbc.data'
data = pd.read_csv(path,header=None)
data.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9	...	22	23	24	25	26	27	28	29	30	
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	25.38	17.33	184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.11
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07

5 rows × 32 columns

Step 2: Drop column 0, which is ID column.

In [3]:
```
data = data.drop(columns=[0])
data
```

Out[3]:

	1	2	3	4	5	6	7	8	9	10	...	22	23	24	25	26	27	28	29	30
0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	...	25.380	17.33	184.60	2019.0	0.16220	0.66560	0.7119	0.2654	0.4601
1	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	...	24.990	23.41	158.80	1956.0	0.12380	0.18660	0.2416	0.1860	0.2750
2	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	...	23.570	25.53	152.50	1709.0	0.14440	0.42450	0.4504	0.2430	0.3613
3	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	...	14.910	26.50	98.87	567.7	0.20980	0.86630	0.6869	0.2575	0.6638
4	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	...	22.540	16.67	152.20	1575.0	0.13740	0.20500	0.4000	0.1625	0.2364
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
564	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1726	...	25.450	26.40	166.10	2027.0	0.14100	0.21130	0.4107	0.2216	0.2060
565	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1752	...	23.690	38.25	155.00	1731.0	0.11660	0.19220	0.3215	0.1628	0.2572
566	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1590	...	18.980	34.12	126.70	1124.0	0.11390	0.30940	0.3403	0.1418	0.2218
567	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2397	...	25.740	39.42	184.60	1821.0	0.16500	0.86810	0.9387	0.2650	0.4087
568	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1587	...	9.456	30.37	59.16	268.6	0.08996	0.06444	0.0000	0.0000	0.2871

569 rows × 31 columns

Step 3: Map the label type; M = malignant to 1, and B = benign to 0. Also, confirm that the data is not missing.

In [5]:
```
#map M to 1, B to 0
data.loc[data.iloc[:,0] == 'M', 1] = 1
data.loc[data.iloc[:,0] == 'B', 1] = 0
#check if labels are not missing (label 0: 357, label 1: 212)
print(data.iloc[:,0].value_counts())

#prepare dataset for analysis
data_arr = data.to_numpy()
labels = data_arr[:,0] # dimension: (569,1)
features = data_arr[:,1:] # dimension: (569,30)
```

```
0    357
1    212
Name: 1, dtype: int64
```

Step 4: Create a function for stochastic gradient descent and mini-batch gradient descent to find optimum w

Stochastic gradient descent

In [6]:
```
#stochastic gradient descent
def sgd_gradient(features,labels,weights,step,epochs):
    for e in range(0,epochs):
        data = np.hstack((features,labels))
        np.random.shuffle(data)
        for i in range(0,len(labels)):
            w = np.zeros((features.shape[1],1)) #element of weights
            a = np.dot(features[i,:],weights) #prediction using sigmoid function. Features dim(1,30) weight dim(30,1
            sigm = 1.0/(1+math.exp(-a))

            for j in range(0,len(weights)): #iterate through weights,elementwise
                w[j] = w[j] + (sigm - labels[i,:])*features[i,j]
            #update
            for k in range(0,len(weights)): #update weights after complete each sample
                weights[k] = weights[k] - step*w[k]/len(labels)
    return weights
```

Mini-batch gradient descent

```
def mnb_gradient(features,labels,weights,step,epochs):

    for e in range(0,epochs):
        data = np.hstack((features,labels))
        np.random.shuffle(data)
        mini_batches = get_batches(features,labels,batch_size=50)

        for m in mini_batches:
            f,l = m #get individual mini batch
            w = np.zeros((f.shape[1],1)) #element of weights

            for i in range(0,len(l)):

                a = np.dot(f[i,:],weights) #f dimension(1,30) weights dimension(30,1)
                sigm = 1/(1+math.exp(-a)) #prediction using sigmoid function

                for j in range(0,len(weights)):
                    w[j] = w[j] + (sigm - l[i,:])*f[i,j]

            for k in range(0,len(weights)): #update weights after complete each batch
                weights[k] =  weights[k] - step*w[k]/len(labels)

    return weights
```

Note that mini-batch depends on getting a small batch every iteration, therefore; the function ‘get_batch’ is created.

In [7]:
```
#mini-batch gradient descent from Lasso
def get_batches(features,labels,batch_size):
    batches = []
    data = np.hstack((features,labels))
    n_batch = data.shape[0]

    for i in range(0, n_batch):
        mini_batch = data[i*batch_size:(i+1)*batch_size,:]
        features_mini = mini_batch[:,:-1]
        labels_mini = mini_batch[:,-1].reshape((-1,1))
        batches.append((features_mini,labels_mini)) #append pairs of (features_mini, labels_mini) to batches

        if n_batch % batch_size != 0: #if data is small than batch_size
            mini_batch = data[i*batch_size:n_batch]
            features_mini = mini_batch[:,:-1]
            labels_mini = mini_batch[:,-1].reshape((-1,1))
            batches.append((features_mini,labels_mini))

    return batches
```

Step 5: Create a predict function.

```
def predict(features,weights_updated):
    threshold = 0.5
    Y_pred = []

    fb = np.c_[np.ones((len(features[:,1]),1)),features]

    for i in range(0,len(features[:,1])):

        a = np.dot(fb[i,:],weights_updated) #features dim(1,30) weights dim(30,1)

        sigm = 1/(1+math.exp(-a)) #predict using sigmoid function

        pc1 = np.round(sigm,2) #class 1 class2: pc2 = 1-pc1
        if pc1 < threshold: #pc2 > threshold
            Y_pred.append(0) #class 1: benign
        else:
            Y_pred.append(1) #class 2: malignant

    return np.array(Y_pred).reshape((-1,1))
```

Step 6: Create a train model function, using step = 0.01 and iterations = 100

In [8]:
```
def train_sgd(features,labels,step=0.01,iterations=100):

    weights = np.ones((features.shape[1]+1,1))
    features_wbias = np.c_[np.ones((len(labels),1)),features]

    w_trained = sgd_gradient(features_wbias,labels,weights,step,iterations)

    return w_trained

def train_mnb(features,labels,step=0.01,iterations=100):

    weights = np.ones((features.shape[1]+1,1))
    features_wbias = np.c_[np.ones((len(labels),1)),features]

    w_trained = mnb_gradient(features_wbias,labels,weights,step,iterations)

    return w_trained
```

Step 7: Validate the model performance by using a 5-fold validation. Split a new train data and test data, train model, predict model, and report the classification performance for every fold.

In [9]:
```
#normalize features
features_normed = StandardScaler().fit_transform(features) # dimension: (569,30)
labels = labels.reshape((-1,1)) # dimension: (569,1)
data = np.hstack((features_normed,labels))
#print(labels[1,0])
#split data for train-test use cross validation
k_fold = 5
data_test_size = int(np.floor((1/k_fold)*data.shape[0]))
label_names = ['benign(-)','malignant(+)']
for k in range(0,k_fold):

    data_test = data[k*data_test_size:(k+1)*data_test_size,:]
    data_train = np.delete(data,slice(k*data_test_size,(k+1)*data_test_size),0)

    features_train = data_train[:,:-1]
    labels_train = data_train[:,-1].reshape((-1,1))

    features_test = data_test[:,:-1]
    labels_test = data_test[:,-1].reshape((-1,1))
    labels_test=labels_test.flatten().astype('int32')

    #train SGD
    W_trained_sgd = train_sgd(features_train,labels_train)
    #predict SGD
    labels_pred_sgd = predict(features_test,W_trained_sgd)
    labels_pred_sgd=labels_pred_sgd.flatten().astype('int32')
    print('report for SGD model iteration {}'.format(k))
    print(classification_report(labels_test,labels_pred_sgd,target_names=label_names))

    #train Mini-batch
    W_trained_mnb = train_mnb(features_train,labels_train)
    #predict MNB
    labels_pred_mnb = predict(features_test,W_trained_mnb)
    labels_pred_mnb = labels_pred_mnb.flatten().astype('int32')
    print('report for MNB model iteration {}'.format(k))
    print(classification_report(labels_test,labels_pred_mnb,target_names=label_names))
```

Below is the report for each iteration's stochastic model(SGD) and mini-batch model(MNB). Notice that the MNB model performs better precision and accuracy than SGD. However, SGD is much faster than the MNB, which is a trade-off.

report for SGD model iteration 0

	precision	recall	f1-score	support
benign(-)	0.85	0.87	0.86	45
malignant(+)	0.91	0.90	0.90	68
accuracy			0.88	113
macro avg	0.88	0.88	0.88	113
weighted avg	0.89	0.88	0.89	113

report for MNB model iteration 0

	precision	recall	f1-score	support
benign(-)	0.87	0.91	0.89	45
malignant(+)	0.94	0.91	0.93	68
accuracy			0.91	113
macro avg	0.91	0.91	0.91	113
weighted avg	0.91	0.91	0.91	113

report for SGD model iteration 1

	precision	recall	f1-score	support
benign(-)	0.90	0.86	0.88	64
malignant(+)	0.83	0.88	0.85	49
accuracy			0.87	113
macro avg	0.86	0.87	0.87	113
weighted avg	0.87	0.87	0.87	113

report for MNB model iteration 1

	precision	recall	f1-score	support
benign(-)	0.92	0.91	0.91	64
malignant(+)	0.88	0.90	0.89	49
accuracy			0.90	113
macro avg	0.90	0.90	0.90	113
weighted avg	0.90	0.90	0.90	113

report for SGD model iteration 2

	precision	recall	f1-score	support
benign(-)	0.94	0.92	0.93	74
malignant(+)	0.85	0.90	0.88	39
accuracy			0.91	113
macro avg	0.90	0.91	0.90	113
weighted avg	0.91	0.91	0.91	113

report for MNB model iteration 2

	precision	recall	f1-score	support
benign(-)	0.97	0.95	0.96	74
malignant(+)	0.90	0.95	0.92	39
accuracy			0.95	113
macro avg	0.94	0.95	0.94	113
weighted avg	0.95	0.95	0.95	113

report for SGD model iteration 3

	precision	recall	f1-score	support
benign(-)	0.97	0.94	0.96	83
malignant(+)	0.85	0.93	0.89	30
accuracy			0.94	113
macro avg	0.91	0.94	0.92	113
weighted avg	0.94	0.94	0.94	113

report for MNB model iteration 3

	precision	recall	f1-score	support
benign(-)	1.00	0.95	0.98	83
malignant(+)	0.88	1.00	0.94	30
accuracy			0.96	113
macro avg	0.94	0.98	0.96	113
weighted avg	0.97	0.96	0.97	113

report for SGD model iteration 4

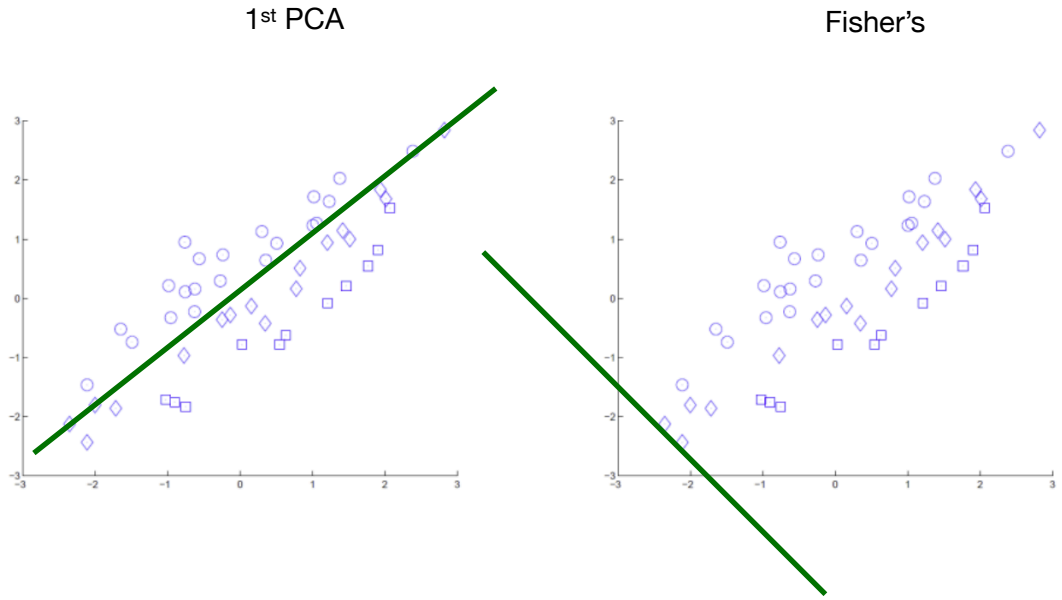
	precision	recall	f1-score	support
benign(-)	0.97	0.86	0.91	90
malignant(+)	0.62	0.91	0.74	23
accuracy			0.87	113
macro avg	0.80	0.88	0.82	113
weighted avg	0.90	0.87	0.88	113

report for MNB model iteration 4

	precision	recall	f1-score	support
benign(-)	0.98	0.89	0.93	90
malignant(+)	0.68	0.91	0.78	23
accuracy			0.89	113
macro avg	0.83	0.90	0.85	113
weighted avg	0.91	0.89	0.90	113

Problem 3

Part 1
Solution



Part 2
Solution

A)
Find the covariance matrix.

Calculate the mean vector. It value is [0,0] for x and y.

```
In [1]: import math
        import numpy
        from numpy import linalg
```

```
In [2]: #Given three points
        p1 = numpy.array([[2],[2]])
        p2 = numpy.array([[0],[0]])
        p3 = numpy.array([[-2],[-2]])
        N = 3
```

```
In [3]: #calculate mean for both x and y directions
        Mean = (1/N)*(p1+p2+p3)
        Mean
```

```
Out[3]: array([[0.],
               [0.]])
```

Calculate Cov(x,y), Cov(y,x), Var(x), and Var(y).
Create a covariance matrix, Σ.

In [7]:
```
#create covariance matrix

Cov_m = numpy.array([[Var_x[0], Cov_x_y[0]],[Cov_y_x[0], Var_y[0]]])
Cov_m
```

Out[7]:
```
array([[4., 4.],
       [4., 4.]])
```

In [5]:
```
#calculate variances
Var_x = ((p1[0] - Mean[0])**2 + (p2[0] - Mean[0])**2 + (p3[0] - Mean[0])**2)/(N-1)

Var_x
```

Out[5]:
```
array([4.])
```

In [6]:
```
Var_y = ((p1[1] - Mean[1])**2 + (p2[1] - Mean[1])**2 + (p3[1] - Mean[1])**2)/(N-1)

Var_y
```

Out[6]:
```
array([4.])
```

Compute the eigenvalues and eigenvectors by using SVD method, in the form $\Sigma = U\Lambda V^{\top}$. First, find the vector V and Λ from $\Sigma^{\top}\Sigma$. Note that the diagonal matrix for eigenvalues is Λ^2. Take square-root for Λ^2 to get Λ.

In [8]:
```
#find the eigenvalues and corresponding eigenvectors for the covariance matrix
#symmetric matrix of covariance matrix
sym_cov1 = numpy.matmul(numpy.transpose(Cov_m),Cov_m)
w1, v = linalg.eig(sym_cov1)
w1  = numpy.sqrt(numpy.round(numpy.diag(w1),2))
w1 # the eigenvalues matrix
```

Out[8]:
```
array([[8., 0.],
       [0., 0.]])
```

In [9]:
```
v #the orthogonal matrix v for symmetric matrix
v_transpose = numpy.transpose(v)
v_transpose
```

Out[9]:
```
array([[ 0.70710678,  0.70710678],
       [-0.70710678,  0.70710678]])
```

Λ = [8,0; 0,0] and $V^{\top}$ = $\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

Second, find vector U from $\Sigma\Sigma^{\top}$.

In [14]:
```
#find symmetric matrix of covariance matrix
sym_cov2 = numpy.matmul(Cov_m,numpy.transpose(Cov_m))
w2, u = linalg.eig(sym_cov2)
u #another orthogonal matrix u for symmetric matrix
```

Out[14]:
```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

U = $\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

Third, find the first principle component of Σ

```
In [15]: #compute first pc
         pc1 = u[:,0]*w1[0,0]*v_transpose[0,:]
         pc1
Out[15]: array([4., 4.])
```

W = 1st PC = $u_1\sigma_1 v_1^\top$ = [4,4]

B)
Project the data points on W

$s_1 = W^\top p_1$ = 16
$s_2 = W^\top p_2$ = 0
$s_3 = W^\top p_3$ = -16

```
In [21]: s1 = numpy.dot(pc1,p1)
         s1
Out[21]: array([16.])

In [23]: s2 = numpy.dot(pc1,p2)
         s2
Out[23]: array([0.])

In [24]: s3 = numpy.dot(pc1,p3)
         s3
Out[24]: array([-16.])
```

New variance after projection = $W^\top \Sigma W$ = 256

```
In [26]: #new variance after projection

         new_var = numpy.matmul(numpy.transpose(pc1),numpy.matmul(Cov_m,pc1))
         new_var
Out[26]: 255.99999999999994
```

C)
The cumulative variance is defined by the fraction of summation up to kth component of lambdas by the total components of lambdas. In this case, we have 2 components, $\lambda_1 = 8, \lambda_2 = 0$. Therefore, the fraction is 1, which means all variances are captured.

Problem 4

Part 1
Solution

According to the table, the support vectors(a vector having $\alpha > 0$) are x_1, x_4, x_7, x_9.

Calculate the optimum w; $W = \alpha_1 y_1 x_1 + \alpha_4 y_4 x_4 + \alpha_7 y_7 x_7 + \alpha_9 y_9 x_9$

```
In [3]: #define variables
        x1 = numpy.array([[4],[2.9]])
        x4 = numpy.array([[2.5],[1]])
        x7 = numpy.array([[3.5],[4]])
        x9 = numpy.array([[2],[2.1]])

        alpha1 = 0.414 #x1
        alpha2 = 0.018 #x4
        alpha3 = 0.018 #x7
        alpha4 = 0.414 #x9

        y1 = 1
        y2 = -1
        y3 = 1
        y4 = -1
```

```
In [5]: #calculate w
        w = alpha1*y1*x1 + alpha2*y2*x4 + alpha3*y3*x7 + alpha4*y4*x9
        w

Out[5]: array([[0.846 ],
               [0.3852]])
```

W = [0.846; 0.3852]

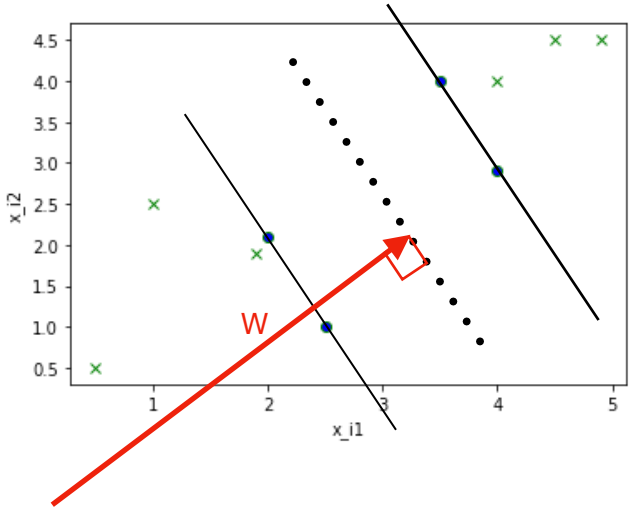
Calculate b optimum; $1/y_1 - W^{\top} x_1$

```
In [14]: #calculate b
         b = 1/y1 - numpy.dot(w.T,x1)
         b

Out[14]: array([[-3.50108]])
```

b = -3.5

Plot the hyperplane with 10 points; marker ‘O’ indicated the support vectors.



In [41]:
```
plt.xlabel('x_i1')
plt.ylabel('x_i2')
plt.plot(xi_1, xi_2, color='green', linestyle='dashed', linewidth = 0,
         marker='o', markerfacecolor='blue', markersize=6, zorder=1)
plt.plot(xi_3, xi_4, color='green', linestyle='dashed', linewidth = 0,
         marker='x', markerfacecolor='blue', markersize=6, zorder=2)
```

Part 2
Solution

The distance x6(y = -1) from the hyperplane is -1.249, further away from -1; therefore, it is not within the margin of the classifier.

In [53]:
```
#find the distance x6 from hyperplane
norm_w = linalg.norm(w)

d6 = (numpy.dot(w.T,x6)+b)/norm_w
d6
```

Out[53]:
```
array([[-1.24982905]])
```

Part 3
Solution

In [56]:
```
#classify the point (3,3)
z = numpy.array([[3],[3]])
val_z = (numpy.dot(w.T,z) + b)/norm_w
val_z
```

Out[56]:
```
array([[0.20710715]])
```

Point z distance from the hyperplane is 0.207; therefore, it is +1 type and within the margin of the classifier.