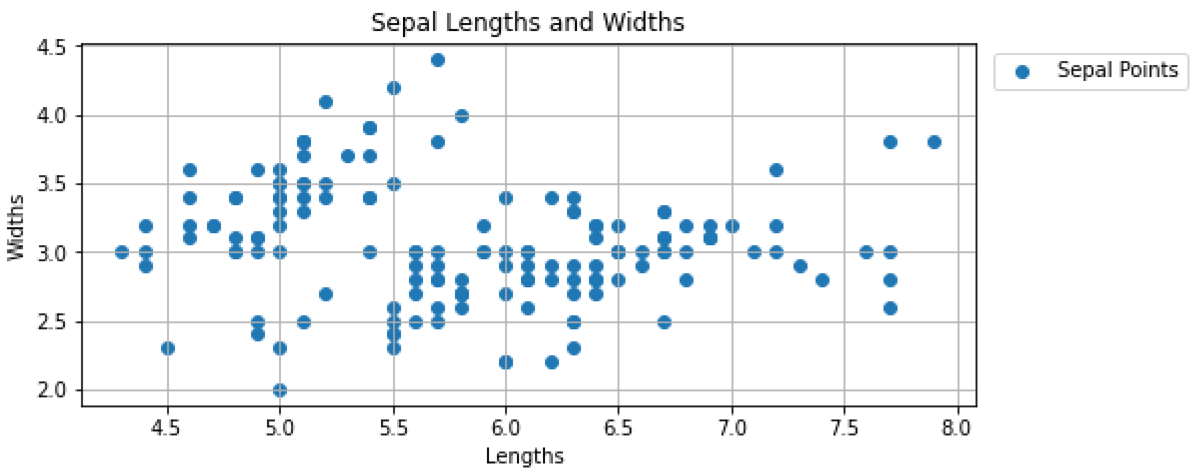
Summary: 33.5/40
Q1: 2/2
Q2: 2.5/3
Q3: 1.5/2
Q4: 5/5
Q5: 3/3
Q6: 1/3
Q7: 3/5
Q8: 3/3
Q9: 4/5
Q10: 8.5/9

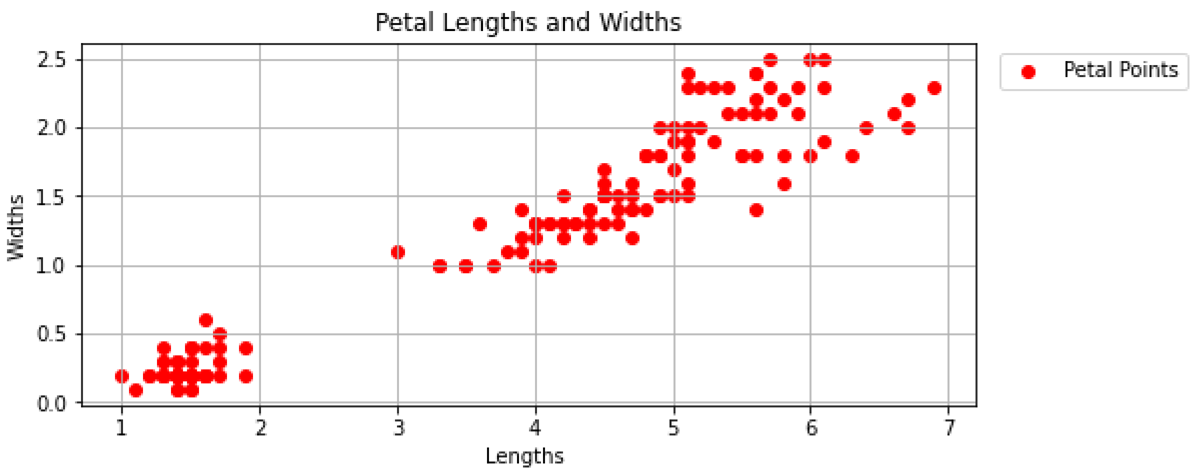
Q1

In [1]:
```
#code given from the question 1:
#: # Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
# load the data
iris = load_iris()
X = iris['data']
y = iris['target']
names = iris['target_names']
# Let's observe the content of the dataset
print(f"The measured features are:\n{iris['feature_names']} \n")
print(f"The classes of Iris in the dataset are are:\n{names} \n")
print(f"The first 3 measurements are:\n{X[:3]}\n")
print(f"The class identifiers of the first 3 plants are: \n {y[:3]}")
print(f"Which can be translated to classes: \n {names[y[:3]]}")
```

```
The measured features are:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

The classes of Iris in the dataset are are:
['setosa' 'versicolor' 'virginica']

The first 3 measurements are:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]]

The class identifiers of the first 3 plants are:
 [0 0 0]
Which can be translated to classes:
 ['setosa' 'setosa' 'setosa']
```

In [2]:
```
# Define X_sepal and X_petal
### BEGIN SOLUTION (~ 2 lines)
X_sepal = X[:,0:2]
X_petal = X[:,2:4] ✓
### END SOLUTION


#increase the size of plot for presentation
plt.figure(figsize = (8,8))
#move the plots to avoid overlap of titles and xlabels for presentation
plt.subplots_adjust(hspace=0.4)


# Plot sepal length and width by class
### BEGIN SOLUTION (~ 5 lines)
# setup the subplot grid
plt.subplot(2, 1, 1)

# plot all rows of sepal lengths against sepal widths
plt.scatter(X_sepal[:,0],X_sepal[:,1], label = "Sepal Points")
plt.title("Sepal Lengths and Widths")
plt.xlabel("Lengths ")
plt.ylabel("Widths")
plt.legend()
plt.legend(bbox_to_anchor = (1.25,1)) #moves the legend off the plot
plt.grid()
### END SOLUTION

# Plot petal length and width by class
### BEGIN SOLUTION (~ 5 lines)

# setup the subplot grid
plt.subplot(2, 1, 2)
# plot all rows of sepal lengths against sepal widths
plt.scatter(X_petal[:,0],X_petal[:,1], label = "Petal Points", color = "r")

plt.title("Petal Lengths and Widths")
plt.xlabel("Lengths ")
plt.ylabel("Widths")
plt.legend(bbox_to_anchor = (1.25,1)) #moves the legend off the plot
plt.grid()
plt.show()
### END SOLUTION
```





Q2

- With the current plots its hard to distinguish wether or not we can draw lines to separate the classes.
- Although there seems to be clusters of data (top left of Sepal plot and bottom left of Petal plot) we should plot the same graph with colors representing the clss they fall into before concluding, after this we may be able determine the use of a clustering process or even be able to spot the lines ourself.

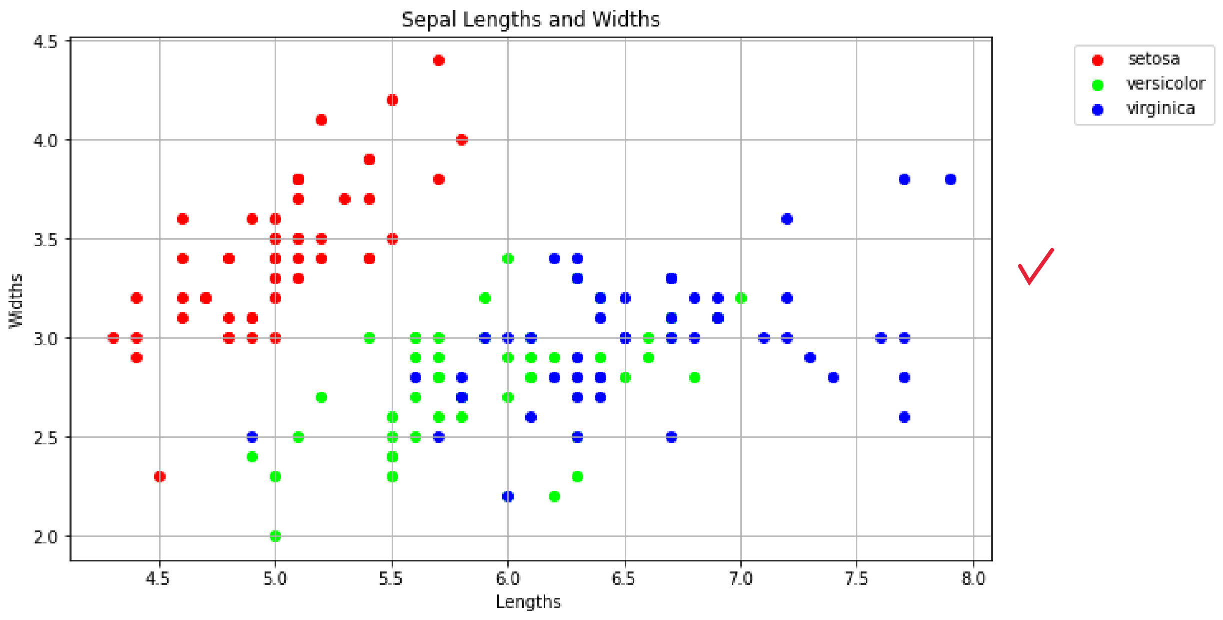
In [3]:

```
def FetchColor(yi):
    #function that takes in the class identifier
    #and returns a specific output of a color for nice colors (rbg)
    if yi == 0:
        return( (1,0,0) )
    elif yi == 1:
        return( (0,1,0) )
    else:
        return( (0,0,1) )
```

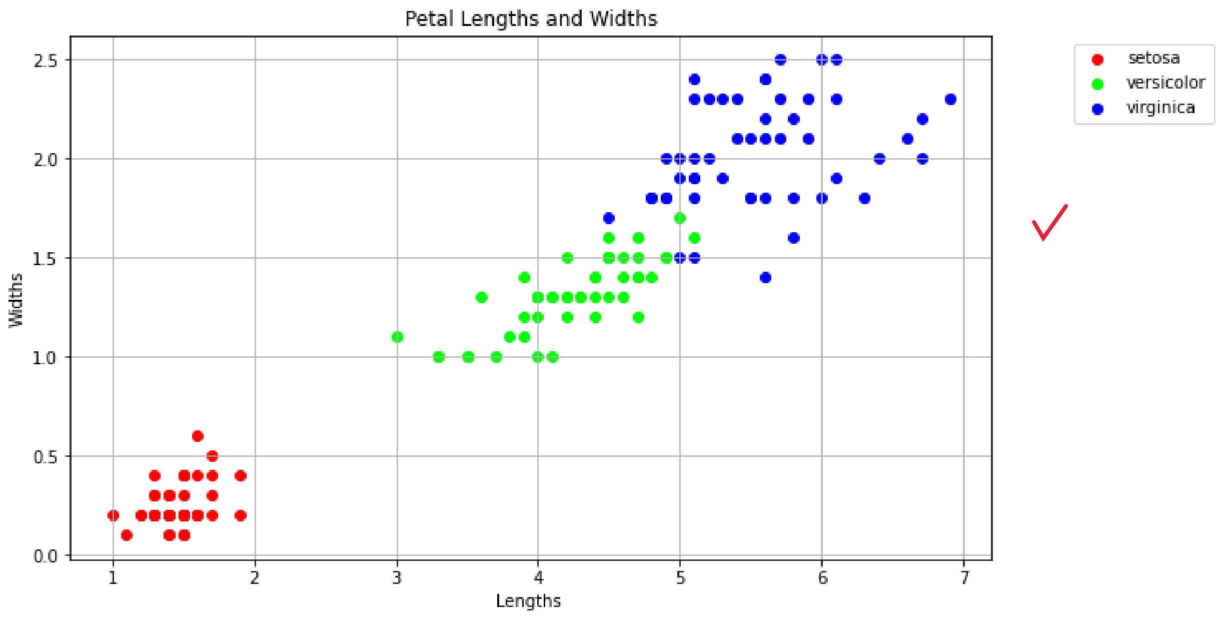
In [4]:

```
## make the same plot as before

## PRESENTATION

#increase the size of plot for presentation
plt.figure(figsize = (10,14))
#move the plots to avoid overlap of titles and xlabels for presentation
plt.subplots_adjust(hspace=0.4)


## Sepals
plt.subplot(2, 1, 1)

#define variable to count how many labels have been written
y0Count,y1Count,y2Count = 0,0,0

#iterate through all the points and set color
#based on class identifier using FetchColor func.
for i in range(len(y)):

    #if statements to ensure no more than one label get spawned
    if y[i] == 0 and y0Count == 0:
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y0Count+= 1
    elif y[i] == 1 and y1Count == 0:
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y1Count+= 1
    elif y[i] == 2 and y2Count == 0:
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y2Count+= 1
    else:
        #no label needed case
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_sepal[i,0],X_sepal[i,1], color = FetchColor(y[i]) )


##Plot Presentation
plt.title("Sepal Lengths and Widths")
plt.xlabel("Lengths ")
plt.ylabel("Widths")
plt.legend()
plt.legend(bbox_to_anchor = (1.25,1)) #moves the legend off the plot
plt.grid()

## Petals
```

```python
plt.subplot(2, 1, 2)

#define variable to count how many labels have been written
y0Count,y1Count,y2Count = 0,0,0

#iterate through all the points and set color
#based on class identifier using FetchColor func.
for i in range(len(y)):

    #if statements to ensure no more than one label get spawned
    if y[i] == 0 and y0Count == 0:
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y0Count+= 1
    elif y[i] == 1 and y1Count == 0:
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y1Count+= 1
    elif y[i] == 2 and y2Count == 0:
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y2Count+= 1
    else:
        #no label needed case
        # plot all rows of sepal lengths against sepal widths
        plt.scatter(X_petal[i,0],X_petal[i,1], color = FetchColor(y[i]) )

##Plot Presentation
plt.title("Petal Lengths and Widths")
plt.xlabel("Lengths ")
plt.ylabel("Widths")
plt.legend()
plt.legend(bbox_to_anchor = (1.25,1)) #moves the legend off the plot
plt.grid()
```



✓



✓

Now we have the improved plots with the colors depending on the classes, we can clearly see we will be able to separate Setosa from Versicolor and Virginica for the Petal and Sepal Lengths and Widths plots. ✓

However, it seems more complicated to draw the second line in order to seperate Versicolor and Virginica for both the Sepal and Petal Lengths and Widths, due to there being no straight line ✓ that gives 100% accuracy to that classification.

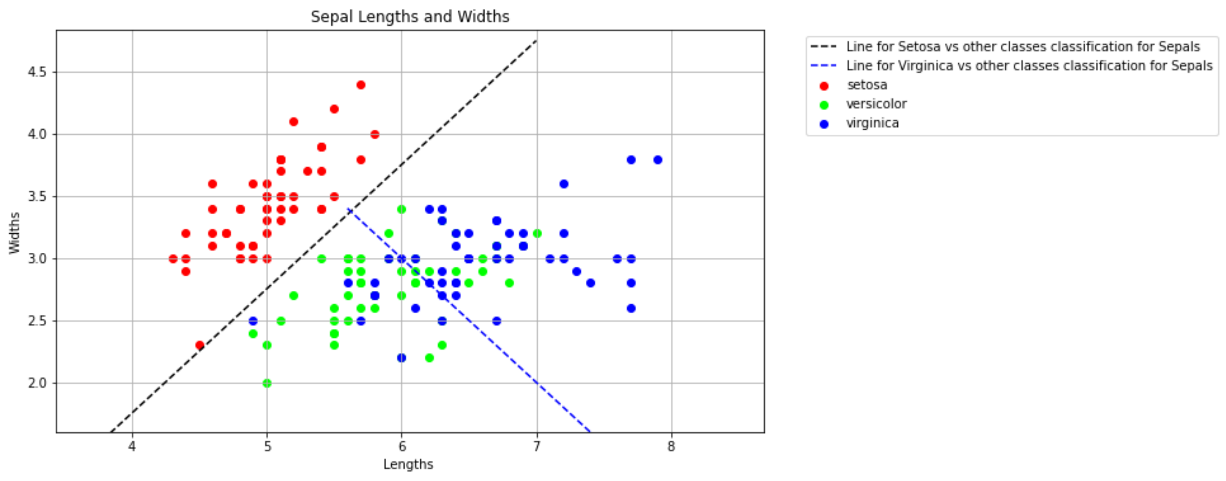
Hence, we could use established classification processes to find the **'best'** line that would separate Setosa and the other two types, but due to the simplicity of the problem we can just draw a line to separate Setosa from the other two classes of Iris, and place our best guess for the other classes as we havent been explicitly asked to do otherwise.

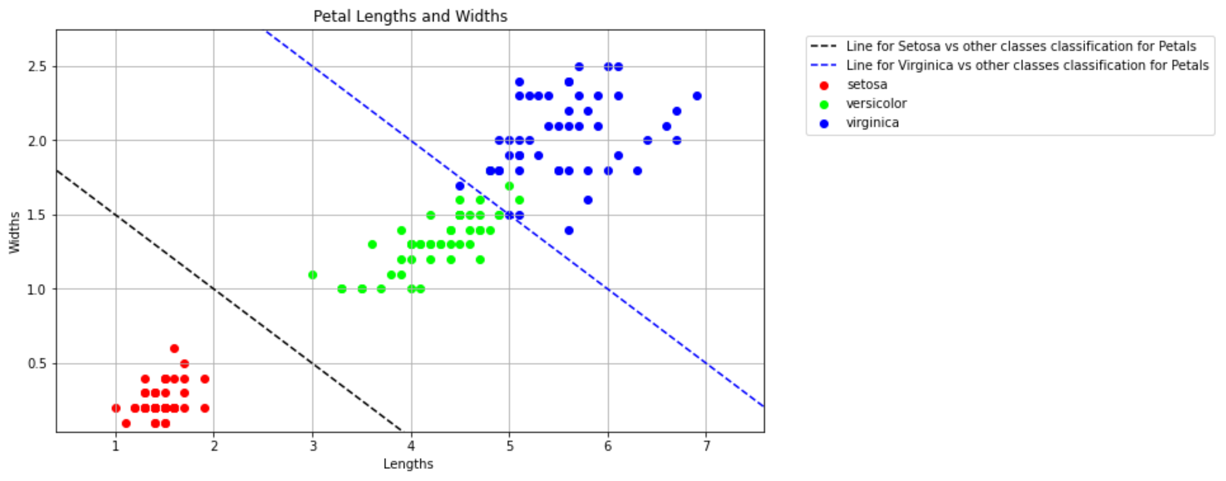
Once again, we **cannot** draw a line that would give **100%** accuracy for classfying between Versicolor and Viginica due to the points being spread in one large cluster of the two classes in both the petal and sepal plots.

In [5]:

```
## make the same plot as before

## PRESENTATION

#increase the size of plot for presentation
plt.figure(figsize = (10,14))
#move the plots to avoid overlap of titles and xlabels for presentation
plt.subplots_adjust(hspace=0.4)


## Sepals
plt.subplot(2, 1, 1)

#define variable to count how many labels have been written
y0Count,y1Count,y2Count = 0,0,0

#iterate through all the points and set color
#ased on class identifier using FetchColor func.
for i in range(len(y)):

    #if statements to ensure no more than one label get spawned
     # plot all rows of sepal lengths against sepal widths
    if y[i] == 0 and y0Count == 0:
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y0Count+= 1
    elif y[i] == 1 and y1Count == 0:
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y1Count+= 1
    elif y[i] == 2 and y2Count == 0:
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y2Count+= 1
    else:
        #no label needed case
        plt.scatter(X_sepal[i,0],X_sepal[i,1],
                    color = FetchColor(y[i]) )


#define the line that separates sepal Setosa from teh other classes
x2 = np.arange(0,max(X_sepal[:,0]))
x3 = np.arange(5.6,max(X_sepal[:,0]))
plt.plot(x2,-2.25+ x2, color = "k",
         label = "Line for Setosa vs other classes classification for Sepals",
         linestyle = "--")
plt.plot(x3,+9 - x3, color = "b",
         label = "Line for Virginica vs other classes classification for Sepals",
         linestyle = "--")

##Plot Presentation
plt.title("Sepal Lengths and Widths")
plt.xlabel("Lengths ")
plt.ylabel("Widths")
plt.legend(bbox_to_anchor = (1.05,1)) #moves the legend off the plot
plt.grid()
#produce a nicer domain for the plot, (keeps it similar the the previous plots)
plt.xlim([min(X_sepal[:,0])*0.8,max(X_sepal[:,0])*1.1])
plt.ylim([min(X_sepal[:,1])*0.8,max(X_sepal[:,1])*1.1])

## Petals
plt.subplot(2, 1, 2)
```

```
#define variable to count how many labels have been written
y0Count,y1Count,y2Count = 0,0,0

#iterate through all the points and set color
#based on class identifier using FetchColor func.
for i in range(len(y)):

    #if statements to ensure no more than one label get spawned
    # plot all rows of sepal lengths against sepal widths
    if y[i] == 0 and y0Count == 0:
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y0Count+= 1
    elif y[i] == 1 and y1Count == 0:
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y1Count+= 1
    elif y[i] == 2 and y2Count == 0:
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    label = names[y[i]], color = FetchColor(y[i]) )
        y2Count+= 1
    else:
        #no label needed case
        plt.scatter(X_petal[i,0],X_petal[i,1],
                    color = FetchColor(y[i]) )


#produce the line htat separates the Setosa from the other classes
x2 = np.arange(0,max(X_petal[:,0]+5))
plt.plot(x2,2-0.5*x2, color = "k",
         label = "Line for Setosa vs other classes classification for Petals",
         linestyle = "--")
plt.plot(x2,4-0.5*x2, color = "b",
         label = "Line for Virginica vs other classes classification for Petals",
         linestyle = "--")

##Plot Presentation
plt.title("Petal Lengths and Widths")
plt.xlabel("Lengths ")
plt.ylabel("Widths")
plt.legend(bbox_to_anchor = (1.05,1)) #moves the legend off the plot
plt.grid()
#produce a nicer domain for the plot, (keeps it similar the the previous plots)
plt.xlim([min(X_petal[:,0])*0.4,max(X_petal[:,0])*1.1])
plt.ylim([min(X_petal[:,1])*0.4,max(X_petal[:,1])*1.1])
plt.show()
```





Once again, as I have tried to express before, the dotted black line for the Line separating Setosa and the other classes is the only class which we may plot a straight line with 100% classification accuracy, the dotted blue line is just aproximately right, as no matter where you place it, you will never have 100% accuracy due to the spread of the data.

Q3

For a Valid PDF, we must have:

(a). $1 \geq \hat{y}_k^{(i)} \geq 0$ ✓

(b). $\sum_{k=1}^{K} \hat{y}_k^{(i)} = 1$ ✓

Answering (a):

$e^x > 0, \forall x$, hence both numerator and denominator are positive. Therefore $\hat{y}_k^{(i)}$ is a fraction for any k and i and hence satisfies $\hat{y}_k^{(i)} \geq 0.$ ✓

for the smallest amount of K, when $K = 1$, we have $\hat{y}_k^{(i)} = \frac{e^{[W^T x^{(i)} + b]_1}}{e^{[W^T x^{(i)} + b]_1}} = 1$ hence satisfying the ✓ condition. □

Answering (b):

$$\sum_{k=1}^{K}\hat{y}_k^{(i)} = \sum_{k=1}^{K}\frac{e^{[W^T x^{(i)}+b]_k}}{\sum_{j=1}^{K} e^{[W^T x^{(i)}+b]_j}}$$

, by definition (2)

by noticing that the denominator can be factored out as it is equal to some constant...

$$\sum_{j=1}^{K} e^{[W^T x^{(i)}+b]_j} = a \in \mathrm{R}$$

then ...

$$\sum_{k=1}^{K}\hat{y}_k^{(i)} = \frac{\sum_{k=1}^{K} e^{[W^T x^{(i)}+b]_k}}{\sum_{j=1}^{K} e^{[W^T x^{(i)}+b]_j}} = \frac{\sum_{k=1}^{K} e^{[W^T x^{(i)}+b]_k}}{a} \checkmark$$

notice numerator is the same definition of a, and hence the same as the denominator, hence we may write...

$$\sum_{k=1}^{K}\hat{y}_k^{(i)} = \frac{\sum_{k=1}^{K} e^{[W^T x^{(i)}+b]_k}}{\sum_{j=1}^{K} e^{[W^T x^{(i)}+b]_j}} = \frac{\sum_{k=1}^{K} e^{[W^T x^{(i)}+b]_k}}{a} = \frac{a}{a} = 1 \checkmark$$

□

4a)

We know that for a single training example, $x^{(i)}$, this is has output defined from (2) as :

$$\hat{y}_k^{(i)} = \sum_{k=1}^{K}\frac{e^{[W^T x^{(i)}+b]_k}}{\sum_{j=1}^{K} e^{[W^T x^{(i)}+b]_j}} \checkmark$$

.

If we considered a one hot coding vector, $\mathbf{\hat{y}_k^{(i)}}$, this will only have 1 out of k components set to 1, and the rest set to 0, as stated in the question, (the kth component is set to 1). ✓

So the probability of predicting the true class (predicting the correct k^{th} variable to be 1), is the likelihood of observing the true label. This is lets us construct the following for **one** training example:

$$\mathrm{L}(W, b, x^{(i)}, y^{(i)}) = \mathbf{\hat{y}_k^{(i)}}$$

As $\mathbf{\hat{y}_k^{(i)}} = \left[0, 0, \ldots, 1, \ldots, 0, 0\right]$, we can say that $\hat{y}_k^{(i)} = 1$ and $\hat{y}_j^{(i)} = 0, \forall j \neq k,$ and $j \in \{1, \ldots, k, \ldots, n\}$ for $j, n \in \mathrm{N}$ and $n \geq k$

This notation allows us to write the general notation as:

$$\mathrm{L}(W, b, x^{(i)}, y^{(i)}) = \Pi_{j=1}^{K}(\hat{y}_j^{(i)})^{y_j^{(i)}} \checkmark$$

.

Explanation:

if the exponent, $y_j^{(i)} = 1$, that is, if the class b is the TRUE class, then clearly $(\hat{y}_j^{(i)})^{y_j^{(i)}} = (\hat{y}_j^{(i)})^1 = (\hat{y}_j^{(i)})$. ✓

However if the exponent, $y_j^{(i)} = 0$, that is, if the class b is the WRONG class, then the likelihood dissolves to 1, $(\hat{y}_j^{(i)})^0 = 1$. ✓

This leaves us with:

$$\mathrm{l}(W, b, x^{(i)}, y^{(i)}) = \log(\mathrm{L}(W, b, x^{(i)}, y^{(i)}))$$

$$\log(\Pi_{j=1}^{K}(\hat{y}_j^{(i)})^{y_j^{(i)}}) = \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$ ✓

.

In this line $j \in \{1, \ldots, K\}$ including $j = k$!!

Finally we may take into account all i = 1,...,m indipendent training samples the new log likelyhood equation is: ✓

$$\mathrm{l}(W, b, x, y) = \sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$ ✓

For a given Maximum Likelyhood equation problem, recorgnise that by maximising the likelyhood, the same problem can be expressed as minimising a loss fucntion, hence we negate the and produce the following: (see 21st page of pdf lecture notes top paragraphs as this follows the same explanation aim, exact quote to explain this is "The difference in the negative sign is because usually when training a learning algorithm we want to make probabilities large whereas in logistic regression we want to minimise the loss function. ")

$$\mathrm{J}(W, b) = -\sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$ ✓

and finally taking an average over the m training samples leaves us with the final result in the question:

$$\mathrm{J}(W, b) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$ ✓

where $y_j^{(i)}$ is the hot vector evaluated at class j on the ith training sample and $\hat{y}_j^{(i)}$ is the predicted probability that the ith training sample belongs to the jth class

4b)

(3) is a good choice for a couple of reasons: a) when K = 2 and m = 1, then the equation reduces to a binary cross entropy loss function (seen similar forms of cross entropy loss in ML2 Lab 4 or ✓ page 17 (20th page overall) equation (3) from the pdf lecture notes of ML2).

A very useful property (which i will label (*) for now as I use it below) that follows form this binary classification problem is that: $\hat{y}_1 = 1 - \hat{y}_2$. (*). (This follows from Q3 (b)) ✓

This function has already been defined for us as:

$$\mathrm{L}(\hat{y}, y) = -[y\log(\hat{y}) + (1-y)\log(1-\hat{y})]$$ ✓

which can be seen quickly follows from

$$\mathrm{J}(W, b) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_j^{(i)}\log(\hat{y}_j^{(i)})$$ ✓

as, Let $K = 2$:

$$\Longrightarrow \mathrm{J}(W, b) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{b=1}^{2} y_b^{(i)}\log(\hat{y}_b^{(i)})$$ ✓

$$= -\frac{1}{m}\sum_{i=1}^{m}(y_1^{(i)}\log(\hat{y}_1^{(i)}) + y_2^{(i)}\log(\hat{y}_2^{(i)}))$$

then Let $m = 1$:

$$\Longrightarrow \mathrm{J}(W, b) = -[y_1^{(1)}\log(\hat{y}_1^{(1)}) + y_2^{(1)}\log(\hat{y}_2^{(1)})]$$ ✓

remembering that $y_1^{(1)}$ is a hot vector with subscript value being the vector evaluated at that subscript, and $y_1^{(1)}$ takes the values of either 0 or 1 if the correct classification has taken place or not, this leaves us with two cases:

Case 1:

$$y_1^{(1)} = 1$$

and hence

$$y_2^{(1)} = 0$$

$$\Longrightarrow \mathrm{J}(W, b) = -[y_1^{(1)}\log(\hat{y}_1^{(1)}) + y_2^{(1)}\log(\hat{y}_2^{(1)})]$$

$$= -[1\log(\hat{y}_1^{(1)}) + 0\log(\hat{y}_2^{(1)})] = -y_1^{(1)}\log(\hat{y}_1^{(1)})$$ ✓

or Case 2:

$$y_1^{(1)} = 0$$

and hence

$$y_2^{(1)} = 1$$

$$\Longrightarrow \mathrm{J}(W, b) = -[y_1^{(1)}\log(\hat{y}_1^{(1)}) + y_2^{(1)}\log(\hat{y}_2^{(1)})]$$ ✓

$$= -[0\log(\hat{y}_1^{(1)}) + 1\log(\hat{y}_2^{(1)})] = -y_2^{(1)}\log(\hat{y}_2^{(1)})$$

Note how by (*) we have:

$$-y_2^{(1)}\log(\hat{y}_2^{(1)}) = -y_2^{(1)}\log(1-\hat{y}_1^{(1)})$$

Finally combining leaves us with

$$-[y_1^{(1)}\log(\hat{y}_1^{(1)}) + y_2^{(1)}\log(1-\hat{y}_1^{(1)})] \quad ✓$$

which is by definition what $L(\hat{y}, y)$ was.

It is good because:

When we assume the predicted probability is wrong in classifying the correct class,$y_1^{(1)} = 1$ and $\hat{y}_1^{(1)} = 0 \implies L(\hat{y}, y) = -y_1^{(1)}\log(\hat{y}_1^{(1)})$, then the term, $\log(\hat{y}_1^{(1)})$ is very large (in magnitude) which implies a very large cost. Thus, this penalizes harshly for misclassifications. ✓

By the same logic, correct classifications result in very small cost when compared to the misclasification.

$y_1^{(1)} = 1$ and $\hat{y}_1^{(1)} = 1 \implies L(\hat{y}, y) = -y_1^{(1)}\log(\hat{y}_1^{(1)}) = -\log(1)$ ✓

then finally this leads us to the statment that the correct classification is smaller cost than the wrong classification as:

$$|-\log(1)| < |-\log(0)|$$
$$|0| < |\infty| \quad ✗$$

c) It is also a good choice as it is a convex function, implying that there is not a possibility that the loss function could converge to an alternate minima than the global minima. ✓

5)

In [6]:
```python
def p_model(X,W,b):
    """
    Compute the probabilities in equation (2)
    Arguments:
    X - data of size (num_meas, number of examples)
    W - weights, a numpy array of size (num_meas, k)
    b - bias, a numpy array of size (k,1)
    Return:
    y_hat - vector of probabilities
    """
    ### BEGIN SOLUTION (~ 3 lines)

    b2 = b.reshape(3,1)#format the b vector correctly ✓
    z= np.dot(W.T,X.T) + b2 # intermediate step for ease
     # apply the formula as defined
    y_hat = np.exp(z)/ np.sum(np.exp(z),axis = 0)  ✓
    return(y_hat)
```

In [7]:
```python
# Test your code
W_t = np.array([[0.976104, 0.0604506, 0.135916],
                [0.310450, 0.051029, 0.23037],
                [0.298401, 0.276497, 0.465858],
                  [0.828231, 0.877787, 0.23206]])
b_t = np.array([0.964628, 0.810731, 0.602419])
y_t = p_model(X, W_t, b_t)
print(np.mean(y_t,axis=1))
# it should return [0.99276867 0.00229554 0.0049358]
```

```
[0.99276867 0.00229554 0.0049358 ] ✓
```

6)

Before we Start this I want to define a few things for clarity.

As there are three classes to choose from, 'setosa' 'versicolor' 'virginica', this implies K = 3

In [8]:
```
def cost(y,y_hat):
    """
    Compute the cost function in equation (3)
    Arguments:
    y - true "label" (containing 0 if setosa, 1 if Versicolor or 2 if␣
    ↪Virginica)
    y_hat - vector containing the probabilities of a flower with measurements x␣
    ↪being in each class
    Return:
    tot_cost - value of the cost function
    """



    ### BEGIN SOLUTION (~ 4 lines)
    # Compute one-hot encode associated with the true labels y
    k = y_hat.shape[1] # a general way incase k changes later from 3
    m = y_hat.shape[0] # m is the number of samples per class (150)

    yHot = np.zeros((m,k)) # the target matrix to be filled

    #For loop version

    for i in range(m):
        # for the ith element, the class class collumn is set to 1
        yHot[i, y[i]] = 1


    # Compute the loss

    # only need to sum over m, k is "in" the dimentions of the two vectors.
    tot_cost = -(1/m) *  np.sum( yHot * np.log(y_hat))


    ### ENd SOLUTION

    return (tot_cost)
```

Q7)

Lets start by being clear about some definitions again:

1. The Cost Fucntion, $\mathrm{J}(W, b)$ took the form of

$$\mathrm{J}(W, j) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$ ✓

2. Estimator, $\hat{y}_k^{(i)}$, took the form of

$$\hat{y}_k^{(i)} = \frac{e^{W_k^T x^{(i)} + b_k}}{\sum_j^K e^{W_k^T x^{(i)} + b_j}}$$

To get to either the derivative of loss fucntion w.r.t. W_k or b_k, we first have to take derivative of Loss Function w.r.t. $\hat{y}_k^{(i)}$...

$$\frac{\partial \mathrm{J}}{\partial W} = \frac{\partial \mathrm{J}}{\partial \hat{y}_k^{(i)}} \frac{\partial \hat{y}_k^{(i)}}{\partial W}$$ ✓

and

$$\frac{\partial \mathrm{J}}{\partial b_k} = \frac{\partial \mathrm{J}}{\partial \hat{y}_k^{(i)}} \frac{\partial \hat{y}_k^{(i)}}{\partial b_k}$$ ✓

by chain rule for both cases.

So we calculate the following..

$$\frac{\partial \mathrm{J}}{\partial \hat{y}_k^{(i)}} = \frac{\partial}{\partial \hat{y}_k^{(i)}} \left(-\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)}) \right)$$

Now here we can implement a very important step to notice that if we take the derivative with respect to $\hat{y}_k^{(i)}$, as we are dealing with the k^{th} case of the variable we can expand the sumnation and notice that the only term that doesnt turn to zero is the $\hat{y}_k^{(i)\,th}$.

Here's an example: Let's take K = 2 and k = 2, m = 1. ✓

The brackets expand to

$$\frac{\partial \mathrm{J}}{\partial \hat{y}_2^{(i)}} = \frac{\partial}{\partial \hat{y}_2^{(i)}} \left(-\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{2} y_j^{(i)} \log(\hat{y}_j^{(i)}) \right)$$

$$= \frac{\partial}{\partial \hat{y}_2^{(1)}} \left(y_1^{(1)} \log(\hat{y}_1^{(1)}) + y_2^{(1)} \log(\hat{y}_2^{(1)}) \right)$$

which clearly is just

$$= \frac{\partial}{\partial \hat{y}_2^{(1)}} \left(y_2^{(1)} \log(\hat{y}_2^{(1)}) \right)$$

as

$$\frac{\partial}{\partial \hat{y}_2^{(1)}} \left(y_1^{(1)} \log(\hat{y}_1^{(1)}) \right) = 0$$

due to no terms being related to the derivative

Hence this gives us:

$$\frac{\partial \mathrm{J}}{\partial \hat{y}_k^{(i)}} = \frac{\partial}{\partial \hat{y}_k^{(i)}} \left(-\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)}) \right)$$

$$= -\frac{1}{m} \sum_{i=1}^{m} \frac{y_k^{(i)}}{(\hat{y}_k^{(i)})} \quad \checkmark$$

The next step is to find

$$\frac{\partial \hat{y}_k^{(i)}}{\partial W_k} \text{ and } \frac{\partial \hat{y}_k^{(i)}}{\partial b_k}$$

Remember, 2. Estimator, $\hat{y}_k^{(i)}$, took the form of

$$\hat{y}_k^{(i)} = \frac{e^{W_k^T x^{(i)} + b_k}}{\sum_j^K e^{W_j^T x^{(i)} + b_j}}$$

Via Quotient Rule,

$$\frac{d}{dx}\frac{u}{v} = \frac{(v\frac{du}{dx}) - u\frac{dv}{dx}}{v^2} \quad \checkmark$$

This leaves

$$\frac{\partial \hat{y}_k^{(i)}}{\partial W_k} = \frac{(\sum_j^K e^{W_j^T x^{(i)} + b_j})(x^{(i)} e^{W_k^T x^{(i)} + b_k}) - (e^{W_k^T x^{(i)} + b_k})(x^{(i)} e^{W_k^T x^{(i)} + b_k})}{\sum_j^K (e^{W_j^T x^{(i)} + b_j})^2} \quad \checkmark$$

$$\Rightarrow \frac{d\hat{y}_k^{(i)}}{dW_k} = x^{(i)} \frac{e^{W_k^T x^{(i)} + b_k}}{\sum_j^K (e^{W_j^T x^{(i)} + b_j})} \frac{(\sum_j^K e^{W_j^T x^{(i)} + b_j}) - (e^{W_k^T x^{(i)} + b_k})}{\sum_j^K (e^{W_j^T x^{(i)} + b_j})} \quad \checkmark$$

$$= x^{(i)} \frac{e^{W_k^T x^{(i)} + b_k}}{\sum_j^K (e^{W_j^T x^{(i)} + b_j})} (1 - \frac{(e^{W_k^T x^{(i)} + b_k})}{\sum_j^K (e^{W_j^T x^{(i)} + b_j})}) \quad \checkmark$$

and by using def of $\hat{y}_k^{(i)}$ as I've stated many time in this question, this boils down to:

$$\frac{d\hat{y}_k^{(i)}}{dW_k} = x^{(i)} \hat{y}_k^{(i)} (1 - \hat{y}_k^{(i)}) \quad \checkmark$$

With little effort, we can state

$$\frac{d\hat{y}_k^{(i)}}{db_k} = \hat{y}_k^{(i)} (1 - \hat{y}_k^{(i)})$$

as the only reason the factor of $x^{(i)}$ was there in the previous derivation of $\frac{d\hat{y}_k^{(i)}}{dW}$ was due to the chain rule from the $e^{W^T x^{(i)}} + b$ term, however in the case of $\frac{d\hat{y}_k^{(i)}}{db}$ this would return 1 not $x^{(i)}$. ✓

Hence, replace $x^{(i)}$ with 1 and you will have the derived $\frac{d\hat{y}_k^{(i)}}{db}$, hence the answer stated above.

$$\frac{dJ}{dW} = \frac{dJ}{d\hat{y}_b^{(i)}} \frac{d\hat{y}_b^{(i)}}{dW}$$

$$\implies \frac{dJ}{dW} = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} \frac{-y_j^{(i)}}{\hat{y}_j^{(i)}} x^{(i)} \hat{y}_k^{(i)} (1 - \hat{y}_k^{(i)})$$

✗ So for $y_k = 1$ this simplifies to

$$\frac{1}{m}\sum_{i=1}^{m} x^{(i)} (\hat{y}_k^{(i)} - 1)$$

✗ And for $y_k = 0$ this simplifies to

$$\frac{1}{m}\sum_{i=1}^{m} x^{(i)} \hat{y}_k^{(i)}$$

combining the two cases and writing this in matrix form gives us:

$$\frac{dJ}{dW} = \frac{1}{m}(\hat{Y} - Y)X^T$$

This is due to the combined term of $y_j^{(i)} x^{(i)} (\hat{y}_j^{(i)} - 1)$ representing the difference between the predicted and the true label, which is the same as represented by $(\hat{Y} - Y)$, over the entire training sample. Whilst also containing the required feature contribution from $x^{(i)}$ is still held by replacing with the matrix version of X^T and finally as we are generalising over the entire training sample region, we can take the average w.r.t. m samples. Hence the $\frac{1}{m}$ term. ✗

Finally, by similar inspection

$$\frac{dJ}{db} = \frac{1}{m}(Y - \hat{Y}) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - \mathbf{y^{(i)}})$$

Once again, as we have hown the derivative of the cost funciton w.r.t. b does not contain the feature input from chain rule, hence the same as above applies to this derivative without the matrix X, once again we note that the derivative of the cost function w.r.t. b depends on the difference between the predicted probability of the label and the true label for all terms in the training set and averaged over the range too hence $\frac{1}{m}$ term and the sumnation over m as we are using vector notation here. ✓

Q8

In [9]:

```
def grad_cost(X, y, W, b):
    """
    Compute the gradient of the cost function
    Arguments:
    X - data of size (num_meas, number of examples)
    y - true "label" (containing 0 if setosa, 1 if Versicolor or 2 if␣
↪Virginica)
    W - weights, a numpy array of size (num_meas, k)
    b - bias, a numpy array of size (k,1)
    Return:
    W_grad - gradient cost function w.r.t. W
    b_grad - gradient cost function w.r.t. b
    """
    ### BEGIN SOLUTION (~ 6 lines)
    # Compute the probabilities
    yHat = p_model(X,W,b)

    #same as cost function def of yHOt
    k = 3
    m = y.shape[0] # m is the number of samples per class (150)  ✓
    yHot = np.zeros((k,m)) # the target matrix to be filled

    #Vectorised version
    mVector = np.arange(m)
    yHot[y,mVector] = 1  ✓

    #intermediate definition
    differenceInY = yHat - yHot  ✓

    # Compute final gradient
    W_grad = (1/m)* np.dot(X.T,differenceInY.T)  ✓

    b_grad = (1/m)* np.sum(differenceInY, axis= 1)  ✓
    ### END SOLUTION
    return W_grad, b_grad
```

Q9

In [10]:

```
def train(X, y, lr, iters, W,b):
    """
    Builds the model by calling the function implemented previously
    Arguments:
    X - data of size (num_meas, number of examples)
    y - true "label" (containing 0 if setosa, 1 if Versicolor or 2 if␣
↪Virginica)
    lr - learning rate of the gradient descent update rule
    iters - number of iterations of the optimisation loop
    W - initialisation for weights, a numpy array of size (num_meas, k)
    b - initialisation for biases, a numpy array of size (k,1)
    Return:
    W - learned weights
    b - learned biases
    costs - array containing the values of the cost function
    """
    # Empty list to store cost function values
    costs = []
    # Training loop
    for iter in range(iters):
        ### BEGIN SOLUTION (~ 6 lines)
        # prediction again
        yHat = p_model(X,W,b)

        # Updating the parameters
        # finding gradient
        W_grad,b_grad = grad_cost(X,y,W,b)
        #updating the params now
        W = W - lr* W_grad
        b = b - lr* b_grad

        # Append cost function value
        costs.append(cost(y,yHat)) #do in one line


        ### END SOLUTION

        # Printing out the loss at every 100th iteration
        ### BEGIN SOLUTION (~ 2 lines)

        if iter % 100 == 0 and iter >0:
            print(str(iter)+'th iteration of loss is :' +
                  str(round(costs[iter],2)) + " (3.s.f)")
    #indexs = np.arange(0,iters,100)
    #print([costs[i] for i in indexs])

    ### END SOLUTION
    return W, b, costs
```

Q10 a

In [11]:
```
#dimention defs
mSubSet = X_petal.shape[1] # only 2 as only lenght and width
mAll = X.shape[1] # should be 4
```

In [12]:
```
k = 3
# question defs
lr = 0.2
iters = 20000
```

In [13]:
```
# (1)
# define W and b for petal only
WPetal = np.random.randn(mSubSet, k)
bPetal = np.random.randn(k)
# run the trian for petal only
WPetalOptimal, bPetalOptimal, costsPetal = train(X_petal, y, lr, iters, WPetal, bPet
```

```
100th iteration of loss is :1.45 (3.s.f)
200th iteration of loss is :1.91 (3.s.f)
300th iteration of loss is :2.26 (3.s.f)
400th iteration of loss is :2.56 (3.s.f)
500th iteration of loss is :2.81 (3.s.f)
600th iteration of loss is :3.04 (3.s.f)
700th iteration of loss is :3.25 (3.s.f)
800th iteration of loss is :3.44 (3.s.f)
900th iteration of loss is :3.61 (3.s.f)
1000th iteration of loss is :3.78 (3.s.f)
1100th iteration of loss is :3.93 (3.s.f)
1200th iteration of loss is :4.08 (3.s.f)
1300th iteration of loss is :4.21 (3.s.f)
1400th iteration of loss is :4.34 (3.s.f)
```

```
1500th iteration of loss is :4.47 (3.s.f)
1600th iteration of loss is :4.59 (3.s.f)
1700th iteration of loss is :4.7 (3.s.f)
1800th iteration of loss is :4.81 (3.s.f)
1900th iteration of loss is :4.91 (3.s.f)
2000th iteration of loss is :5.01 (3.s.f)
2100th iteration of loss is :5.11 (3.s.f)
2200th iteration of loss is :5.2 (3.s.f)
2300th iteration of loss is :5.29 (3.s.f)
2400th iteration of loss is :5.38 (3.s.f)
2500th iteration of loss is :5.46 (3.s.f)
2600th iteration of loss is :5.54 (3.s.f)
2700th iteration of loss is :5.62 (3.s.f)
2800th iteration of loss is :5.7 (3.s.f)
2900th iteration of loss is :5.77 (3.s.f)
3000th iteration of loss is :5.84 (3.s.f)
3100th iteration of loss is :5.91 (3.s.f)
3200th iteration of loss is :5.98 (3.s.f)
3300th iteration of loss is :6.05 (3.s.f)
3400th iteration of loss is :6.12 (3.s.f)
3500th iteration of loss is :6.18 (3.s.f)
3600th iteration of loss is :6.24 (3.s.f)
3700th iteration of loss is :6.3 (3.s.f)
3800th iteration of loss is :6.36 (3.s.f)
3900th iteration of loss is :6.42 (3.s.f)
4000th iteration of loss is :6.48 (3.s.f)
4100th iteration of loss is :6.54 (3.s.f)
4200th iteration of loss is :6.59 (3.s.f)
4300th iteration of loss is :6.64 (3.s.f)
4400th iteration of loss is :6.7 (3.s.f)
4500th iteration of loss is :6.75 (3.s.f)
4600th iteration of loss is :6.8 (3.s.f)
4700th iteration of loss is :6.85 (3.s.f)
4800th iteration of loss is :6.9 (3.s.f)
4900th iteration of loss is :6.95 (3.s.f)
5000th iteration of loss is :7.0 (3.s.f)
5100th iteration of loss is :7.04 (3.s.f)
5200th iteration of loss is :7.09 (3.s.f)
5300th iteration of loss is :7.13 (3.s.f)
5400th iteration of loss is :7.18 (3.s.f)
5500th iteration of loss is :7.22 (3.s.f)
5600th iteration of loss is :7.27 (3.s.f)
5700th iteration of loss is :7.31 (3.s.f)
5800th iteration of loss is :7.35 (3.s.f)
5900th iteration of loss is :7.39 (3.s.f)
6000th iteration of loss is :7.43 (3.s.f)
6100th iteration of loss is :7.47 (3.s.f)
6200th iteration of loss is :7.51 (3.s.f)
6300th iteration of loss is :7.55 (3.s.f)
6400th iteration of loss is :7.59 (3.s.f)
6500th iteration of loss is :7.63 (3.s.f)
6600th iteration of loss is :7.67 (3.s.f)
6700th iteration of loss is :7.71 (3.s.f)
6800th iteration of loss is :7.74 (3.s.f)
6900th iteration of loss is :7.78 (3.s.f)
7000th iteration of loss is :7.82 (3.s.f)
7100th iteration of loss is :7.85 (3.s.f)
7200th iteration of loss is :7.89 (3.s.f)
7300th iteration of loss is :7.92 (3.s.f)
7400th iteration of loss is :7.96 (3.s.f)
7500th iteration of loss is :7.99 (3.s.f)
7600th iteration of loss is :8.02 (3.s.f)
7700th iteration of loss is :8.06 (3.s.f)
7800th iteration of loss is :8.09 (3.s.f)
```

```
7900th iteration of loss is :8.12 (3.s.f)
8000th iteration of loss is :8.15 (3.s.f)
8100th iteration of loss is :8.18 (3.s.f)
8200th iteration of loss is :8.22 (3.s.f)
8300th iteration of loss is :8.25 (3.s.f)
8400th iteration of loss is :8.28 (3.s.f)
8500th iteration of loss is :8.31 (3.s.f)
8600th iteration of loss is :8.34 (3.s.f)
8700th iteration of loss is :8.37 (3.s.f)
8800th iteration of loss is :8.4 (3.s.f)
8900th iteration of loss is :8.43 (3.s.f)
9000th iteration of loss is :8.46 (3.s.f)
9100th iteration of loss is :8.49 (3.s.f)
9200th iteration of loss is :8.51 (3.s.f)
9300th iteration of loss is :8.54 (3.s.f)
9400th iteration of loss is :8.57 (3.s.f)
9500th iteration of loss is :8.6 (3.s.f)
9600th iteration of loss is :8.63 (3.s.f)
9700th iteration of loss is :8.65 (3.s.f)
9800th iteration of loss is :8.68 (3.s.f)
9900th iteration of loss is :8.71 (3.s.f)
10000th iteration of loss is :8.73 (3.s.f)
10100th iteration of loss is :8.76 (3.s.f)
10200th iteration of loss is :8.79 (3.s.f)
10300th iteration of loss is :8.81 (3.s.f)
10400th iteration of loss is :8.84 (3.s.f)
10500th iteration of loss is :8.86 (3.s.f)
10600th iteration of loss is :8.89 (3.s.f)
10700th iteration of loss is :8.91 (3.s.f)
10800th iteration of loss is :8.94 (3.s.f)
10900th iteration of loss is :8.96 (3.s.f)
11000th iteration of loss is :8.99 (3.s.f)
11100th iteration of loss is :9.01 (3.s.f)
11200th iteration of loss is :9.04 (3.s.f)
11300th iteration of loss is :9.06 (3.s.f)
11400th iteration of loss is :9.08 (3.s.f)
11500th iteration of loss is :9.11 (3.s.f)
11600th iteration of loss is :9.13 (3.s.f)
11700th iteration of loss is :9.15 (3.s.f)
11800th iteration of loss is :9.18 (3.s.f)
11900th iteration of loss is :9.2 (3.s.f)
12000th iteration of loss is :9.22 (3.s.f)
12100th iteration of loss is :9.24 (3.s.f)
12200th iteration of loss is :9.27 (3.s.f)
12300th iteration of loss is :9.29 (3.s.f)
12400th iteration of loss is :9.31 (3.s.f)
12500th iteration of loss is :9.33 (3.s.f)
12600th iteration of loss is :9.35 (3.s.f)
12700th iteration of loss is :9.38 (3.s.f)
12800th iteration of loss is :9.4 (3.s.f)
12900th iteration of loss is :9.42 (3.s.f)
13000th iteration of loss is :9.44 (3.s.f)
13100th iteration of loss is :9.46 (3.s.f)
13200th iteration of loss is :9.48 (3.s.f)
13300th iteration of loss is :9.5 (3.s.f)
13400th iteration of loss is :9.52 (3.s.f)
13500th iteration of loss is :9.54 (3.s.f)
13600th iteration of loss is :9.56 (3.s.f)
13700th iteration of loss is :9.58 (3.s.f)
13800th iteration of loss is :9.6 (3.s.f)
13900th iteration of loss is :9.62 (3.s.f)
14000th iteration of loss is :9.64 (3.s.f)
14100th iteration of loss is :9.66 (3.s.f)
14200th iteration of loss is :9.68 (3.s.f)
```

```
14300th iteration of loss is :9.7 (3.s.f)
14400th iteration of loss is :9.72 (3.s.f)
14500th iteration of loss is :9.74 (3.s.f)
14600th iteration of loss is :9.76 (3.s.f)
14700th iteration of loss is :9.78 (3.s.f)
14800th iteration of loss is :9.8 (3.s.f)
14900th iteration of loss is :9.82 (3.s.f)
15000th iteration of loss is :9.84 (3.s.f)
15100th iteration of loss is :9.85 (3.s.f)
15200th iteration of loss is :9.87 (3.s.f)
15300th iteration of loss is :9.89 (3.s.f)
15400th iteration of loss is :9.91 (3.s.f)
15500th iteration of loss is :9.93 (3.s.f)
15600th iteration of loss is :9.95 (3.s.f)
15700th iteration of loss is :9.96 (3.s.f)
15800th iteration of loss is :9.98 (3.s.f)
15900th iteration of loss is :10.0 (3.s.f)
16000th iteration of loss is :10.02 (3.s.f)
16100th iteration of loss is :10.03 (3.s.f)
16200th iteration of loss is :10.05 (3.s.f)
16300th iteration of loss is :10.07 (3.s.f)
16400th iteration of loss is :10.09 (3.s.f)
16500th iteration of loss is :10.1 (3.s.f)
16600th iteration of loss is :10.12 (3.s.f)
16700th iteration of loss is :10.14 (3.s.f)
16800th iteration of loss is :10.15 (3.s.f)
16900th iteration of loss is :10.17 (3.s.f)
17000th iteration of loss is :10.19 (3.s.f)
17100th iteration of loss is :10.21 (3.s.f)
17200th iteration of loss is :10.22 (3.s.f)
17300th iteration of loss is :10.24 (3.s.f)
17400th iteration of loss is :10.25 (3.s.f)
17500th iteration of loss is :10.27 (3.s.f)
17600th iteration of loss is :10.29 (3.s.f)
17700th iteration of loss is :10.3 (3.s.f)
17800th iteration of loss is :10.32 (3.s.f)
17900th iteration of loss is :10.34 (3.s.f)
18000th iteration of loss is :10.35 (3.s.f)
18100th iteration of loss is :10.37 (3.s.f)
18200th iteration of loss is :10.38 (3.s.f)
18300th iteration of loss is :10.4 (3.s.f)
18400th iteration of loss is :10.41 (3.s.f)
18500th iteration of loss is :10.43 (3.s.f)
18600th iteration of loss is :10.44 (3.s.f)
18700th iteration of loss is :10.46 (3.s.f)
18800th iteration of loss is :10.48 (3.s.f)
18900th iteration of loss is :10.49 (3.s.f)
19000th iteration of loss is :10.51 (3.s.f)
19100th iteration of loss is :10.52 (3.s.f)
19200th iteration of loss is :10.54 (3.s.f)
19300th iteration of loss is :10.55 (3.s.f)
19400th iteration of loss is :10.57 (3.s.f)
19500th iteration of loss is :10.58 (3.s.f)
19600th iteration of loss is :10.6 (3.s.f)
19700th iteration of loss is :10.61 (3.s.f)
19800th iteration of loss is :10.62 (3.s.f)
19900th iteration of loss is :10.64 (3.s.f)
```

In [14]: ✓

```python
#do the same for sepals
WSepal = np.random.randn(mSubSet, k)
bSepal = np.random.randn(k)
WSepalOptimal, bSepalOptimal, costsSepal = train(X_sepal, y, lr, iters, WSepal, bSep
100th iteration of loss is :1.97 (3.s.f)
200th iteration of loss is :2.32 (3.s.f)
300th iteration of loss is :2.59 (3.s.f)
400th iteration of loss is :2.81 (3.s.f)
500th iteration of loss is :2.99 (3.s.f)
600th iteration of loss is :3.14 (3.s.f)
700th iteration of loss is :3.28 (3.s.f)
800th iteration of loss is :3.4 (3.s.f)
900th iteration of loss is :3.51 (3.s.f)
1000th iteration of loss is :3.61 (3.s.f)
1100th iteration of loss is :3.7 (3.s.f)
1200th iteration of loss is :3.79 (3.s.f)
1300th iteration of loss is :3.86 (3.s.f)
1400th iteration of loss is :3.93 (3.s.f)
1500th iteration of loss is :4.0 (3.s.f)
1600th iteration of loss is :4.06 (3.s.f)
1700th iteration of loss is :4.12 (3.s.f)
1800th iteration of loss is :4.17 (3.s.f)
1900th iteration of loss is :4.22 (3.s.f)
2000th iteration of loss is :4.27 (3.s.f)
2100th iteration of loss is :4.32 (3.s.f)
2200th iteration of loss is :4.36 (3.s.f)
2300th iteration of loss is :4.4 (3.s.f)
2400th iteration of loss is :4.44 (3.s.f)
2500th iteration of loss is :4.48 (3.s.f)
2600th iteration of loss is :4.51 (3.s.f)
2700th iteration of loss is :4.55 (3.s.f)
2800th iteration of loss is :4.58 (3.s.f)
2900th iteration of loss is :4.61 (3.s.f)
3000th iteration of loss is :4.64 (3.s.f)
3100th iteration of loss is :4.67 (3.s.f)
3200th iteration of loss is :4.69 (3.s.f)
3300th iteration of loss is :4.72 (3.s.f)
3400th iteration of loss is :4.74 (3.s.f)
3500th iteration of loss is :4.77 (3.s.f)
3600th iteration of loss is :4.79 (3.s.f)
3700th iteration of loss is :4.81 (3.s.f)
3800th iteration of loss is :4.83 (3.s.f)
3900th iteration of loss is :4.85 (3.s.f)
4000th iteration of loss is :4.87 (3.s.f)
4100th iteration of loss is :4.89 (3.s.f)
4200th iteration of loss is :4.91 (3.s.f)
4300th iteration of loss is :4.93 (3.s.f)
4400th iteration of loss is :4.95 (3.s.f)
4500th iteration of loss is :4.97 (3.s.f)
4600th iteration of loss is :4.98 (3.s.f)
4700th iteration of loss is :5.0 (3.s.f)
4800th iteration of loss is :5.01 (3.s.f)
4900th iteration of loss is :5.03 (3.s.f)
5000th iteration of loss is :5.05 (3.s.f)
5100th iteration of loss is :5.06 (3.s.f)
5200th iteration of loss is :5.08 (3.s.f)
5300th iteration of loss is :5.09 (3.s.f)
5400th iteration of loss is :5.1 (3.s.f)
5500th iteration of loss is :5.12 (3.s.f)
5600th iteration of loss is :5.13 (3.s.f)
5700th iteration of loss is :5.14 (3.s.f)
5800th iteration of loss is :5.16 (3.s.f)
5900th iteration of loss is :5.17 (3.s.f)
6000th iteration of loss is :5.18 (3.s.f)
6100th iteration of loss is :5.19 (3.s.f)
6200th iteration of loss is :5.2 (3.s.f)
6300th iteration of loss is :5.22 (3.s.f)
6400th iteration of loss is :5.23 (3.s.f)
```

```
6500th iteration of loss is :5.24 (3.s.f)
6600th iteration of loss is :5.25 (3.s.f)
6700th iteration of loss is :5.26 (3.s.f)
6800th iteration of loss is :5.27 (3.s.f)
6900th iteration of loss is :5.28 (3.s.f)
7000th iteration of loss is :5.29 (3.s.f)
7100th iteration of loss is :5.3 (3.s.f)
7200th iteration of loss is :5.31 (3.s.f)
7300th iteration of loss is :5.32 (3.s.f)
7400th iteration of loss is :5.33 (3.s.f)
7500th iteration of loss is :5.34 (3.s.f)
7600th iteration of loss is :5.35 (3.s.f)
7700th iteration of loss is :5.35 (3.s.f)
7800th iteration of loss is :5.36 (3.s.f)
7900th iteration of loss is :5.37 (3.s.f)
8000th iteration of loss is :5.38 (3.s.f)
8100th iteration of loss is :5.39 (3.s.f)
8200th iteration of loss is :5.4 (3.s.f)
8300th iteration of loss is :5.41 (3.s.f)
8400th iteration of loss is :5.41 (3.s.f)
8500th iteration of loss is :5.42 (3.s.f)
8600th iteration of loss is :5.43 (3.s.f)
8700th iteration of loss is :5.44 (3.s.f)
8800th iteration of loss is :5.44 (3.s.f)
8900th iteration of loss is :5.45 (3.s.f)
9000th iteration of loss is :5.46 (3.s.f)
9100th iteration of loss is :5.46 (3.s.f)
9200th iteration of loss is :5.47 (3.s.f)
9300th iteration of loss is :5.48 (3.s.f)
9400th iteration of loss is :5.49 (3.s.f)
9500th iteration of loss is :5.49 (3.s.f)
9600th iteration of loss is :5.5 (3.s.f)
9700th iteration of loss is :5.51 (3.s.f)
9800th iteration of loss is :5.51 (3.s.f)
9900th iteration of loss is :5.52 (3.s.f)
10000th iteration of loss is :5.52 (3.s.f)
10100th iteration of loss is :5.53 (3.s.f)
10200th iteration of loss is :5.54 (3.s.f)
10300th iteration of loss is :5.54 (3.s.f)
10400th iteration of loss is :5.55 (3.s.f)
10500th iteration of loss is :5.55 (3.s.f)
10600th iteration of loss is :5.56 (3.s.f)
10700th iteration of loss is :5.56 (3.s.f)
10800th iteration of loss is :5.57 (3.s.f)
10900th iteration of loss is :5.57 (3.s.f)
11000th iteration of loss is :5.58 (3.s.f)
11100th iteration of loss is :5.59 (3.s.f)
11200th iteration of loss is :5.59 (3.s.f)
11300th iteration of loss is :5.6 (3.s.f)
11400th iteration of loss is :5.6 (3.s.f)
11500th iteration of loss is :5.61 (3.s.f)
11600th iteration of loss is :5.61 (3.s.f)
11700th iteration of loss is :5.61 (3.s.f)
11800th iteration of loss is :5.62 (3.s.f)
11900th iteration of loss is :5.62 (3.s.f)
12000th iteration of loss is :5.63 (3.s.f)
12100th iteration of loss is :5.63 (3.s.f)
12200th iteration of loss is :5.64 (3.s.f)
12300th iteration of loss is :5.64 (3.s.f)
12400th iteration of loss is :5.65 (3.s.f)
12500th iteration of loss is :5.65 (3.s.f)
12600th iteration of loss is :5.65 (3.s.f)
12700th iteration of loss is :5.66 (3.s.f)
12800th iteration of loss is :5.66 (3.s.f)
```

```
12900th iteration of loss is :5.67 (3.s.f)
13000th iteration of loss is :5.67 (3.s.f)
13100th iteration of loss is :5.67 (3.s.f)
13200th iteration of loss is :5.68 (3.s.f)
13300th iteration of loss is :5.68 (3.s.f)
13400th iteration of loss is :5.69 (3.s.f)
13500th iteration of loss is :5.69 (3.s.f)
13600th iteration of loss is :5.69 (3.s.f)
13700th iteration of loss is :5.7 (3.s.f)
13800th iteration of loss is :5.7 (3.s.f)
13900th iteration of loss is :5.7 (3.s.f)
14000th iteration of loss is :5.71 (3.s.f)
14100th iteration of loss is :5.71 (3.s.f)
14200th iteration of loss is :5.71 (3.s.f)
14300th iteration of loss is :5.72 (3.s.f)
14400th iteration of loss is :5.72 (3.s.f)
14500th iteration of loss is :5.72 (3.s.f)
14600th iteration of loss is :5.73 (3.s.f)
14700th iteration of loss is :5.73 (3.s.f)
14800th iteration of loss is :5.73 (3.s.f)
14900th iteration of loss is :5.74 (3.s.f)
15000th iteration of loss is :5.74 (3.s.f)
15100th iteration of loss is :5.74 (3.s.f)
15200th iteration of loss is :5.75 (3.s.f)
15300th iteration of loss is :5.75 (3.s.f)
15400th iteration of loss is :5.75 (3.s.f)
15500th iteration of loss is :5.75 (3.s.f)
15600th iteration of loss is :5.76 (3.s.f)
15700th iteration of loss is :5.76 (3.s.f)
15800th iteration of loss is :5.76 (3.s.f)
15900th iteration of loss is :5.77 (3.s.f)
16000th iteration of loss is :5.77 (3.s.f)
16100th iteration of loss is :5.77 (3.s.f)
16200th iteration of loss is :5.77 (3.s.f)
16300th iteration of loss is :5.78 (3.s.f)
16400th iteration of loss is :5.78 (3.s.f)
16500th iteration of loss is :5.78 (3.s.f)
16600th iteration of loss is :5.78 (3.s.f)
16700th iteration of loss is :5.79 (3.s.f)
16800th iteration of loss is :5.79 (3.s.f)
16900th iteration of loss is :5.79 (3.s.f)
17000th iteration of loss is :5.79 (3.s.f)
17100th iteration of loss is :5.8 (3.s.f)
17200th iteration of loss is :5.8 (3.s.f)
17300th iteration of loss is :5.8 (3.s.f)
17400th iteration of loss is :5.8 (3.s.f)
17500th iteration of loss is :5.8 (3.s.f)
17600th iteration of loss is :5.81 (3.s.f)
17700th iteration of loss is :5.81 (3.s.f)
17800th iteration of loss is :5.81 (3.s.f)
17900th iteration of loss is :5.81 (3.s.f)
18000th iteration of loss is :5.82 (3.s.f)
18100th iteration of loss is :5.82 (3.s.f)
18200th iteration of loss is :5.82 (3.s.f)
18300th iteration of loss is :5.82 (3.s.f)
18400th iteration of loss is :5.82 (3.s.f)
18500th iteration of loss is :5.83 (3.s.f)
18600th iteration of loss is :5.83 (3.s.f)
18700th iteration of loss is :5.83 (3.s.f)
18800th iteration of loss is :5.83 (3.s.f)
18900th iteration of loss is :5.83 (3.s.f)
19000th iteration of loss is :5.84 (3.s.f)
19100th iteration of loss is :5.84 (3.s.f)
19200th iteration of loss is :5.84 (3.s.f)
```

```
19300th iteration of loss is :5.84 (3.s.f)
19400th iteration of loss is :5.84 (3.s.f)
19500th iteration of loss is :5.84 (3.s.f)
19600th iteration of loss is :5.85 (3.s.f)
19700th iteration of loss is :5.85 (3.s.f)
19800th iteration of loss is :5.85 (3.s.f)
19900th iteration of loss is :5.85 (3.s.f)
```

In [15]: ✓

```python
# do the same for petals AND sepals
WAll = np.random.randn(mAll, k)
bAll = np.random.randn(k)
WAllOptimal, bAllOptimal, costsAll = train(X, y, lr, iters, WAll, bAll)
```

```
100th iteration of loss is :5.53 (3.s.f)
200th iteration of loss is :7.56 (3.s.f)
300th iteration of loss is :8.3 (3.s.f)
400th iteration of loss is :8.81 (3.s.f)
500th iteration of loss is :9.24 (3.s.f)
600th iteration of loss is :9.62 (3.s.f)
700th iteration of loss is :9.95 (3.s.f)
800th iteration of loss is :10.25 (3.s.f)
900th iteration of loss is :10.53 (3.s.f)
1000th iteration of loss is :10.78 (3.s.f)
1100th iteration of loss is :11.02 (3.s.f)
1200th iteration of loss is :11.24 (3.s.f)
1300th iteration of loss is :11.45 (3.s.f)
1400th iteration of loss is :11.64 (3.s.f)
1500th iteration of loss is :11.83 (3.s.f)
1600th iteration of loss is :12.0 (3.s.f)
1700th iteration of loss is :12.17 (3.s.f)
1800th iteration of loss is :12.32 (3.s.f)
1900th iteration of loss is :12.47 (3.s.f)
2000th iteration of loss is :12.62 (3.s.f)
2100th iteration of loss is :12.76 (3.s.f)
2200th iteration of loss is :12.89 (3.s.f)
2300th iteration of loss is :13.02 (3.s.f)
2400th iteration of loss is :13.14 (3.s.f)
2500th iteration of loss is :13.26 (3.s.f)
2600th iteration of loss is :13.37 (3.s.f)
2700th iteration of loss is :13.48 (3.s.f)
2800th iteration of loss is :13.59 (3.s.f)
2900th iteration of loss is :13.69 (3.s.f)
3000th iteration of loss is :13.79 (3.s.f)
3100th iteration of loss is :13.89 (3.s.f)
3200th iteration of loss is :13.98 (3.s.f)
3300th iteration of loss is :14.08 (3.s.f)
3400th iteration of loss is :14.17 (3.s.f)
3500th iteration of loss is :14.25 (3.s.f)
3600th iteration of loss is :14.34 (3.s.f)
3700th iteration of loss is :14.42 (3.s.f)
3800th iteration of loss is :14.5 (3.s.f)
3900th iteration of loss is :14.58 (3.s.f)
4000th iteration of loss is :14.66 (3.s.f)
4100th iteration of loss is :14.73 (3.s.f)
4200th iteration of loss is :14.8 (3.s.f)
4300th iteration of loss is :14.88 (3.s.f)
4400th iteration of loss is :14.95 (3.s.f)
4500th iteration of loss is :15.01 (3.s.f)
4600th iteration of loss is :15.08 (3.s.f)
4700th iteration of loss is :15.15 (3.s.f)
4800th iteration of loss is :15.21 (3.s.f)
4900th iteration of loss is :15.27 (3.s.f)
5000th iteration of loss is :15.34 (3.s.f)
```

```
5100th iteration of loss is :15.4 (3.s.f)
5200th iteration of loss is :15.46 (3.s.f)
5300th iteration of loss is :15.51 (3.s.f)
5400th iteration of loss is :15.57 (3.s.f)
5500th iteration of loss is :15.63 (3.s.f)
5600th iteration of loss is :15.68 (3.s.f)
5700th iteration of loss is :15.74 (3.s.f)
5800th iteration of loss is :15.79 (3.s.f)
5900th iteration of loss is :15.84 (3.s.f)
6000th iteration of loss is :15.9 (3.s.f)
6100th iteration of loss is :15.95 (3.s.f)
6200th iteration of loss is :16.0 (3.s.f)
6300th iteration of loss is :16.05 (3.s.f)
6400th iteration of loss is :16.09 (3.s.f)
6500th iteration of loss is :16.14 (3.s.f)
6600th iteration of loss is :16.19 (3.s.f)
6700th iteration of loss is :16.24 (3.s.f)
6800th iteration of loss is :16.28 (3.s.f)
6900th iteration of loss is :16.33 (3.s.f)
7000th iteration of loss is :16.37 (3.s.f)
7100th iteration of loss is :16.41 (3.s.f)
7200th iteration of loss is :16.46 (3.s.f)
7300th iteration of loss is :16.5 (3.s.f)
7400th iteration of loss is :16.54 (3.s.f)
7500th iteration of loss is :16.58 (3.s.f)
7600th iteration of loss is :16.62 (3.s.f)
7700th iteration of loss is :16.66 (3.s.f)
7800th iteration of loss is :16.7 (3.s.f)
7900th iteration of loss is :16.74 (3.s.f)
8000th iteration of loss is :16.78 (3.s.f)
8100th iteration of loss is :16.82 (3.s.f)
8200th iteration of loss is :16.86 (3.s.f)
8300th iteration of loss is :16.89 (3.s.f)
8400th iteration of loss is :16.93 (3.s.f)
8500th iteration of loss is :16.97 (3.s.f)
8600th iteration of loss is :17.0 (3.s.f)
8700th iteration of loss is :17.04 (3.s.f)
8800th iteration of loss is :17.07 (3.s.f)
8900th iteration of loss is :17.11 (3.s.f)
9000th iteration of loss is :17.14 (3.s.f)
9100th iteration of loss is :17.17 (3.s.f)
9200th iteration of loss is :17.21 (3.s.f)
9300th iteration of loss is :17.24 (3.s.f)
9400th iteration of loss is :17.27 (3.s.f)
9500th iteration of loss is :17.31 (3.s.f)
9600th iteration of loss is :17.34 (3.s.f)
9700th iteration of loss is :17.37 (3.s.f)
9800th iteration of loss is :17.4 (3.s.f)
9900th iteration of loss is :17.43 (3.s.f)
10000th iteration of loss is :17.46 (3.s.f)
10100th iteration of loss is :17.49 (3.s.f)
10200th iteration of loss is :17.52 (3.s.f)
10300th iteration of loss is :17.55 (3.s.f)
10400th iteration of loss is :17.58 (3.s.f)
10500th iteration of loss is :17.61 (3.s.f)
10600th iteration of loss is :17.64 (3.s.f)
10700th iteration of loss is :17.67 (3.s.f)
10800th iteration of loss is :17.7 (3.s.f)
10900th iteration of loss is :17.72 (3.s.f)
11000th iteration of loss is :17.75 (3.s.f)
11100th iteration of loss is :17.78 (3.s.f)
11200th iteration of loss is :17.81 (3.s.f)
11300th iteration of loss is :17.83 (3.s.f)
11400th iteration of loss is :17.86 (3.s.f)
```

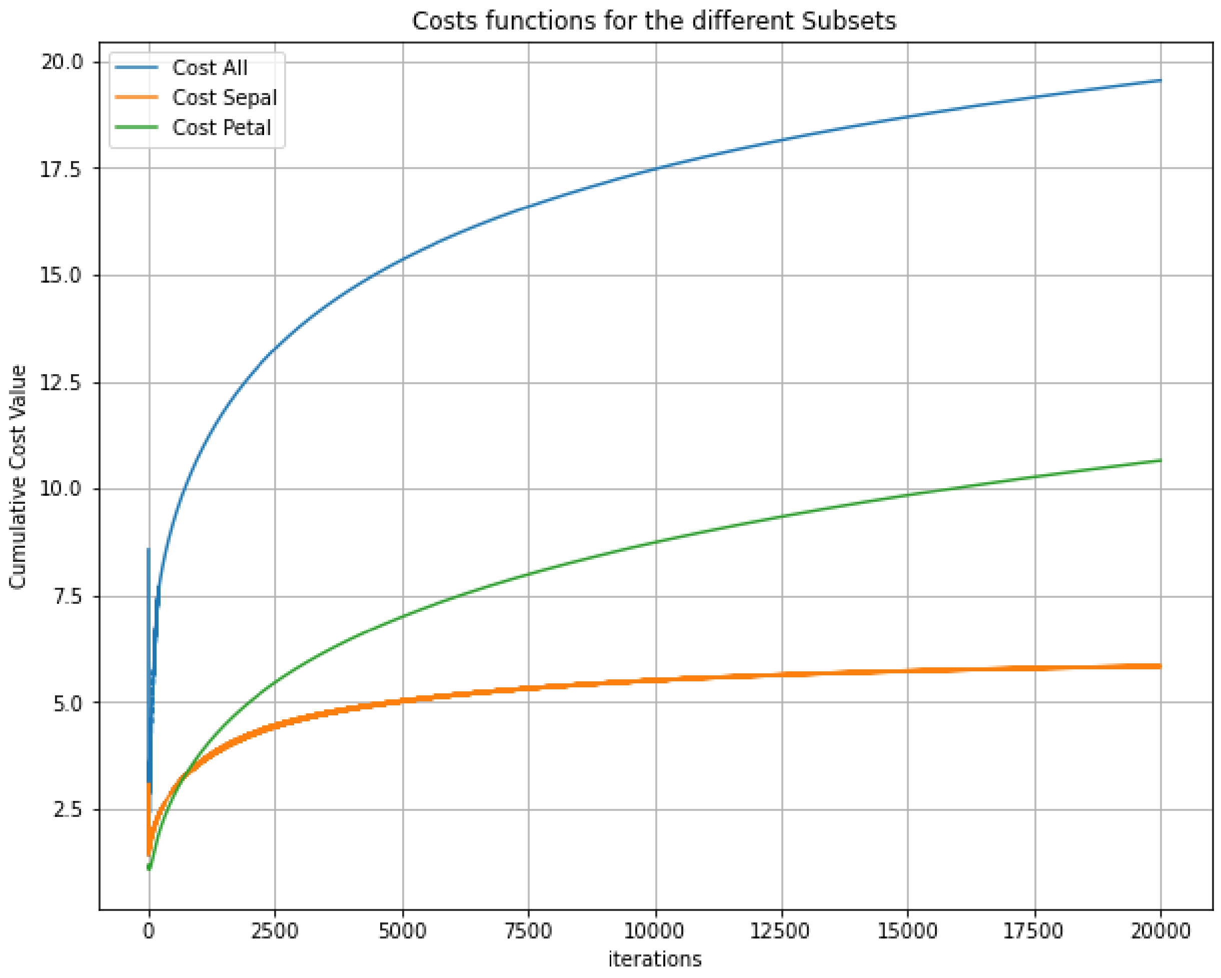
```
11500th iteration of loss is :17.89 (3.s.f)
11600th iteration of loss is :17.91 (3.s.f)
11700th iteration of loss is :17.94 (3.s.f)
11800th iteration of loss is :17.96 (3.s.f)
11900th iteration of loss is :17.99 (3.s.f)
12000th iteration of loss is :18.01 (3.s.f)
12100th iteration of loss is :18.04 (3.s.f)
12200th iteration of loss is :18.06 (3.s.f)
12300th iteration of loss is :18.09 (3.s.f)
12400th iteration of loss is :18.11 (3.s.f)
12500th iteration of loss is :18.14 (3.s.f)
12600th iteration of loss is :18.16 (3.s.f)
12700th iteration of loss is :18.18 (3.s.f)
12800th iteration of loss is :18.21 (3.s.f)
12900th iteration of loss is :18.23 (3.s.f)
13000th iteration of loss is :18.25 (3.s.f)
13100th iteration of loss is :18.28 (3.s.f)
13200th iteration of loss is :18.3 (3.s.f)
13300th iteration of loss is :18.32 (3.s.f)
13400th iteration of loss is :18.34 (3.s.f)
13500th iteration of loss is :18.37 (3.s.f)
13600th iteration of loss is :18.39 (3.s.f)
13700th iteration of loss is :18.41 (3.s.f)
13800th iteration of loss is :18.43 (3.s.f)
13900th iteration of loss is :18.45 (3.s.f)
14000th iteration of loss is :18.48 (3.s.f)
14100th iteration of loss is :18.5 (3.s.f)
14200th iteration of loss is :18.52 (3.s.f)
14300th iteration of loss is :18.54 (3.s.f)
14400th iteration of loss is :18.56 (3.s.f)
14500th iteration of loss is :18.58 (3.s.f)
14600th iteration of loss is :18.6 (3.s.f)
14700th iteration of loss is :18.62 (3.s.f)
14800th iteration of loss is :18.64 (3.s.f)
14900th iteration of loss is :18.66 (3.s.f)
15000th iteration of loss is :18.68 (3.s.f)
15100th iteration of loss is :18.7 (3.s.f)
15200th iteration of loss is :18.72 (3.s.f)
15300th iteration of loss is :18.74 (3.s.f)
15400th iteration of loss is :18.76 (3.s.f)
15500th iteration of loss is :18.78 (3.s.f)
15600th iteration of loss is :18.8 (3.s.f)
15700th iteration of loss is :18.82 (3.s.f)
15800th iteration of loss is :18.84 (3.s.f)
15900th iteration of loss is :18.85 (3.s.f)
16000th iteration of loss is :18.87 (3.s.f)
16100th iteration of loss is :18.89 (3.s.f)
16200th iteration of loss is :18.91 (3.s.f)
16300th iteration of loss is :18.93 (3.s.f)
16400th iteration of loss is :18.95 (3.s.f)
16500th iteration of loss is :18.96 (3.s.f)
16600th iteration of loss is :18.98 (3.s.f)
16700th iteration of loss is :19.0 (3.s.f)
16800th iteration of loss is :19.02 (3.s.f)
16900th iteration of loss is :19.03 (3.s.f)
17000th iteration of loss is :19.05 (3.s.f)
17100th iteration of loss is :19.07 (3.s.f)
17200th iteration of loss is :19.09 (3.s.f)
17300th iteration of loss is :19.1 (3.s.f)
17400th iteration of loss is :19.12 (3.s.f)
17500th iteration of loss is :19.14 (3.s.f)
17600th iteration of loss is :19.15 (3.s.f)
17700th iteration of loss is :19.17 (3.s.f)
17800th iteration of loss is :19.19 (3.s.f)
```

```
17900th iteration of loss is :19.2 (3.s.f)
18000th iteration of loss is :19.22 (3.s.f)
18100th iteration of loss is :19.24 (3.s.f)
18200th iteration of loss is :19.25 (3.s.f)
18300th iteration of loss is :19.27 (3.s.f)
18400th iteration of loss is :19.28 (3.s.f)
18500th iteration of loss is :19.3 (3.s.f)
18600th iteration of loss is :19.32 (3.s.f)
18700th iteration of loss is :19.33 (3.s.f)
18800th iteration of loss is :19.35 (3.s.f)
18900th iteration of loss is :19.36 (3.s.f)
19000th iteration of loss is :19.38 (3.s.f)
19100th iteration of loss is :19.39 (3.s.f)
19200th iteration of loss is :19.41 (3.s.f)
19300th iteration of loss is :19.42 (3.s.f)
19400th iteration of loss is :19.44 (3.s.f)
19500th iteration of loss is :19.46 (3.s.f)
19600th iteration of loss is :19.47 (3.s.f)
19700th iteration of loss is :19.49 (3.s.f)
19800th iteration of loss is :19.5 (3.s.f)
19900th iteration of loss is :19.51 (3.s.f)
```

As the above doesnt seem too easy to understand, i'm going to plot the costs on the same axis... ✓

✓ In [16]:

```
#plot the different costs
plt.figure(figsize=(10,8))
plt.plot(costsAll, label = "Cost All")
plt.plot(costsSepal, label = "Cost Sepal")
plt.plot(costsPetal, label = "Cost Petal")
plt.title("Costs functions for the different Subsets")
plt.xlabel("iterations")
plt.ylabel("Cumulative Cost Value")
plt.legend()
plt.grid()
plt.show()
```



Q10 b

to create a decision boundary, i will need to evaluate the whole region in the p_model fucntion, and then plot the class output for the whole domain as a plot like in Q1/Q2. ✓

In [17]:

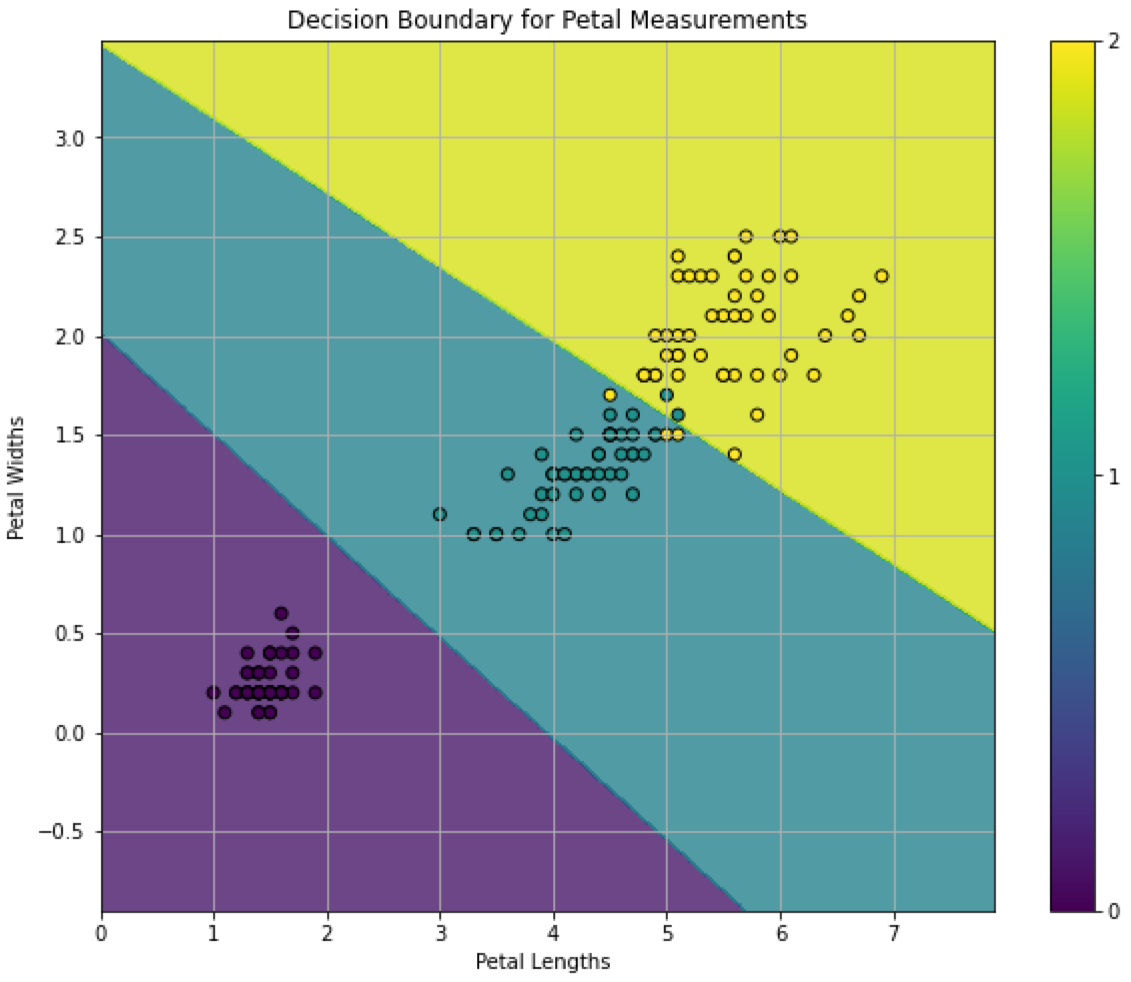
```
def PlotDecisionBoundary(X, y, W, b, title):
    # the title argument is mainyl for the presentation as seen below
    # create a grid of poinst
    x_min,x_max= X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min,y_max= X[:, 1].min() - 1, X[:, 1].max() + 1       ✓

    #set small increment for below arange
    h=0.01

    #deifne ranges
    xxrange = np.arange(x_min, x_max, h)
    xyrange =  np.arange(y_min,y_max,h)       ✓

    #define meshgid
    xx, yy  = np.meshgrid(xxrange,xyrange )

    # get the grid by matching all the xx and yy
    grid = np.c_[ xx.ravel(), yy.ravel() ]

    # send in the whole grid to pmodel
    Z = p_model(grid, W, b)
    Z = np.argmax(Z, axis=0)
    Z = Z.reshape(xx.shape)

    # Plot
    plt.figure(figsize = (10,8))
    #contours aka coloured domain
    plt.contourf(xx, yy, Z, alpha=0.8)
    #points
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
    #presentation - here we can use title arg for ease
    plt.ylabel(str(title) + " Widths")
    plt.xlabel(str(title) + " Lengths ")
    plt.colorbar(ticks = [2,1,0])
    plt.grid(True)
    plt.title("Decision Boundary for "+ str(title) + " Measurements")
    plt.show()
```

✓

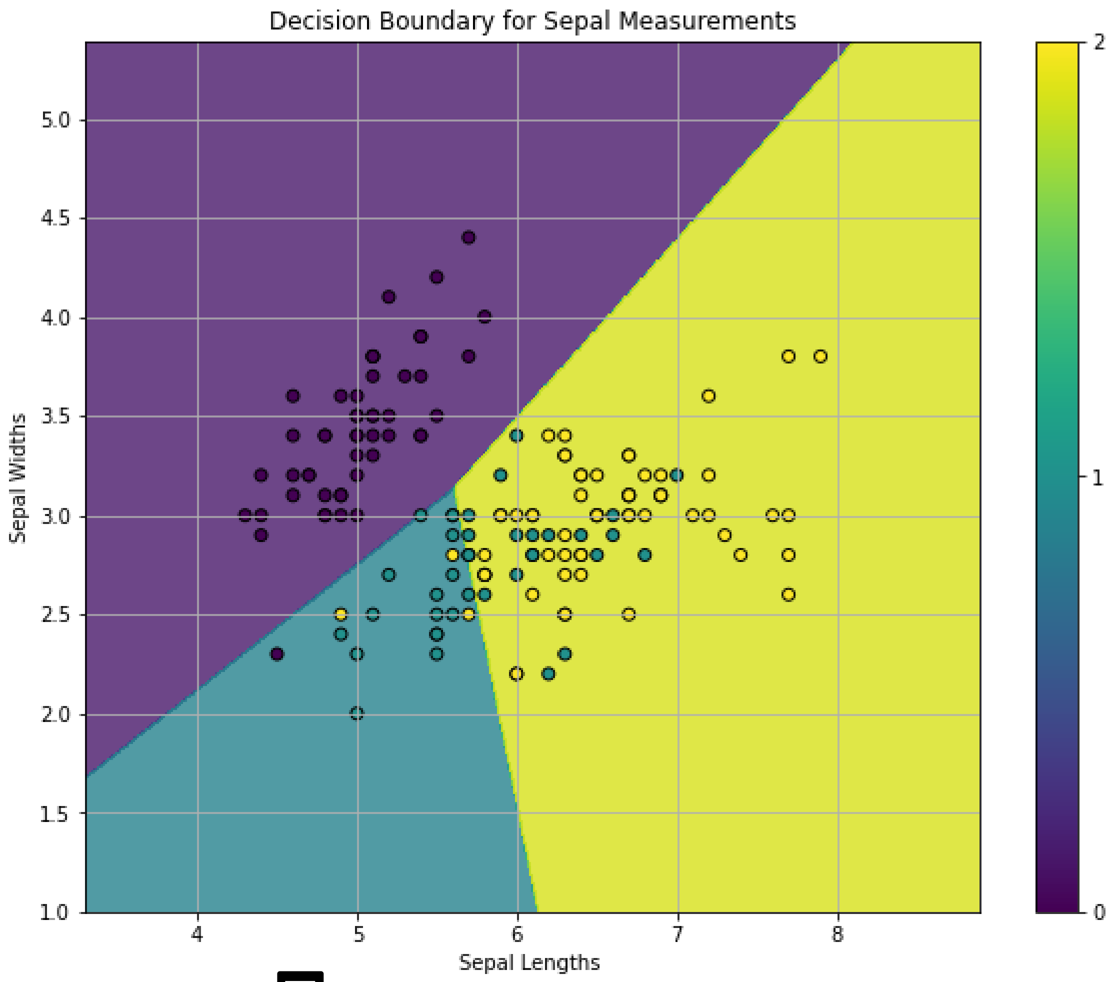
In [18]:

```
# Call the function
PlotDecisionBoundary(X_petal, y, WPetalOptimal, bPetalOptimal, "Petal")
PlotDecisionBoundary(X_sepal, y, WSepalOptimal, bSepalOptimal, "Sepal")
```

✓



✓



✓

The plots do suggest sensible classification rules, some issues may arrise when close to the boundry (missclassifications), but other than that the only other issues may be for erronious data points, or samples of species which do not follow the common trends of the length and width of the petals and sepals. For example, if a data point has negative value by accident, there would be a misclassification potentially, or an unusually large length for a versicolor (1) may present misclassification as a versicolor (0).

Finally, these plots are very similar to the lines drawn in Q2, slightly different placement of the lines but largely very similar locations. ✓

Q10c

To find accuracy, i'll find how many wrongly classified cases there are for each class of each case, then display the data.

Accuracy ill define as relative to the population,

$$Accuracy = \frac{\text{Total Number of Correct Data in Class}}{\text{Total Number of Data class}} * 100\%$$ ✓

In [19]:
```
def GetAccuracy(X, y, W, b):
    #send in the params to the pmodel again
    y_hats = p_model(X, W, b)
    # choose the highest probability again
    predLabels = np.argmax(y_hats, axis=0)

    # Finally compute accuracy as defined above
    accuracy = np.mean(predLabels == y) * 100

    return(accuracy)
```

In [20]:

```
# call the func for accuracy
PetalAcc = GetAccuracy(X_petal, y, WPetalOptimal, bPetalOptimal)
SepalAcc = GetAccuracy(X_sepal, y, WSepalOptimal, bSepalOptimal)
BothAcc = GetAccuracy(X, y, WAllOptimal, bAllOptimal)

#print
print("Accuracy of Petals = " + str(round(PetalAcc,1)) + "%")
print("Accuracy of Sepals = " + str(round(SepalAcc,1)) + "%")
print("Accuracy of Petals AND Sepals = " + str(round(BothAcc,1)) + "%")
```

```
Accuracy of Petals = 96.0%
Accuracy of Sepals = 75.3%
Accuracy of Petals AND Sepals = 98.0%
```

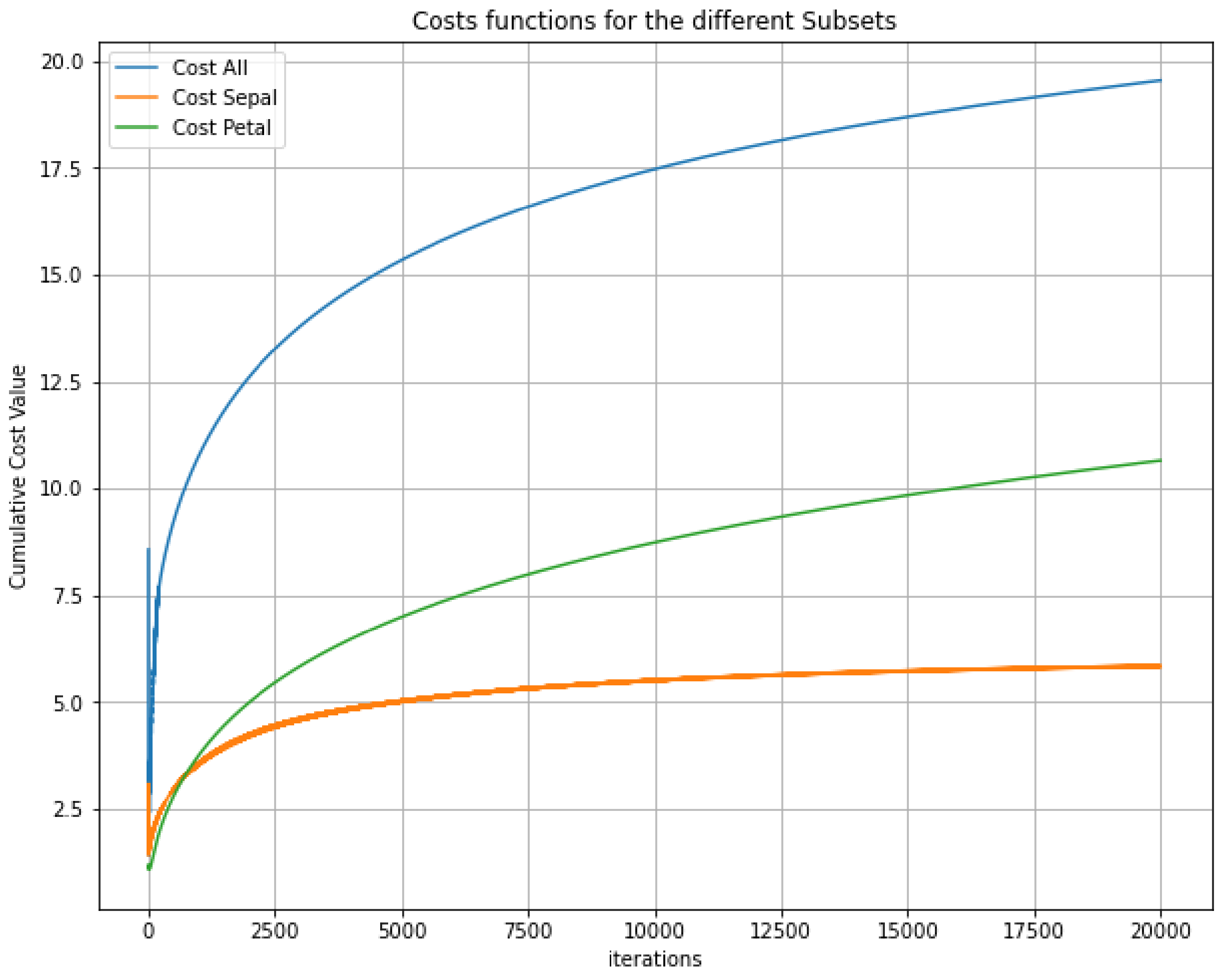
✓

This hence means overall, we had greater accuracy with the Petals, which was what we expected from the begining with the line being much easier to draw between the clusters of data on the petal graph.

Convieniently reffering to the graph I made earlier for the cost function ... which I'll put below again too ...

In [21]:

```
#plot the different costs
plt.figure(figsize=(10,8))
plt.plot(costsAll, label = "Cost All")
plt.plot(costsSepal, label = "Cost Sepal")
plt.plot(costsPetal, label = "Cost Petal")
plt.title("Costs functions for the different Subsets")
plt.xlabel("iterations")
plt.ylabel("Cumulative Cost Value")
plt.legend()
plt.grid()
plt.show()
```


Costs functions for the different Subsets
Cost All
Cost Sepal
Cost Petal
Cumulative Cost Value
iterations
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0
2500
5000
7500
10000
12500
15000
17500
20000

Comments on Results

We see that all the cost functions are positive definite, monotonoically increasing, functions and that the speed at which they plateu is inversly related to the accuracy.

For example, the bext accuracy classifier was the "Petals and Sepals" set at 98% ✓, the best of all the sets - however the convergence rate to optimal paramters and minimal cost was the slowest of the sets (being furthest from convergence after 20,000 iterations).

The same can be said for the Sepals Set. The Sepals had the fastest rate of cost function convergence and yet the classification accuracy was lowest of all teh sets.

However, in the question "what configuration works best?", context matters.

If you mean the best accuracy and hence best classifier, the Combined set of Petals and Sepals is the best although having the most room for improvement in terms of converegnce in cost function availiable... It could potentially get better than 98% if we gave it more iterations! On the contrary, if you believe that the fastest converging and hence best with respect to the most optimal selection of paramters, the Sepals set takes the least time to converge and hence over 20,000 iterations we see that this set is the set with the set with the lowest cost, and hence most optimal parameters and hence best.

For the sake of the question I choose the Combined set of "Petals and Sepals" as the best classifier due to having the highest accuracy of all other sets. ✓