Lab 3: Bayes Classifier and Boosting

Jupyter notebooks

In this lab, you can use Jupyter https://jupyter.org/ to get a nice layout of your code and plots in one document. However, you may also use Python as usual, without Jupyter.

If you have Python and pip, you can install Jupyter with `sudo pip install jupyter`. Otherwise you can follow the instruction on http://jupyter.readthedocs.org/en/latest/install.html.

And that is everything you need! Now use a terminal to go into the folder with the provided lab files. Then run `jupyter notebook` to start a session in that folder. Click `lab3.ipynb` in the browser window that appeared to start this very notebook. You should click on the cells in order and either press `ctrl+enter` or `run cell` in the toolbar above to evaluate all the expressions.

Be sure to put `%matplotlib inline` at the top of every code cell where you call plotting functions to get the resulting plots inside the document.

Import the libraries

In Jupyter, select the cell below and press `ctrl + enter` to import the needed libraries. Check out `labfuns.py` if you are interested in the details.

In [1]:
```
import numpy as np
from scipy import misc
from imp import reload
from labfuns import *
import random
```

Bayes classifier functions to implement

The lab descriptions state what each function should do.

In [32]:
```
# NOTE: you do not need to handle the W argument for this part!
# in: labels - N vector of class labels
# out: prior - C x 1 vector of class priors
def computePrior(labels, W=None):
    Npts = labels.shape[0]
    if W is None:
        W = np.ones((Npts,1))/Npts
    else:
        assert(W.shape[0] == Npts)
    classes = np.unique(labels)
    Nclasses = np.size(classes)

    prior = np.zeros((Nclasses,1))
```

```
    # TODO: compute the values of prior for each class!
    # Eq. 12
    # ==========================

    for jdx, c in enumerate(classes):
        idx = np.where(labels == c)[0]
        prior[jdx] = np.sum(W[idx])/np.sum(W)

    """
    NON-BOOSTED
    for jdx, c in enumerate(classes):
        idx = np.where(labels == c)[0]
        prior[jdx] = len(idx)/Npts
    """
    # ==========================

    return prior

# NOTE: you do not need to handle the W argument for this part!
# in:      X - N x d matrix of N data points
#     labels - N vector of class labels
# out:    mu - C x d matrix of class means (mu[i] - class i mean)
#      sigma - C x d x d matrix of class covariances (sigma[i] - class i sigma)
def mlParams(X, labels, W=None):
    assert(X.shape[0]==labels.shape[0])
    Npts,Ndims = np.shape(X)
    classes = np.unique(labels)
    Nclasses = np.size(classes)

    if W is None:
        W = np.ones((Npts,1))/float(Npts)

    mu = np.zeros((Nclasses,Ndims))
    sigma = np.zeros((Nclasses,Ndims,Ndims))

    # TODO: fill in the code to compute mu and sigma!
    # Eq. 8 and 10
    # ==========================

    for i in range(Nclasses):
        classidx = np.where(labels == classes[i])[0]
        Xi = X[classidx,:]
        Wi = W[classidx]
        mu[i] = sum(Xi*Wi)/sum(Wi)

        sigma[i] = np.diag(sum(Wi * np.square(Xi - mu[i]))/sum(Wi))


    """
    NON-BOOSTED
    for i in range(Nclasses):
        classidx = np.where(labels == classes[i])[0]
        Xi = X[classidx,:]
        mu[i] = sum(Xi)/Nclasses

        sigma[i] = np.diag(sum(np.square(Xi - mu[i]))/Nclasses)
    """
    # ==========================

    return mu, sigma

# in:      X - N x d matrix of M data points
#      prior - C x 1 matrix of class priors
#         mu - C x d matrix of class means (mu[i] - class i mean)
```

```
#       sigma - C x d x d matrix of class covariances (sigma[i] - class i sigma)
# out:      h - N vector of class predictions for test points
def classifyBayes(X, prior, mu, sigma):

    Npts = X.shape[0]
    Nclasses,Ndims = np.shape(mu)
    logProb = np.zeros((Nclasses, Npts))

    # TODO: fill in the code to compute the log posterior logProb!
    # Eq. 11
    # ========================
    for jdx in range(Nclasses):
        diff = X - mu[jdx] # Matrix  C x d with diffs between x - μ
        lnSigma = - np.log(np.linalg.det(sigma[jdx])) / 2 # N vector
        lnPrior = np.log(prior[jdx]) # N vector
        for i in range(Npts):
            logProb[jdx][i] = lnSigma - np.inner(diff[i]/np.diag(sigma[jdx]), diff
    # ========================

    # one possible way of finding max a-posteriori once
    # you have computed the log posterior
    h = np.argmax(logProb,axis=0)
    return h
```

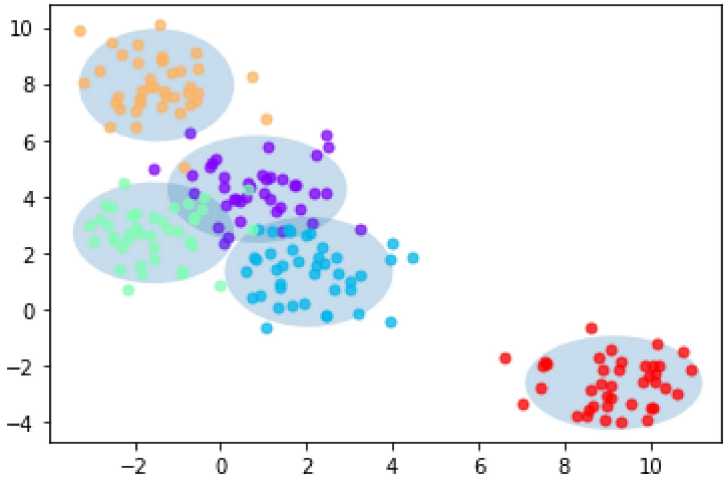
The implemented functions can now be summarized into the `BayesClassifier` class, which we will use later to test the classifier, no need to add anything else here:

In [3]:
```
# NOTE: no need to touch this
class BayesClassifier(object):
    def __init__(self):
        self.trained = False

    def trainClassifier(self, X, labels, W=None):
        rtn = BayesClassifier()
        rtn.prior = computePrior(labels, W)
        rtn.mu, rtn.sigma = mlParams(X, labels, W)
        rtn.trained = True
        return rtn

    def classify(self, X):
        return classifyBayes(X, self.prior, self.mu, self.sigma)
```

Test the Maximum Likelihood estimates

Call `genBlobs` and `plotGaussian` to verify your estimates.

In [31]:
```
%matplotlib inline

X, labels = genBlobs(centers=5)
mu, sigma = mlParams(X,labels)
plotGaussian(X,labels,mu,sigma)
```



Call the `testClassifier` and `plotBoundary` functions for this part.

In [12]:
```
testClassifier(BayesClassifier(), dataset='iris', split=0.7)
```

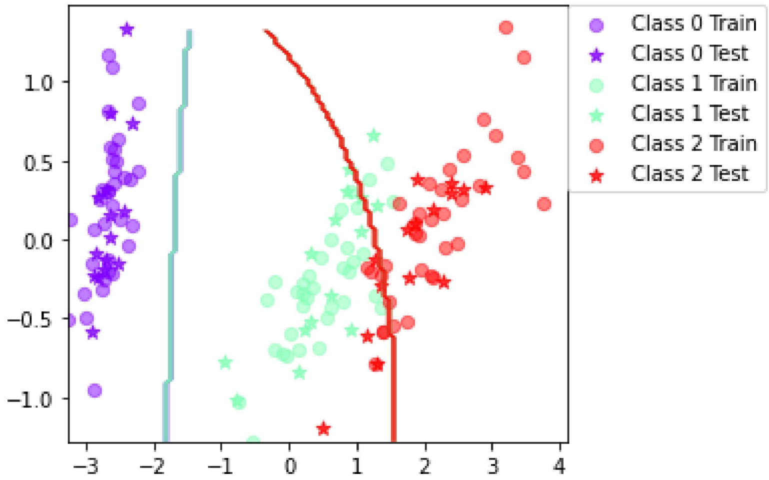
```
Trial: 0 Accuracy 84.4
Trial: 10 Accuracy 95.6
Trial: 20 Accuracy 93.3
Trial: 30 Accuracy 86.7
Trial: 40 Accuracy 88.9
Trial: 50 Accuracy 91.1
Trial: 60 Accuracy 86.7
Trial: 70 Accuracy 91.1
Trial: 80 Accuracy 86.7
Trial: 90 Accuracy 91.1
Final mean classification accuracy  89 with standard deviation 4.16
```

In [29]:
```
testClassifier(BayesClassifier(), dataset='vowel', split=0.7)
```

```
Trial: 0 Accuracy 61
Trial: 10 Accuracy 66.2
Trial: 20 Accuracy 74
Trial: 30 Accuracy 66.9
Trial: 40 Accuracy 59.7
Trial: 50 Accuracy 64.3
Trial: 60 Accuracy 66.9
Trial: 70 Accuracy 63.6
Trial: 80 Accuracy 62.3
Trial: 90 Accuracy 70.8
Final mean classification accuracy  64.7 with standard deviation 4.03
```

In [30]:
```
%matplotlib inline
plotBoundary(BayesClassifier(), dataset='iris',split=0.7)
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
```



Boosting functions to implement

The lab descriptions state what each function should do.

In [18]:
```
# in: base_classifier - a classifier of the type that we will boost, e.g. BayesClas
#                   X - N x d matrix of N data points
#              labels - N vector of class labels
#                   T - number of boosting iterations
# out:    classifiers - (maximum) length T Python list of trained classifiers
#              alphas - (maximum) length T Python list of vote weights
def trainBoost(base_classifier, X, labels, T=10):
    # these will come in handy later on
    Npts,Ndims = np.shape(X)

    classifiers = [] # append new classifiers to this list
    alphas = [] # append the vote weight of the classifiers to this list

    # The weights for the first iteration
    wCur = np.ones((Npts,1))/float(Npts)

    for i_iter in range(0, T):
```

```
        # a new classifier can be trained like this, given the current weights
        classifiers.append(base_classifier.trainClassifier(X, labels, wCur))

        # do classification for each point
        vote = classifiers[-1].classify(X)

        # TODO: Fill in the rest, construct the alphas etc.
        # ==========================

        # Compute the error
        error = 0
        for i in range(Npts):
            if vote[i] != labels[i]:
                error += wCur[i]

        # Compute the alpha
        if error == 0:
            alpha = 1
        else:
            alpha = np.log((1-error)/error)/2

        # Update the weights
        for i in range(Npts):
            if vote[i] == labels[i]:
                wCur[i] = wCur[i] * np.exp(-alpha)
            else:
                wCur[i] = wCur[i] * np.exp(alpha)

        # Normalize the weights
        wCur = wCur / sum(wCur)

        # ==========================

        alphas.append(alpha)

        # alphas.append(alpha) # you will need to append the new alpha
        # ==========================

    return classifiers, alphas

# in:       X - N x d matrix of N data points
# classifiers - (maximum) length T Python list of trained classifiers as above
#      alphas - (maximum) length T Python list of vote weights
#    Nclasses - the number of different classes
# out:  yPred - N vector of class predictions for test points
def classifyBoost(X, classifiers, alphas, Nclasses):
    Npts = X.shape[0]
    Ncomps = len(classifiers)

    # if we only have one classifier, we may just classify directly
    if Ncomps == 1:
        return classifiers[0].classify(X)
    else:
        votes = np.zeros((Npts,Nclasses))

        # TODO: implement classificiation when we have trained several classifiers
        # here we can do it by filling in the votes vector with weighted votes
        # ==========================

        # Compute the votes
        for i in range(Ncomps):
            vote = classifiers[i].classify(X)
            for j in range(Npts):
                votes[j][vote[j]] += alphas[i]
```

```
        # ==========================

        # one way to compute yPred after accumulating the votes
        return np.argmax(votes,axis=1)
```

The implemented functions can now be summarized another classifer, the `BoostClassifier` class. This class enables boosting different types of classifiers by initializing it with the `base_classifier` argument. No need to add anything here.

In [16]:
```
# NOTE: no need to touch this
class BoostClassifier(object):
    def __init__(self, base_classifier, T=10):
        self.base_classifier = base_classifier
        self.T = T
        self.trained = False

    def trainClassifier(self, X, labels):
        rtn = BoostClassifier(self.base_classifier, self.T)
        rtn.nbr_classes = np.size(np.unique(labels))
        rtn.classifiers, rtn.alphas = trainBoost(self.base_classifier, X, labels, s
        rtn.trained = True
        return rtn

    def classify(self, X):
        return classifyBoost(X, self.classifiers, self.alphas, self.nbr_classes)
```

Run some experiments

Call the `testClassifier` and `plotBoundary` functions for this part.

In [19]:
```
testClassifier(BoostClassifier(BayesClassifier(), T=10), dataset='iris',split=0.7)
```

```
Trial: 0 Accuracy 95.6
Trial: 10 Accuracy 100
Trial: 20 Accuracy 93.3
Trial: 30 Accuracy 91.1
Trial: 40 Accuracy 97.8
Trial: 50 Accuracy 93.3
Trial: 60 Accuracy 93.3
Trial: 70 Accuracy 97.8
Trial: 80 Accuracy 95.6
Trial: 90 Accuracy 93.3
Final mean classification accuracy  94.7 with standard deviation 2.82
```

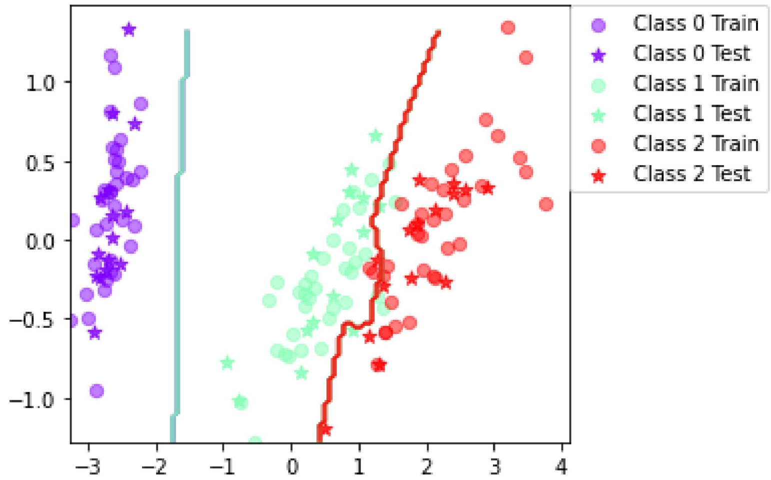
In [20]:
```
testClassifier(BoostClassifier(BayesClassifier(), T=10), dataset='vowel',split=0.7)
```

```
Trial: 0 Accuracy 76.6
Trial: 10 Accuracy 86.4
Trial: 20 Accuracy 83.1
Trial: 30 Accuracy 80.5
Trial: 40 Accuracy 72.7
Trial: 50 Accuracy 76
Trial: 60 Accuracy 81.8
Trial: 70 Accuracy 82.5
Trial: 80 Accuracy 79.9
Trial: 90 Accuracy 83.1
Final mean classification accuracy  80.2 with standard deviation 3.52
```

In [28]:
```
%matplotlib inline
```

```
plotBoundary(BoostClassifier(BayesClassifier()), dataset='iris',split=0.7)
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
```



Now repeat the steps with a decision tree classifier.

In [22]:
```
testClassifier(DecisionTreeClassifier(), dataset='iris', split=0.7)
```

```
Trial: 0 Accuracy 95.6
Trial: 10 Accuracy 100
Trial: 20 Accuracy 91.1
Trial: 30 Accuracy 91.1
Trial: 40 Accuracy 93.3
Trial: 50 Accuracy 91.1
Trial: 60 Accuracy 88.9
Trial: 70 Accuracy 88.9
Trial: 80 Accuracy 93.3
Trial: 90 Accuracy 88.9
Final mean classification accuracy  92.4 with standard deviation 3.71
```

In [23]:
```
testClassifier(BoostClassifier(DecisionTreeClassifier(), T=10), dataset='iris',spl
```

```
Trial: 0 Accuracy 95.6
Trial: 10 Accuracy 100
Trial: 20 Accuracy 95.6
Trial: 30 Accuracy 93.3
Trial: 40 Accuracy 93.3
Trial: 50 Accuracy 95.6
Trial: 60 Accuracy 88.9
Trial: 70 Accuracy 93.3
Trial: 80 Accuracy 93.3
Trial: 90 Accuracy 93.3
Final mean classification accuracy  94.6 with standard deviation 3.65
```

In [24]:
```
testClassifier(DecisionTreeClassifier(), dataset='vowel',split=0.7)
```

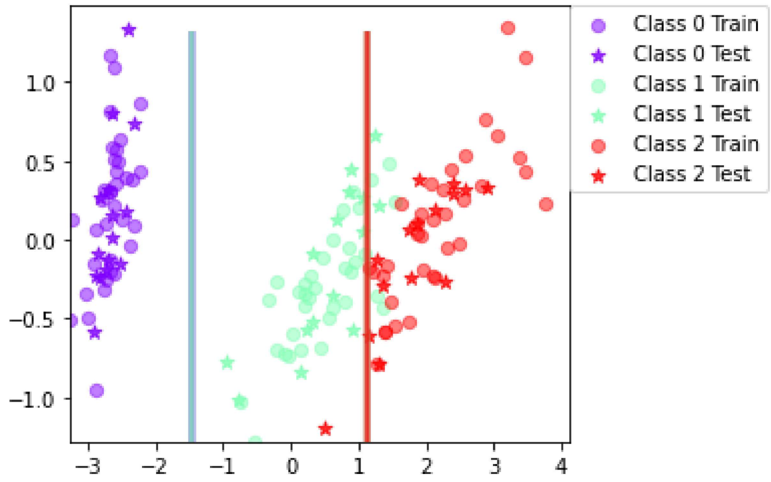
```
Trial: 0 Accuracy 63.6
Trial: 10 Accuracy 68.8
Trial: 20 Accuracy 63.6
Trial: 30 Accuracy 66.9
Trial: 40 Accuracy 59.7
Trial: 50 Accuracy 63
Trial: 60 Accuracy 59.7
Trial: 70 Accuracy 68.8
Trial: 80 Accuracy 59.7
Trial: 90 Accuracy 68.2
Final mean classification accuracy  64.1 with standard deviation 4
```

In [25]:
```
testClassifier(BoostClassifier(DecisionTreeClassifier(), T=10), dataset='vowel',sp
```

```
Trial: 0 Accuracy 85.7
Trial: 10 Accuracy 89.6
Trial: 20 Accuracy 86.4
Trial: 30 Accuracy 92.2
Trial: 40 Accuracy 80.5
Trial: 50 Accuracy 81.8
Trial: 60 Accuracy 85.7
Trial: 70 Accuracy 86.4
Trial: 80 Accuracy 86.4
Trial: 90 Accuracy 89.6
Final mean classification accuracy  86.5 with standard deviation 2.75
```

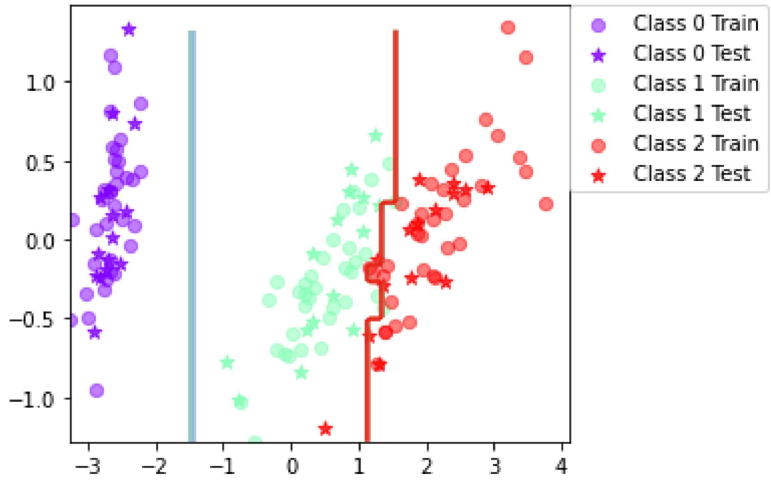
In [26]:
```
%matplotlib inline
plotBoundary(DecisionTreeClassifier(), dataset='iris',split=0.7)
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with *x* & *
y*.  Please use the *color* keyword-argument or provide a 2D array with a single r
ow if you intend to specify the same RGB or RGBA value for all points.
```



In [27]:
```
%matplotlib inline
plotBoundary(BoostClassifier(DecisionTreeClassifier(), T=10), dataset='iris',split=
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.
```



Bonus: Visualize faces classified using boosted decision trees

Note that this part of the assignment is completely voluntary! First, let's check how a boosted decision tree classifier performs on the olivetti data. Note that we need to reduce the dimension a bit using PCA, as the original dimension of the image vectors is `64 x 64 = 4096` elements.

In [ ]:
```
testClassifier(BayesClassifier(), dataset='olivetti',split=0.7, dim=20)
```

In [ ]:
```
testClassifier(BoostClassifier(DecisionTreeClassifier(), T=10), dataset='olivetti'
```

You should get an accuracy around 70%. If you wish, you can compare this with using pure decision trees or a boosted bayes classifier. Not too bad, now let's try and classify a face as belonging to one of 40 persons!

In [ ]:
```
%matplotlib inline
```

```
X,y,pcadim = fetchDataset('olivetti') # fetch the olivetti data
xTr,yTr,xTe,yTe,trIdx,teIdx = trteSplitEven(X,y,0.7) # split into training and test
pca = decomposition.PCA(n_components=20) # use PCA to reduce the dimension to 20
pca.fit(xTr) # use training data to fit the transform
xTrpca = pca.transform(xTr) # apply on training data
xTepca = pca.transform(xTe) # apply on test data
# use our pre-defined decision tree classifier together with the implemented
# boosting to classify data points in the training data
classifier = BoostClassifier(DecisionTreeClassifier(), T=10).trainClassifier(xTrpca
yPr = classifier.classify(xTepca)
# choose a test point to visualize
testind = random.randint(0, xTe.shape[0]-1)
# visualize the test point together with the training points used to train
# the class that the test point was classified to belong to
visualizeOlivettiVectors(xTr[yTr == yPr[testind],:], xTe[testind,:])
```