DD2421: Machine Learning

Lab 3 : Bayes Classifiers and Boosting

Amna Irshad (amnai@kth.se)
Aravind Nair (aanair@kth.se)

March 24, 2023

KTH Royal Institute of Technology
Stockholm, Sweden

[2]:
```
pip install sklearn
```

```
Requirement already satisfied: sklearn in
/Users/aanair/miniconda3/envs/mlenv/lib/python3.7/site-packages (0.0.post1)
Note: you may need to restart the kernel to use updated packages.
```

[26]:
```
import numpy as np
from scipy import misc
from imp import reload
from labfuns import *
import random
```

0.1 Bayes classifier functions to implement

The lab descriptions state what each function should do.

[27]:
```
# 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!
    # ==========================
    for idx, classname in enumerate(labels):
        prior[classname] += W[idx]

    prior /= np.linalg.norm(prior)
    # ==========================

    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!
    # ==========================
    for jdx, classname in enumerate(classes):

        idx = labels == classname # Returns a true or false with the length of y
        # Or more compactly extract the indices for which y==class is true,
        # analogous to MATLAB's find
        idx = np.where(labels==classname)[0]
        xlc = X[idx,:] # Get the x for the class labels. Vectors are rows.

        weightsum = sum(W[idx,:])

        # Mean
        mu[jdx] += np.sum(X[idx,:]*W[idx,:], axis=0) / weightsum

        # Sigma - contains the covariance
        for dim in range(Ndims):
            totsum = 0
            for ind in idx:
                totsum += W[ind]*(X[ind][dim] - mu[jdx][dim]) ** 2
            sigma[jdx][dim][dim] = totsum / weightsum
    # ==========================

    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!
    # ==========================
```

```
    for x_ind in range(Npts):
        for classname in range(Nclasses):
            term_one = (1/2)*np.log(np.linalg.det(sigma[classname]))

            term_two_sub_one = X[x_ind] - mu[classname]
            term_two_sub_two = np.diag(1/np.diag(sigma[classname]))
            term_two_sub_three = np.transpose(X[x_ind] - mu[classname])
            term_two = np.linalg.multi_dot([term_two_sub_one, term_two_sub_two,␣
↪term_two_sub_three])

            term_three = np.log(prior[classname])

            logProb[classname][x_ind] = - term_one - term_two + term_three
    # ==========================

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

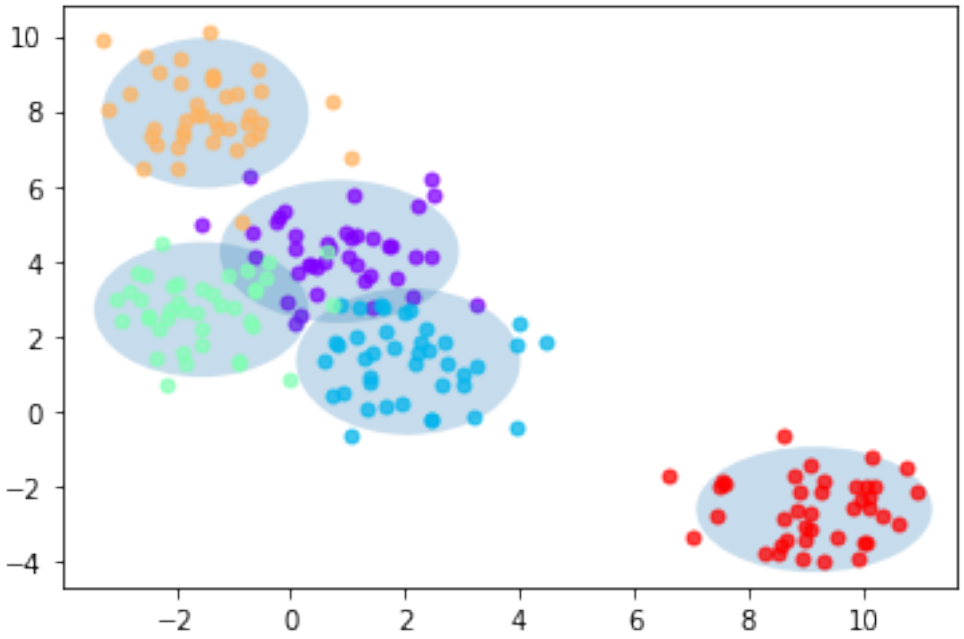
The implemented functions can now be summarized into the `BayesClassifier` class, which we will use later to test the classifier, no need to add anything else here:

[28]:
```
# 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)
```

0.2 Test the Maximum Likelihood estimates

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

[29]:
```
%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.

[30]:
```
testClassifier(BayesClassifier(), dataset='iris', split=0.7)
```

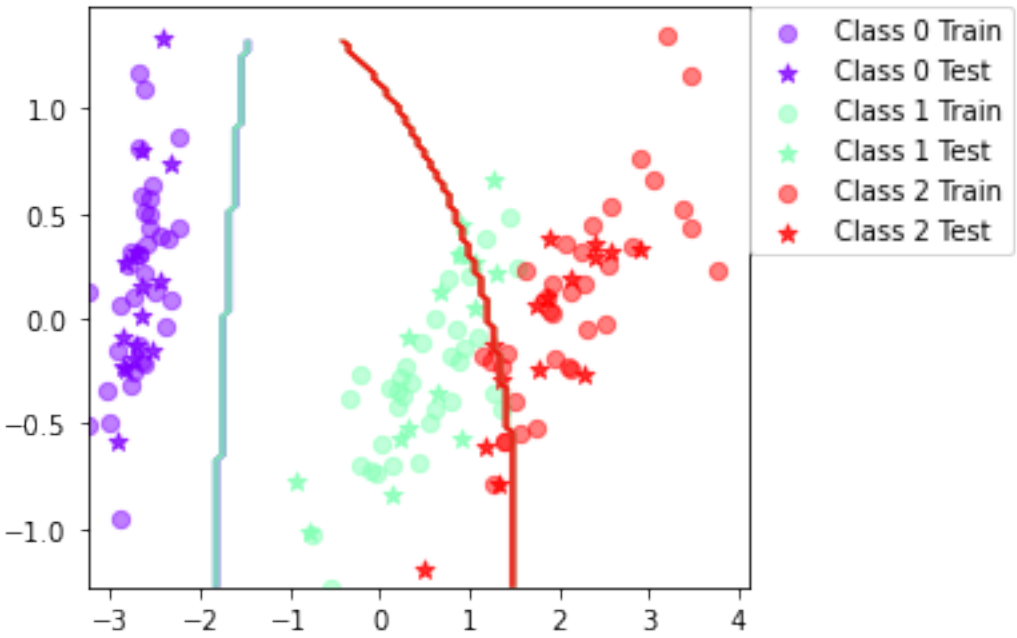
```
Trial: 0 Accuracy 84.4
Trial: 10 Accuracy 97.8
Trial: 20 Accuracy 91.1
Trial: 30 Accuracy 86.7
Trial: 40 Accuracy 88.9
Trial: 50 Accuracy 91.1
Trial: 60 Accuracy 86.7
Trial: 70 Accuracy 91.1
Trial: 80 Accuracy 86.7
Trial: 90 Accuracy 91.1
Final mean classification accuracy  89.2 with standard deviation 4.19
```

[31]:
```
testClassifier(BayesClassifier(), dataset='vowel', split=0.7)
```

```
Trial: 0 Accuracy 52.6
Trial: 10 Accuracy 61.7
Trial: 20 Accuracy 68.2
Trial: 30 Accuracy 62.3
Trial: 40 Accuracy 56.5
Trial: 50 Accuracy 63
Trial: 60 Accuracy 64.3
Trial: 70 Accuracy 62.3
Trial: 80 Accuracy 60.4
```

```
Trial: 90 Accuracy 65.6
Final mean classification accuracy  61.3 with standard deviation 3.48
```

[9]:
```
%matplotlib inline
plotBoundary(BayesClassifier(), dataset='iris',split=0.7)
```



0.3 Boosting functions to implement

The lab descriptions state what each function should do.

[10]:
```
# in: base_classifier - a classifier of the type that we will boost, e.g. BayesClassifier
#                   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.
        # ==========================
        weightsum = np.sum(wCur)

        # Adaboost step (2)
        error = weightsum
        correctvote = np.where(vote == labels)[0]
        for i in correctvote:
            error -= wCur[i]

        # Adaboost step (3)
        curAlpha = (np.log(1 - error) - np.log(error))/2 # Compute new alpha
        alphas.append(curAlpha) # you will need to append the new alpha

        # Adaboost step (4)
        falsevote = np.where(vote != labels)[0]
        wOld = wCur

        for i in correctvote:
            wCur[i] = wOld[i] * np.exp(-curAlpha)
        for i in falsevote:
            wCur[i] = wOld[i] * np.exp(curAlpha)
        wCur /= np.sum(wCur)
        # 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
        # ==========================
        for index, classifier in enumerate(classifiers):
            classified = classifier.classify(X)
            for i in range(Npts):
                votes[i][classified[i]] += alphas[index]
        # ==========================

        # 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.

[11]:
```
# 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, self.T)
        rtn.trained = True
        return rtn

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

[ ]:

0.4 Run some experiments

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

[12]:
```
testClassifier(BoostClassifier(BayesClassifier(), T=10), dataset='iris',split=0.
↪7)
```

```
Trial: 0 Accuracy 100
Trial: 10 Accuracy 97.8
Trial: 20 Accuracy 93.3
Trial: 30 Accuracy 93.3
Trial: 40 Accuracy 97.8
Trial: 50 Accuracy 86.7
Trial: 60 Accuracy 93.3
Trial: 70 Accuracy 95.6
Trial: 80 Accuracy 93.3
Trial: 90 Accuracy 95.6
Final mean classification accuracy  94.2 with standard deviation 6.83
```

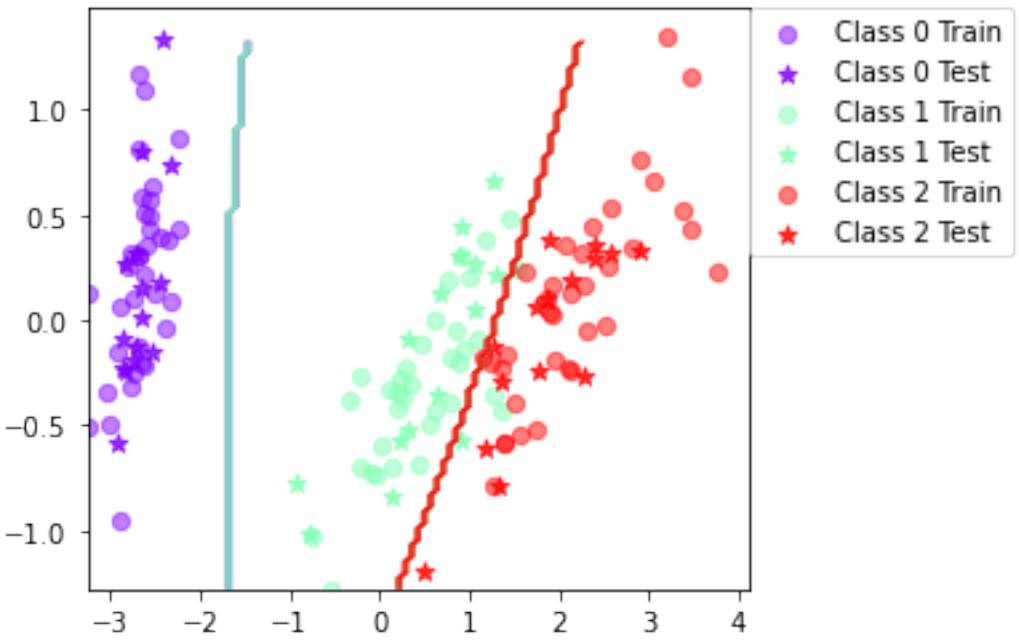
[ ]:
```

```

[13]:
```
testClassifier(BoostClassifier(BayesClassifier(), T=10), dataset='vowel',split=0.
↪7)
```

```
Trial: 0 Accuracy 68.8
Trial: 10 Accuracy 77.9
Trial: 20 Accuracy 77.3
Trial: 30 Accuracy 70.1
Trial: 40 Accuracy 68.8
Trial: 50 Accuracy 68.2
Trial: 60 Accuracy 77.3
Trial: 70 Accuracy 70.8
Trial: 80 Accuracy 71.4
Trial: 90 Accuracy 82.5
Final mean classification accuracy  74.1 with standard deviation 3.73
```

[ ]:
```

```

[15]:
```
%matplotlib inline
plotBoundary(BoostClassifier(BayesClassifier()), dataset='iris',split=0.7)
```



[ ]:

Now repeat the steps with a decision tree classifier.

[16]:
```
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
```

[ ]:

[17]:
```
testClassifier(BoostClassifier(DecisionTreeClassifier(), T=10),␣
↪dataset='iris',split=0.7)
```

```
Trial: 0 Accuracy 95.6
Trial: 10 Accuracy 100
Trial: 20 Accuracy 95.6
Trial: 30 Accuracy 91.1
```

```
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 3.67
```

[ ]:

[18]:
```
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
```

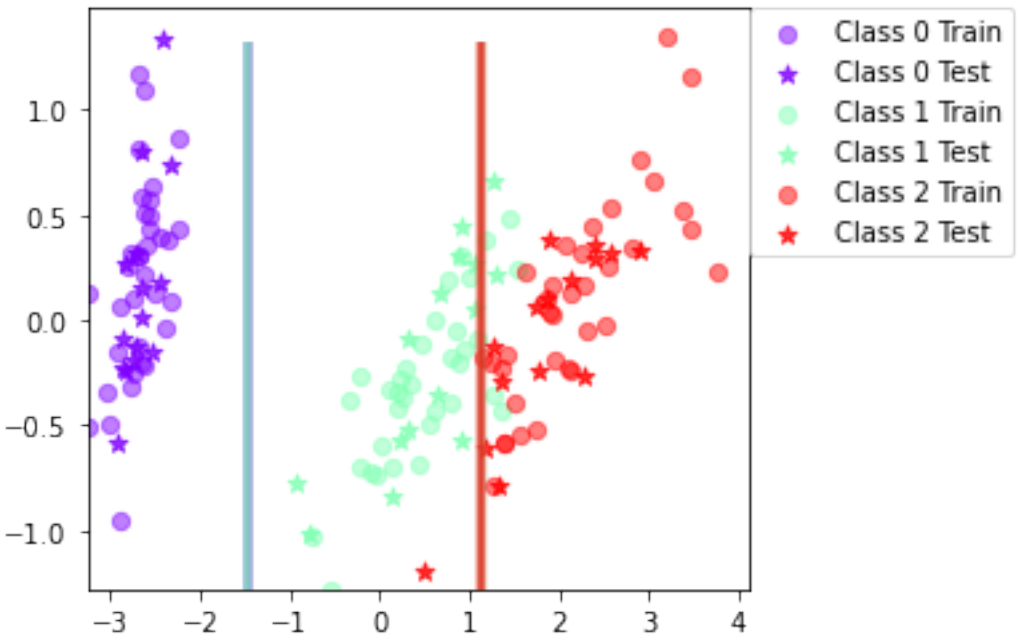
[ ]:

[19]:
```
testClassifier(BoostClassifier(DecisionTreeClassifier(), T=10),␣
↪dataset='vowel',split=0.7)
```

```
Trial: 0 Accuracy 86.4
Trial: 10 Accuracy 87.7
Trial: 20 Accuracy 87
Trial: 30 Accuracy 92.9
Trial: 40 Accuracy 83.8
Trial: 50 Accuracy 81.2
Trial: 60 Accuracy 90.3
Trial: 70 Accuracy 89.6
Trial: 80 Accuracy 86.4
Trial: 90 Accuracy 86.4
Final mean classification accuracy  86.5 with standard deviation 2.96
```

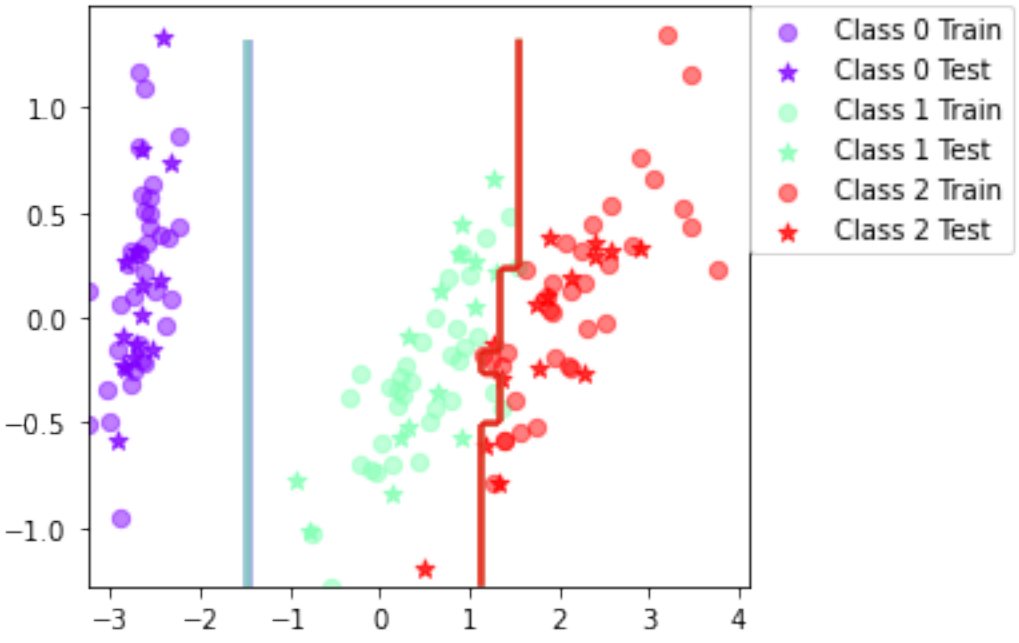
[ ]:

[20]:
```
%matplotlib inline
plotBoundary(DecisionTreeClassifier(), dataset='iris',split=0.7)
```



[ ]:

[21]:
```
%matplotlib inline
plotBoundary(BoostClassifier(DecisionTreeClassifier(), T=10),␣
 ↪dataset='iris',split=0.7)
```



[ ]:

0.5 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.

[22]:
```
testClassifier(BayesClassifier(), dataset='olivetti',split=0.7, dim=20)
```

```
Trial: 0 Accuracy 82.5
Trial: 10 Accuracy 88.3
Trial: 20 Accuracy 83.3
Trial: 30 Accuracy 85.8
Trial: 40 Accuracy 86.7
Trial: 50 Accuracy 80.8
Trial: 60 Accuracy 88.3
Trial: 70 Accuracy 81.7
Trial: 80 Accuracy 79.2
Trial: 90 Accuracy 84.2
Final mean classification accuracy  84.2 with standard deviation 3.23
```

[ ]:

[23]:
```
testClassifier(BoostClassifier(DecisionTreeClassifier(), T=10),␣
↪dataset='olivetti',split=0.7, dim=20)
```

```
Trial: 0 Accuracy 80
Trial: 10 Accuracy 77.5
Trial: 20 Accuracy 70
Trial: 30 Accuracy 74.2
Trial: 40 Accuracy 68.3
Trial: 50 Accuracy 68.3
Trial: 60 Accuracy 70.8
Trial: 70 Accuracy 77.5
Trial: 80 Accuracy 74.2
Trial: 90 Accuracy 74.2
Final mean classification accuracy  71.4 with standard deviation 6.17
```

[ ]:

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!

[25]:
```
%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␣
 ↪testing
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, yTr)
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,:])
```

[ ]:

[ ]:

1 Theoretical Questions

1.1 Naive Bayes

- When can a feature independence assumption be reasonable and when not?

  Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable. However, in real-world datasets, features tend to be dependent, but we still assume them as independent, example being words in a sentence. Surprisingly NB models perform well despite the conditional independence assumption. In the case of the iris dataset, we can suppose that sepal length and sepal width are positively correlated, as well as petal length and petal width. Assuming independence allows us to greatly simplify the computations needed to produce and train our naive bayes classifier.

- How does the decision boundary look for the Iris dataset? How could one improve the classification results for this scenario by changing classifier or, alternatively, manipulating the data?

  The decision boundary between class 0 and class 1 looks like a straight line and seems to classify the datasets often correctly. However, the decision boundary between class 1 and class 2 could be more optimal, as the misclassified points are shown in the figure 2. This is due to the inherent overlap between the class1 and class2 datapoints in the dataset. In such cases, non-linear models like SVM or ensemble methods like random forests can find a better boundary separation. Another way will be to apply log or similar non-linear transformations on data to separate the overlapped data points.

1.2 Boosting

- Is there any improvement in classification accuracy? Why/why not?

  Yes, all datasets saw improvement in classification accuracy after boosting, especially the vowel dataset. The improvement is because the weights of points harder for the model to classify are increased. Also, multiple Naive Bayes models can be ensembled to form a complex model capable of making better decisions on challenging datasets.

- Plot the decision boundary of the boosted classifier on iris and compare it with that of the basic. What differences do you notice? Is the boundary of the boosted version more complex?

  The decision boundary that was not optimal before now looks much better in separating class 1 and 2 data points. The boundaries seem less complex, almost linear.

- Can we make up for not using a more advanced model in the basic classifier (e.g. independent features) by using boosting?

  Yes, boosting improves classification accuracy, as demonstrated by the lab experiments. However, this is achieved by giving more weightage to misclassified data points. It does not change the existing problems such as lack of feature independence. Also, boosting can tend to overfitting if overused.

1.3 Which Classifier

- If you had to pick a classifier, naive Bayes or a decision tree or the boosted versions of these, which one would you pick?

Picking a classifier depends on the dataset. Listed are the choice prefernces based on the given criterias:

Outliers: Naive Bayes without boosting. Decision trees would tend to overfit the data and also a boosted Bayes classifier would give too much weight to the outliers.

Irrelevant inputs: Decision Trees as they will ignore the irrelevant part of the feature space concentrating only on attributes with high information gain.

Predictive power: Naive Bayes with boosting.

Mixed types of data: Both models work well with mixed kind of data. So this is a bit difficult to answer. However, if the data is continuous in nature, I would slightly prefer Naive Bayes model.

Scalability: Decision Trees. Bayes works well with small datasets. However it suffers from the 'Curse of dimensionality' and to deal with this it assumes all features are independent of one another.