Machine Learning

Exercise 4: Generative non-parametric classification: Naive Bayes and Density trees

November 24, 2014

Group: Sergej Kraft
Elias Roeger
Ekaterina Tikhoncheva

Contents

1 Data set

In this exercise we used the MNIST dataset, containing images of the handwritten digits. The size of the original images was to big for our purpose, so we used

provided compressed version of the dataset, from which we picked up images with handwritten threes and eights.
For further dimension reduction of the feature space to the size of 2 we used as corresponding function, we wrote for the exercise 2:

dr.py

```
#                              Dimension Reduction Function
# size(x) = n x d.
# size(dr(x)) = n x 2
def dr(x, y, d=[3,8]):
    # Calculate the average digit d[0]
    x_1 = x[y==d[0]]
    n1 = len(x_1)
    average1 = np.sum(x_1[:,:], axis = 0)/float(n1)


    # Calculate the average digit d[1]
    x_2 = x[y==d[1]]
    n2 = len(x_2)
    average2 = np.sum(x_2[:,:], axis = 0)/float(n2)

    # Differences between average1 and average7
    diff = np.abs(average1-average2)

    # Sort in descending order
    diff_sortInd = np.argsort(diff);
    diff_sortInd = diff_sortInd[::-1]

    # leave only indices of the two first elements
    # It means, we choose two dimensions, where average digits have highest
    # difference
    diff_sortInd = diff_sortInd[0:2:1]

    xn = x[:,diff_sortInd]

    return xn
#end def dr
```

2 Naive Bayes

2.1 Classification

As we know from the theory the assumption of the naive Bayes Methode is, that all features are independent. That means:

$$p(y = k|x) = \frac{\prod_{j=1}^{d} p(x_j|y = k)p(y = k)}{\prod_{j=1}^{d} p(x_j)}$$

Our aim is therefore to learn for each class k one dimension histograms $p(x_j|y = k)$ and prior $p(y = k)$.
The prior $p(y = k)$ is simply N_k/N, where N_k is the number of elements in the training set from the class k and N is the size of the whole training set. To learn the histograms we need an appropriate binning. We selected a common bin number for each dimension in the following way: for each dimension we used the Freeman-Dice rule to compute the bin width. From the bin width we got the necessary number of bins pro dimension and class. We selected the smallest necessary number of bin over all dimensions and recalculated the corresponding bin width.

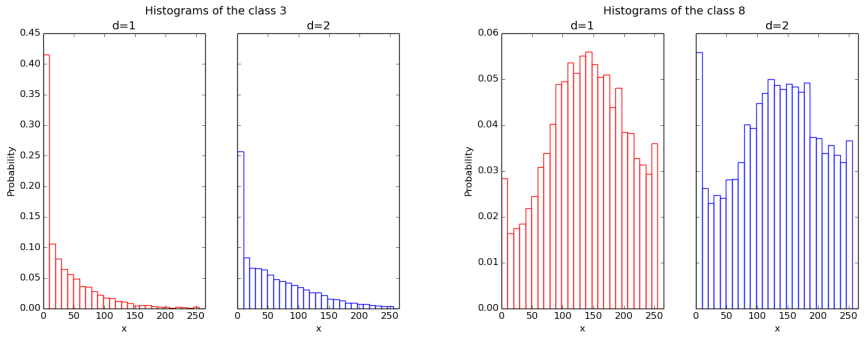
```
#                          Choose proper bin size
def chooseBinSize(trainingx):

    n = trainingx.shape[0]
    d = trainingx.shape[1]

    # Choose bin width
    dx = np.zeros(d, dtype = np.float128)
    m = np.zeros(d, dtype = np.int32)
    for j in range(0,d):
        # for each dimension apply Freeman-Diace Rule
        ind_sort =  np.argsort(trainingx[:,j]); # j-th feature dimension
        IQR = trainingx[ind_sort[3*n/4],j] - trainingx[ind_sort[n/4],j]
        dx[j] = 2*IQR/np.power(n, 1/3.)
        if dx[j]<0.01:
           dx[j] =  3.5/np.power(n, 1/3.)
        m_j = (np.max(trainingx[:,j])-np.min(trainingx[:,j]))/dx[j]
        m[j] = np.floor(m_j) + 1
    # end for j

    L = np.min(m);  # total number of bins as minimum over all bin sizes
                    # in all dimensions

    # recalculate bin width according to the new bin size L
    for j in range(0,d):
        dx[j] = (np.max(trainingx[:,j])-np.min(trainingx[:,j]))/float(L-1)
    # end for j

    return L, dx
# end chooseBinSize
```

The learning function uses the given number of bins and bin width to calculate 1-dimensional histograms pro class and dimension:

```
##                           Naive Bayes Training
# determine priors  and likelihoods (for each feature and class individual
```

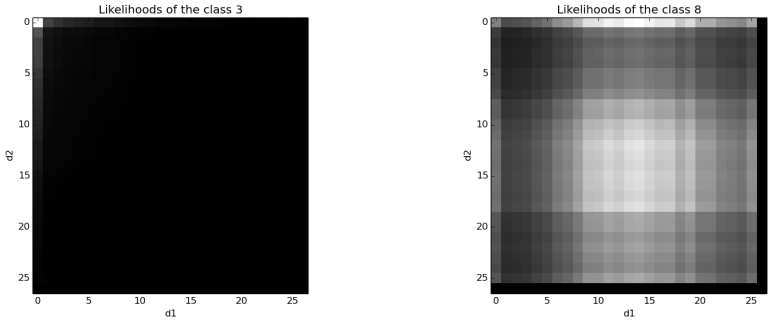
```
# histogram <=> 4 histogramms   )
def naiveBayes_train_single_class(trainingx, trainingy, c, dx, L):
    # we consider one class c
    #
    # trainingx is our training set
    # trainingy are class labels for each element from trainingx
    # dx bin width
    # L total number of bins pro dimension

    n = trainingx.shape[0]       # size of the training set
    d = trainingx.shape[1]       # size of the feature space

    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]        # Class of digit c
    nc = xc.shape[0]

    ## Priors
    prior = nc/float(n)

    ## Likelihood p(x|y=c)

    likelihood = np.zeros((d, L), dtype = np.float32)

    for j in range(0,d):
        for i in range(0,nc):
            l = np.floor(xc[i,j]/dx[j])+1 # bin
            if l>=L+1:
                print xc[i,j]
            likelihood[j, l-1] = likelihood[j, l-1] + 1
        # end for i=1..nc
        likelihood[j,:] = likelihood[j,:]/float(nc)
    #end for j=1..d

    return prior, likelihood
#end def naiveBayes_train
```

If we visualize calculated histograms we get following result:
One the next two images we can see the visualization of the likelihoods for each class:



(a) Class of handwritten threes (b) Class of handwritten eights

Figure 1: 1 dimensional histograms pro class and dimension



(a) Class of handwritten threes (b) Class of handwritten eights

Figure 2: 2D likelihoods

After learning phase of the classifier we apply it on the test set images. The correct classification rate of the classifier is 0.8266 (error rate = 0.1734).
The classification function:

```
##                              Naive Bayes Classifier
#
def naiveBayesClassifier(testx, p3, p8, p_k3, p_k8, dx):
    n = testx.shape[0]

    prediction = np.zeros(n, dtype = np.int8)

    for i in range(0,n):
        x = testx[i,:]

        # p(y = 3| x)

        l_y3_d1 = np.floor(x[0]/dx[0])+1 # bin number
        p_x_y3_d1 = p_k3[0,l_y3_d1-1]

        l_y3_d2 = np.floor(x[1]/dx[1])+1 # bin number
        p_x_y3_d2 = p_k3[1,l_y3_d2-1]

        p_y3_x = p_x_y3_d1*p_x_y3_d2*p3

        # p(y = 8| x)

        l_y8_d1 = np.floor(x[0]/dx[0])+1 # bin number
        p_x_y8_d1 = p_k8[0,l_y8_d1-1]

        l_y8_d2 = np.floor(x[1]/dx[1])+1 # bin number
        p_x_y8_d2 = p_k8[1,l_y8_d2-1]

        p_y8_x = p_x_y8_d1*p_x_y8_d2*p8

        # argmax (p_y3_x, p_y8_x)
        #print p_y3_x
        if p_y3_x>p_y8_x :
            prediction[i] = 3
        else:
            prediction[i] = 8
        # end if

    # end for i
    return prediction
#end def naiveBayesClassifier
```

2.2 Generate threes

To construct a new digits we applied our learning algorithm to the full dimension training set (81 dimension) to get the likelihood for the class of threes. The sampling algorithm we used is :

- Since we assumed, that all features are independent, we can sample in each dimension separately
- So for each dimension calculate the cumulative distribution function (CDF); pick the uniform distributed number in $\alpha \in [0, 1)$; calculate for each $CDF_j, j = 1 \dots d$, α-quantil; return the rounded value of the α-quantil as the jth component of the new number.

Here are some results of the described generation function and the code of the function:

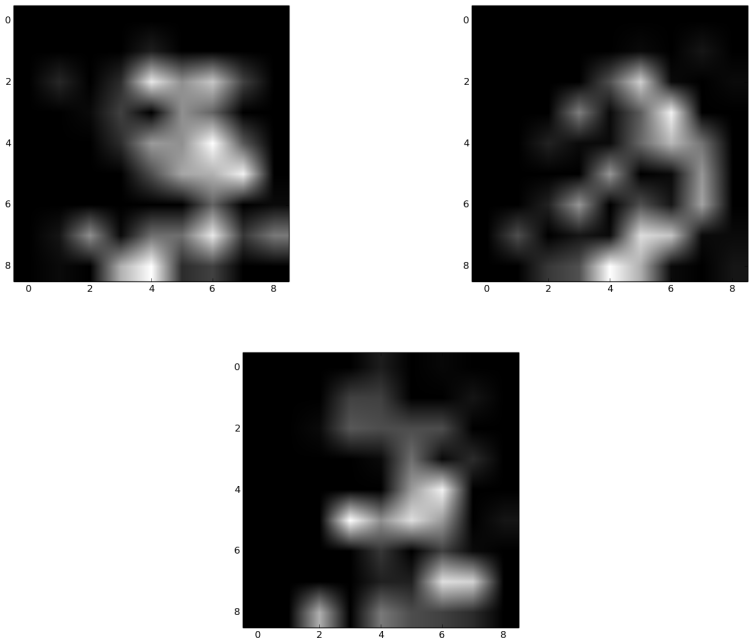


Figure 3: Generated threes

```
# Generate Number from the given pdf
# samply in each of d dimensions independently
def generate_number(pdf,dx):
```

```
    d = pdf.shape[0]    # number of dimension

    newnumber = np.zeros(d, dtype = np.int32)
    # calculate cumulative distribution function (cdf) from pdf
    cdf = np.zeros(pdf.shape, dtype = np.float32)
    for j in range(0, d):
        cdf[j,:] = np.cumsum(pdf[j,:])
    # end for

    for j in range(0,d):
        # randomly select a uniformly distribut number in range [0., 1.)
        alpha = random.random()
        # calculate quantile on the level alpha
        dist = abs(cdf[j,:] - alpha)
        binx = np.argsort(dist)

        newnumber[j] = np.floor(dx[j]*binx[0])+1
    # for j
    return newnumber
# def generate_number(pdf)
```

The whole code to the naive Bayes part can be found in the file *naiveBayes.py*.

3 Density trees

As opposite to the naive Bayes method density tree method should keep the possible interactions between features.

3.1 Building the DT

We wrote recursive function *DT_cut*, which split each node of the tree in two parts and call itself in each of two new nodes. The inputs arguments are the whole number of points in the training set, list of the leaves nodes, current node, depth of the current node and split method.
The first call is made from the function $DT\_learning(trainingx, trainingy, c, splitmethod)$ which creates the root node - node containing all points of the training set.
The result of the learning is returned in form of the list with leaves nodes.
We tried out different termination criteria, such as restriction of the allowed depth or minimum density/number of points in the leaf nodes, but were not satisfied with none of them. The restriction of the allowed depth was often not sufficient to obtain a good probability distribution. Condition on minimum density/number of points led sometimes to overflow of the stack in cases of few points in a big bin. So we chose a combination of the minimum density condition supported with the condition, that N_{bin}/N is not already to small. The idea is, that if N_{bin}/N is already too small, the probability N_{bin}/NV_{bin} would be even smaller and that would lead at the end to the distribution with values around zero.

```
#               Learning DT
# we consider one class at time

def DT_learning(trainingx, trainingy, c, splitmethod):
    # in this version wir use naive split criterion on the nodes of the DT
    print "Learning DT for the class {}". format(c)

    n = trainingx.shape[0]      # size of the training set
    d = trainingx.shape[1]      # size of the feature space

    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]
    nc = xc.shape[0]

    ## Priors
    prior = nc/float(n)

    ## Root node
    region = np.zeros((d,2), dtype = np.float32)
    for j in range(0,d):
        region[j,0] = np.min(xc[:,j])
        region[j,1] = np.max(xc[:,j])
    # end j

    leavesnodes = []
#     build recursively a Density Tree and get all it's leaves
    rootnode = DTnode(xc, 1/volume(region), region)
    DT_cut(nc, leavesnodes, rootnode, 0, splitmethod)

    return prior,leavesnodes
#end def DT_learning_naive



# Recursive call for node splitting
def DT_cut(n, leaveslist, parentnode, depth, splitmethod):

    # if termination condition is satisfied:
    pointsdensity = parentnode.points.shape[0]/float(n)
    # if min density is reached or region has only few points
    if parentnode.p>=0.0001 or pointsdensity<0.001:
        leaveslist.append(parentnode)
        return
    else: # if split further
        if splitmethod == 'naive':
            # split value : split on the middle of the interval
            splitval, splitdim = splitnaive(parentnode, depth)
        # end if naive
        else :
```

```
            # select theoretically best split
            splitval, splitdim = splitclever(parentnode)
        # end if clever

        # new regions
        regionL = np.copy(parentnode.region)
        regionL[splitdim,:] = [regionL[splitdim,0], splitval]

        regionR = np.copy(parentnode.region)
        regionR[splitdim,:] = [splitval, regionR[splitdim,1] ]

        # split poins of the node according to the new regions
        pointsL, pointsR = splitpoints(parentnode.points,parentnode.region, \
                                       splitval, splitdim)
        if pointsL.size ==0:
            nleft = 0
        else:
            nleft = pointsL.shape[0]
        # end if

        if pointsR.size ==0:
            nright = 0
        else:
            nright = pointsR.shape[0]
        # end if

        # calculate density of the new nodes
        pL = nleft/float(n)/volume(regionL)
        pR = nright/float(n)/volume(regionR)

        # create two new nodes
        nodeL = DTnode(pointsL, pL, regionL)
        nodeR = DTnode(pointsR, pR, regionR)

        DT_cut(n, leaveslist, nodeL, depth+1, splitmethod)
        DT_cut(n, leaveslist, nodeR, depth+1, splitmethod)
    # end if
# end DT_cut()
```

The task of the exercise was to implement two splitting methods: naive one and the theoretically best one. The naive splitting method split each node in the middle (dimension of the splitting is changed in circle from 1 to d). The theoretically best criterion tries by each splitting $d(2N_c-2)$ possible split values and selects one, that maximizes the loss function (the algorithm was taken from the lecture notes).
Unfortunately our implementation of the theoretically best splitting is very slow, so we provide the classification results only for naive splitting method, but include the code for the theoretically best splitting as well.

```
#                            Splitting criteria
# split on the middle of the interval in next dimension
def splitnaive(node, depth):
    d = node.points.shape[1]

    splitdim = depth%d;
    splitval =  np.sum(node.region[splitdim,:])/2.

    return splitval, splitdim
# end splitnaive

# select theoretically best split
def splitclever(node):
    x = node.points
    d = x.shape[1]
    n = x.shape[0]
    eps = 0.001

    loss = np.zeros((2*n,d), dtype = np.float32 )
    for j in range(0,d):
        splitdim = j
        ind = np.argsort(node.points[:,j])
        for i in range(0,n):
            for s in [-1,1]:
                splitval = x[ind[i],j]+s*eps
                # new regions
                regionL = np.copy(node.region)
                regionL[splitdim,:] = [regionL[splitdim,0], splitval]

                regionR = np.copy(node.region)
                regionR[splitdim,:] = [splitval, regionR[splitdim,1] ]

                if volume(regionL)==0 or volume(regionR==0):
                    continue

                # split poins of the node according to the new regions
                pointsL, pointsR = splitpoints(x,node.region, \
                                                splitval, splitdim)
                if pointsL.size ==0:
                    nleft = 0
                else:
                    nleft = pointsL.shape[0]
                # end if

                if pointsR.size ==0:
                    nright = 0
                else:
                    nright = pointsR.shape[0]
                # end if
```

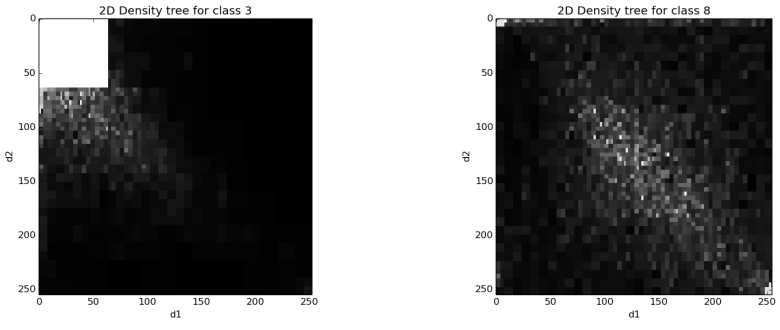
```
                loss[2*i+(s+1)/2,j] = np.square(nleft/float(n))/volume(regionL) + \
                                      np.square(nright/float(n))/volume(regionR)
            # end for s
        # end for i
    #end for j
    maxval = loss.max()
    splitval, splitdim  = np.where(loss==maxval[0])
    print maxval
    print splitval, splitdim

    return splitval, splitdim
# end splitnaive
```

3.2 Classification

The classification task by the density trees as by naive Bayes consist of the calculation of the probability $p(y = k|x)$ and selecting k, which maximizes this value. But in opposite to the naive Bayes method

$$p(y = k|x) = \frac{p(x|y = k)p(y = k)}{p(x)}$$

. To calculate the probability $p(x|y = k)$ we traverse all leaf nodes of the density tree to find for a given x a bin, where it belongs to. In this case $p(x|y = k) = N_{bin}/NV_{bin}$, where N_{bin} is the number of points in the found bin and V_{bin} is its volume.
For the given test set we obtained the correct classification rate 0.8353 (error rate 0.1647).



(a) Class of handwritten threes (b) Class of handwritten eights

Figure 4: Adaptive bins for the density trees with naive splitting criterion

```
def DT_Classifier_2classes(testx, prior1, prior2, DT1, DT2, c = [3,8]):
    n = testx.shape[0]

    prediction = np.zeros(n, dtype = np.int8)

    for i in range(0,n):
        x = testx[i,:]

        # p(y = 3| x)
        # find right bin in DT:
        likelihood1 = 0
        for node in DT1:
            if point_in_region(x, node.region):
                likelihood1 = node.p
                break
            # end if
        # end for node
        p_y1_x = likelihood1*prior1

        # p(y = 8| x)
        # find right bin in DT:
        likelihood2 = -1
        for node in DT2:
            if point_in_region(x, node.region):
                likelihood2 = node.p
                break
            # end if
        # end for node
        p_y2_x = likelihood2*prior2

        # argmax (p_y3_x, p_y8_x)
        if p_y1_x>p_y2_x :
            prediction[i] = c[0]
        else:
            prediction[i] = c[1]
        # end if

    # end for i

    return prediction
# end DT_Classifier
```

3.3 Generate threes

For generation of new digits from the given likelihood function we implemented the following algorithm:

- use the whole dimensional training set to train the density tree for the digit 3

- start from the root node of the DT
- with the selected probability q go left in the tree and with probability $1-q$ go right; set $q = q * N_left/N$
- repeat selection of the next node until a leaf node is reached
- in the reached leaf node sample the points uniformly for each dimension

Here is the code to the described algorithm: The traversal of the tree is done recursive with function $DT\_traverse$.

```
## Generate Number from the given pdf
# samply in each of d dimensions independently
def generate_number(DT, trainingx,trainingy, c):
    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]
    N = xc.shape[0]
    d = xc.shape[1]

    ## Root node
    region = np.zeros((d,2), dtype = np.float32)
    for j in range(0,d):
        region[j,0] = np.min(xc[:,j])
        region[j,1] = np.max(xc[:,j])
    # end j

    rootnode = DTnode(xc, 1/volume(region), region)
    pmin = 1.
    pmax = 0.
    for node in DT:
        if node.p>pmax:
            pmax = node.p
        if node.p<pmin:
            pmin = node.p
    #end for

    # number of all points, probability to go left, root node, depth
    selectednode = DT_traverse(N, pmin, pmax, 0.5, rootnode, 0)
    # in the leaf node sample uniformly in each direction
    newnumber = np.zeros(d, dtype = np.int32)
    for j in range(0,d):
        a = selectednode.region[j,0]
        b = selectednode.region[j,1]

        alpha = random.random()
        # transforme random number in [0,1) in random number in [a,b)
        newnumber[j] = np.floor(a + alpha*(b-a))
    # for j
```

```
    return newnumber
# def generate_number(pdf)



# traverse the density tree: go left with probability q and right with
# probability 1-q, generate a uniform distributed number in range [qmin, qmax)
def DT_traverse(N, qmin, qmax, q, parentnode, depth):

    # if termination condition is satisfied (same as in construction of DT)
    pointsdensity = parentnode.points.shape[0]/float(N)
    if parentnode.p>=0.0001 or pointsdensity<0.001:
        return parentnode
    else:
        # split value : split on the middle of the interval

        splitval, splitdim = splitnaive(parentnode, depth)

        # split poins of the node according to the new regions
        pointsL, pointsR = splitpoints(parentnode.points,parentnode.region,\
                                               splitval, splitdim)

        x = random.uniform(qmin,qmax)
        if x<=q:
            # split region
            region = np.copy(parentnode.region)
            region[splitdim,:] = [region[splitdim,0], splitval]
            points = pointsL
        else:
            # split region
            region = np.copy(parentnode.region)
            region[splitdim,:] = [splitval, region[splitdim,1] ]
            points = pointsR
        # else if

        # calculate new p
        q *= points.shape[0]/float(N)

        # go deeper
        node = DTnode(points, points.shape[0]/float(N)/volume(region), region)
        return DT_traverse(N, qmin, qmax, q, node, depth+1)
    # end if
# end DT_traverse()
```

Unfortunately, as you can seen on the next images this algorithm didn't provide good generation results.

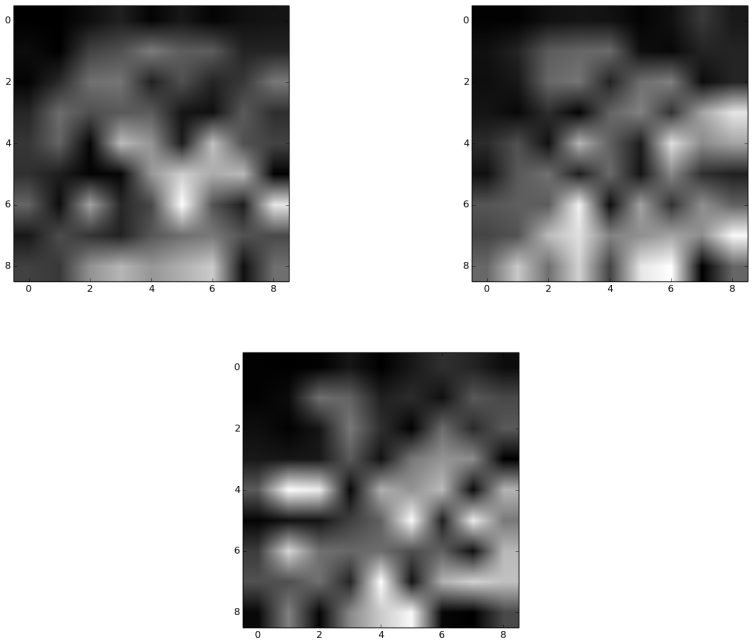


Figure 5: Generated threes

All function from this section can be found in the *densityTree.py* file in attachment.

4 Combination of the density tree and Naive Bayes

Unfortunately we didn't finished the implementation of the tasks in this section. That is why we provide only learning step for the method, that combines density tree with the naive Bayes method.

```
    print
    print "Classification"
    print

# train 1D-histogramms for each feature and class

    tstart = time.time()
    L, dx = chooseBinSize(images_train_38) # number of bins, bins size
```

```
# pdf d x L matrices
prior3, pdf3 = naiveBayes_train_single_class(images_train_38, \
                                             labels_train_38, 3, dx, L)
prior8, pdf8 = naiveBayes_train_single_class(images_train_38, \
                                             labels_train_38, 8, dx, L)
# compute the cdf of each histogramm
cdf3 = np.zeros(pdf3.shape, dtype = np.float32)
cdf8 = np.zeros(pdf8.shape, dtype = np.float32)
for j in range(0, d):
    cdf3[j,:] = np.cumsum(pdf3[j,:])
    cdf8[j,:] = np.cumsum(pdf8[j,:])
# end for

tstop = time.time()
print "Learning 1D histograms and computing cdf's took {} sec".\
                                                    format(tstop-tstart)

# map data to copula using rank order transformation
u = np.zeros(images_train_38.shape, dtype = np.float32)
for j in range(0,d):
    ind = np.sort(images_train_38[:,j])
    u[:,j] = ind[:]/float(n+1)
# end for j

# train a DT on u
tstart = time.time()

prior3, DT3 = DT_learning(u, labels_train_38, 3, 'naive')
prior8, DT8 = DT_learning(u, labels_train_38, 8, 'naive')

tstop = time.time()
print "Learning DTs took {} sec". format(tstop-tstart)

print
print "Classification"
print

ntest = images_test.shape[0]
prediction = np.zeros(ntest, dtype = np.int8)

for i in range(0,n):
    x = images_test[i,:]
    // u_j = F_j(x_j)
    u3 = np.zeros(d, dtype = np.float32)
    u8 = np.zeros(d, dtype = np.float32)

    # compute density according to naive Bayes method
    naiveBayesDensity3 = 1.
    naiveBayesDensity8 = 1.
    for j in range(0, d):
```

```
            l= np.floor(x[j]/dx[j])+1 # bin number
            if l>L-1:
                l=L-1
            naiveBayesDensity3 *= pdf3[j,l]
            naiveBayesDensity8 *= pdf8[j,l]

            u3[j]= cdf3[j,l]
            u8[j]= cdf8[j,l]
        # end for j

        copulaDensity3 = 0
        for node in DT3:
            if point_in_region(u3, node.region):
                copulaDensity3 = node.p
                break
            # end if
        # end for node

        copulaDensity8 = 0
        for node in DT8:
            if point_in_region(u8, node.region):
                copulaDensity8 = node.p
                break
            # end if
        # end for node

        p_y3_x = naiveBayesDensity3*copulaDensity3*prior3
        p_y8_x = naiveBayesDensity8*copulaDensity8*prior8

        # argmax (p_y3_x, p_y8_x)
        #print p_y3_x
        if p_y3_x>p_y8_x :
            prediction[i] = 3
        else:
            prediction[i] = 8
        # end if
    # end for i
```

5 Complete code

5.1 Naive Bayes method

../naiveBayes.py

```
# -*- coding: utf-8 -*-
"""
 Naive Bayes
```

```
"""
import numpy as np
import random

#-------------------------------------------------------------------------
#                        Choose proper bin size
def chooseBinSize(trainingx):

    n = trainingx.shape[0]
    d = trainingx.shape[1]

    # Choose bin width
    dx = np.zeros(d, dtype = np.float128)
    m = np.zeros(d, dtype = np.int32)
    for j in range(0,d):
        # for each dimension apply Freeman-Diace Rule
        ind_sort =  np.argsort(trainingx[:,j]); # j-th feature dimension
        IQR = trainingx[ind_sort[3*n/4],j] - trainingx[ind_sort[n/4],j]
        dx[j] = 2*IQR/np.power(n, 1/3.)
        if dx[j]<0.01:
           dx[j] =  3.5/np.power(n, 1/3.)
        m_j = (np.max(trainingx[:,j])-np.min(trainingx[:,j]))/dx[j]
        m[j] = np.floor(m_j) + 1
    # end for j

    L = np.min(m);  # total number of bins as minimum over all bin sizes
                    # in all dimensions
#    print 'Total number of bins {}'. format(L)

    # recalculate bin width according to the new bin size L
    for j in range(0,d):
        dx[j] = (np.max(trainingx[:,j])-np.min(trainingx[:,j]))/float(L-1)
    # end for j

    return L, dx
# end chooseBinSize

#-------------------------------------------------------------------------
##                          Naive Bayes Training
# determine priors  and likelihoods (for each feature and class individual
# histogram <=> 4 histogramms   )
def naiveBayes_train_single_class(trainingx, trainingy, c, dx, L):
    # we consider one class c
    #
    # trainingx is our training set
    # trainingy are class labels for each element from trainingx
    # dx bin width
    # L total number of bins pro dimension

    n = trainingx.shape[0]      # size of the training set
```

```
    d = trainingx.shape[1]      # size of the feature space

    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]        # Class of digit c
    nc = xc.shape[0]

    ## Priors
    prior = nc/float(n)

    ## Likelihood p(x|y=c)

    likelihood = np.zeros((d, L), dtype = np.float32)

    for j in range(0,d):
        for i in range(0,nc):
            l = np.floor(xc[i,j]/dx[j])+1 # bin
            if l>=L+1:
                print xc[i,j]
            likelihood[j, l-1] = likelihood[j, l-1] + 1
        # end for i=1..nc
        likelihood[j,:] = likelihood[j,:]/float(nc)
    #end for j=1..d

    return prior, likelihood
#end def naiveBayes_train

#-------------------------------------------------------------------------
##                          Naive Bayes Classifier
#
def naiveBayesClassifier(testx, p3, p8, p_k3, p_k8, dx):
    n = testx.shape[0]

    prediction = np.zeros(n, dtype = np.int8)

    for i in range(0,n):
        x = testx[i,:]

        # p(y = 3| x)

        l_y3_d1 = np.floor(x[0]/dx[0])+1 # bin number
        p_x_y3_d1 = p_k3[0,l_y3_d1-1]

        l_y3_d2 = np.floor(x[1]/dx[1])+1 # bin number
        p_x_y3_d2 = p_k3[1,l_y3_d2-1]

        p_y3_x = p_x_y3_d1*p_x_y3_d2*p3

        # p(y = 8| x)

        l_y8_d1 = np.floor(x[0]/dx[0])+1 # bin number
```

```
            p_x_y8_d1 = p_k8[0,l_y8_d1-1]

            l_y8_d2 = np.floor(x[1]/dx[1])+1 # bin number
            p_x_y8_d2 = p_k8[1,l_y8_d2-1]

            p_y8_x = p_x_y8_d1*p_x_y8_d2*p8

            # argmax (p_y3_x, p_y8_x)
            #print p_y3_x
            if p_y3_x>p_y8_x :
                prediction[i] = 3
            else:
                prediction[i] = 8
            # end if

        # end for i
        return prediction
    #end def naiveBayesClassifier

    #-------------------------------------------------------------------------
    ## Generate Number from the given pdf
    # samply in each of d dimensions independently
    def generate_number(pdf,dx):

        d = pdf.shape[0]    # number of dimension

        newnumber = np.zeros(d, dtype = np.int32)
        # calculate cumulative distribution function (cdf) from pdf
        cdf = np.zeros(pdf.shape, dtype = np.float32)
        for j in range(0, d):
            cdf[j,:] = np.cumsum(pdf[j,:])
        # end for

        for j in range(0,d):
            # randomly select a uniformly distribut number in range [0., 1.)
            alpha = random.random()
            # calculate quantile on the level alpha
            dist = abs(cdf[j,:] - alpha)
            binx = np.argsort(dist)

            newnumber[j] = np.floor(dx[j]*binx[0])+1
        # for j
        return newnumber
    # def generate_number(pdf)
```

5.2 Density tree method

../densityTree.py

```
# -*- coding: utf-8 -*-
```

```
"""
Density tree
"""

import numpy as np
import matplotlib.pyplot as plot
import random
#import DTnode

from collections import namedtuple
DTnode = namedtuple("DTnode", "points p region")

# volume of a region
def volume(region):
    V = 1.
    d = region.shape[0]
    for j in range(0,d):
        V *= region[j,1]-region[j,0]
    # end j
    return V
#end volume

def point_in_region(x, region):
    flag = True
    d = region.shape[0]
    for j in range(0,d):
        if (x[j]<region[j,0] or x[j]>region[j,1]):
            flag = False
            return flag
        # end if
    # end j
    return flag
# end point_in_region
#-------------------------------------------------------------------------------
def splitpoints(points,region,splitval, splitdim):

    # create two arrays bigger than we actually need
    # we will delete zero entries afterwards
    pointsLeft = np.zeros(points.shape, dtype = np.int32)
    pointsRight = np.zeros(points.shape, dtype = np.int32)

    nLeft = 0
    nRight = 0

    for i in range(0, points.shape[0]):
        if points[i,splitdim]< splitval:
            pointsLeft[nLeft,:] = points[i,:]
            nLeft += 1
        else :
            pointsRight[nRight,:] = points[i,:]
```

```
            nRight += 1
        # end if
    #end for

    pointsLeft = pointsLeft[0:nLeft,:]
    pointsRight = pointsRight[0:nRight,:]

    assert pointsLeft.shape[0]+pointsRight.shape[0]==points.shape[0],\
                                        'Wrong splitting of points!'

    return pointsLeft, pointsRight
#end point_in region

#-------------------------------------------------------------------------------
#                             Splitting criteria
# split on the middle of the interval in next dimension
def splitnaive(node, depth):
    d = node.points.shape[1]

    splitdim = depth%d;
    splitval =  np.sum(node.region[splitdim,:])/2.

    return splitval, splitdim
# end splitnaive

# select theoretically best split
def splitclever(node):
    x = node.points
    d = x.shape[1]
    n = x.shape[0]
    eps = 0.001

    loss = np.zeros((2*n,d), dtype = np.float32 )
    for j in range(0,d):
        splitdim = j
        ind = np.argsort(node.points[:,j])
        for i in range(0,n):
            for s in [-1,1]:
                splitval = x[ind[i],j]+s*eps
                # new regions
                regionL = np.copy(node.region)
                regionL[splitdim,:] = [regionL[splitdim,0], splitval]

                regionR = np.copy(node.region)
                regionR[splitdim,:] = [splitval, regionR[splitdim,1] ]

                if volume(regionL)==0 or volume(regionR==0):
                    continue

                # split poins of the node according to the new regions
```

```
                pointsL, pointsR = splitpoints(x,node.region, \
                                                splitval, splitdim)
                if pointsL.size ==0:
                    nleft = 0
                else:
                    nleft = pointsL.shape[0]
                # end if

                if pointsR.size ==0:
                    nright = 0
                else:
                    nright = pointsR.shape[0]
                # end if


                loss[2*i+(s+1)/2,j] = np.square(nleft/float(n))/volume(regionL) + \
                            np.square(nright/float(n))/volume(regionR)
            # end for s
        # end for i
    #end for j
    maxval = loss.max()
    splitval, splitdim  = np.where(loss==maxval[0])
    print maxval
    print splitval, splitdim

    return splitval, splitdim
# end splitnaive
#-------------------------------------------------------------------------------
#               Learning DT
# we consider one class at time

def DT_cut(n, leaveslist, parentnode, depth, splitmethod):

    # if termination condition is satisfied:
    pointsdensity = parentnode.points.shape[0]/float(n)
    # if min density is reached or region has only few points
    if parentnode.p>=0.0001 or pointsdensity<0.001: # if maximal depth of the tree is
#    if depth>5:
        leaveslist.append(parentnode)
        return
    else: # if split further
        if splitmethod == 'naive':
            # split value : split on the middle of the interval
            splitval, splitdim = splitnaive(parentnode, depth)
        # end if naive
        else :
            # select theoretically best split
            splitval, splitdim = splitclever(parentnode)
        # end if clever
```

```
        # new regions
        regionL = np.copy(parentnode.region)
        regionL[splitdim,:] = [regionL[splitdim,0], splitval]

        regionR = np.copy(parentnode.region)
        regionR[splitdim,:] = [splitval, regionR[splitdim,1] ]

        # split poins of the node according to the new regions
        pointsL, pointsR = splitpoints(parentnode.points,parentnode.region, \
                                                splitval, splitdim)
        if pointsL.size ==0:
            nleft = 0
        else:
            nleft = pointsL.shape[0]
        # end if

        if pointsR.size ==0:
            nright = 0
        else:
            nright = pointsR.shape[0]
        # end if

        # calculate density of the new nodes
        pL = nleft/float(n)/volume(regionL)
        pR = nright/float(n)/volume(regionR)

        # create two new nodes
        nodeL = DTnode(pointsL, pL, regionL)
        nodeR = DTnode(pointsR, pR, regionR)

        DT_cut(n, leaveslist, nodeL, depth+1, splitmethod)
        DT_cut(n, leaveslist, nodeR, depth+1, splitmethod)
    # end if
# end DT_cut()

def DT_learning(trainingx, trainingy, c, splitmethod):
    # in this version wir use naive split criterion on the nodes of the DT
    print "Learning DT for the class {}". format(c)

    n = trainingx.shape[0]      # size of the training set
    d = trainingx.shape[1]      # size of the feature space

    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]
    nc = xc.shape[0]

    ## Priors
    prior = nc/float(n)

    ## Root node
```

```
    region = np.zeros((d,2), dtype = np.float32)
    for j in range(0,d):
        region[j,0] = np.min(xc[:,j])
        region[j,1] = np.max(xc[:,j])
    # end j

    leavesnodes = []
#     build recursively a Density Tree and get all it's leaves
    rootnode = DTnode(xc, 1/volume(region), region)
#    rootnode = DTnode(xc, 1, region)
    DT_cut(nc, leavesnodes, rootnode, 0, splitmethod)

    return prior,leavesnodes
#end def DT_learning_naive
#-------------------------------------------------------------------------------

def DT_Classifier_2classes(testx, prior1, prior2, DT1, DT2, c = [3,8]):
    n = testx.shape[0]

    prediction = np.zeros(n, dtype = np.int8)

    for i in range(0,n):
        x = testx[i,:]

        # p(y = 3| x)
        # find right bin in DT:
        likelihood1 = 0
        for node in DT1:
            if point_in_region(x, node.region):
                likelihood1 = node.p
                break
            # end if
        # end for node
        p_y1_x = likelihood1*prior1

        # p(y = 8| x)
        # find right bin in DT:
        likelihood2 = -1
        for node in DT2:
            if point_in_region(x, node.region):
                likelihood2 = node.p
                break
            # end if
        # end for node
        p_y2_x = likelihood2*prior2

        # argmax (p_y3_x, p_y8_x)
        if p_y1_x>p_y2_x :
            prediction[i] = c[0]
        else:
```

```
            prediction[i] = c[1]
        # end if

    # end for i


    return prediction
# end DT_Classifier
#-------------------------------------------------------------------------
def DT_visualize2D(leaveslist, trainingx, trainingy, c, saveName):

    d = trainingx.shape[1]      # size of the feature space

    assert d==2, 'I can visualize density trees only for two features :('

    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]

    d1min = np.ceil(np.min(xc[:,0]))
    d1max = np.ceil(np.max(xc[:,0]))

    d2min = np.ceil(np.min(xc[:,1]))
    d2max = np.ceil(np.max(xc[:,1]))

    img = np.zeros((d1max-d1min, d2max-d2min), dtype = np.float64)

    pmax = 0.
    for node in leaveslist:
        if node.p>pmax:
            pmax = node.p
        xmin = np.ceil(node.region[0,0])
        xmax = np.ceil(node.region[0,1])

        ymin = np.ceil(node.region[1,0])
        ymax = np.ceil(node.region[1,1])

        img[xmin:xmax, ymin:ymax] = node.p
    #end for

    im = np.array(img * 255/pmax, dtype = np.uint8)

    f = plot.figure()
    plot.gray()
    plot.imshow(im.transpose(), interpolation = 'nearest')
    plot.title("2D Density tree for class %d" %c)
    plot.xlabel('d1')
    plot.ylabel('d2')

    plot.show()
```

```
    f.savefig(saveName)
# end DT_visualize(DT)
#-------------------------------------------------------------------------------
# traverse the density tree: go left with probability q and right with
# probability 1-q
def DT_traverse(N, qmin, qmax, q, parentnode, depth):

    # if termination condition is satisfied (same as in construction of DT)
    pointsdensity = parentnode.points.shape[0]/float(N)
    if parentnode.p>=0.0001 or pointsdensity<0.001:
#     if depth>5:
        return parentnode
    else:
        # split value : split on the middle of the interval

        splitval, splitdim = splitnaive(parentnode, depth)

        # split poins of the node according to the new regions
        pointsL, pointsR = splitpoints(parentnode.points,parentnode.region, splitval,

        x = random.uniform(qmin,qmax)
        if x<=q:
            # split region
            region = np.copy(parentnode.region)
            region[splitdim,:] = [region[splitdim,0], splitval]
            points = pointsL
        else:
            # split region
            region = np.copy(parentnode.region)
            region[splitdim,:] = [splitval, region[splitdim,1] ]
            points = pointsR
        # else if

        # calculate new p
        q *= points.shape[0]/float(N)

        # go deeper
        node = DTnode(points, points.shape[0]/float(N)/volume(region), region)
        return DT_traverse(N, qmin, qmax, q, node, depth+1)
    # end if
# end DT_traverse()

## Generate Number from the given pdf
# samply in each of d dimensions independently
def generate_number(DT, trainingx,trainingy, c):
    # find  in training set all members of the class c
    xc = trainingx[trainingy==c, :]
    N = xc.shape[0]
    d = xc.shape[1]
```

```
    ## Root node
    region = np.zeros((d,2), dtype = np.float32)
    for j in range(0,d):
        region[j,0] = np.min(xc[:,j])
        region[j,1] = np.max(xc[:,j])
    # end j

    rootnode = DTnode(xc, 1/volume(region), region)
    pmin = 1.
    pmax = 0.
    for node in DT:
        if node.p>pmax:
            pmax = node.p
        if node.p<pmin:
            pmin = node.p
    #end for

    # number of all points, probability to go left, root node, depth
    selectednode = DT_traverse(N, pmin, pmax, 0.5, rootnode, 0)
    # in the leaf node sample uniformly in each direction
    newnumber = np.zeros(d, dtype = np.int32)
    for j in range(0,d):
        a = selectednode.region[j,0]
        b = selectednode.region[j,1]

        alpha = random.random()
        # transforme random number in [0,1) in random number in [a,b)
        newnumber[j] = np.floor(a + alpha*(b-a))
    # for j
    return newnumber
# def generate_number(pdf)
```

5.3 Main function and Task 3

../ex04.py

```
"""
Exercise 4 : Generative Non-parametric Classification

"""

import numpy as np
import matplotlib.pyplot as plot
import vigra

import time

from correctClassificationRate import correctClassRate

from naiveBayes import chooseBinSize
```

```
from naiveBayes import naiveBayes_train_single_class
from naiveBayes import naiveBayesClassifier
from naiveBayes import generate_number as generate3naiveBayes

from densityTree import point_in_region
from densityTree import DT_learning
from densityTree import DT_visualize2D
from densityTree import DT_Classifier_2classes
from densityTree import generate_number as generate3DT
#-------------------------------------------------------------------------------
#                           Dimension Reduction Function
# size(x) = n x d.
# size(dr(x)) = n x 2
def dr(x, y, d=[3,8]):
    # Calculate the average digit d[0]
    x_1 = x[y==d[0]]
    n1 = len(x_1)
    average1 = np.sum(x_1[:,:], axis = 0)/float(n1)


    # Calculate the average digit d[1]
    x_2 = x[y==d[1]]
    n2 = len(x_2)
    average2 = np.sum(x_2[:,:], axis = 0)/float(n2)

    # Differences between average1 and average7
    diff = np.abs(average1-average2)

    # Sort in descending order
    diff_sortInd = np.argsort(diff);
    diff_sortInd = diff_sortInd[::-1]

    # leave only indices of the two first elements
    # It means, we choose two dimensions, where average digits have highest
    # difference
    diff_sortInd = diff_sortInd[0:2:1]

    xn = x[:,diff_sortInd]

    return xn

#end def dr

#-------------------------------------------------------------------------------
#                    Plot 1D histograms
def plot_histogram(pdf,dx, title = 'Histograms for each of d dimensions',\
                                              imageName = 'histogram.png'):

    m = pdf.shape[1]
    f, (ax1, ax2) = plot.subplots(1, 2, sharey=True)
```

```
    f.suptitle(title, fontsize=14)
    ax1.set_xlabel('x')
    ax1.set_ylabel('Probability')

    ax2.set_xlabel('x')
    ax1.set_ylabel('Probability')

    ax1.set_title('d=1')
    ax2.set_title('d=2')


    for i in range(0,m):
        # subplot 1
        ax1.plot([i*dx[0], i*dx[0], (i+1)*dx[0],(i+1)*dx[0]], \
                 [0, pdf[0,i], pdf[0,i], 0 ], 'r-')
        ax1.set_xlim([0,m*dx[0]])
        # subplot 2
        ax2.plot([i*dx[1], i*dx[1], (i+1)*dx[1],(i+1)*dx[1]], \
                 [0, pdf[1,i], pdf[1,i], 0 ], 'b-')
        ax2.set_xlim([0,m*dx[1]])
    # end for

    plot.show()

    f.savefig(imageName)
#end def

#-------------------------------------------------------------------------
#                    Plot 2D Likelihood
def plot_likelihood(pdf,dx, title = 'likelihood',\
                                          imageName = 'likelihood.png'):
    L = pdf.shape[1] # number of bins
    img = np.zeros((L,L), dtype = np.float64)

    for i in range(0,L):
        img[i,:] = pdf[0,:]*pdf[1,i]
    #end for

    f = plot.figure()
    plot.gray()
    plot.imshow(img, interpolation = 'nearest')
    plot.title(title)
    plot.xlabel('d1')
    plot.ylabel('d2')

    plot.show()

    f.savefig(imageName)
#end def
```

```
#------------------------------------------------------------------------------
#                              Main Function
def main():
    plot.close('all')

#       0 Read data, selecting digits 3 and 8, dimension reduction
    print
    print "Read data, selecting digits 3 and 8, dimension reduction"
    print

    test_path     = "test.h5"
    training_path = "train.h5"

    images_train = vigra.readHDF5(test_path, "images")
    labels_train = vigra.readHDF5(test_path, "labels")

    images_test = vigra.readHDF5(training_path, "images")
    labels_test = vigra.readHDF5(training_path, "labels")

    print 'Size of the training set: {}'. format(np.shape(images_train))
    print np.shape(labels_train)
    print 'Size of the test set: {}'. format(np.shape(images_test))
    print np.shape(labels_test)

    # Reshape data

    n = images_train.shape[0]
    d = images_train.shape[1]
    images_train  =  images_train.reshape(n,d*d)

    n = images_test.shape[0]
    assert d!=images_test.shape[0], 'Test and training sets have different dim'
    images_test  =  images_test.reshape(n,d*d)

    # Select 3s and 8s

    ind3 = (labels_train==3)
    ind8 = (labels_train==8)

    images_train_38 = images_train[ind3+ind8]
    labels_train_38 = labels_train[ind3+ind8]

    ind3 = (labels_test==3)
    ind8 = (labels_test==8)

    images_test_38 = images_test[ind3+ind8]
    labels_test_38 = labels_test[ind3+ind8]

    # Dimension reduction
```

```
rimages_train_38 = dr(images_train_38, labels_train_38, [3,8])
rimages_test_38 = dr(images_test_38, labels_test_38, [3,8])

print 'Size of the training set of 3s and 8s: {}'. format(np.shape(rimages_train_
print 'Size of the test set of 3s and 8s: {}'. format(np.shape(rimages_test_38))


print
print "1 Naive Bayes"
print
print "1.1 Classification"
print

# Training: priors and likelihood for each d=1,2
# for each feature and class individual histograms <=> 4 histogramms

n = rimages_train_38.shape[0]
d = rimages_train_38.shape[1]

# Choose bin width
L, dx = chooseBinSize(rimages_train_38)

# train classifier for each class separatly
p3, pdf3 = naiveBayes_train_single_class(rimages_train_38, \
                                         labels_train_38, 3, dx, L)
p8, pdf8 = naiveBayes_train_single_class(rimages_train_38, \
                                         labels_train_38, 8, dx, L)

rimages_test_38_predict = naiveBayesClassifier(rimages_test_38, \
                                       p3, p8, pdf3, pdf8, dx)

ccr_naiveBayes = correctClassRate(rimages_test_38_predict,\
                                  labels_test_38, [3,8], \
                                  print_confMatrix = True)

print 'Correct Classification rate on the test set:{}'.format(ccr_naiveBayes)
print 'Error rate on the test set:{}'.format(1-ccr_naiveBayes)


plot_histogram(pdf3, dx, "Histograms of the class 3", "histograms3.png")
plot_histogram(pdf8, dx, "Histograms of the class 8", "histograms8.png")

plot_likelihood(pdf3, dx, "Likelihoods of the class 3", "likelihoods3.png")
plot_likelihood(pdf8, dx, "Likelihoods of the class 8", "likelihoods8.png")


print
print "1.2 Generate Threes"
print
```

```
# use function  naiveBayes_train_single_class to compute the likelihood
# for all feature dimension

n = images_train_38.shape[0]
d = images_train_38.shape[1]

# Choose bin width
# Choose bin width
L, dx = chooseBinSize(images_train_38)

# train classifier for each class separatly
p3, pdf3 = naiveBayes_train_single_class(images_train_38, \
                                         labels_train_38, 3, dx, L)


# generate 5 new threes
new3th = np.zeros((5,d), dtype = np.int32)
for i in range(0,3) :
    new3th[i,:] = generate3naiveBayes(pdf3, dx)

    img = new3th[i,:].reshape(np.sqrt(d),np.sqrt(d))
    plot.figure()
    plot.gray()
    plot.imshow(img);
    plot.show()
# end for i


print
print "2 Density Tree"
print
print "Naive splitting"
print

# class 3

tstart = time.time()

prior3, DT3 = DT_learning(rimages_train_38, labels_train_38, 3, 'naive')
DT_visualize2D(DT3, rimages_train_38, labels_train_38, 3, "naiveDT3.png")
# class 8
prior8, DT8 = DT_learning(rimages_train_38, labels_train_38, 8, 'naive')
DT_visualize2D(DT8, rimages_train_38, labels_train_38, 8, "naiveDT8.png")

tstop = time.time()
print "DT learning time (naive splitting) {}". format(tstop-tstart)

tstart = time.time()
rimages_test_38_predict = DT_Classifier_2classes(rimages_test_38,
```

```
                                                    prior3, prior8, DT3, DT8, [3,8])
    tstop = time.time()
    print "DT classification time (naive splitting) {}". format(tstop-tstart)

    ccr_DT = correctClassRate(rimages_test_38_predict,\
                                    labels_test_38, [3,8], \
                                    print_confMatrix = True)

    print 'Correct Classification rate on the test set:{}'.format(ccr_DT)
    print 'Error rate on the test set:{}'.format(1-ccr_DT)




    print
    print 'Generate new threes: '
    print
    d = images_train_38.shape[1]
    prior3, DT3 = DT_learning(images_train_38, labels_train_38, 3, 'naive')
    # new threes
    new3th = np.zeros((5,d), dtype = np.int32)
    for i in range(0,3) :
        new3th[i,:] = generate3DT(DT3, images_train_38, labels_train_38, 3 )
        img = new3th[i,:].reshape(np.sqrt(d),np.sqrt(d))

        plot.figure()
        plot.gray()
        plot.imshow(img);
        plot.show()
    # end for i

    print
    print "Clever splitting"
    print

#    # class 3
#    prior3, DT3 = DT_learning(rimages_train_38, labels_train_38, 3, 'clever')
##    DT_visualize2D(DT3, rimages_train_38, labels_train_38, 3, "naiveDT3.png")
#    # class 8
#    prior8, DT8 = DT_learning(rimages_train_38, labels_train_38, 8, 'clever')
##    DT_visu    alize2D(DT8, rimages_train_38, labels_train_38, 8, "naiveDT8.png")
#
#
#    rimages_test_38_predict = DT_Classifier_2classes(rimages_test_38,
#                                                     prior3, prior8, DT3, DT8, [3,8])
#
#    ccr_DT = correctClassRate(rimages_test_38_predict,\
#                                    labels_test_38, [3,8], \
#                                    print_confMatrix = True)
#
```

```
#     print 'Correct Classification rate on the test set:{}'.format(ccr_DT)
#     print 'Error rate on the test set:{}'.format(1-ccr_DT)


    print
    print "3 Combine DT and Naive Bayes"
    print


    n = images_train_38.shape[0]
    d = images_train_38.shape[1]


    print
    print "Learning phase"
    print

    # train 1D-histogramms for each feature and class

    tstart = time.time()
    L, dx = chooseBinSize(images_train_38) # number of bins, bins size

    # pdf dxL matrices
    prior3, pdf3 = naiveBayes_train_single_class(images_train_38, \
                                                 labels_train_38, 3, dx, L)
    prior8, pdf8 = naiveBayes_train_single_class(images_train_38, \
                                                 labels_train_38, 8, dx, L)
    # compute the cdf of each histogramm
    cdf3 = np.zeros(pdf3.shape, dtype = np.float32)
    cdf8 = np.zeros(pdf8.shape, dtype = np.float32)
    for j in range(0, d):
        cdf3[j,:] = np.cumsum(pdf3[j,:])
        cdf8[j,:] = np.cumsum(pdf8[j,:])
    # end for

    tstop = time.time()
    print "Learning 1D histograms and computing cdf's took {} sec".\
                                                        format(tstop-tstart)

    # map data to copula using rank order transformation
    u = np.zeros(images_train_38.shape, dtype = np.float32)
    for j in range(0,d):
        ind = np.sort(images_train_38[:,j])
        u[:,j] = ind[:]/float(n+1)
    # end for j

    # train a DT on u
    tstart = time.time()

    prior3, DT3 = DT_learning(u, labels_train_38, 3, 'naive')
```

```
prior8, DT8 = DT_learning(u, labels_train_38, 8, 'naive')

tstop = time.time()
print "Learning DTs took {} sec". format(tstop-tstart)

print
print "Classification"
print

ntest = images_test.shape[0]
prediction = np.zeros(ntest, dtype = np.int8)

for i in range(0,n):
    x = images_test[i,:]
    u3 = np.zeros(d, dtype = np.float32)
    u8 = np.zeros(d, dtype = np.float32)

    naiveBayesDensity3 = 1.
    naiveBayesDensity8 = 1.
    for j in range(0, d):
        l= np.floor(x[j]/dx[j])+1 # bin number
        if l>L-1:
            l=L-1
        naiveBayesDensity3 *= pdf3[j,l]
        naiveBayesDensity8 *= pdf8[j,l]

        u3[j]= cdf3[j,l]
        u8[j]= cdf8[j,l]
    # end for j

    copulaDensity3 = 0
    for node in DT3:
        if point_in_region(u3, node.region):
            copulaDensity3 = node.p
            break
        # end if
    # end for node

    copulaDensity8 = 0
    for node in DT8:
        if point_in_region(u8, node.region):
            copulaDensity8 = node.p
            break
        # end if
    # end for node

    p_y3_x = naiveBayesDensity3*copulaDensity3*prior3
    p_y8_x = naiveBayesDensity8*copulaDensity8*prior8

    # argmax (p_y3_x, p_y8_x)
```

```
            #print p_y3_x
            if p_y3_x>p_y8_x :
                prediction[i] = 3
            else:
                prediction[i] = 8
            # end if
        # end for i

        ccr = correctClassRate(prediction, labels_test_38, [3,8], \
                                        print_confMatrix = True)

        print 'Correct Classification rate on the test set:{}'.format(ccr)
        print 'Error rate on the test set:{}'.format(1-ccr)

# end main


if __name__ == "__main__":
    main()
```