Machine Learning
Exercise 1

October 24, 2014

Group: Sergej Kraft
Elias Rger
Ekaterina Tikhoncheva

1 Tasks: Running Results and Comments

1.1 Exploring the Data

For purposes of this exercise we downloaded the data set of hand written digits from *http://scikit-learn.org/stable/datasets/*.
This data set consists of 1797 images. Further we show an example of an image from this data set (see Figure 1).

ex01_1.py

```
digits = load_digits()
print digits.keys()

data = digits['data']
images = digits['images']
target = digits['target']
target_names = digits['target_names']

print 'Size of the digit set {}'. format(digits.data.shape)
#print np.dtype(data) # TypeError :data type not understood

# get all images with 3
img3 = images[target == 3 ]
# show the first one
img = img3[0]
assert 2 == np.size(np.shape(img))

plot.figure()
plot.gray();
plot.imshow(img, interpolation = 'nearest');
plot.show()
```

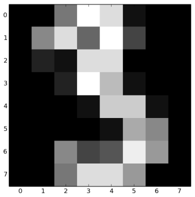

Figure 1: Digit 3

1.2 Nearest Neighbor Classifier

The main aim of this exercise is to implement the k-Nearest Neighbor Classifier and apply it on the digit data set.
First step is to implement a function, that computes Euclidean distance between all digits in training and test set. We did it in two ways: the first implementation uses *for*-loops and the second uses advantages of vectorization. We explain shortly algorithms used in each function.
Assuming, $a_1, a_2, ..a_{d_a}$ and $b_1, b_2, ..b_{d_b}$ are respective our training and test images. Notice that each $a_i, b_j \in \mathbb{R}^d$, where d is dimension of the image space ($d = 64$ in our case). To calculate the distance between a_i and b_j the first function uses the simple formula $dist(a_i, b_j) = \sqrt(a_i^2 - b_j^2)$ (see the code below).

dist_loop.py

```
# Euclidean distance between two sets of points
# realisation with loops
def dist_loop(training, test):

    n1, d = training.shape
    n2, d1 = test.shape

    assert n1 != 0, 'Training set is empty'
    assert n2 != 0, 'Test set is empty'
    assert d==d1, 'Images in training and test sets have different size'

    tstart = time.time()

    dist =  np.zeros((n1,n2), dtype = np.float32)

    for i in range(0,n1):
        for j in range(0,n2):
            diff = training[i,:]-test[j,:]
            dist[i,j] = np.sum(np.square(diff), axis=0)

    dist = np.sqrt(dist)
    tstop = time.time()
    return dist, tstop-tstart
# end dist_loops
```

To avoid slow loops we use a following vectorization in the second function: $dist(a_i, b_j) = \sqrt{(}a_i^2 - b_j^2) = a_i^2 + b_j^2 - 2a_ib_j$ implies the following matrix form

$$dist(A,B) = \begin{bmatrix} a_1^2 & a_1^2 & \cdots & a_1^2 \\ a_2^2 & a_2^2 & \cdots & a_2^2 \\ \vdots & \ddots & \vdots & \\ a_{d_a}^2 & a_{d_a}^2 & \cdots & a_{d_a}^2 \end{bmatrix} + \begin{bmatrix} b_1^2 & b_2^2 & \cdots & b_{d_b}^2 \\ b_1^2 & b_2^2 & \cdots & b_{d_b}^2 \\ \vdots & \ddots & \vdots & \\ b_1^2 & b_2^2 & \cdots & b_{d_b}^2 \end{bmatrix} - 2 \begin{bmatrix} a_1^2 \\ a_2^2 \\ \vdots \\ a_{d_a}^2 \end{bmatrix} \times \begin{bmatrix} b_1^2 & b_2^2 & \cdots & b_{d_b}^2 \end{bmatrix}$$

dist_vec.py

```
# Euclidean distance between two sets of points
# realisation with vectors
def dist_vec(training, test):

    n1, d = training.shape
    n2, d1 = test.shape

    assert n1 != 0, 'Training set is empty'
    assert n2 != 0, 'Test set is empty'
    assert d==d1, 'Images in training and test sets have different size'

    tstart = time.time()

    train_squared = np.sum(np.square(training), axis = 1)
    test_squared = np.sum(np.square(test), axis = 1)

    A = np.tile(train_squared, (n2,1)) # n2xn1 matrix
    A = A.transpose((1,0))     # n1xn2 matrix
    B = np.tile(test_squared, (n1,1) ) # n2xn2 matrix

    a = np.tile(training, (1,1,1)) # 1xn1x64 matrix
    a = a.transpose((1,0,2))     # n1x1x64 matrix
    b = np.tile(test, (1,1,1) ) # 1xn2x64 matrix

    C = np.tensordot(a,b, [[1,2],[0,2]])

    dist = A + B - C - C

    dist = np.sqrt(dist)
    np.float16(dist)

    tstop = time.time()

    return dist, tstop-tstart
# end dist_vec
```

The running time of both function on the same training and data set can be founded in the Table 1.

function	run time
dist_ loop	9.78541 sec
dist_ vec	0.02423 sec

Table 1: Running time of two distance functions

The complete k-Nearest Neighbor Classifier function uses the distance function to find for each image from the test set it's k nearest neighbors in the training set. For $k > 1$ the function collects the votes (classes of the neighbors) and pick the most common one.

kNN.py

```
# k-Nearest Neighbor Classifier (default k=1)
def kNN(x_training, y_training, x_test, k=1):

    nTr, dTr = x_training.shape
    nTest, dTest = x_test.shape

    assert k <= nTr, 'kNN Error: k cannot be larger than size of training set'

    # compute distance between all points in training and test sets
    dist, time = dist_vec(np.array(x_training), np.array(x_test))

    # sort each column of the dist matrix in descending order
    # save indices that would sort the matrix
    dist_sortInd = np.argsort(dist, axis = 0);

    # leave only k nearest neighbors : rows 1:k in the sorted array
    dist_sortInd = dist_sortInd[0:k,:]

    # classification results (for k>1 : the majority vote from k-nearest neighbors)
    y_pred = np.zeros(nTest, dtype = np.int8)
    for i in range(0,nTest):
        votes = y_training[dist_sortInd[:,i]] # k votes
        # take the majority vote in each column
        votes_bin = np.bincount(votes)
        y_pred[i] = np.argmax(votes_bin)
    # end for loop
    return  y_pred
# end kNN
```

Here are results of running the implemented function on the sets of the digits $1, 3$ and $1, 7$ with different k (see Table 2).

set	k	correct classification rate
1 and 3	k=1	0.6
1 and 7	k=1	0.57931
1 and 7	k=3	0.61379
1 and 7	k=5	0.63448
1 and 7	k=9	0.67586
1 and 7	k=17	0.64827
1 and 7	k=33	0.65517

Table 2: Results of kNN-Classifier

We can see that increasing the number of neighbors results in better classification rate. The possible explanation might be that the higher k makes algorithm robust agains outliers.

1.3 Cross-Validation

In this part of the exercise we implement an n-fold cross validation scheme to test our k-Nearest Neighbors classifier.
To be able to do this find a function which splits our digit set in n groups with equal number of elements in each group. We implemented a function which returns the indices of the corresponding images of the dataset in each of n groups :

split_dataset.py

```
# Split the given annotated data in n parts (=2 default)
def split_data_n_equal_parts(x, y, n=2):

    nx, d = x.shape
    ny = y.shape

    assert nx != ny, 'Split data function: x and y sets have different sizes'
    assert nx != 0,  'Split data function: data sets are empty'



    # minimum number of elements in each of n parts
    minnE = nx/n

    # if there not enough element to split data in groups of equal size
    # we add at the end of the data set elements from it's beginning
    if nx%n!=0:
        nE = minnE+1
    else:
        nE = minnE
    # end if
    print '...split into {} groups.Number of elements in each group {}'. format(n, nE
    # number of element to be added
    r = n*nE - nx


    indx = np.zeros((n,nE), dtype = np.int16)
    for i in range(0,nx+r):
        gr = i % n
        if i>=nx:
            e = nE-1
        else :
            e = i/n
        # end if
        indx[gr, e] = i % nx
    #end for-loop

    return indx
# end split_data_n_parts
```

Each of n subparts will be used one time as a test set and the remaining parts as a training set. For each n we calculate the mean classification rate and the variance. Results are shown in the table 3.

n	mean correct classification rate	variance
n=1	0.113464	$7.93933868408e-05$
n=5	0.116699	0.00079345703125
n=10	0.106079101562	0.000761985778809

Table 3: n-fold cross validation of kNN-Classifier, k = 1

2 Complete Code

../ex01.py

```
"""
Created on Mon Oct 20 16:07:47 2014

@author: kitty
"""

import numpy as np
import matplotlib.pyplot as plot

import time

from sklearn.datasets import load_digits
from sklearn import cross_validation
```

```
# Euclidean distance between two sets of points
# ----------------------------------------------------------------------------
# realisation with loops
def dist_loop(training, test):

    n1, d = training.shape
    n2, d1 = test.shape

    assert n1 != 0, 'Training set is empty'
    assert n2 != 0, 'Test set is empty'
    assert d==d1, 'Images in training and test sets have different size'

    tstart = time.time()

    dist =  np.zeros((n1,n2), dtype = np.float32)

    for i in range(0,n1):
        for j in range(0,n2):
            diff = training[i,:]-test[j,:]
            dist[i,j] = np.sum(np.square(diff), axis=0)

    dist = np.sqrt(dist)
    tstop = time.time()

    return dist, tstop-tstart
# end dist_loops
# ----------------------------------------------------------------------------
# realisation with vectors
def dist_vec(training, test):

    n1, d = training.shape
    n2, d1 = test.shape

    assert n1 != 0, 'Training set is empty'
    assert n2 != 0, 'Test set is empty'
    assert d==d1, 'Images in training and test sets have different size'

    tstart = time.time()

    train_squared = np.sum(np.square(training), axis = 1)
    test_squared = np.sum(np.square(test), axis = 1)

    A = np.tile(train_squared, (n2,1)) # n2xn1 matrix
    A = A.transpose((1,0))     # n1xn2 matrix
    B = np.tile(test_squared, (n1,1) ) # n2xn2 matrix

    a = np.tile(training, (1,1,1)) # 1xn1x64 matrix
    a = a.transpose((1,0,2))     # n1x1x64 matrix
    b = np.tile(test, (1,1,1) ) # 1xn2x64 matrix
    C = np.tensordot(a,b, [[1,2],[0,2]])

    dist = A + B - C - C

    dist = np.sqrt(dist)
    np.float16(dist)

    tstop = time.time()

    return dist, tstop-tstart
# end dist_vec
# ----------------------------------------------------------------------------

# k-Nearest Neighbor Classifier (default k=1)
def kNN(x_training, y_training, x_test, k=1):

    nTr, dTr = x_training.shape
    nTest, dTest = x_test.shape

    assert k <= nTr, 'kNN Error: k cannot be larger than size of training set'

    # compute distance between all points in training and test sets
    dist, time = dist_vec(np.array(x_training), np.array(x_test))

    # sort each column of the dist matrix in ascending order
    # save indices that would sort the matrix
    dist_sortInd = np.argsort(dist[::-1], axis = 0, );

    # leave only k nearest neighbors : rows 1:k in the sorted array
    dist_sortInd = dist_sortInd[0:k,:]

    # classification results (for k>1 : the majority vote from k-nearest neighbors)
    y_pred = np.zeros(nTest, dtype = np.int8)
    for i in range(0,nTest):
        votes = y_training[dist_sortInd[:,i]] # k votes
        # take the majority vote in each column
        votes_bin = np.bincount(votes)
        y_pred[i] = np.argmax(votes_bin)
    # end for loop
    return  y_pred
# end kNN
# ----------------------------------------------------------------------------
## Calculate correct classification rate of the k-NN Classifier
# D = [d1, d2,...d10] digits
def correctClassRate(y_pred, y_test, D, print_confMatrix = False):
    n = len(D)

    # calculate confusion matrix
    confusionM = np.zeros((n,n), dtype = np.float16)
    for i in range(0,n):
        # find positions of the digit n in test set

        indn = (y_test == D[i])
        # how often is digit seen in the test set
        nDi = len(y_test[indn])
        # get predicted values on the corresponding positions
        predict = y_pred[indn]
        votes_bin = np.bincount(predict , minlength = 10)
        confusionM[i,:] = np.array(votes_bin[D])
    # end for-loop

    if print_confMatrix:
        print
        print 'Confusion Matrix '
        print confusionM

        print 'Size of the test set = {}'. format(len(y_test))
    # end if print_confMatrix

    # correct classification rate
    ccr = np.trace(confusionM)/len(y_test)
    return  ccr
# end correctClassRate
# ----------------------------------------------------------------------------

# Split the given annotated data in n parts (=2 default)
def split_data_n_equal_parts(x, y, n=2):

    nx, d = x.shape
    ny = y.shape

    assert nx != ny, 'Split data function: x and y sets have different sizes'
    assert nx != 0,  'Split data function: data sets are empty'



    # minimum number of elements in each of n parts
    minnE = nx/n

    # if there not enough element to split data in groups of equal size
    # we add at the end of the data set elements from it's beginning
    if nx%n!=0:
        nE = minnE+1
    else:
        nE = minnE
    # end if
    print '...split into {} groups.Number of elements in each group {}'. format(n, nE

    # number of element to be added
    r = n*nE - nx


    indx = np.zeros((n,nE), dtype = np.int16)
    for i in range(0,nx+r):
        gr = i % n
        if i>=nx:
            e = nE-1
        else :
            e = i/n
        # end if
        indx[gr, e] = i % nx
    #end for-loop

    return indx
# end split_data_n_parts
# ----------------------------------------------------------------------------
# ----------------------------------------------------------------------------
def main():

    plot.close('all')

    print '-------------------------------------------------------------------'
    print ' 1 Exploring the Data '
    print '-------------------------------------------------------------------'

    digits = load_digits()
    print digits.keys()

    data = digits['data']
    images = digits['images']
    target = digits['target']
    target_names = digits['target_names']

    print 'Size of the digit set {}'. format(digits.data.shape)
    #print np.dtype(data) # TypeError :data type not understood

    # get all images with 3
    img3 = images[target == 3 ]
    # show the first one
    img = img3[0]
    assert 2 == np.size(np.shape(img))

    f = plot.figure()
    plot.gray()
    plot.imshow(img, interpolation = 'nearest');
    plot.show()
    f.savefig('task1.png')

    print
    print '-------------------------------------------------------------------'
    print ' 2 Nearest Neighbor Classifier '
    print '-------------------------------------------------------------------'

    # Write a NN-Classifier that distinguishes the digit '3' from all other digits

    np.set_printoptions(precision=5)

    # 2.1 Split data into a training-/test set

    x_all = data
    y_all = target

    x_train, x_test, y_train, y_test = cross_validation.train_test_split(x_all,y_all,
                                                    test_size = 0.4, random_state = 0

    print
    print ' Distance function computation '
    print

    # 2.2 Distance function computation using loops

    dist1, time1 = dist_loop(np.array(x_train), np.array(x_test))

    print 'Distance(loops) between ''1'' and ''3'':'
#    print dist1
    print 'Spend time: {}'. format(time1)

    # 2.3 Distance function computation using vectorization

    dist2, time2 = dist_vec(np.array(x_train), np.array(x_test))

    print 'Distance(vec) between ''1'' and ''3'':'
#    print dist2
    print 'Spend time: {}'. format(time2)

    # Compare results from 2.2 and 2.4
    similar = np.allclose(dist1, dist2, rtol=1e-05, atol=1e-08)
    assert similar, 'Functions dist_loop and dist_vec do not provide similar results'
    print 'Functions dist_loop and dist_vec provide similar results'


    print
    print ' A NN-Classifier'
    print

    # 2.4 A NN-Classifier

    # Indices of images with '1','3' and '7' on them
    ind1 = (target==1)
```

```
ind3 = (target==3)
ind7 = (target==7)

# Save images with ’1’ and ’3’
x13 = data[ind1+ind3]
y13 = target[ind1+ind3]
# split sets into training and test sets
x13_train, x13_test, y13_train, y13_test = cross_validation.train_test_split(x13,
                                                test_size = 0.3, random_state = 0

y13_predict = kNN(x13_train,y13_train, x13_test, 1)
rate13 = correctClassRate(y13_predict, y13_test, [1,3], print_confMatrix = True)
print ’1-NN Classifier: 1 OR 3? Correct classification rate is {}’. format(rate13
print


# Save images with ’1’ and ’7’
x17 = data[ind1+ind7]
y17 = target[ind1+ind7]

# split sets into training and test sets
x17_train, x17_test, y17_train, y17_test = cross_validation.train_test_split(x17,
                                                test_size = 0.4, random_state = 0
y17_predict = kNN(x17_train,y17_train, x17_test, 1)
rate17 = correctClassRate(y17_predict, y17_test, [1,7], print_confMatrix = True)
print ’1-NN Classifier: 1 OR 7? Correct classification rate is {}’. format(rate17

# 2.5 Try k=3,5,9,17

K = [3,5,9,17,33]
for k in K:
    y17_predict = kNN(x17_train,y17_train, x17_test, k)
    rate17 = correctClassRate(y17_predict, y17_test, [1, 7], print_confMatrix = F
    print ’{}-NN Classifier: 1 OR 7? Correct classification rate is {}’. format(k
# end for loop


## 3 Cross-validation of nearest neighbor

print
print ’------------------------------------------------------------------’
print ’ 3 Cross-validation of nearest neighbor  ’
print ’------------------------------------------------------------------’

nData,d = x_all.shape

print ’Size of complete data set: {} images’. format(nData)

# split data in n parts of equal size
N = [2,5,10]
```

```
    for n in N:

        # get indices of the images in each group after splitting
        indx_split = split_data_n_equal_parts(x_all,y_all, n)

        # for each of n groups
        ccr = np.zeros(n, dtype = np.float16)   # vector of correct classification ra
        for i in range(0,n):
            curr_group = indx_split[i,:]
            remaining = [j for j in range(0,nData) if j not in curr_group]
            # use current group as test set
            x_test = x_all[curr_group]
            y_test = y_all[curr_group]
            # use remaining data as training set
            x_training = x_all[remaining]
            y_training = y_all[remaining]
            # compute correct classification rate
            y_predict = kNN(x_training,y_training, x_test, 1)
            ccr[i] = correctClassRate(y_predict, y_test, range(0,10), print_confMatri
        # end for each of n groups
        mean_ccr = np.mean(ccr)
        var_ccr = np.var(ccr)

        #print '\t Correct classification Rates: {}'. format (ccr)
        print '\t Mean classification rate = {}, variance = {}'. format(mean_ccr, var
    # end for n in N

    return 0

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