Machine Learning Worksheet 3

Thomas Blocher – MatrNr. 03624034
Raphael Dümig – MatrNr. 03623199

November 1, 2015

Problem 1

see PDF-printout of jupyter notebook *kNN_implementationHW*

Problem 2

see PDF-printout of jupyter notebook *decision-tree*

Problem 3

results from the decision tree implementation of problem 2:

$$\hat{z}_a = 1$$
$$\hat{z}_b = 2$$

$$p(c = \hat{z}_a | x_a, T) = 1$$
$$p(c = \hat{z}_b | x_b, T) = \frac{2}{3}$$

Problem 4

results from the 3NN implementation of problem 2 without Standardization:

$$\hat{z}_a = 0$$
$$\hat{z}_b = 2$$

results from the 3NN implementation of problem 2 with Standardization:

$$\hat{z}_a = 1$$
$$\hat{z}_b = 0$$

Problem 5

results from the 3NN implementation of problem 2 without Standardization regression:

$$\hat{z}_a = 1.00000$$
$$\hat{z}_b = 0.50904$$

Problem 6

LOOCV on the Dataset showed that the quality of 3-NN is not very good. Using standardization improves the quality a little bit but it's still below 30%. (see Octave results.) The decision trees perform better on the dataset.

Code:

kNN_implementationHW

October 31, 2015

1 Implementation exercise: KNN

Thomas Blocher (MatrNr. 03624034) Raphael Dümig (MatrNr. 03623199)

```
In [2]: import random
        import numpy as np
        import operator
        from sklearn import datasets
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
/usr/lib/python3.5/site-packages/sklearn/utils/fixes.py:64: DeprecationWarning: inspect.getargspec() is
  if 'order' in inspect.getargspec(np.copy)[0]:
```

1.1 Load dataset

The iris data set (https://en.wikipedia.org/wiki/Iris_flower_data_set) it loaded by the function loadDataset.

Arguments:

- *split*: int Split rate between test and training set e.g. 0.67 corresponds to 1/3 test and 2/3 validation

Returns:

- *X*: list(array of length 4); Trainig data
- *Z*: list(int); Training labels
- *XT*: list(array of length 4); Test data
- *ZT*: list(int); Test labels

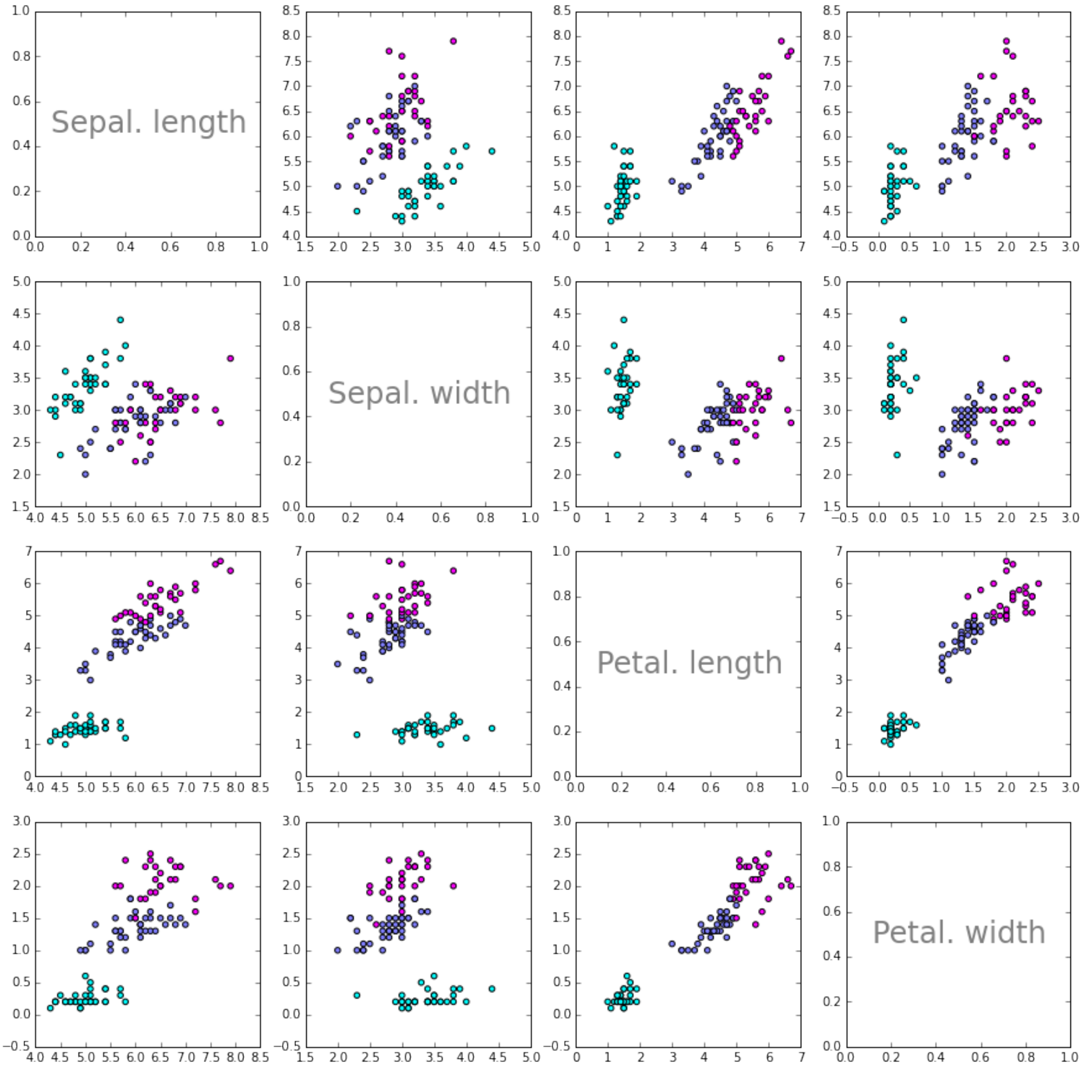
```
In [3]: def loadDataset(split, X=[] , XT=[], Z = [], ZT = []):
            dataset = datasets.load_iris()
            c = list(zip(dataset['data'], dataset['target']))
            random.seed(224)
            random.shuffle(c)
            x, t = zip(*c)
            sp = int(split*len(c))
            X = x[:sp]
            XT = x[sp:]
            Z = t[:sp]
            ZT = t[sp:]
            return X, XT, Z, ZT
```

```
In [4]: # prepare data
        split = 0.67
        X, XT, Z, ZT = loadDataset(split)
```

1.2 Plot dataset

Since X is dimentionality 4, 16 scatterplots (4x4) are plotted showing the dependencies of each two features.

```
In [5]: Xa = np.asarray(X)
        f, axes = plt.subplots(4, 4,figsize=(15, 15))
        for i in range(4):
            for j in range(4):
                if j == 0 and i == 0:
                    axes[i,j].text(0.5, 0.5, 'Sepal. length', ha='center', va='center', size=24, alpha=
                elif j == 1 and i == 1:
                    axes[i,j].text(0.5, 0.5, 'Sepal. width', ha='center', va='center', size=24, alpha=.
                elif j == 2 and i == 2:
                    axes[i,j].text(0.5, 0.5, 'Petal. length', ha='center', va='center', size=24, alpha=
                elif j == 3 and i == 3:
                    axes[i,j].text(0.5, 0.5, 'Petal. width', ha='center', va='center', size=24, alpha=.
                else:
                    axes[i,j].scatter(Xa[:,j],Xa[:,i], c = Z, cmap=plt.cm.cool)
```



1.3 Exercise 1: Euclidean distance

Compute euclidean distance between two data points.

arguments: * *x1*: array of length 4; data point * *x2*: array of length 4; data point

returns: * *distance*:int; euclidean distance between *x1* and *x2*

```
In [7]: def euclideanDistance(x1, x2):
            d = x1-x2
            return np.sqrt(np.sum(d*d, axis=-1))
```

1.4 Exercise 2: get k nearest neighbours

For one data point xt compute all k nearest neighbours.

arguments: * *X*: list(array of length 4); Training data * *Z*: list(int); Training labels * *xt*: array of length 4; Test data point

returns: * neighbours: list of length *k* of tuples (X_neighbor, Z_neighbor, distance between neighbor and xt); **this is the list of k nearest neighbours to xt**

```
In [10]: def getNeighbors(X, Z, xt, k):
             dists = euclideanDistance(X, xt)

             return [(X[i], Z[i], dists[i]) for i in np.argsort(dists)[:k]]
```

1.5 Exercise 3: get neighbor response

For the previously computed k nearest neighbors compute the actual response. I.e. give back the class of the majority of nearest neighbors. What do you do with a tie?

arguments: * neighbors

returns * y: int; majority target

```
In [16]: from collections import defaultdict

         def getResponse(neighbors):
             l = {}
             for n in neighbors:
                 try:
                     l[n[1]] += 1
                 except KeyError:
                     l[n[1]] = 1

             n_max = max(l.values())
             max_labels = list(filter(lambda k: l[k] == n_max, l.keys()))

             return max_labels[0]
```

1.6 Exercise 4: Compute accuracy

arguments: * YT:list(int); predicted targets * ZT:list(int); actual targets

returns: * accuracy (percentage of correctly classified test data points)

```
In [12]: def getAccuracy(YT, ZT):
             return np.sum(np.asarray(YT) == np.asarray(ZT)) / len(YT)
```

```
In [13]: def predict(X, Z, XT, k):
             Y=[]
             for xt in XT:
                 neighbors = getNeighbors(X, Z, xt, k)
                 Y.append(getResponse(neighbors))
             return Y
```

1.7 Testing

Should output an accuracy of 0.9599999999999996%.

```
In [17]: # prepare data
         split = 0.67
         X, XT, Z, ZT = loadDataset(split)
         print('Train set: ' + repr(len(X)))
         print('Test set: ' + repr(len(XT)))
         # generate predictions
         k = 3
         YT = predict(X, Z, XT, k)
         accuracy = getAccuracy(YT, ZT)
         print('Accuracy: ' + repr(accuracy) + '%')
```

```
Train set: 100
Test set: 50
Accuracy: 0.9599999999999996%
```

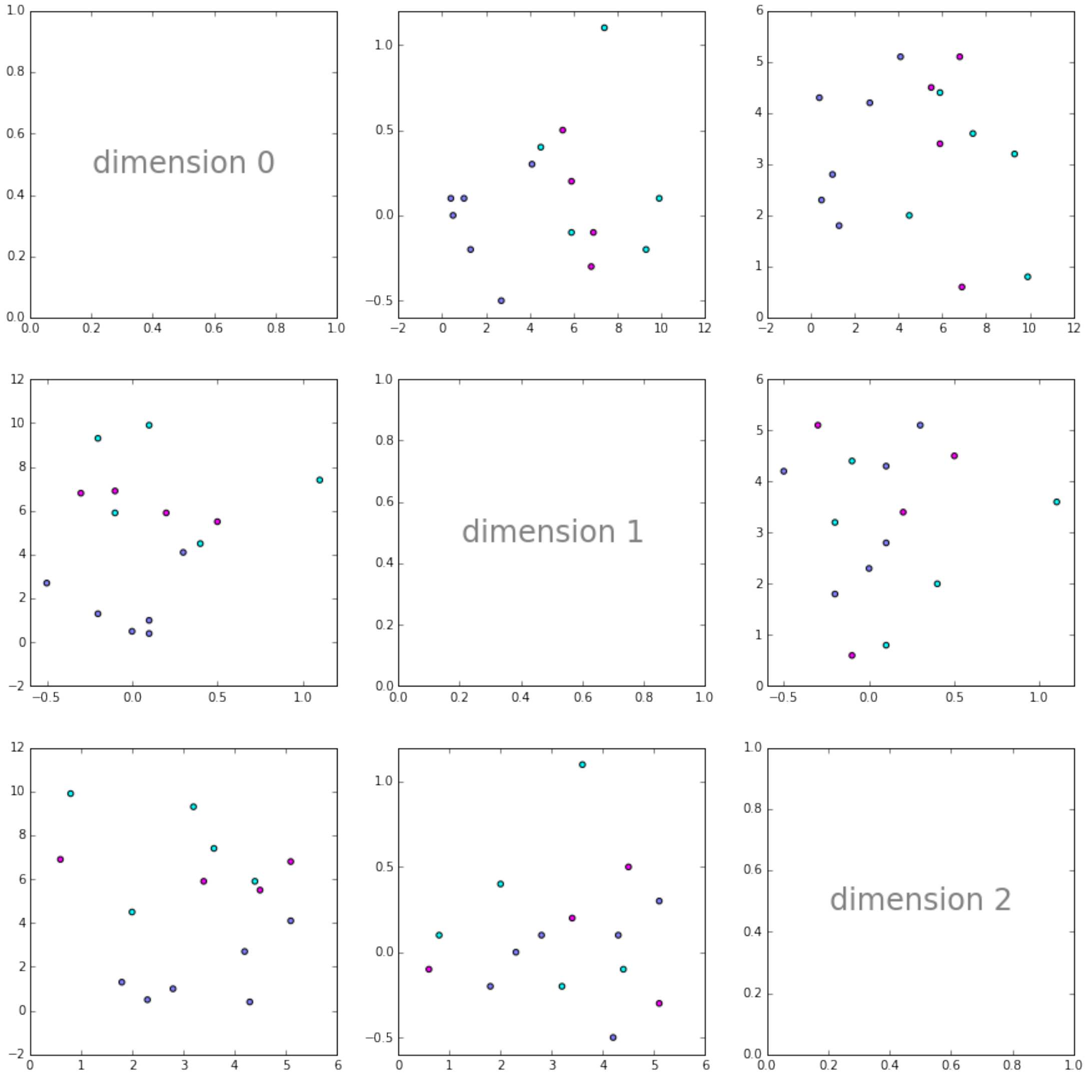
```
In [ ]:
```

decision_tree

October 31, 2015

```
In [3]: # Raphael Dümig (MatrNr. 03623199)
        # Thomas Blocher (MatrNr. 03624034)

        # load the data

        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.cm as cm
        import math as m
        %matplotlib inline

        data = np.genfromtxt('../homework03.csv', delimiter=',', skip_header=1)

        X = data[:,:3]
        Z = data[:,3]
```

```
In [10]: # plot the data

         fig, axes = plt.subplots(3,3, figsize=(15,15))

         for i in range(3):
             for j in range(3):
                 if i != j:
                     axes[i,j].scatter(X[:,i], X[:,j], c=Z, cmap=cm.cool)
                 else:
                     axes[i,j].text(0.5, 0.5, 'dimension %d' % i,
                                    ha='center', va='center', size=24, alpha=.5)
```



```
In [4]: # build a decision tree using the data

        import itertools

        def gini_index(labels, label_values):
            if labels.size == 0:
                return 0
            else:
                n_label = np.sum(labels[:, np.newaxis] == label_values, axis=0)
                return 1 - np.sum(n_label**2)/np.sum(n_label)**2


        class DecisionTreeNode:
            def __init__(self, X, Z, depth=np.infty):
                #print(’creating node with a dataset of size %d (%d levels remaining)’ % (X.shape[0], d
                self.X = X
                self.Z = Z
                self.left = None
                self.right = None
                self.split_pos = None
                self.n = Z.shape[0]
                self.depth = depth
                self.split()

            def gini(self):
                if self.right is None:
                    return gini_index(self.Z, list(set(self.Z)))
                else:
                    return (self.right.gini()*self.right.n + self.left.gini()*self.left.n) / self.n

            def get_cut_positions(self):
                s = np.sort( self.X, axis=0 )
                return (s[1:] + s[:-1]) / 2

            def get_labels(self):
                return list(set(self.Z))

            def is_pure(self):
                return len(self.get_labels()) == 1

            def get_label_counts(self):
                ls = self.get_labels()
                n_label = np.sum(self.Z[:, np.newaxis] == ls, axis=0)
                return { l:n for (l,n) in zip(ls,n_label) }

            def split(self):
                if self.is_pure():
                    print(’node is pure! refusing to split...’)
                    return
                if self.depth <= 0:
                    print(’reached maximum depth! refusing to split...’)
                    return

                label_values = self.get_labels()
                cuts = self.get_cut_positions()
                zz = cuts[:,:,np.newaxis] < self.X.T
                indices = np.array([
                        [ (  gini_index(self.Z[ zz[i,j,:]], label_values)*np.sum( zz[i,j,:])
                           + gini_index(self.Z[˜zz[i,j,:]], label_values)*np.sum(˜zz[i,j,:])
                          )
                          for j in range(zz.shape[1])
                        ]
                        for i in range(zz.shape[0])
                    ])
                min_index = np.argmin(indices)
                split_dim = min_index % indices.shape[1]
                split_dest = min_index // indices.shape[1]
                self.split_pos = (split_dim , cuts[split_dest, split_dim])
                mask = zz[split_dest,split_dim,:]
                # put all elements greater than the splitting boundary into the left node
                self.left = DecisionTreeNode(self.X[mask], self.Z[mask], self.depth-1)
                # all others in the right node
                self.right = DecisionTreeNode(self.X[~mask], self.Z[~mask], self.depth-1)
                return

            def subtree_to_str(self, d=0):
                res = self.node_to_str(d)
                if self.left is not None:
                    res += '\n'.join(['|' + l for l in itertools.chain(
                                self.left.subtree_to_str(d+1).split('\n'),
                                self.right.subtree_to_str(d+1).split('\n'))
                    ])

                return res

            def node_to_str(self, d=0):
                res = ''
                if self.split_pos is not None:
                    res = ('depth: %d -- X%s > %.2f' % (d, chr(ord('₀') + self.split_pos[0] + 1),
                                                        self.split_pos[1]))
                else:
                    res = ('depth: %d -- leaf' % d)
                res += ' (gini: %.2f)\n' % self.gini()
                nl = self.get_label_counts()
                for l in sorted(nl.keys()):
                    res += ('\t%s: %f%%\n' % (str(l), 100*nl[l]/self.n))
                return res


class DecisionTree:
    def __init__(self, X, Z, max_depth=np.infty):
        self.X = X
        self.Z = Z
        self.max_depth = max_depth
        self.root = DecisionTreeNode(X,Z,max_depth)

    def gini_index(self):
        return self.root.gini_index()

    def print_tree(self):
        print('DFS traversal of the tree:')
        print(self.root.subtree_to_str(0))
```

```
In [5]: T = DecisionTree(X,Z.astype(int),2)
        print(30 * '=' + '\n')
        T.print_tree()
```

```
node is pure! refusing to split...
reached maximum depth! refusing to split...
node is pure! refusing to split...
==============================

DFS traversal of the tree:
depth: 0 -- X₁ > 4.30 (gini: 0.18)
        0: 33.333333%
        1: 40.000000%
        2: 26.666667%
|depth: 1 -- X_1 > 7.15 (gini: 0.30)
|        0: 55.555556%
|        2: 44.444444%
||depth: 2 -- leaf (gini: 0.00)
||        0: 100.000000%
||
||depth: 2 -- leaf (gini: 0.44)
||        0: 33.333333%
||        2: 66.666667%
||
|depth: 1 -- leaf (gini: 0.00)
|        1: 100.000000%
|
```

Source code: 3NN

```
%% Read in Dataset
data = csvread('homework03.csv');
%remove first line (Remove labels)
data = data(2:end,:);
coords = data(:,1:3)
labels = data(:,4)

xa = [4.1, -0.1, 2.2]
xb = [6.1, 0.4, 1.3]


%% Since the axis are not equaly derivated we have to do some standardization

%% Doing the Standardization
  xaStd(1) = (xa(1) - mean(coords(:,1)))./std(coords(:,1));
  xaStd(2) = (xa(2) - mean(coords(:,2)))./std(coords(:,2));
  xaStd(3) = (xa(3) - mean(coords(:,3)))./std(coords(:,3));

  xbStd(1) = (xb(1) - mean(coords(:,1)))./std(coords(:,1));
  xbStd(2) = (xb(2) - mean(coords(:,2)))./std(coords(:,2));
  xbStd(3) = (xb(3) - mean(coords(:,3)))./std(coords(:,3));

  coordsStd(:,1) = (coords(:,1) - mean(coords(:,1)))./std(coords(:,1));
  coordsStd(:,2) = (coords(:,2) - mean(coords(:,2)))./std(coords(:,2));
  coordsStd(:,3) = (coords(:,3) - mean(coords(:,3)))./std(coords(:,3));
%%LOOCV Tests
%Without Standardization
for valid = 1:size(labels,1)
  dist =[];
  x = data(valid,1:3);
  for i = 1:size(labels,1)
    %Calculating the dist to each point
    dist = [norm(coords(i)-x),dist];
  end
  [~,i] = sort(dist);
  %Since Training data are unique we get the 3 NN to Validation Point by taking the 3 Closest point with dist ~= 0 (i(1)=valid)
  % the second part of the expression returns true iff there is a tie
  out(valid) = (mode(labels(i(2:4))) == labels(valid)) || size(unique(labels(i(2:4))),2) == 3;
```

```
 end

 %With Standardization
 for valid = 1:size(labels,1)
  dist =[];
  x = coords(valid,1:3);
  for i = 1:size(labels,1)
    dist = [norm(coordsStd(i,:)-x),dist];
  end
  [~,i] = sort(dist);
  outStd(valid) = (mode(labels(i(2:4))) == labels(valid)) || size(unique(labels(i(2:4))),2) == 3;
 end

 %%%LOOCV Results

 quality = mean(out)

 qualityStd = mean(outStd)

%% Labeling the Points
%Classification without Standardization
for i = 1:size(labels,1)
  dist(i) = norm(coords(i,:)-xa);
end
[~,i] = sort(dist);
va = mode(data(i(1:3),4))

for i = 1:size(labels,1)
  dist(i) = norm(coords(i,:)-xb);
end

[~,index] = sort(dist);
vb = mode(data(index(1:3),4))



%Classification with Standardization
for i = 1:size(labels,1)
  dist(i) = norm(coordsStd(i,:)-xaStd);
end

[~,i] = sort(dist);
vaStandardized = mode(labels(i(1:3)))

for i = 1:size(labels,1)
  dist(i) = norm(coordsStd(i,:)-xbStd);
end

```

```
[~,i] = sort(dist);
vbStandardized = mode(labels(i(1:3)))


%% Regession (just using standardization)

for i = 1:size(labels,1)
  dist(i) = norm(coordsStd(i,:)-xaStd);
end

[~,i] = sort(dist);
vareg = (1/sum(1./dist(i(1:3)))) * sum ( 1./(dist(i(1:3))) * labels(i(1:3)))

for i = 1:size(labels,1)
  dist(i) = norm(coordsStd(i,1:3)-xbStd);
end

[~,i] = sort(dist);
vbreg = (1/sum(1./dist(i(1:3)))) * sum ( 1./(dist(i(1:3))) * labels(i(1:3)))
```

Execution results

```
octave>hw2_4_5
coords =

   5.50000   0.50000   4.50000
   7.40000   1.10000   3.60000
   5.90000   0.20000   3.40000
   9.90000   0.10000   0.80000
   6.90000  -0.10000   0.60000
   6.80000  -0.30000   5.10000
   4.10000   0.30000   5.10000
   1.30000  -0.20000   1.80000
   4.50000   0.40000   2.00000
   0.50000   0.00000   2.30000
   5.90000  -0.10000   4.40000
   9.30000  -0.20000   3.20000
   1.00000   0.10000   2.80000
   0.40000   0.10000   4.30000
   2.70000  -0.50000   4.20000

labels =

   2
   0
   2
   0
   2
   2
   1
   1
   0
   1
   0
   0
   1
   1
   1

xa =

   4.10000  -0.10000   2.20000

xb =

   6.10000   0.40000   1.30000

quality =  0.13333
qualityStd =  0.26667
va = 0
vb =  2
vaStandardized =  1
vbStandardized = 0
vareg =  1.00000
vbreg =  0.50904
```