In [1]:

```
import numpy as np
import pandas as pd
```

In [2]:

```
data=pd.read_csv('01_homework_dataset.csv')
y_train=data[' z'].values
X1_train=data['x1'].values
X2_train=data[' x2'].values
X3_train=data[' x3'].values
data.head()
```

Out[2]:

	x1	x2	x3	z
0	5.5	0.5	4.5	2
1	7.4	1.1	3.6	0
2	5.9	0.2	3.4	2
3	9.9	0.1	0.8	0
4	6.9	-0.1	0.6	2

In [3]:

```
def calculate_gini(classes):
    if len(classes)==0:
        return 0 # best gini value as homogenous nothing
    num_classes=len(np.unique(classes))

    class_votes = np.zeros(3)
    for label in classes:
        class_votes[label]+=1

    return 1-np.sum(np.square(class_votes/len(classes)))
```

Decision Tree: Root Level

Find the best root nodes

In [4]:

```
# lets split by x1 first
initGini=calculate_gini(y_train)
SplitNodeX1={}
for x in X1_train:
    LGini=calculate_gini(data[data['x1']<x][' z'].values)
    RGini=calculate_gini(data[data['x1']>=x][' z'].values)
    GiniChange=initGini-(len(data[data['x1']<x])/len(data))*LGini-(len(data[data['x1']>=x])/len(data))*RGini
    SplitNodeX1[x]=GiniChange
print('Best split for root for x1=',max(SplitNodeX1, key=SplitNodeX1.get), ' with gini imporvement=',SplitNodeX1[max(SplitNodeX1, key=Spl
itNodeX1.get)])
```

```
Best split for root for x1= 4.5  with gini imporvement= 0.3614814814814815
```

In [5]:

```
# lets split by x2
initGini=calculate_gini(y_train)
SplitNodeX2={}
for x in X2_train:
    LGini=calculate_gini(data[data[' x2']<x][' z'].values)
    RGini=calculate_gini(data[data[' x2']>=x][' z'].values)
    GiniChange=initGini-(len(data[data[' x2']<x])/len(data))*LGini-(len(data[data[' x2']>=x])/len(data))*RGini
    SplitNodeX2[x]=GiniChange
print('Best split for root for x2=',max(SplitNodeX2, key=SplitNodeX2.get), ' with gini improvement=',SplitNodeX2[max(SplitNodeX2, key=Spl
itNodeX2.get)])
```

```
Best split for root for x2= 0.4  with gini improvement= 0.0688888888888889
```

In [6]:

```
# lets split by x3
initGini=calculate_gini(y_train)
SplitNodeX3={}
for x in X3_train:
    LGini=calculate_gini(data[data[' x3']<x][' z'].values)
    RGini=calculate_gini(data[data[' x3']>=x][' z'].values)
    GiniChange=initGini-(len(data[data[' x3']<x])/len(data))*LGini-(len(data[data[' x3']>=x])/len(data))*RGini
    SplitNodeX3[x]=GiniChange
print('Best split for root for x3=',max(SplitNodeX3, key=SplitNodeX3.get), ' with gini improvement=',SplitNodeX3[max(SplitNodeX3, key=Spl
itNodeX3.get)])
```

```
Best split for root for x3= 4.5  with gini improvement= 0.0688888888888889
```

From above 3 cells we see that the root note node should x1 with value of 4.5

In [7]:

```
# for the tree at the root
print('Class Distrubution before split: ', y_train)
print('Gini Value before split: ',calculate_gini(y_train))
print('')
print('The best node is at x1=4.5')
print('')
print('Class Distrubution on Left after split: ', data[data['x1']<4.5][' z'].values)
print('Gini Value on the left after split: ',calculate_gini(data[data['x1']<4.5][' z'].values))
print('')
print('Class Distrubution on Right after split: ', data[data['x1']>=4.5][' z'].values)
print('Gini Value on the right after split: ',calculate_gini(data[data['x1']>=4.5][' z'].values))
print('')
```

```
Class Distrubution before split:  [2 0 2 0 2 2 1 1 0 1 0 0 1 1 1]
Gini Value before split:  0.6577777777777778

The best node is at x1=4.5

Class Distrubution on Left after split:  [1 1 1 1 1 1]
Gini Value on the left after split:  0.0

Class Distrubution on Right after split:  [2 0 2 0 2 2 0 0 0]
Gini Value on the right after split:  0.49382716049382713
```

Decision Tree: First Left Child

Find the best nodes for the left child of the root.

This step is not required as we got a perfect gini of 0.0 in the previous split

Decision Tree: First Right Child

Find the best nodes for the split of the groups in the right child of the root.

In [8]:

```
# Create a dataframe of remaining data points in the right child
firstRightData=data[data['x1']>=4.5]

y_train=firstRightData[' z'].values
X1_train=firstRightData['x1'].values
X2_train=firstRightData[' x2'].values
X3_train=firstRightData[' x3'].values
```

In [9]:

```
firstRightData
```

Out[9]:

	x1	x2	x3	z
0	5.5	0.5	4.5	2
1	7.4	1.1	3.6	0
2	5.9	0.2	3.4	2
3	9.9	0.1	0.8	0
4	6.9	-0.1	0.6	2
5	6.8	-0.3	5.1	2
8	4.5	0.4	2.0	0
10	5.9	-0.1	4.4	0
11	9.3	-0.2	3.2	0

In [10]:

```
# lets split by x1 first
initGini=calculate_gini(y_train)
SplitNodeX1={}
for x in X1_train:
    LGini=calculate_gini(firstRightData[firstRightData['x1']<x][' z'].values)
    RGini=calculate_gini(firstRightData[firstRightData['x1']>=x][' z'].values)
    GiniChange=initGini-(len(firstRightData[firstRightData['x1']<x])/len(firstRightData))*LGini-(len(firstRightData[firstRightData['x1']>
=x])/len(firstRightData))*RGini
    SplitNodeX1[x]=GiniChange
print('Best split for root for x1=',max(SplitNodeX1, key=SplitNodeX1.get), ' with gini imporvement=',SplitNodeX1[max(SplitNodeX1, key=Spl
itNodeX1.get)])
```

```
Best split for root for x1= 7.4  with gini imporvement= 0.19753086419753085
```

In [11]:

```
# lets split by x2
initGini=calculate_gini(y_train)
SplitNodeX2={}
for x in X2_train:
    LGini=calculate_gini(firstRightData[firstRightData[' x2']<x][' z'].values)
    RGini=calculate_gini(firstRightData[firstRightData[' x2']>=x][' z'].values)
    GiniChange=initGini-(len(firstRightData[firstRightData[' x2']<x])/len(firstRightData))*LGini-(len(firstRightData[firstRightData[' x2'
]>=x])/len(firstRightData))*RGini
    SplitNodeX2[x]=GiniChange
print('Best split for root for x2=',max(SplitNodeX2, key=SplitNodeX2.get), ' with gini improvement=',SplitNodeX2[max(SplitNodeX2, key=Spl
itNodeX2.get)])
```

```
Best split for root for x2= -0.2  with gini improvement= 0.0771604938271605
```

In [12]:

```
# lets split by x3
initGini=calculate_gini(y_train)
SplitNodeX3={}
for x in X3_train:
    LGini=calculate_gini(firstRightData[firstRightData[' x3']<x][' z'].values)
    RGini=calculate_gini(firstRightData[firstRightData[' x3']>=x][' z'].values)
    GiniChange=initGini-(len(firstRightData[firstRightData[' x3']<x])/len(firstRightData))*LGini-(len(firstRightData[firstRightData[' x3'
]>=x])/len(firstRightData))*RGini
    SplitNodeX3[x]=GiniChange
print('Best split for root for x3=',max(SplitNodeX3, key=SplitNodeX3.get), ' with gini improvement=',SplitNodeX3[max(SplitNodeX3, key=Spl
itNodeX3.get)])
```

```
Best split for root for x3= 4.5  with gini improvement= 0.17636684303350963
```

From above 3 cells we see that the root note node should x1 with value of 7.4

In [13]:

```
# for the tree at the root
print('Class Distrubution before split: ', y_train)
print('Gini Value before split: ',calculate_gini(y_train))
print('')
print('The best node is at x1=7.4')
print('')
print('Class Distrubution on Left after split: ', firstRightData[firstRightData['x1']<7.4][' z'].values)
print('Gini Value on the left after split: ',calculate_gini(firstRightData[firstRightData['x1']<7.4][' z'].values))
print('')
print('Class Distrubution on Right after split: ', firstRightData[firstRightData['x1']>=7.4][' z'].values)
print('Gini Value on the right after split: ',calculate_gini(firstRightData[firstRightData['x1']>=7.4][' z'].values))
print('')
```

```
Class Distrubution before split:  [2 0 2 0 2 2 0 0 0]
Gini Value before split:  0.49382716049382713

The best node is at x1=7.4

Class Distrubution on Left after split:  [2 2 2 2 0 0]
Gini Value on the left after split:  0.4444444444444444

Class Distrubution on Right after split:  [0 0 0]
Gini Value on the right after split:  0.0
```

Final Tree

In [14]:

```
def decision_tree(train_data):
    if train_data[0]<4.5:
        return 1
    elif train_data[0]<7.4:
        return 0
    else:
        return 2
```

In [15]:

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

In [16]:

```
ya_pred=decision_tree(xa)
ya_pred
```

Out[16]:

```
1
```

In [17]:

```
yb_pred=decision_tree(xb)
yb_pred
```

Out[17]:

```
0
```

Finding probabilities

probability is given by the formula

$p(y = c|R) =$ n_c,R/(\sum n_C_i,R)

so $p(c = 1|T) = 6/6 = 1$

$p(c = 0|T) = 5/9$

Programming assignment 1: k-Nearest Neighbors classification

In [1]:

```
import numpy as np
from sklearn import datasets, model_selection
import matplotlib.pyplot as plt
%matplotlib inline
```

Introduction

For those of you new to Python, there are lots of tutorials online, just pick whichever you like best :)

If you never worked with Numpy or Jupyter before, you can check out these guides

- https://docs.scipy.org/doc/numpy-dev/user/quickstart.html (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html)
- http://jupyter.readthedocs.io/en/latest/ (http://jupyter.readthedocs.io/en/latest/)

Your task

In this notebook code to perform k-NN classification is provided. However, some functions are incomplete. Your task is to fill in the missing code and run the entire notebook.

In the beginning of every function there is docstring, which specifies the format of input and output. Write your code in a way that adheres to it. You may only use plain python and `numpy` functions (i.e. no scikit-learn classifiers).

Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is

1. Run all the cells of the notebook.
2. Download the notebook in HTML (click File > Download as > .html)
3. Convert the HTML to PDF using e.g. https://www.sejda.com/html-to-pdf (https://www.sejda.com/html-to-pdf) or `wkhtmltopdf` for Linux (tutorial (https://www.cyberciti.biz/open-source/html-to-pdf-freeware-linux-osx-windows-software/))
4. Concatenate your solutions for other tasks with the output of Step 3. On a Linux machine you can simply use `pdfunite`, there are similar tools for other platforms too. You can only upload a single PDF file to Moodle.

This way is preferred to using `nbconvert`, since `nbconvert` clips lines that exceed page width and makes your code harder to grade.

Load dataset

The iris data set (https://en.wikipedia.org/wiki/Iris_flower_data_set (https://en.wikipedia.org/wiki/Iris_flower_data_set)) is loaded and split into train and test parts by the function `load_dataset`.

In [2]:

```
def load_dataset(split):
    """Load and split the dataset into training and test parts.

    Parameters
    ----------
    split : float in range (0, 1)
        Fraction of the data used for training.

    Returns
    -------
    X_train : array, shape (N_train, 4)
        Training features.
    y_train : array, shape (N_train)
        Training labels.
    X_test : array, shape (N_test, 4)
        Test features.
    y_test : array, shape (N_test)
        Test labels.
    """
    dataset = datasets.load_iris()
    X, y = dataset['data'], dataset['target']
    X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, random_state=123, test_size=(1 - split))
    return X_train, X_test, y_train, y_test
```

In [3]:

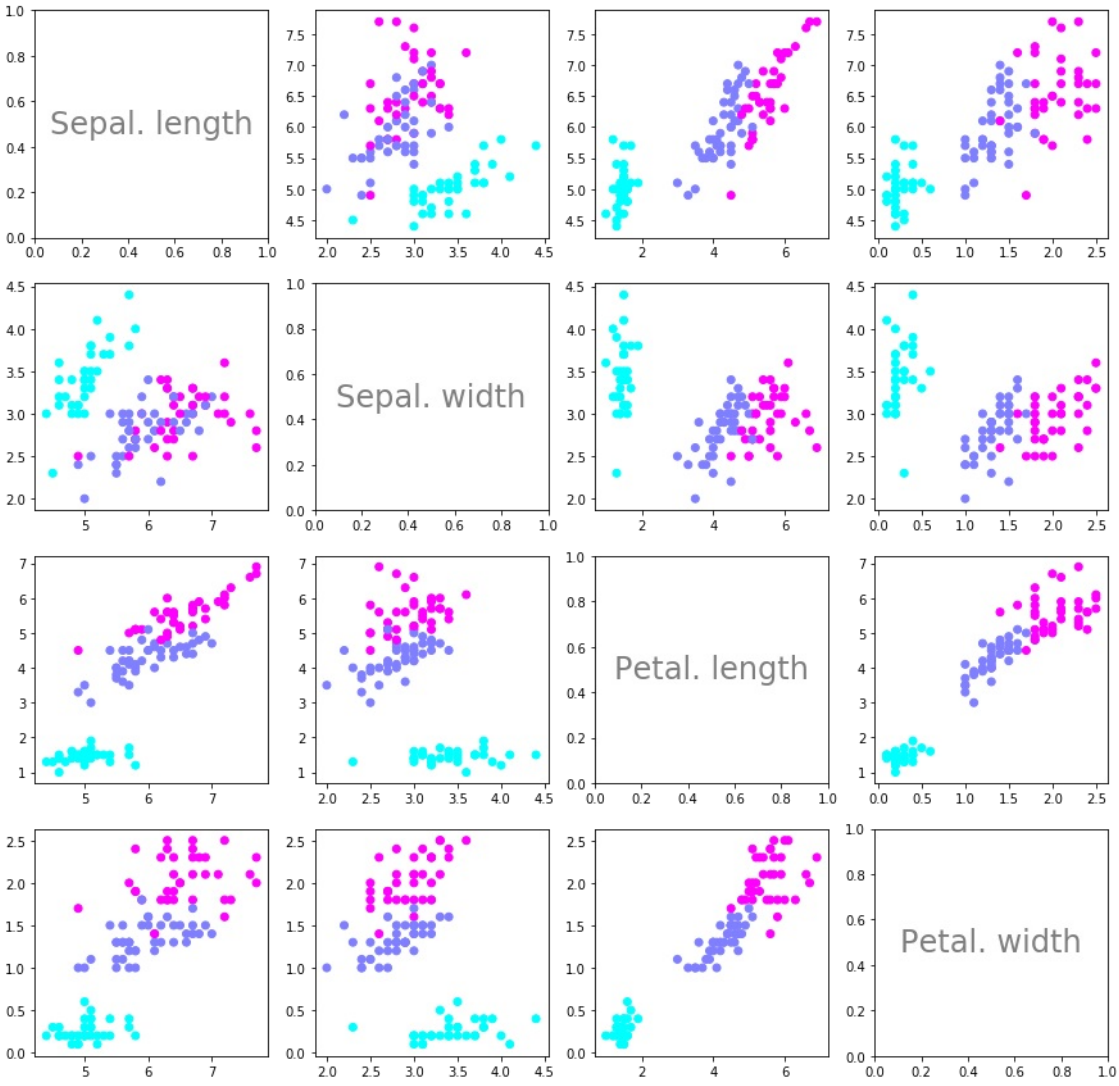
```
# prepare data
split = 0.75
X_train, X_test, y_train, y_test = load_dataset(split)
```

Plot dataset

Since the data has 4 features, 16 scatterplots (4x4) are plotted showing the dependencies between each pair of features.

In [4]:

```
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=.5)
        elif j == 1 and i == 1:
            axes[i,j].text(0.5, 0.5, 'Sepal. width', ha='center', va='center', size=24, alpha=.5)
        elif j == 2 and i == 2:
            axes[i,j].text(0.5, 0.5, 'Petal. length', ha='center', va='center', size=24, alpha=.5)
        elif j == 3 and i == 3:
            axes[i,j].text(0.5, 0.5, 'Petal. width', ha='center', va='center', size=24, alpha=.5)
        else:
            axes[i,j].scatter(X_train[:,j],X_train[:,i], c=y_train, cmap=plt.cm.cool)
```



Task 1: Euclidean distance

Compute Euclidean distance between two data points.

In [5]:

```
def euclidean_distance(x1, x2):
    """Compute Euclidean distance between two data points.

    Parameters
    ----------
    x1 : array, shape (4)
        First data point.
    x2 : array, shape (4)
        Second data point.

    Returns
    -------
    distance : float
        Euclidean distance between x1 and x2.
    """

    return np.sqrt(np.sum((x1-x2)**2))
```

Task 2: get k nearest neighbors' labels

Get the labels of the *k* nearest neighbors of the datapoint *x_new*.

In [6]:

```
def get_neighbors_labels(X_train, y_train, x_new, k):
    """Get the labels of the k nearest neighbors of the datapoint x_new.

    Parameters
    ----------
    X_train : array, shape (N_train, 4)
        Training features.
    y_train : array, shape (N_train)
        Training labels.
    x_new : array, shape (4)
        Data point for which the neighbors have to be found.
    k : int
        Number of neighbors to return.

    Returns
    -------
    neighbors_labels : array, shape (k)
        Array containing the labels of the k nearest neighbors.
    """
    distance=np.float32(np.zeros_like(y_train))
    for i,x in enumerate(X_train):
        distance[i] = euclidean_distance(x,x_new)
    #print(distance)

    idx = (distance).argsort()[:k]
    #print(distance[idx])
    return y_train[idx]
```

Task 3: get the majority label

For the previously computed labels of the *k* nearest neighbors, compute the actual response. I.e. give back the class of the majority of nearest neighbors. In case of a tie, choose the "lowest" label (i.e. the order of tie resolutions is 0 > 1 > 2).

In [7]:

```
def get_response(neighbors_labels, num_classes=3):
    """Predict label given the set of neighbors.

    Parameters
    ----------
    neighbors_labels : array, shape (k)
        Array containing the labels of the k nearest neighbors.
    num_classes : int
        Number of classes in the dataset.

    Returns
    -------
    y : int
        Majority class among the neighbors.
    """
    # TODO
    class_votes = np.zeros(num_classes)

    for label in neighbors_labels:
        class_votes[label]+=1

    #print(class_votes)
    return np.argmax(class_votes)
```

Task 4: compute accuracy

Compute the accuracy of the generated predictions.

In [8]:

```
def compute_accuracy(y_pred, y_test):
    """Compute accuracy of prediction.

    Parameters
    ----------
    y_pred : array, shape (N_test)
        Predicted labels.
    y_test : array, shape (N_test)
        True labels.
    """
    # TODO
    return np.sum(y_pred==y_test)/np.size(y_pred)
```

In [9]:

```
# This function is given, nothing to do here.
def predict(X_train, y_train, X_test, k):
    """Generate predictions for all points in the test set.

    Parameters
    ----------
    X_train : array, shape (N_train, 4)
        Training features.
    y_train : array, shape (N_train)
        Training labels.
    X_test : array, shape (N_test, 4)
        Test features.
    k : int
        Number of neighbors to consider.

    Returns
    -------
    y_pred : array, shape (N_test)
        Predictions for the test data.
    """
    y_pred = []
    for x_new in X_test:
        neighbors = get_neighbors_labels(X_train, y_train, x_new, k)
        #print(neighbors)
        y_pred.append(get_response(neighbors))
    return y_pred
```

Testing

Should output an accuracy of 0.9473684210526315.

In [10]:

```
# prepare data
split = 0.75
X_train, X_test, y_train, y_test = load_dataset(split)
print('Training set: {0} samples'.format(X_train.shape[0]))
print('Test set: {0} samples'.format(X_test.shape[0]))

# generate predictions
k = 3
y_pred = predict(X_train, y_train, X_test, k)
accuracy = compute_accuracy(y_pred, y_test)
print('Accuracy = {0}'.format(accuracy))
```

```
Training set: 112 samples
Test set: 38 samples
Accuracy = 0.9473684210526315
```

In [1]:

```
import numpy as np
import pandas as pd
```

In [2]:

```
def distance_measure(x1, x2, distType='euclidean'):
    if distType=='euclidean':
        return np.sqrt(np.sum((x1-x2)**2))

    #TODO Implement other distances
```

In [3]:

```
def knn_classifier(X_train, y_train, x_new, k):
    distance=np.float32(np.zeros_like(y_train))
    for i,x in enumerate(X_train):
        distance[i] = distance_measure(x,x_new)

    idx = (distance).argsort()[:k]
    neighbors_label=y_train[idx]

    class_votes = np.zeros(np.size(np.unique(y_train)))

    for label in neighbors_label:
        class_votes[label]+=1

    return np.argmax(class_votes)
```

In [4]:

```
def knn_regression(X_train, y_train, x_new, k):
    distance=np.float32(np.zeros_like(y_train))
    for i,x in enumerate(X_train):
        distance[i] = distance_measure(x,x_new)

    idx = (distance).argsort()[:k]
    neighbors_label=y_train[idx]

    summation=0.0
    for loc in idx:
        summation+=y_train[loc]/distance[loc]

    return summation/np.sum(distance[idx])
```

In [5]:

```
data=pd.read_csv('01_homework_dataset.csv')
y_train=data[' z'].values
X_train=data[['x1', ' x2', ' x3']].values
data
```

Out[5]:

	x1	x2	x3	z
0	5.5	0.5	4.5	2
1	7.4	1.1	3.6	0
2	5.9	0.2	3.4	2
3	9.9	0.1	0.8	0
4	6.9	-0.1	0.6	2
5	6.8	-0.3	5.1	2
6	4.1	0.3	5.1	1
7	1.3	-0.2	1.8	1
8	4.5	0.4	2.0	0
9	0.5	0.0	2.3	1
10	5.9	-0.1	4.4	0
11	9.3	-0.2	3.2	0
12	1.0	0.1	2.8	1
13	0.4	0.1	4.3	1
14	2.7	-0.5	4.2	1

In [6]:

```
k=3
xa=(4.1, -0.1, 2.2)
xb = (6.1, 0.4, 1.3)
```

In [7]:

```
ya_pred=knn_classifier(X_train, y_train,xa,k)
ya_pred
```

Out[7]:

```
0
```

In [8]:

```
yb_pred=knn_classifier(X_train, y_train,xb,k)
yb_pred
```

Out[8]:

```
2
```

In [9]:

```
ya_pred=knn_regression(X_train, y_train,xa,k)
ya_pred
```

Out[9]:

```
0.2477075097571343
```

In [10]:

```
yb_pred=knn_regression(X_train, y_train,xb,k)
yb_pred
```

Out[10]:

```
0.5250610813452723
```

The problem with ***X*** is that the values are in different scales, some have huge ranges, some have non negative values etc. and so are no in the same scale.

The problem can be solved by simply using a z-score scaling on each of the column fo the data ***X***.

Decision Trees do not face this problem as the decsion nodes are based on values and have no connection accross columns, like in KNN for measuring the distance.