03_Nearest Neighbours

March 25, 2022

[1]:
```
from IPython.display import Image
```

1 Nearest Neighbours

2 Outline

1. Section **??**
2. Section **??**

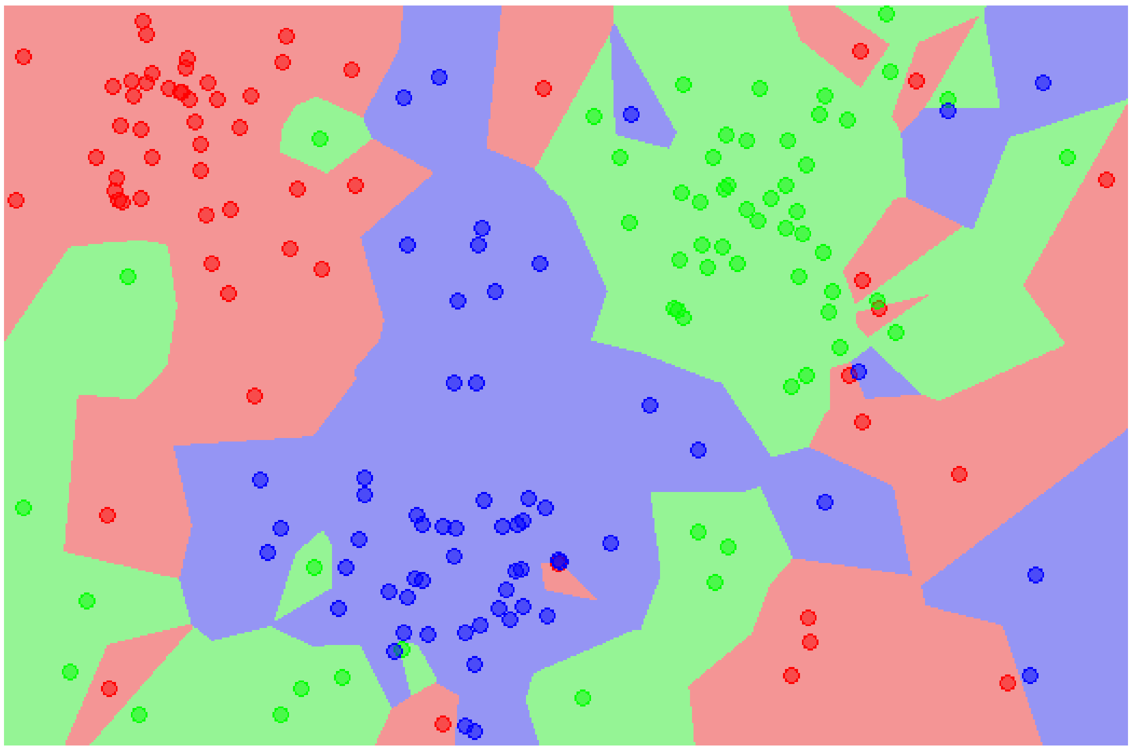
3 Nearest neighbours: algorithm

kNN algorithm is a **supervised** machine learning algorithm that can be used both for classification and regression. It's an **instance-based** algorithm. So instead of estimating a model, it stores all training examples in memory and makes predictions using a **similarity measure**.

an interisting feature of kNN is that no assumption is made about a model. The only requirement is that a *distance* or *dissimilarity* measure can be calculated between two instances. Hence it is a non-parametric and non linear model.

[2]:
```
Image("knn1.png")
```

[2]:



Given an input, kNN retrieves the k closest (most similar) instances from memory (the neighbours). Similarity is defined in terms of distance, that is, the training examples with the smallest (e. g. euclidean, Hamming, Manhattan) distance to the input example are considered to be most similar.

The prediction is done as follows:

Classification:
a) unweighted: output the most common classification among the k-nearest neighbors: True, False, False is False
b) weighted: sum up the weights of the k-nearest neighbors for each classification value, output classification with highest weight

Regression:
a) unweighted: output the average of the values of the k-nearest neighbors
b) weighted: for all regression values, sum up regression value∗weight and divide the result trough the sum of all weights

kNN depends on the type of similarity used (as in clustering applications). It can be slow if used on very large datasets.
Also, distance measures in high dimensionality spaces can be problematic.

3.1 Example

Classify cars as ``fast'' or ``slow'' on the basis of a similarity measure. We are given features and a label (fast or slow) for these cars:

car	horsepower	racing stripes	is_fast
Honda Accord	180	False	False
Yugo	500	True	True
Delorean DMC-12	200	True	True

and we are asked if this car:

Chevrolet Camaro,400,True,Unknown

is fast

In order to make a prediction with kNN, we first find the most similar known car. Comparing the two features, horsepower and racing_stripes, the most similar car is the Yugo. Since it is fast, we would predict that the Camaro is also fast. This is an example of 1-nearest neighbors.

If we performed a 2-nearest neighbors, we would end up with 2 True values (for the Delorean and the Yugo), which would average out to True. The Delorean and Yugo are the two most similar cars, giving us a k of 2.

if we did 3-nearest neighbors, we would end up with 2 True values and a False value, which would average out to True.

The number of neighbors we use for k-nearest neighbors (k) can be any value less than the number of rows in our dataset. In practice, looking at only a few neighbors makes the algorithm perform better, because the less similar the neighbors are to our data, the worse the prediction will be.

3.2 Similarity

Before we can predict using KNN, we need to find some way to figure out which data rows are ``closest'' to the row we're trying to predict on.

the most common measure of distance in a R^n space if the *Euclidean* or L^2 distance:

$$D = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

There are many other types of distance definition that can be used on different types of problems. E. g. to measure the real distance you have to walk in a city without crashing into building can be calculated with the *cityblock* or *Manhattan* or *L1* distance:

$$D = \sum_{i=1}^{n} |x_i - y_i|$$

see wikipedia. Also the scipy.spatial.distance module includes a lot of definitions.

[3]:
```
import scipy.spatial.distance as distance
```

[4]:
```
# distance between accord and Yugo
distance.euclidean([180,0,0],[500,1,1])
```

[4]:
```
320.00312498474136
```

You may have noticed that *horsepower* in the cars example had a much larger impact on the final distance than *racing_stripes* did, because values are much larger in absolute terms.
The same happened for surface and number of rooms in the housing example, so we can adopt once again **feature normalization** to get a data set where all features have a mean of 0 and a variance of 1.

[5]:
```
def feature_scale(data):
    data_scaled = np.empty(data.shape)
    for j in range(data.shape[1]):
        mean = np.mean(data[:,j])
        std  = np.std(data[:,j])
        data_scaled[:,j] = (data[:,j] - mean) / std
    return data_scaled
```

4 Hands on

In conclusion, to implement a kNN classifier we need: - a data set and data set loading and preprocessing utils (feature scaling) - a way to measure distance (we'll use scipy) - a way to determine the k nearest neighbours given a matrix of distances - a way to measure the accuracy (in the regression model, we have the final error)

4.0.1 Data set

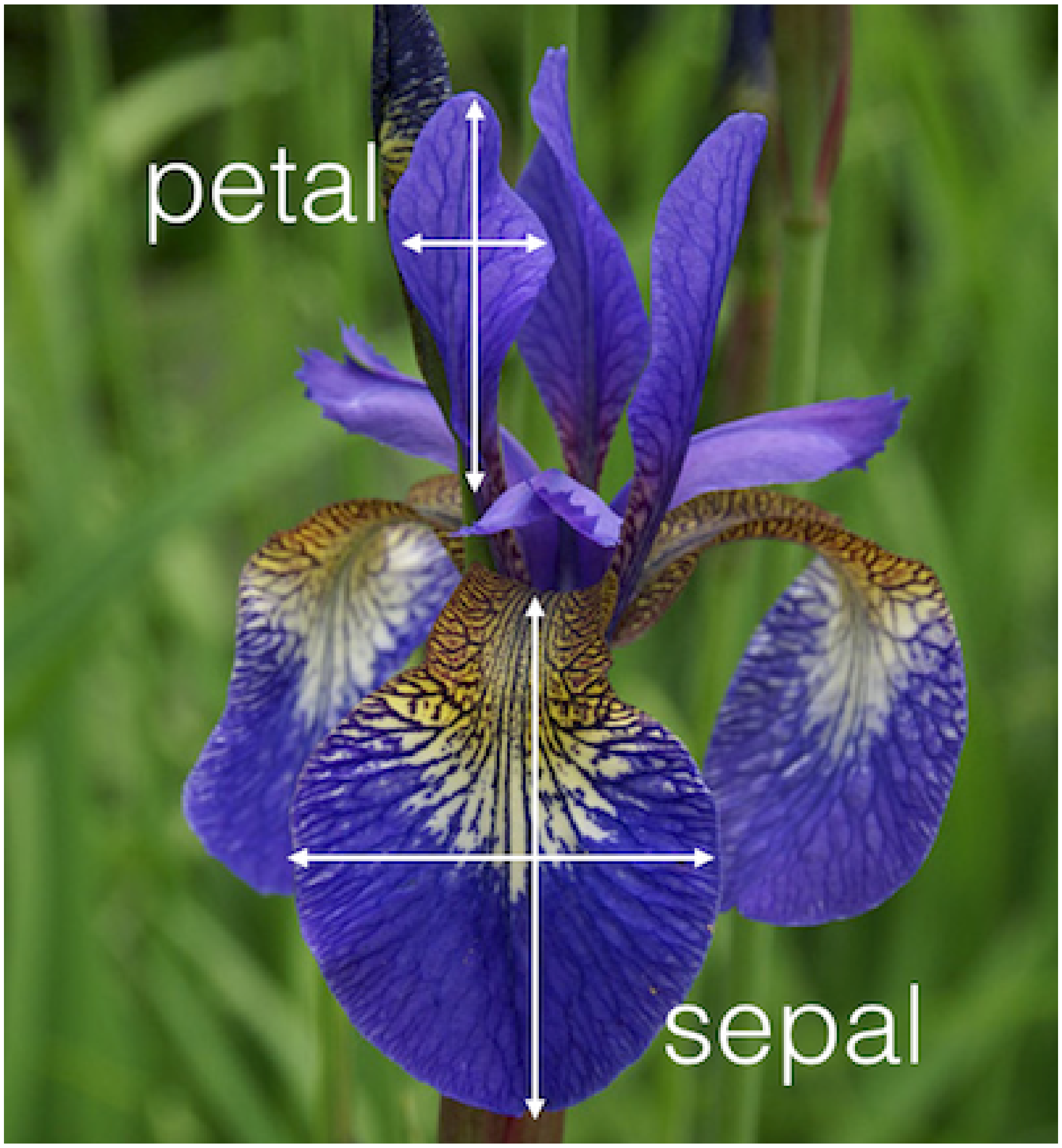
The Section ?? data set is a(n in)famous multivariate data set used for demonstration of statistical methods, used for the first time by the great Section ?? himself. It annotates the variation of morphological characters among three species of Iris flowers, (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample:

- the length of sepals and petals (in cm)
- the width of the sepals and petals (in cm)

50 specimens for species were collected (i.e. you will work with a 150x4 matrix). Let's have alook at it:

[6]:
```
Image(filename="iris_petal_sepal.png")
```

[6]:



[7]:
```
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
Y = iris.target
```

[8]:
```
X[:5], X.shape
```

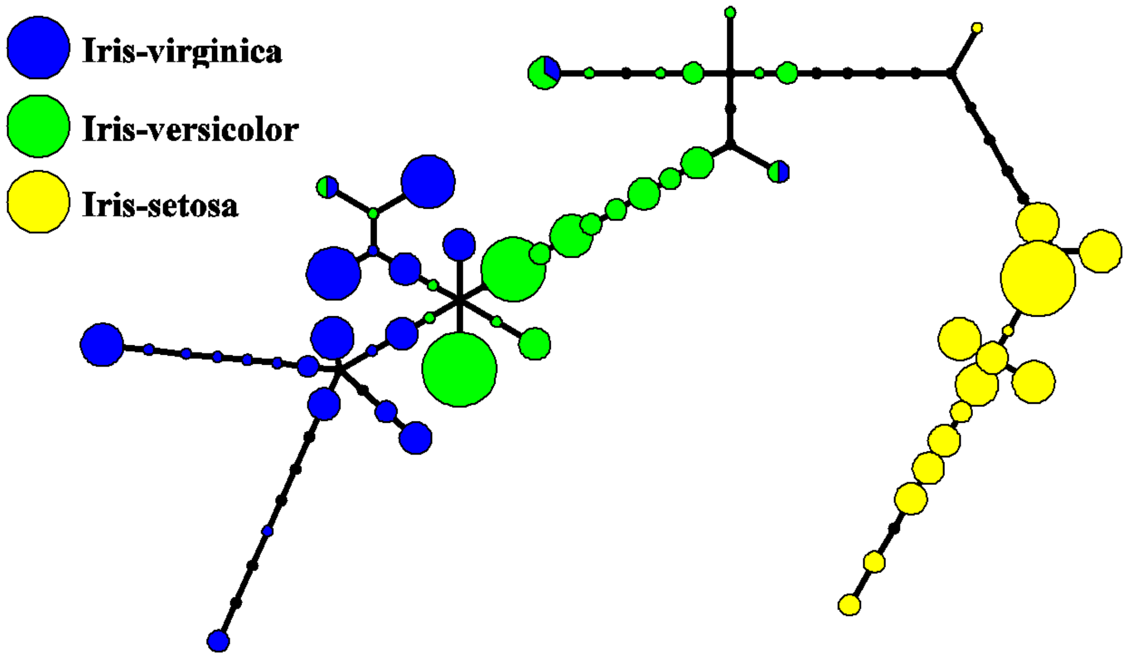
[8]:
```
(array([[5.1, 3.5, 1.4, 0.2],
        [4.9, 3. , 1.4, 0.2],
        [4.7, 3.2, 1.3, 0.2],
        [4.6, 3.1, 1.5, 0.2],
        [5. , 3.6, 1.4, 0.2]]),
 (150, 4))
```

[9]:
```
set(Y)
```

[9]:
```
{0, 1, 2}
```

[10]:
```
Image(filename="Principal_tree_for_Iris_data_set.png")
```

[10]:



The first question that you may ask yourself could be: are the selected features ``good variables'' for classifying flowers?

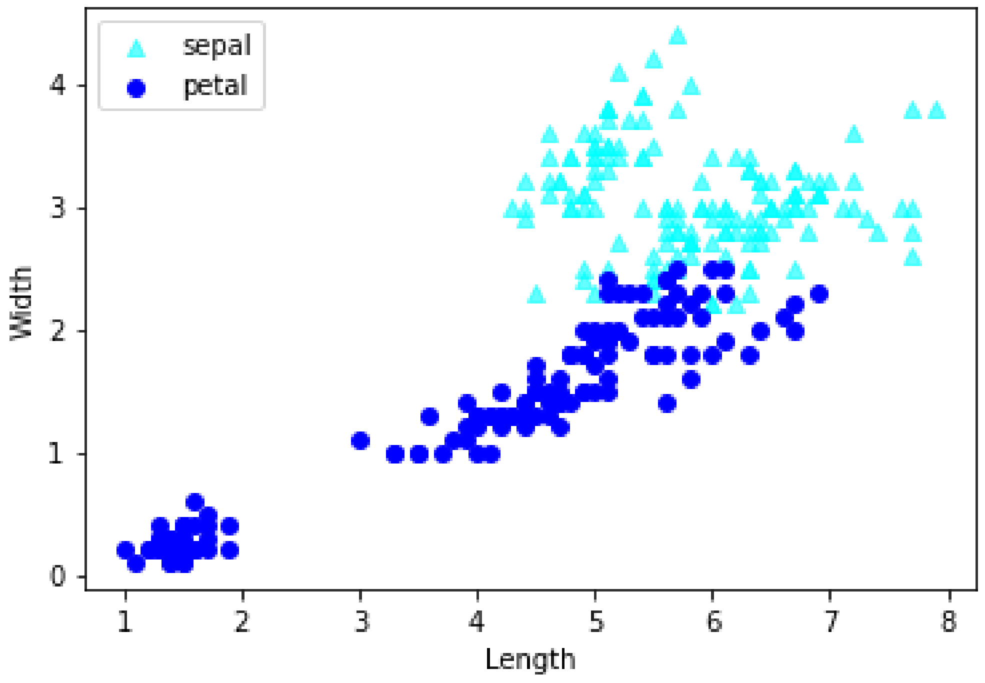
We can start with a visual inspection, starting with sepals and petals:

[11]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import numpy.linalg as LA

%matplotlib inline
```

[12]:
```
plt.scatter(X[:, 0], X[:, 1], c='cyan', marker='^',label="sepal",alpha=0.
 ↪6,edgecolor=None)
plt.scatter(X[:, 2], X[:, 3], c='blue', marker='o',label="petal",edgecolor=None)
plt.xlabel('Length')
plt.ylabel('Width')
plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x7f4d240e22b0>
```



[13]:
```
X = feature_scale(X)
X.shape
```

[13]:
```
(150, 4)
```

4.1 Find neighbours

we take advantage of distance.cdist:

Compute distance between each pair of the two collections of inputs.

and np.argsort:

Returns the indices that would sort an array.

[14]:
```
def find_knn(k, test, train, D):
    """
    given a train set and a test set,
    return the closest k neighbours
    """
    # 1 x n vector
    d = D[test,train]
    knn = np.argsort(d)[:k]
    return knn,d[knn]
```

4.2 Predict

Once we have located the most similar neighbors for a test instance, the next task is to devise a predicted response based on those neighbors.

We can do this by allowing each neighbor to vote for their class attribute, and take the majority vote as the prediction.

[15]:
```
def predict(knn, y):
    N = len(set(y))
    count   = np.zeros(N,dtype='int')
    Voters  = y[knn]
    for v in Voters:
        count[v] += 1
    return np.argmax(count)
```

4.3 Accuracy

[16]:
```
def accuracy(N, predictions, y):
    correct = 0
    for i in predictions:
        if y[i] == predictions[i]:
            correct += 1
    print(correct,N)
    return (correct/N) * 100.0
```

4.4 Run

we will pick a part of the Iris data set as a training set and try to predict to which species the remaining flowers belong to

[17]:
```
import random
```

[18]:
```
def RunKNN(k, X, y):
    # split the data set
    test_set_dim = 0.3
    nset = int(X.shape[0]*test_set_dim)
    test_set = random.sample(range(X.shape[0]),nset)
    A = set(list(range(X.shape[0])))
    train_set = A-set(test_set)
    test_set = np.asarray(list(test_set))
    train_set = np.asarray(list(train_set))

    #pre-compute distances
    D = distance.cdist(X,X,metric='euclidean')

    # for each member in the test_set, predict its class
    Predictions = dict()
    for t in test_set:
        knn, DD = find_knn(k, t, train_set, D)
        pred = predict(knn, y[train_set])
        Predictions[t] = pred

    score = accuracy(nset, Predictions, y)
    return score
```

[19]:
```
RunKNN(3, X, Y)
```

```
41 45
```

[19]:
```
91.11111111111111
```