k-Nearest Neighbor Classification (KNN)

KNN is a simple way of classifying data based on a set of training data: data with known classifications. Each observation point is classified based on the closest observation points in the training set. This gives us several parameters to play with: the number of closest neighbors taken into account (this is the k), the way we measure distance (what does "nearest" mean?), and the way we choose the class based on the classes of the closest neighbors (simple majority, or do closer neighbors carry more weight?). Some possible distance measures are:

- Minkowski distance:
$$d(x, y) = \sqrt[q]{\sum_i \Big(\big|x[i] - y[i]\big| \Big)^q}.$$
- Euclidean: Minkowski with $q = 2$. (Default in `sklearn`)
- Manhattan: Minkowski with $q = 1$.
- Hamming: number of indices in which the vectors are different. This is often used for binary data (in which case it is the same as Manhattan).

Examples of weighting schemes are:

- Uniform: all points in the set of k closest neighbors are weighted equally. (Default in `sklearn`)
- Distance: Points are weighted by $1/distance$.
- User defined: Any user-defined weighting scheme can be input as a function that turns an array of distances into an array of weights.

We use the model as follows. Suppose that we have a training set X, with a set of known labels Y. Then we use those to initialize the model. (In this case, we set $k = 1$.)

```
X= np.array([[1,1,3,4],[2,1,5,5],[5,5,1,2],[5,4,1,3],[5,5,1,1]])
Y = np.array([1,2,3,4,5])
nbrs = neighbors.KNeighborsClassifier(1)
nbrs.fit(X, Y)
```

Now, suppose we would like to classify the following training set T, we do that as follows:

```
nbrs.fit(X, Y)
nbrs.predict(T)
Out:   array([1, 2])
```

Have a look at the example data yourself. The classification should make a lot of sense.

k-Means Clustering

k-means clustering has a lot in common with Otsu's method that we learned about in detail in the image processing topic. The idea is to group a set of data points into k clusters (not to be confused with the k int he previous technique), such that we make the clusters as clustered, or as concentrated, as possible. So, if we wish to cluster into clusters $\{C_0, C_1, \ldots C_{k-1}$, then we wish to minimize the total within-cluster variation:

$$\sum_{i=0}^{k-1} \sum_{x \in C_i} d(x, \mu_i)^2.$$

Note that when we were *classifying* we did not take such things into account. The goal then was not to create cohesive sets, and the data points were classified independently of one another. When clustering, the points are clustered very much dependently on each other. Usually, the distance function is the Euclidean distance (as before). Unlike in Otsu's method, it is not usually feasible to search the space of all possible solutions, so some faster optimization algorithm is used (sometimes several times). In `sklearn`, you as a user can control what method is used. k-Means clustering is faster and more robust if the data is as low-dimensional as possible. This means that linearly correlated variables should be combined. This is called dimensionality reduction, and you have already seen one example of a method that does this: NMF. k-Means clustering is implemented as follows.

```
from sklearn.cluster import KMeans
X=  np.array([[1,1,3,4],[2,1,5,5],[5,5,1,2],[5,4,1,3],[5,5,1,1]])
kmeans = KMeans(2)
kmeans.fit(X)
kmeans.labels_
Out:   array([0, 0, 1, 1, 1], dtype=int32)
```

Again, if you look at the data, these are likely the clusters that you would have detected intuitively.