# Topic 2: $k$-Nearest Neighbours (kNN)



The *k-nearest neighbours* (kNN) algorithm is a **non-parametric** method used for classification and regression (`https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm`). When $k = 1$, it is called the ***nearest neighbour (NN) algorithm***. The strengths of kNN are simple, effective, makes no assumptions about the underlying data distribution and has a fast training phase; The weaknesses of kNN are having slow classification phase and the inability to provide mathematical reasoning on how the features are related to the class [Lantz, 2015, Chapter 3].

The kNN algorithm will not work for data which are uniformly distributed in very high dimensional feature space, i.e. when $p \gg 1$. This is because the points in $p$ dimensional spaces will be distributed "close" to the boundary and approximately equal distance from each other due to the volume $(1 - 2\epsilon)^d \ll 1$ (curse of dimensionality). So kNN only works for data of "intrinsicly" low dimension.

The assumption of kNN is "similar inputs have similar outputs". Based on this assumption, a test input $\boldsymbol{x}$ should be assigned the most common label amongst its $k$ most similar training inputs.

Application: Use in classical recommender system such as online bookshop, online movie recommendation, etc.

Given a positive integer $k$ (must be smaller than the number of data, $n$) and an input $\boldsymbol{x}$, the kNN algorithm first identifies the $k$ points in the training data $(\boldsymbol{x}_i, y_i)$ that are "similar" to $\boldsymbol{x}$ , represented by $N(\boldsymbol{x})$.

**Definition 2.0.1.** $N(\boldsymbol{x}) \subseteq D$ satisfies $|N(\boldsymbol{x})| = k$ and $\forall(\mathbf{x}', y') \in D \setminus N(\boldsymbol{x})$,

$$d(\boldsymbol{x}, \boldsymbol{x}') \geq \max_{(\boldsymbol{x}'', y'') \in N(\boldsymbol{x})} d(\boldsymbol{x}, \boldsymbol{x}''),$$

(i.e. every point in $D$ but not in $N(\boldsymbol{x})$ is at least as far away from $\mathbf{x}$ as the furthest point in $N(\boldsymbol{x})$).

We can then define

- kNN classifier:

$$\widehat{Y} = h(\mathbf{x}) = \text{mode}(\{y'' : (\mathbf{x}'', y'') \in N(\boldsymbol{x})\}), \tag{2.1}$$

$$\mathbb{P}(Y = j | \boldsymbol{X} = \boldsymbol{x}) = \frac{1}{k} \sum_{\boldsymbol{x}_i \in N(\boldsymbol{x})} I(y_i = j). \tag{2.2}$$

  where mode$(\cdot)$ means to select the label of the highest occurrence.

- kNN regressor:

$$h(\boldsymbol{x}) = \frac{1}{k} \sum_{(\boldsymbol{x}'', y'') \in N(\boldsymbol{x})} y''. \tag{2.3}$$

  In case of a draw, a good solution is to return the result of $k$-NN with smaller $k$.

## 2.1 Dissimilarity/Distance Measure

The kNN algorithm, clustering algorithms, recommender systems fundamentally relies on the **dissimilarity** metric — a function which takes two "vectors" and returns a non-negative number defined below.

**Definition 2.1.1.** A ***dissimilarity function*** $d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a function which satisfies the following conditions: For any $\boldsymbol{x}_i$, $\boldsymbol{x}_j$, $\boldsymbol{x}_k$,

(i) $d(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0$ and $d(\boldsymbol{x}_i, \boldsymbol{x}_j) = 0$ iff $\boldsymbol{x}_i = \boldsymbol{x}_j$;

(ii) $d(\boldsymbol{x}_i, \boldsymbol{x}_j) = d(\boldsymbol{x}_j, \boldsymbol{x}_i)$; and

When the dissimilarity function satisfies

(iii) $d(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq d(\boldsymbol{x}_i, \boldsymbol{x}_k) + d(\boldsymbol{x}_k, \boldsymbol{x}_j)$ (triangle inequality).

it is called a **distance/metric function**.

For a kNN classifier, the better that metric reflects label similarity, the better the classification will be.

The most common choice is the *Minkowski distance*:

$$d(\boldsymbol{x}, \boldsymbol{z}) = \|\boldsymbol{x} - \boldsymbol{z}\|_r = \left( \sum_{i=1}^{p} |x_i - z_i|^r \right)^{\frac{1}{r}}, \quad \boldsymbol{x},\ \boldsymbol{z} \in \mathbb{R}^p. \tag{2.4}$$

Note that $\|\cdot\|^r$ is called the $\ell^r$ norm.

When $r = 1$, we have the *Manhattan distance*:

$$\|\boldsymbol{x} - \boldsymbol{z}\|_1 = |x_1 - z_1| + |x_2 - z_2| + \cdots + |x_p - z_p|. \tag{2.5}$$

When $r = 2$, we have the *Euclidean distance*:

$$\|\boldsymbol{x} - \boldsymbol{z}\|_2 = \sqrt{(x_1 - z_1)^2 + (x_2 - z_2)^2 + \cdots + (x_p - z_p)^2}. \tag{2.6}$$

When $r = \infty$, we have the *Chebyshev distance*:

$$\|\boldsymbol{x} - \boldsymbol{z}\|_\infty = \max\{|x_1 - z_1|, |x_2 - z_2|, \cdots, |x_p - z_p|\}. \tag{2.7}$$

Note that the Euclidean distance is more sensitive to outliers than the Manhattan distance. When outliers are rare, the Euclidean distance performs very well and is generally preferred. When the outliers are significant, the Manhattan distance is more stable.

The three most common metrics in user-based recommender system are

- Pearson correlation coefficient (PC):

$$PC \ : \ \mathbb{R}^p \times \mathbb{R}^p \to [-1,1], \quad (\boldsymbol{x},\boldsymbol{z}) \mapsto \frac{\sum_j (x_j - \overline{\boldsymbol{x}})(z_j - \overline{\boldsymbol{z}})}{\sqrt{\sum_j (x_j - \overline{\boldsymbol{x}})^2 \cdot \sum_j (z_j - \overline{\boldsymbol{z}})^2}}.$$

- Cosine measure or *cosine similarity*:

$$COS \ : \ [0,\infty)^p \times [0,\infty)^p \to [0,1], \quad (\boldsymbol{x},\boldsymbol{z}) \mapsto \frac{\sum_j x_j z_j}{\sqrt{\sum_j x_j^2} \cdot \sqrt{\sum_j z_j^2}} =: \frac{\langle \boldsymbol{x},\boldsymbol{y} \rangle}{\|\boldsymbol{x}\|_2 \cdot \|\boldsymbol{y}\|_2}.$$

- Mean square difference (similar to MSE except that it compares common rating items only):

$$MSD \ : \ \mathbb{R}^p \times \mathbb{R}^p \to [0,\infty], \quad (\boldsymbol{x},\boldsymbol{z}) \mapsto \frac{\sum_{j \in R_x \cap R_z} (x_j - z_j)^2}{|R_x \cap R_z|}.$$

### 2.1.1 Manhattan Distance

The Manhattan distance (2.5) is measured along axes at right angles and in R programming, we need to specify its usage by the expression `dist(x,method='manhattan')`. In Python, it is implemented as `cityblock` in `scipy.spatial.distance` and `manhattan_distances` in `sklearn.metrics.pairwise`.

**Example 2.1.2** (Final Exam Jan 2019, Q1(a))**.** State and explain any two types of distance measures. (4 marks)

**Solution**: Consider two $n$-dimensional points be $\boldsymbol{x} = (x_1, \cdots, x_n)$, $\boldsymbol{y} = (y_1, \cdots, y_n)$.
Euclidean distance: $\sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}$
Manhattan distance: $|x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n|$

### 2.1.2 Cosine Similarity

The cosine similarity is generally used as a metric for measuring distance when the *magnitude of the vectors does not matter.*

The ***cosine distance*** is a complement of cosine similarity and it is the angular distance between two points, i.e.

$$\text{cos-dist}(\boldsymbol{x},\boldsymbol{y}) = 1 - \text{COS}(\boldsymbol{x},\boldsymbol{y}).$$

It is not implemented in R by default. It is implemented in additional R packages such as `proxy::dist(x, method="cosine")`, `text2vec::sim2(x,method="cosine")`, `lsa::cosine()`, `clv::dot_product()`, `rules:: dissimilarity()`. `scipy.spatial.distance.cosine` and `sklearn.metrics.pairwise.cosine_distances` are the implementations in Python.

Cosine distance is a dissimilarity function but **not a distance function**. Consider $A = (1,0)$, $B = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$, $C = (0,1)$. Note that vectors A and C are orthogonal, so we would get simply 0:

$$\text{COS}(A,C) = \frac{0}{\sqrt{1}\sqrt{1}} = 0.$$

Each pair of vectors A and B as well as B and C would give the same value:

$$\text{COS}(A,B) = \frac{\frac{\sqrt{2}}{2} + 0}{\sqrt{1}\sqrt{1}} = \frac{\sqrt{2}}{2}, \quad \text{COS}(B,C) = \frac{0 + \frac{\sqrt{2}}{2}}{\sqrt{1}\sqrt{1}} = \frac{\sqrt{2}}{2}.$$

Since the cosine distance does not satisfy the triangle inequality and it is therefore not a distance function:

$$\text{cos-dist}(A,C) = 1 - 0 \not\leq \text{cos-dist}(A,B) + \text{cos-dist}(B,C) = (1 - \frac{\sqrt{2}}{2}) + (1 - \frac{\sqrt{2}}{2}) = 2 - \sqrt{2}.$$

Despite the cosine distance is not a distance function, it is used in the construction of distance matrix for text data represented by word counts. We could assume that when a word (e.g. science) occurs more frequent in document A than document B, then document A is more related to the topic of science (shorter distance). However, it could also be the case that we are working with documents of uneven lengths. Then, the word 'science' probably occurred more in document A just because it was way longer than document B. Cosine similarity corrects for this.

**Example 2.1.3.** Consider two 'naive word counts' for Calculus 1 and Calculus 2 texts with six 'popular common words':

| | . | = | + | the | f | x |
|---|---|---|---|---|---|---|
| Calculus 1 | 3626 | 1446 | 915 | 798 | 552 | 556 |
| Calculus 2 | 926 | 476 | 317 | 356 | 283 | 146 |

Calculate the cosine dissimilarity.

**Solution:**

$$\text{cos-dist}(\text{Calculus 1}, \text{Calculus 2}) = 1 - \frac{4857507}{\sqrt{17326661}\sqrt{1412682}} = 1 - 0.981824 = 0.018176$$

So calculus 1 and calculus 2 text are 'similar' based on the 6 words. However, if we include the technical words and remove common words, it may not be the case. Therefore, for text comparison, choosing the right 'dictionary' (i.e. the 'headers' of the table) is essential.

### 2.1.3 Other Dissimilarities

**Mahalanobis distance** (knnGarden::knnMCN):

$$Mahalanobis(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} - \boldsymbol{y})\Sigma^{-1}(\boldsymbol{x} - \boldsymbol{y})^T$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are row vectors and $\Sigma$ is the covariance of the data.

The Mahalanobis distance is useful when attributes are correlated, but having different ranges of values and the distribution of the data is approximately Gaussian/normal.

**Similarity Measures for Binary Data**: Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two objects with binary attributes and $p$ be the number of attributes.

1. Simple Matching Coefficients: $SMC(\boldsymbol{x}, \boldsymbol{y}) = \dfrac{\text{number of matching attribute values}}{p}$
2. Jaccard score: $JAC(\boldsymbol{x}, \boldsymbol{y}) = \dfrac{\text{number of matching attribute values}}{\text{number of attribute not involved in 00 matches}}$

**Example 2.1.4.** Let $\boldsymbol{x} = (1,0,0,0,0,0,0,0,0,0)$, $\boldsymbol{y} = (0,0,0,0,0,0,1,0,0,1)$.

$$SMC(\boldsymbol{x}, \boldsymbol{y}) = \frac{7}{10} = 0.7$$
$$JAC(\boldsymbol{x}, \boldsymbol{y}) = \frac{0}{3} = 0$$

**Tanimoto score**: It is used for data that can take on continuous values. It and Jaccard score are used in Chemoinformatics, plagiarism detection, thesauras extration, market-basket transactional data, anomalies detection in spatio-temporal data.

**Gower distance**: It can be used to process mixed numeric and categorical data.

## 2.2 Bayes Optimal Classifier Theory

The best classifier that minimises the *misclassification rate* or the *Bayes error rate / Bayes risk* [Devroye et al., 1996, Chapter 3],

$$L(f) = \mathbb{P}(\{\omega : Y \neq f(\boldsymbol{X})\})$$

is called the *Bayes (optimal) classifier* [Mitchell, 1997, Chapter 6]:

$$h_{\text{opt}}(\boldsymbol{x}) = \underset{y \in \{1,2,\ldots,K\}}{\operatorname{argmax}} \mathbb{P}(Y = y \mid \boldsymbol{X} = \boldsymbol{x}). \tag{2.8}$$

The **error rate** of $h_{\text{opt}}(\boldsymbol{x})$ is

$$\epsilon_{BayesOpt} = 1 - \mathbb{P}(y^*|\boldsymbol{x}).$$

It is the **theoretical lower bound of the error rate for any classifier**.

**Example 2.2.1.** Assume an email $\boldsymbol{x}$ can either be classified as spam $(+1)$ or ham $(-1)$. For the same email $\boldsymbol{x}$ the conditional class probabilities are:

$$\mathbb{P}(+1|\boldsymbol{x}) = 0.8, \quad \mathbb{P}(-1|\boldsymbol{x}) = 0.2.$$

In this case the Bayes optimal classifier would predict the label $y^* = +1$ as it is most likely, and its error rate would be $\epsilon_{BayesOpt} = 0.2$.

The **upper bound on the error** is given by the constant classifier.

In practise, the distribution of the random variables $\boldsymbol{X}$ is unknown and so the classifier has to be determined from the observations $\{(\boldsymbol{x}_i, y_i)\}$ by minimising the expected 0-1 loss:

$$\widehat{L}(f) = \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(\hat{y}_i \neq y_i)\right]. \tag{2.9}$$

The kNN Classifier's 'ability to predict' is characterised by the following theory.

**Theorem 2.2.2.** *As $n \to \infty$, the 1-NN error is no more than twice the error of the Bayes Optimal classifier:*

$$\epsilon_{\text{BayesOpt}} \leq \epsilon_{NN} \leq 2\epsilon_{\text{BayesOpt}}$$

*Similar guarantees hold for $k > 1$.*

According to Thomas Cover and Peter Hart, "Nearest neighbor pattern classification", IEEE Transactions on Information Theory, 1967, 13(1):21-27, a **Proof** goes like:

Let $\boldsymbol{x}_{NN}$ be the nearest neighbour of our test point $\boldsymbol{x}_t$. As $n \to \infty$, $d(\boldsymbol{x}_{NN}, \boldsymbol{x}_t) \to 0$, i.e. $\boldsymbol{x}_{NN} \to \boldsymbol{x}_t$. This means the nearest neighbour is identical to $\boldsymbol{x}_t$. You return the label of $\boldsymbol{x}_{NN}$. What is the probability that this is not the label of $\boldsymbol{x}_t$? It is the probability of drawing two different label of $\boldsymbol{x}$:

$$\begin{aligned}\epsilon_{NN} &= \mathrm{P}(y^*|\mathbf{x}_t)(1 - \mathrm{P}(y^*|\mathbf{x}_{NN})) + \mathrm{P}(y^*|\mathbf{x}_{NN})(1 - \mathrm{P}(y^*|\mathbf{x}_t)) \\ &\leq (1 - \mathrm{P}(y^*|\mathbf{x}_{NN})) + (1 - \mathrm{P}(y^*|\mathbf{x}_t)) = 2(1 - \mathrm{P}(y^*|\mathbf{x}_t)) = 2\epsilon_{\text{BayesOpt}},\end{aligned}$$

where the inequality follows from $\mathrm{P}(y^*|\mathbf{x}_t) \leq 1$ and $\mathrm{P}(y^*|\mathbf{x}_{NN}) \leq 1$. We also used that $\mathrm{P}(y^*|\mathbf{x}_t) = \mathrm{P}(y^*|\mathbf{x}_{NN})$.

## 2.3 Algorithms of kNN Classifier

A pure Python implementation of kNN is given in Grus [2015, Chapter 12]. If numpy library is used, a simple implementation (involving brute force search) can be stated below.

```
# https://github.com/joelgrus/data-science-from-scratch
from scipy.spatial.distance import euclidean
import logging
logging.basicConfig(level=logging.DEBUG)

def knn_classify(k: int, X, y, x_new, distance=euclidean) -> str:
    import numpy as np, collections
    dist_list = [distance(x, x_new) for x in X]
    logging.debug('distances={}'.format([round(d,4) for d in dist_list]))
    order = np.argsort(dist_list)
    k_nearest_labels = y[order[:k]]
    logging.debug('knn={}'.format(k_nearest_labels))
    vote_counts = collections.Counter(k_nearest_labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    return winner, winner_count/k

if __name__=='__main__':
    from sklearn import datasets
    iris_df = datasets.load_iris()
    X = iris_df['data']; y = iris_df['target']
    knn_classify(1,X,y,[6,3,5,2]); knn_classify(1,X,y,[5,3,1,0])
```

A more sophisticated algorithm involving KD tree and ball tree is implemented in `sklearn.neighbors.KNeighb`

In R, the `knn` classifier is available in the `class` library which only supports Euclidean distance and has the following form.

```
knn(train, test, cl, k = 1, l = 0, prob = FALSE, use.all = TRUE)
```

Here,

- `train`: matrix or data frame of training set cases.
- `test`: matrix or data frame of test set cases.
- `cl`: factor of true classifications of training set
- `k`: number of neighbours considered.

Let us practise the algorithm with an example. More examples are found in the tutorial.

**Example 2.3.1.** A sport school would like to group their new enrolled students into 2 groups, as according to the existing students' weight and height. The weight and height of 7 existing students with group are shown in the table below.

| Student | Weight (kg) | Height (cm) | Group |
|---|---|---|---|
| A | 29 | 118 | A |
| B | 53 | 137 | B |
| C | 38 | 127 | B |
| D | 49 | 135 | B |
| E | 28 | 111 | A |
| F | 24 | 111 | A |
| G | 30 | 121 | A |

(a) Perform and use $k = 3$-NN method (with Euclidean distance) to predict which group the following students will be grouped into, based on **cut-off of 0.7** on group A.

| Student | Weight (kg) | Height (cm) |
|---|---|---|
| H | 35 | 120 |
| I | 47 | 131 |
| J | 22 | 115 |
| K | 38 | 119 |
| L | 31 | 136 |

**Solution**: First, we construct a distance table from the testing data to the training data and take note that $k = 3$ and cut-off $= 0.7$:

| | | H | I | J | K | L |
|---|---|---|---|---|---|---|
| Student | Group | Distance | Distance | Distance | Distance | Distance |
| A | A | 6.3246 | 22.2036 | 7.6158 | 9.0554 | 18.1108 |
| B | B | 24.7588 | 8.4853 | 38.0132 | 23.4307 | 22.0227 |
| C | B | 7.6158 | 9.8489 | 20.0000 | 8.0000 | 11.4018 |
| D | B | 20.5183 | 4.4721 | 33.6006 | 19.4165 | 18.0278 |
| E | A | 11.4018 | 27.5862 | 7.2111 | 12.8062 | 25.1794 |
| F | A | 14.2127 | 30.4795 | 4.4721 | 16.1245 | 25.9615 |
| G | A | 5.0990 | 19.7231 | 10.0000 | 8.2462 | 15.0333 |
| $P(Y = A)$ | | 0.6667 | 0.0000 | 1.0000 | 0.6667 | 0.3333 |
| Cut-off $= 0.5$ | $\hat{y}$ | A | B | A | A | B |
| Cut-off $= 0.7$ | $\hat{y}$ | B | B | A | B | B |

How do we get $d(A, H) = 6.3246$? The calculation is as follows:

$$d(A, H) = \sqrt{(29 - (35))^2 + (118 - (120))^2} = \sqrt{40} = 0.5673$$

The rest is similar.

(b) The actual groups of the students are {A, B, A, B, B} for students {H, I, J, K, L} respectively. Construct a confusion matrix and calculate the accuracy measurements for a cut-off of 0.5 and a cut-off of 0.7.

(c) Write a Python script to produce the above calculations using the simple implementation by lecturer in Section 2.3 and also use sklearn's implementation.

**Solution**:

```
import pandas as pd
d_train = pd.DataFrame({
    'weight': [29,53,38,49,28,24,30],
    'height': [118,137,127,135,111,111,121],
    'group' : ['A','B','B','B','A','A','A']
}, index = list('ABCDEFG'))
d_test = pd.DataFrame({
    'weight': [35, 47, 22, 38, 31],
    'height': [120, 131, 115, 119, 136]
}, index = list('HIJKL'))

from knn_naive import *
X = d_train[['weight','height']].values
y = d_train['group']
X_test = d_test[['weight','height']].values
for new_x in X_test:
    print(knn_classify(3,X,y,new_x))

from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3, p=2)
model.fit(X,y)
print(model.predict(X_test))  # cut-off = 0.5
print(model.predict_proba(X_test))
```

(d) Write an R script to produce the above calculations.

**Solution**:

```
X_y.train = data.frame(row.names = LETTERS[1:7],
    weight = c(29,53,38,49,28,24,30),
    height = c(118,137,127,135,111,111,121),
    group  = c('A','B','B','B','A','A','A'))
X.test = data.frame(row.names = LETTERS[8:12],
    weight = c(35, 47, 22, 38, 31),
    height = c(120, 131, 115, 119, 136))

dist.table = data.frame(row.names = LETTERS[1:7])
norm2 = function(x){return(sqrt(sum(x^2)))}
X = X_y.train[,1:2]
print(X)
for(i in 1:nrow(X.test)) {
    dist.table[,i] = apply(t(X)-t(X.test)[,i],2,norm2)
}
names(dist.table) = row.names(X.test)
print(round(dist.table,4))

library(class)  # for knn
```

```
M = ncol(X_y.train)
X.test$group = knn(train=X_y.train[,-M], test=X.test,
                   cl   =X_y.train[, M], k=3, prob=TRUE)
X.test$prob  = attr(X.test$group,"prob")
X.test$prob  = ifelse(X.test$group=="A", X.test$prob, 1-X.test$prob)
cut.off = 0.7
X.test$group = ifelse(X.test$prob>cut.off, "A", "B")
print(cbind(round(X.test[,1:2],4),prob.A=round(X.test[,4],4),
            predicted=X.test$group, actual=c('A','B','A','B','B')))
```

## 2.4 Algorithms of kNN Regressor

The algorithms for the *kNN regressor* are similar to the kNN classifier except that the "mean" is used to estimate the output as in (2.3) [Altman, 1992].

In Python, `sklearn.neighbors.KNeighborsRegressor` implements the kNN regressor with Minkowski distance; In R, `FNN`'s `knn.reg` provides an implementation with the Euclidean distance.

We can practise the algorithm with an example below.

**Example 2.4.1** (Exam SRM Study Manual, p225, Q15.10)**.** A continuous variable $Y$ is modelled as a function of $X$ using kNN with $k = 3$. With the following data:

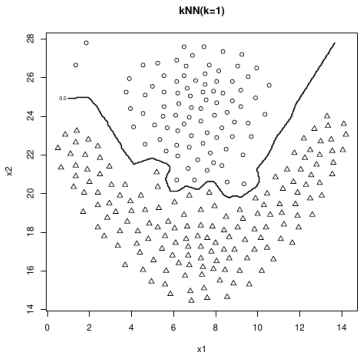
| $X$ | 5 | 8 | 15 | 22 | 30 |
|---|---|---|---|---|---|
| $Y$ | 4 | 1 | 10 | 16 | 30 |

Calculate the fitted value of $Y$ at $X = 12$. Try write a script using R (or Python) to perform the calculation for you.

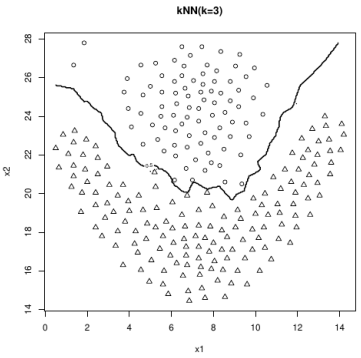
## 2.5 Classifier Boundary

**Classifier boundary** = Points $\boldsymbol{x}$ in the domain which do not belong to any output class because $P(Y|X = \boldsymbol{x})$ is not clearly defined.

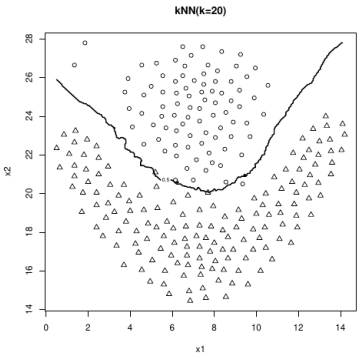
**Example 2.5.1.** Analyse the 'flame' dataset from `https://cs.joensuu.fi/sipu/datasets/flame.txt` using the kNN classifier with $k = 1$, $k = 3$, $k = 20$, $k = 60$, $k = 100$.
**Solution:** The boundaries of 1-NN (Figure 2.1a) and 3-NN (Figure 2.1b) are rather "nonlinear" and "rough". As $k$ is increased to 100, the boundary becomes "smoother" and usually "straighter": when $k = 20$, the boundary seems to be "smoother" in Figure 2.1c while the boundary becomes "straighter" changing when $k$ is increased to $k = 60$ (Figure 2.1d) to $k = 100$ (Figure 2.1e).



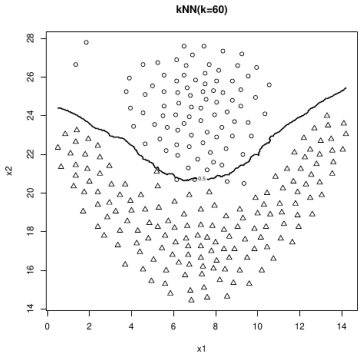

(a) 1-NN boundary



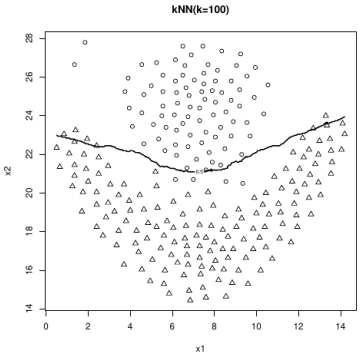

(b) 3-NN boundary




(c) 20-NN boundary




(d) 60-NN boundary



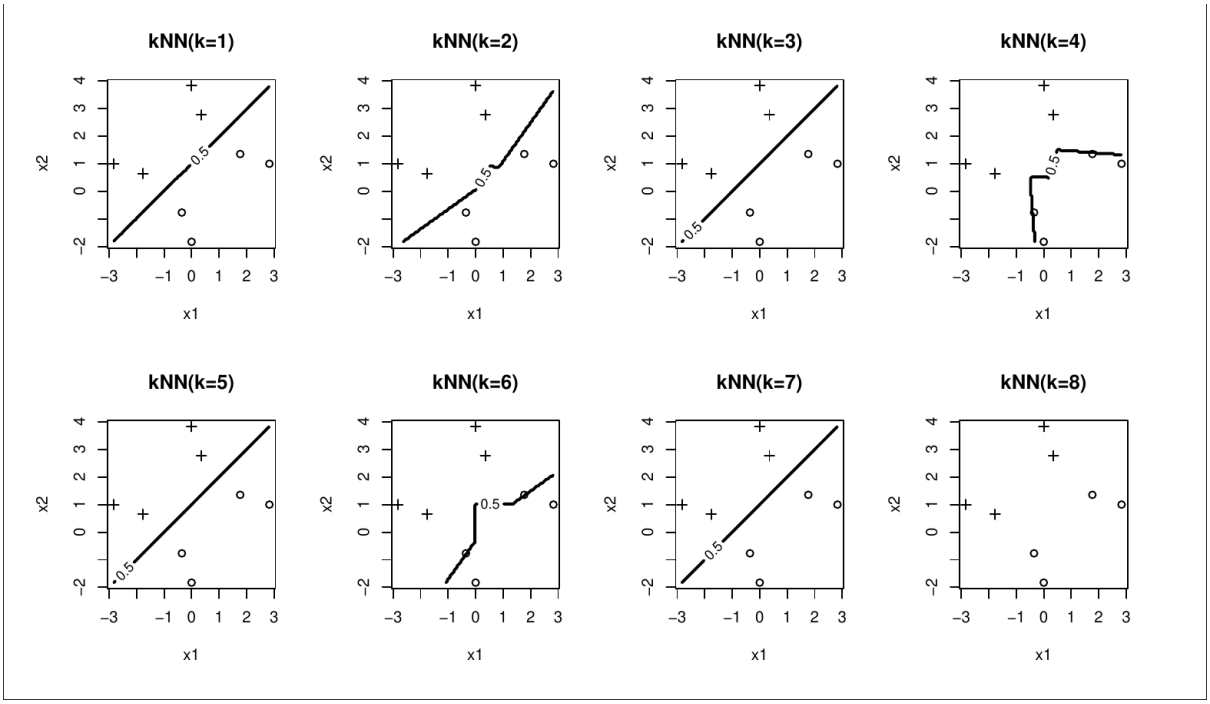

(e) 100-NN boundary

**Example 2.5.2** (Symmetric Data Can Lead to Bias due to Class Order)**.** Consider the data

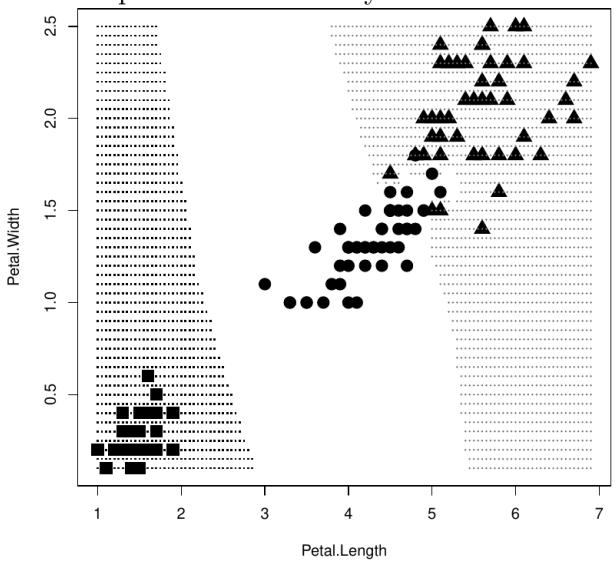
| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0.0000 | 3.8284 | + |
| 0.3536 | 2.7678 | + |
| −2.8284 | 1.0000 | + |
| −1.7678 | 0.6464 | + |
| 2.8284 | 1.0000 | ∘ |
| 1.7678 | 1.3536 | ∘ |
| 0.0000 | −1.8284 | ∘ |
| −0.3536 | −0.7678 | ∘ |

Analyse the decision boundary for $k = 1$ to $k = 8$.

**Solution:** By definition, the decision should be a straight line for all $k$ but when $k$ is even, we will have a draw and in R's implementation, the class order first will be chosen, leading to non-diagonal line as shown below:




**Example 2.5.3** (Three-class Output)**.** The iris data has 4 variables, but if we pick just two variables, we can "visualise" the decision boundary using 'point labels' or 'colours' rather than the contour plot as in the binary classifications in the earlier examples.




We are fortunate that the three classes of iris flowers are 'distinguishable'. Otherwise, the boundary will be mixed and very difficult to identify.

**Conclusion:** the choice of $k$ depends upon the data. In general, larger values of $k$ reduces effect of the noise on the classification, but make boundaries between classes less distinct. A good $k$ can be selected by various heuristic techniques (e.g. based of prediction accuracy). When $k = 1$, the decision boundary is over "flexible". This corresponds to a classifier that has low bias but very high variance. As $k$ grows, the classifier becomes less flexible and produces a decision boundary that is closed to linear. This corresponds to a low-variance but high bias classifier.

## 2.6 Feature Scaling

- Features with large variations may shadow features with small variations in Euclidean distance measures.
- Scaling such as min-max scaling and standardisation (using R's `scale`) can help putting all features to the same ground.

**Example 2.6.1** (Standardisation)**.** For the training data and testing data from Example 2.3.1, apply the standardisation and then perform $k$ = 3-NN (with Euclidean distance) to predict which group the following students will be grouped into, based on a **cut-off of 0.5** and a **cut-off of 0.7** on group A.

**Solution:**

| Student | Weight (kg) | Height | Std_Wgt | Std_Hgt | Group |
|---|---|---|---|---|---|
| A | 29 | 118 | -0.6113 | -0.4587 | A |
| B | 53 | 137 | 1.5284 | 1.3355 | B |
| C | 38 | 127 | 0.1910 | 0.3912 | B |
| D | 49 | 135 | 1.1717 | 1.1467 | B |
| E | 28 | 111 | -0.7005 | -1.1197 | A |
| F | 24 | 111 | -1.0571 | -1.1197 | A |
| G | 30 | 121 | -0.5222 | -0.1754 | A |
| Mean | 35.8571 | 122.8571 | | | |
| Std. dev | 11.2165 | 10.5898 | | | |

**Important Note: The training data MUST used the standardisation used in the training data because the kNN classifier is TRAINED with the training data**.

$$\text{I:}\quad \frac{47-35.8571}{11.2165}\approx 0.9934,\quad \frac{131-122.8571}{10.5898}\approx 0.7689$$

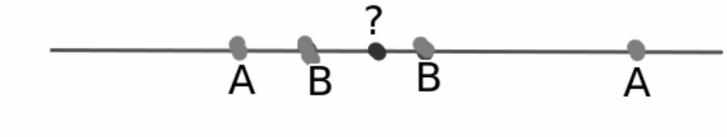
| Student | Weight (kg) | Height (cm) | Std_Wgt | Std_Wgt |
|---|---|---|---|---|
| H | 35 | 120 | -0.0764 | -0.2698 |
| I | 47 | 131 | 0.9934 | 0.7689 |
| J | 22 | 115 | -1.2354 | -0.7420 |
| K | 38 | 119 | 0.1910 | -0.3642 |
| L | 31 | 136 | -0.4330 | 1.2411 |

By performing Euclidean distance calculation, we obtain $d(A,H) = \sqrt{(-0.6113-(-0.0764))^2+(-0.4587-(-0.2698))^2} = \sqrt{0.3218} = 0.5673$, etc. leading to the table below.

| | | H | I | J | K | L |
|---|---|---|---|---|---|---|
| | Group | Distance | Distance | Distance | Distance | Distance |
| A | A | 0.5673 | 2.0205 | 0.6854 | 0.8079 | 1.7091 |
| B | B | 2.2699 | 0.7792 | 3.4575 | 2.1628 | 1.9637 |
| C | B | 0.7131 | 0.8869 | 1.8218 | 0.7554 | 1.0544 |
| D | B | 1.8879 | 0.4177 | 3.0596 | 1.8013 | 1.6076 |
| E | A | 1.0544 | 2.5370 | 0.6548 | 1.1686 | 2.3759 |
| F | A | 1.2977 | 2.7878 | 0.4177 | 1.4590 | 2.4419 |
| G | A | 0.4557 | 1.7857 | 0.9109 | 0.7378 | 1.4193 |
| $P(Y=A)$ | | 0.6667 | 0.0000 | 1.0000 | 0.6667 | 0.3333 |
| Cut-off = 0.5 | $\hat{y}$ | A | B | A | A | B |
| Cut-off = 0.7 | $\hat{y}$ | B | B | A | B | B |

## 2.7 Weighted kNN Models

Some academicians propose an extension to kNN by considering the following scenario:




Here, a new point marked with '?' has 4 similar/closest points. Suppose the kNN with $k = 4$ is used. The conditional probability

$$P(Y = A|X =?) = P(Y = B|X =?) = 0.5$$

is obtained. If we use dictionary ordering, $A$ is before $B$, so kNN($k = 4$) will give $Y = A$. This may not be reasonable since B is closer to '?'.

**Weighted kNN** (wkNN, available in R in `kknn` library) introduces weights (called kernel) to solve this problem [Hechenbichler and Schliep, 2004].

1. Let $N(\boldsymbol{x}, k+1)$ be the $k+1$ nearest neighbours to $\boldsymbol{x}$ according to a distance function $d(\boldsymbol{x}, \boldsymbol{x}_i)$.
2. The $(k+1)$th neighbour is used for "standardisation" of the $k$ smallest distances via

$$D_i = \frac{d(\boldsymbol{x}, \boldsymbol{x}_i)}{d(\boldsymbol{x}, \boldsymbol{x}_{k+1})}.$$

3. A weighted majority of the $k$ nearest neighbour

$$\hat{y} = \max_j \left\{ \sum_{i=1}^{k} K(D_i) I(y_i = j) \right\}.$$

In order for the 'weight' to make sense, we need it to satisfy some requirements. The 'weight' $K$ is called a *kernel (function)* if it satisfies

(1) $K(x) \geq 0$ for all $x \in \mathbb{R}$;
(2) $K(x)$ is maximum when $x = 0$;
(3) $K(x)$ descents monotonously when $x \to \pm\infty$.

Available kernels in R's kknn or Python's sklearn:

- rectangular/uniform: $K(x) = \frac{1}{2} I(|x| \leq 1)$;
- inv/distance: $K(x) = 1/|x|$
- triangular: $K(x) = (1 - |x|) \cdot I(|x| \leq 1)$.
- optimal: The number of neighbours used for this kernel should be $\left(\frac{2(d+4)}{d+2}\right)^{d/(d+4)} k \in [1.2, 2]$, where $k$ is used
- Others: cos, gaussian, rank, epanechnikov (or beta(2,2)), biweight (or beta(3,3)), triweight (or beta(4,4)).

The wkNN is implemented in Python's `KNeighborsClassifier` and `KNeighborsRegressor` and R's kknn's `kknn`. By default, Python's kNN algorithms sets the "weights" parameter to "uniform", which is the same as the usual kNN.

However, R's `kknn` sets the "kernel" to "optimal" by default, which leads to a different calculation from the usual kNN. To set it to be the usual kNN, the "kernel" needs to be set to "rectangular". For each row of the test set, the k nearest training set vectors (according to Minkowski distance) are found, and the classification is done via the maximum of summed kernel densities. In addition, even ordinal and continuous variables can be predicted.

## 2.8 Optimal: Data Partitioning Algorithms

To **make kNN faster during prediction**, Python sklearn's kNNs implement more sophisticated methods to "find" the nearest neighbours.

`brute`**-force method**. It finds all distances and sort in order.

`kd_tree` **method**. It partitions the feature space by (a) dividing the data into two halves, e.g. left and right, along one feature. (b) For each training input, remember the half it lies in.

How can this partitioning speed up prediction? Let's think about it for the one neighbour case.

1. Identify which side the test point lies in, e.g. the right side.
2. Find the nearest neighbour $x^R_{\text{NN}}$ of $x_t$ in the same side. The $R$ denotes that our nearest neighbour is also on the right side.
3. Compute the distance between $x_y$ and the dividing "wall". Denote this as $d_w$. If $d_w > d(x_t, x^R_{\text{NN}})$ you are done, and we get a 2x speedup.

In other words: if the distance to the partition is larger than the distance to our closest neighbour, we know that none of the data points inside that partition can be closer. We can avoid computing the distance to any of the points in that entire partition. We can prove this formally with the triangular inequality.

Let $d(x_t, x)$ denote the distance between our test point $x_t$ and a candidate $x$. We know that $x$ lies on the other side of the wall, so this distance is dissected into two parts $d(x_t, x) = d_1 + d_2$, where $d_1$ is the part of the distance on $x'_t s$ side of the wall and $d_2$ is the part of the distance on $x's$ side of the wall. Also let $d_w$ denote the shortest distance from $x_t$ to the wall. We know that $d_1 > d_w$ and therefore it follows that

$$d(x_t, x) = d_1 + d_2 \geq d_w + d_2 \geq d_w.$$

This implies that if $d_w$ is already larger than the distance to the current best candidate point for the nearest neighbor, we can safely discard $x$ as a candidate.

The tree construction involves two steps: (a) Split data recursively in half on exactly one feature. (b) Rotate through features by picking the feature with maximum variance.

The pros of the tree construction is its "exactness" and its easiness to build. The cons are (1) The curse of dimensionality makes KD-Trees ineffective for higher number of dimensions; (2) All splits are axis aligned.

`ball_tree` **method**. It uses hyper-spheres (balls) instead of the boxes in KD-tree. As in `kd_tree` method, we can dissect the distance and use the triangular inequality

$$d(x_t, x) = d_1 + d_2 \geq d_b + d_2 \geq d_b \tag{2.10}$$

If the distance to the ball, $d_b$, is larger than distance to the currently closest neighbor, we can safely ignore the ball and all points within. The ball structure allows us to partition the data along an underlying manifold that our points are on, instead of repeatedly dissecting the entire feature space (as in KD-Trees).

Ball-trees are slower than KD-Trees in low dimensions ($d \leq 3$) but a lot faster in high dimensions. Both are affected by the curse of dimensionality, but Ball-trees tend to still work if data exhibits local structure (e.g. lies on a low-dimensional manifold).

The kNN is slow during prediction because it does a lot of unecessary work. KD-trees partition the feature space so we can rule out whole partitions that are further away than our closest $k$ neighbours. However, the splits are axis aligned which does not extend well to higher dimensions. Ball-trees partition the manifold the points are on, as opposed to the whole space. This allows it to perform much better in higher dimensions.

The default algorithm in Python is `auto`, which attempts to decide the most appropriate algorithm (out of `brute`, `ball_tree`, `kd_tree`) based on the values passed to fit method.

## 2.9 More Performance Evaluation — $k$-fold cross validation (CV)

The development of CRISP-DM until now:

- Business understanding
- Data understanding
- Data preparation / preprocessing: feature scaling
- Modelling: kNN, wkNN
- Evaluation: MSE, $R^2$ for regression problems; confusion matrix / contingency table, ROC for (binary) classification problems; holdout/validation set/train-test-split method
- Deployment

**A few issues regarding the holdout/validation set method:**

- **It works mostly for time-independent data**. For a time-related data (e.g. the ISLR's Smarket data in the Practical 3 script `p03_knn1.R`), we have to split the data by earlier period and later period based on a cut-off time.
- For classification problem, **linear sampling** is allowed when the size of the data is large and the classes are **uniformly distributed**, otherwise, **stratified sampling** should be used.
- **The result depends on the sampling of the data**. Note that in Practical 3, there is a `set.seed(123)` which fixes a particular **sampling**. If we change the number to 124 or 125, we will get different results for the performance due to **different sampling**. If we apply stratified sampling with odd index, we get an accuracy of 0.8011988 and with even index, we get an accuracy of 0.7654691

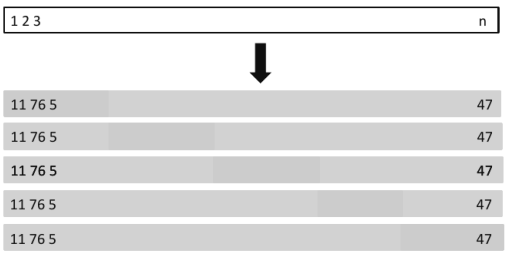
Since holdout method is "sampling" dependent, to try to learn the 'real' performance of the (kNN) model, we can cover all possible sampling. However,

> for a data with 1000 rows, if we choose 700 rows for training and 300 rows for testing using linear sampling, there are
>
> $$\frac{1000!}{700!300!} \approx 5.428250 \times 10^{263}$$
>
> possible combinations! The number is just **too large**.

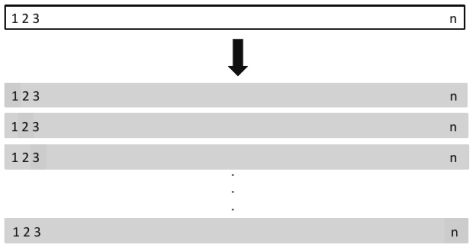
To have a better confidence on the performance than the holdout method we can use the **cross-validation methods** which are found to be acceptable by statisticians.

- Randomly divides the set of observations into $k$ "equal" "folds";
- First fold = validation set & remaining $k-1$ folds = training set.
- Second fold = validation set & remaining $k-1$ folds = training set.
- etc.

**Example 2.9.1** (5-fold)**.**




**Example 2.9.2** (LOOCV)**.**




**leave-one-out cross validation (LOOCV)** is a special case k-fold CV with $k = n$.

Software support:

R: `caret::createDataPartition`, `createResamples` (for bootstrapping), `createFolds` (for $k$-fold cross validation), `createMultiFolds` (for repeated cross-validation), `createTimeSlices` (for dealing with time-series).

Python: `train_test_split` (simple random split), `KFold` or `StratifiedKFold` ($k$-fold CV), `LeaveOneOut` (LOOCV) from `sklearn.cross_validation`.

If we don't have the right to install additional package like the caret library, we can still use R's basic commands to perform k-fold cross validation. The $k = 10$ example is shown below.

https://stats.stackexchange.com/questions/61090/how-to-split-a-data-set-to-do-10-fold-cross-validation

```
#Randomly shuffle the data & create 10 fold
d.f = d.f[sample(nrow(d.f)),]
folds = cut(seq(1,nrow(d.f)),breaks=10,labels=FALSE)
#Perform 10 fold cross validation
for(i in 1:10){
    #Segement your data by fold using which()
    testIndexes = which(folds==i,arr.ind=TRUE)
    testData  = d.f[testIndexes, ]
    trainData = d.f[-testIndexes, ]
    ...
}
```

Using `caret` library (it is a complex R packages with many dependencies):

```
set.seed(123)
train.control <- trainControl(method="cv", number=5)
model <- train(y ~., data=train.data, method="lm",
               trControl=train.control)
print(model)
```