Machine Learning Methods
for Agricultural & Food Data Management

— Part 2 —

— ("Primal") Machine Learning Algorithms —

Paul HONEINE

LITIS Lab
paul.honeine@univ-rouen.fr

— Spring 2021 —
version 2.01

Outline

Prologue

Machine Learning Methods: Outline

Outline:

Outline

Elements of Estimation Theory and Decision Theory

Elements of Estimation Theory and Decision Theory

An introductory example

Problem:

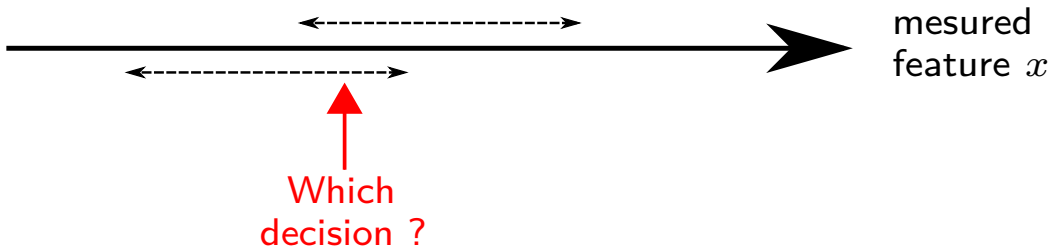
- Binary classification for a signal, or image, or patch, ...

Method:

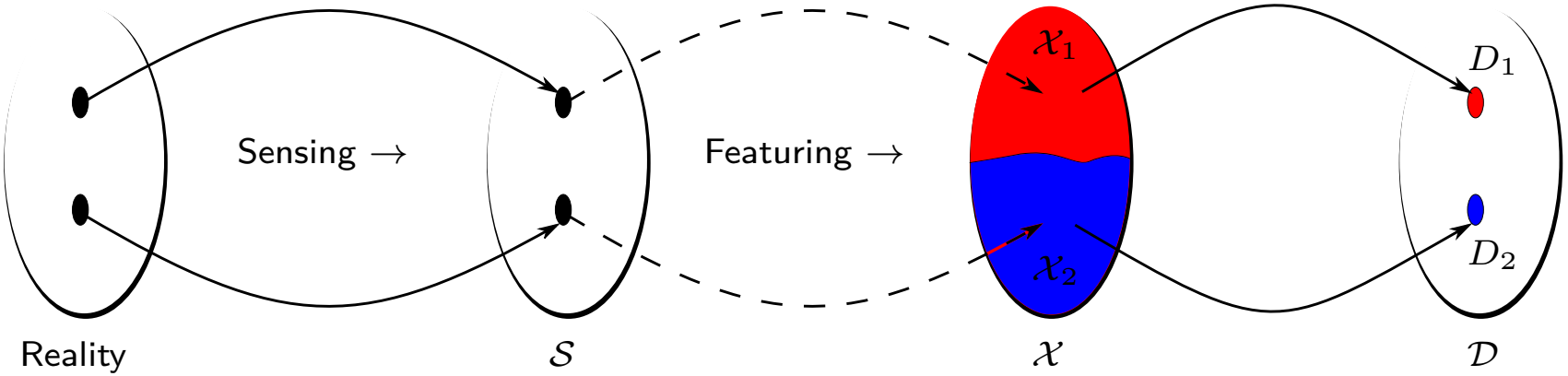
- Collect data
- Extract some parameters from the data
- Choose model/machine
- Train classifier using a set of data with label (*i.e.*, training data)
- Classify new data using the trained classifier

Difficulties:

- Since (training) data are generated from a pdf, then measurements are random variables, and so are model parameters, ...



Problem formulation



- Reality: the real thing
- $\mathcal{S}$: Raw data from sensing (signal, image, ...)
- $\mathcal{X}$: Space of features (measurements $\boldsymbol{x} \in \mathbb{R}^d$)
- $\mathcal{D}$: Decision space (D_1 versus D_2)

How to get from $\mathcal{X}$ to $\mathcal{D}$? What partition in $\mathcal{X}$?

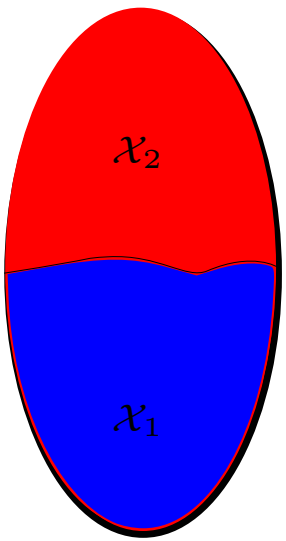
Notations:

- A priori probabilities: $P(\omega_1)$ and $P(\omega_2)$
- The vector of measurements $\boldsymbol{x}$ is a realization of a random variable
- Conditional probabilities (to measurements): $p(\boldsymbol{x} \mid \omega_1)$ and $p(\boldsymbol{x} \mid \omega_2)$
- Measurement probability: $p(\boldsymbol{x}) = p(\boldsymbol{x} \mid \omega_1)\, P(\omega_1) + p(\boldsymbol{x} \mid \omega_2)\, P(\omega_2)$

Problem formulation

From observation to decision:

Partitioning the observation space into $\mathcal{X}_1$ and $\mathcal{X}_2$, where $\mathcal{X}_1 \cup \mathcal{X}_2 = \mathcal{X}$ and $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$



Decision:

- Decision $\boldsymbol{D}_1$ if $\boldsymbol{x} \in \mathcal{X}_1$
- Decision $\boldsymbol{D}_2$ if $\boldsymbol{x} \in \mathcal{X}_2$

Probabilities on decision:

- Conditional

$$P(\boldsymbol{D}_i \mid \omega_j) = \int_{\mathcal{X}_i} p(\boldsymbol{x} \mid \omega_j)\, d\boldsymbol{x}$$

$$P(\boldsymbol{D}_1 \mid \omega_j) + P(\boldsymbol{D}_2 \mid \omega_j) = 1$$

- Total

$$P(\boldsymbol{D}_i) = \int_{\mathcal{X}_i} p(\boldsymbol{x})\, d\boldsymbol{x}$$

$$P(\boldsymbol{D}_i) = P(\boldsymbol{D}_i \mid \omega_1)P(\omega_1) + P(\boldsymbol{D}_i \mid \omega_2)P(\omega_2)$$

- Good decisions: $\omega_i \Leftrightarrow \boldsymbol{D}_i$, for $i = 1, 2$

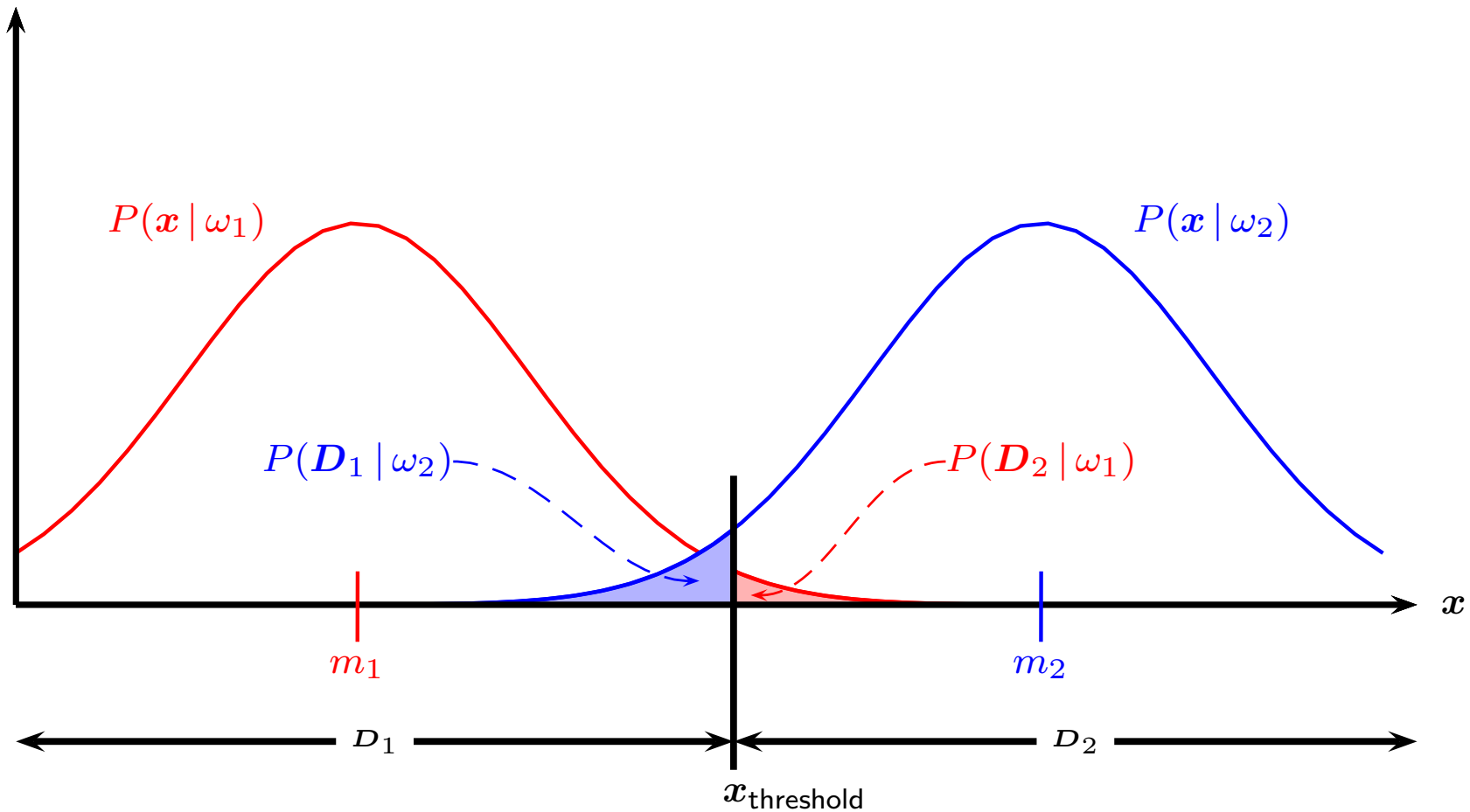
$$P_{GD} = P(\boldsymbol{D}_1 \mid \omega_1)P(\omega_1) + P(\boldsymbol{D}_2 \mid \omega_2)P(\omega_2)$$

- Bad decisions: $\omega_i \Leftrightarrow \boldsymbol{D}_j$, for $i \neq j$

$$P_{BD} = P(\boldsymbol{D}_2 \mid \omega_1)P(\omega_1) + P(\boldsymbol{D}_1 \mid \omega_2)P(\omega_2)$$

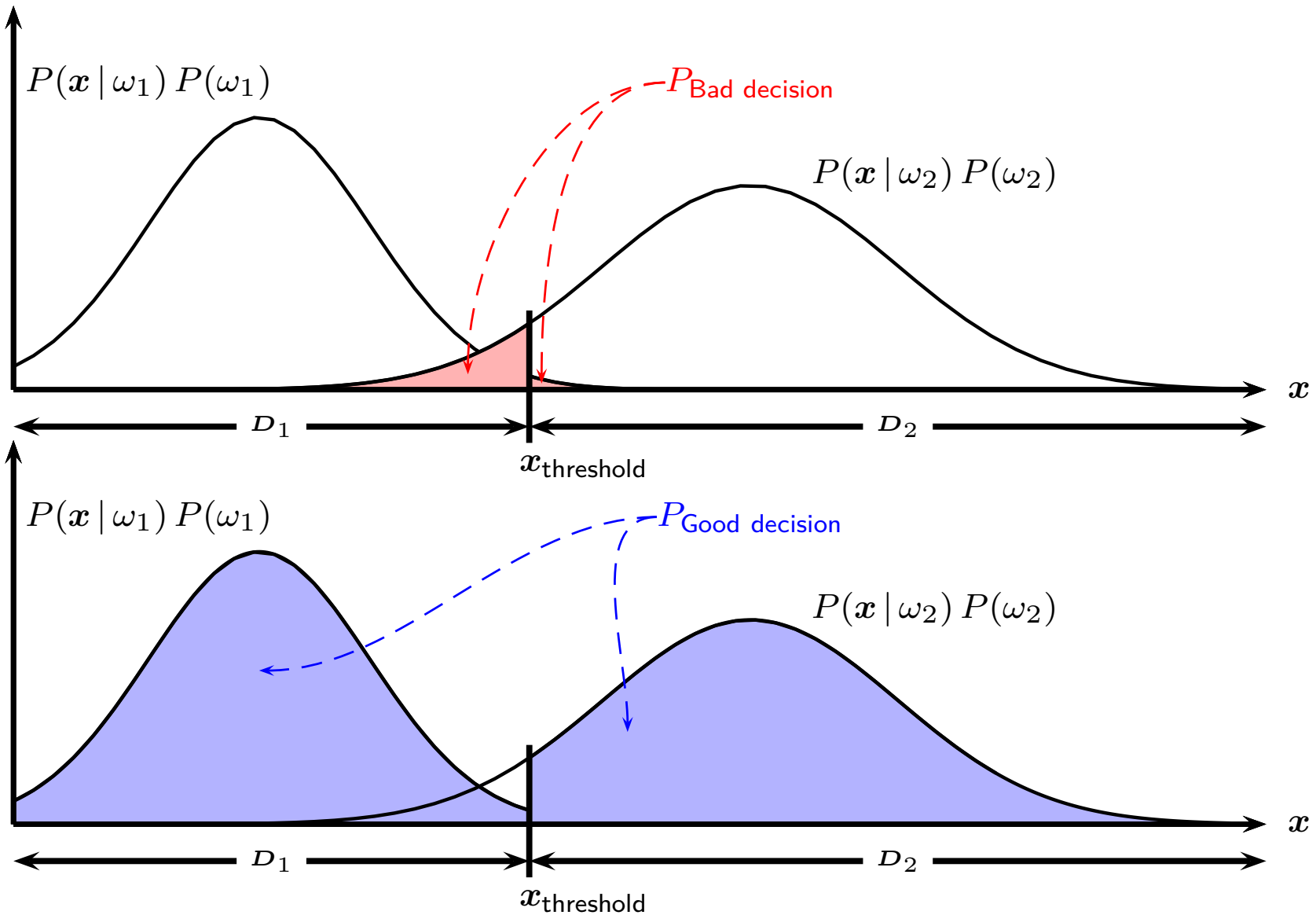
Problem formulation

Study of a decision rule



Problem formulation

Study of a decision rule



Problem formulation

Study of a decision rule

Principe:

Measurements are known, we choose the decision that has the maximum probability

- Decision $\boldsymbol{D}_1$ if $p(\omega_1 \mid \boldsymbol{x}) > p(\omega_2 \mid \boldsymbol{x})$
- Decision $\boldsymbol{D}_2$ if $p(\omega_1 \mid \boldsymbol{x}) < p(\omega_2 \mid \boldsymbol{x})$

$$\frac{p(\omega_2 \mid \boldsymbol{x})}{p(\omega_1 \mid \boldsymbol{x})} \underset{\boldsymbol{D}_1}{\overset{\boldsymbol{D}_2}{\gtrless}} 1$$

The maximum a posteriori probability estimate

$$\frac{p(\boldsymbol{x} \mid \omega_2)}{p(\boldsymbol{x} \mid \omega_1)} \underset{\boldsymbol{D}_1}{\overset{\boldsymbol{D}_2}{\gtrless}} \frac{p(\omega_1)}{p(\omega_2)}$$

This rule corresponds to comparing the likelihood ratio to a threshold:

$$\Lambda(\boldsymbol{x}) \underset{\boldsymbol{D}_1}{\overset{\boldsymbol{D}_2}{\gtrless}} \lambda$$

Bayes rule: $p(A|B) = \frac{p(B|A)p(A)}{p(B)}$

Elements of estimation theory

- In probability theory, the parameters θ are determinist.
- In statistics, the parameters are variables.
  $\hat{\theta}$: estimation of θ, *i.e.*, a random variable as a function of the r. v. samples

$$\hat{\theta} = f(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$$

Example 0: Proportions

An estimator is a random variable. Its performance can only be determined statistically

Bias

- Definition: $b(\hat{\theta}) = E(\hat{\theta}) - \theta$
- Unbiased estimator: $\hat{\theta}$ such as $E(\hat{\theta}) = \theta$
- PS: The bias is not a random variable !

Definition (Mean Square Error (MSE))

$$mse(\hat{\theta}) = E\left((\hat{\theta} - \theta)^2\right)$$

$$mse(\hat{\theta}) = E\left((\hat{\theta} - \theta)^2\right) = E\left(\left(\hat{\theta} - E(\hat{\theta}) + E(\hat{\theta}) - \theta\right)^2\right) = \ldots = var(\hat{\theta}) + b^2(\hat{\theta})$$

Outline

Nonparametric Methods:
Density Estimation and Nearest Neighbor

Nonparametric Methods

Problem formulation and solutions

Difficulty: The prior and conditional probabilities are (almost) always unknown !

Good news: We have access to some samples (*i.e.*, training data)

Two broad families of solutions:

- **The parametric way**
  - At best, there is a vague knowledge of the probability distributions
  - Objective: estimate the unknown "physical" parameters
  - Methods: Maximum likelihood, Bayesian methods

- **The nonparametric way**
  - In general, there is no knowledge available !
  - Solution: Machine Learning methods
    - Kernel density estimation (aka Parzen windows)
    - The nearest neighbor estimation (and rule with k-NN)
    - Mixture models (e.g. Gaussian mixture model or GMM)

Density Estimation and the Nearest Neighbor

Goal of density estimation

Given a set of samples, estimate the underlying probability density function (pdf)

Density estimation

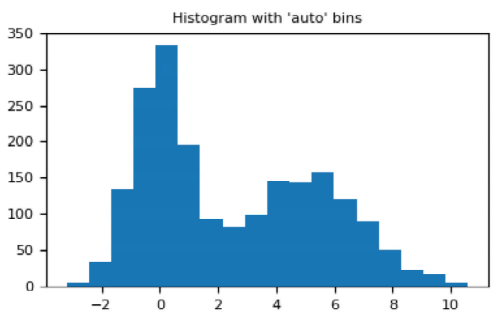
The simplest way is the **histogram**:

```
>>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
(array([0, 2, 1]), array([0, 1, 2, 3]))
>>> np.histogram(np.arange(4), bins=np.arange(5), density=True)
(array([ 0.25,  0.25,  0.25,  0.25]), array([0, 1, 2, 3, 4]))
>>> np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3])
(array([1, 4, 1]), array([0, 1, 2, 3]))
```

```
>>> a = np.arange(5)
>>> hist, bin_edges = np.histogram(a, density=True)
>>> hist
array([ 0.5,  0. ,  0.5,  0. ,  0. ,  0.5,  0. ,  0.5,  0. ,  0.5])
>>> hist.sum()
2.4999999999999996
>>> np.sum(hist * np.diff(bin_edges))
1.0
```

Automated Bin Selection Methods example (with `bins='auto'`), using a 2-peak random data:

```
>>> import matplotlib.pyplot as plt
>>> rng = np.random.RandomState(10)  # deterministic random data
>>> a = np.hstack((rng.normal(size=1000),
                   rng.normal(loc=5, scale=2, size=1000)))
>>> plt.hist(a, bins='auto')  # arguments are passed to np.histogram
>>> plt.title("Histogram with 'auto' bins")
>>> plt.show()
```



We can also do 2D and multi-D with `numpy.histogram2d` and `numpy.histogramdd`

How to choose the bin width and their number ? Proposed heuristics with expressions [here]

Density Estimation with Parzen Windows

The histogram method has many issues: discontinuities, curse of dimensionality, ...

Good news: **Kernel density estimation**

Density Estimation: The Underlying Theory Behind It

- Historically initated by Fix and Hodges (1951) in an unpublished technical report[1]
- Underlying idea: any sample from a pdf $p(\cdot)$ has the following probability that it lies in a region $\mathcal{R}$: $P = \int_{\mathcal{R}} p(\boldsymbol{x})d\boldsymbol{x}$.
- For n available samples from the same pdf, the probability that k samples lie in $\mathcal{R}$ follows a binomial distribution $\mathcal{B}(P, n)$ with $P(K = k) = C_k^n P^k (1 - P)^{n-k}$.
  (This is also the likelihood function $p(\boldsymbol{x}_1, \boldsymbol{x}_2 \ldots, \boldsymbol{x}_n|P)$, and its MLE is $\widehat{P} = k/n$.)
- The expectation of K is $E(K) = nP$, and P is usually estimated with $\widehat{P} = k/n$.
- If the volume is small enough, then we can assume that $p(\cdot)$ does not vary much inside $\mathcal{R}$, and therefore the following approximation is fairly good: $P = \int_{\mathcal{R}} p(\boldsymbol{x})d\boldsymbol{x} \approx Vp(\boldsymbol{x})$, for any sample $\boldsymbol{x} \in \mathcal{R}$.
- By substituting P with its estimate, we get $\widehat{p}(\boldsymbol{x}) = \frac{k/n}{V}$.
- Convergence is guaranteed under some mild conditions[2] on the choice of V and k.
- In practice, the estimator requires to set ahead either the volume or the value of k:
  - If the volume is set around $\boldsymbol{x}$, then the estimator $\widehat{p}(\boldsymbol{x})$ requires to counts the number of underlying samples. In classification, estimate likelihood $p(\boldsymbol{x}|\omega_j)$.
    This method is called **Parzen windows**;
  - If the number k is set, one finds the volume V around each $\boldsymbol{x}$ that contains k samples. In classification, bypass likelihood and go directly to posterior estimation $P(\omega_j|\boldsymbol{x})$.
    This is the so-called **nearest neighbor** method (estimation and rule).

[1] https://www.jstor.org/stable/1403796?seq=1

[2] Both V and k being function of n, $\lim_{n\to\infty} V = 0$, $\lim_{n\to\infty} k = \infty$, $\lim_{n\to\infty} k/n = 0$

Density Estimation with Parzen Windows

A first step

- The simplest region in a d-dimensional space is the hypercube.
- Let h be the length of an edge, then its volume is $V = h^d$.
- To determine an analytical expression k, the number of samples lying inside the hypercube, we consider the window function

$$\varphi(\boldsymbol{x}) = \begin{cases} 1 & |[\boldsymbol{x}]_j| \leq 1/2 \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } j = 1, 2, \ldots d;$$

  Therefore, $\varphi((\boldsymbol{x} - \boldsymbol{x}_i)/h)$ is equal to unity if $\boldsymbol{x}_i$ lies within the hypercube of volume V centered at $\boldsymbol{x}$, and zero otherwise.
- The number of samples in this hypercube is given by $k = \sum_{i=1}^{n} \varphi\left(\frac{\boldsymbol{x}-\boldsymbol{x}_i}{h}\right)$.
- The estimate is therefore:

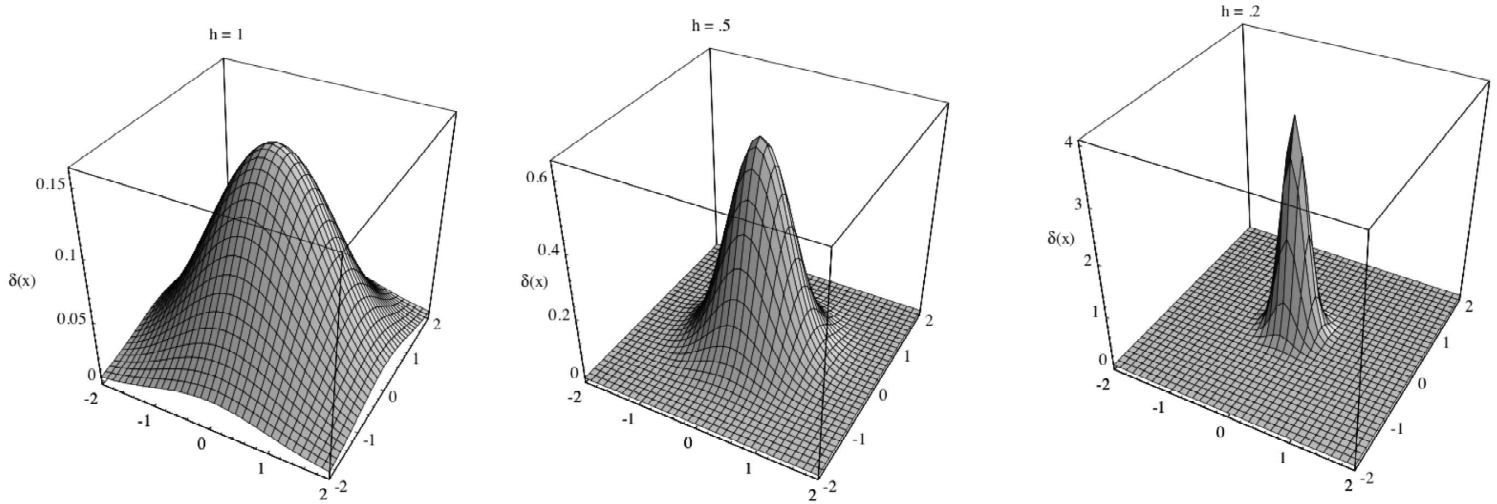
$$\widehat{p}(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{V} \varphi\left(\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h}\right).$$

- This expression suggest a more general approach to estimating density functions, without limiting ourselves to the hypercube window function.
- Only conditions on $\varphi(\cdot)$ for $\widehat{p}(\boldsymbol{x})$ to be a valid pdf: $\varphi(\boldsymbol{x}) \geq 0$ and $\int \varphi(\boldsymbol{x}) d\boldsymbol{x} = 1$
- How to choose the best window function and the best bandwidth parameter ?

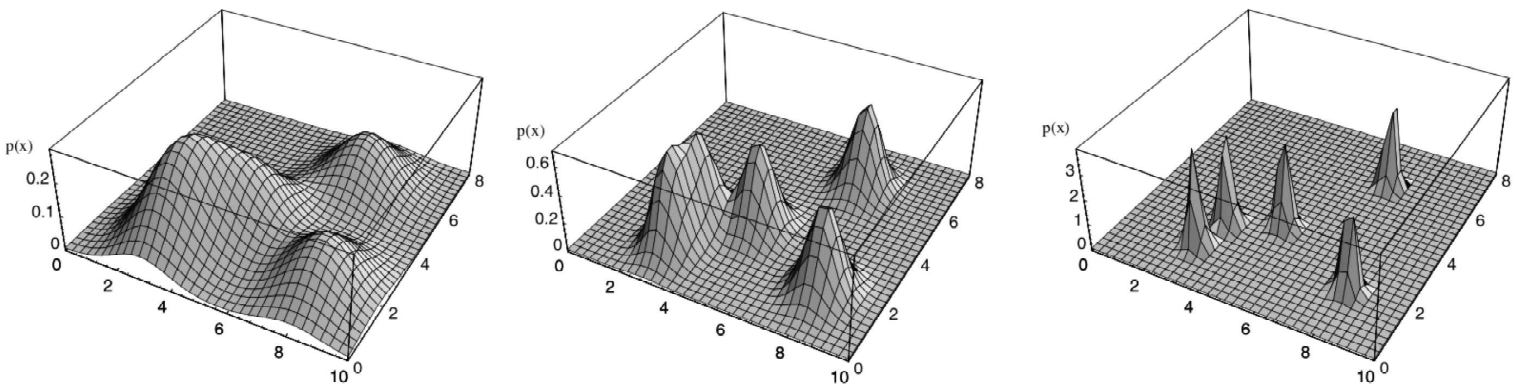
Density Estimation with Parzen Windows

Influence of the bandwidth value

A 2D circularly symmetric normalized Parzen window $\delta(\boldsymbol{x}) = \frac{1}{V}\varphi(\boldsymbol{x}/h)$ for three different values of h:



Parzen-window density estimates based on the same set of 5 samples, using the above window functions:



Density Estimation with Parzen Windows

Theoretical convergence

Convergence of the mean:

$$E(\widehat{p}(\boldsymbol{x})) = \frac{1}{n}\sum_{i=1}^{n} E\left(\frac{1}{V}\varphi\left(\frac{\boldsymbol{x}-\boldsymbol{x}_i}{h}\right)\right) = \int \frac{1}{V}\varphi\left(\frac{\boldsymbol{x}-\boldsymbol{u}}{h}\right) p(\boldsymbol{u})d\boldsymbol{u} = \int \delta(\boldsymbol{x}-\boldsymbol{u})p(\boldsymbol{u})d\boldsymbol{u} \quad (1)$$

It is a convolution of the unknown pdf and the window function. Thus, it is a blurred version of $p(\boldsymbol{x})$, as seen through the averaging window. From condition $\lim_{n\to\infty} V = 0$, $\delta(\boldsymbol{x}-\boldsymbol{u})$ approaches a delta function centered at $\boldsymbol{x}$, and therefore the expectation will approach $p(\boldsymbol{x})$ if p is continuous at $\boldsymbol{x}$.

Convergence of the variance: Because $\widehat{p}(\boldsymbol{x})$ is the sum of functions of statistically independent random variables, its variance is the sum of the variances of the separate terms:

$$\begin{aligned}
Var(\widehat{p}(\boldsymbol{x})) &= \sum_{i=1}^{n} E\left(\left(\frac{1}{nV}\varphi\left(\frac{\boldsymbol{x}-\boldsymbol{x}_i}{h} - \frac{1}{n}E(\widehat{p}(\boldsymbol{x}))\right)\right)^2\right) \\
&= nE\left(\frac{1}{n^2V^2}\varphi^2\left(\frac{\boldsymbol{x}-\boldsymbol{x}_i}{h}\right)\right) - \frac{1}{n}\left(E(\widehat{p}(\boldsymbol{x}))\right)^2 \\
&= \frac{1}{nV}\int \frac{1}{V}\varphi^2\left(\frac{\boldsymbol{x}-\boldsymbol{u}}{h}\right) p(\boldsymbol{u})d\boldsymbol{u} - \frac{1}{n}\left(E(\widehat{p}(\boldsymbol{x}))\right)^2 \\
&\leq \frac{\sup(\varphi)(\cdot)E(\widehat{p}(\boldsymbol{x}))}{nV}
\end{aligned}$$

where the upper bound follows from dropping the second term, bounding $\varphi(\cdot)$, using (1).

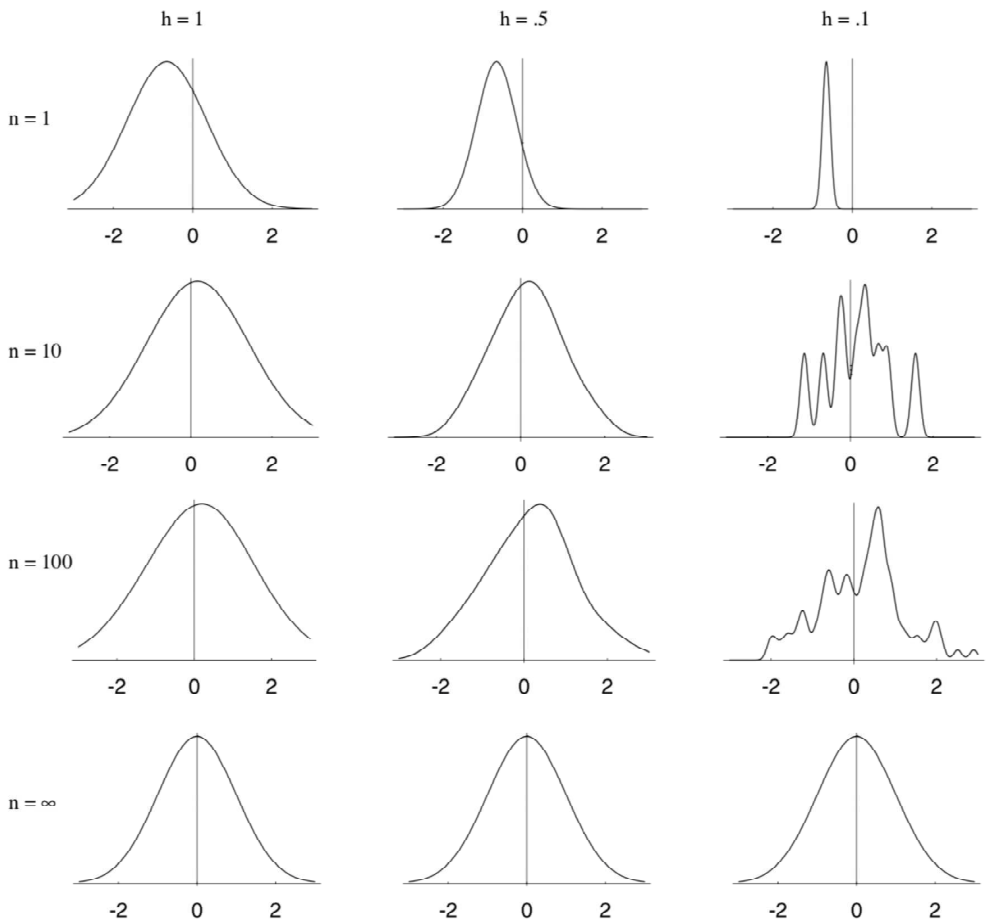
Clearly, to obtain a small variance we want a large value for V, not a small one, in order to smooth out the local variations in density. However, since the numerator stays finite as $n \to \infty$, we can let V approach zero and still obtain zero variance, provided that $nV \to \infty$. For example, we can let $V = V_1/\sqrt{n}$ or $V_1/\ln n$ or any other function satisfying $\lim_{n\to\infty} V = 0$ and $\lim_{n\to\infty} nV = \infty$.

Density Estimation with Parzen Windows

Influence of the bandwidth value and the number of samples — univariate

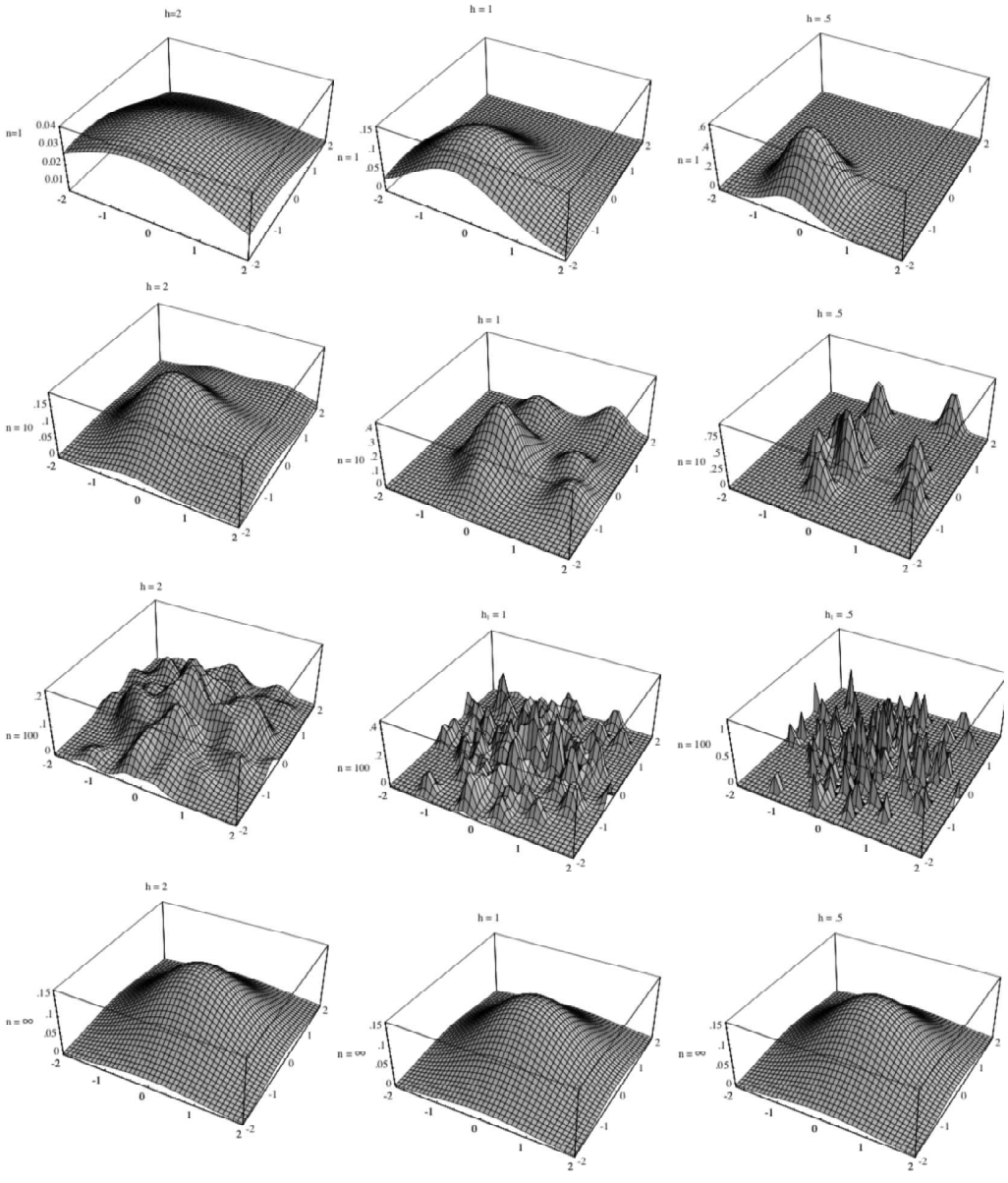
Let $p(x)$ be a zero-mean, unit-variance, univariate normal density.

Let $\varphi(u) = \frac{1}{\sqrt{2\pi}} e^{-u^2/2}$, with $h = h_1/\sqrt{n}$, then $\widehat{p}(x)$ is an average of normal densities centered at the samples: $\widehat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi h^2}} e^{-\frac{(x-x_i)^2}{2h}}$.



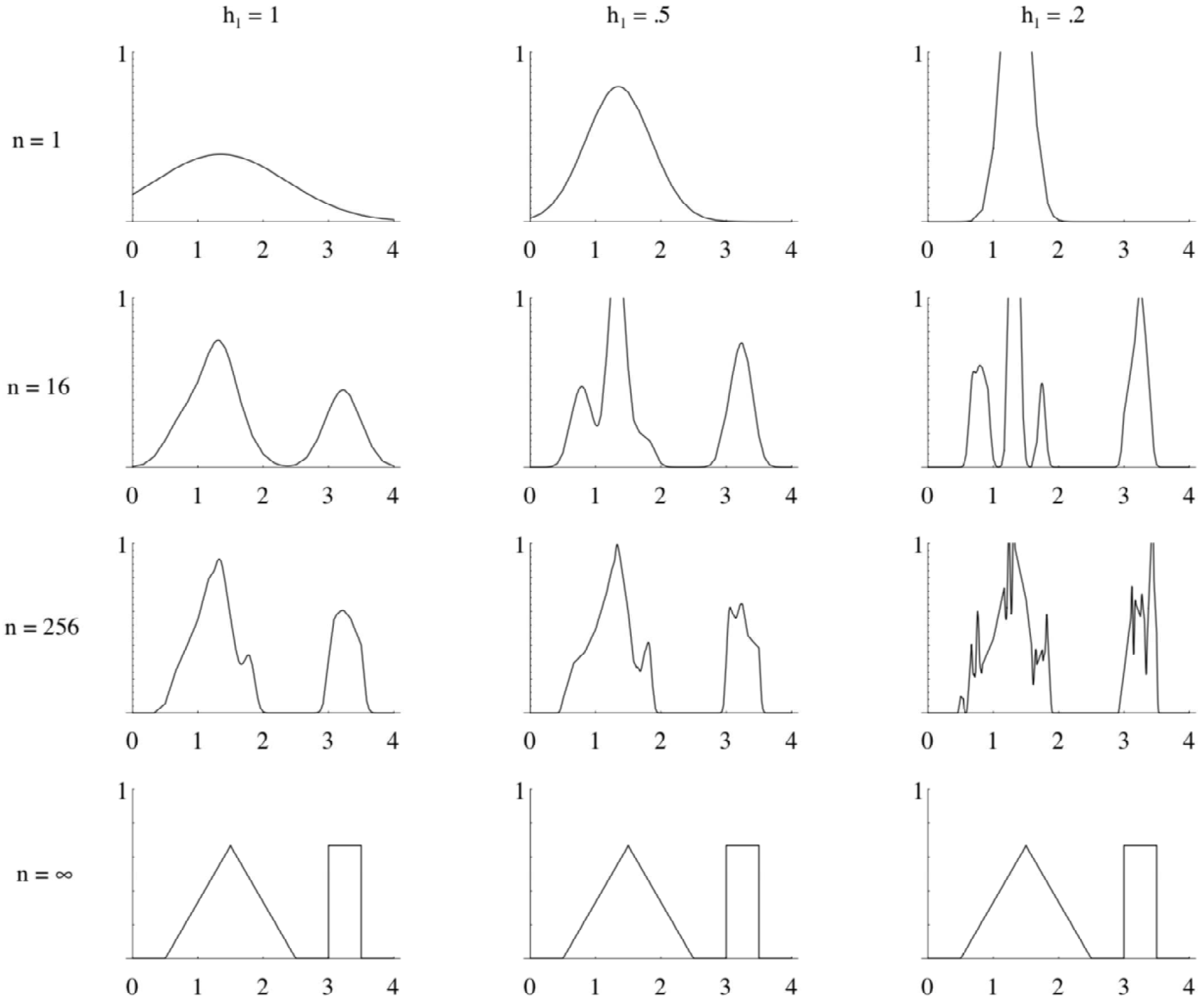
Density Estimation with Parzen Windows

Influence of the bandwidth value and the number of samples — bivariate



Density Estimation with Parzen Windows

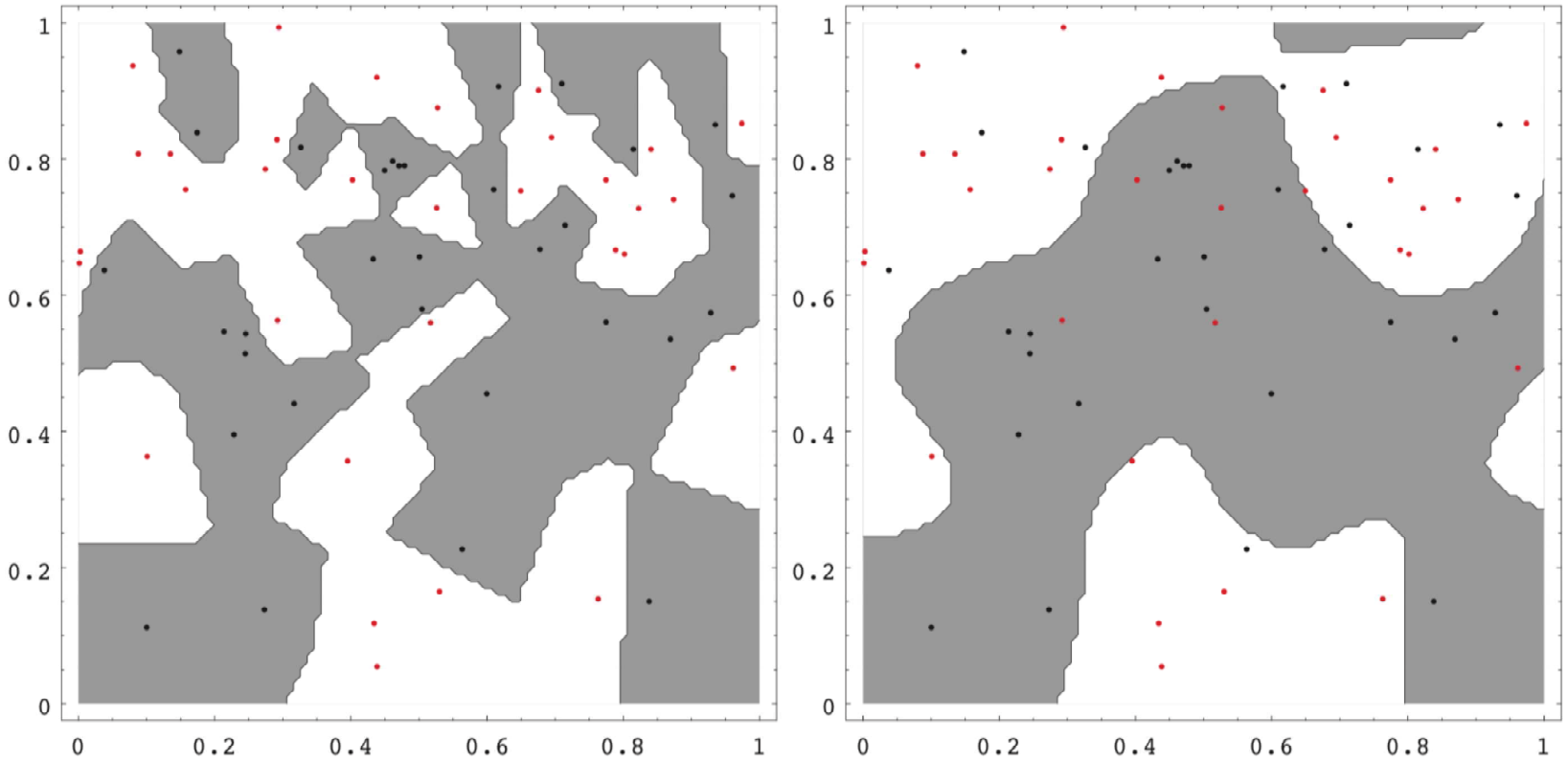
Influence of the bandwidth value and the number of samples — univariate and bimodal



Density Estimation with Parzen Windows

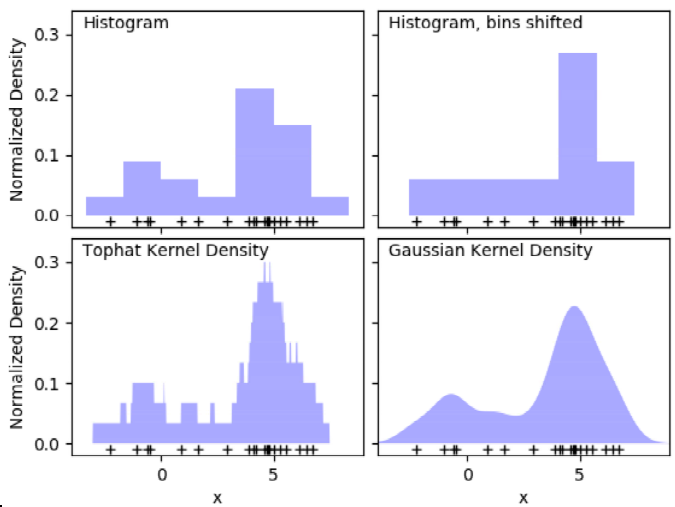
Classification example

In classifiers based on Parzen-window estimation, we estimate the densities for each category and classify a test sample by the label corresponding to the maximum posterior. The decision regions for a Parzen-window classifier depend upon the choice of window function, of course. At the left a small h leads to boundaries that are more complicated than for large h on same data set, shown at the right. Apparently, for this data a small h would be appropriate for the upper region, while a large h for the lower region; no single window width is ideal overall.



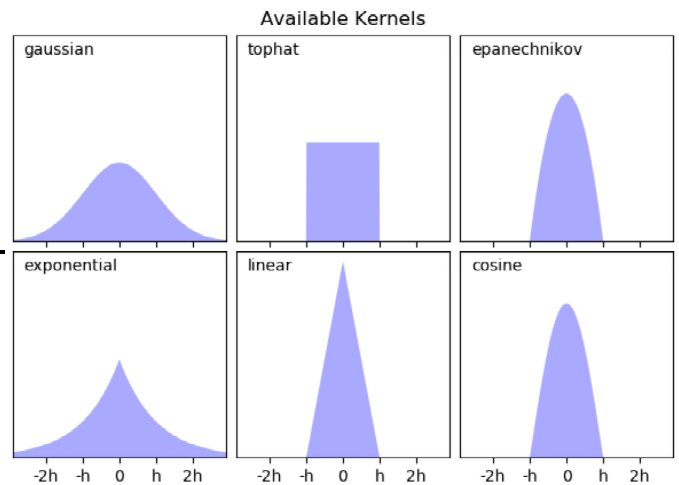
sklearn.neighbors.KernelDensity

```
# Author: Jake Vanderplas
#
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from distutils.version import LooseVersion
from scipy.stats import norm
from sklearn.neighbors import KernelDensity

# 'normed' is being deprecated in favor of 'density' in histograms
if LooseVersion(matplotlib.__version__) >= '2.1':
    density_param = {'density': True}
else:
    density_param = {'normed': True}

# Plot the progression of histograms to kernels
np.random.seed(1)
N = 20
X = np.concatenate((np.random.normal(0,1,int(0.3*N)), np.random.normal(5,1,int(0.7*N))))[:,np.newaxis]
X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]
bins = np.linspace(-5, 10, 10)

fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)
fig.subplots_adjust(hspace=0.05, wspace=0.05)

# histogram 1
ax[0, 0].hist(X[:, 0], bins=bins, fc='#AAAAFF', **density_param)
ax[0, 0].text(-3.5, 0.31, "Histogram")
# histogram 2
ax[0, 1].hist(X[:, 0], bins=bins + 0.75, fc='#AAAAFF', **density_param)
ax[0, 1].text(-3.5, 0.31, "Histogram, bins shifted")
# tophat KDE
kde = KernelDensity(kernel='tophat', bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 0].fill(X_plot[:, 0], np.exp(log_dens), fc='#AAAAFF')
ax[1, 0].text(-3.5, 0.31, "Tophat Kernel Density")
# Gaussian KDE
kde = KernelDensity(kernel='gaussian', bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 1].fill(X_plot[:, 0], np.exp(log_dens), fc='#AAAAFF')
ax[1, 1].text(-3.5, 0.31, "Gaussian Kernel Density")

for axi in ax.ravel():
    axi.plot(X[:, 0], np.full(X.shape[0], -0.01), '+k')
    axi.set_xlim(-4, 9)
    axi.set_ylim(-0.02, 0.34)

for axi in ax[:, 0]:
    axi.set_ylabel('Normalized Density')

for axi in ax[1, :]:
    axi.set_xlabel('x')
```



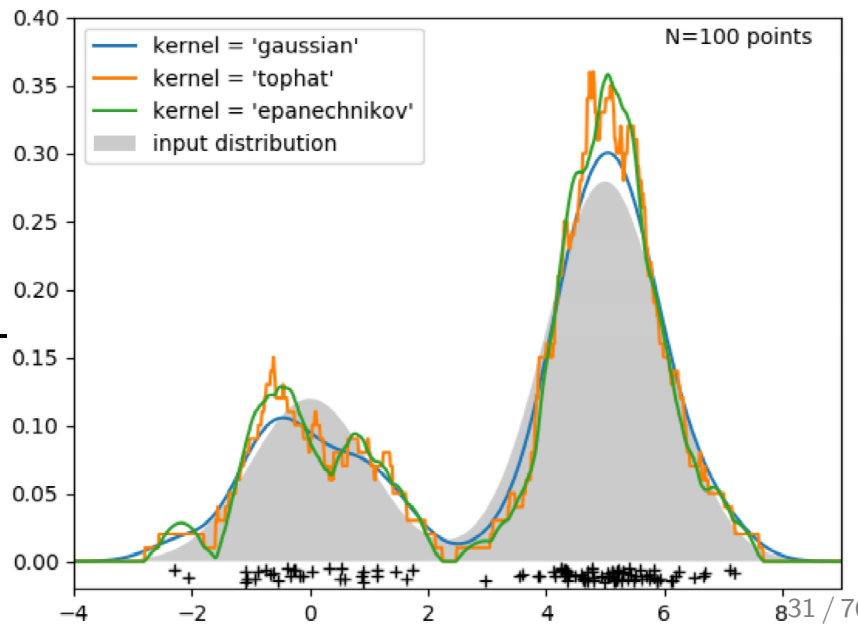
Density Estimation with Parzen Windows

```
# Plot all available kernels
X_plot = np.linspace(-6, 6, 1000)[:, None]
X_src = np.zeros((1, 1))
fig, ax = plt.subplots(2, 3, sharex=True, sharey=True)
fig.subplots_adjust(left=0.05, right=0.95, hspace=0.05, wspace=0.05)

def format_func(x, loc):
    if x == 0:
        return '0'
    elif x == 1:
        return 'h'
    elif x == -1:
        return '-h'
    else:
        return '%ih' % x

for i, kernel in enumerate(['gaussian', 'tophat', 'epanechnikov', 'exponential', 'linear', 'cosine']):
    axi = ax.ravel()[i]
    log_dens = KernelDensity(kernel=kernel).fit(X_src).score_samples(X_plot)
    axi.fill(X_plot[:, 0], np.exp(log_dens), '-k', fc='#AAAAFF')
    axi.text(-2.6, 0.95, kernel)
    axi.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
    axi.xaxis.set_major_locator(plt.MultipleLocator(1))
    axi.yaxis.set_major_locator(plt.NullLocator())
    axi.set_ylim(0, 1.05)
    axi.set_xlim(-2.9, 2.9)

ax[0, 1].set_title('Available Kernels')
```



Density Estimation with Parzen Windows

```
# Plot a 1D density example
N = 100
np.random.seed(1)
X = np.concatenate((np.random.normal(0, 1, int(0.3 * N)),
                    np.random.normal(5, 1, int(0.7 * N))))[:, np.newaxis]

X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]

true_dens = (0.3 * norm(0, 1).pdf(X_plot[:, 0]) + 0.7 * norm(5, 1).pdf(X_plot[:, 0]))

fig, ax = plt.subplots()
ax.fill(X_plot[:, 0], true_dens, fc='black', alpha=0.2, label='input distribution')

for kernel in ['gaussian', 'tophat', 'epanechnikov']:
    kde = KernelDensity(kernel=kernel, bandwidth=0.5).fit(X)
    log_dens = kde.score_samples(X_plot)
    ax.plot(X_plot[:, 0], np.exp(log_dens), '-', label="kernel = '{0}'".format(kernel))

ax.text(6, 0.38, "N={0} points".format(N))

ax.legend(loc='upper left')
ax.plot(X[:, 0],
        -0.005 - 0.01 * np.random.random(X.shape[0]),
        '+k')

ax.set_xlim(-4, 9)
ax.set_ylim(-0.02, 0.4)
plt.show()
```



Density Estimation with Parzen Windows: Conclusion

Strengths:

- It can be applied to the data from any distribution, without any assumption about the forms of the underlying densities
- In theory, it converges as the number of samples goes to infinity

Weaknesses:

- In practice, the number of available sample is limited
- Choosing the appropriate window function and window size h is difficult
- One may need a large number of samples for accurate density estimation.
- In classification, it is computationally expensive because, to classify a single sample, we have to compute a function which depends on all samples

$$\widehat{p}(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h^d} \varphi \left(\frac{\boldsymbol{x} - \boldsymbol{x}_i}{h} \right).$$

Nearest neighbor (estimation and rule)

Goal

Provide a solution for the problem of the unknown "best" window function

Solution: nearest neighbor approach

if k is set, one needs to find the volume V around $\boldsymbol{x}$ that contains k samples.

- Nearest neighbor (density) estimation
- k-nearest neighbor (decision) rule

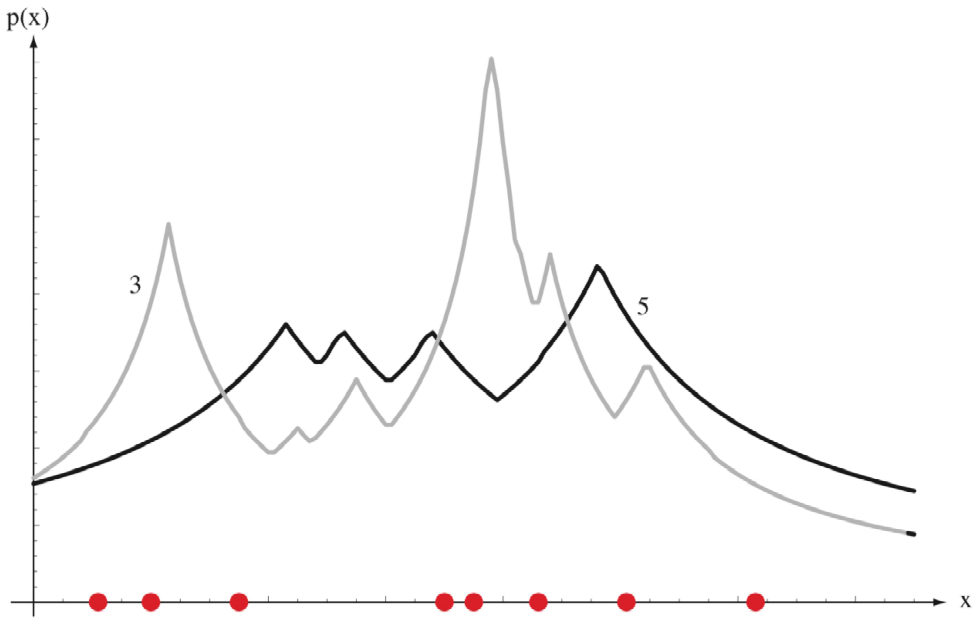
k-Nearest Neighbor Estimation

Algorithm: Set k as a function of n. Center a region about x and let it grow until it captures k samples. These samples are called the k-nearest neighbors of x.

Two possibilities can occur:

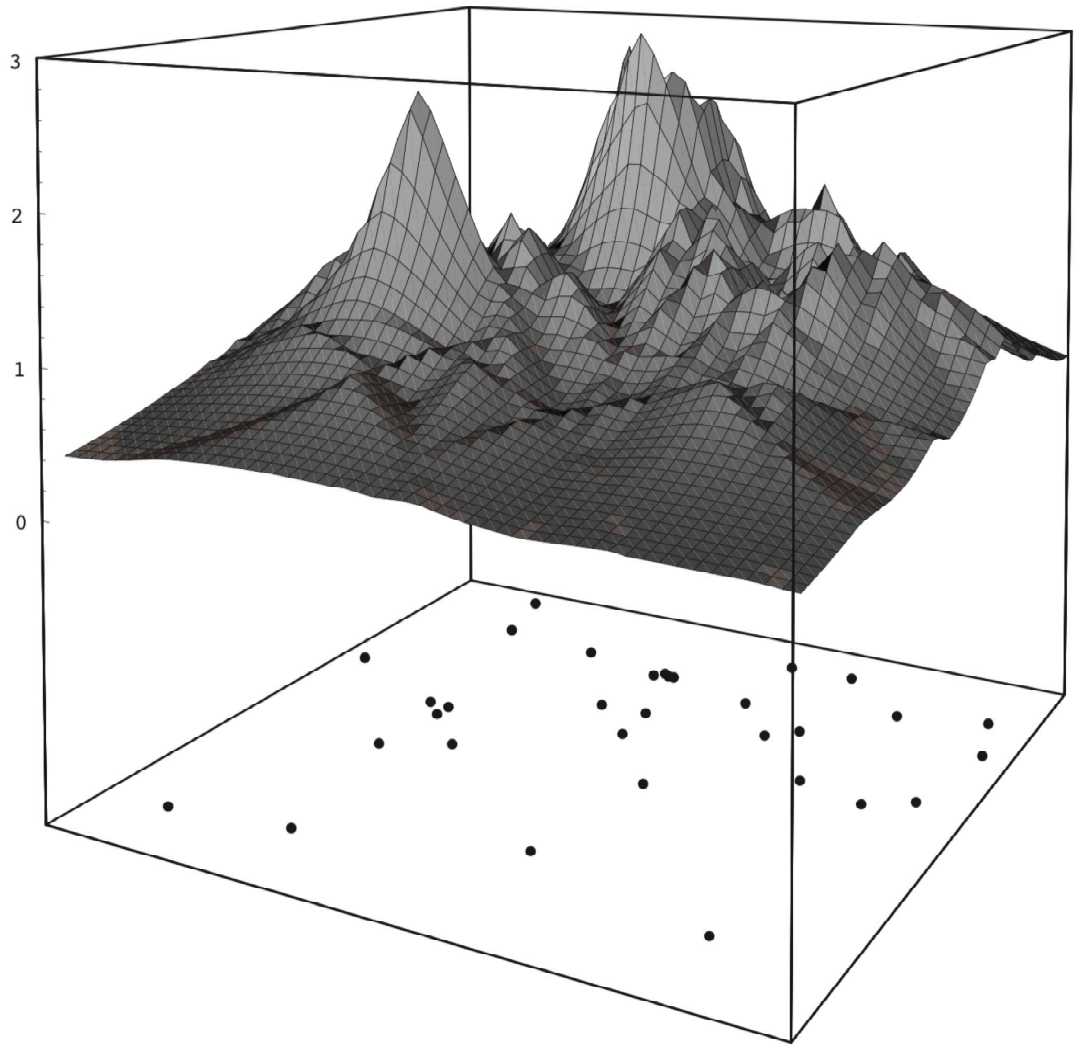
- If density is high near x, the region will be small which provides a good resolution.
- If density is low, the region will grow large and not stop until higher density regions are reached.

We can obtain a family of estimates by setting $k = k_1\sqrt{n}$ and choosing different values for k_1



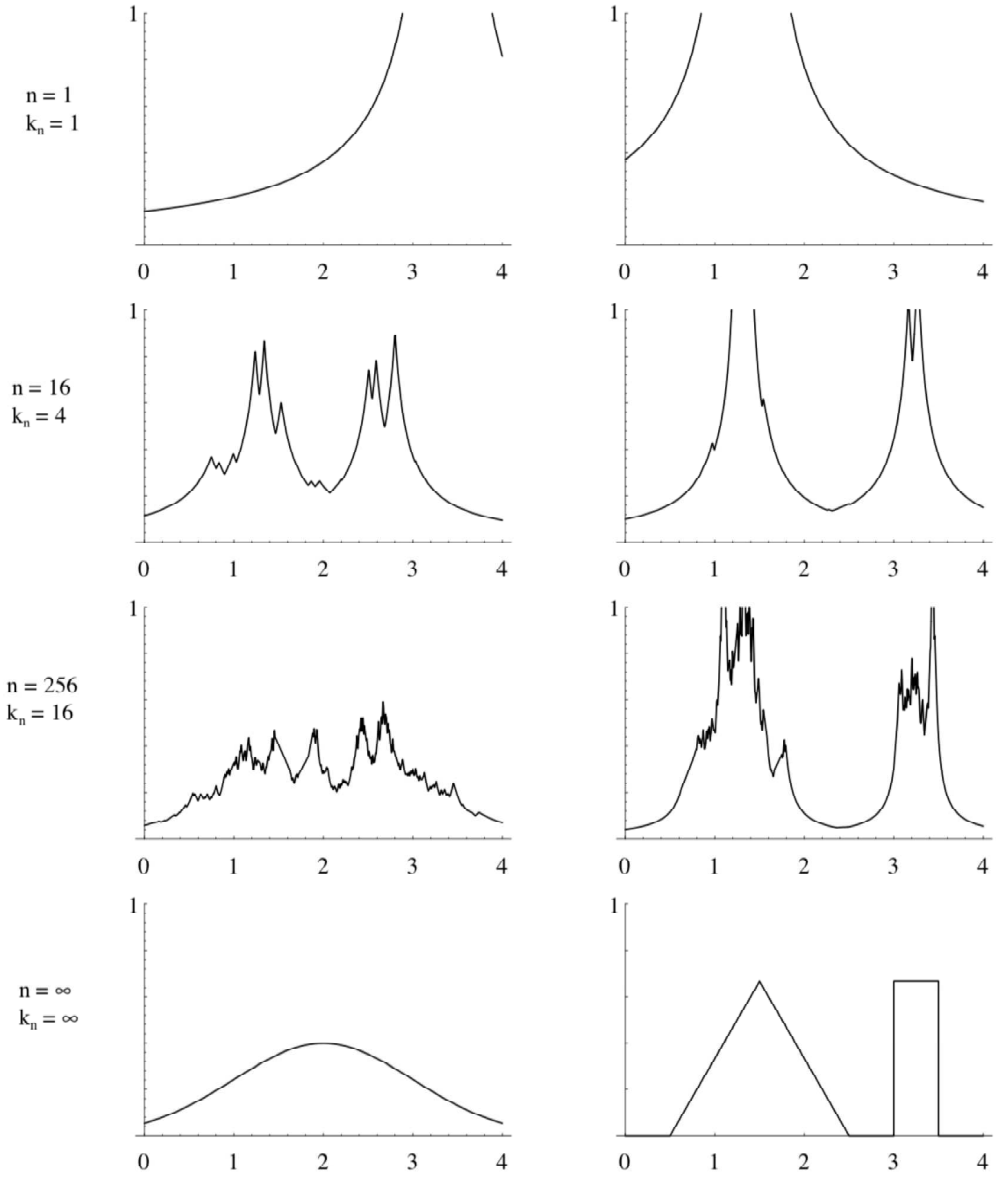
Nearest Neighbor Estimation

Example of a bivariate density



Nearest Neighbor Estimation

Influence of n and k with $k = k_1\sqrt{n}$



Nearest Neighbor Density Estimation: Summary

Strengths:

- The approach seems to be a good solution for the problem of the "best" window size
- In theory, the nearest neighbor estimate converges to the true density if infinite number of samples is available.

Weaknesses:

- In practice, the number of samples is always limited; thus, the convergence theorem is not useful.
- The resulting density estimate is not even a density, and has a lot of discontinuities (looks very spiky, not differentiable)
- Even for large regions with no observed samples, the estimated density is far from zero (tails are too heavy)

Conclusion: Thus straightforward density estimation $p(\boldsymbol{x})$ does not work very well with the nearest neighbor approach.

The k-nearest neighbor decision rule

Good news: We don't even need $p(\boldsymbol{x})$ if we can get a good estimate of $P(\omega_j|\boldsymbol{x})$. Instead of approximating the density $p(\boldsymbol{x})$, we can use the nearest neighbor method to approximate the posterior distribution $P(\omega_j|\boldsymbol{x})$.

k-Nearest Neighbor Rule

The theory behind it

- Goal: Estimate $P(\omega_j|\boldsymbol{x})$ from a set of n samples from all classes
- Recall the density estimate $\widehat{p}(\boldsymbol{x}) = \frac{k/n}{V}$
- For any sample $\boldsymbol{x}$, we consider the region of volume V containing k samples.
- Among these k samples, let k_j be the number of samples from class ω_j, then $\widehat{p}(\boldsymbol{x}, \omega_j) = \frac{k_i/n}{V}$
- The posterior probability can be estimated with

$$p(\omega_j|\boldsymbol{x}) = \frac{p(\boldsymbol{x}, \omega_j)}{p(\boldsymbol{x})} = \frac{p(\boldsymbol{x}, \omega_j)}{\sum_j p(\boldsymbol{x}, \omega_j)} \approx \frac{k_j/n}{V \sum_j \frac{k_j/n}{V}} = \frac{k_j}{\sum_j k_j} = \frac{k_j}{k}$$

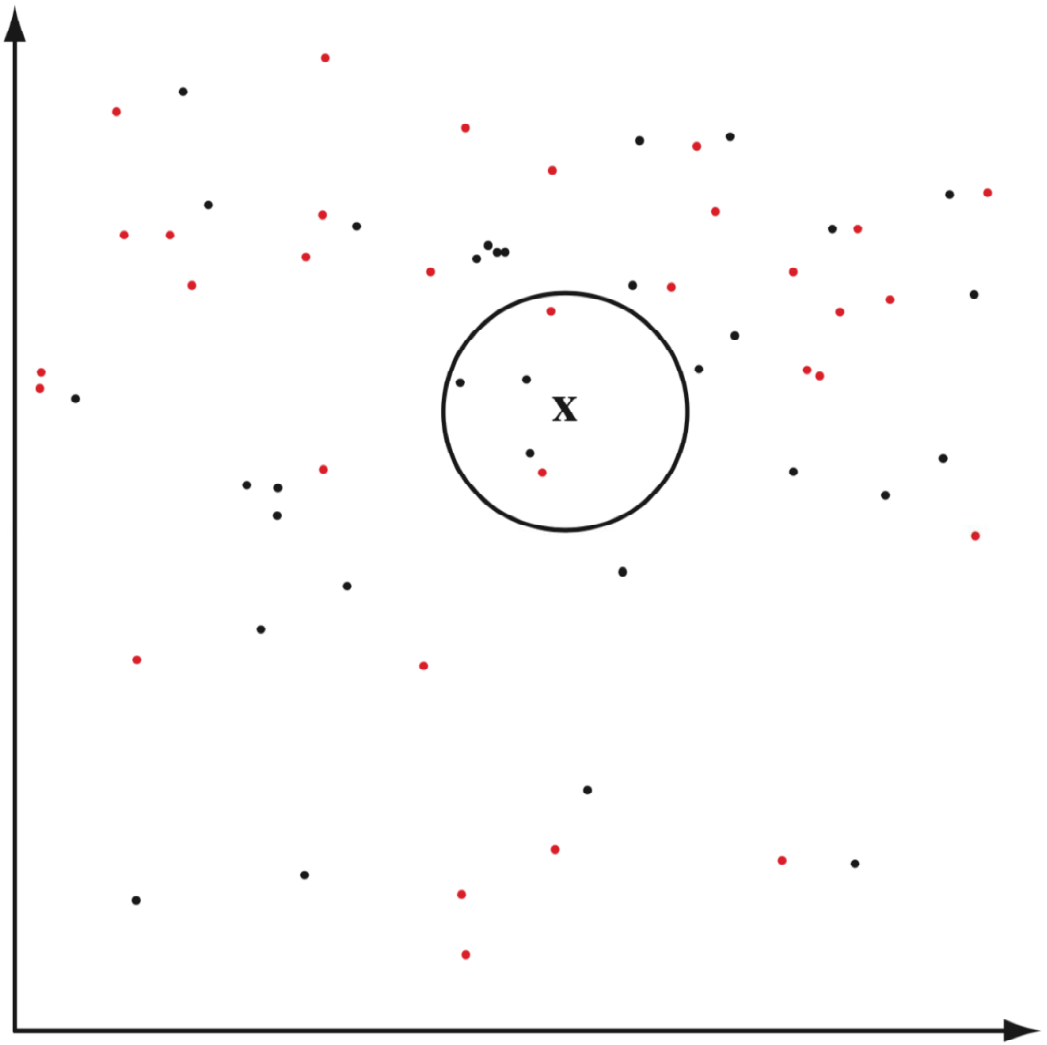
  This is a very simple and intuitive estimate.

- Under the zero-one loss function (MAP classifier), just choose the class which has the largest number of samples in its region.
- Interpretation: given an unlabeled example, find the k most similar labeled examples (*i.e.*, closest neighbors) and assign the most frequent class among them.
- Other interpretation: It is a majority vote of its k neighbors (or average of their values), where the decision function is only approximated locally.

k-Nearest Neighbor Rule

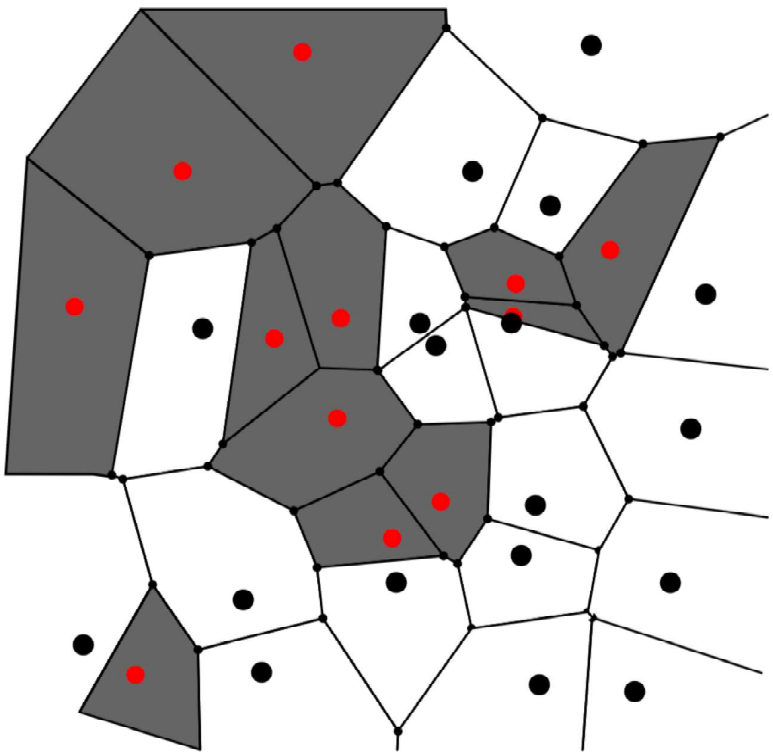
Illustration of binary classification in 2D

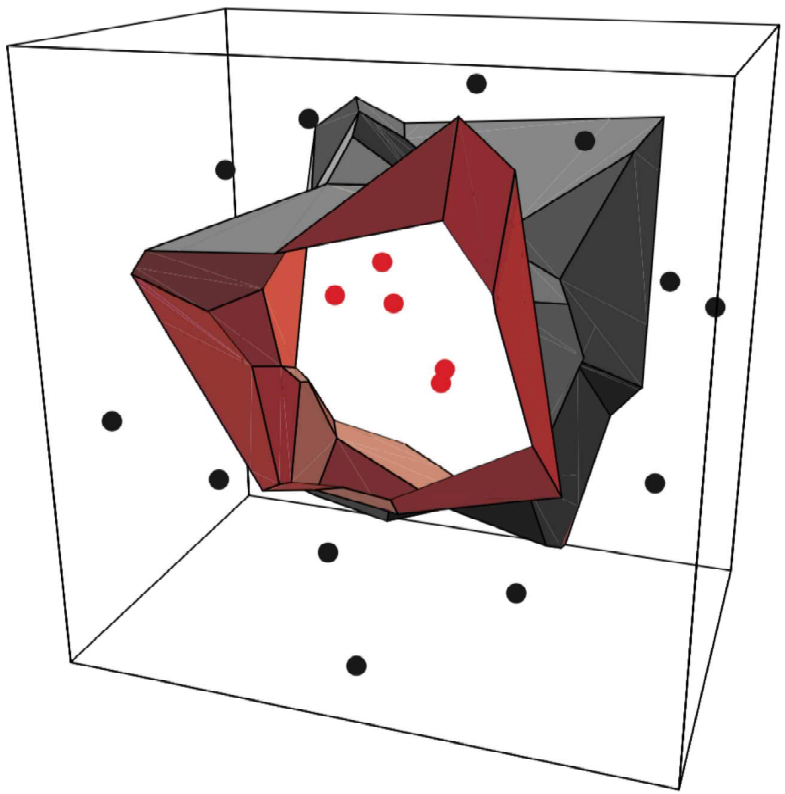
Algorithm: Starting at the test sample, the spherical region is grown until it encloses k training samples. The test sample is labelled by a majority vote of these samples.



k-Nearest Neighbor Rule

Illustration: The simplest rule is the 1-nearest neighbor rule, which leads to a split of the space into Voronoi partition, each labeled by the class of the sample it contains.





k-Nearest Neighbor Rule

Theoretical analysis

Convergence analysis: ...

k-Nearest Neighbor Rule

Choice of the parameter k

- In theory, when an infinite number of samples is available, the larger the k, the better is the classification (error rate gets closer to the optimal Bayes error rate)
- In practice,
  - k should be large to minimize the error rate, since too small values will lead to noisy decision boundaries
  - k should be small enough to include only nearby samples, since too large values will lead to over-smoothed boundaries
- Balancing the two conditions is not trivial: needs to smooth data/result, but not too much.

k-Nearest Neighbor Rule

Computational complexity

For n samples in a d-dimensional space, the algorithm stores all samples and requires

- $\mathcal{O}(d)$ to compute distance to one sample;
- $\mathcal{O}(nd)$ to find one nearest neighbor;
- $\mathcal{O}(knd)$ to find the k nearest samples.

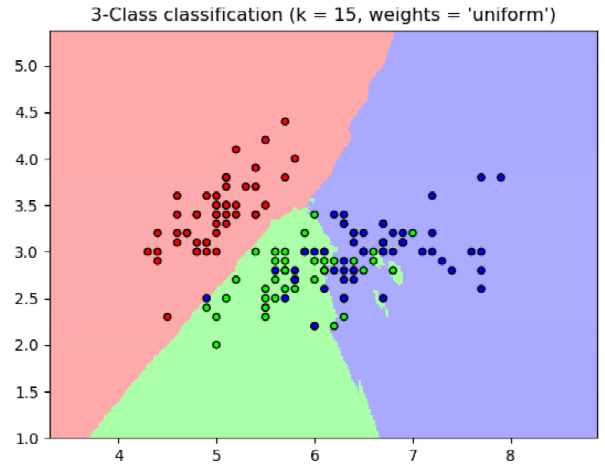
Thus, its complexity is in $\mathcal{O}(knd)$, which is prohibitively expensive since large number of samples is needed so that k-nearest neighbor rule works well.

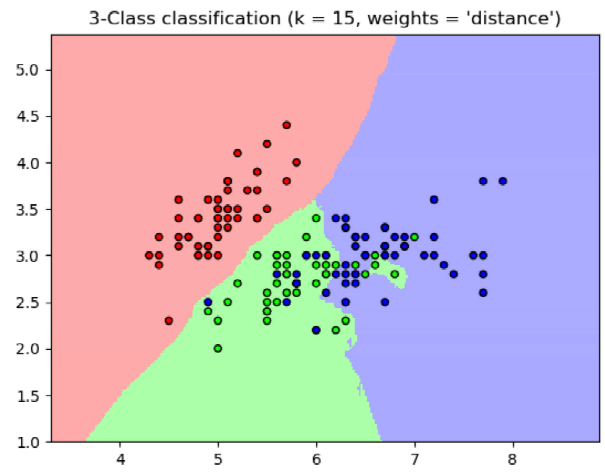
Some solutions to reduce the complexity...

k-Nearest Neighbor Rule

sklearn.neighbors.KNeighborsClassifier

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

# import some data to play with
iris = datasets.load_iris()

# we only take the first two features.
X = iris.data[:, :2]
y = iris.target

n_neighbors = 15
h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color
    #  to each point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("3-Class classification (k = %i, weights = '%s')" % (n_neighbors, weights))

plt.show()
```





k-Nearest Neighbor Rule: Summary

Strengths:

- It can be applied to the data from any distribution
- The algorithm is very simple and intuitive
- It yields good classification if the number of samples is large enough

Weaknesses:

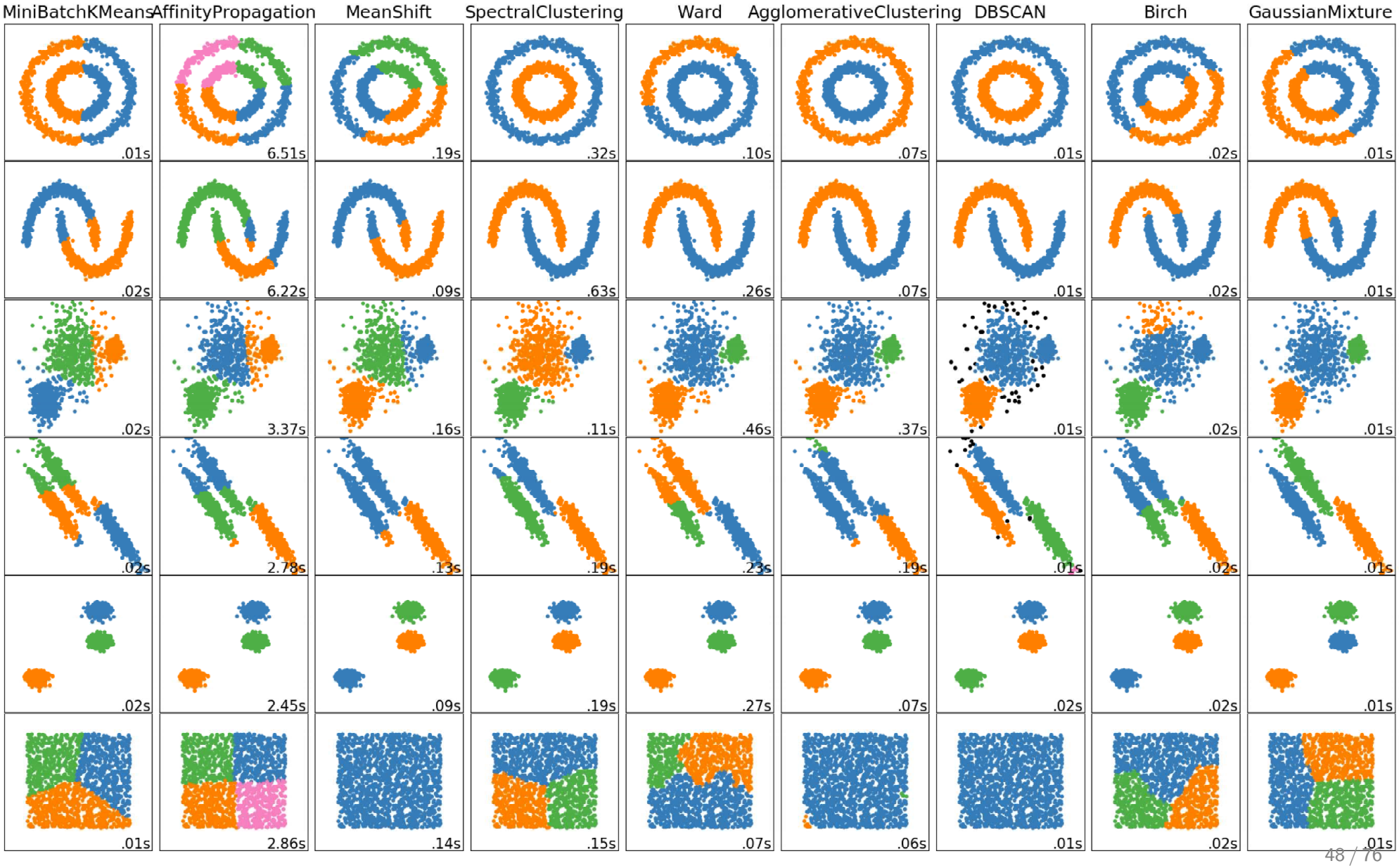
- The choice of the best k may be difficult
- It is computationally heavy, but improvements are possible
- It needs large number of samples for accuracy
  (can never fix this issue without assuming parametric distribution)

Outline

Clustering

Clustering

Algorithms



scikit-learn: Clustering Comparison

Plot clustering comparison

```
import time
import warnings
import numpy as np
import matplotlib.pyplot as plt
from sklearn import cluster, datasets, mixture
from sklearn.neighbors import kneighbors_graph
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice

np.random.seed(0)

# ============
# Generate datasets.
# We choose the size big enough to see the scalability of the
# algorithms, but not too big to avoid too long running times
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5, noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None

# Anisotropicly distributed data
random_state = 170
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

# blobs with varied variances
varied = datasets.make_blobs(n_samples=n_samples, cluster_std=[1.0, 2.5, 0.5],
                             random_state=random_state)

# ============
# Set up cluster parameters
plt.figure(figsize=(9 * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05, hspace=.01)

plot_num = 1

default_base = {'quantile': .3,
                'eps': .3,
                'damping': .9,
                'preference': -200,
                'n_neighbors': 10,
                'n_clusters': 3}

datasets = [(noisy_circles, {'damping': .77, 'preference': -240,
                             'quantile': .2, 'n_clusters': 2}),
            (noisy_moons, {'damping': .75, 'preference': -220, 'n_clusters': 2}),
            (varied, {'eps': .18, 'n_neighbors': 2}),
            (aniso, {'eps': .15, 'n_neighbors': 2}),
            (blobs, {}),
            (no_structure, {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
    # update parameters with dataset-specific values
    params = default_base.copy()
    params.update(algo_params)

    X, y = dataset

    # normalize dataset for easier parameter selection
    X = StandardScaler().fit_transform(X)

    # estimate bandwidth for mean shift
    bandwidth = cluster.estimate_bandwidth(X, quantile=params['quantile'])

    # connectivity matrix for structured Ward
    connectivity = kneighbors_graph(X, n_neighbors=params['n_neighbors'], include_self=False)
    # make connectivity symmetric
    connectivity = 0.5 * (connectivity + connectivity.T)
```

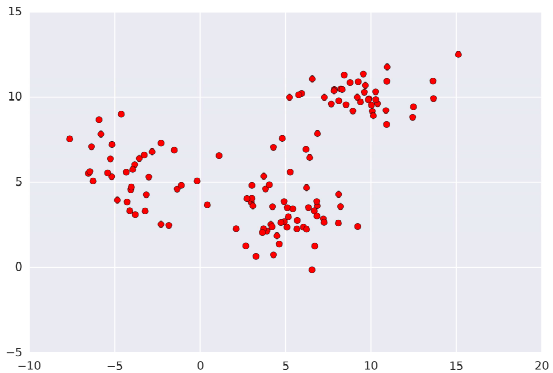
```
    # ============
    # Create cluster objects
    ms = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
    two_means = cluster.MiniBatchKMeans(n_clusters=params['n_clusters'])
    ward = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='ward',
        connectivity=connectivity)
    spectral = cluster.SpectralClustering(
        n_clusters=params['n_clusters'], eigen_solver='arpack',
        affinity="nearest_neighbors")
    dbscan = cluster.DBSCAN(eps=params['eps'])
    affinity_propagation = cluster.AffinityPropagation(
        damping=params['damping'], preference=params['preference'])
    average_linkage = cluster.AgglomerativeClustering(
        linkage="average", affinity="cityblock",
        n_clusters=params['n_clusters'], connectivity=connectivity)
    birch = cluster.Birch(n_clusters=params['n_clusters'])
    gmm = mixture.GaussianMixture(
        n_components=params['n_clusters'], covariance_type='full')

    clustering_algorithms = (('MiniBatchKMeans', two_means),
                             ('AffinityPropagation', affinity_propagation),
                             ('MeanShift', ms),
                             ('SpectralClustering', spectral),
                             ('Ward', ward),
                             ('AgglomerativeClustering', average_linkage),
                             ('DBSCAN', dbscan),
                             ('Birch', birch),
                             ('GaussianMixture', gmm))
    for name, algorithm in clustering_algorithms:
        t0 = time.time()
        # catch warnings related to kneighbors_graph
        with warnings.catch_warnings():
            warnings.filterwarnings(
                "ignore",
                message="the number of connected components of the " +
                "connectivity matrix is [0-9]{1,2}" +
                " > 1. Completing it to avoid stopping the tree early.",
                category=UserWarning)
            warnings.filterwarnings(
                "ignore",
                message="Graph is not fully connected, spectral embedding" +
                " may not work as expected.",
                category=UserWarning)
            algorithm.fit(X)

        t1 = time.time()
        if hasattr(algorithm, 'labels_'):
            y_pred = algorithm.labels_.astype(np.int)
        else:
            y_pred = algorithm.predict(X)

        plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                             '#f781bf', '#a65628', '#984ea3',
                                             '#999999', '#e41a1c', '#dede00']),
                                      int(max(y_pred) + 1))))
        colors = np.append(colors, ["#000000"]) # add black color for outliers (if any)
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])

        plt.xlim(-2.5, 2.5)
        plt.ylim(-2.5, 2.5)
        plt.xticks(())
        plt.yticks(())a
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),transform=plt.gca().transAxes,
                 size=15, horizontalalignment='right')
        plot_num += 1
plt.show()
```

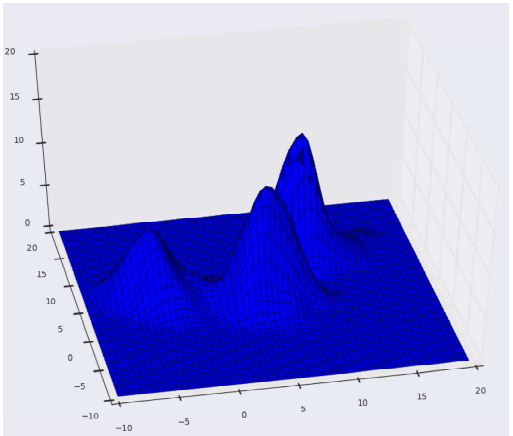
The Mean-shift Algorithm

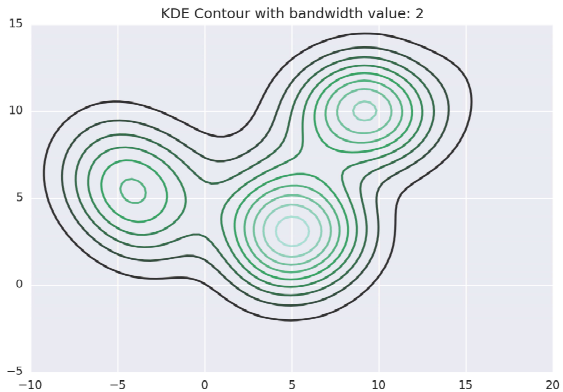
The Mean-shift Algorithm

The theory behind it

From Parzen density estimate:







... to clustering:

- Goal: Find the modes of the estimated density function
- Resolution: Nullify the gradient of the estimated density function to get $\boldsymbol{x}^*$
- Algorithm: We obtain the mean-shift algorithm as a fixed-point iteration

$$\boldsymbol{x}_{t+1}^* = \frac{\sum_{i=1}^n \kappa(\boldsymbol{x}_t^*, \boldsymbol{x}_i)\ \boldsymbol{x}_i}{\sum_{i=1}^n \kappa(\boldsymbol{x}_t^*, \boldsymbol{x}_i)},$$

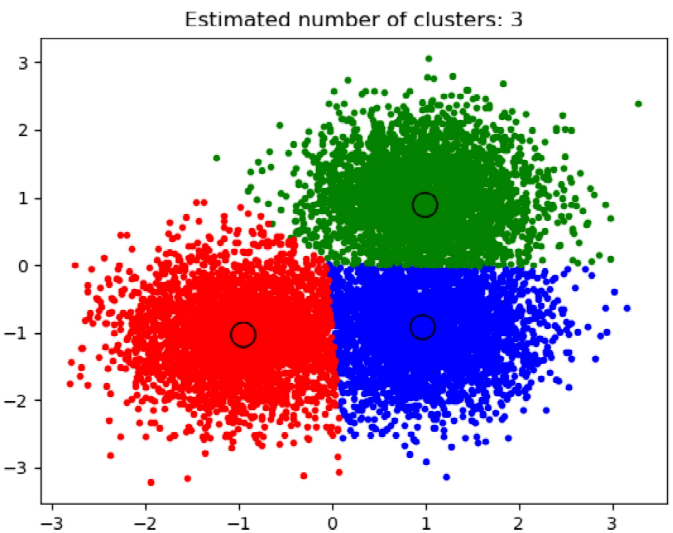
with $\kappa(\boldsymbol{x}_t^*, \boldsymbol{x}_i) = \exp(-\|\boldsymbol{x}_t^* - \boldsymbol{x}_i\|^2/2\sigma^2)$ for the Gaussian kernel.

The Mean-shift Algorithm

First example

Plot Mean-shift

```
import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.datasets.samples_generator import make_blobs

# #############################################################
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, _ = make_blobs(n_samples=10000, centers=centers, cluster_std=0.6)

# #############################################################
# Compute clustering with MeanShift

# The following bandwidth can be automatically detected using
bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)

ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)

print("number of estimated clusters : %d" % n_clusters_)

# #############################################################
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```



The Mean-shift Algorithm

Second example (with more details and illustrations)

Mean-shift-clustering

MeanShift_py on GitHub

The Mean-shift Algorithm: Summary

Strengths:

- The mean-shift is an application-independent tool suitable for real data analysis.
- It does not assume any predefined shape on data clusters.
- It is capable of handling arbitrary feature spaces.
- The procedure relies on choice of a single parameter: bandwidth.
- The number of clusters is not required
- The bandwidth/window size has a physical meaning, unlike k-means.

Weaknesses:

- The selection of a window size is not trivial.
- An inappropriate window size can cause modes to be merged, or generate additional "shallow" modes.
- It often requires using adaptive window size.

k-means Clustering

k-means Clustering

The theory behind it

- Goal: Partition the samples into k clusters by minimizing the within-variance
- k-means clustering: Given a set of (unlabeled) samples $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$, it aims to partition them into k sets (aka "clusters") $\boldsymbol{S} = \{S_1, S_2, \ldots, S_k\}$ by minimizing the within-cluster sum of squares (*i.e.*, variance), namely

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$$

  where $\boldsymbol{\mu}_i$ is the mean of points in S_i.
- Equivalent formulation/interpretation: minimizing the pairwise squared deviations of points in the same cluster

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\boldsymbol{x}, \boldsymbol{z} \in S_i} \|\boldsymbol{x} - \boldsymbol{z}\|^2 \arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\boldsymbol{x}, \boldsymbol{z} \in S_i} \|\boldsymbol{x} - \boldsymbol{z}\|^2,$$

  the equivalence is due to

$$\sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2 = \sum_{\boldsymbol{x} \neq \boldsymbol{z} \in S_i} (\boldsymbol{x} - \boldsymbol{\mu}_i)(\boldsymbol{\mu}_i - \boldsymbol{z}) \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2 = \sum_{\boldsymbol{x} \neq \boldsymbol{z} \in S_i} (\boldsymbol{x} - \boldsymbol{\mu}_i)(\boldsymbol{\mu}_i - \boldsymbol{z})$$

- Equivalent formulation/interpretation: Because the total variance is constant, this is equivalent to maximizing the sum of squared deviations between samples in different clusters (between-cluster sum of squares criterion), which follows from the law of total variance.

k-means Clustering

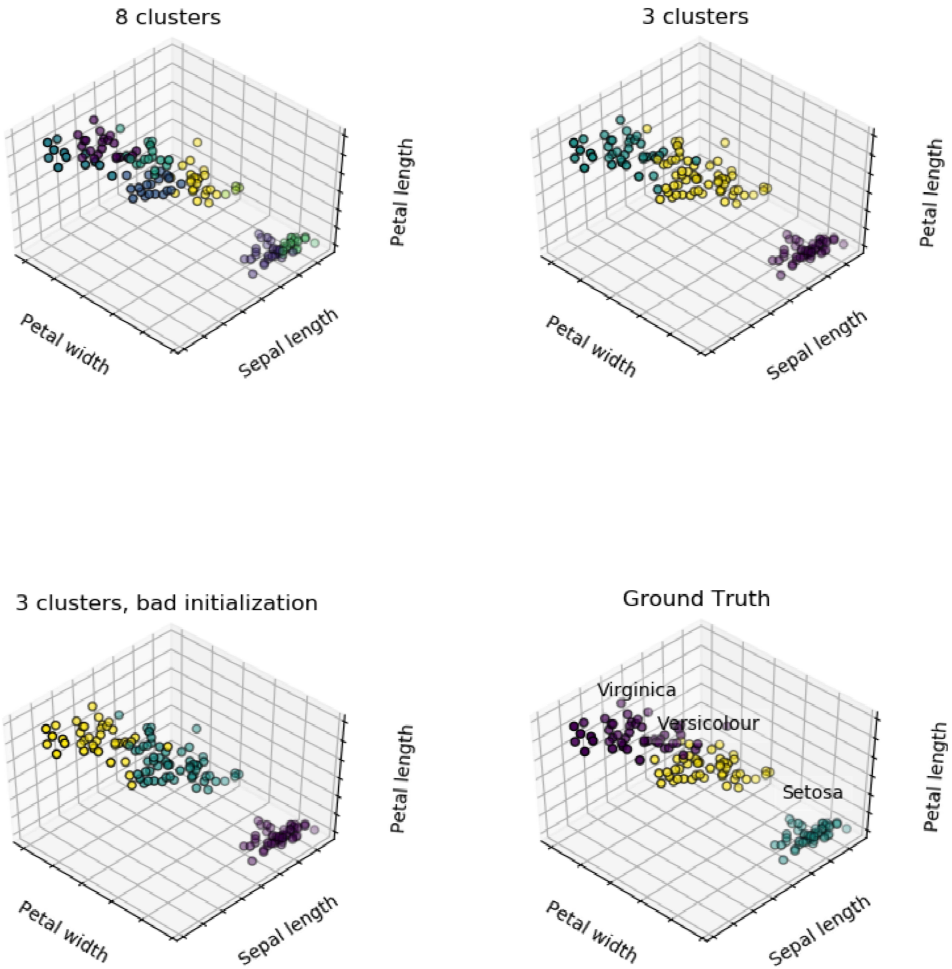
Illustration

K-means_convergence (wiki)

k-means Clustering

Plot cluster iris

```
# Code source: Gael Varoquaux; modified for documentation by Jaques Grobler; License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
# Though the following import is not directly being used, it is required for 3D projection to work
from mpl_toolkits.mplot3d import Axes3D
from sklearn.cluster import KMeans
from sklearn import datasets

np.random.seed(5)
iris = datasets.load_iris()
X = iris.data
y = iris.target

estimators = [('k_means_iris_8', KMeans(n_clusters=8)),
              ('k_means_iris_3', KMeans(n_clusters=3)),
              ('k_means_iris_bad_init', KMeans(n_clusters=3,
               n_init=1, init='random'))]
fignum = 1
titles = ['8 clusters', '3 clusters', '3 clusters, bad initialization']
for name, est in estimators:
    fig = plt.figure(fignum, figsize=(4, 3))
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
    est.fit(X)
    labels = est.labels_

    ax.scatter(X[:, 3], X[:, 0], X[:, 2], c=labels.astype(np.float), edgecolor='k')
    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])
    ax.set_xlabel('Petal width')
    ax.set_ylabel('Sepal length')
    ax.set_zlabel('Petal length')
    ax.set_title(titles[fignum - 1])
    ax.dist = 12
    fignum = fignum + 1
# Plot the ground truth
fig = plt.figure(fignum, figsize=(4, 3))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

for name, label in [('Setosa', 0),
                    ('Versicolour', 1),
                    ('Virginica', 2)]:
    ax.text3D(X[y == label, 3].mean(),
              X[y == label, 0].mean(),
              X[y == label, 2].mean() + 2, name,
              horizontalalignment='center',
              bbox=dict(alpha=.2, edgecolor='w', facecolor='w'))
# Reorder the labels to have colors matching the cluster results
y = np.choose(y, [1, 2, 0]).astype(np.float)
ax.scatter(X[:, 3], X[:, 0], X[:, 2], c=y, edgecolor='k')

ax.w_xaxis.set_ticklabels([])
ax.w_yaxis.set_ticklabels([])
ax.w_zaxis.set_ticklabels([])
ax.set_xlabel('Petal width')
ax.set_ylabel('Sepal length')
ax.set_zlabel('Petal length')
ax.set_title('Ground Truth')
ax.dist = 12

fig.show()
```



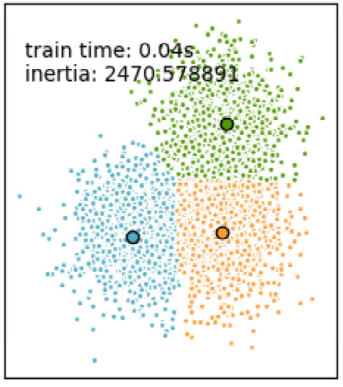
k-means Clustering

Comparison of the k-means and MiniBatchKMeans clustering algorithms

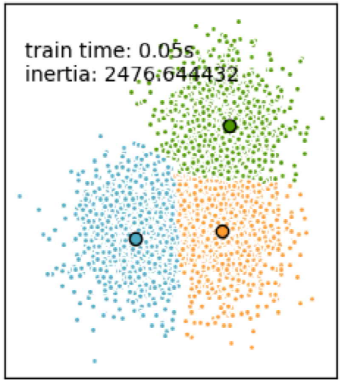
Plot mini batch k-means

```
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MiniBatchKMeans, KMeans
from sklearn.metrics.pairwise import pairwise_distances_argmin
from sklearn.datasets.samples_generator import make_blobs

# #############################################################################
# Generate sample data
np.random.seed(0)
batch_size = 45
centers = [[1, 1], [-1, -1], [1, -1]]
n_clusters = len(centers)
X, labels_true = make_blobs(n_samples=3000, centers=centers, cluster_std=0.7)

# #############################################################################
# Compute clustering with Means
k_means = KMeans(init='k-means++', n_clusters=3, n_init=10)
t0 = time.time()
k_means.fit(X)
t_batch = time.time() - t0

# #############################################################################
# Compute clustering with MiniBatchKMeans
mbk = MiniBatchKMeans(init='k-means++', n_clusters=3, batch_size=batch_size,
                      n_init=10, max_no_improvement=10, verbose=0)
t0 = time.time()
mbk.fit(X)
t_mini_batch = time.time() - t0

# #############################################################################
# Plot result
fig = plt.figure(figsize=(8, 3))
fig.subplots_adjust(left=0.02, right=0.98, bottom=0.05, top=0.9)
colors = ['#4EACC5', '#FF9C34', '#4E9A06']

# We want to have the same colors for the same cluster from the MiniBatchKMeans and the KMeans algorithm.
# Let's pair the cluster centers per closest one.
k_means_cluster_centers = np.sort(k_means.cluster_centers_, axis=0)
mbk_means_cluster_centers = np.sort(mbk.cluster_centers_, axis=0)
k_means_labels = pairwise_distances_argmin(X, k_means_cluster_centers)
mbk_means_labels = pairwise_distances_argmin(X, mbk_means_cluster_centers)
order = pairwise_distances_argmin(k_means_cluster_centers, mbk_means_cluster_centers)

# KMeans
ax = fig.add_subplot(1, 3, 1)
for k, col in zip(range(n_clusters), colors):
    my_members = k_means_labels == k
    cluster_center = k_means_cluster_centers[k]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w', markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('KMeans')
ax.set_xticks(())
ax.set_yticks(())
plt.text(-3.5, 1.8,  'train time: %.2fs\ninertia: %f' % (t_batch, k_means.inertia_))

# MiniBatchKMeans
ax = fig.add_subplot(1, 3, 2)
for k, col in zip(range(n_clusters), colors):
    my_members = mbk_means_labels == order[k]
    cluster_center = mbk_means_cluster_centers[order[k]]
    ax.plot(X[my_members, 0], X[my_members, 1], 'w', markerfacecolor=col, marker='.')
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('MiniBatchKMeans')
ax.set_xticks(())
ax.set_yticks(())
plt.text(-3.5, 1.8, 'train time: %.2fs\ninertia: %f' %(t_mini_batch, mbk.inertia_))

# Initialise the different array to all False
different = (mbk_means_labels == 4)
ax = fig.add_subplot(1, 3, 3)

for k in range(n_clusters):
    different += ((k_means_labels == k) != (mbk_means_labels == order[k]))

identic = np.logical_not(different)
ax.plot(X[identic, 0], X[identic, 1], 'w', markerfacecolor='#bbbbbb', marker='.')
ax.plot(X[different, 0], X[different, 1], 'w', markerfacecolor='m', marker='.')
ax.set_title('Difference')
ax.set_xticks(())
ax.set_yticks(())

plt.show()
```

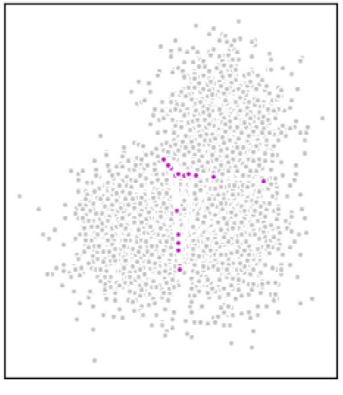
KMeans



MiniBatchKMeans



Difference



k-means Clustering: Summary

Strengths:

- The algorithm is easy to implement
- With a large number of variables, it may be computationally faster than hierarchical clustering (if k is small).
- It may produce tighter clusters than hierarchical clustering
- A sample can change cluster when the centroids are recomputed.

Weaknesses:

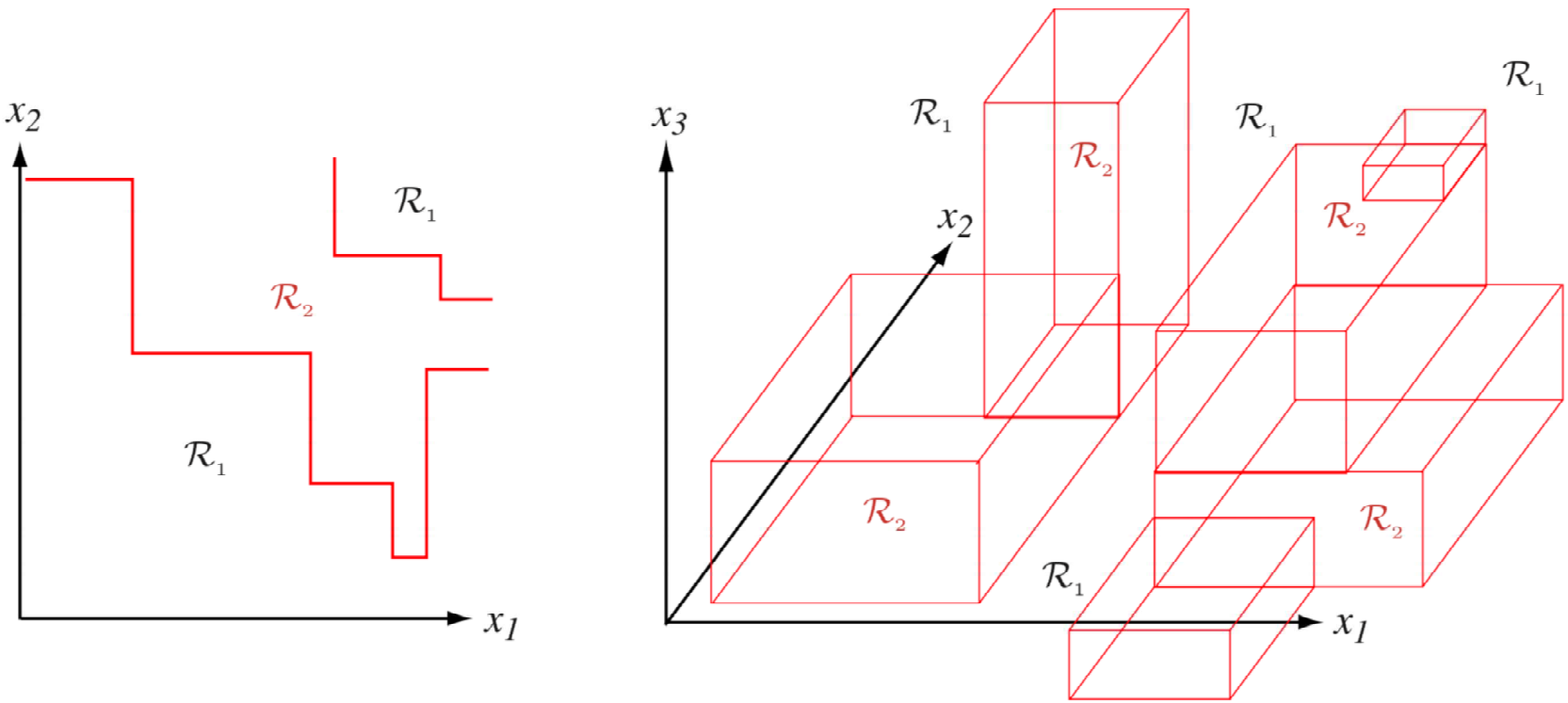
- Difficult to predict the number of clusters (*i.e.*, the optimal k)
- Initialization (*i.e.*, initial seeds) have a strong impact on the final results
- The order of the data has an impact on the final results
- Sensitive to scale: rescaling your datasets (normalization or standardization) will completely change the results. While this itself is not bad, not realizing that you have to spend extra attention to scaling your data might be bad
- Weakness of arithmetic mean is not robust to outliers. Very far data from the centroid may pull the centroid away from the real one

Outline

Decision Trees and Random Forests

Decision trees

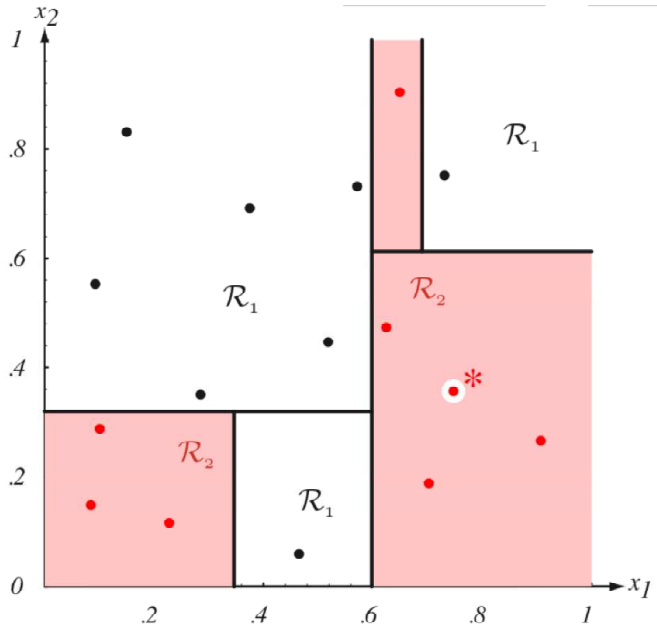
Monothetic decision trees create decision boundaries with portions perpendicular to the feature axes. The decision regions are marked $\mathcal{R}_1$ and $\mathcal{R}_2$ in these two-dimensional and three-dimensional two-class examples. With a sufficiently large tree, any decision boundary can be approximated arbitrarily well.

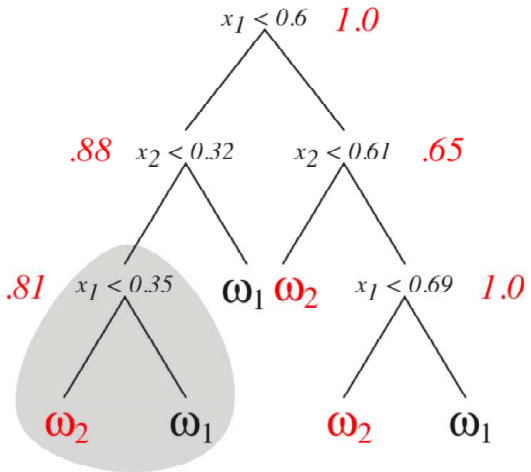


Decision trees: Example

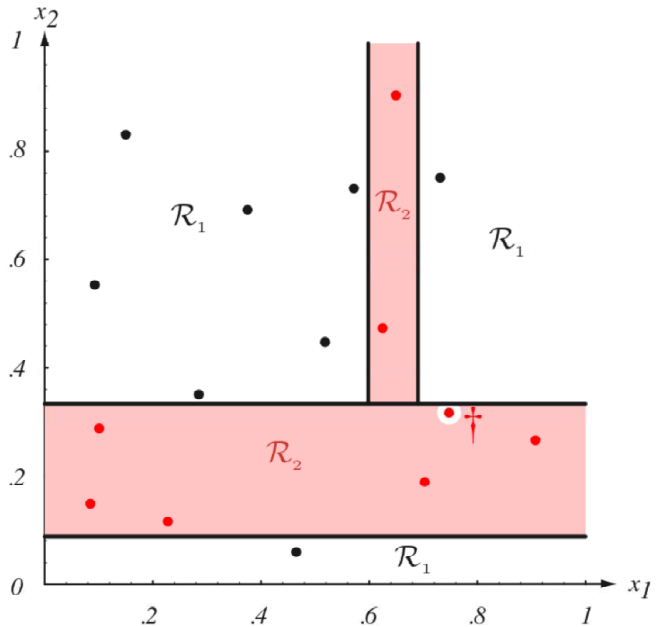
ω_1 (black)	
x_1	x_2
.15	.83
.09	.55
.29	.35
.38	.70
.52	.48
.57	.73
.73	.75
.47	.06

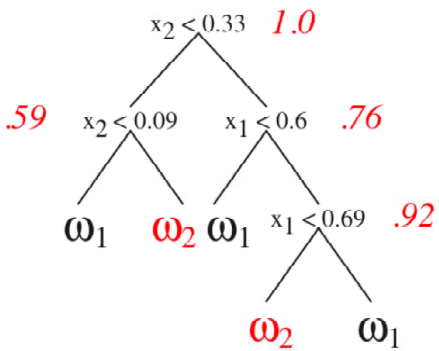
ω_2 (red)	
x_1	x_2
.10	.29
.08	.15
.23	.16
.70	.19
.62	.47
.91	.27
.65	.90
.75	.36* (.32†)



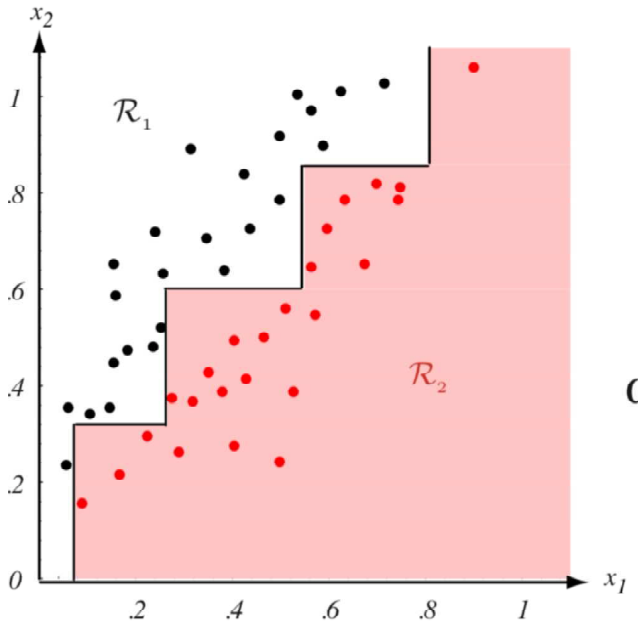


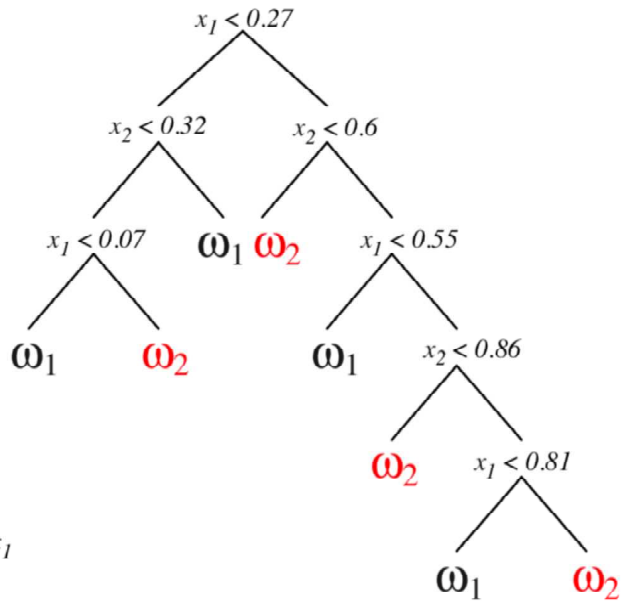
This particular training set shows how trees can be sensitive to details of the training points (A slight modification of $*$ into †). Such instability is due in large part to the discrete nature of decisions early in the tree learning.

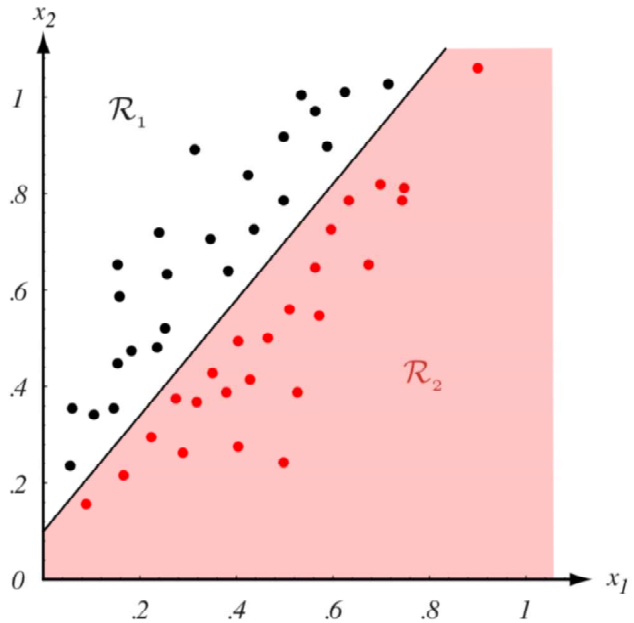


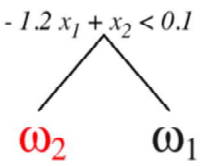


Decision trees





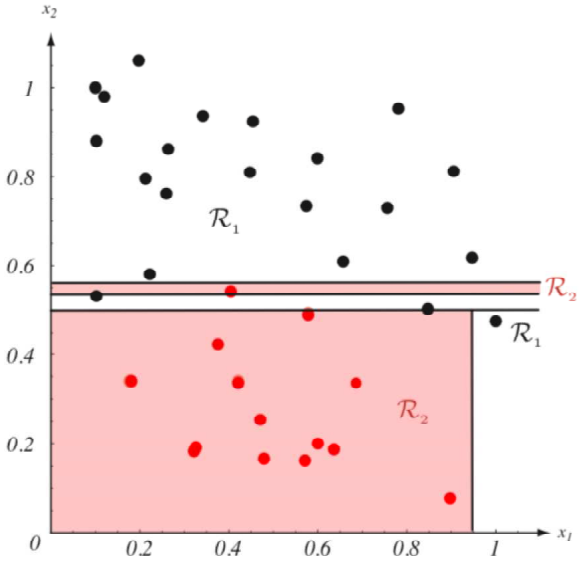


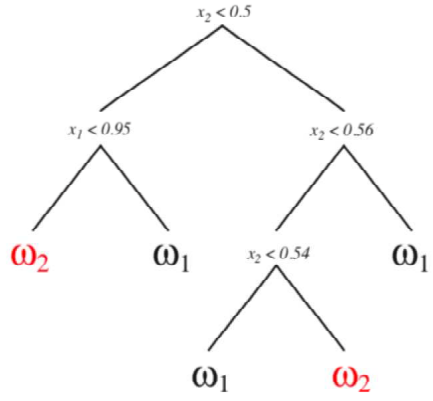


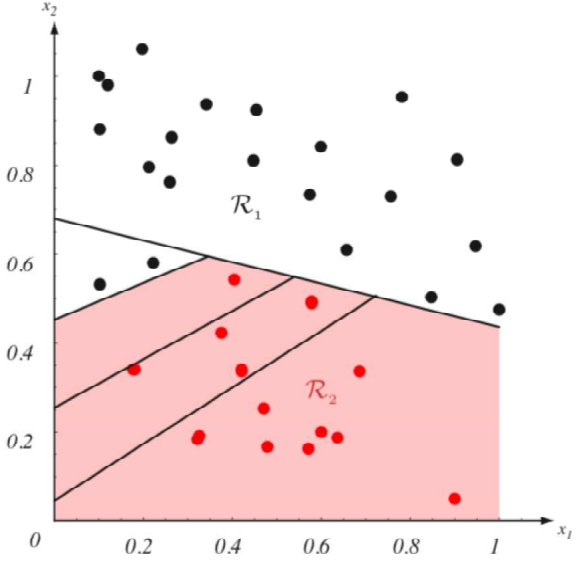
If the class of node decisions does not match the form of the training data, a very complicated decision tree will result, as shown at the top. Here decisions are parallel to the axes while in fact the data is better split by boundaries along another direction. If however "proper" decision forms are used (here, linear combinations of the features), the tree can be quite simple, as shown at the bottom.

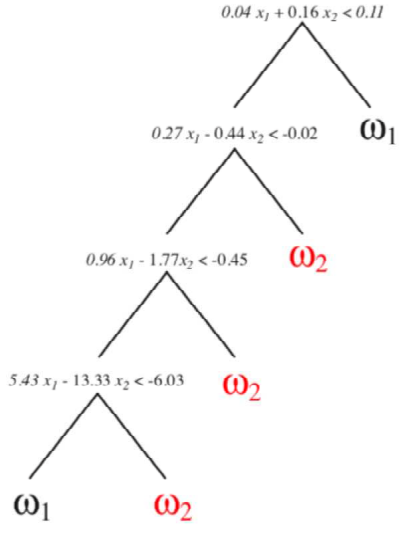
Decision trees

Another example





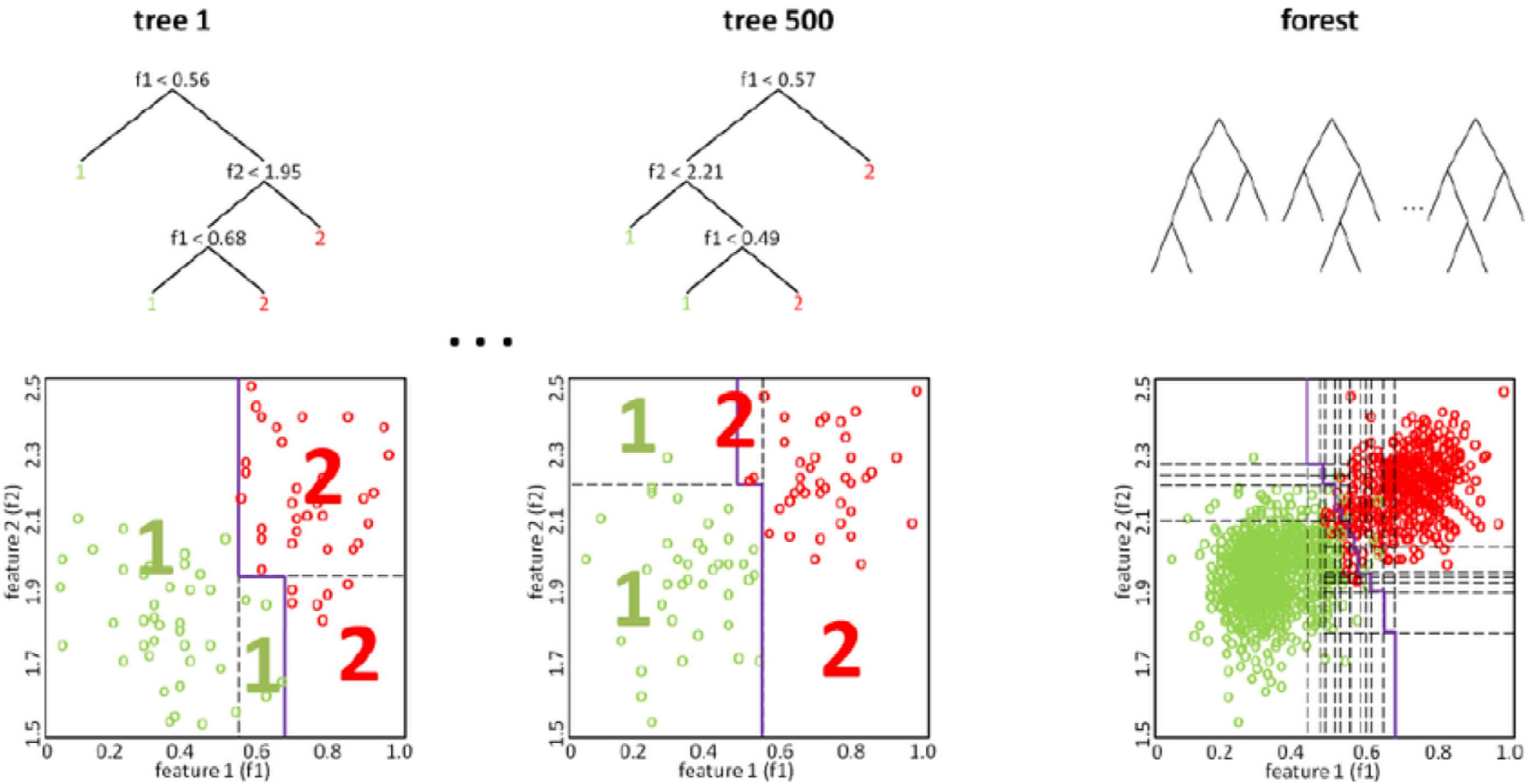




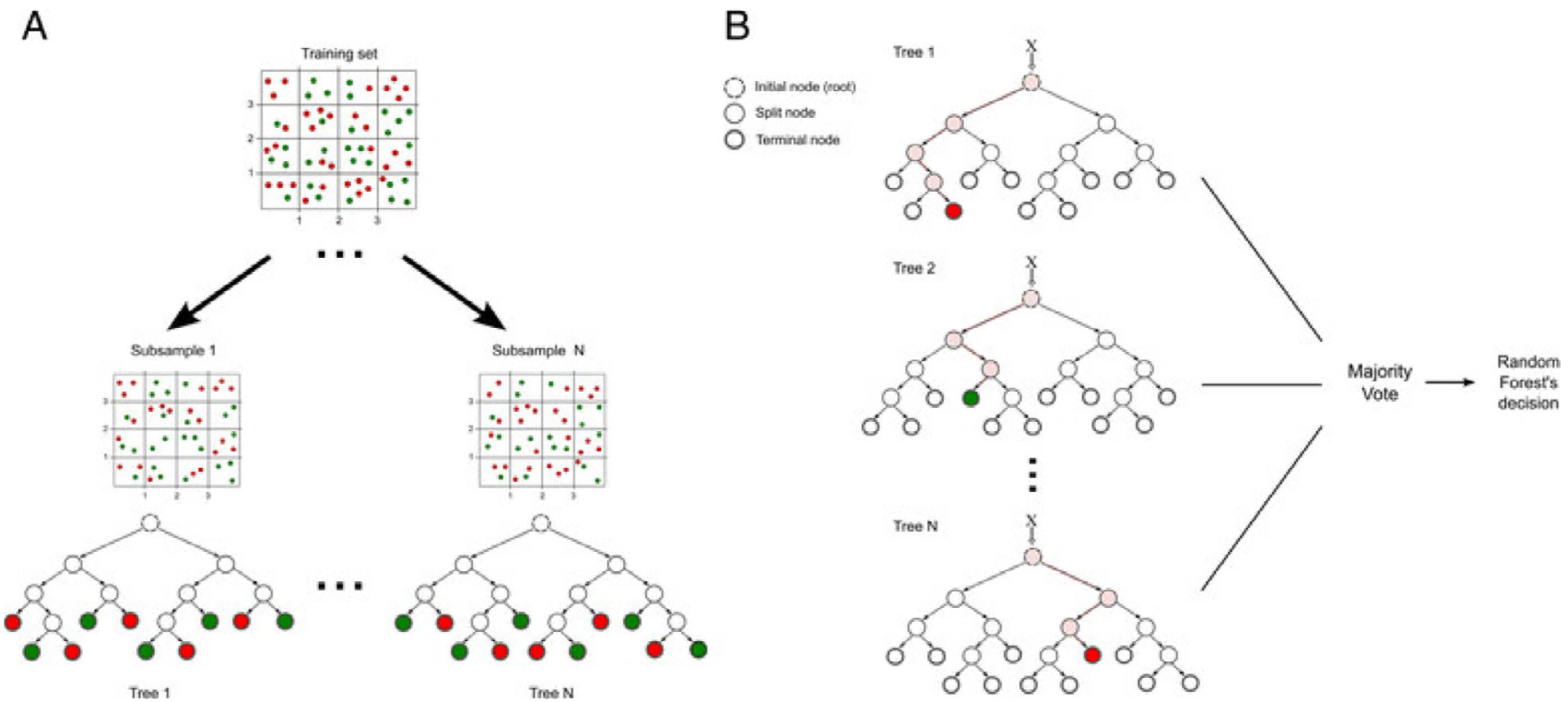
Random Forest

Random Forest

Main idea:
A random forest generates multiple small decision trees from random subsets of the training dataset. A majority vote is taken from these trees for classification.



Random Forest



Random Forest in Python

The `sklearn.ensemble` module includes two averaging algorithms based on randomized decision trees: the RandomForest algorithm and the Extra-Trees method. Both algorithms are perturb-and-combine techniques specifically designed for trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

As other classifiers, forest classifiers have to be fitted with two arrays: a sparse or dense array X of size `[n_samples, n_features]` holding the training samples, and an array Y of size `[n_samples]`[3] holding the target values (class labels) for the training samples:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

In random forests (see `RandomForestClassifier` and `RandomForestRegressor` classes), each tree in the ensemble is built from a sample drawn with replacement (*i.e.*, a bootstrap sample) from the training set. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, hence yielding an overall better model.

[3] Forests of trees also extend to multi-output problems (if Y is an array of size `[n_samples, n_outputs]`)

Random Forest in Python

Extremely Randomized Trees

In extremely randomized trees (see `ExtraTreesClassifier` and `ExtraTreesRegressor` classes), randomness goes one step further in the way splits are computed. As in random forests, a random subset of candidate features is used, but instead of looking for the most discriminative thresholds, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias:

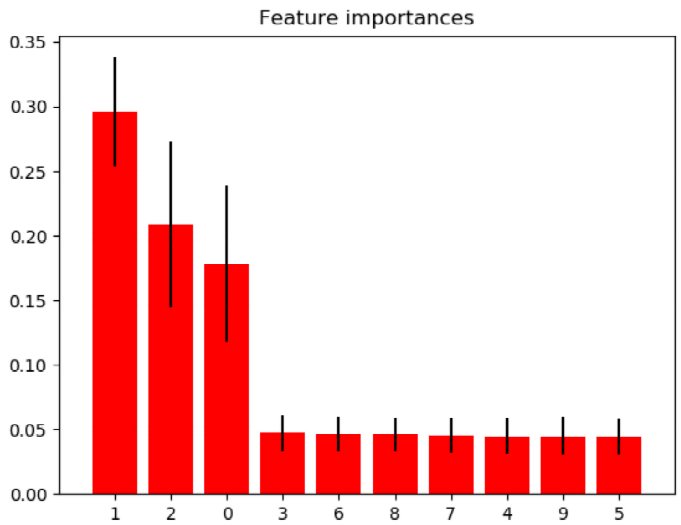
```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> X, y = make_blobs(n_samples=10000, n_features=10, centers=100, random_state=0)

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.98...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None, min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean() > 0.999
True
```

Random Forest in Python

Feature importances with forests of trees

Plot forest importances

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.ensemble import ExtraTreesClassifier

# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000,
                           n_features=10,
                           n_informative=3,
                           n_redundant=0,
                           n_repeated=0,
                           n_classes=2,
                           random_state=0,
                           shuffle=False)

# Build a forest and compute the feature importances
forest = ExtraTreesClassifier(n_estimators=250, random_state=0)

forest.fit(X, y)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```



Output

```
Feature ranking:
1. feature 1 (0.295902)
2. feature 2 (0.208351)
3. feature 0 (0.177632)
4. feature 3 (0.047121)
5. feature 6 (0.046303)
6. feature 8 (0.046013)
7. feature 7 (0.045575)
8. feature 4 (0.044614)
9. feature 9 (0.044577)
10. feature 5 (0.043912)
```

Random Forest in Python

See also: Plot forest IRIS

Outline

Final Remark

Outline: Next

Part 1: Introduction to Machine Learning

Part 2: ("Primal") Machine Learning Algorithms

Part 3: Statistical Learning Theory and Support Vector Machines

Part 4: Multiclass and Regression