Machine Learning

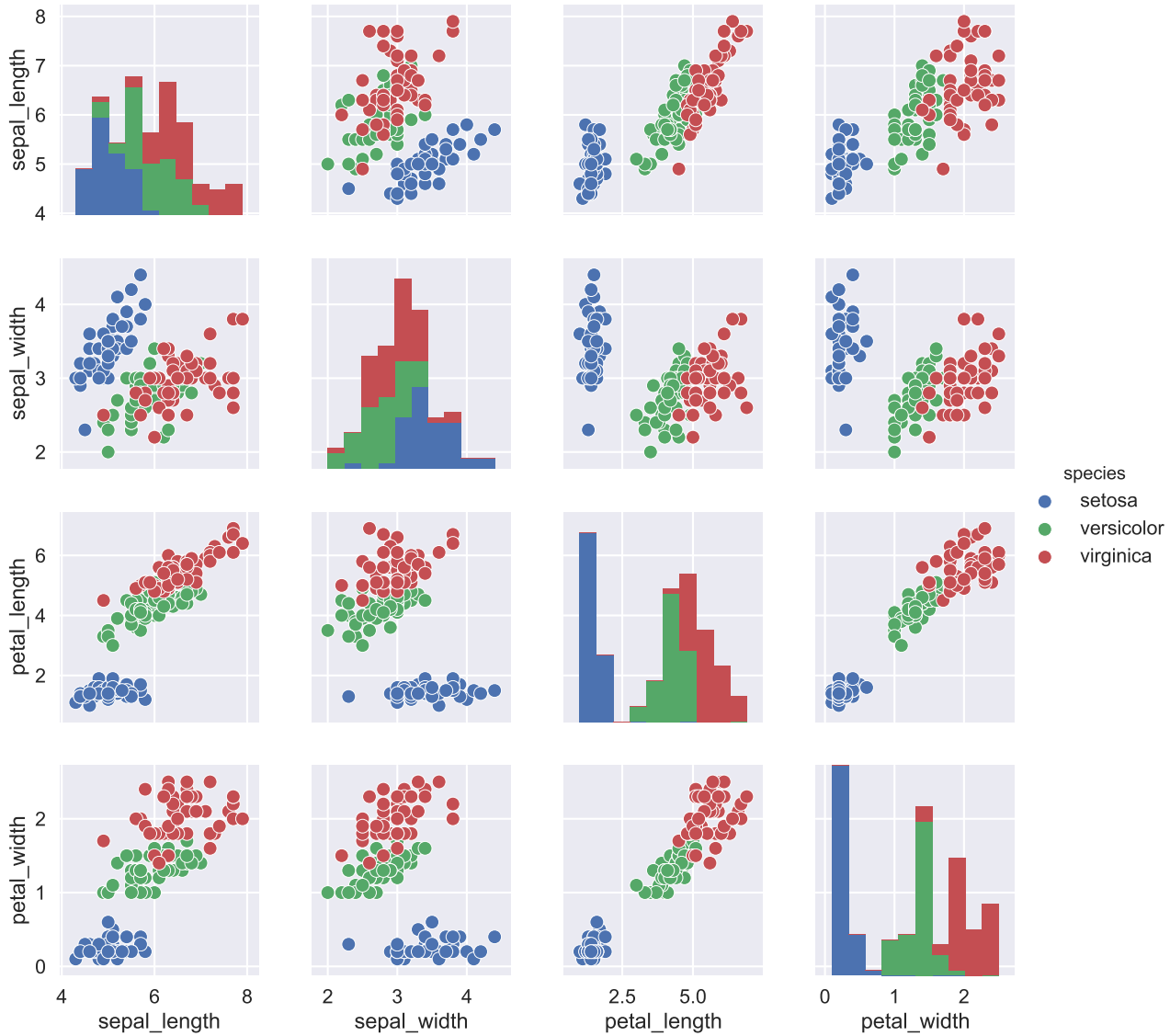
Lecture 2: k-Nearest Neighbors

Prof. Dr. Stephan Günnemann

Data Analytics and Machine Learning
Technical University of Munich

Winter term 2020/2021

Iris dataset



Iris dataset: 2 features

How do we intuitively label new samples by hand?
Look at the *surrounding* points. Do as your neighbor does.

1-NN algorithm

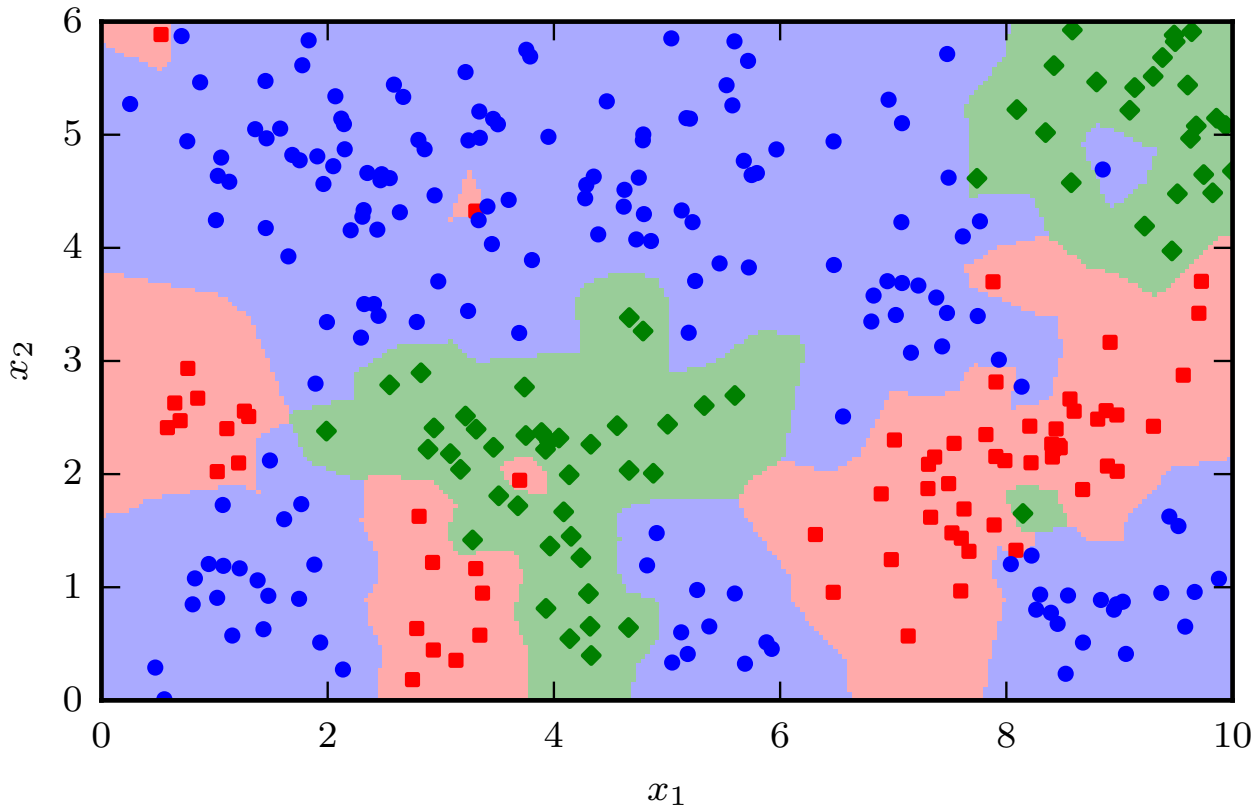
Given a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$
where $\boldsymbol{x}_i \in \mathbb{R}^D$ are features and $y_i \in \{1, \ldots, C\}$ are class labels

To classify new observations:

- define a distance measure (e.g. Euclidean distance)
- compute the nearest neighbor for all new data points
- and label them with the label of their nearest neighbor

This works for both *classification* and *regression*.

1-NN



This corresponds to a Voronoi tesselation.
And results in poor generalization...

k-Nearest Neighbor classification

More *robust* against errors in the training set:

Look at multiple nearest neighbors and pick the majority label.

k-Nearest Neighbor classification

More *robust* against errors in the training set:

Look at multiple nearest neighbors and pick the majority label.

Let $\mathcal{N}_k(\boldsymbol{x})$ be the k nearest neighbors of a vector $\boldsymbol{x}$, then in classification tasks:

$$p(y = c \mid \boldsymbol{x}, k) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\boldsymbol{x})} \mathbb{I}(y_i = c),$$

$$\hat{y} = \arg\max_c p(y = c \mid \boldsymbol{x}, k)$$

with the *indicator variable* $\mathbb{I}(e)$ is defined as:

$$\mathbb{I}(e) = \begin{cases} 1 \text{ if } e \text{ is true} \\ 0 \text{ if } e \text{ is false.} \end{cases}$$

k-Nearest Neighbor classification

More *robust* against errors in the training set:

Look at multiple nearest neighbors and pick the majority label.

Let $\mathcal{N}_k(\boldsymbol{x})$ be the k nearest neighbors of a vector $\boldsymbol{x}$, then in classification tasks:

$$p(y = c \mid \boldsymbol{x}, k) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\boldsymbol{x})} \mathbb{I}(y_i = c),$$

$$\hat{y} = \arg\max_c p(y = c \mid \boldsymbol{x}, k)$$

with the *indicator variable* $\mathbb{I}(e)$ is defined as:

$$\mathbb{I}(e) = \begin{cases} 1 \text{ if } e \text{ is true} \\ 0 \text{ if } e \text{ is false.} \end{cases}$$

i.e., the vector will be labeled by the mode of its neighbors' labels.

k-Nearest Neighbor classification: weighted

Look at multiple nearest neighbors and pick the weighted majority label.

k-Nearest Neighbor classification: weighted

Look at multiple nearest neighbors and pick the weighted majority label.
The weight is inversely proportional to the distance.

Let $\mathcal{N}_k(\boldsymbol{x})$ be the k nearest neighbors of a vector $\boldsymbol{x}$, then in classification tasks:

$$p(y = c \mid \boldsymbol{x}, k) = \frac{1}{Z} \sum_{i \in \mathcal{N}_k(\boldsymbol{x})} \frac{1}{\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)} \mathbb{I}(y_i = c),$$

$$\hat{y} = \arg\max_c p(y = c \mid \boldsymbol{x}, k)$$

with $Z = \sum_{i \in \mathcal{N}_k(x)} \frac{1}{\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)}$ the normalization constant and $\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)$ being a distance measure between $\boldsymbol{x}$ and $\boldsymbol{x}_i$.

k-Nearest-Neighbor regression

Regression is similar:

Let $\mathcal{N}_k(\boldsymbol{x})$ be the k nearest neighbors of a vector $\boldsymbol{x}$, then for regression:

$$\hat{y} = \frac{1}{Z} \sum_{i \in \mathcal{N}_k(\boldsymbol{x})} \frac{1}{\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)} \, y_i,$$

with $Z = \sum_{i \in \mathcal{N}_k(x)} \frac{1}{\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)}$ the normalization constant and $\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)$ being a distance measure between $\boldsymbol{x}$ and $\boldsymbol{x}_i$,

k-Nearest-Neighbor regression

Regression is similar:

Let $\mathcal{N}_k(\boldsymbol{x})$ be the k nearest neighbors of a vector $\boldsymbol{x}$, then for regression:

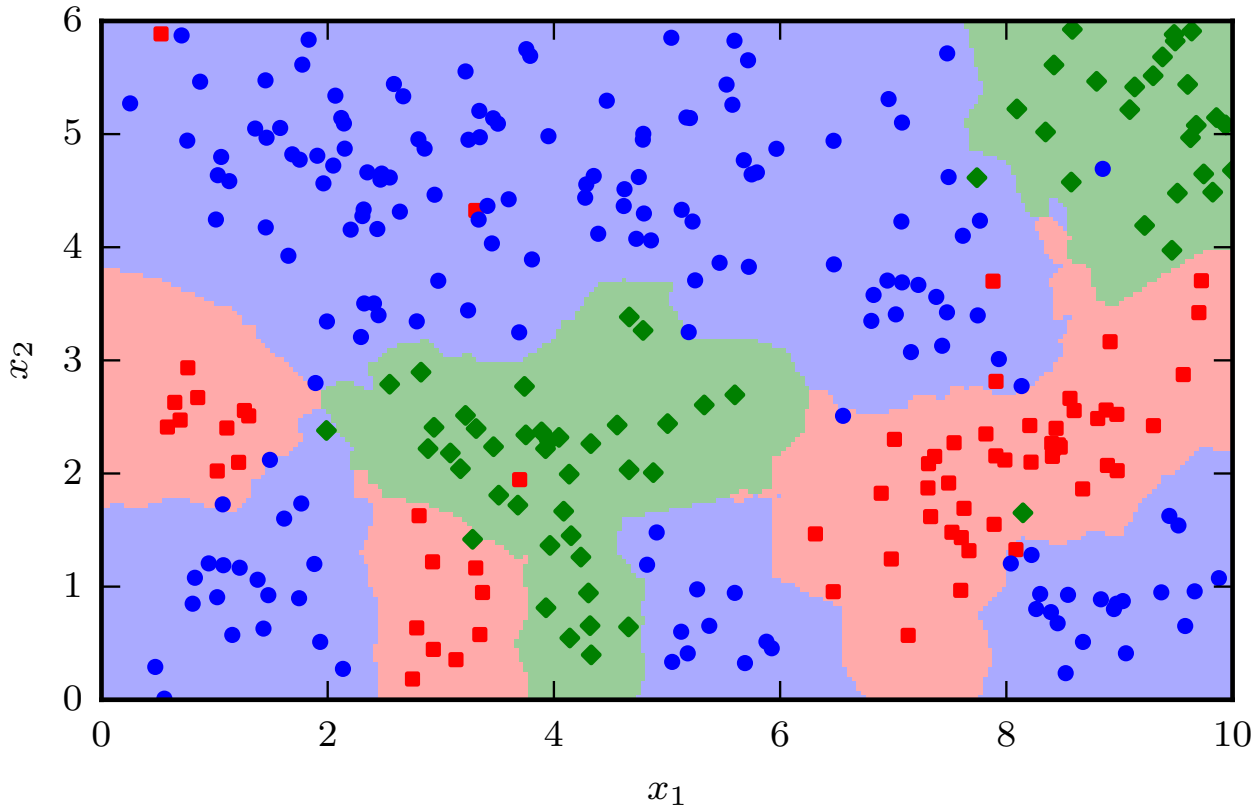
$$\hat{y} = \frac{1}{Z} \sum_{i \in \mathcal{N}_k(\boldsymbol{x})} \frac{1}{\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)} y_i,$$

with $Z = \sum_{i \in \mathcal{N}_k(x)} \frac{1}{\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)}$ the normalization constant and $\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i)$ being a distance measure between $\boldsymbol{x}$ and $\boldsymbol{x}_i$,

i.e., the vector will be labeled by a weighted mean of its neighbors' values.

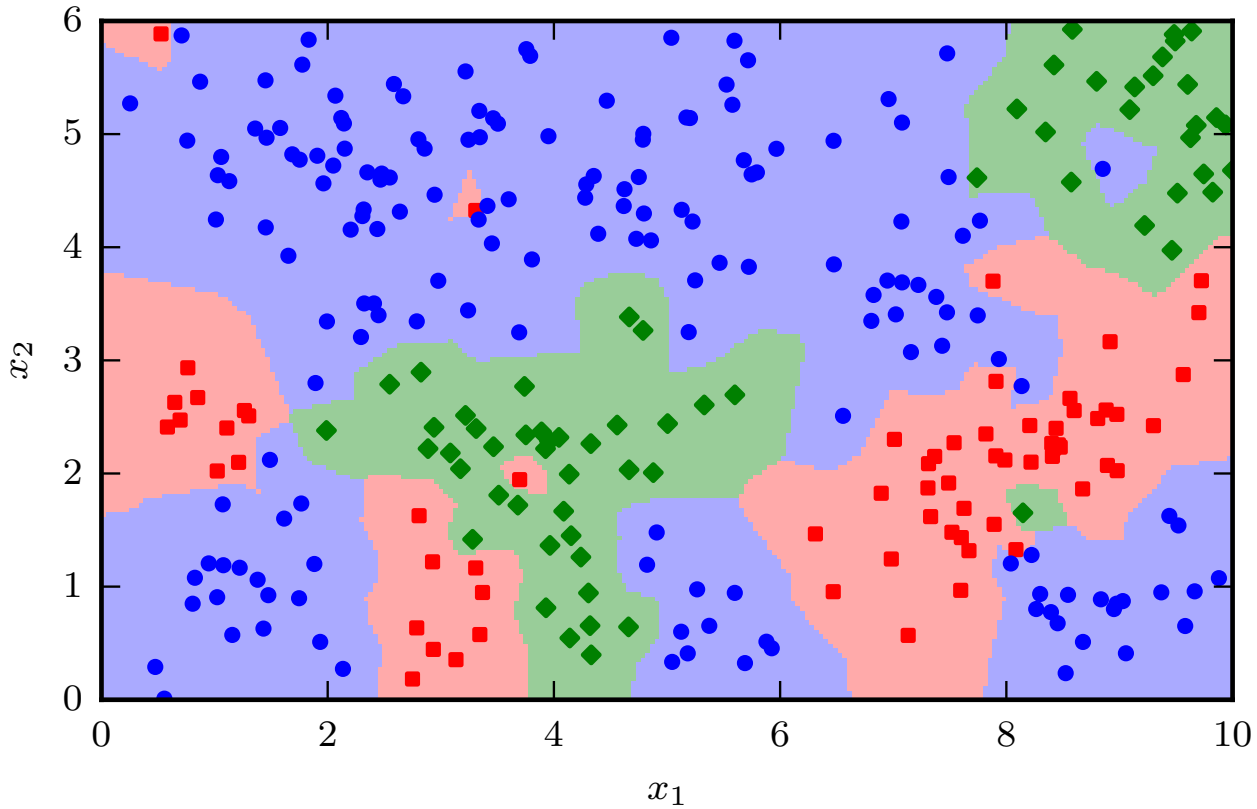
Note: y_i is a real number here (rather then categorical label).

3-NN



So, how many neighbors are best?

1-NN

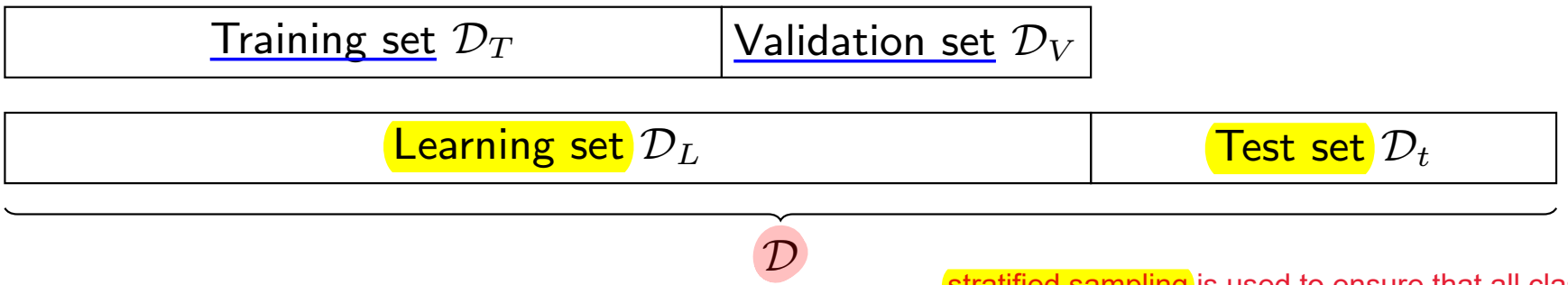


Compare the decision boundaries of 1-NN and 3-NN

Choosing k

Goal is generalization: pick k (called a *hyper-parameter*) that performs best[1] on unseen (future) data.

Unfortunately, no access to future data, so split the dataset $\mathcal{D}$:



stratified sampling is used to ensure that all classes would be included in all sets. i.e. avoid having unrepresentative classes in datasets

class proportionality shall be kept along the way!

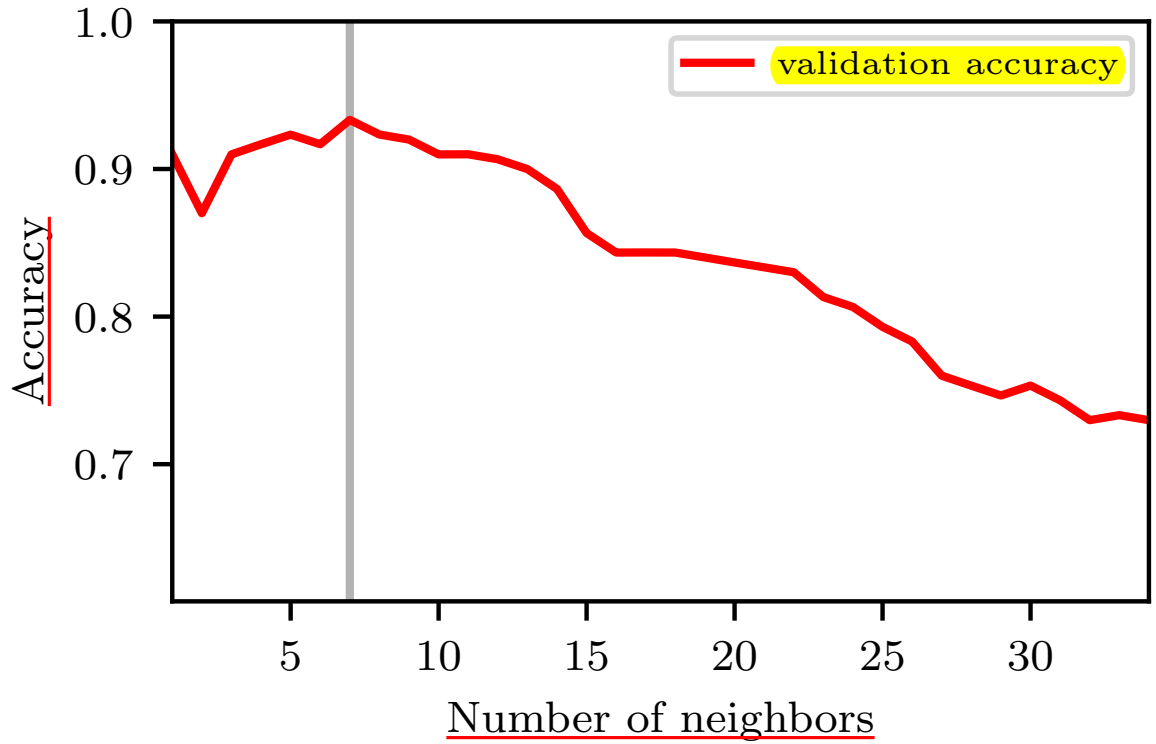
Hyper-parameter tuning procedure

- Learn the model using the training set
- Evaluate performance with different k on the *validation set* picking the best k
- Report final performance on the test set.[2]

[1] In terms of some predefined metric, i.e. accuracy

[2] Good data science practices: See slides on Decision Trees

Using validation set to choose k



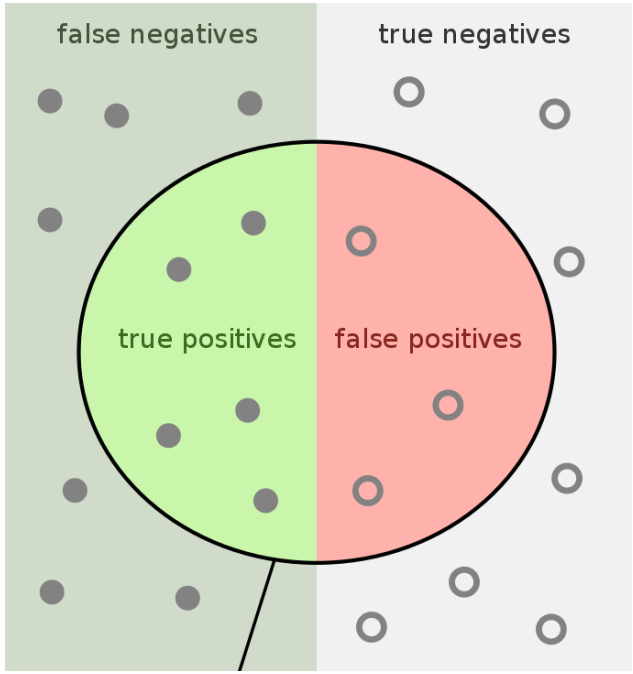
We choose $k = 7$.

Measuring classification performance

How can we assess the performance of a (binary) classification algorithm?

$\Rightarrow$ Confusion table

	True condition	
Predicted	$y = 1$	$y = 0$
$y = 1$	TP	FP
$y = 0$	FN	TN



TP = true prositive
TN = true negative
} correct predictions

FP = false prositive
FN = false negative
} wrong predictions

Accuracy: $$\text{acc} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision: $$\text{prec} = \frac{TP}{TP + FP}$$

Sensitivty/Recall: $$\text{rec} = \frac{TP}{TP + FN}$$

Specificity: $$\text{tnr} = \frac{TN}{FP + TN}$$

False Negative Rate: $$\text{fnr} = \frac{FN}{TP + FN}$$

False Positive Rate: $$\text{fpr} = \frac{FP}{FP + TN}$$

the higher the better

F1 Score: $$f1 = \frac{2 \cdot \text{prec} \cdot \text{rec}}{\text{prec} + \text{rec}}$$

$\Rightarrow$ Trade-off between precision and recall: increasing one (most often) leads to decreasing the other

General note: Be careful when you have imbalanced classes!

Distance measures

- K-NN can be used with various distance measures $\rightarrow$ highly flexible
- Euclidean distance (L_2 norm): $\sqrt{\sum_i (u_i - v_i)^2}$

Distance measures

- K-NN can be used with various distance measures $\rightarrow$ highly flexible
- Euclidean distance (L_2 norm): $\sqrt{\sum_i (u_i - v_i)^2}$
- L_1 norm: $\sum_i |u_i - v_i|$

Distance measures

- K-NN can be used with various distance measures → highly flexible
- Euclidean distance (L_2 norm): $\sqrt{\sum_i (u_i - v_i)^2}$

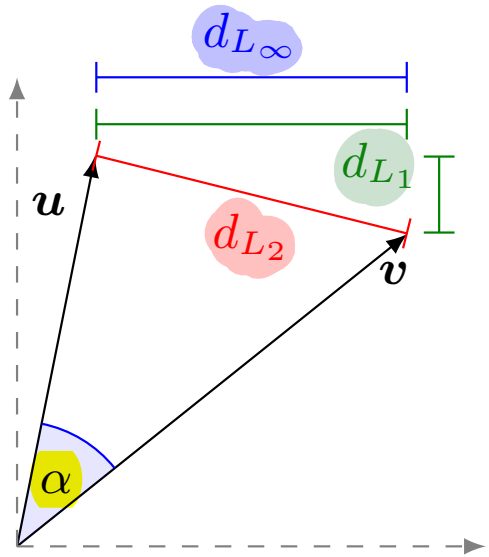
- L_1 norm: $\sum_i |u_i - v_i|$
- L_∞ norm: $\max_i |u_i - v_i|$

Distance measures

- K-NN can be used with various distance measures → highly flexible
- Euclidean distance (L_2 norm): $\sqrt{\sum_i (u_i - v_i)^2}$

- L_1 norm: $\sum_i |u_i - v_i|$
- L_∞ norm: $\max_i |u_i - v_i|$
- Angle:

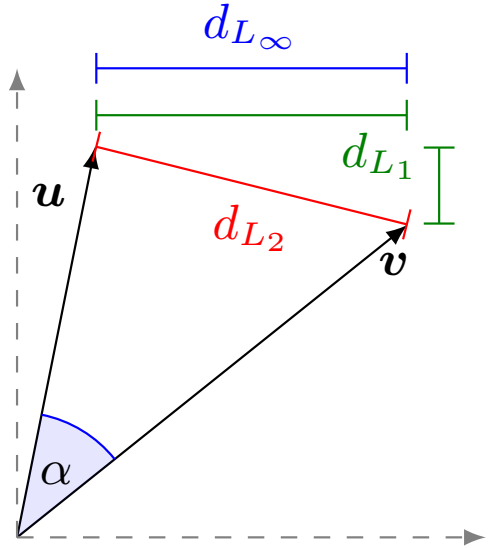
$$\cos\alpha = \frac{\boldsymbol{u}^T \boldsymbol{v}}{\|\boldsymbol{u}\| \|\boldsymbol{v}\|}$$



- K-NN can be used with various distance measures → highly flexible
- Euclidean distance (L_2 norm): $\sqrt{\sum_i (u_i - v_i)^2}$

- L_1 norm: $\sum_i |u_i - v_i|$
- L_∞ norm: $\max_i |u_i - v_i|$
- Angle:

$$\cos\alpha = \frac{\boldsymbol{u}^T \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}$$



covariance matrix

- *Mahalanobis distance* ($\boldsymbol{\Sigma}$ is *positive (semi) definite* and *symmetric*):

$$\sqrt{(\boldsymbol{u} - \boldsymbol{v})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{u} - \boldsymbol{v})}$$

is a multi-dim generalization of measuring how many std away point P is from the mean of distribution D

If covariance is identity matrix i.e. data is normalized,

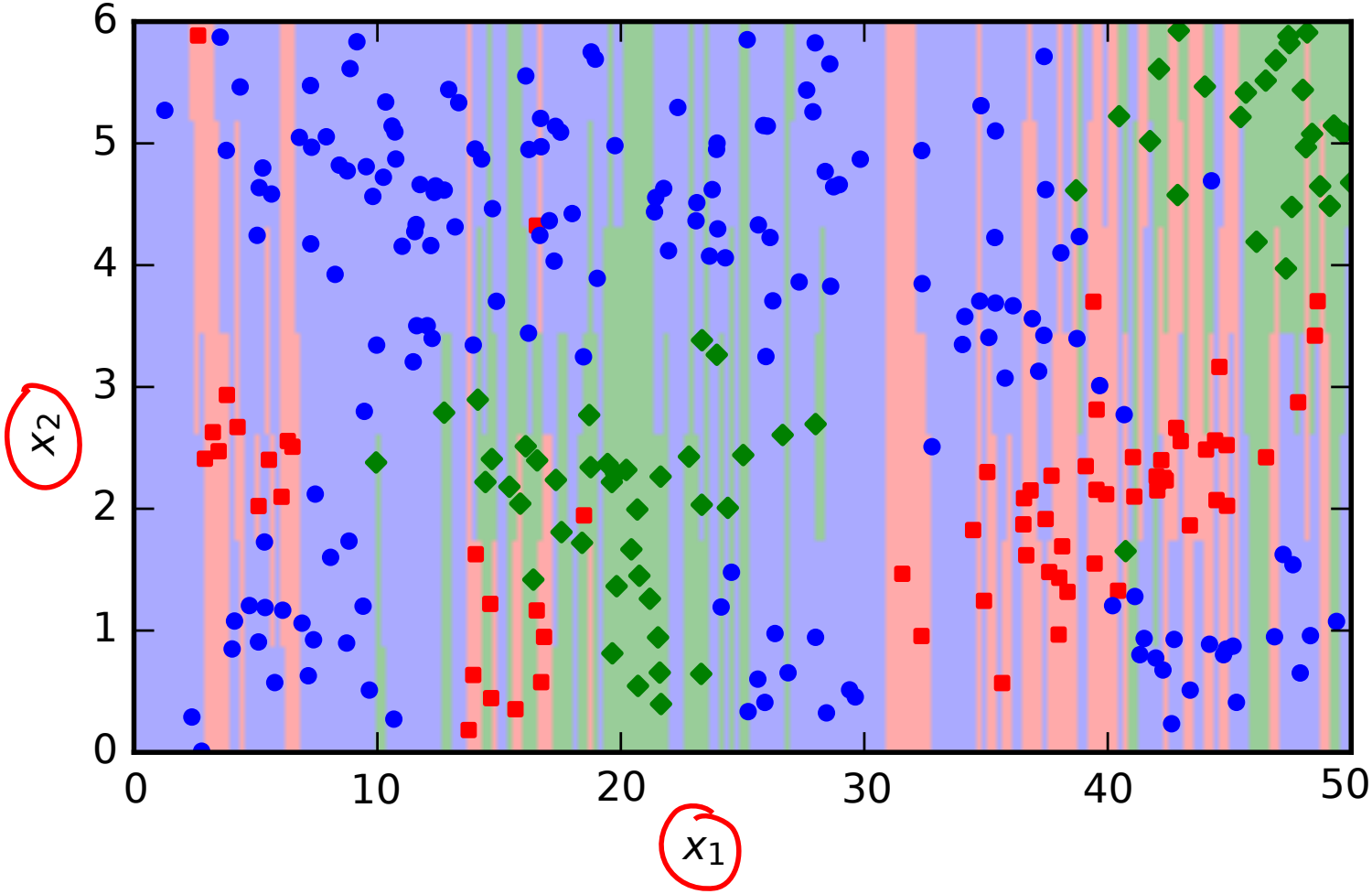
then Mahalanobis dist reduces to Euclidean dist

- Hamming distance, Edit distance, ...

Scaling issues

Assuming x1 is of larger magnitude than x2,

Euclidean dis changes significantly along x1, while along x2 changes are very small



The same old example but one of our features is in the order of meters, the other in the order of centimeters. $(k = 1)$

Circumventing scaling issues

- Data *standardization*
  Scale each feature to zero mean and unit variance.

$$x_{i,\text{std}} = \frac{x_i - \mu_i}{\sigma_i}$$

Z-score

(This is a standard procedure in machine learning. Many models are sensitive to differences in scale.)

≠ Normalization typically means rescales the values into a range of [0, 1], yet outliers shall be observed!

- Data *standardization*
  Scale each feature to zero mean and unit variance.

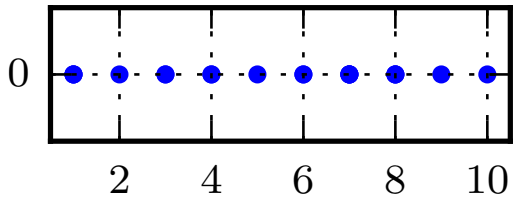
$$x_{i,\text{std}} = \frac{x_i - \mu_i}{\sigma_i}$$

  (This is a standard procedure in machine learning. Many models are sensitive to differences in scale.)
- Use the Mahalanobis distance.

$$\text{mahalanobis}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \sqrt{(\boldsymbol{x}_1 - \boldsymbol{x}_2)^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_1 - \boldsymbol{x}_2)}$$

$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \cdots & 0 \\ 0 & 0 & \sigma_n^2 \end{bmatrix}$ is equal to Euclidean distance on normalized data

The curse of dimensionality



Given a discrete one-dimensional input space $x \in \{1, 2, \ldots, 10\}$

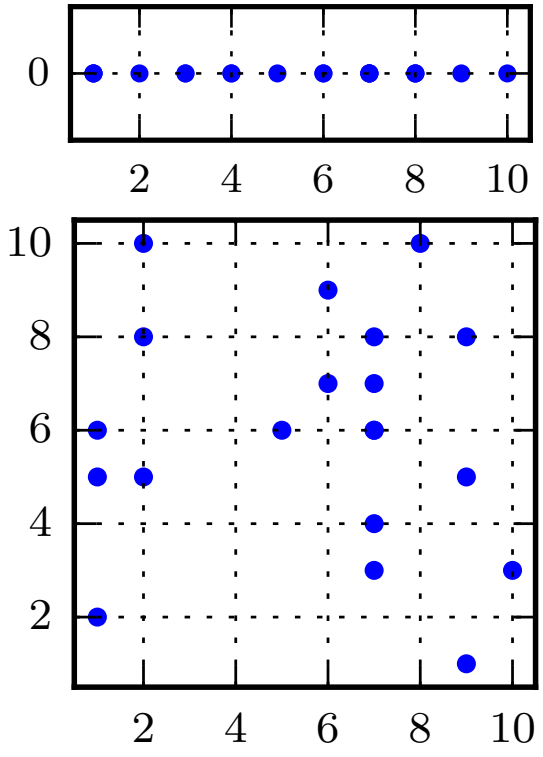
For $N = 20$ uniformly distributed samples the data covers 100% of the input space.

The curse of dimensionality

Given a discrete one-dimensional input space $x \in \{1, 2, \ldots, 10\}$

For $N = 20$ uniformly distributed samples the data covers 100% of the input space.

Add a second dimension (now $\boldsymbol{x} \in \{1, \ldots, 10\}^2$) and your data only covers 18% of the input space.



When the dimensionality increases, the volume of space increase so fast that the available data become sparse.

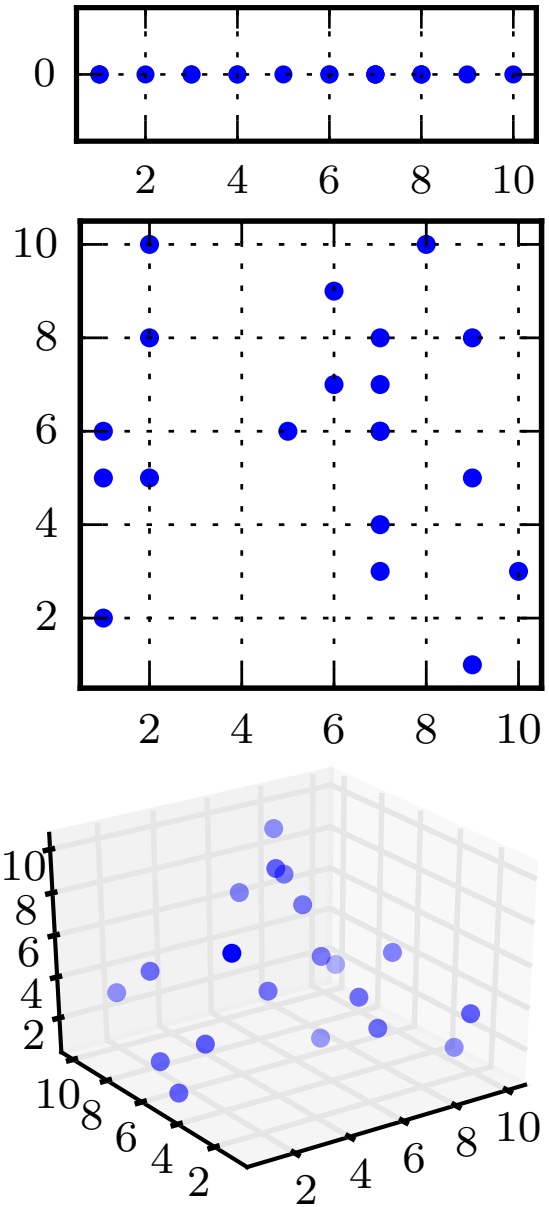
Given a discrete one-dimensional input space $x \in \{1, 2, \ldots, 10\}$

For $N = 20$ uniformly distributed samples the data covers 100% of the input space.

Add a second dimension (now $\boldsymbol{x} \in \{1, \ldots, 10\}^2$) and your data only covers 18% of the input space.

Once you add a third dimension you only cover 2%.

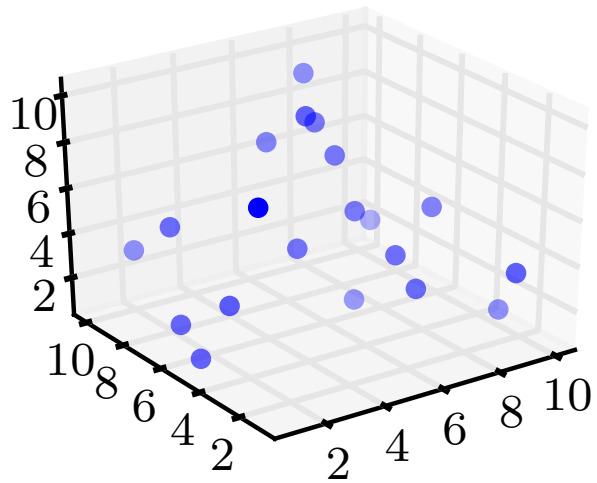
The space becomes rather empty; finding neighbours would become harder.



- The nearest neighbor will now be pretty far away..
- N has to grow exponentially with the number of features. Consider this when using k-NN on high-dimensional data.

training examples

In general, using kNN on high-dim data is NOT a good idea!



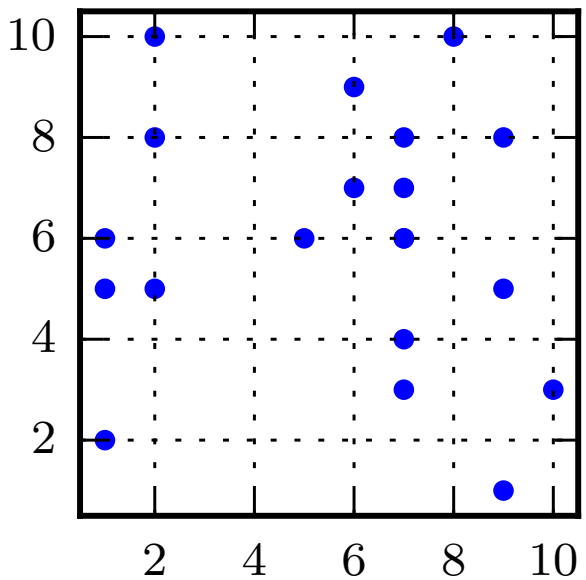
Practical considerations

Expensive: memory and naive inference are both $O(N)$:

we need to store the entire training data and compare with all training instances to find the nearest neighbor

Alternatively,

Solution: use tree-based search structures (e.g. k-d tree) for efficient (approximate) NN [3]



kNN is considered a lazy learner; since there is technically no model to be trained. It waits to get a test sample; to do calculations.

[3] At the expense of an additional computation performed only once

What we learned

- k-NN Algorithm
- Train-validation-test split
- Measuring classification performance
- Distance metrics
- Curse of dimensionality

Reading material

Main reading

- "Machine Learning: A Probabilistic Perspective" by Murphy [ch. 1.4.1 - 1.4.3]

Extra reading

- "Bayesian Reasoning and Machine Learning" by Barber [ch. 14]

Slides adapted from previous versions by W. Koepp & D. Korhammer