

# Machine Learning

## Lecture 2: $k$ -Nearest Neighbors

---

Prof. Dr. Stephan Günnemann

Data Mining and Analytics  
Technical University of Munich

15.10.2019

# Reading material

## Main reading

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

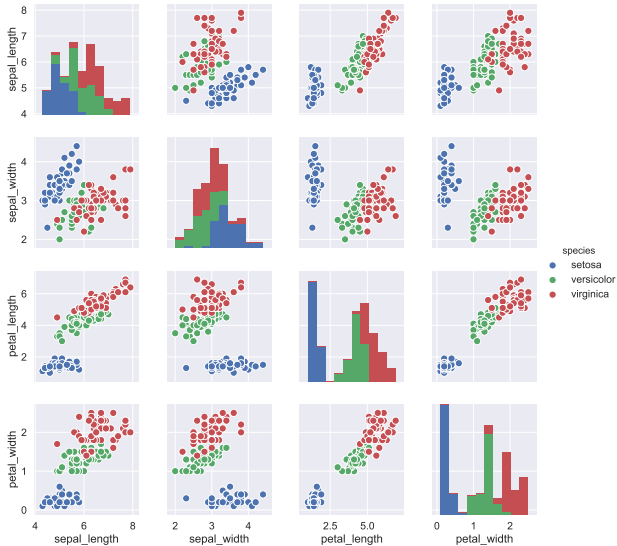
## Extra reading

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

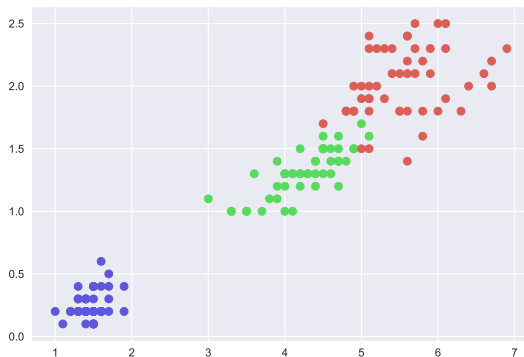
---

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

# 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

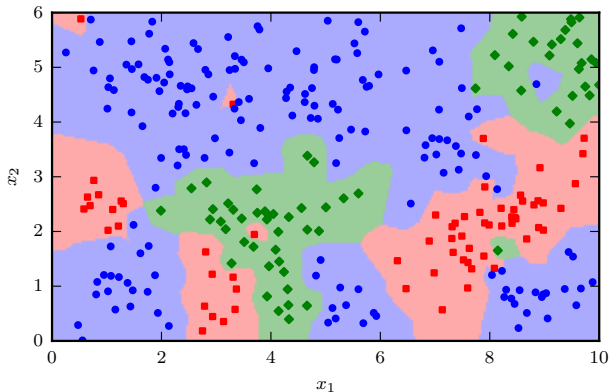
Given a training dataset  $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$   
where  $\mathbf{x}_i \in \mathbb{R}^D$  are features and  $y_i \in \{1, \dots, C\}$  are class labels

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 tessellation.  
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.

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

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

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

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. The weight is **inversely proportional** to the distance.

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

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

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

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



# $k$ -Nearest-Neighbor regression

Regression is similar:

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

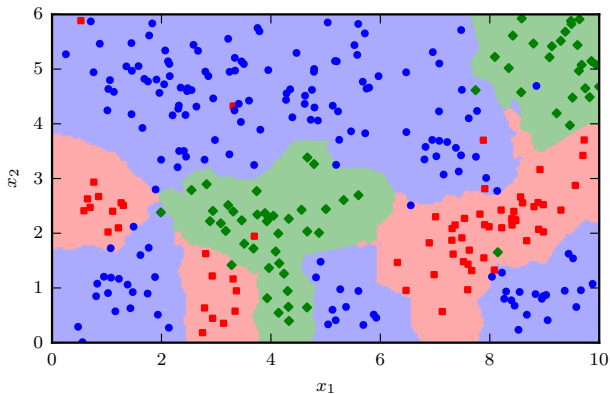
$$\hat{y} = \frac{1}{Z} \sum_{i \in \mathcal{N}_k(\mathbf{x})} \frac{1}{d(\mathbf{x}, \mathbf{x}_i)} y_i,$$

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

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

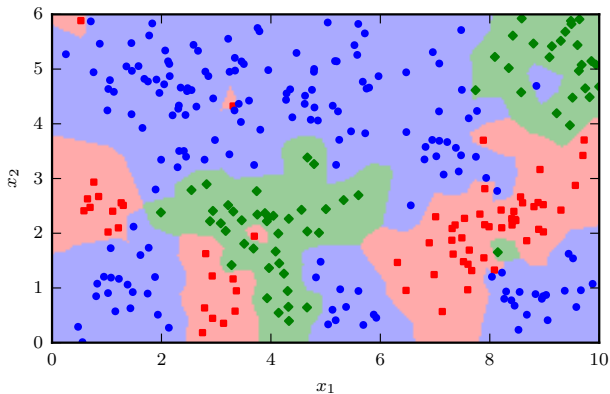
Note:  $y_i$  is a real number here (rather than categorical label).

# 3-NN



So, how many neighbors are best?

# 1-NN

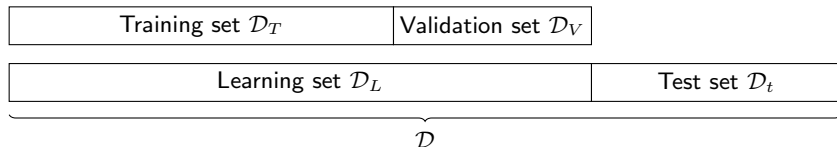


Compare the decision boundaries of 1-NN and 3-NN

# Choosing $k$

Goal is **generalization**: pick  $k$  (called a *hyper-parameter*) that performs best<sup>1</sup> on unseen (future) data.

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



Hyper-parameter tuning procedure

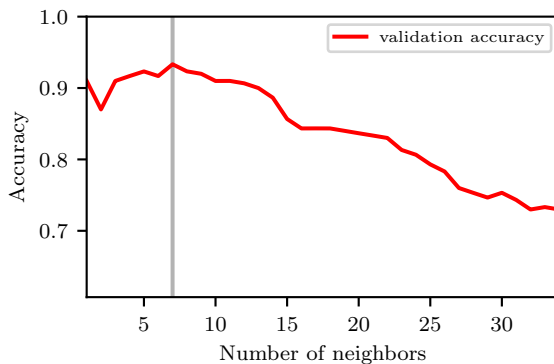
- Learn the model using the training set
- Evaluate performance with different  $k$  on the *validation set* picking the best  $k$
- Report final performance on the test set.<sup>2</sup>

---

<sup>1</sup>In terms of some predefined metric, i.e. accuracy

<sup>2</sup>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?

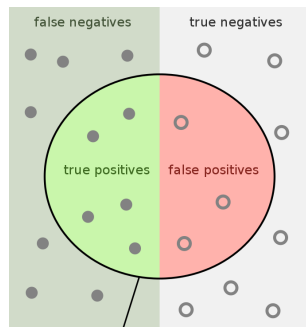
⇒ Confusion table

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

$TP$  = true positive  
 $TN$  = true negative  
 $FP$  = false positive  
 $FN$  = false negative

} correct predictions

} wrong predictions



# Measuring classification performance

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

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

Sensitivity/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: 
$$\text{fpr} = \frac{FP}{FP + TN}$$

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

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

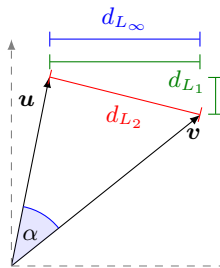
General note: Be careful when you have imbalanced classes!

# Distance measures

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

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

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



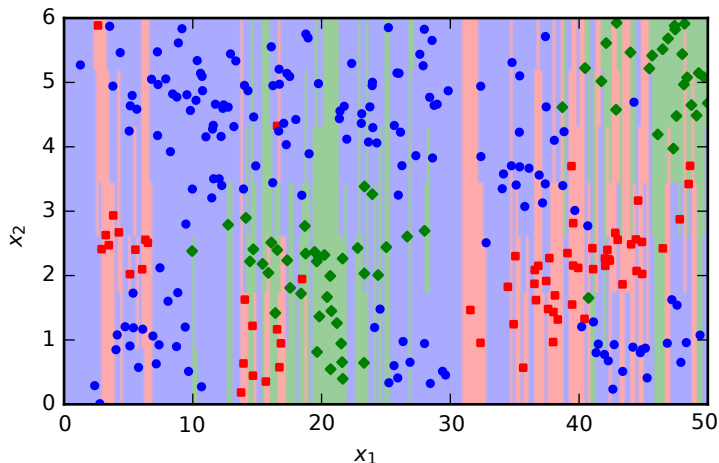
- Mahalanobis distance ( $\Sigma$  is positive (semi) definite and symmetric):

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

- Hamming distance, Edit distance, ...



# Scaling issues



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.

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

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

- Use the Mahalanobis distance.

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

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

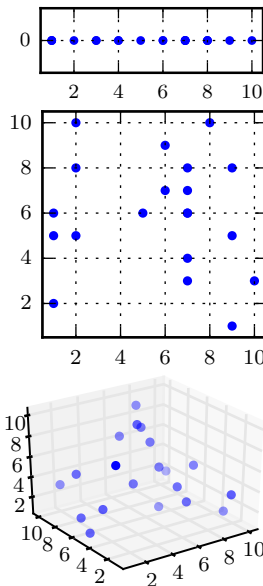
# The curse of dimensionality

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

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

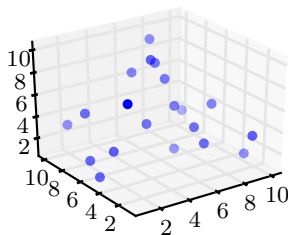
Add a second dimension (now  
 $\mathbf{x} \in \{1, \dots, 10\}^2$ ) and your data only covers  
18% of the input space.

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



# The curse of dimensionality

- 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.

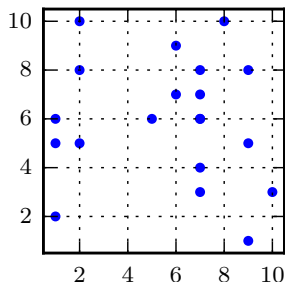


# Practical considerations

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

Solution: use tree-based search structures (e.g. k-d tree) for efficient (approximate) NN<sup>3</sup>



---

<sup>3</sup>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