Lecture 3: Supervised Learning

Machine Learning (BBWL)

Michael Mommert, University of St. Gallen

Today’s lecture

Supervised learning setup

Supervised learning concepts

Benchmarking and metrics

Linear models

Nearest Neighbor models

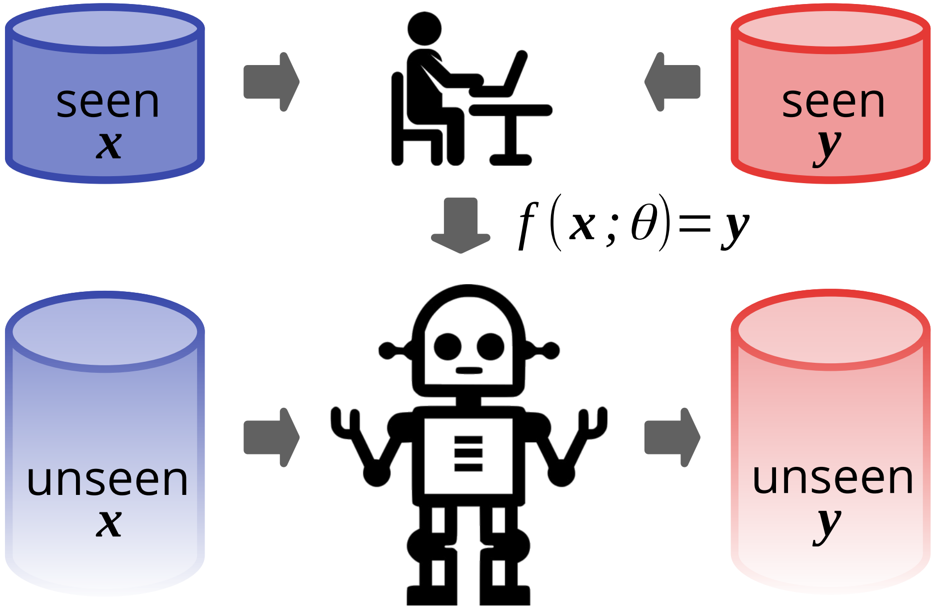
Tree-based models

Supervised Learning setup

General goal for supervised problems:
Find a function ("task") that relates input data ($\boldsymbol{x}$) to output data ($\boldsymbol{y}$) with hyperparameters (θ) such that: $f(\boldsymbol{x};\theta)=\boldsymbol{y}$

A **hyperparameter** is a model parameter that the model not learns.

Traditional (Rule-based) Approach:



seen $\boldsymbol{x}$

seen $\boldsymbol{y}$

$f(\boldsymbol{x};\theta)=\boldsymbol{y}$

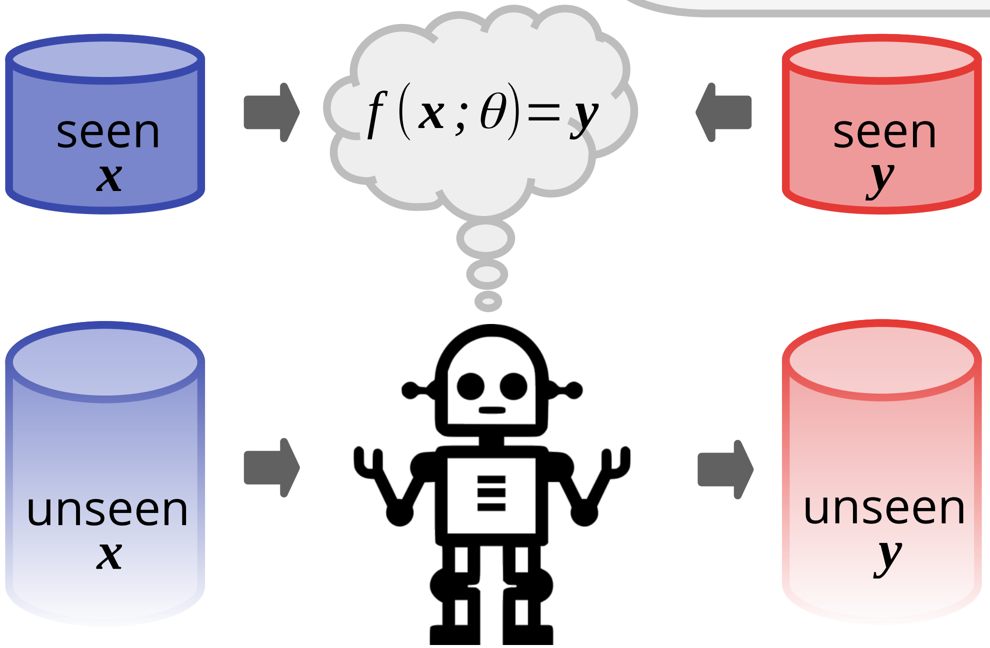
unseen $\boldsymbol{x}$

unseen $\boldsymbol{y}$

Input data ($\boldsymbol{x}$)

Output data ($\boldsymbol{y}$)

Machine-Learning Approach:



seen $\boldsymbol{x}$

$f(\boldsymbol{x};\theta)=\boldsymbol{y}$

seen $\boldsymbol{y}$

unseen $\boldsymbol{x}$

unseen $\boldsymbol{y}$

Input data ($\boldsymbol{x}$)

Output data ($\boldsymbol{y}$)

What tasks can ML learn?

(Multi-class) Classification:
Mapping input features to discrete classes of a single label

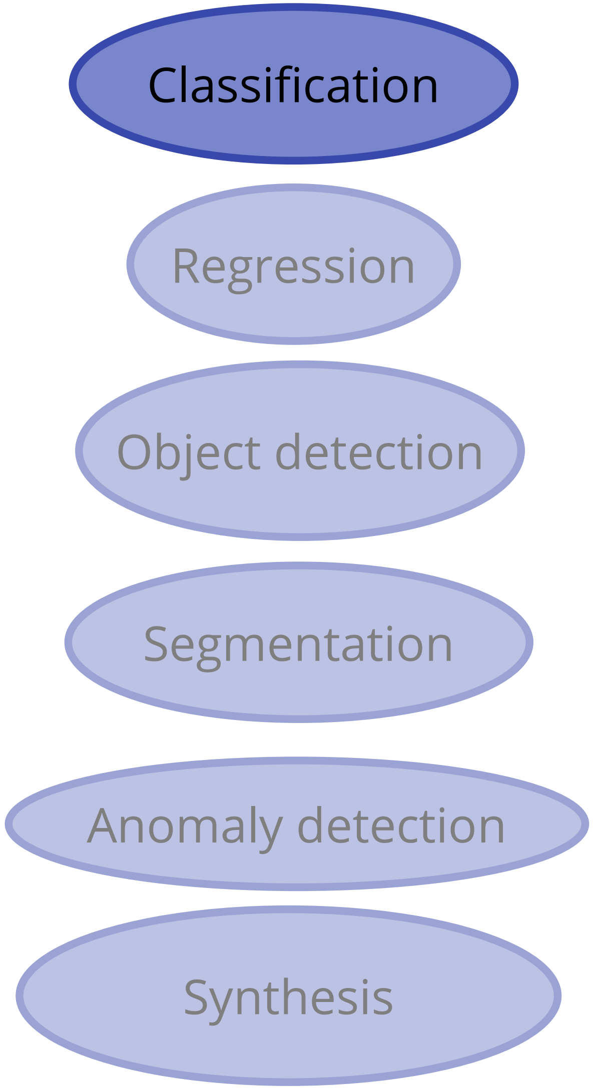
Example: label / classes

Color
red
green
blue

Binary classification:
Mapping input features to a binary label

Example: label / two classes

Status
on
off



Classification

Regression

Object detection

Segmentation

Anomaly detection

Synthesis

Multi-label classification:
Mapping input features to discrete classes of multiple labels

Example: labels / classes

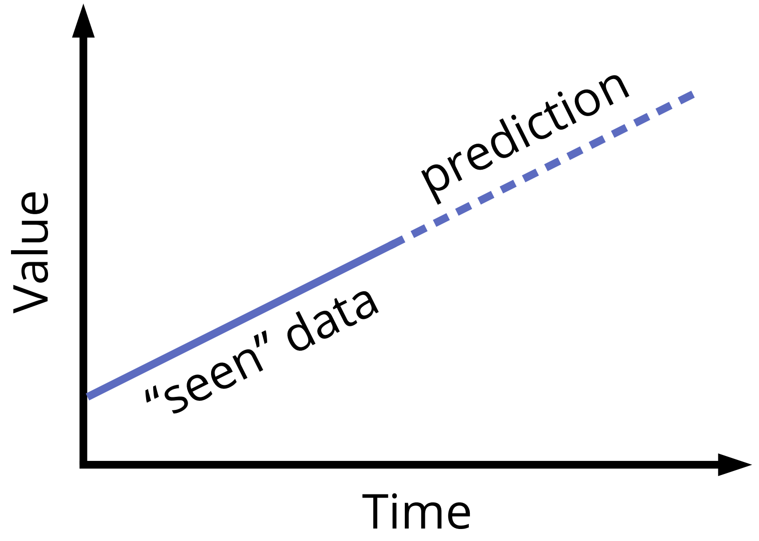
Color	Sort	Quality
red	A	good
green	B	medium
blue	C	bad
...	...	...

What tasks can ML learn?

- Classification
- **Regression**
- Object detection
- Segmentation
- Anomaly detection
- Synthesis

Regression:
Mapping input features to continuous variable

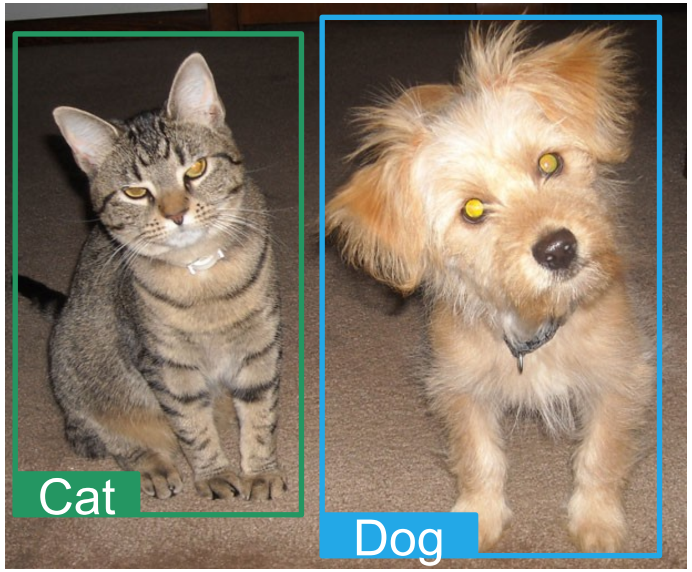
Example:

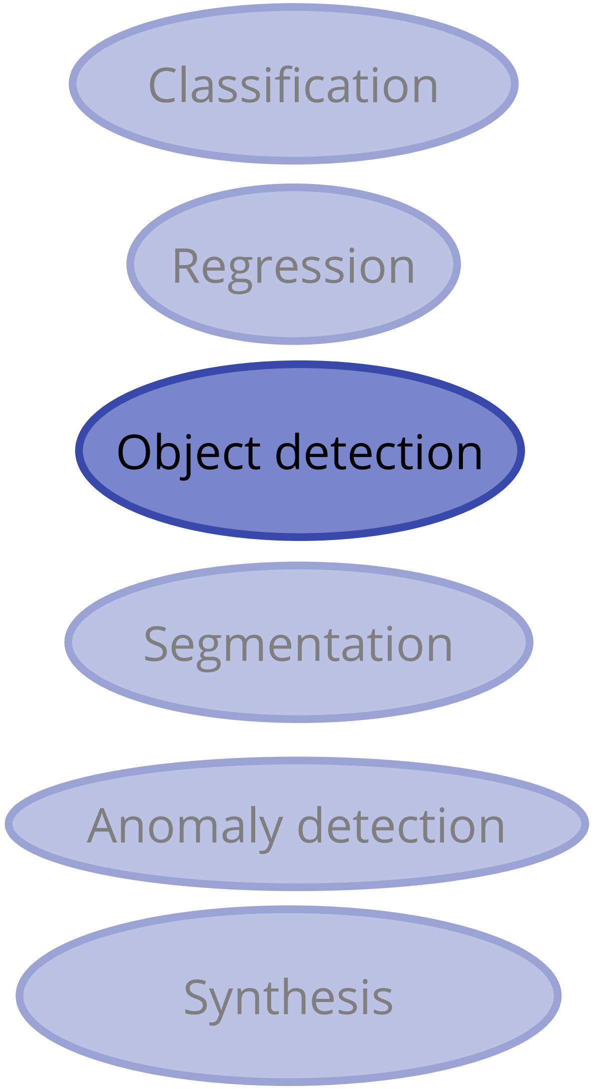


What tasks can ML learn?

Object detection:
Approximately localize features in image data with bounding boxes

Example:

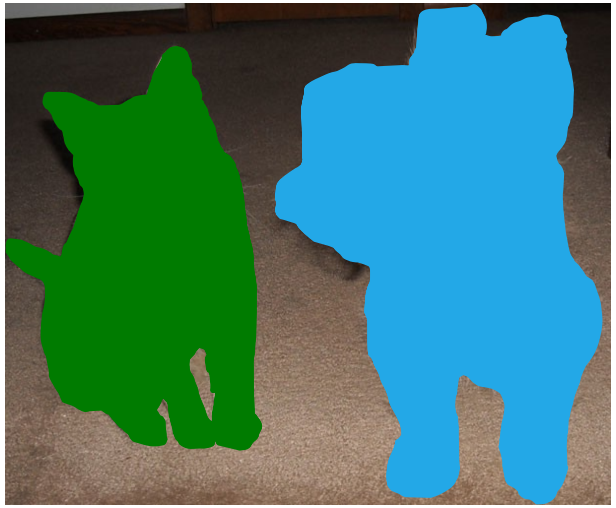


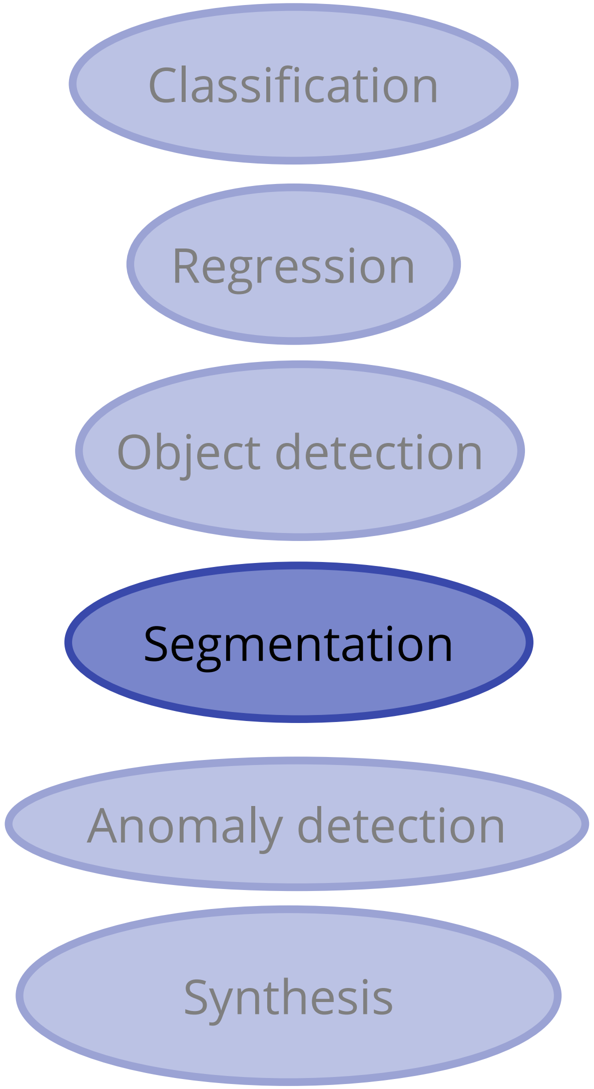


What tasks can ML learn?

Semantic segmentation: Assign class label to each pixel of an image based on what it is showing

Example:




Classification

Regression

Object detection

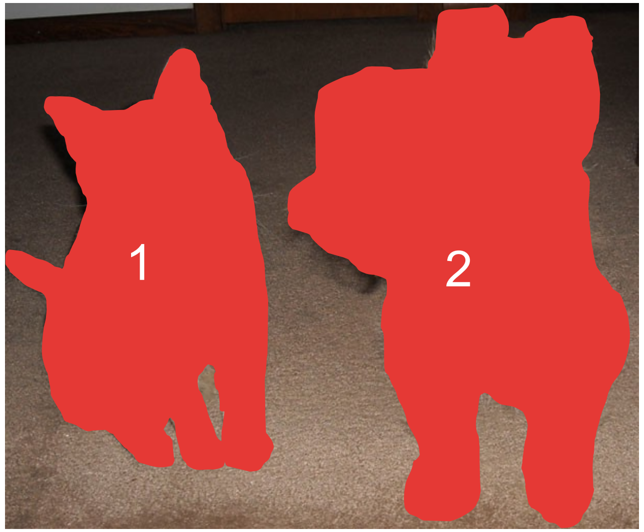
Segmentation

Anomaly detection

Synthesis

Instance segmentation: Assign class label to each pixel of an image based on what it is showing and discriminate different instances of the class

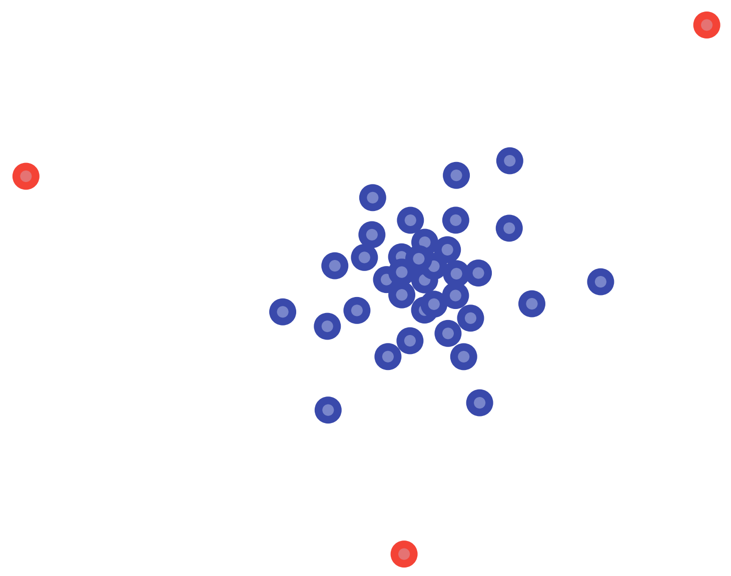
Example:



What tasks can ML learn?

Anomaly detection:
Identify anomalous or unusual data points within a data set

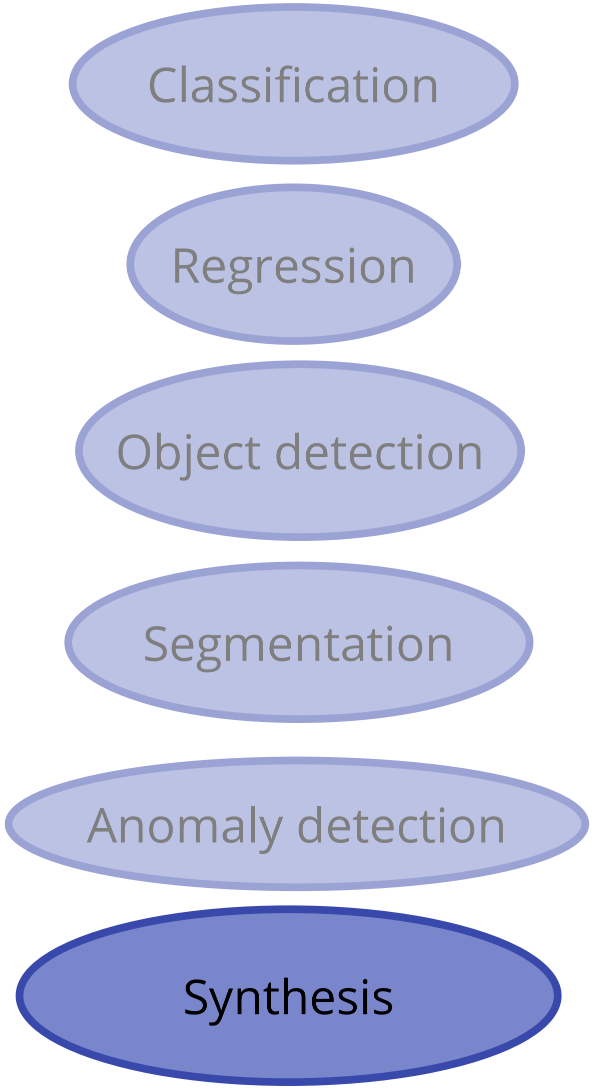
- Classification
- Regression
- Object detection
- Segmentation
- **Anomaly detection**
- Synthesis

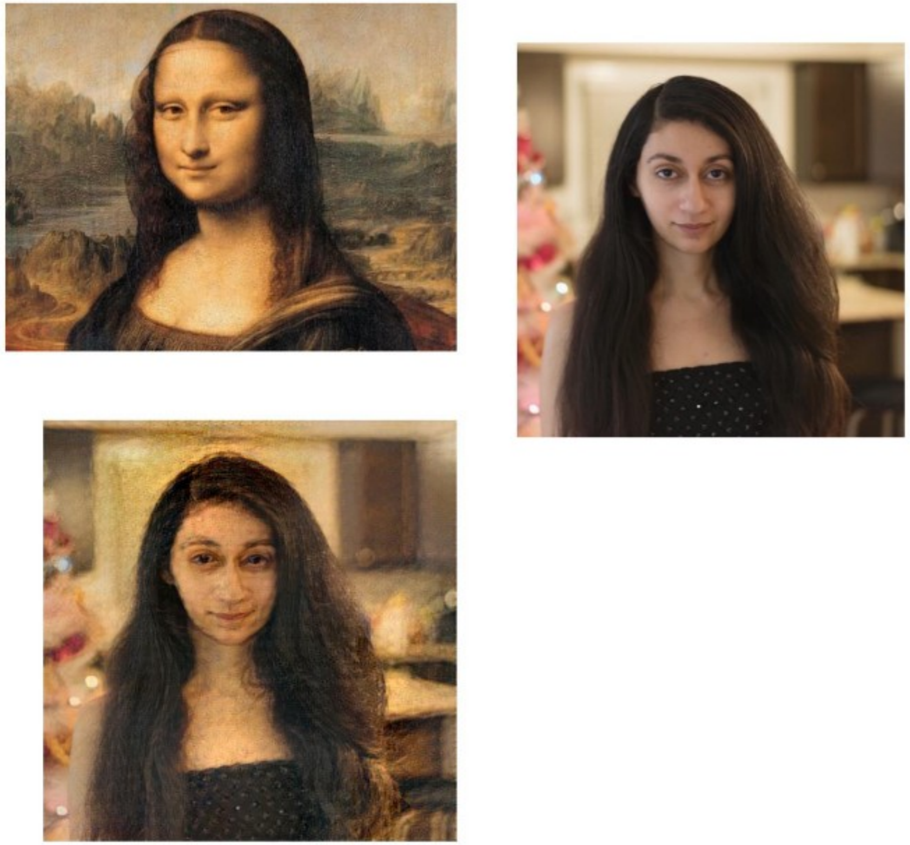


What tasks can ML learn?

Synthesis:
Generate new data points based on a learned distribution

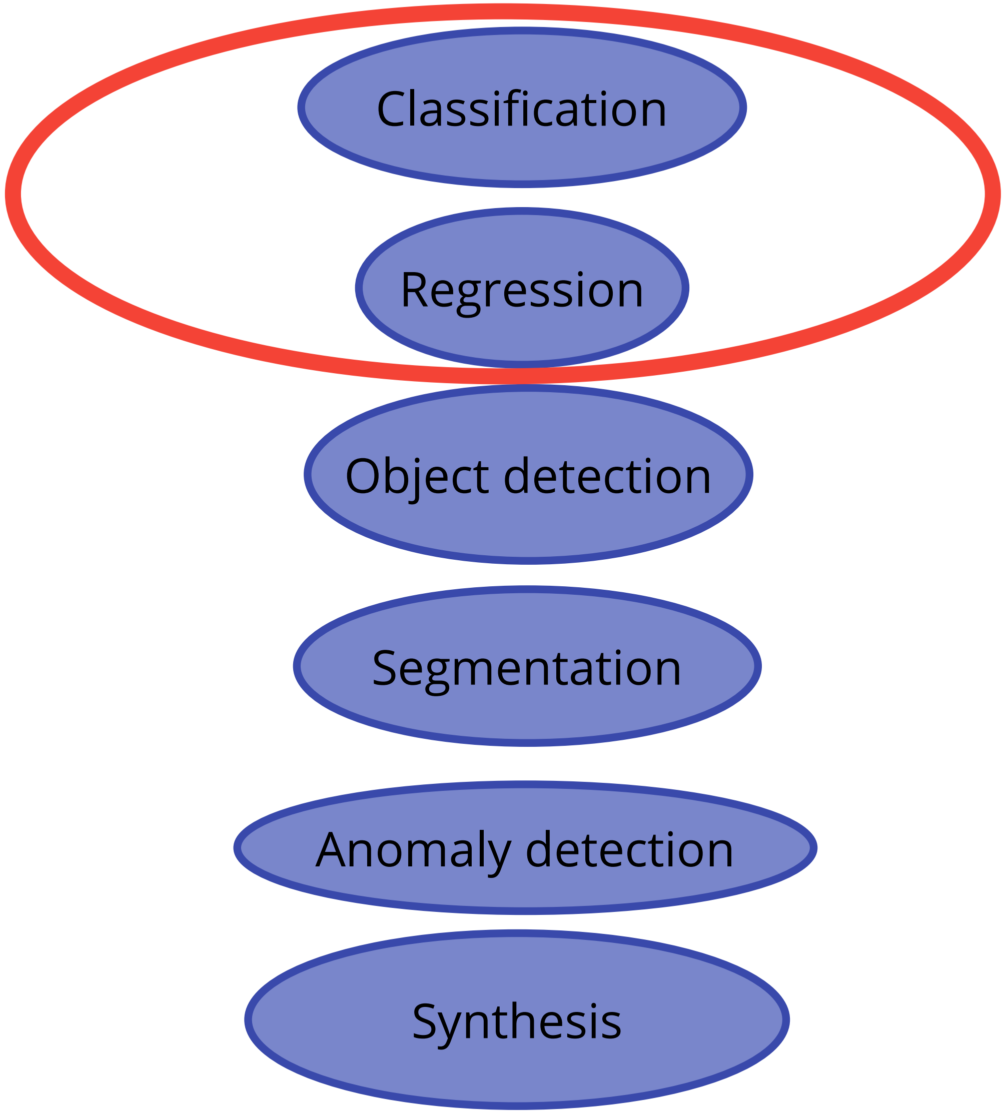
StyleGAN2 (Karras et al. 2020)





Style Transfer (Gatys et al. 2016)

What tasks can ML learn?



Classification

Regression

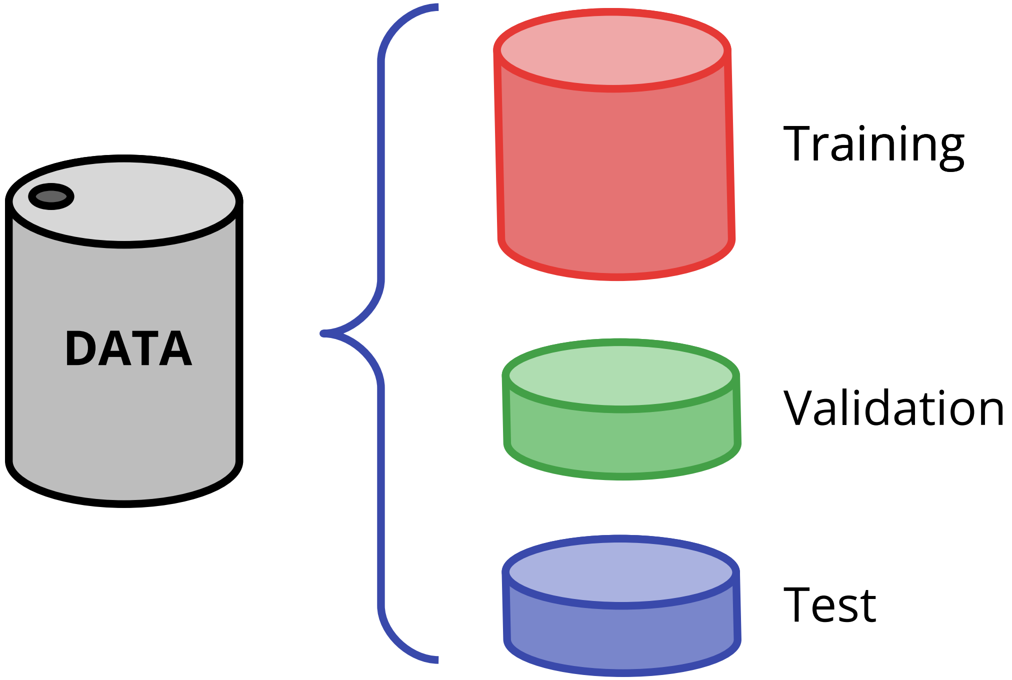
Object detection

Segmentation

Anomaly detection

Synthesis

Supervised learning concepts

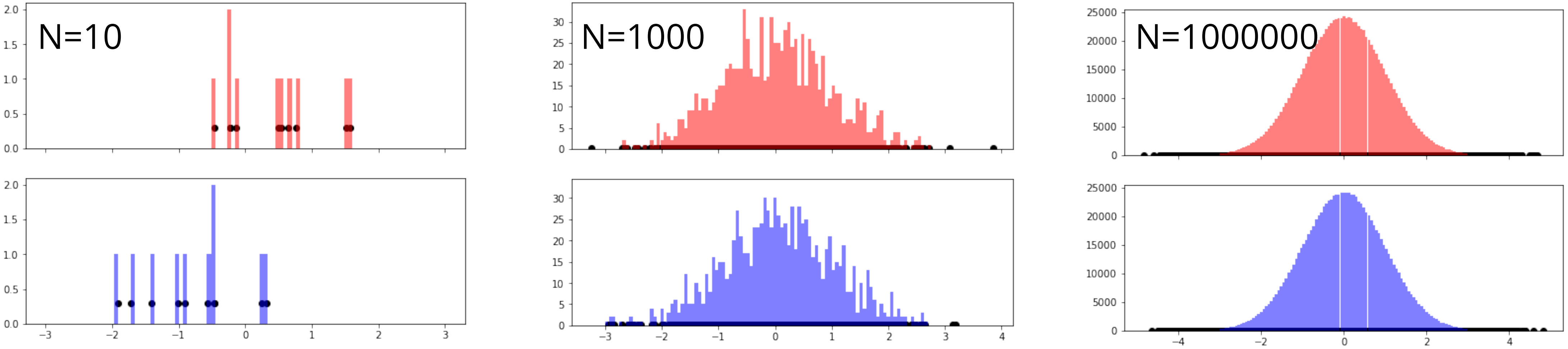


Independent and identically distributed (iid) data

iid is a core concept of ML. When running an ML model on previously unseen data, we implicitly assume that the unseen (new) data and the already seen (training) data are **iid**, i.e., the indiviual samples in both data sets are *produced by the same data generation process*.

This does not imply that the seen and unseen data sets are identical!

Example: sampling from a Normal distribution

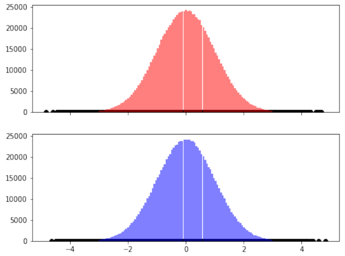


Lesson here: For small sample sizes, data sets that are iid may still differ significantly.

Real data sets

ML models are trained on existing data sets (seen data) and will be evaluated/applied to a new, previously unseen data set. Since **real data sets have a limited extent** (size), these distributions will look different, despite their iid nature.

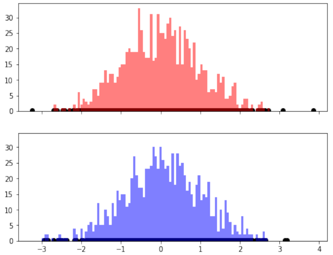
Ideally, the two distributions should be very similar:



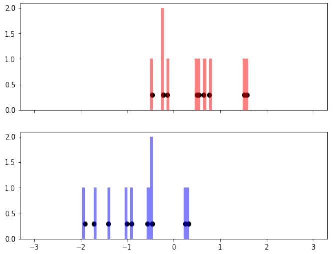
(seen)

(unseen)

But in real life, they tend to look more like this



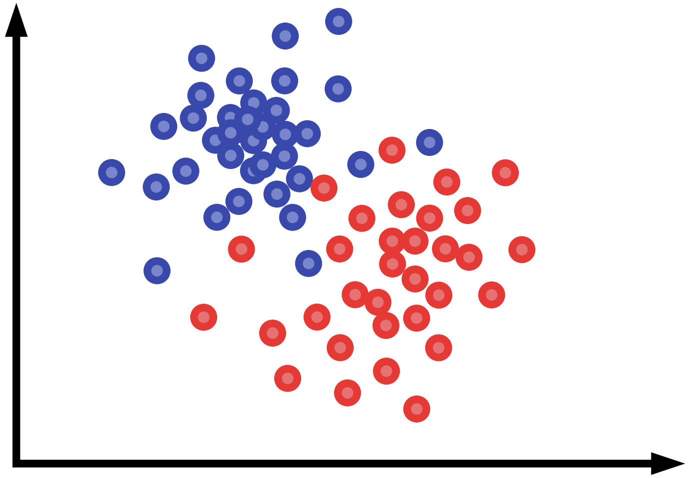
or even this



Successful training on one data set does not imply good performance on unseen data
→ the model has to **generalize** well by preventing **overfitting**

Generalization, regularization, overfitting and underfitting

Consider the following data set on which we train a ML model to classify two distinct class:

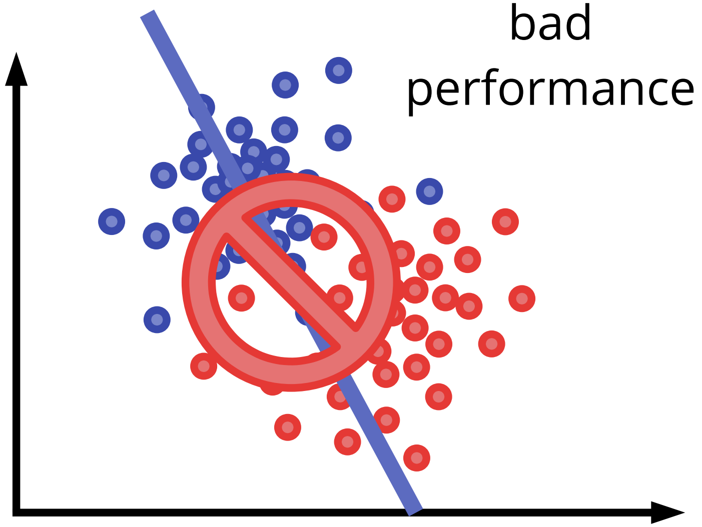


What would be a good decision boundary between the two classes?

The **decision boundary** separates the different classes as learned by the trained model.

Generalization, regularization, overfitting and underfitting

Consider the following data set on which we train a ML model to classify two distinct class:

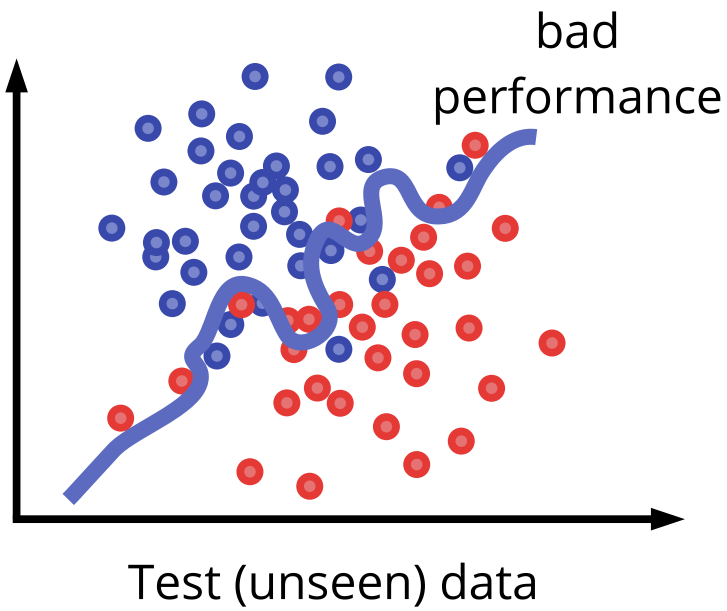


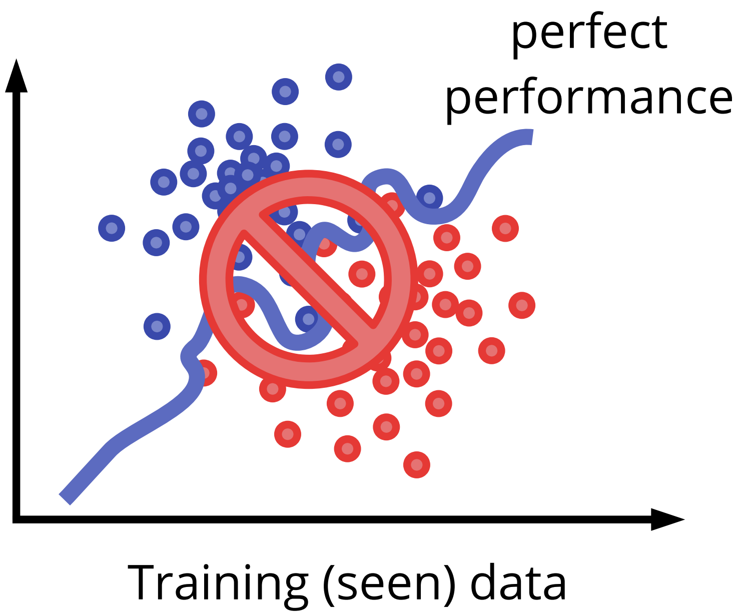
This would not be a good decision boundary as it barely allows to distinguish the two classes.

This model clearly **underfits** the data and leads to **poor performance**.

Generalization, regularization, overfitting and underfitting

Consider the following data set on which we train a ML model to classify two distinct class:





This decision boundary perfectly delineates the two classes in our data set.

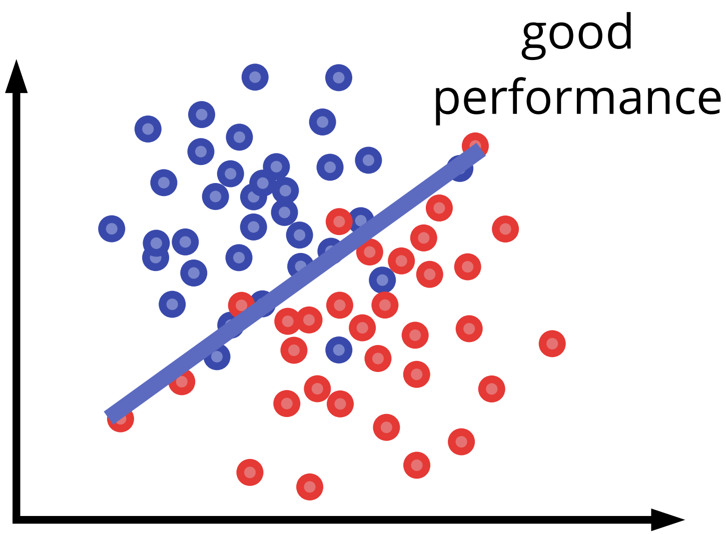
Is this good?

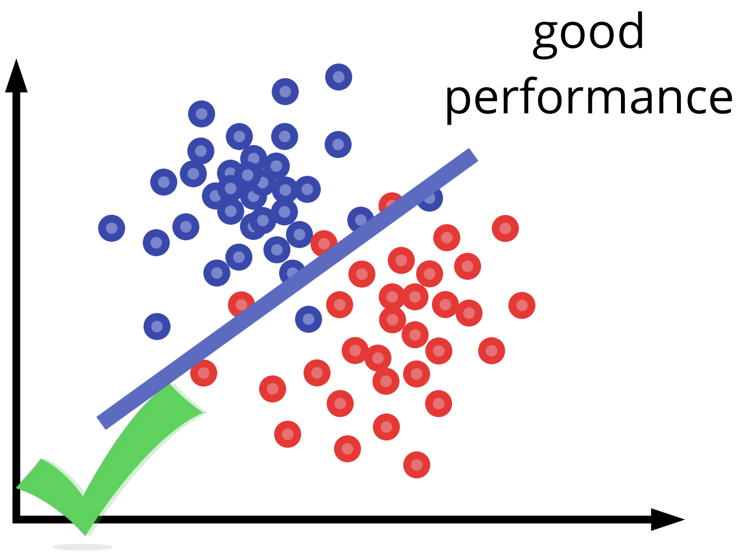
No, see what happens if we apply the model to previously unseen data: while it seems to perform perfectly on the seen data, it performs much worse on the unseen data.

This model is **overfitting**: it memorizes the structure of the training (seen) data and as a result **generalizes badly** on the overall data distribution.
We can improve its performance through **regularization** methods.

Generalization, regularization, overfitting and underfitting

Consider the following data set on which we train a ML model to classify two distinct class:





This decision boundary leads to **equally good performance on both data sets**.

The model therefore **generalizes well** on previously unseen data.

This is the desired situation as it provides you a realistic view on the expected performance of your model.

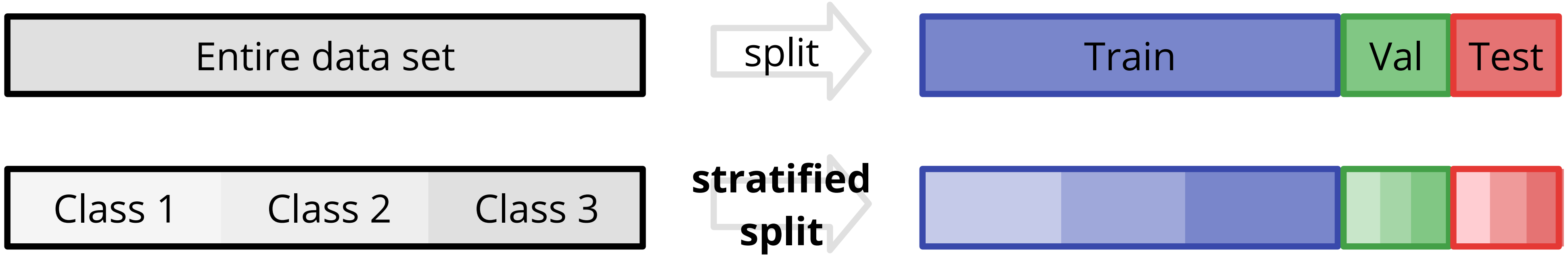
How can we check how well the model generalizes?

Train/Validation/Test splits

How can we check how well the model generalizes? We synthesize our own unseen data set.

In supervised learning, an existing data set is typically randomly split into three parts:

- **Training** data set – the model is trained exclusively on this data set
- **Validation** data set – this data set is used to tune our model's hyperparameter
- **Test** data set – this data set is used to evaluate the model's performance on unseen data



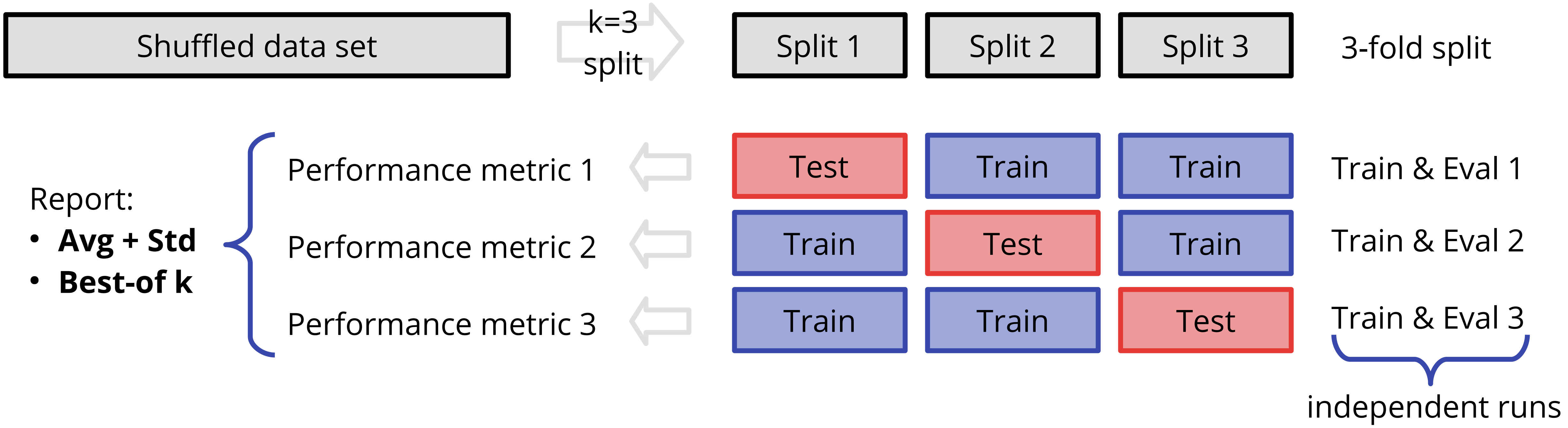
Typical ratios: 0.7/0.15/0.15

Preserve class fractions in split data sets.

k-fold cross-validation

Often times the data set available to you is not large enough to perform a meaningful split.

You can “recycle” your data by using cross-validation to get a better estimate of your model performance.



Shuffled data set

k=3 split

Split 1	Split 2	Split 3	3-fold split
Test	Train	Train	Train & Eval 1
Train	Test	Train	Train & Eval 2
Train	Train	Test	Train & Eval 3

independent runs

Performance metric 1

Performance metric 2

Performance metric 3

Report:

- **Avg + Std**
- **Best-of k**

Note: Keep in mind that cross-validation will not improve your model performance; it will simply give you a more reliable estimate of its performance.

General supervised learning pipeline

1) Feature engineering: raw data → features

2) Data scaling

3) Data splitting → training, validation, test data

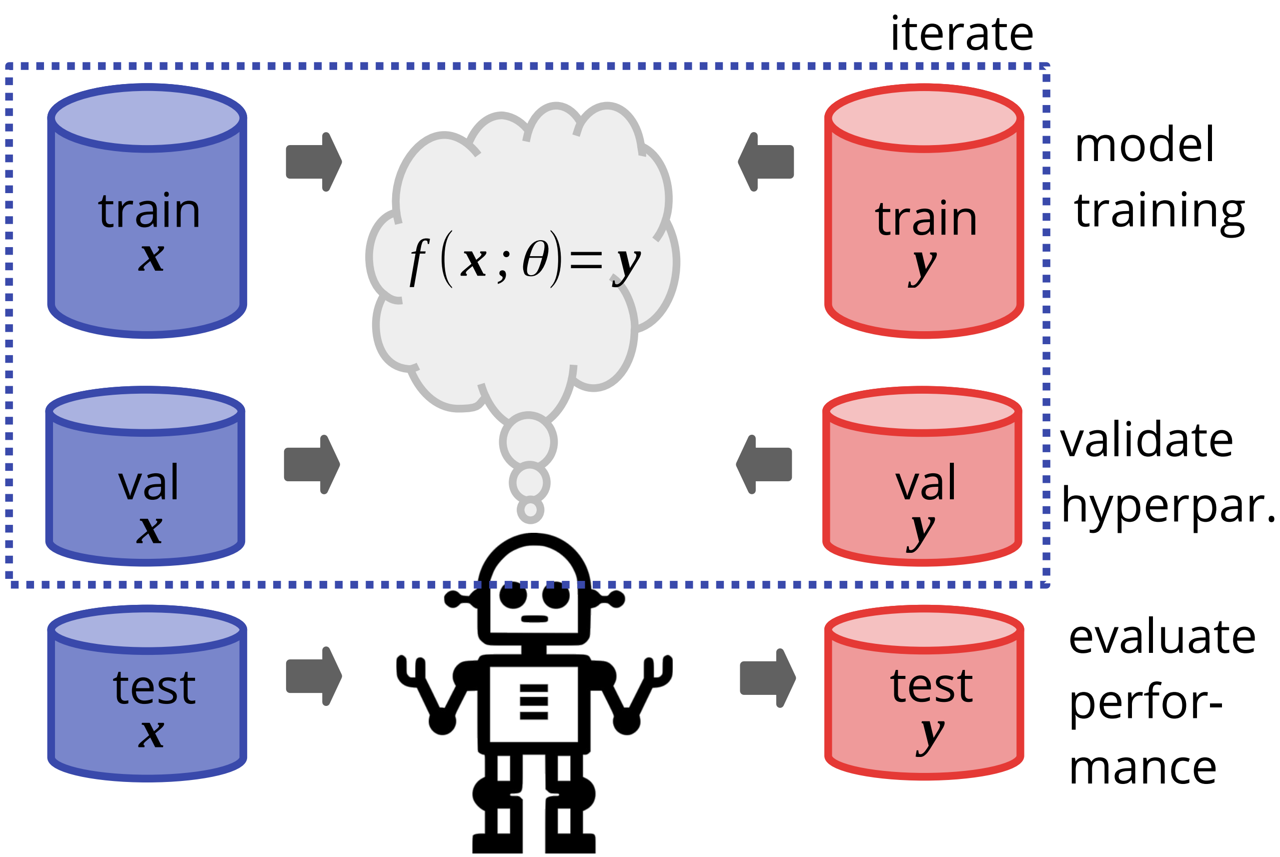
4) Define hyperparameters

5) Train model on training data for fixed hyperparameters

6) Evaluate model on validation data

7) Repeat 4) to 6) until performance on validation data maximized

8) Evaluate trained model on test data
→ report test data performance



Benchmarking and metrics

How do we measure the performance of our model?

Benchmarking refers to the process of quantitatively assessing your ML model's performance.

Performance is measured based on pre-defined metrics; a **metric** can be thought of as a measure for how well an ML model performs on a specific task and data set.

Examples:

Which athlete is best?

Based on…

- speed
- strength
- number of victories
- income

Which company is successful?

Contributing factors:

- revenue
- overall value
- number of employees
- annual CO2 emissions

Medical diagnosis

What is most important?

- correctness of diagnosis
- minimizing failures
- patient's comfort
- cost

metrics

Different ML tasks require different metrics

Regression task:

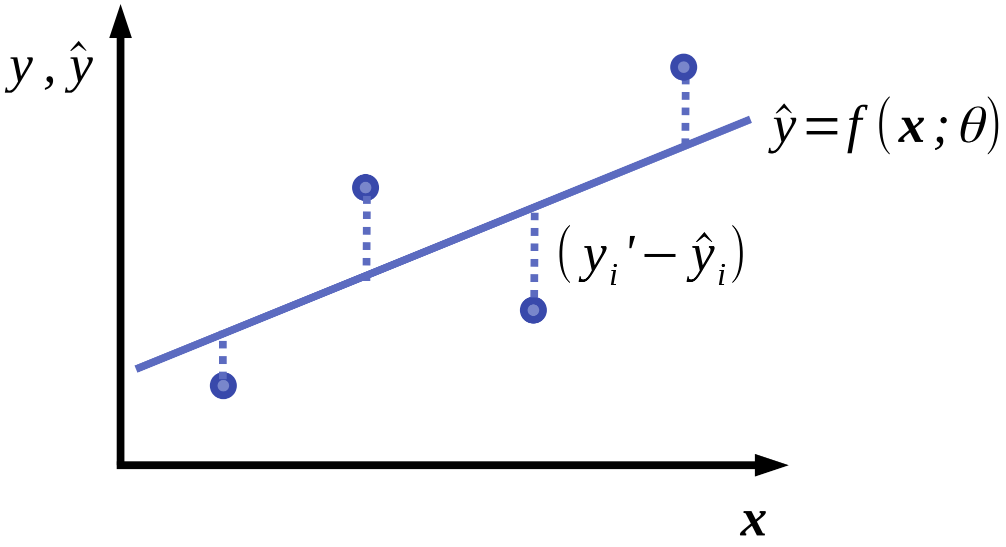
Input data:	$\boldsymbol{x}_i, i \in \{1 \ldots N\}$
Target ground-truth:	y_i'
Target prediction:	$\hat{y}_i = f(\boldsymbol{x}_i; \theta)$

MAE (**M**ean **A**bsolute **E**rror)

$$MAE = \frac{1}{N} \sum_i^N |y_i' - \hat{y}_i|$$

RMSE (**R**oot **M**ean **S**quare **E**rror)

$$RMSE = \sqrt{\frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2}$$



Intuition: by how much deviates your model prediction from the ground-truth on average.

Different ML tasks require different metrics

(Binary) Classification:

Accuracy = (TP + TN) / (TN + TP + FP + FN)
What is the overall fraction of correct (positive and negative) predictions?

Precision = TP / (TP + FP) (quantifies “correctness”)
What fraction of our positive predictions is truly positive?

Recall = TP / (TP + FN) (quantifies “completeness”)
What fraction of actual positives has been identified?

		Ground-Truth	
		positive	negative
Prediction	positive	**True positive**	**False positive**
	negative	**False negative**	**True negative**

Different ML tasks require different metrics

(Binary) Classification:

Accuracy = (TP + TN) / (TN + TP + FP + FN)
What is the overall fraction of correct (positive and negative) predictions?

Precision = TP / (TP + FP) (quantifies “correctness”)
What fraction of our positive predictions is truly positive?

Recall = TP / (TP + FN) (quantifies “completeness”)
What fraction of actual positives has been identified?

Example: Is there a dog in the image?

accuracy = 0.95:
we correctly identified 95% of all dogs in the images

precision = 0.95:
95% of the dogs we predicted are actual dogs

recall = 0.95:
we correctly found 95% of the dogs that are in the images

Different ML tasks require different metrics

(Binary) Classification:

Accuracy = (TP + TN) / (TN + TP + FP + FN)

Requires somewhat balanced classes

Precision = TP / (TP + FP) (quantifies “correctness”)
Less susceptible to imbalance.

Recall = TP / (TP + FN) (quantifies “completeness”)
Less susceptible to imbalance.

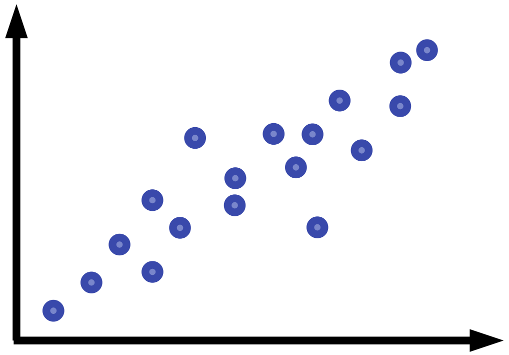
Class imbalance is a real issue!
Example: Will this asteroid impact Earth?

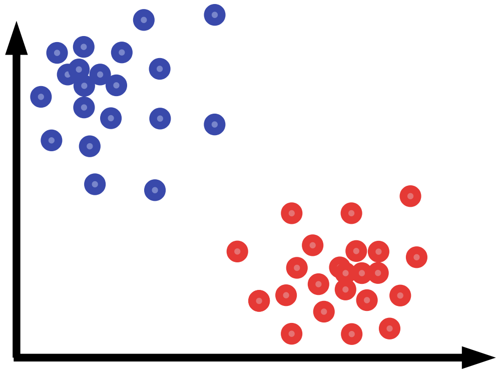
If our model always predicts *False*, we can easily reach 99.9% accuracy, although the model does nothing useful.

Low precision means that we issue some false alarms – this is something we can deal with.

Low recall means that we miss some asteroids that are about to impact, which is not exactly what we want. Therefore, **recall** is the really important metric here and should be **maximized**.

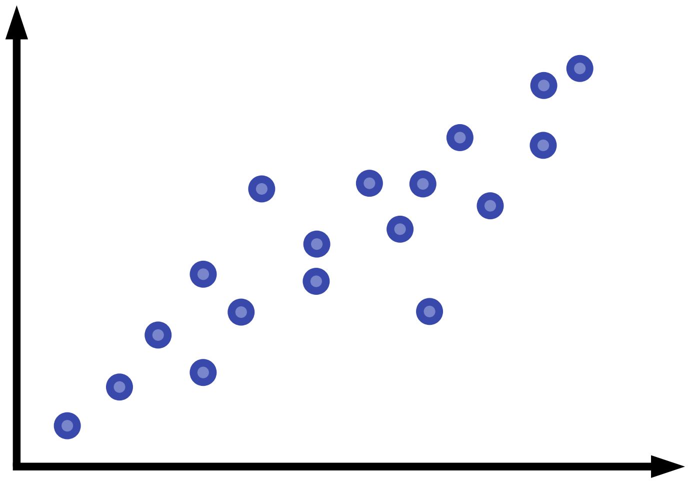
Linear models



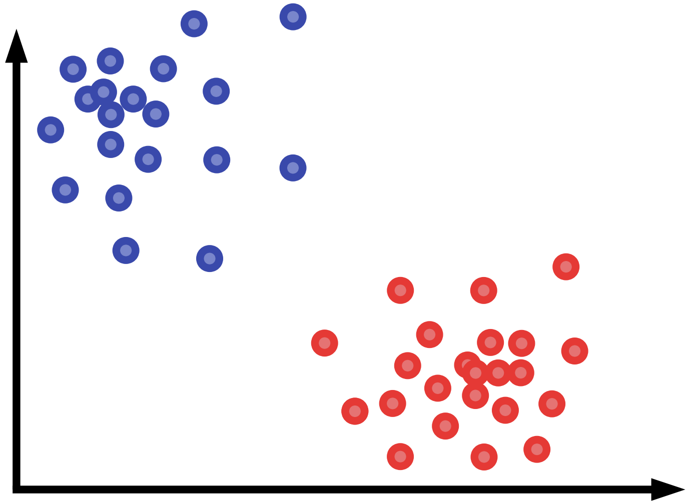


Linear models

Linear models assume linearity in the underlying data. They are rather simple but convey many of the concepts utilized in other, more complex models.



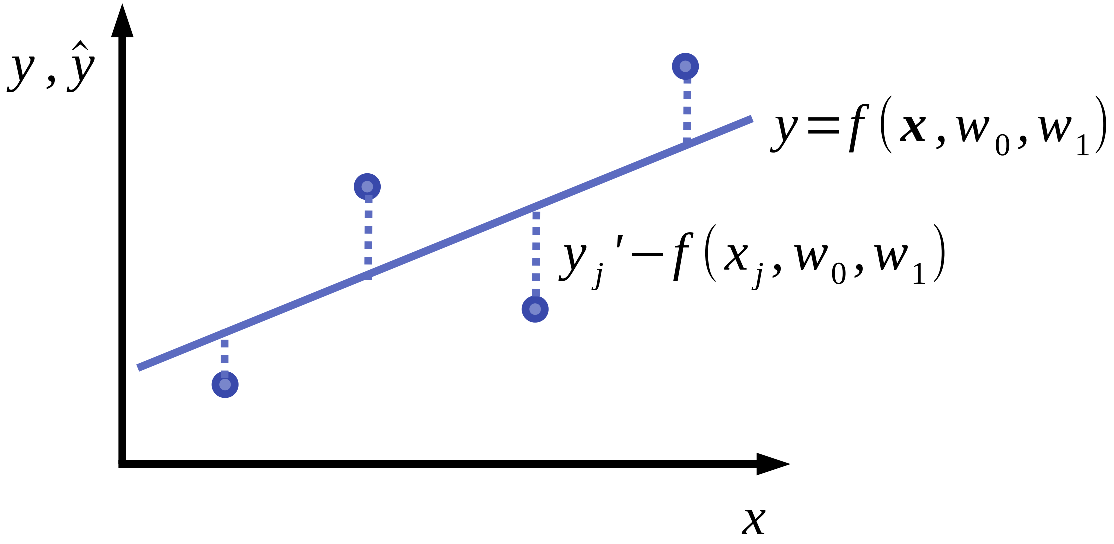
Linear regression



Linear classification

Linear regression (univariate)

Find weights w_0 and w_1 so that the linear function $f(x)=w_1 x+w_o$ with input x and output y best fits the data containing ground-truth values y'.



How can we learn w_0 and w_1 from data?

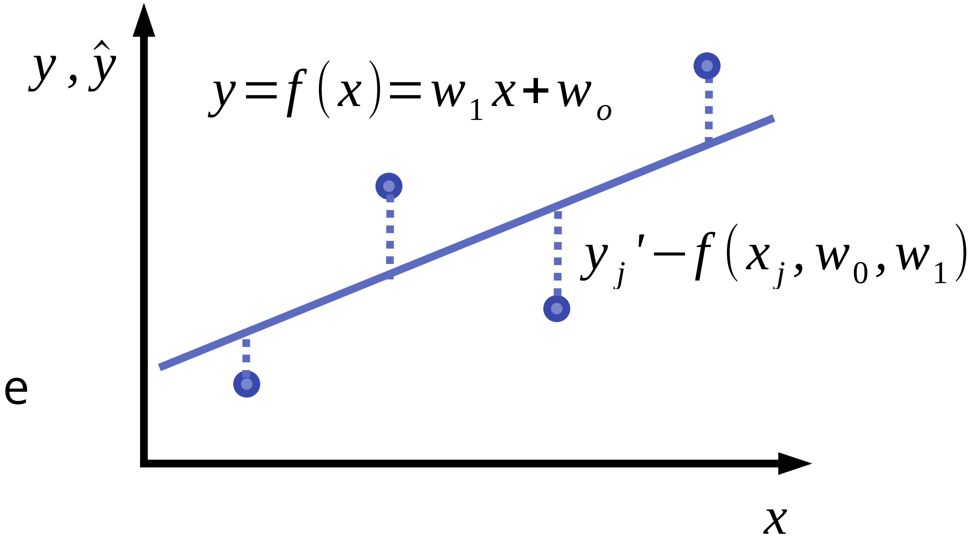
Linear regression (univariate) – Least squares fitting

Idea: minimize squared errors of prediction with respect to ground truth for each data point:

for data point j: $[y_j' - f(x_j, w_0, w_1)]^2$

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N (y_j' - f(x_j, w_0, w_1))^2 = \sum_j^N (y_j' - (w_1 x_j + w_0))^2$$



We can find the best-fit model parameters, by **minimizing** the Loss function with respect to those two model parameters:

$$\frac{\partial L}{\partial w_0} = 0 \qquad \frac{\partial L}{\partial w_1} = 0$$

→ closed-form expressions for best-fit w_0 and w_1.

Least-squares + linear model function: the resulting minimum of the Loss function is **global**, i.e., the solution is by default the best-possible solution!

Linear classifier (two-dimensional case)

Linear functions can also be used as a classifier if the data are linearly separable.

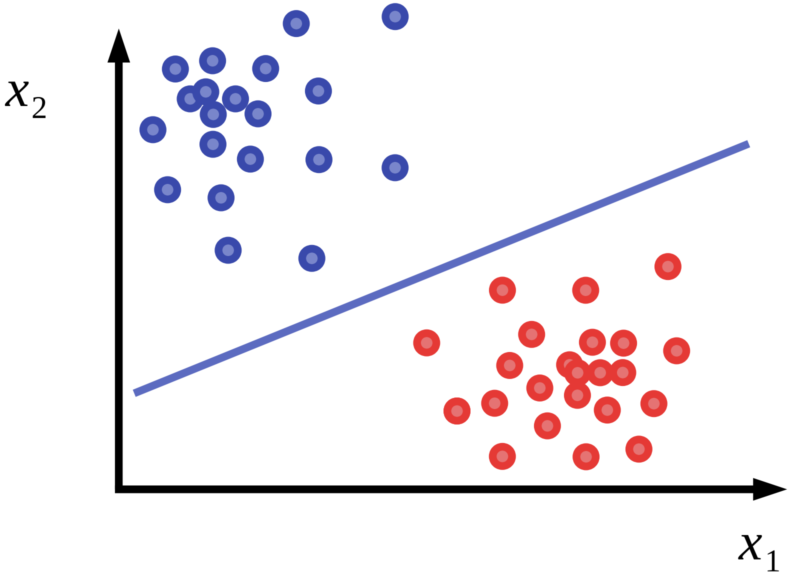
Define decision boundary $f(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}\,\boldsymbol{x} = w_0 + w_1 x_1 + w_2 x_2$ such that:

Class 1: $f(\boldsymbol{x}, \boldsymbol{w}) \geq 0$

Class 0: $f(\boldsymbol{x}, \boldsymbol{w}) < 0$

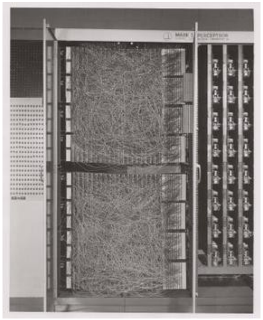
We can define class assignments through a threshold function:

$$\bar{f}(\boldsymbol{x}, \boldsymbol{w}) = \begin{cases} 1 \ \text{if} \ f(\boldsymbol{x}, \boldsymbol{w}) \geq 0 \\ 0 \ \text{if} \ f(\boldsymbol{x}, \boldsymbol{w}) < 0 \end{cases}$$



How can we learn $\boldsymbol{w}$ from data?

Linear classifier (two-dimensional case) – Perceptron learning rule

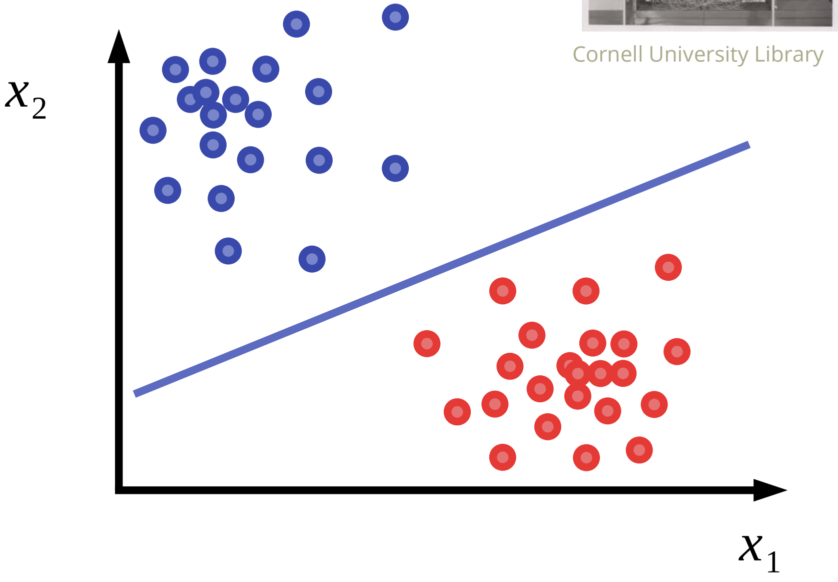


Cornell University Library

We define the following algorithm as the **Perceptron learning rule**:

We consider each data point, consisting of $\mathbf{x}$ and ground-truth label y' and check whether the prediction from $\bar{f}(\mathbf{x},\mathbf{w})$ is correct, or not. If…

- $\bar{f}(\mathbf{x},\mathbf{w})=y$, then do nothing.
- $\bar{f}(\mathbf{x},\mathbf{w})=0$ but $y'=1$, then increase w_i if $x_i \geq 0$, or vice versa.
- $\bar{f}(\mathbf{x},\mathbf{w})=1$ but $y'=0$, then decrease w_i if $x_i \geq 0$, or vice versa.



Weights are adjusted by a step size that is called the **learning rate**. By iteratively running this algorithm over your training data multiple times, the weights can be learned so that the model performs properly.

Linear models – pros and cons

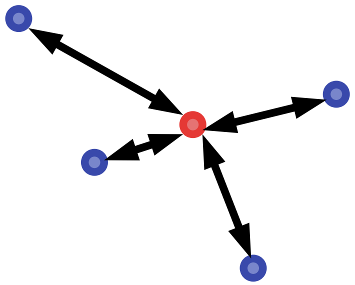
Pros:

- Easy to understand and implement; resource efficient even for large and sparse data sets.
- Least squares method always provides best-fit result, if the data are appropriate.
- Good interpretability due to linear nature of the model.

Cons:

- Very limited in applicability: data distribution must be linear or linearly separable.
- Susceptible to overfitting if not combined with regularizer.

Nearest-Neighbor models

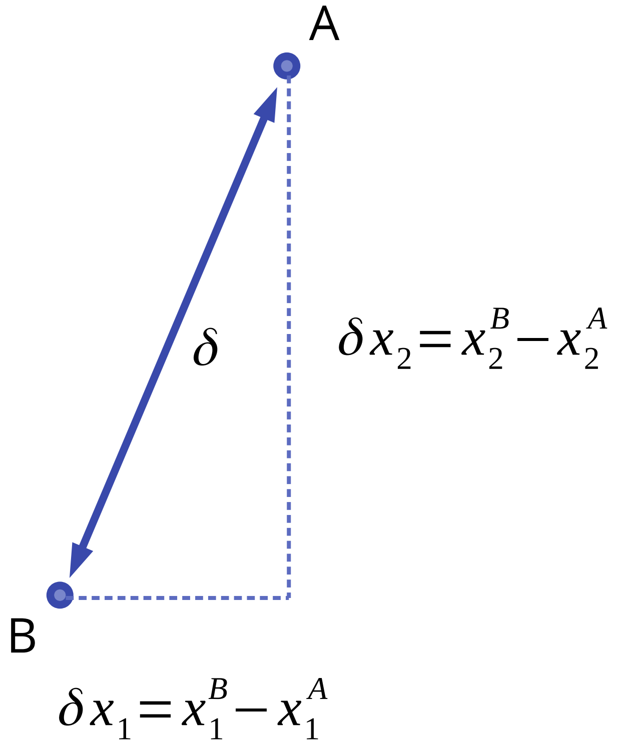


Nearest neighbor models

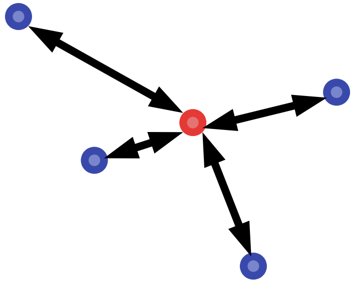
Nearest neighbor models are **non-parametric** and simply rely on **distances** between data points.

Distances can be defined as metrics. A common distance metric is the **Euclidean distance** between two data points $A=(x_1^A, x_2^A)$ and $B=(x_1^B, x_2^B)$:

$$\delta=\sqrt{\delta x_1^2+\delta x_2^2}=\sqrt{(x_1^B-x_1^A)^2+(x_2^B-x_2^A)^2}$$



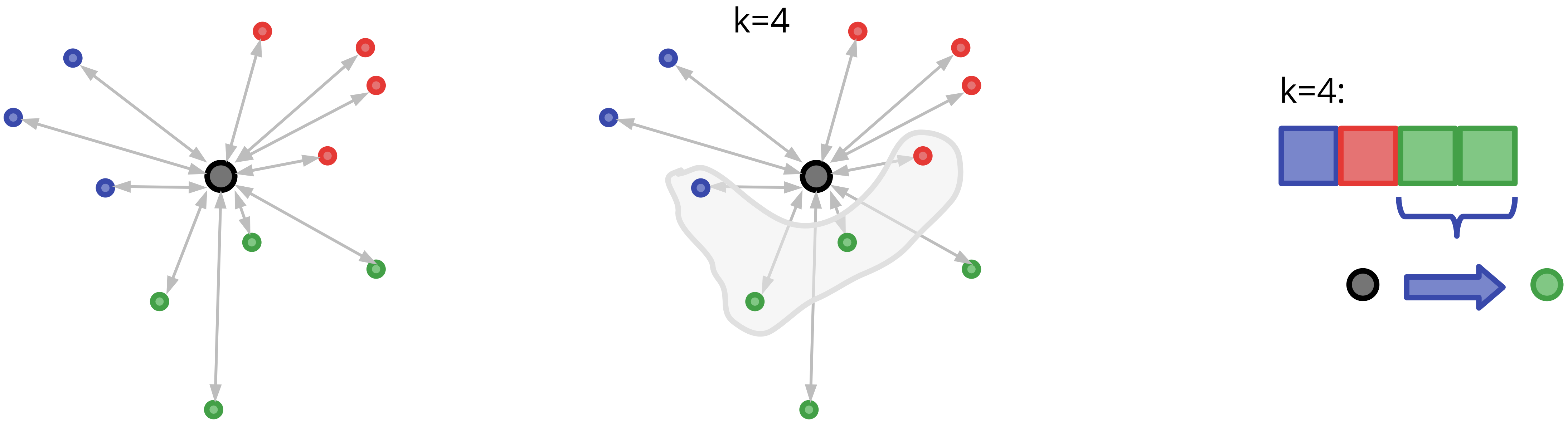
Nearest neighbor methods utilize distances between datapoints for **classification** and **regression** tasks.



k-nearest neighbor classification

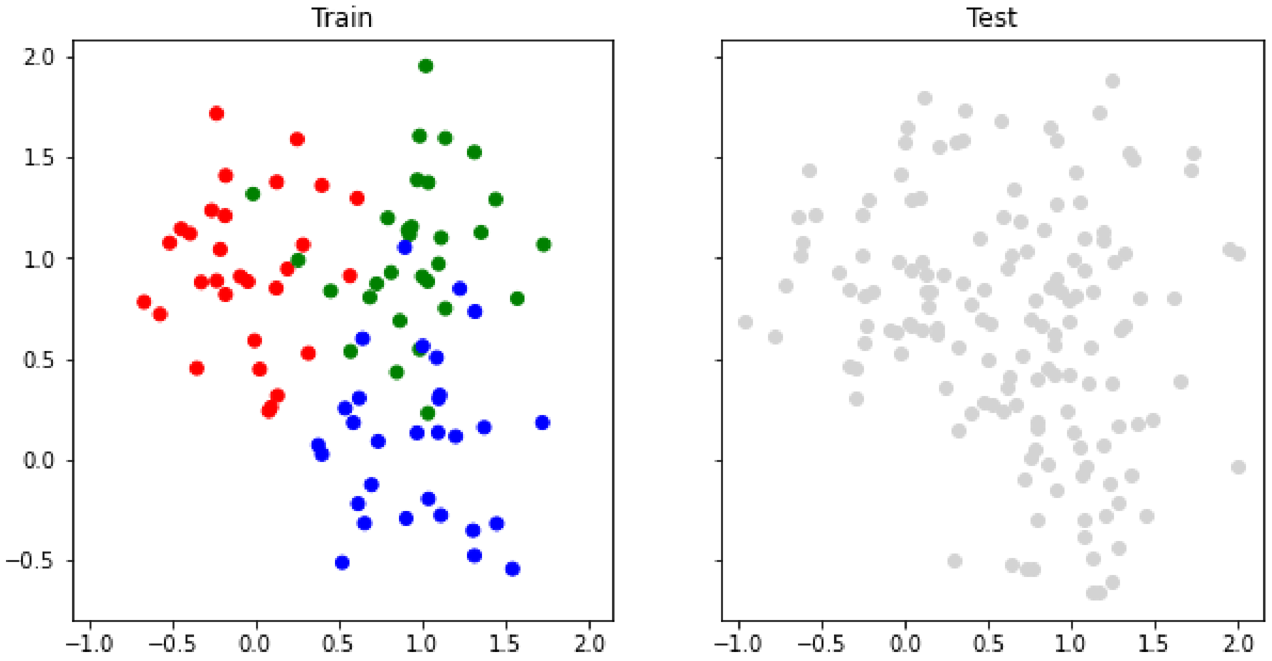
k-nearest neighbor (knn) classifiers predict class affiliation of an unseen data point based on **majority voting** of its ***k* nearest neighbors** in a seen data set with ground-truth labels.

knn models are not trained in the general sense. Instead, the distance of each unseen data point from all seen data points is calculated.



compute distances identify *k* nearest neighbors class assignment

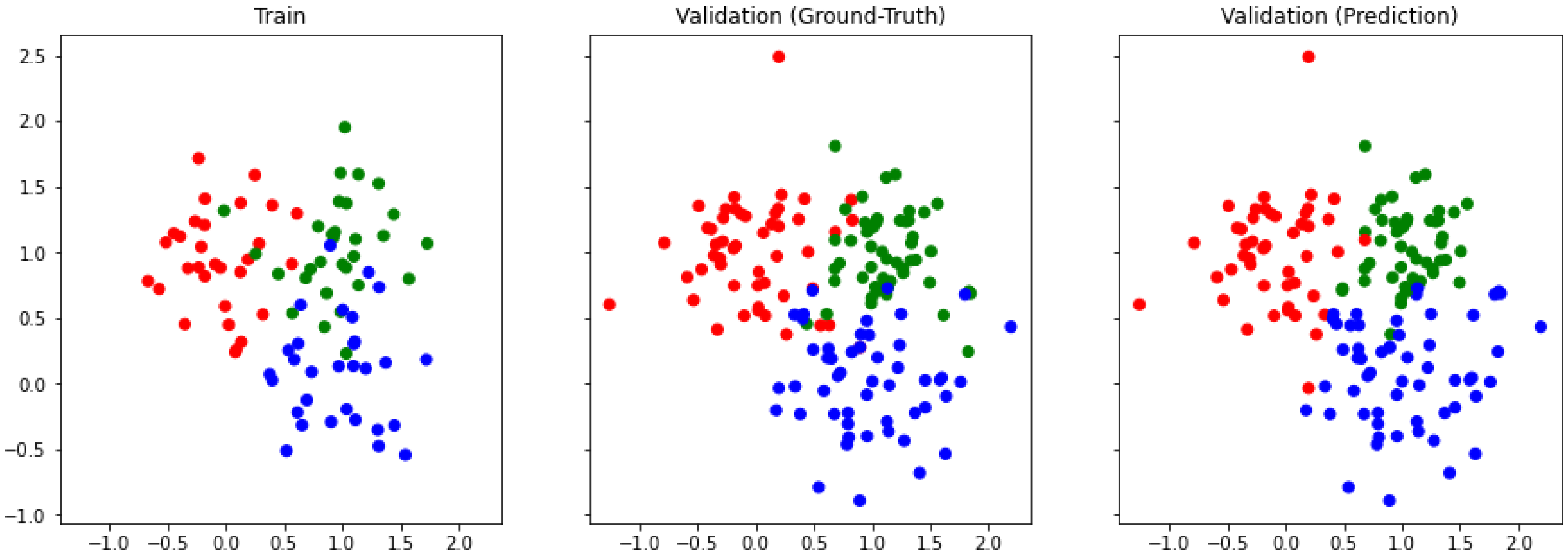
k-nearest neighbor classification - Example



3 overlapping clusters

How well can knn classify our test data set?

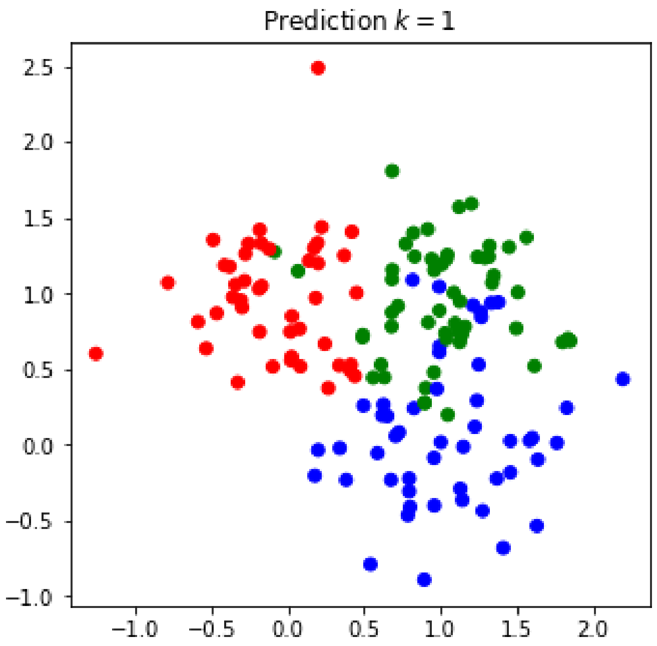
k-nearest neighbor classification - Example



k=5: accuracy=0.867

Hyperparameter *k* has an impact on how well the model generalizes to unseen data: perform a **hyperparameter search**!

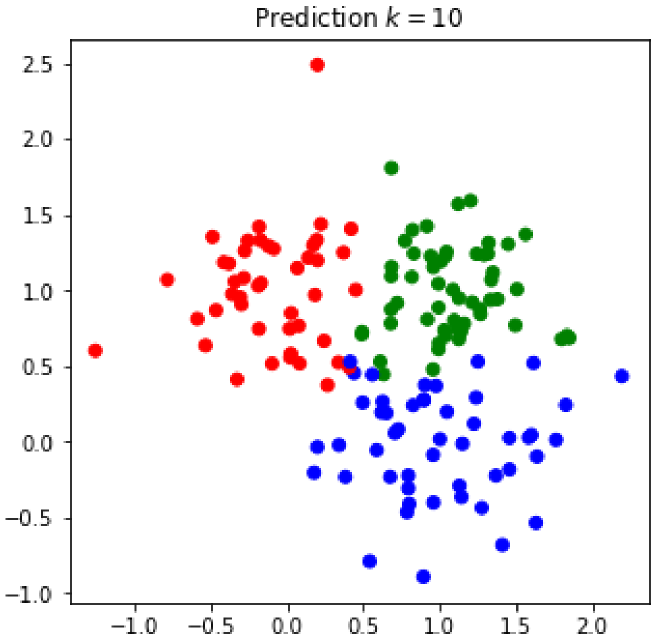
k-nearest neighbor classification - Example



k=1

accuracy$_{\text{val}}$=0.800

Overfitting!

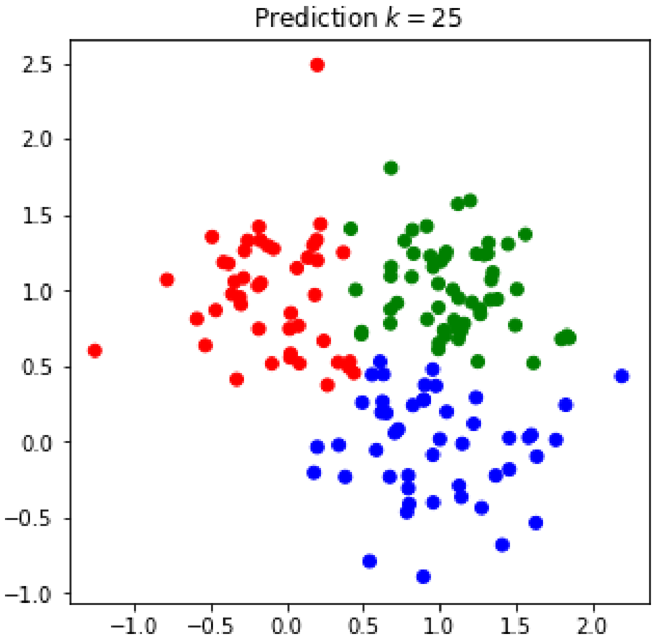


k=10

accuracy$_{\text{val}}$=0.893

accuracy$_{\text{test}}$=0.880

Best performance



k=25

accuracy$_{\text{val}}$=0.873

Underfitting!

k-nearest neighbor classification – pros and cons

Pros:

- Easy to understand, implement and results are highly interpretable.
- Non-parameteric
- Works reliably even with small data sets.

Cons:

- Calculating distances computationally intensive for large data sets.
- Performs poor on sparse data sets; prone to the **curse of dimensionality.**

Number of data points should grow exponentially with data dimensionality.

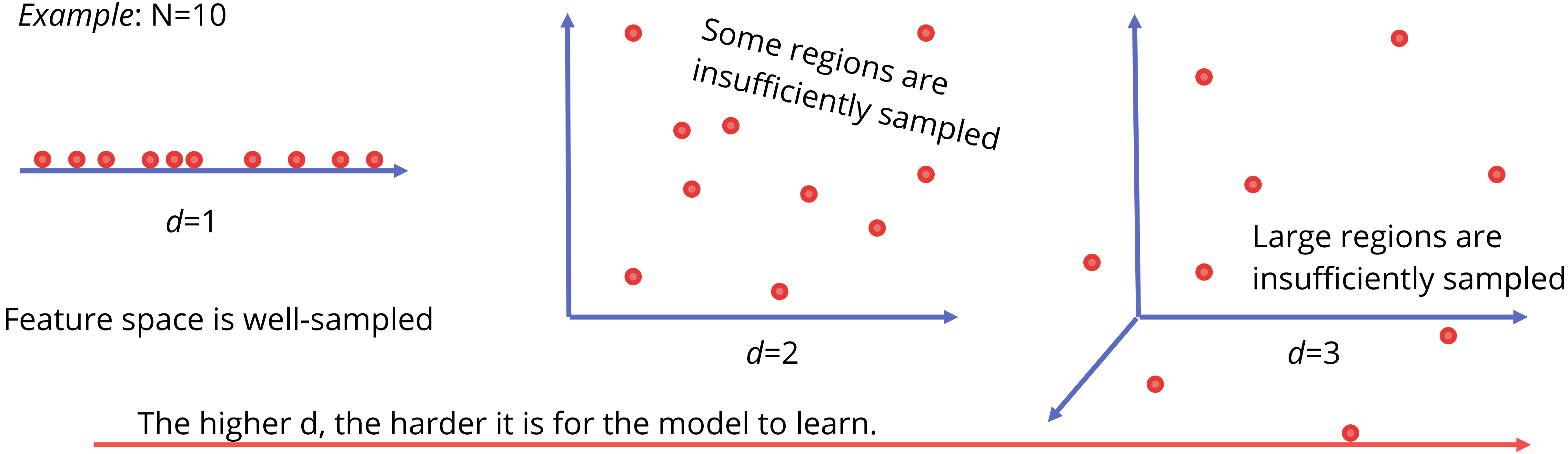
If parameter space is insufficiently sampled, the model does not have enough data points for training properly.

Bonus Slide: The Curse of Dimensionality

Datasets we deal with typically have a limited size, *N*. The number of features in our dataset, *d*, defines the dimensionality of the feature space; the volume of the feature space, *V*, grows exponentially with *d*.

Curse of Dimensionality: for large d (high-dimensional input data), N may be too small to sample V.

Example: N=10



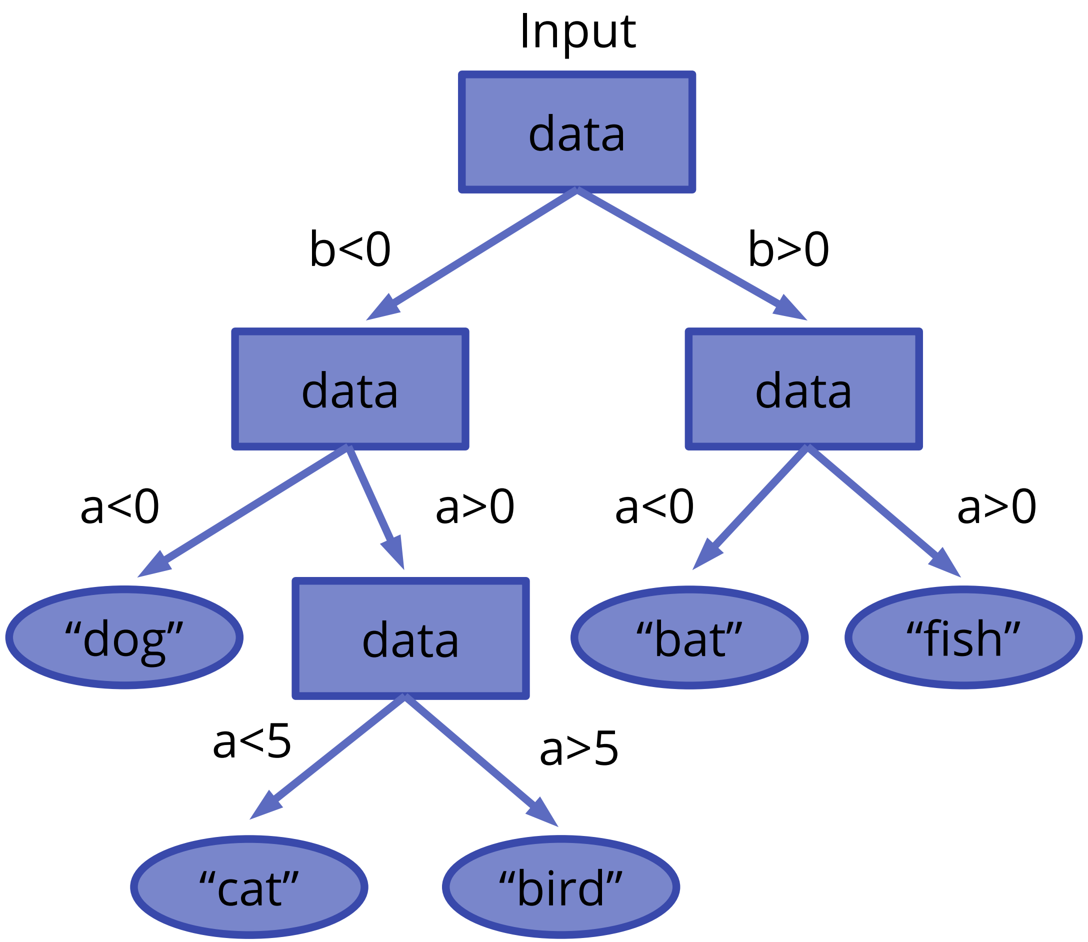
Feature space is well-sampled

Some regions are insufficiently sampled

Large regions are insufficiently sampled

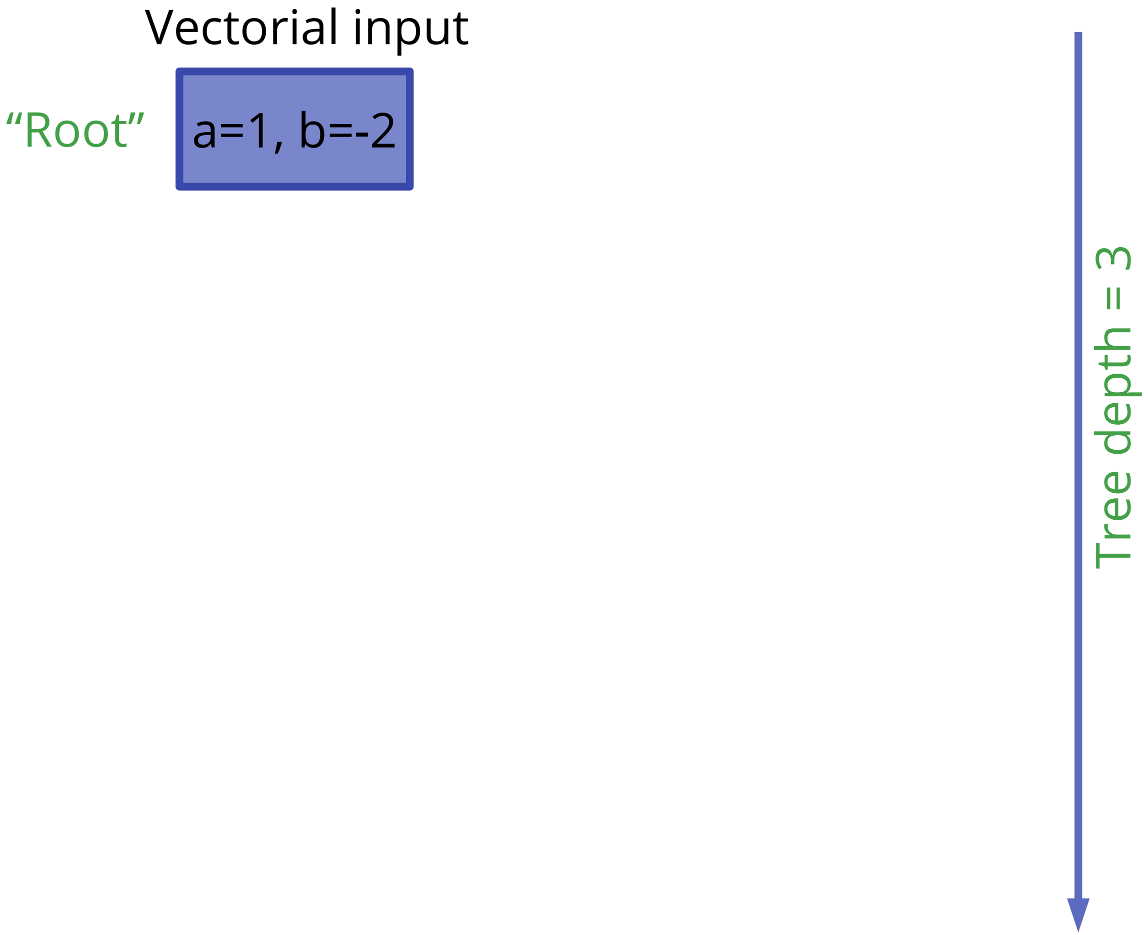
The higher d, the harder it is for the model to learn.

Tree-based models
(a high-level introduction)



Decision tree classification

A decision tree is a **rule-based structure** for prediction of scalar output from (potentially) multi-dimensional input data.



Vectorial input

“Root”

a=1, b=-2

Tree depth = 3

Tree properties (hyperparameters):

- Tree depth
- Number of leaves

How can the rules stored in the nodes be learned?

Decision tree learning

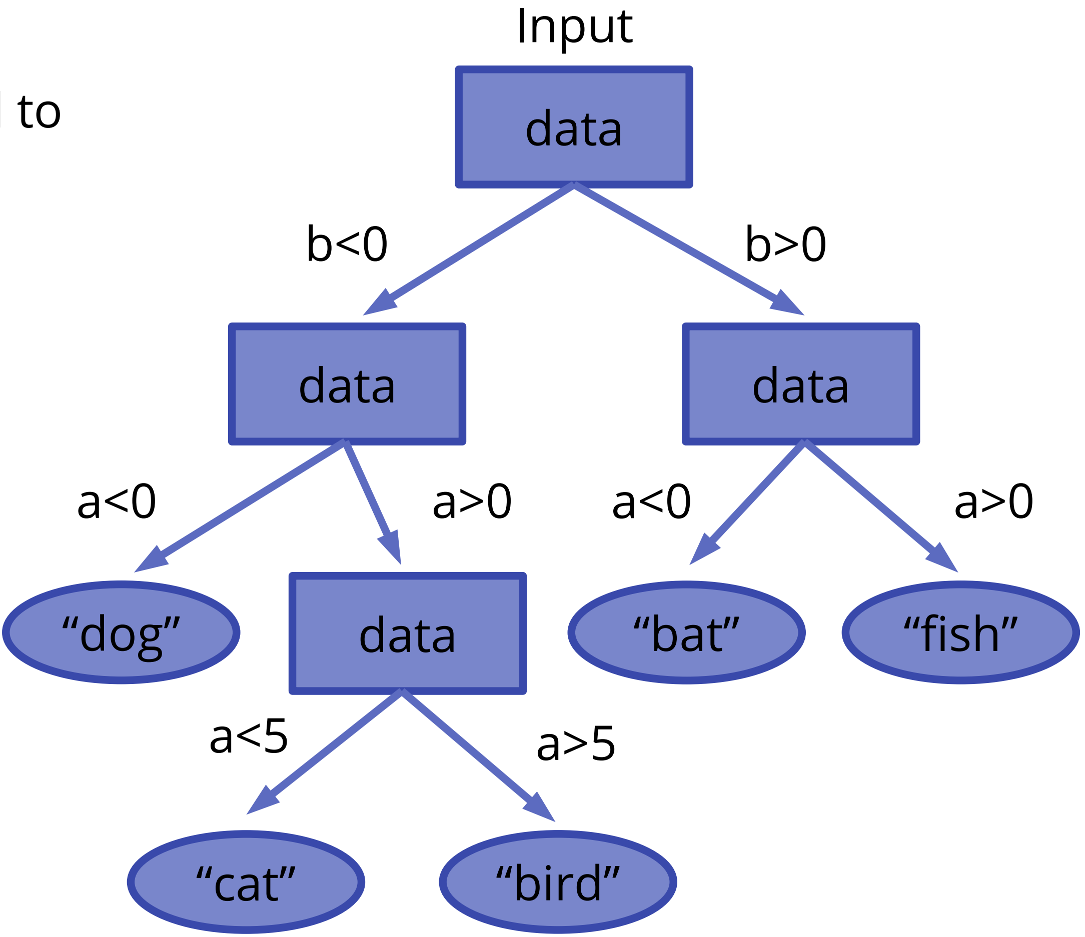
A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the "most important feature" (greedy)

2) Split the samples across this feature (divide)

3) If all samples of a branch are of the same class, create a leaf, unless number of leafs has been reached, and stop.

4) If not all samples of a branch are of the same class, recursively apply algorithm again to that branch until maximum depth reached.



Decision tree learning

A **greedy divide-and-conquer strategy** is adopted to train decision trees on a training data set in a recursive fashion:

1) Identify the “most important feature” (greedy)

2) Split the samples across this feature (divide)

3) If all samples of a branch are of the same class, create a leaf, unless number of leafs has been reached, and stop.

4) If not all samples of a branch are of the same class, recursively apply algorithm again to that branch until maximum depth reached.

But what is the “most important feature”?

Generally, it means that feature that makes the most difference to the classification of a single sample.

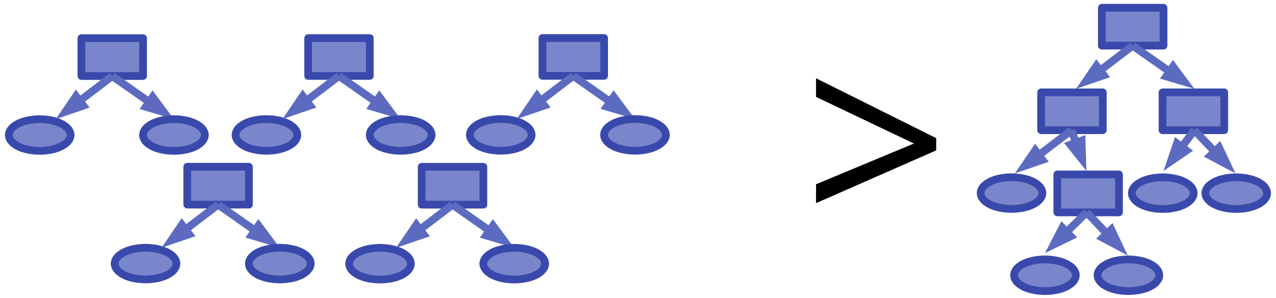
There different implementation of this definition, e.g., utilizing **information entropy** or other useful measures.

Random forests - decision trees as “weak learners”

Single decision trees typically generalize only to some extent due to their limited depth and size; they have **low model capacity**.

Ensemble methods help trees to perform much better: by combining a large number of decision trees and letting them make decisions in an averaged vote or majority vote, thereby **increasing capacity**.

Trees in a **random forest** are shallower than other decision tree models. The trees therefore act as “**weak learners**” that perform badly by themselves. However, combining a large number of weak learners performs much better than individual trees. The intuition behind is that weak learners “on average” compensate for their individual shortcomings.



Gradient-boosted tree-based models

Gradient-boosted tree-based models are random forests (decision tree ensembles) that are built successively in such a way that *every newly created tree compensates for the shortcomings of the previous trees*.

The term **gradient-boosting** refers to the fact that new base learners (individual decision trees) are fitted to the model's pseudo-residuals, based on the gradient of the loss of the ensemble:

$$f(\boldsymbol{x}) = \sum_{m}^{M} \beta_m h(\boldsymbol{x}, \theta_m)$$

Ensemble model with learning rate β_m, base learners $h(\boldsymbol{x}, \theta_m)$ with parameters θ_m.

$$\boldsymbol{r}_{\boldsymbol{m}}^{\boldsymbol{i}} = -\left[\frac{\partial L(y^i, f(x^i))}{\partial f(x^i)}\right]_{f=f_{m-1}}$$

Pseudo-residuals to which the next base learner will be fitted; by default, the loss of the updated ensemble will be less or equal to the loss of the current ensemble.

Gradient-boosted tree-based models

Gradient-boosted models are very successful in regression and classification tasks and still represent state-of-the-art in traditional ML.

If you have a classification or regression problem, it is always worth trying out gradient-boosted methods.

Common implementations:

- XGBoost
- LightGBM

Tree-based models – pros and cons

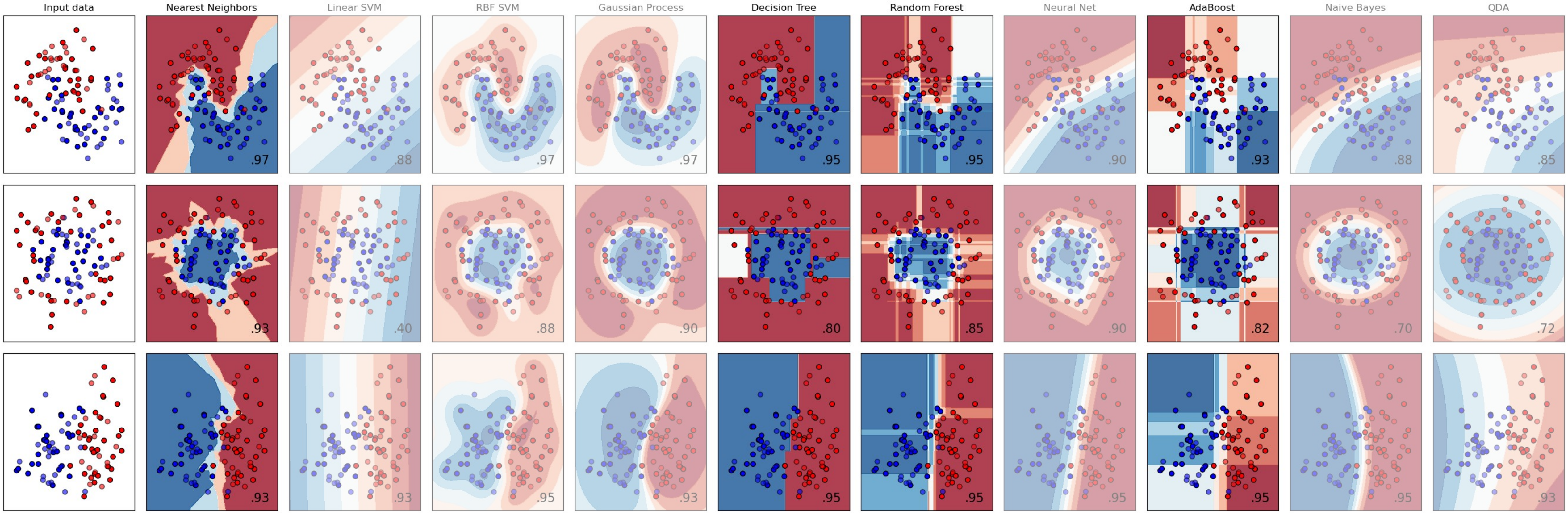
Pros:

- Extremely versatile and robust.
- Can be trained on small amounts of data
- Non-parametric
- Interpretability: tree-based models are able to compute “**feature importances**”

Cons:

- Decision boundaries and regression predictions may be discrete instead of continuous (see next slide)

Supervised learning - summary



(Similar to XGBoost)

Scikit-learn classifier comparison

That’s all folks!

Today’s lecture

Supervised learning

Supervised learning setup

Supervised learning concepts

Benchmarking and metrics

Linear models

Nearest Neighbor models

Tree-based models

Next Week:

Lab 1: Supervised learning

Next lecture (20 March)

Unsupervised learning

Unsupervised learning setup

Hierarchical clustering

k-means clustering

Expectation Maximization clustering

DBSCAN

Principal component analysis