Introduction to Machine Learning

Joaquin Vanschoren, Eindhoven Univeristy of Technology

Artificial Intelligence

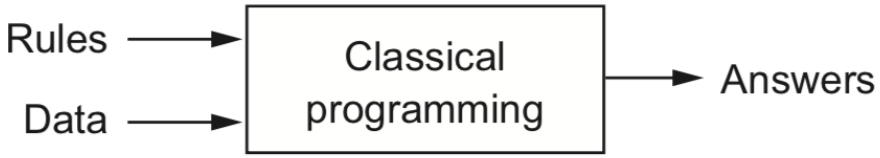
1950s: Can computers be made to 'think'?

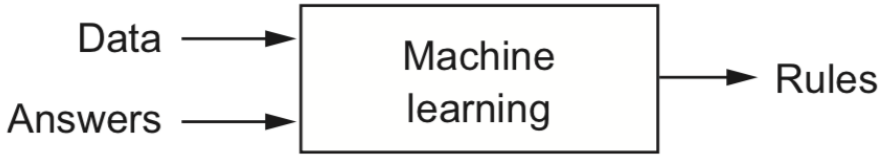
- automate intellectual tasks normally performed by humans
- encompasses learning, but also many other tasks (e.g. logic, planning,...)
- *symbolic AI*: programmed rules/algorithms for manipulating knowledge
  - Great for well-defined problems: chess, expert systems,...
  - Pervasively used today (e.g. chip design)
  - Hard for complex, fuzzy problems (e.g. images, text)

Machine Learning

Are computers capable of learning and originality? Alan Turing: Yes!

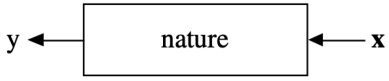
- Learn to perform a task T given experience E, always improving according to some metric M
- New programming paradigm
  - System is *trained* rather than explictly programmed
  - Finds rules or functions (models) to act/predict



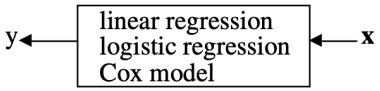


Machine learning vs Statistics

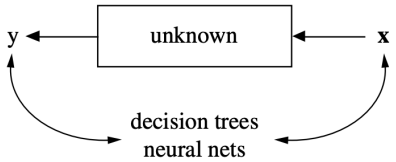
- Both aim to make predictions of natural phenomena:



- Statistics:
  - Help humans understand the world
  - Parametric: assume that data is generated according to parametric model



- Machine learning:
  - Automate a task entirely (replace the human)
  - Non-parametric: assume that data generation process is unknown
  - Engineering-oriented, less (too little?) mathematical theory



See Breiman (2001): Statical modelling: The two cultures

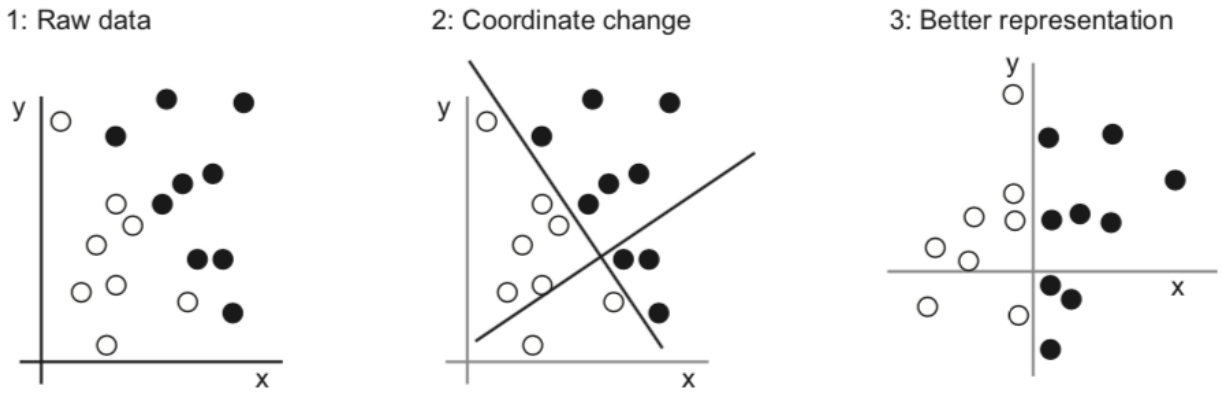
How to represent learning?

All machine learning algorithms consist of 3 components:

- Representation: A model must be represented in a formal language that the computer can handle
  - Defines the 'concepts' it can learn, the _hypothesis space-
  - E.g. a decision tree, neural network, set of annotated data points
- Evaluation: An *internal* way to choose one hypothesis over the other
  - Objective function, scoring/loss function
  - E.g. Difference between correct output and predictions
- Optimization: An *efficient* way to search the hypothesis space
  - Start from simple hypothesis, extend (relax) if it doesn't fit the data
  - Defines speed of learning, number of optima,...
  - E.g. Gradient descent

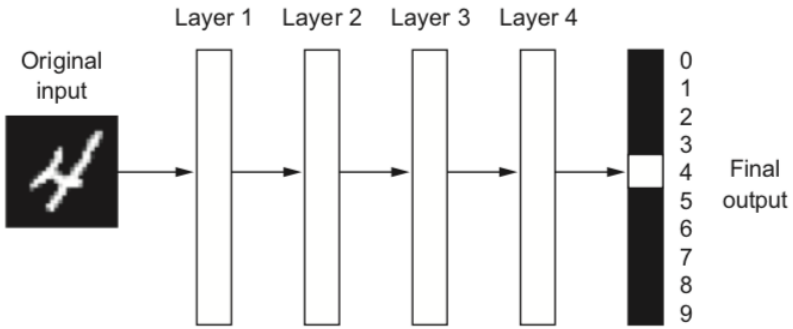
How to represent the problem?

- We need 3 inputs:
  - Input data, e.g. measurements, images, text
  - Expected output: e.g. correct labels produced by humans
  - Performance measure: feedback signal, are we learning the right thing?
- Algorithm needs to correctly transform the inputs to the right outputs
- Often includes transforming the data to a more useful representation (or encoding)
  - Can be done end-to-end (e.g. deep learning) or by first 'preprocessing' the data



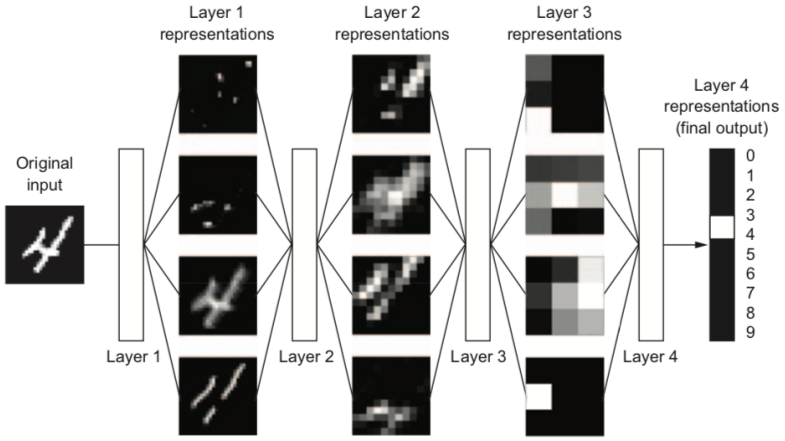
Deep Learning

- Most machine learning techniques require humans to build a good representation of the data
  - Sometimes data is naturally structured (e.g. medical tests)
  - Sometimes not (e.g. images) -> extract features
- Deep learning: learn your own representation of the data
  - Through multiple layers of representation (e.g. layers of neurons)
  - Each layer transforms the data a bit, based on what reduces the error

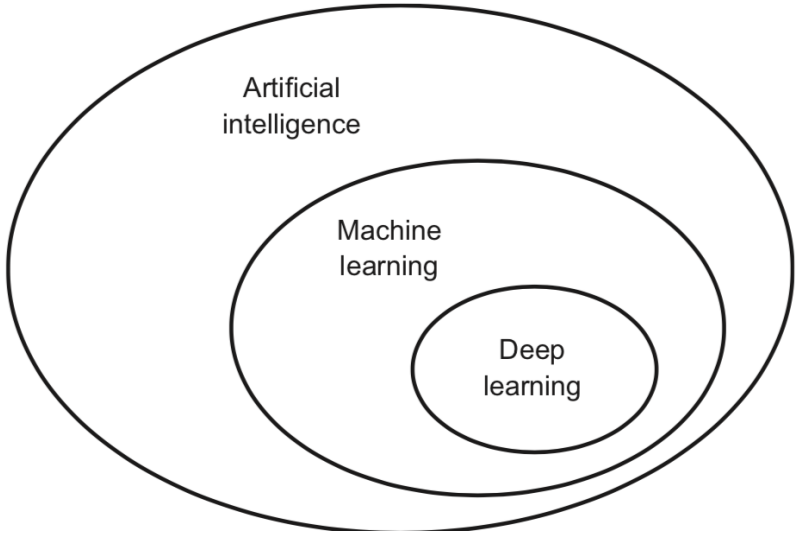


Example: digit classification

- Input pixels go in, each layer transforms them to an increasingly informative representation for the given task
- Often less intuitive for humans



Overview



Success stories:

- Search engines (e.g. Google)
- Recommender systems (e.g. Netflix)
- Automatic translation (e.g. Google Translate)
- Speech understanding (e.g. Siri, Alexa)
- Game playing (e.g. AlphaGo)
- Self-driving cars
- Personalized medicine
- Progress in all sciences: Genetics, astronomy, chemistry, neurology, physics,..

Example: dating

Nr	Day of Week	Type of Date	Weather	TV Tonight	Date?
1	Weekday	Dinner	Warm	Bad	No
2	Weekend	Club	Warm	Bad	Yes
3	Weekend	Club	Warm	Bad	Yes
4	Weekend	Club	Cold	Good	No
Now	Weekend	Club	Cold	Bad	?

- Is there a combination of factor that works? Is one better than others?
- What can we assume about the future? Nothing?
- What if there is noise / errors?
- What if there are factor you don't know about?

Types of machine learning

We often distinguish 3 `types` of machine learning:

- **Supervised Learning**: learn a model from labeled *training data,* then make predictions
- **Unsupervised Learning**: explore the structure of the data to extract meaningful information
- **Reinforcement Learning**: develop an agent that improves its performance based on interactions with the environment

Note:

- Semi-supervised methods combine the first two.
- ML systems can combine many types in one system.

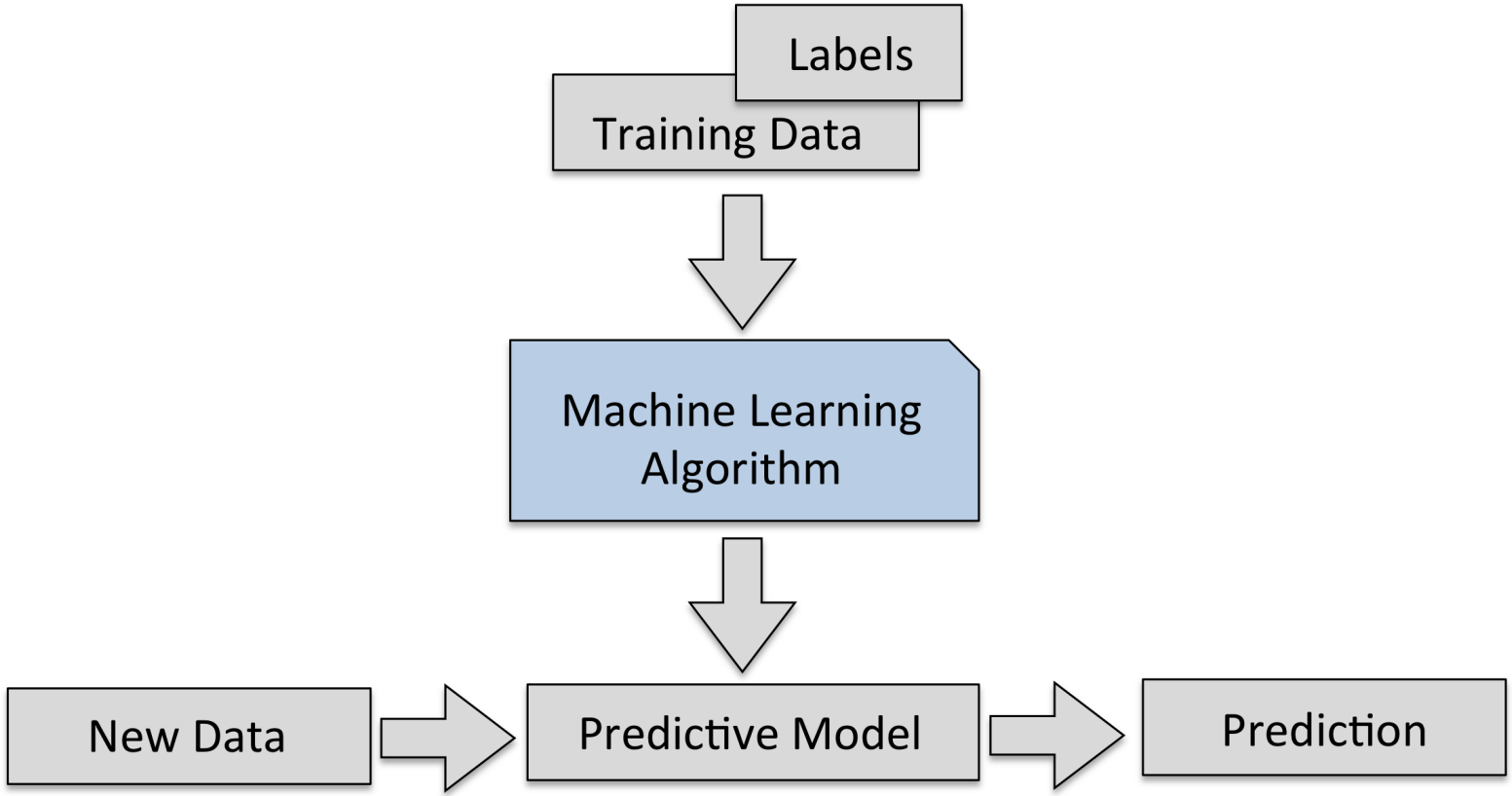
Supervised Machine Learning

- Learn a model from labeled training data, then make predictions
- Supervised: we know the correct/desired outcome (label)

2 subtypes:

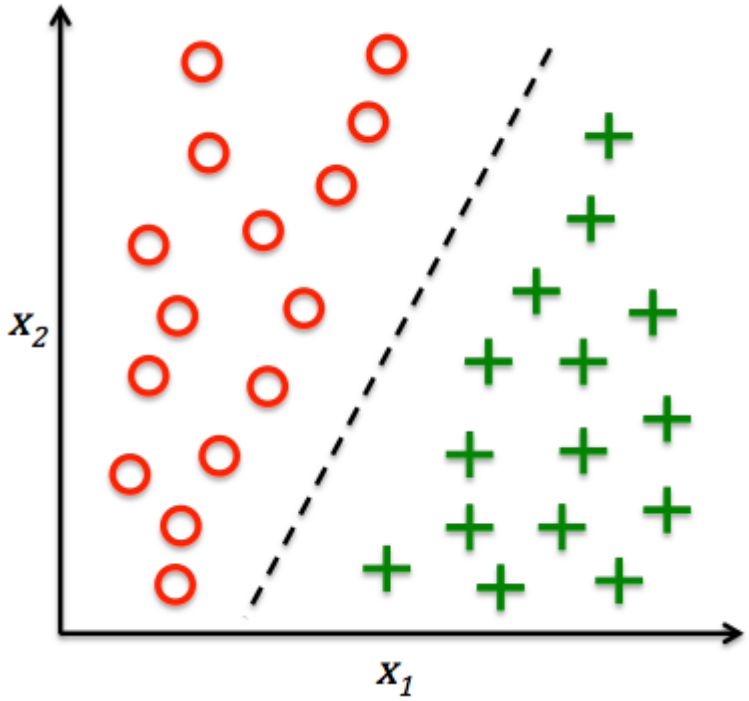
- Classification: predict a *class label* (category), e.g. spam/not spam
  - Many classifiers can also return a *confidence* per class
- Regression: predict a continuous value, e.g. temperature
  - Some algorithms can return a *confidence interval*

Most supervised algorithms that we will see can do both.


Labels
Training Data
Machine Learning Algorithm
New Data
Predictive Model
Prediction

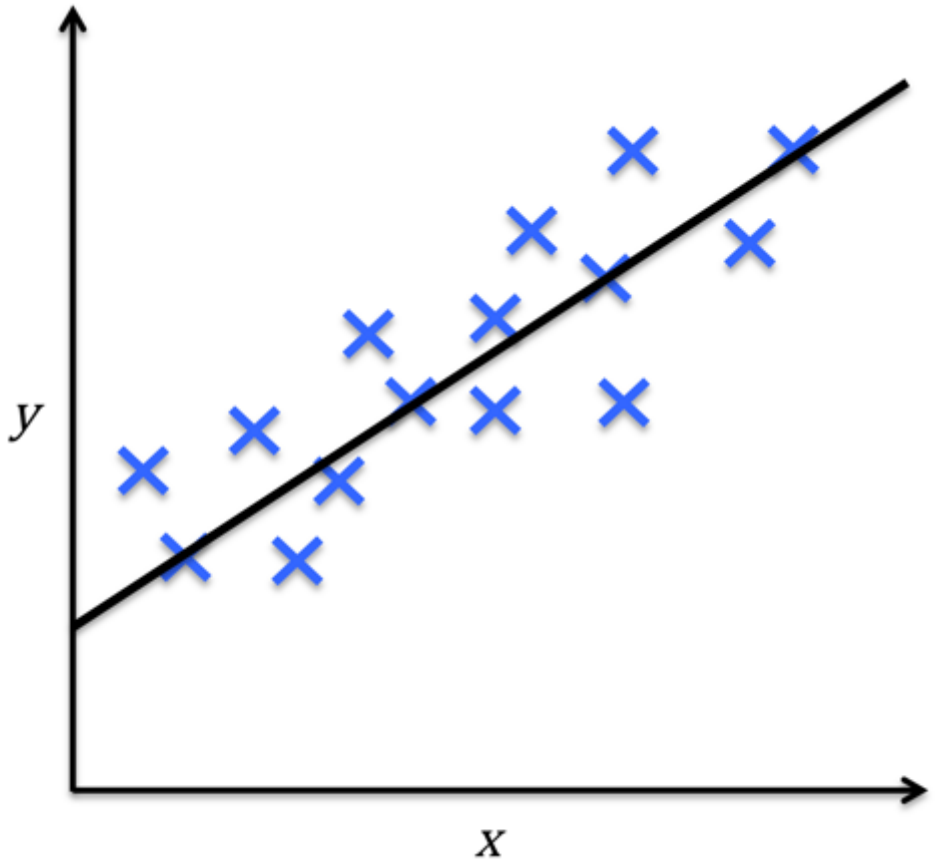
Classification

- Class labels are discrete, unordered
- Can be *binary* (2 classes) or *multi-class* (e.g. letter recognition)
- Dataset can have any number of predictive variables (predictors)
  - Also known as the dimensionality of the dataset
- The predictions of the model yield a *decision boundary* separating the classes



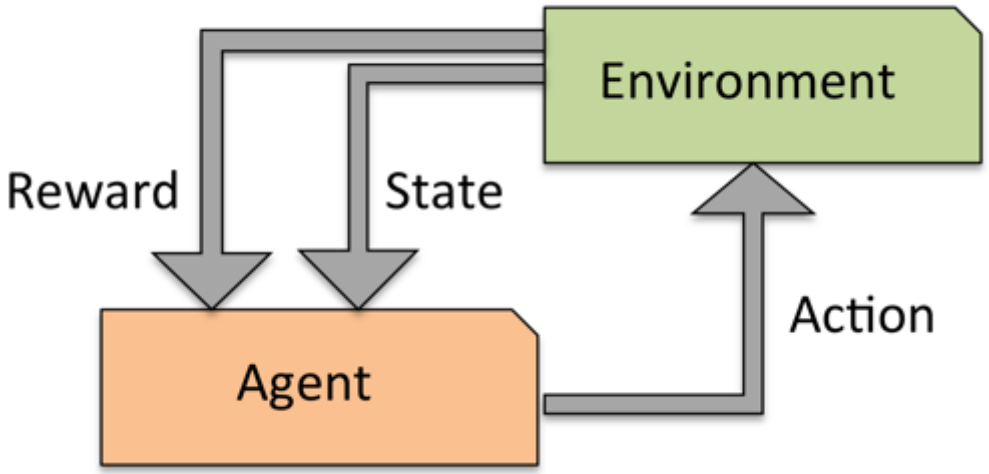
Regression

- Target variable is numeric
- Find the relationship between predictors and the target.
  - E.g. relationship between hours studied and final grade
- Example: Linear regression (fits a straight line)



Reinforcement learning

- Develop an agent that improves its performance based on interactions with the environment
  - Example: games like Chess, Go,...
- *Reward function* defines how well a (series of) actions works
- Learn a series of actions that maximizes reward through exploration

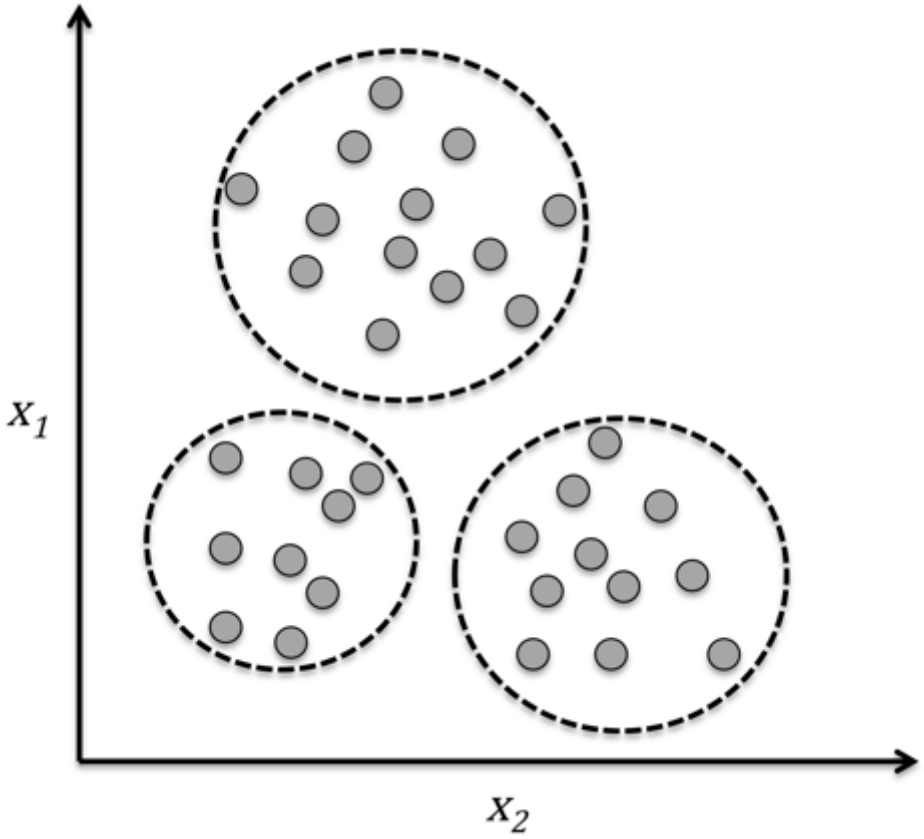


Unsupervised Machine Learning

- Unlabeled data, or data with unknown structure
- Explore the structure of the data to extract information
- Many types, we'll just discuss two.

Clustering

- Organize information into meaningful subgroups (clusters)
- Objects in cluster share certain degree of similarity (and dissimilarity to other clusters)
- Example: distinguish different types of customers



Dimensionality reduction

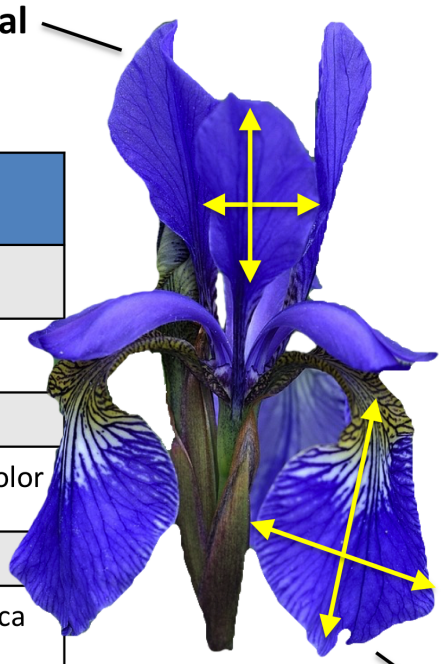
- Data can be very high-dimensional and difficult to understand, learn from, store,...
- Dimensionality reduction can compress the data into fewer dimensions, while retaining most of the information
- Contrary to feature selection, the new features lose their (original) meaning
- Is often useful for visualization (e.g. compress to 2D)

x_1
x_2
x_3
z_1
z_2

Basic Terminology (on Iris dataset)

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica



Features
(attributes, measurements, dimensions)

Class labels
(targets)

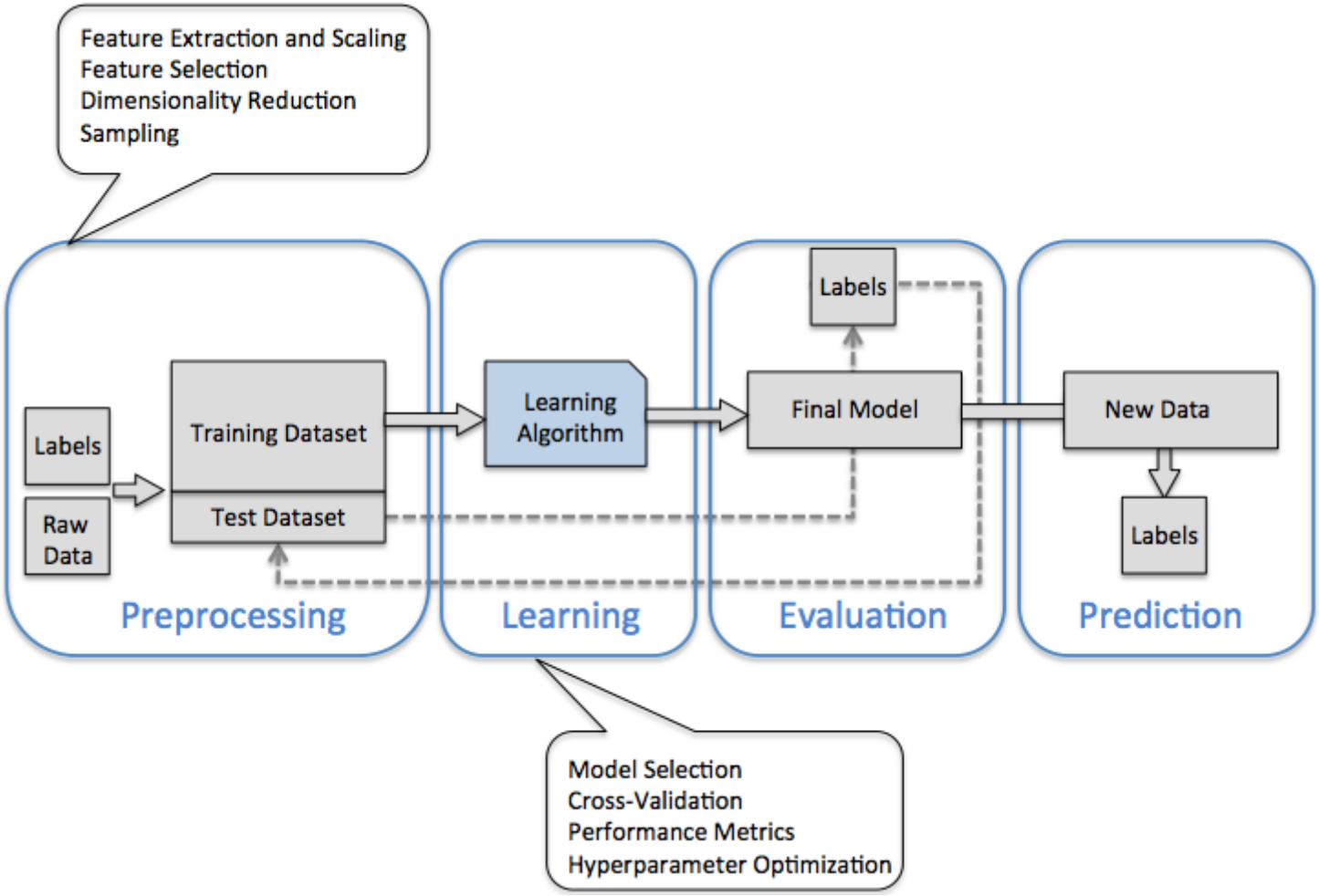
Building machine learning systems

A typical machine learning system has multiple components:

- Preprocessing: Raw data is rarely ideal for learning
  - Feature scaling: bring values in same range
  - Encoding: make categorical features numeric
  - Discretization: make numeric features categorical
  - Feature selection: remove uninteresting/correlated features
  - Dimensionality reduction can also make data easier to learn

- Learning and model selection
  - Every algorithm has its own biases
  - No single algorithm is always best (No Free Lunch)
  - *Model selection* compares and selects the best models
    - Different algorithms
    - Every algorithm has different options (hyperparameters)
  - Split data in training and test sets

- Together they form a *workflow* of *pipeline*



scikit-learn

One of the most prominent Python libraries for machine learning:

- Contains many state-of-the-art machine learning algorithms
- Offers comprehensive documentation (http://scikit-learn.org/stable/documentation) about each algorithm
- Widely used, and a wealth of tutorials (http://scikit-learn.org/stable/user_guide.html) and code snippets are available
- scikit-learn works well with numpy, scipy, pandas, matplotlib,...

Algorithms

See the Reference (http://scikit-learn.org/dev/modules/classes.html)

Supervised learning:

- Linear models (Ridge, Lasso, Elastic Net, ...)
- Support Vector Machines
- Tree-based methods (Classification/Regression Trees, Random Forests,...)
- Nearest neighbors
- Neural networks
- Gaussian Processes
- Feature selection

Unsupervised learning:

- Clustering (KMeans, ...)
- Matrix Decomposition (PCA, ...)
- Manifold Learning (Embeddings)
- Density estimation
- Outlier detection

Model selection and evaluation:

- Cross-validation
- Grid-search
- Lots of metrics

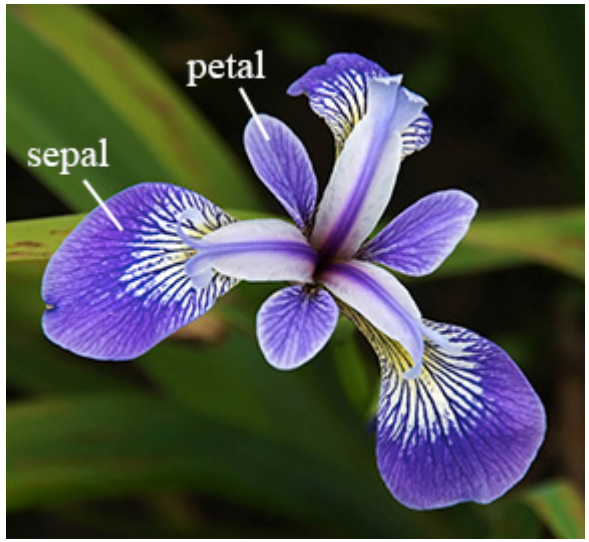
Data import

Multiple options:

- A few toy datasets are included in `sklearn.datasets`
- You can import data files (CSV) with `pandas` or `numpy`
- You can import 1000s of machine learning datasets from OpenML

Example: classification

Classify types of Iris flowers (setosa, versicolor, or virginica) based on the flower sepals and petal leave sizes.



Iris is included in scikitlearn, we can just load it.
This will return a `Bunch` object (similar to a `dict`)

```
Keys of iris_dataset: dict_keys(['data', 'target', 'target_names', 'DESC
R', 'feature_names', 'filename'])
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, pre
...
```

The targets (classes) and features are stored as `lists`, the data as an `ndarray`

```
Targets: ['setosa' 'versicolor' 'virginica']
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
Shape of data: (150, 4)
First 5 rows:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
```

The targets are stored separately as an `ndarray`, with indices pointing to the features

```
Target names: ['setosa' 'versicolor' 'virginica']
Targets:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Building your first model

All scikitlearn classifiers follow the same interface

```
class SupervisedEstimator(...):
    def __init__(self, hyperparam, ...):

    def fit(self, X, y):    # Fit/model the training data
        ...                 # given data X and targets y
        return self

    def predict(self, X):   # Make predictions
        ...                 # on unseen data X
        return y_pred

    def score(self, X, y): # Predict and compare to true
        ...                 # labels y
        return score
```

Training and testing data

To evaluate our classifier, we need to test it on unseen data.
`train_test_split`: splits data randomly in 75% training and 25% test data.

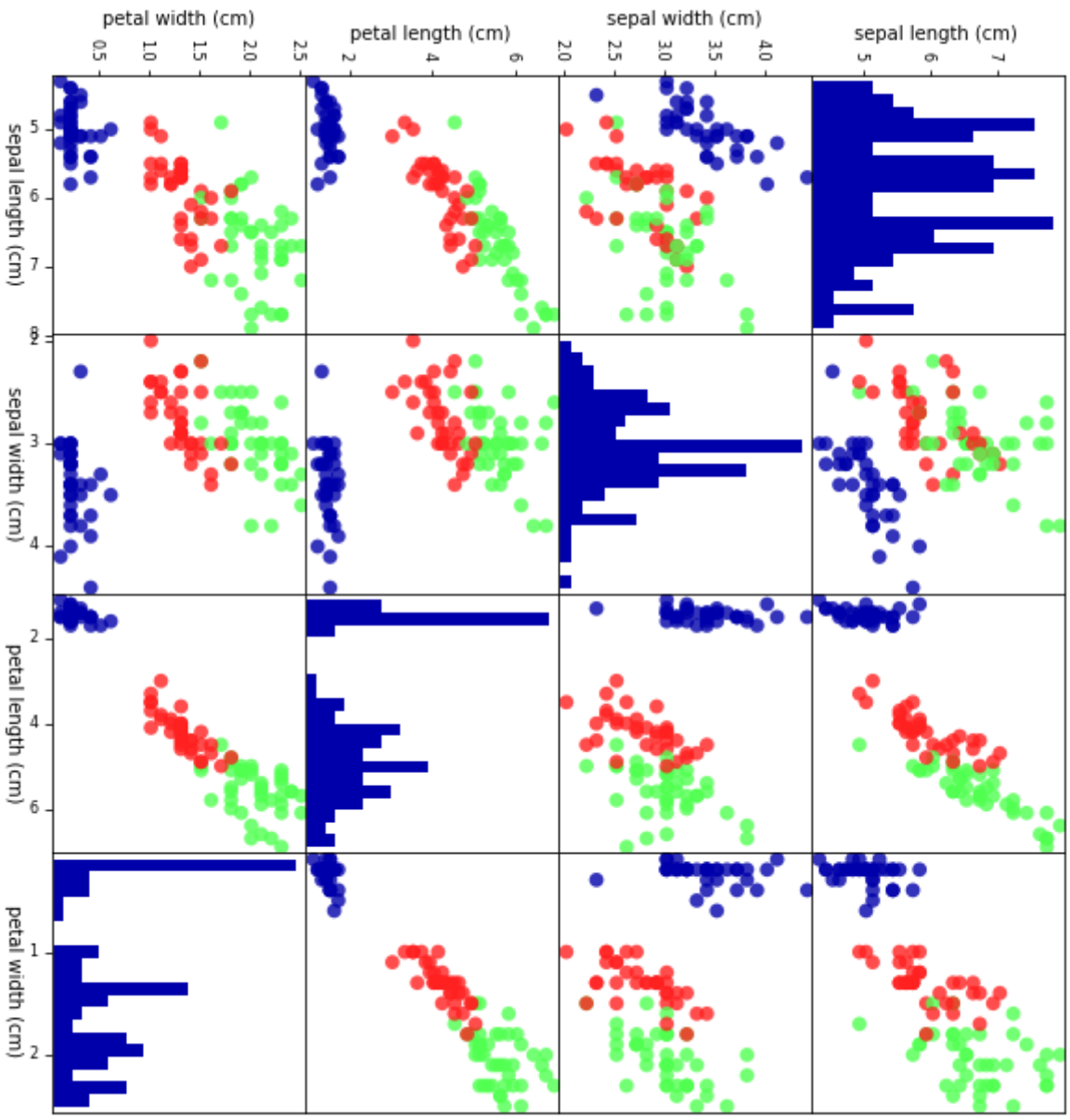
```
X_train shape: (112, 4)
y_train shape: (112,)
X_test shape: (38, 4)
y_test shape: (38,)
```

Note: there are several problems with this approach that we will discuss later:

- Why 75%? Are there better ways to split?
- What if one random split yields different models than another?
- What if all examples of one class all end up in the training/test set?

Looking at your data

We can use a library called `pandas` to easily visualize our data. Note how several features allow to cleanly split the classes.


petal width (cm)
petal length (cm)
sepal width (cm)
sepal length (cm)
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)

Fitting a model

The first model we'll build is called k-Nearest Neighbor, or kNN. More about that soon.
kNN is included in `sklearn.neighbors`, so let's build our first model

Out[7]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=1, p=2,
           weights='uniform')
```

Making predictions

Let's create a new example and ask the kNN model to classify it

```
Prediction: [0]
Predicted target name: ['setosa']
```

Evaluating the model

Feeding all test examples to the model yields all predictions

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1
0
 2]
```

We can now just count what percentage was correct

```
Score: 0.97
```

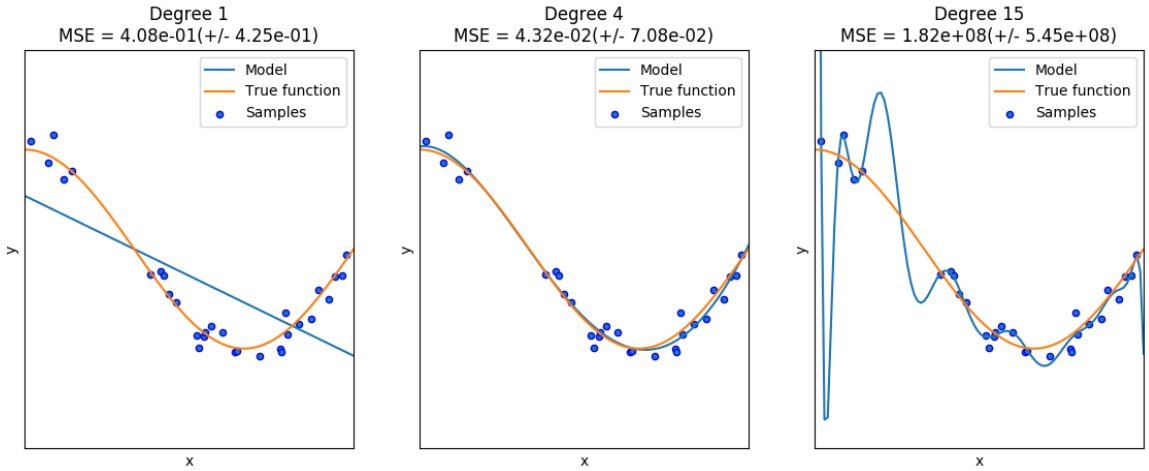
The `score` function does the same thing (by default)

```
Score: 0.97
```

Generalization, Overfitting and Underfitting

- We **hope** that the model can *generalize* from the training to the test data: make accurate predictions on unseen data
- It's easy to build a complex model that is 100% accurate on the training data, but very bad on the test data
- Overfitting: building a model that is *too complex for the amount of data* that we have
  - You model peculiarities in your data (noise, biases,...)
  - Solve by making model simpler (regularization), or getting more data
- Underfitting: building a model that is *too simple given the complexity of the data*
  - Use a more complex model

- There is often a sweet spot that you need to find by optimizing the choice of algorithms and hyperparameters, or using more data.



In all supervised algorithms that we will discuss, we'll cover:

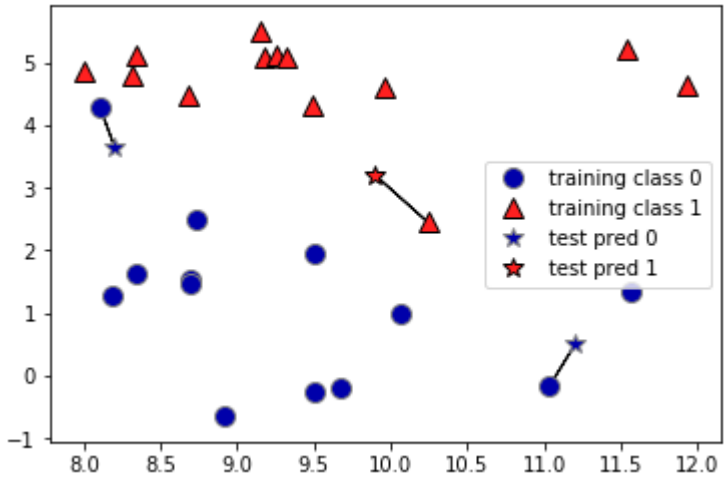
- How do they work
- How to control complexity
- Hyperparameters (user-controlled parameters)
- Strengths and weaknesses

k-Nearest Neighbor

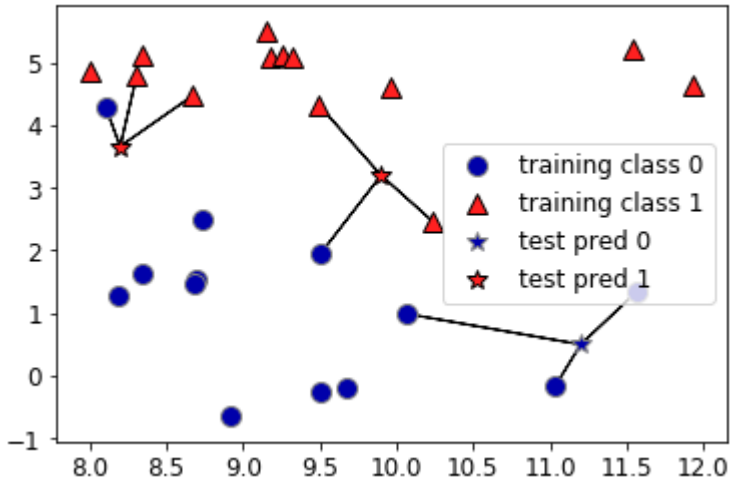
- Building the model consists only of storing the training dataset.
- To make a prediction, the algorithm finds the *k* closest data points in the training dataset

k-Nearest Neighbor Classification

for k=1: return the class of the nearest neighbor



for k>1: do a vote and return the majority (or a confidence value for each class)



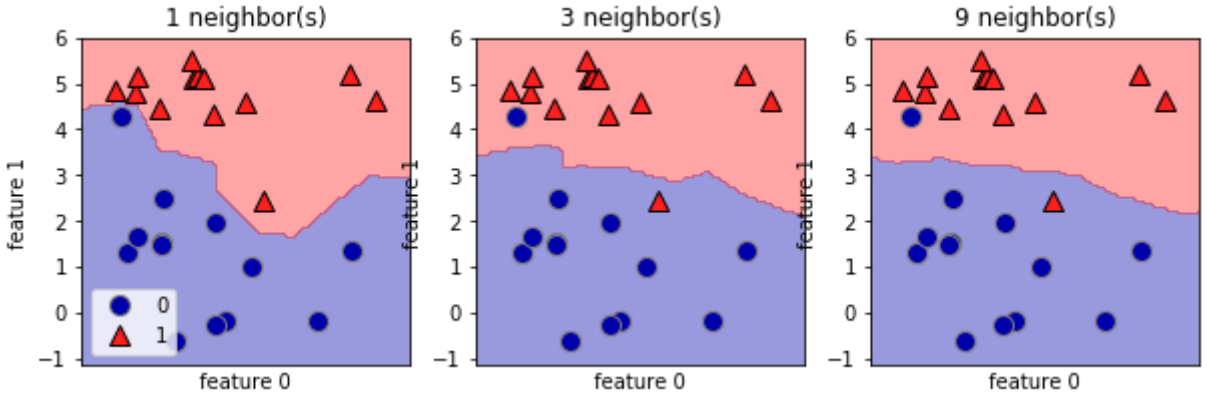
Let's build a kNN model for this dataset (called 'Forge')

```
Out[65]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                              metric_params=None, n_jobs=None, n_neighbors=19, p=2,
                              weights='uniform')
```

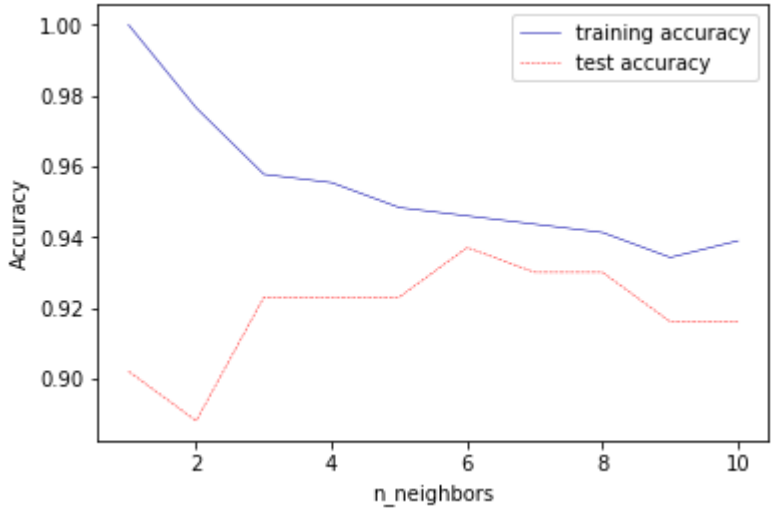
```
Test set accuracy: 0.43
```

Analysis

We can plot the prediction for each possible input to see the *decision boundary*

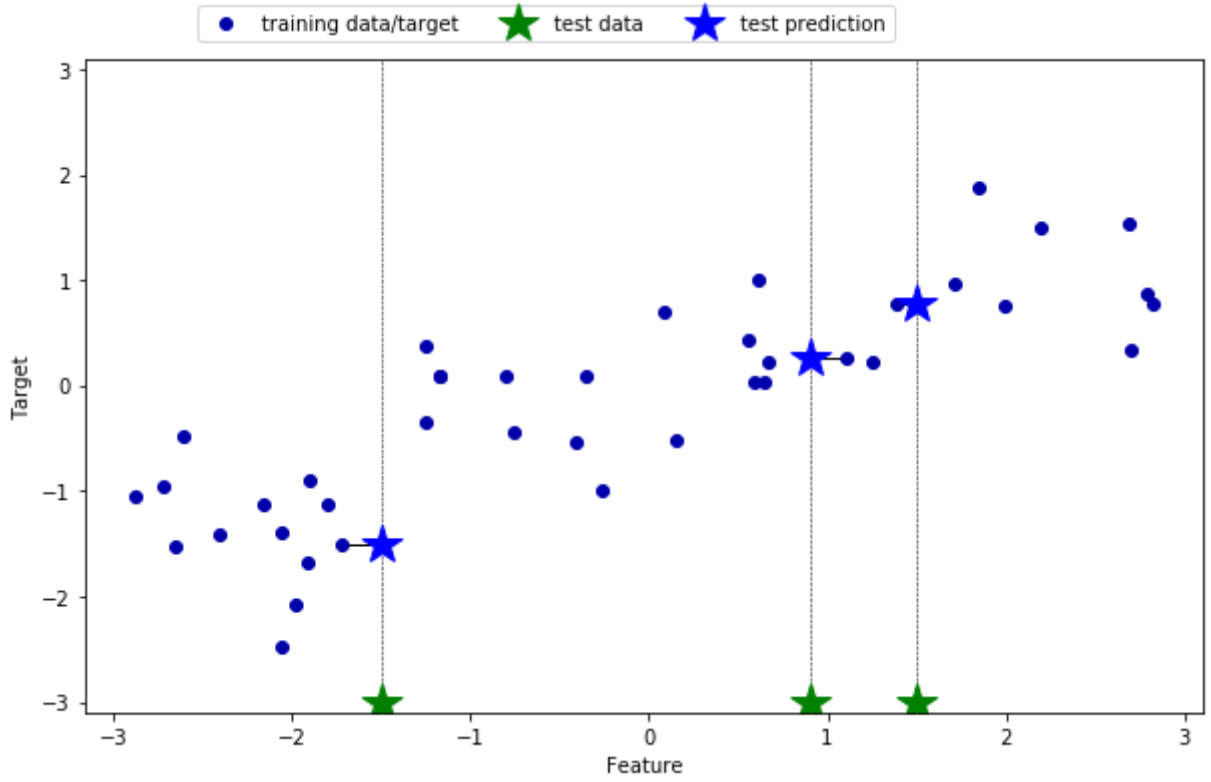


We can more directly measure the effect on the training and test error on a larger dataset (breast_cancer)

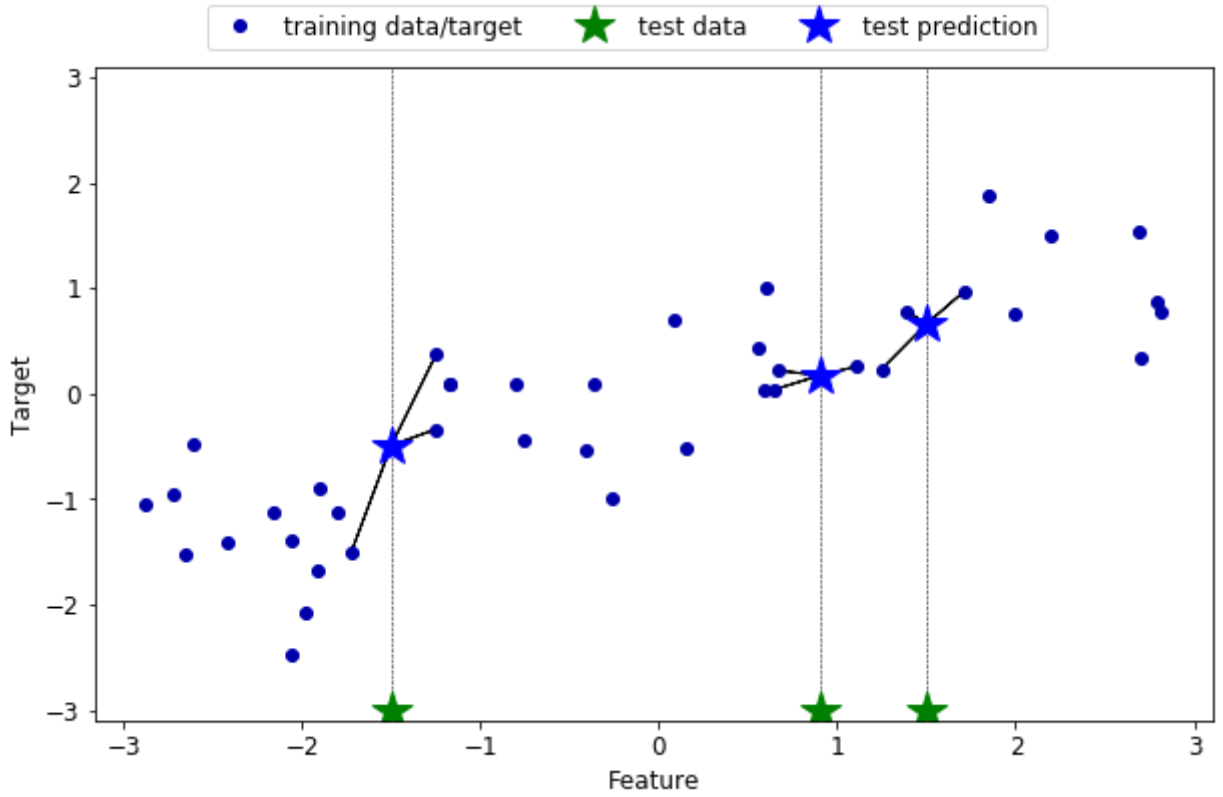


k-Neighbors Regression

for k=1: return the target value of the nearest neighbor



for k>1: return the *mean* of the target values of the *k* nearest neighbors



To do regression, simply use `KNeighborsRegressor` instead

```
Out[20]: KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                             metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                             weights='uniform')
```

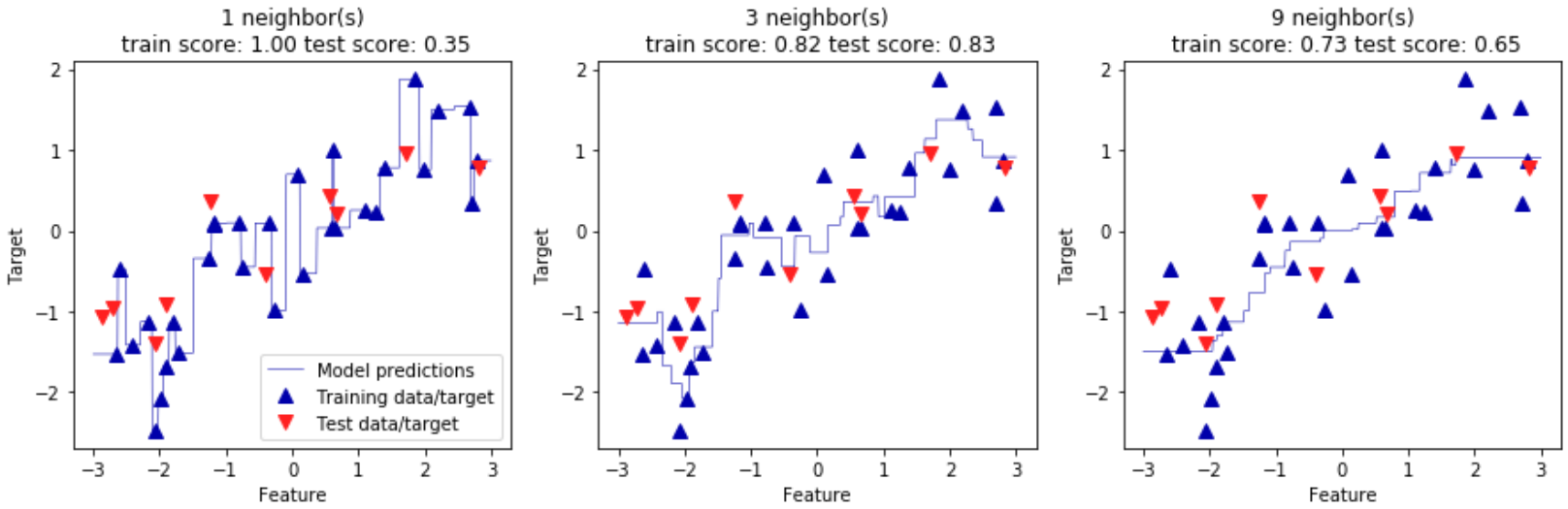
The default scoring function for regression models is R^2. It measures how much of the data variability is explained by the model. Between 0 and 1.

```
Test set predictions:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]

Test set R^2: 0.83
```

Analysis

We can again output the predictions for each possible input, for different values of k.



We see that again, a small k leads to an overly complex (overfitting) model, while a larger k yields a smoother fit.

kNN: Strengths, weaknesses and parameters

- There are two important hyperparameters:
  - n_neighbors: the number of neighbors used
  - metric: the distance measure used
    - Default is Minkowski (generalized Euclidean) distance.
- Easy to understand, works well in many settings
- Training is very fast, predicting is slow for large datasets
- Bad at high-dimensional and sparse data (curse of dimensionality)

Linear models

Linear models make a prediction using a linear function of the input features. Can be very powerful for or datasets with many features.

If you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.

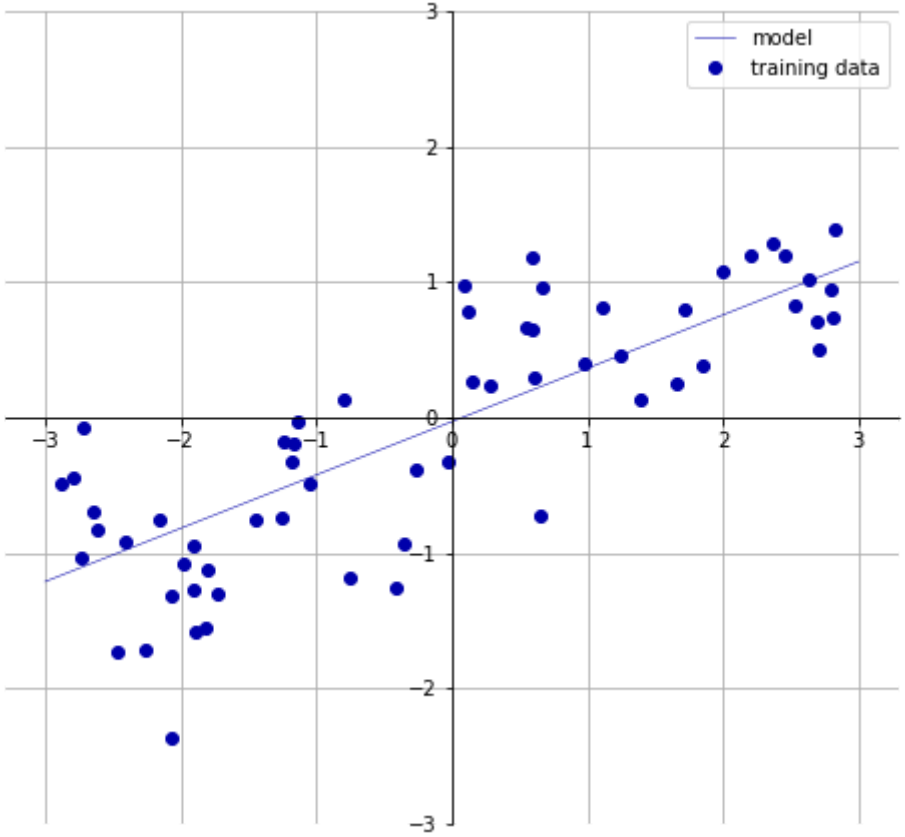
Linear models for regression

Prediction formula for input features x. w_i and b are the *model parameters* that need to be learned.

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b$$

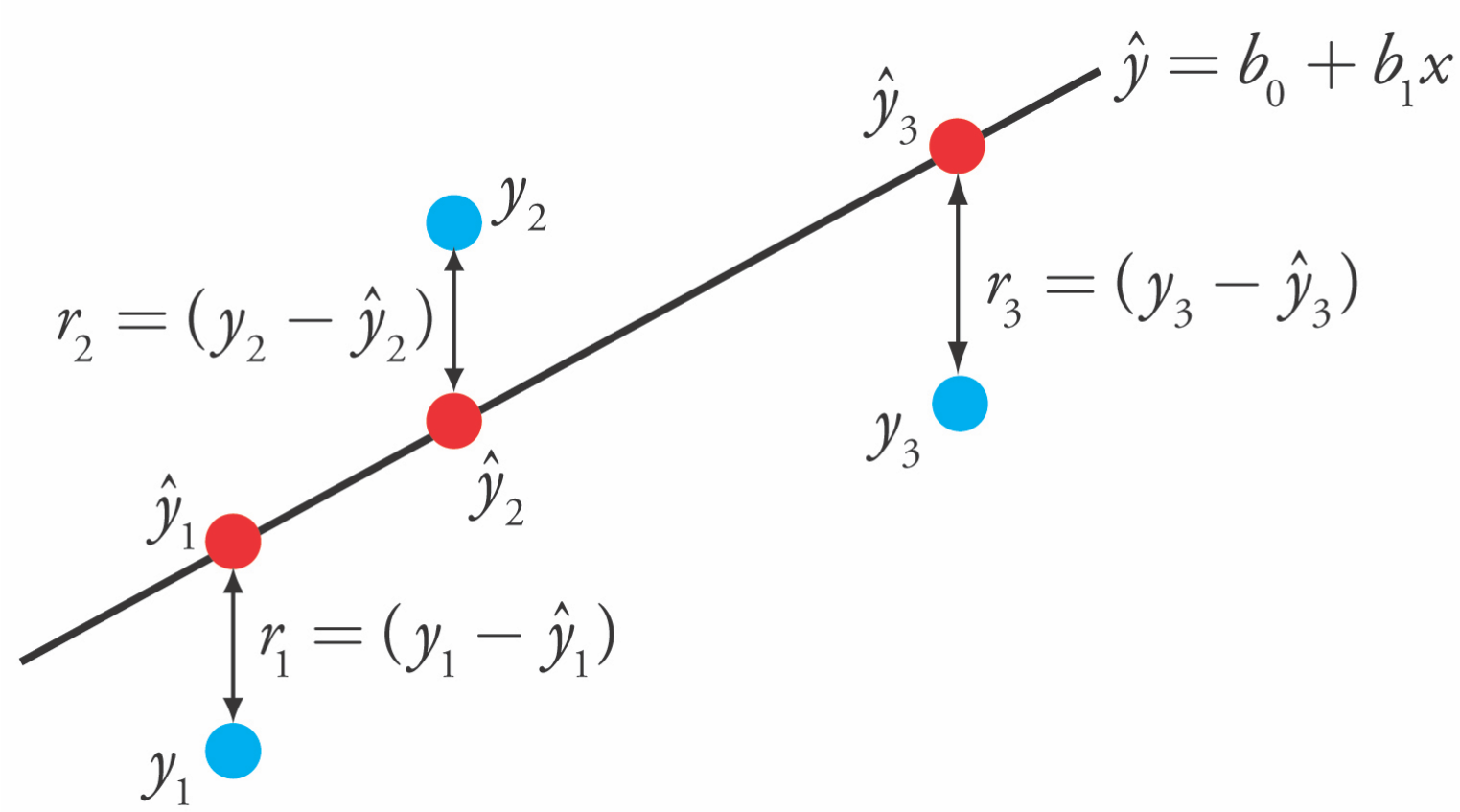
There are many different algorithms, differing in how w and b are learned from the training data.

```
w[0]: 0.393906  b: -0.031804
```



Linear Regression aka Ordinary Least Squares

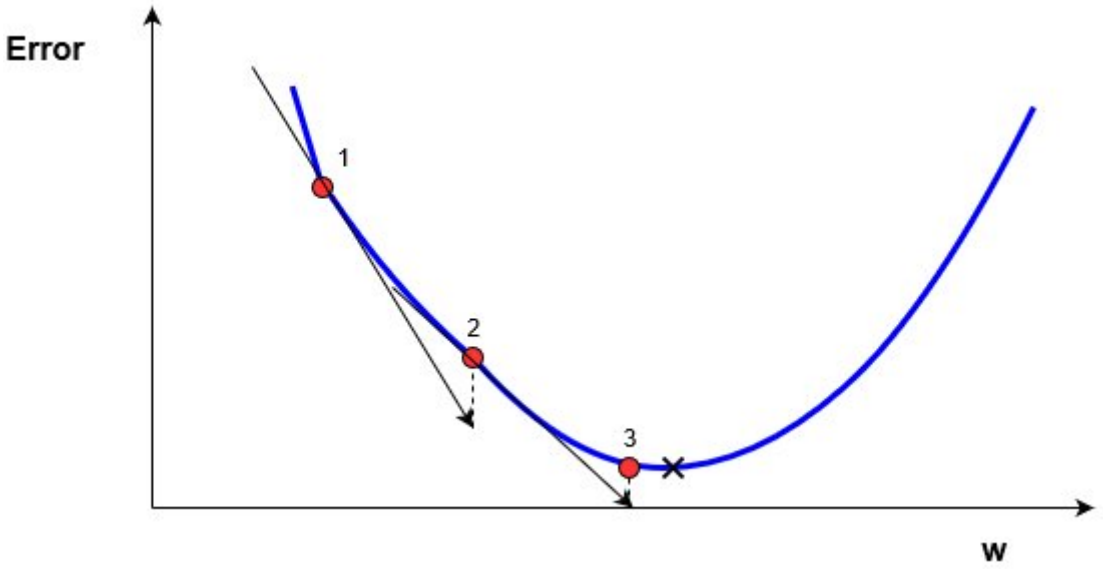
- Finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets, y, on the training set.
  - MSE: Sum of the squared differences between the predictions and the true values.
- Convex optimization problem with unique closed-form solution (if you have more data points than model parameters w)
- It has no hyperparameters, thus model complexity cannot be controlled.
  - Some other algorithms do: Ridge regression and Lasso


$\hat{y} = b_0 + b_1 x$
$\hat{y}_3$
y_2
$r_2 = (y_2 - \hat{y}_2)$
$r_3 = (y_3 - \hat{y}_3)$
y_3
$\hat{y}_1$
$\hat{y}_2$
$r_1 = (y_1 - \hat{y}_1)$
y_1

Linear regression can be found in `sklearn.linear_model`. We'll evaluate it on the Boston Housing dataset.

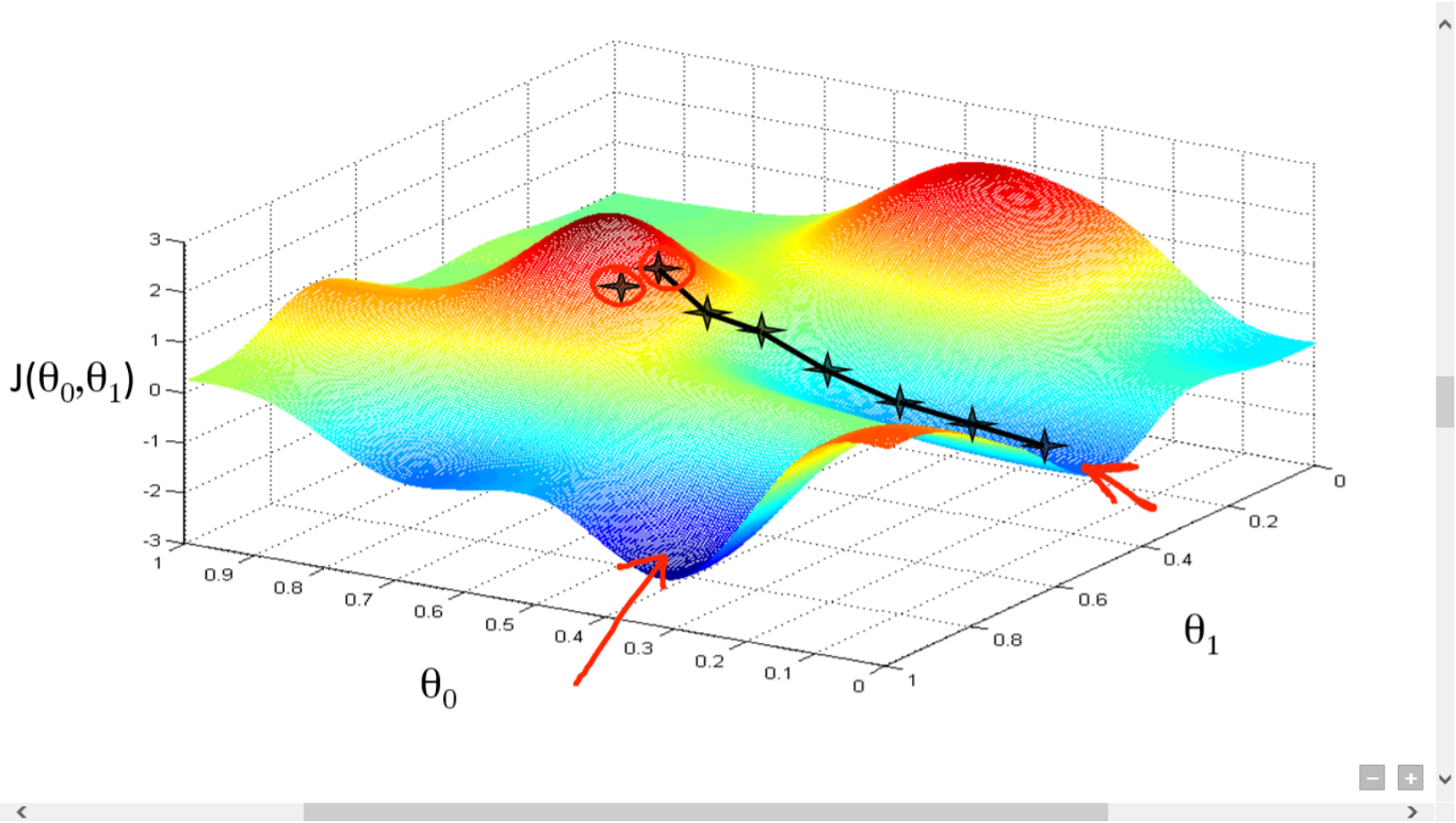
```
Weights (coefficients): [ -412.711   -52.243  -131.899   -12.004   -15.511
    28.716    54.704
   -49.535    26.582    37.062   -11.828   -18.058   -19.525    12.203
  2980.781  1500.843   114.187   -16.97     40.961   -24.264    57.616
  1278.121 -2239.869   222.825    -2.182    42.996   -13.398   -19.389
    -2.575   -81.013     9.66      4.914    -0.812    -7.647    33.784
   -11.446    68.508   -17.375    42.813     1.14     -0.773    56.826
    14.288    53.955   -32.171    19.271   -13.885    60.634   -12.315
   -12.004   -17.724   -33.987     7.09     -9.225    17.198   -12.772
   -11.973    57.387   -17.533     4.101    29.367   -17.661    78.405
   -31.91     48.175   -39.534     5.23     21.998    25.648   -49.998
    29.146     8.943   -71.66    -22.815     8.407    -5.379     1.201
    -5.209    41.145   -37.825    -2.672   -25.522   -33.398    46.227
   -24.151   -17.753   -13.972   -23.552    36.835   -94.689   144.303
   -15.116   -14.951   -28.773   -31.767    24.955   -18.438     3.651
     1.731    35.362    11.955     0.677     2.735    30.372]
Bias (intercept): 30.934563673637626
```

Optimization: Gradient Descent



Gradient Descent

Gradient Descent



Linear models for Classification

Aims to find a (hyper)plane that separates the examples of each class.
For binary classification (2 classes), we aim to fit the following function:

$$\hat{y} = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p + b > 0$$

When $\hat{y} < 0$, predict class -1, otherwise predict class +1

Logistic regression

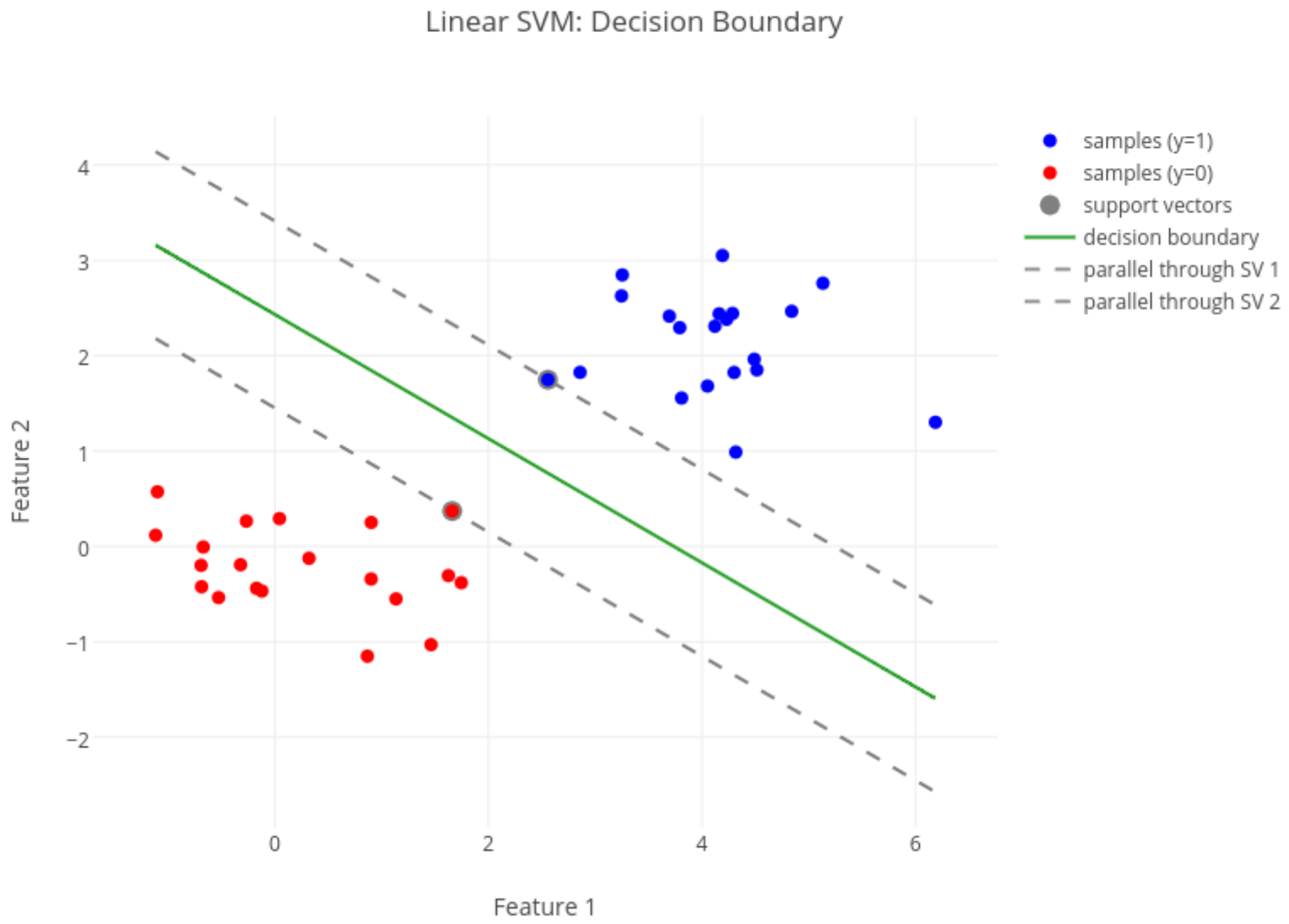
The logistic model uses the *logistic* (or *sigmoid*) function to estimate the probability that a given sample belongs to class 1:

$$z = f(x) = w_0 * x_0 + w_1 * x_1 + \ldots + w_p * x_p$$

$$\hat{y} = Pr[1|x_1, \ldots, x_k] = g(z) = \frac{1}{1 + e^{-z}}$$

Linear Support Vector Machine

Find hyperplane maximizing the *margin* between the classes

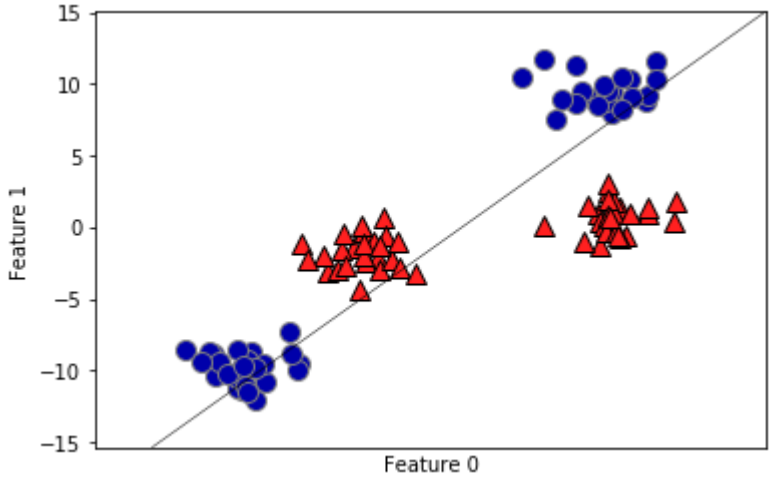


Prediction is identical to weighted kNN: find the support vector that is nearest, according to a distance measure (kernel) and a weight for each support vector.

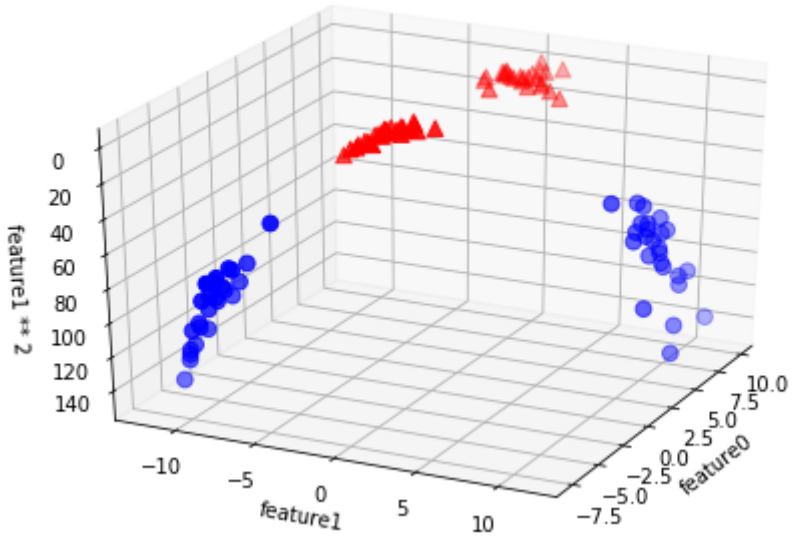
Kernelized Support Vector Machines

- Linear models work well in high dimensional spaces.
- You can *create* additional dimensions yourself.
- Let's start with an example.

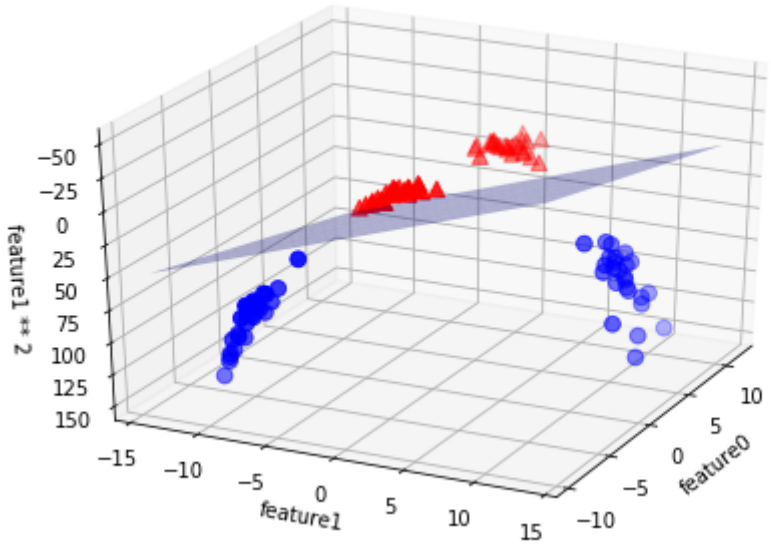
Our linear model doesn't fit the data well



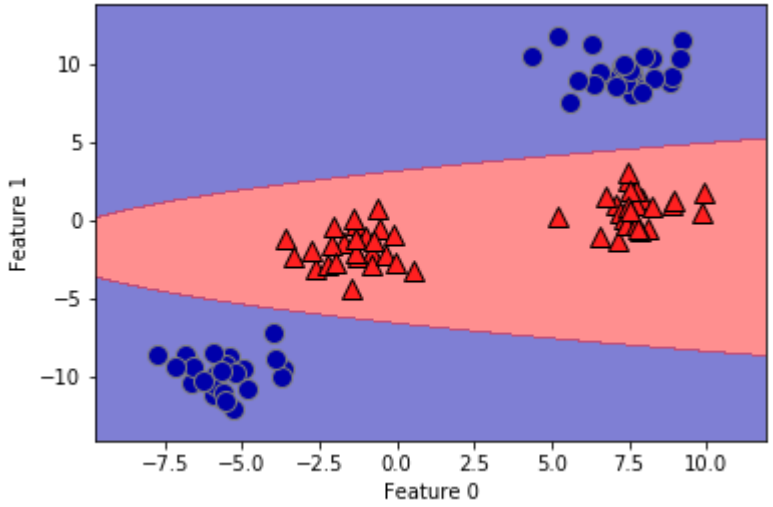
We can add a new feature by taking the squares of feature1 values



We can now fit a linear model



As a function of the original features, the linear SVM model is not actually linear anymore, but more of an ellipse



Kernels

A (Mercer) Kernel on a space X is a (similarity) function

$k : X \times X \to \mathbb{R}$

Of two arguments with the properties:

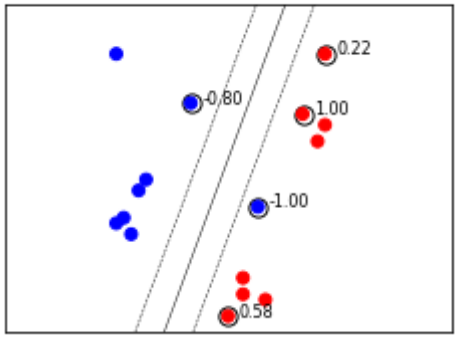
- Symmetry: $k(x_1, x_2) = k(x_2, x_1) \quad \forall x_1, x_2 \in X$
- Positive definite: for each finite subset of data points $x_1, \ldots, x_n$, the kernel Gram matrix is positive semi-definite

Kernel matrix = $K \in \mathbb{R}^{n \times n}$ with $K_{ij} = k(x_i, x_j)$

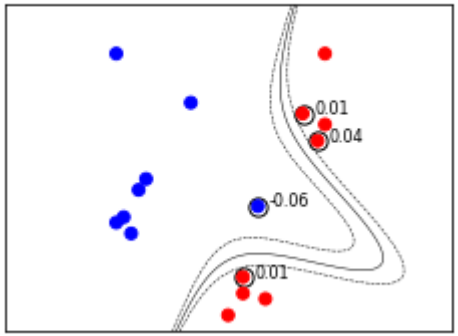
Kernels: examples

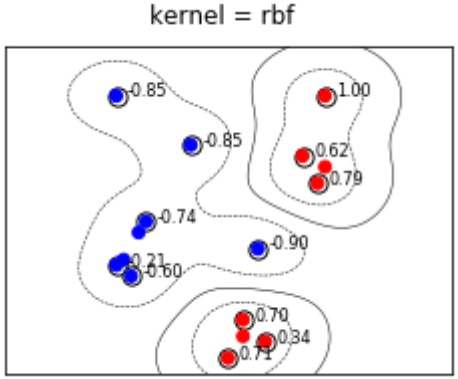
- The inner product is a kernel. The standard inner product is the **linear kernel**:
  $k(x_1, x_2) = x_1^T x_2$
- Kernels can be constructed from other kernels k_1 and k_2:
  - For $\lambda \geq 0, \lambda . k_1$ is a kernel
  - $k_1 + k_2$ is a kernel
  - $k_1 . k_2$ is a kernel (thus also k_1^n)
- This allows to construct the **polynomial kernel**:
  $k(x_1, x_2) = (x_1^T x_2 + b)^d$, for $b \geq 0$ and $d \in \mathbb{N}$

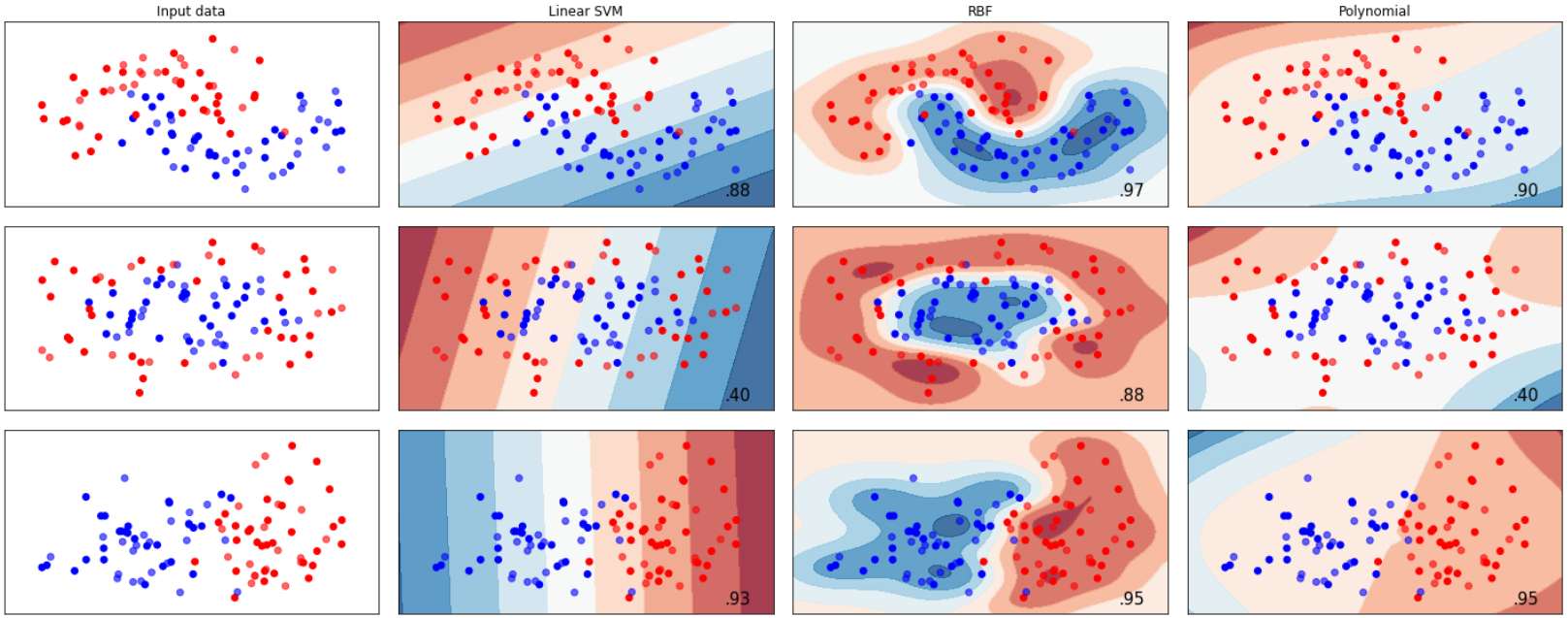
kernel = linear



kernel = poly

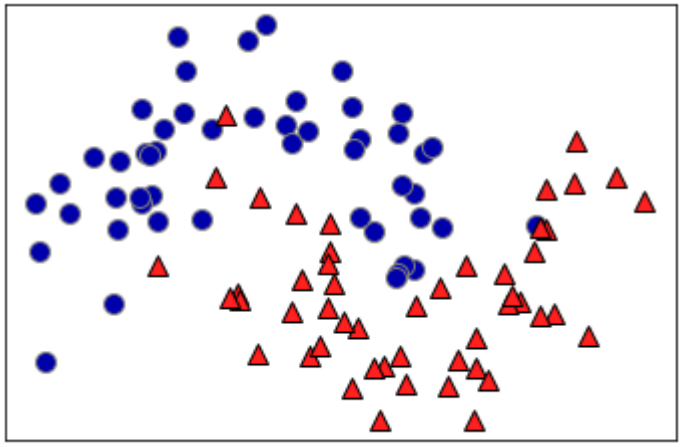


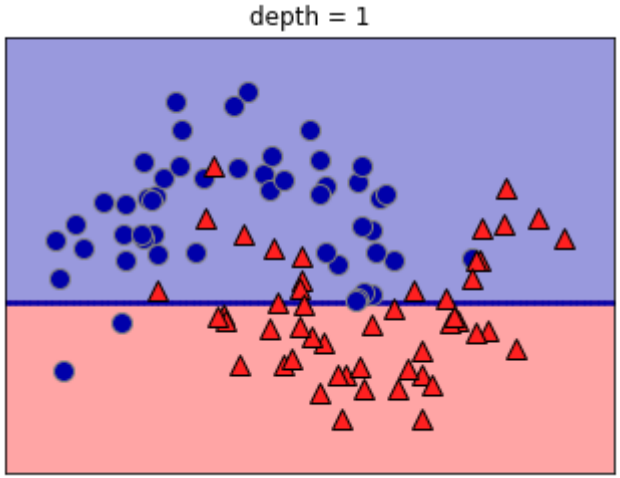

kernel = rbf
-0.85
-0.85
-0.74
-0.21
-0.60
-0.90
1.00
0.62
0.79
0.70
0.34
0.71

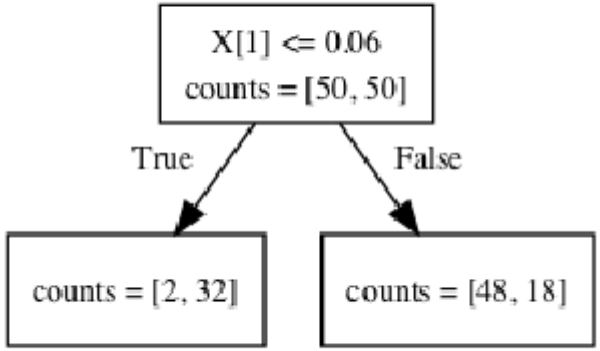

Input data
Linear SVM
RBF
Polynomial
.88
.97
.90
.40
.88
.40
.93
.95
.95

Decision Trees

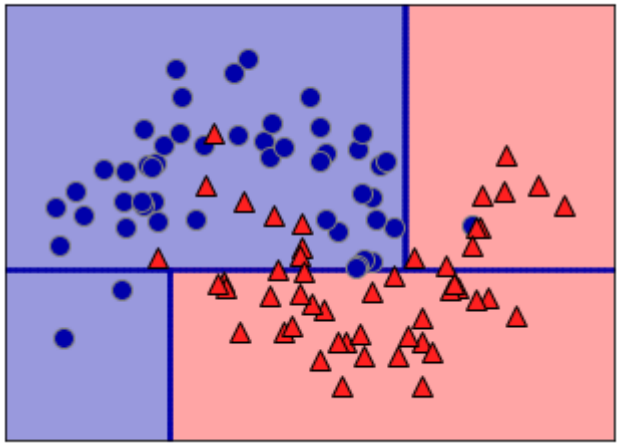
- Split the data in two (or more) parts
- Search over all possible splits and choose the one that is most *informative*
  - Many heuristics
  - E.g. *information gain*: how much does the entropy of the class labels decrease after the split (purer 'leafs')
- Repeat recursive partitioning

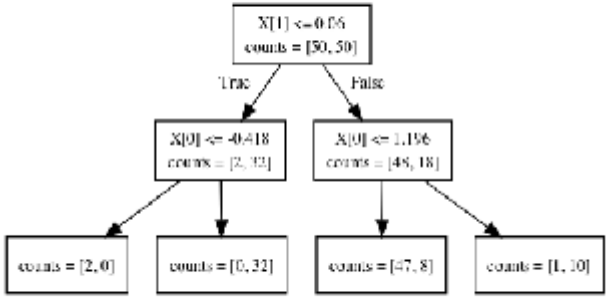



depth = 1

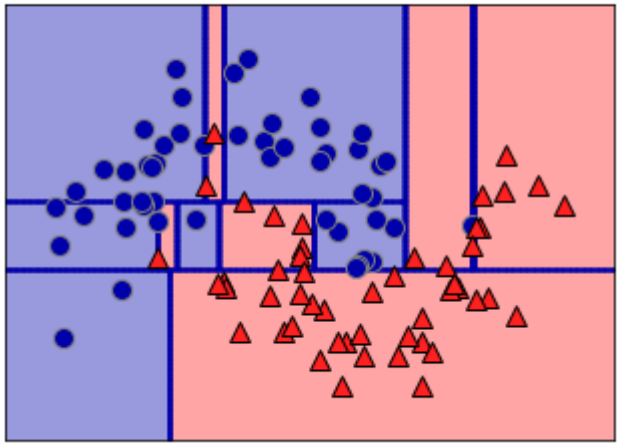

X[1] <= 0.06
counts = [50, 50]
True
False
counts = [2, 32]
counts = [48, 18]

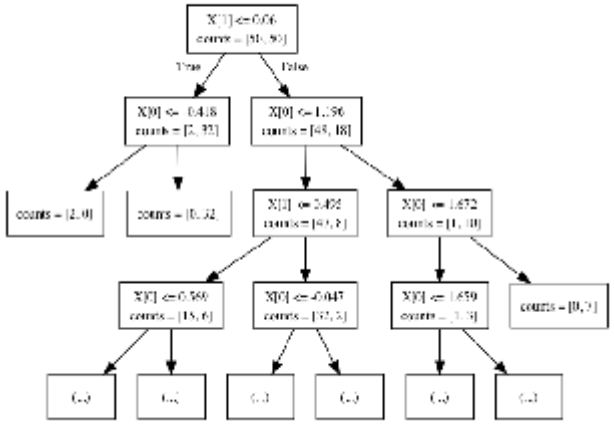
depth = 2



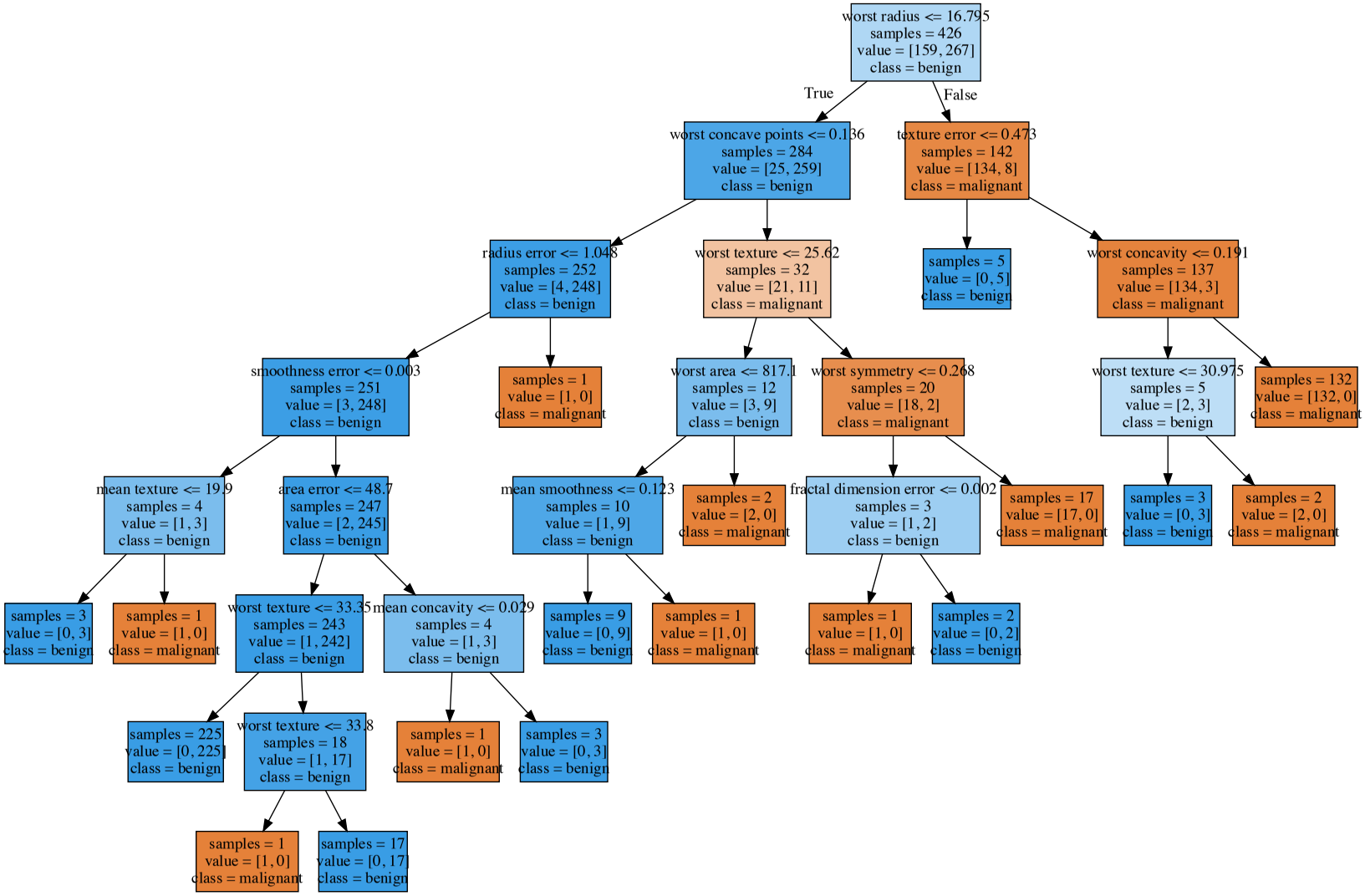


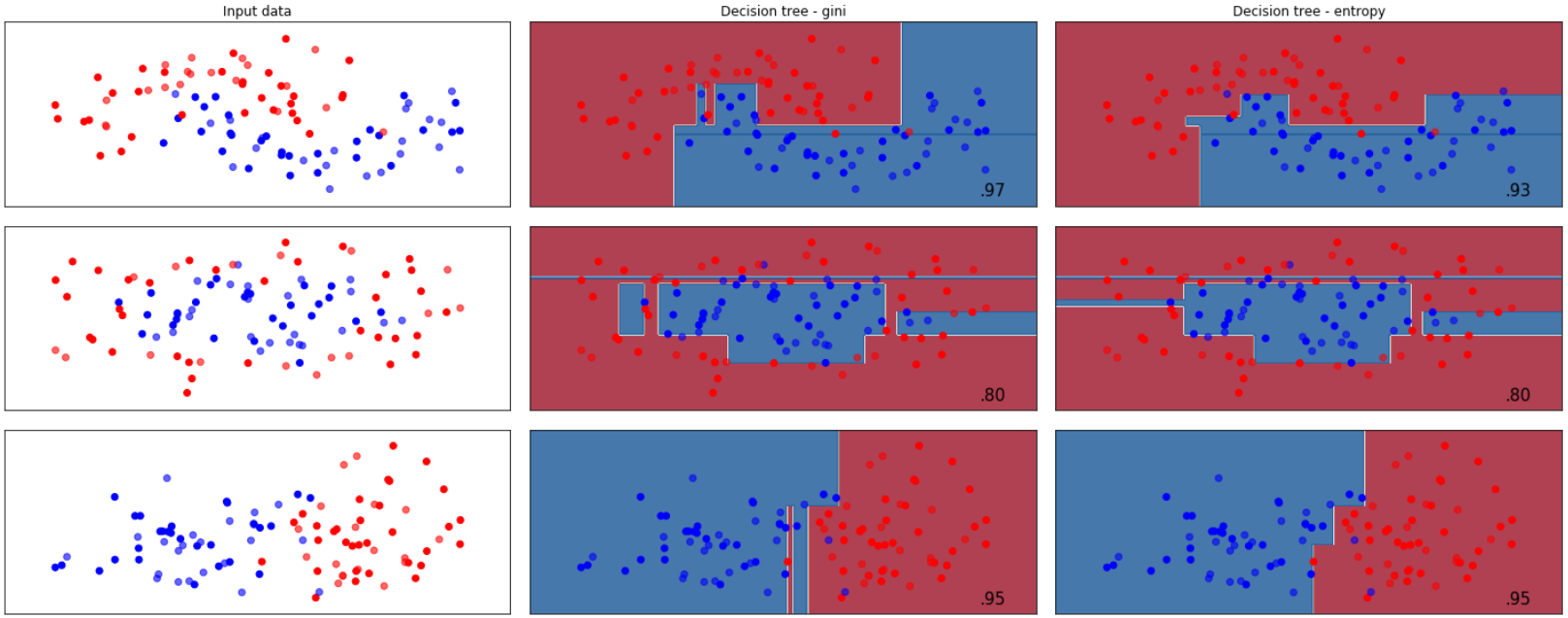
depth = 9





```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```




Input data
Decision tree - gini
Decision tree - entropy
.97
.93
.80
.80
.95
.95

Ensemble learning

Ensembles are methods that combine multiple machine learning models to create more powerful models. Most popular are:

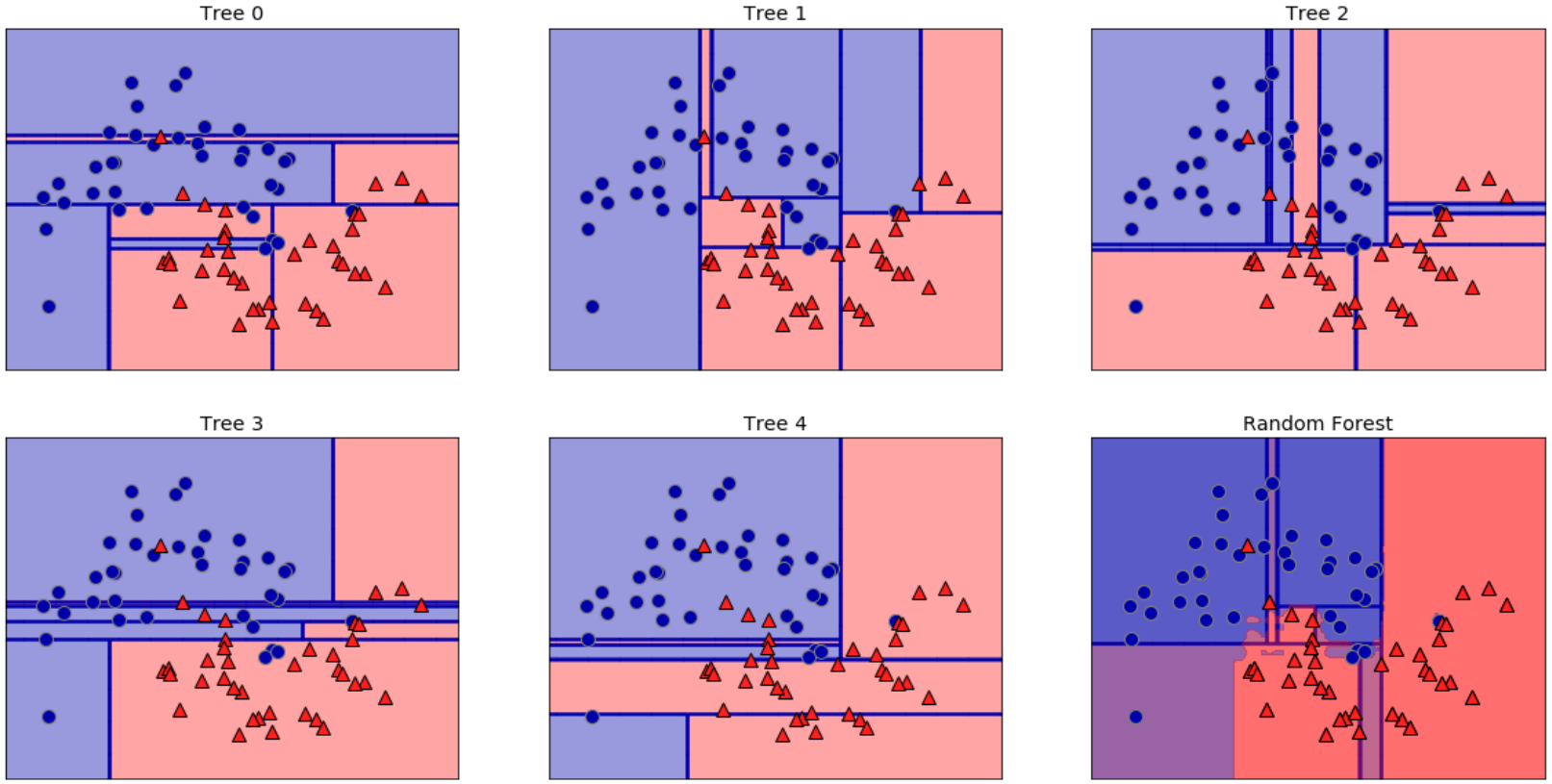
- **RandomForests**: Build randomized trees on random samples of the data
- **Gradient boosting machines**: Build trees iteratively, giving higher weights to the points misclassified by previous trees

In both cases, predictions are made by doing a vote over the members of the example.
Stacking is another technique that builds a (meta)model over the predictions of each member.

RandomForests

Reduce overfitting by averaging out individual predictions (variance reduction)

- Take a *bootstrap sample* of your data
  - Randomly sample with replacement
  - Build a tree on each bootstrap
- Repeat `n_estimators` times
  - Higher values: more trees, more smoothing
  - Make prediction by aggrating the individual tree predictions
    - a.k.a. Bootstrap aggregating (Bagging)
- RandomForest: Randomize trees by considering only a random subset of features of size `max_features` *in each node*
  - Small `max_features` yields more different trees, more smoothing
  - Default: $sqrt(n\_features)$ for classification, $log2(n\_features)$ for regression

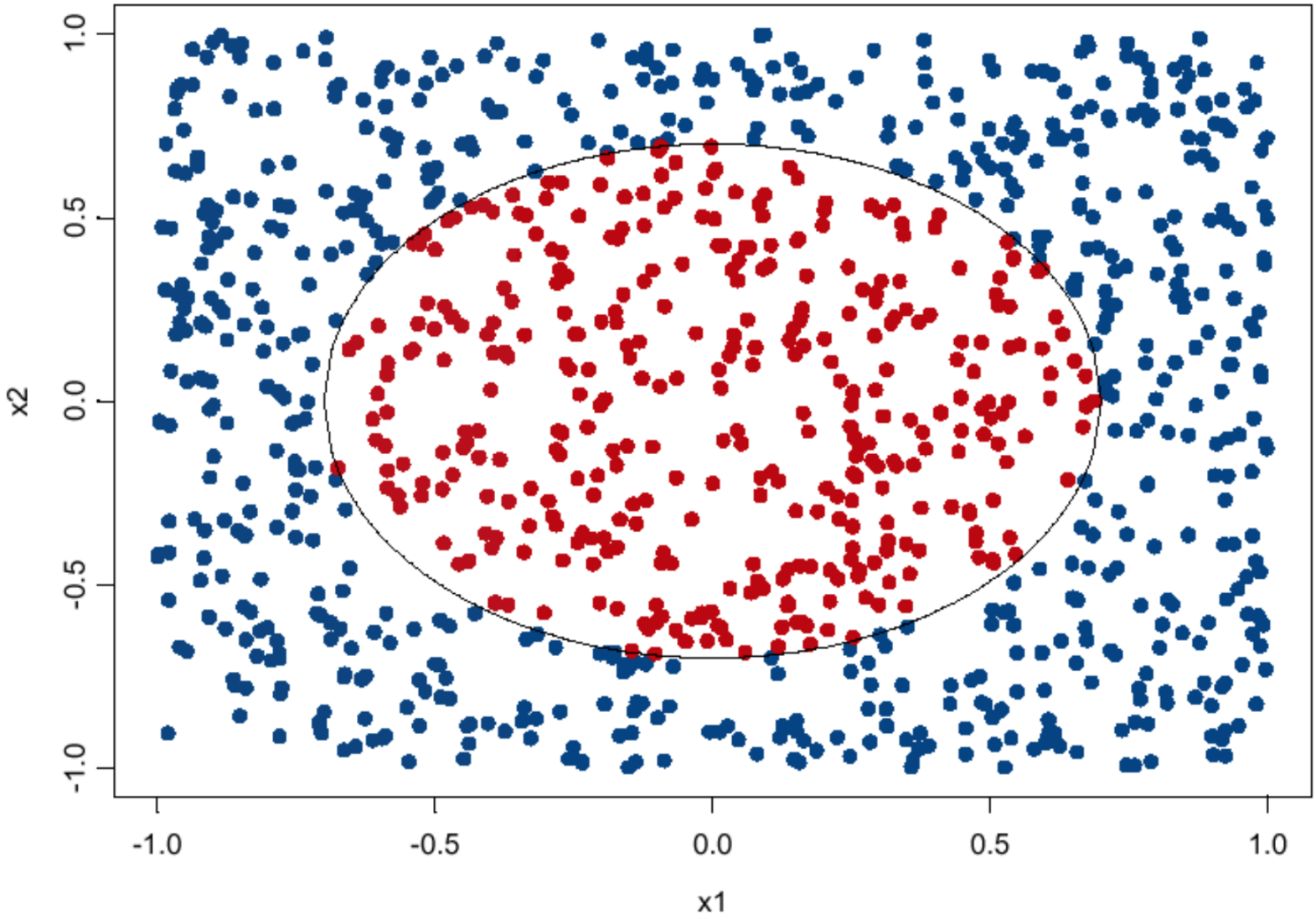

Tree 0
Tree 1
Tree 2
Tree 3
Tree 4
Random Forest

Gradient Boosted Regression Trees (Gradient Boosting Machines)

Instead of reducing the variance of overfitted models, reduce the bias of underfitted models

- Use strong pre-pruning to build very shallow trees
  - Default `max_depth`=3
- Iteratively build new trees by increasing weights of points that were badly predicted
- Example of *additive modelling*: each tree depends on the outcome of previous trees
- Optimization: find optimal weights for all data points
  - Gradient descent (covered later) finds optimal set of weights
  - `learning rate` controls how strongly the weights are altered in each iteration (default 0.1)
- Repeat `n_estimators` times (default 100)

Example:

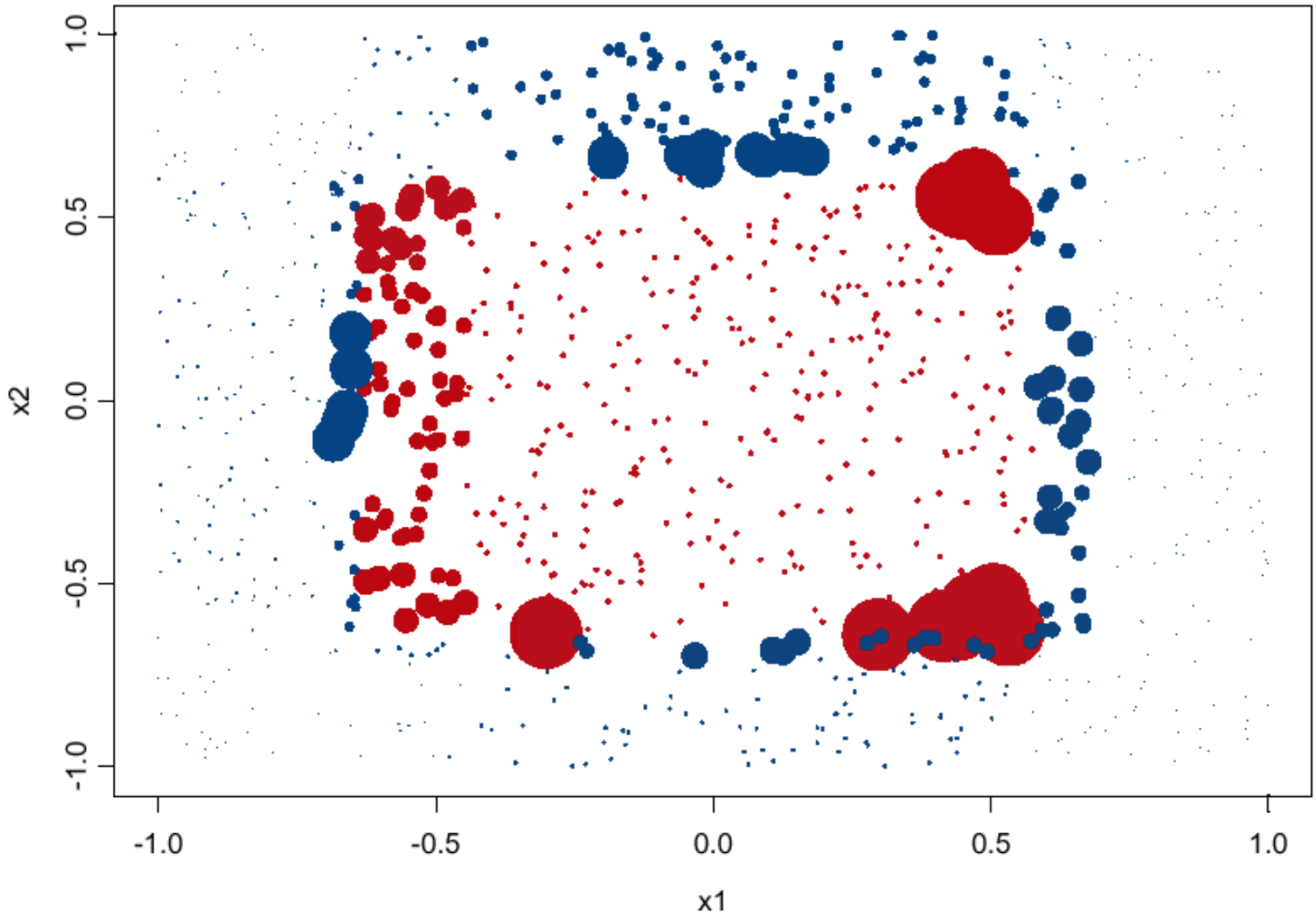


After 1 iteration

- The simple decision tree divides space
- Misclassified points get higher weight (larger dots)

x2
-1.0
-0.5
0.0
0.5
1.0
x1
-1.0
-0.5
0.0
0.5
1.0

After 3 iterations



After 20 iterations