Artificial Intelligence 2

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Introduction

We will describe **agents that can improve** their behavior through diligent study of their own experiences.

- decision trees
- regression
- artificial neural networks
- nonparametric models
- support vector machines
- ensemble learning (boosting)

Why learning is important?

Why would we want an agent to learn (instead of just program in that improvement)?

- the designer **cannot anticipate all possible situations** that the agent might find itself in
  - a robot designed to navigate mazes must learn the layout of each new maze it encounters
- the designer **cannot anticipate all changes over time**
  - a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust
- sometimes human programmers **have no idea how to program a solution** themselves
  - most people are good at recognizing that faces, but even best programmers are unable to program a computer to accomplish that task

Forms of learning

Any component of an agent can be improved by learning.

The improvements, and the techniques used to make them, depend of four major factors:

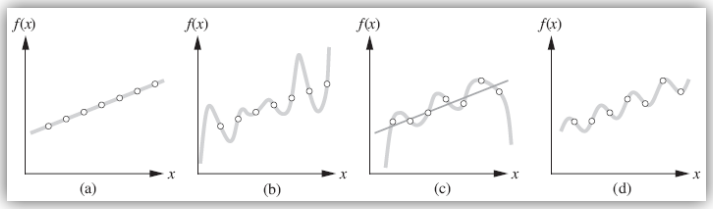
- which **component** is to be improved
  - utility function, mapping from conditions to actions,, …
- what **prior knowledge** the agent already has;
  what **representation** is used for the data and the component
  - logical models, Bayes networks
- what feedback is available to learn from
  - **unsupervised learning**
    - the agent learns patterns in thee input even though no explicit feedback is supplied
  - **reinforcement learning**
    - the agent learns from a series of reinforcements – rewards or punishments
  - **supervised learning**
    - the agent observes some example input-output pairs and learns a function that maps from input to output.

Supervised learning

Given a **training set** of N example input-output pairs $(x_1,y_1),\ldots,(x_N,y_N)$, where $y_i = f(x_i)$ for some unknown function f

Discover a function h, that approximates the true function f.

- function h – **hypothesis** – is selected from a hypothesis space (for example linear functions)



- **hypothesis** is **consistent**, if $h(x_i) = y_i$
- the accuracy of hypothesis is measured using a **test set** of examples

Types of tasks:

- **classification**: the set of outputs y_i is a finite set (such as sunny, cloudy or rainy)
- **regression**: outputs are numbers (such as temperature)

Ockham's razor

How do we choose from among **multiple consistent hypotheses**?

Prefer **the simplest hypothesis** consistent with the same data.

- There is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better (overfitting).

How to define simplicity?

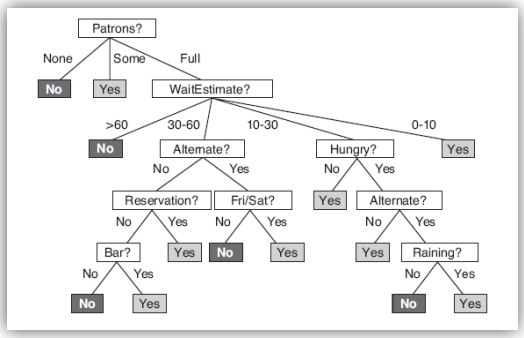
- for example a degree-1 polynomial is simpler than a degree-7 polynomial

The above principle is called **Ockham's razor** – the simplest explanation is probably the correct one.

Decision trees

Decision tree is one of the simplest and yet most successful forms of leaned functions – it takes as input a vector of attribute values and returns a „decision“ a single output value.

- a decision tree reaches its decisions by performing a sequence of tests



Assume a binary decision (Boolean classification)

- for n attributes the decision function can be described using a table with 2^n rows
- that means there are 2^{2^n} different functions
- each such function can be described using a decision tree of maximal depth n

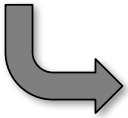
How can we find a small consistent decision tree?

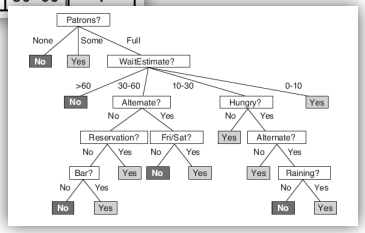
Decision trees - example

The hypothesis space is defined by a set of decision trees and we want a tree that is consistent with the examples and is as small as possible.

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

- examples in the form (**x**, y)
- we assume Boolean decisions
- Will we wait in a restaurant?





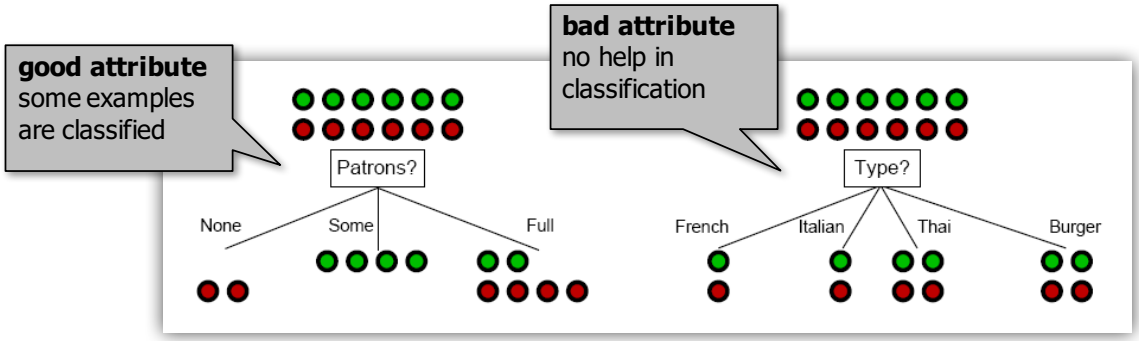
Inducing decision trees

We will construct a small (but not smallest) consistent decision tree by adopting a greedy **divide-and-conquer strategy**:

- select the most important attribute first
- divide the examples based on the attribute value
- when the remaining examples are in the same category, then we are done; otherwise solve smaller sub-problems recursively

What is the "**most important attribute**"?

- that one that that makes the most difference to the classification of examples



Decision tree learning

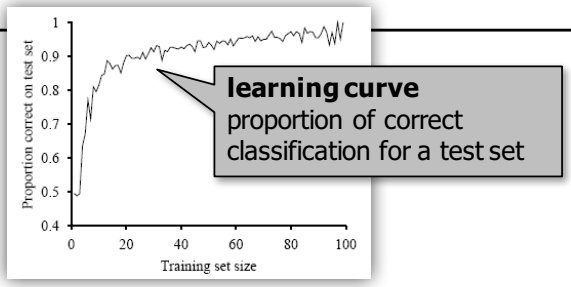
Algorithm ID3

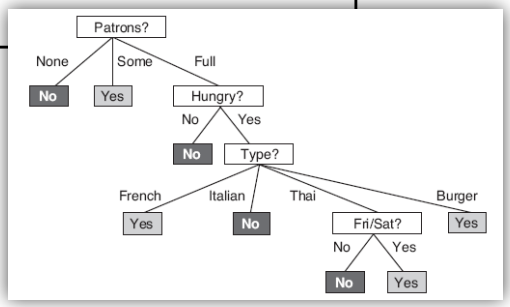
```
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value v_i of best do
            examples_i ← {elements of examples with best = v_i}
            subtree ← DTL(examples_i, attributes − best, MODE(examples))
            add a branch to tree with label v_i and subtree subtree
        return tree
```

the most frequent output from the parent

the most frequent output in examples

the most informative attributed





Choosing attribute tests

How to select the best attribute for the decision?

- We will use the notion of **information gain**, which is defined in terms of **entropy**.
- **Entropy** is a measure of the uncertainty of a random variable.
  - measured in "bits" of information that we obtain after knowing the value of the random variable
    - a coin that always comes up heads – has no uncertainty and thus its entropy is defined as zero
    - a flip of a fair coin is equally likely to come up heads or tails, this counts as "1 bit" entropy
    - The roll of a fair four-sided die has 2-bits of entropy

  $H(V) = -\Sigma_k\, p(v_k)\, \log_2(p(v_k))$, where v_k are values of random variable V

  $B(q) = -q.\log_2 q - (1-q).\log_2(1-q)$ entropy of a Boolean variable

  $H(Goal) = B(p/(p+n))$ entropy of a set of p positive and n negative examples
- An attribute A divides the set examples into subsets based on its value
  - The expected entropy remaining after testing attribute A
    $Remainder(A) = \Sigma_k\, B(p_k/(p_k+n_k)).(p_k+n_k)/(p+n)$
  - the **information gain** from the attribute test on A
    $Gain(A) = B(p/(p+n)) - Remainder(A)$
  - Gain(Patrons) ≈ 0.541 Gain(Type) = 0

Overfitting

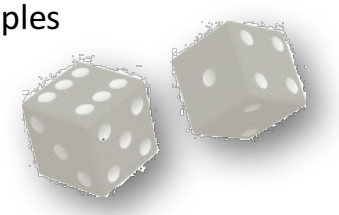
On some problems the algorithm will generate a large tree when there is actually no pattern to be found.

Example: Consider the problem of trying to predict whether the roll of a die will come up as 6 or not. Suppose each training example to include attributes for the color of the die, its weight, whether the experimenters had their fingers crossed etc..

- the learning algorithm will seize any pattern it can find in the input
- if the dice is fair, the right thing to learn is a tree with as ingle node that says "no"

This problem is called **overfitting**.

- occurs even when the target function is not at all random
- becomes more likely as the hypothesis space and the number of input attributes grows
- less likely as we increase the number of training examples



Decision tree pruning

A technique called **decision tree pruning** combats overfitting.

- take a test node that has only leaf nodes as descendants
- If the test appears to be irrelevant – detecting only noise in the data – then eliminate the test, replacing it with a leaf node

How do we detect that a node is testing an **irrelevant attribute**?

- using a statistical **significance test** (χ^2 test)
  - assume that there is no underlying pattern (**null hypothesis**)
  - calculate the extent to which the actual data deviate from a perfect absence of pattern

    $p'_k = p \cdot (p_k+n_k)/(p+n)$ $\quad$ $n'_k = n \cdot (p_k+n_k)/(p+n)$

    $\Delta = \Sigma_k (p_k-p'_k)^2/p'_k + (n_k-n'_k)^2/n'_k$
  - We can use a χ^2 table to see if a particular Δ value confirms or rejects the null hypothesis.

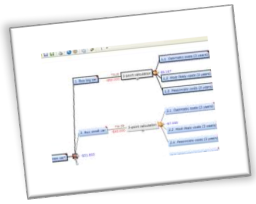
One might think that χ^2 pruning can be used already when constructing the decision tree (**early stopping**).

- The problem with early stopping is that it stops us from recognizing situations where there is no one good attribute, but there are combinations of attributes that are informative (consider the XOR function of two binary attributes).

Applicability of decision trees

In order to extend decision tree induction to a wider variety problems, a number of issues must be addressed:

- **missing data** (not all the attribute values are known)
  - How show one classify an example? How should one modify the information-gain formula?
  - we can use the most frequent value for the missing value of the attribute
- **multivalued attributes** (each example may have a unique value)
  - information gain measure gives an inappropriate indication of the attribute's usefulness
  - It is possible to split the examples based on just one value of the attribute leaving the remaining values to be possibly tested later in the tree.
- **continuous and integer-valued input attributes**
  - infinitely many values may be split using the **split** point that gives the highest information gain (start by sorting the values of the attribute, and then consider only the split points that are between two examples in sorted order that have different classifications
  - splitting is the most expensive part of real-world decision tree learning applications
- **continuous-valued output attributes**
  - for predicting a numeric output value we need a **regression tree** where each leaf has a linear function of some subset of numerical attributes
  - The learning algorithm must decide when to stop splitting and begin applying regression

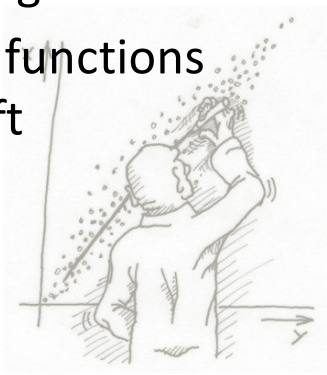


One important property of decision trees is that it is possible for a human to **understand the reason for the output** of the learning algorithm (this property is not shared by other formalisms such as neural networks).

How to handle continuous-valued inputs?

The hypothesis space will consists of **linear functions**.

- we will start with the simplest case: regression with a univariate linear function ("fitting straight line")
- then we will cover multivariate linear regression
- finally, we will show how to turn linear functions into classifiers by applying hard and soft thresholds



Univariate linear regression

Hypothesis is expressed in the form $y = w_1.x + w_0$

Let $h_{\mathbf{w}}(x) = w_1.x + w_0$, where $\mathbf{w} = [w_0, w_1]$

We are looking for a **hypothesis** $h_{\mathbf{w}}$, that fits best the given examples (we are looking for weights w_1 and w_0).

How to measure the error with respect to data?

- square loss function, L_2, is traditionally used:

$$\text{Loss}(h_{\mathbf{w}}) = \Sigma_j (y_j - h_{\mathbf{w}}(x_j))^2 = \Sigma_j (y_j - (w_1.x_j + w_0))^2$$

We are looking for $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} \text{Loss}(h_{\mathbf{w}})$

- which can be done by solving

$$\partial/\partial_{w_0} \Sigma_j (y_j - (w_1.x_j + w_0))^2 = 0$$

$$\partial/\partial_{w_1} \Sigma_j (y_j - (w_1.x_j + w_0))^2 = 0$$

- these equations have a unique solution

$$w_1 = (N \Sigma_j x_j y_j - \Sigma_j x_j \Sigma_j y_j) / (N \Sigma_j x_j^2 - (\Sigma_j x_j)^2)$$

$$w_0 = (\Sigma_j y_j - w_1 . \Sigma_j x_j) / N$$

Beyond linear models

If the hypothesis space is defined by non-linear functions then the equations ∂/∂_{w_i} Loss($h_{\mathbf{w}}$) = 0 will often have no closed-forum solution.

We will use **gradient descent**

- choose any starting point in weight space
- move to a neighboring point that is downhill

  $w_i \leftarrow w_i - \alpha\, \partial/\partial_{w_i}$ Loss(h_w),

  where α is usually called the **learning rate** (it can be fixed constant, or it can decay over time as the learning process proceeds)
- repeat until convergence

For univariate linear regression we will get:

$w_0 \leftarrow w_0 + \alpha\, \Sigma_j\, (y_j - h_w(x_j))$

$w_1 \leftarrow w_1 + \alpha\, \Sigma_j\, (y_j - h_w(x_j)).x_j$

Multivariate linear regression

Hypothesis space is the set of functions of the form $h_{\mathbf{w}}(x) = w_0 + \Sigma_i\, w_i x_i$

We can add a dummy input attribute, which is defined as always equal to 1:

$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$

Multivariate linear regression problem can be solved analytically by finding weight that minimizes loss ∂/∂_{w_i} Loss($h_{\mathbf{w}}$) = 0

$\mathbf{w}^* = (X^TX)^{-1}\, X^T\, y$

where X be the **data matrix** (the matrix of inputs with one n-dimensional example per row)

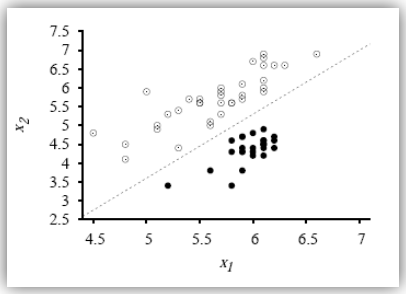
Or we can gradient descent

$w_i \leftarrow w_i + \alpha\, \Sigma_j\, (y_j - h_{\mathbf{w}}(\mathbf{x}_j)).x_{j,i}$

Linear classifiers

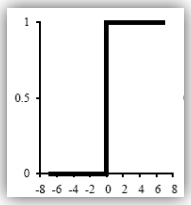
Linear functions can be used to do classification as well as regression.

- **linear separator**
  - we are looking for h_w such that
    - $h_{\mathbf{w}}(\mathbf{x}) = 1$ if $\mathbf{w}.\mathbf{x} \geq 0$, otherwise 0
  - Alternatively, we can think of h as the result of passing the linear function wx through a **threshold function**:
    - $h_{\mathbf{w}}(\mathbf{x})$ = Threshold($\mathbf{w}.\mathbf{x}$), where Threshold(z) = 1, if $z \geq 0$, otherwise 0



- **perceptron learning rule**

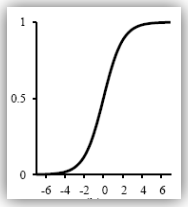
  $w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})).x_i$
  - if the output is correct, then the weights are not changed
  - if $h_{\mathbf{w}}(\mathbf{x}) \neq y$, then the weight is increased/decreased based on x_i



- we can soften the threshold function by using **logistic threshold function**

  $\text{Threshold}(z) = 1 / (1+e^{-z})$

  $w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x}))\,.\,h_{\mathbf{w}}(\mathbf{x}).(1-h_{\mathbf{w}}(\mathbf{x})).x_i$
  - one of the most popular classification technique



Nonparametric models

When we learn the hypothesis, for example via linear regression, we can throw away the training data.

A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a **parametric model**.

When there are thousands or billions of examples to learn from, it seems like a better idea to let the **data speak for themselves** rather than forcing them to speak through a tiny vector of parameters.

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters.

Table lookup: a new example **x** is looked for in a lookup table of all training examples and it is there, return the corresponding y.

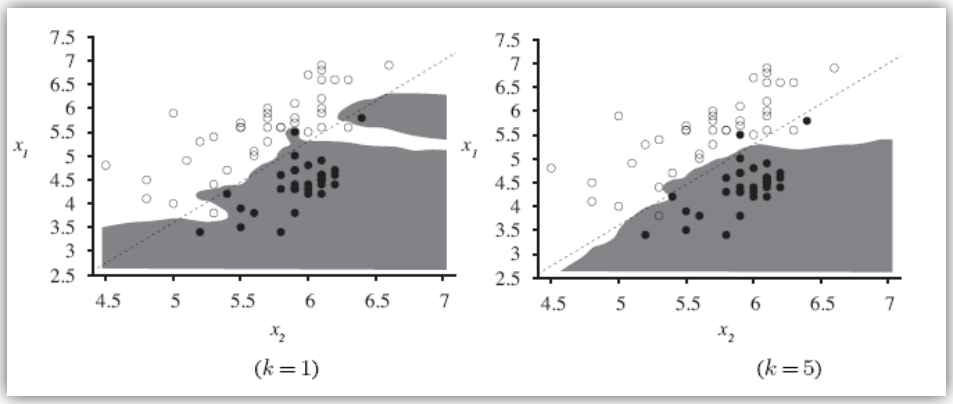
When **x** is not in the table, all the method can do is returning some default value.

Nearest neighbor models

Which value to return if the example is not in the lookup table?

Find the **k** examples that are **nearest** to **x (k-nearest neighbors lookup)** and compose the answer from their y values.

- to do classification take the plurality vote for the neighbors (which is a majority vote in the case of binary classification); to avoid ties, k is always chosen to be an odd number



Nearest neighbors – distance

How do we measure the distance?

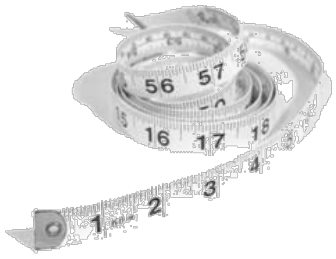
Typically, distances are measured with a **Minkowski distance** defined as

$$L^p(\mathbf{x}_j,\mathbf{x}_q) = (\Sigma_i\ |x_{j,i} - x_{q,i}|^p)^{1/p}$$

- p = 1: **Manhattan distance**
- p = 2: **Euclidian distance**
- with Boolean attribute values, the number of attributes on which two points differ is called the **Hamming distance**

Be careful about the scale!

- it is common to apply **normalization**
- instead of $x_{j,i}$ we can use $(x_{j,i} - \mu_i)/\sigma_i$, where μ_i je is the mean value and σ_i is standard deviation



Nearest neighbors – properties

The curse of dimensionality

- in low-dimensional spaces with plenty of data, nearest neighbors work well
- but as the number of dimension rises we encounter a problem: the nearest neighbors in high-dimensional spaces are usually not very near!

Looking for neighbors

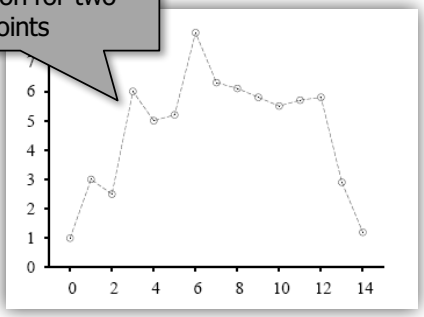
How do we actually find the nearest neighbors?

- **table lookup**: finding an element takes time O(N)
- **binary tree**: finding an element takes time O(log N), but the neighbors might be at different branches
  - works fine if the number of examples is exponential in the number of attributes
- **hash table**: finding an element takes time O(1)
  - we need locally-sensitive hash (LSH) – near points are grouped together in the same bin
  - with a clever used of randomized algorithms, we can find an approximate solution

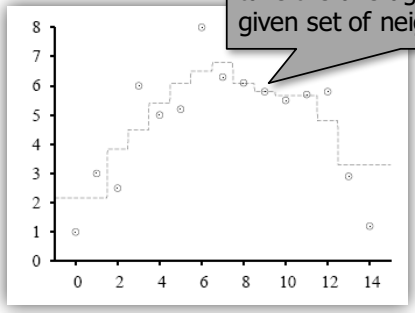
Nonparametric regression

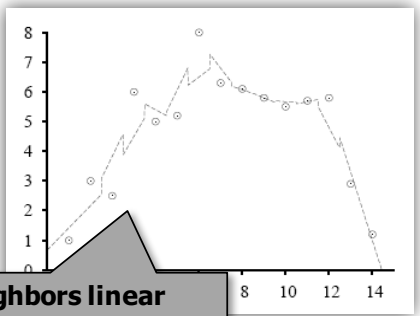
We can apply nonparametric approaches to **regression**.

Connect the dots
linear regression for two neighboring points

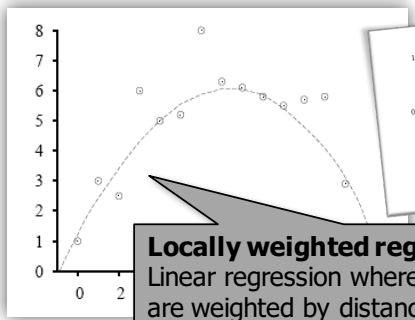


3-nearest neighbors average
take the average y value for a given set of neighbors





3-nearest neighbors linear regression
linear regression of neighboring points



Locally weighted regression
Linear regression where the examples are weighted by distance via the **kernel function** K

$\mathbf{w}^* = \text{argmin}_w \Sigma_j K(\text{Dist}(x_q, x_j))(y_j - \mathbf{w}.\mathbf{x}_j)^2$

The support-vector machine (SVM) is currently the most popular approach of „off-the-shelf“ supervised learning.

There are three properties that make SVM attractive:

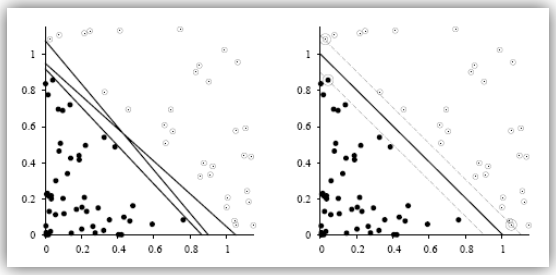
- SVMs construct a **maximum margin separator** – a decision boundary with the largest possible distance to example points
- SVMs create a linear separating hyperplane, but they have the ability to embed the data into a higher-dimensional space, using the so-called **kernel trick**
- SVMs are a **nonparametric method** (in practice they often end up retaining only a small fraction of the number of examples)

Maximum margin separator

Some examples are more important than others, and paying attention to them can lead to better generalization!

Examples closer to the separator are more important.

SVMs use the **maximum margin separator** (the separator that is farthest away from the examples)



- can be found via **dual representation** by solving

  $\text{argmax}_\alpha \Sigma_j \alpha_j - \frac{1}{2} \Sigma_{j,k} \alpha_j \alpha_k y_j y_k (\mathbf{x}_j.\mathbf{x}_k)$, where $\alpha_j \geq 0$, $\Sigma_j \alpha_j y_j = 0$
- this is a quadratic programming optimization problem
- the data enter thee expression only in the form of dot products of pairs of points

The expression of the separator itslef looks as:

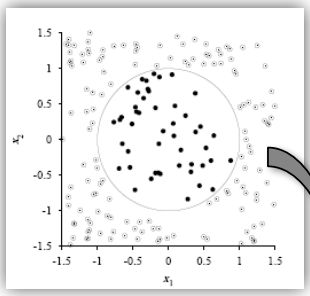
$h(\mathbf{x}) = \text{sign}(\Sigma_j \alpha_j y_j (\mathbf{x}.\mathbf{x}_j) - b)$

In the original representation it looks as $\mathbf{w} = \Sigma_j \alpha_j . \mathbf{x}_j$

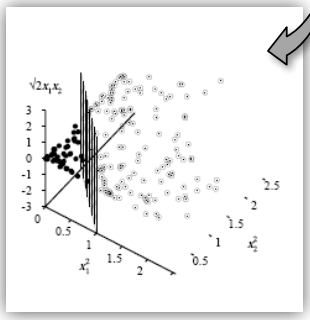
Important property

- the weights α_j associated with each data point are zero except for the **support verctors** – the points closest to the separator
- SVMs gain advantages of parametric models (we keep only a few examples such that $\alpha_j \neq 0$)

Kernel function



$f_1 = (x_1)^2$
$f_2 = (x_2)^2$
$f_3 = \sqrt{2}\, x_1 x_2$



What if the examples are not linearly separable?

The input vector can be mapped via F to a new vector of feature values.

Then we look for a linear separator between points $F(x_j)$ instead of x_j.

It turns out that $F(x_j)\, F(x_k)$ can often be computed without first computing F for each point.

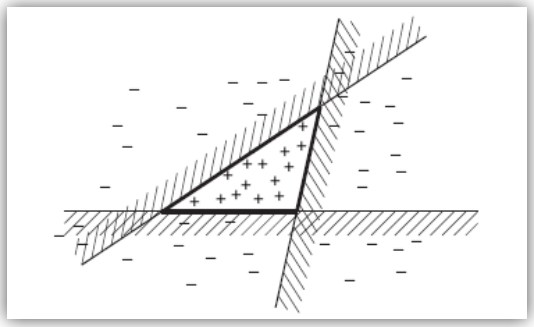
- $F(x_j)\, F(x_k) = (x_j.x_k)^2$
- this expression is called a **kernel function** $K(x_j,x_k)$
- the polynomial kernel $K(x_j,x_k) = (1+x_jx_k)^d$ corresponds to a feature space whose dimension is exponential in d

Ensemble learning

So far we have looked at learning methods in which a **single hypothesis** is used to make predictions.

The idea of **ensemble learning** methods is to select a collection (ensemble) of hypothesis and combine their predictions

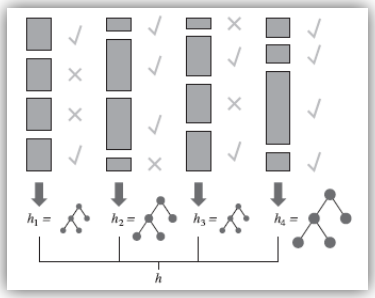
- the hypotheses vote on the best classification for a new example – this **decreases the chances of misclassification**
- it is also a generic way of **enlarging the hypothesis space**
  - linear classifiers can be used to describe linearly non-separable area



Boosting

Boosting is a a widely used ensemble method based on a weighted training set:

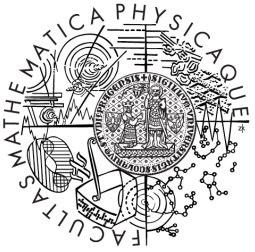
- boosting starts with weight 1 for all the examples
- from this set it generates the **first hypothesis**
- we **increase weights** of the misclassified examples, while **decreasing weights** of the correctly classified examples
- we **repeat** generating of a next hypothesis until K hypotheses are obtained
- each hypothesis contributes to the ensemble hypothesis with the weight according to how well it performed on the training set
- even if the underlying learning method is weak (its accuracy is slightly better than random guessing) the algorithm can return a hypothesis that classifies the examples perfectly for large enough K



AdaBoost

function ADABOOST($examples$, L, K) **returns** a weighted-majority hypothesis
  inputs: $examples$, set of N labeled examples $(x_1, y_1), \ldots, (x_N, y_N)$
    L, a learning algorithm
    K, the number of hypotheses in the ensemble
  local variables: $\mathbf{w}$, a vector of N example weights, initially $1/N$
    $\mathbf{h}$, a vector of K hypotheses
    $\mathbf{z}$, a vector of K hypothesis weights

  for $k = 1$ **to** K **do**
    $\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[k](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[k](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
    $\mathbf{w} \leftarrow$ NORMALIZE($\mathbf{w}$)
    $\mathbf{z}[k] \leftarrow \log(1 - error)/error$
  return WEIGHTED-MAJORITY($\mathbf{h}, \mathbf{z}$)



Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz