Machine learning for automated theorem proving: the story so far

Sean Holden

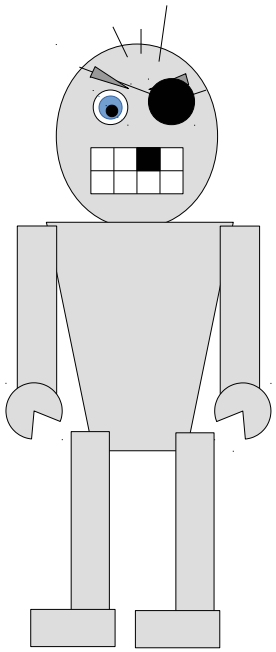
University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD, UK

sbh11@cl.cam.ac.uk

www.cl.cam.ac.uk/∼sbh11

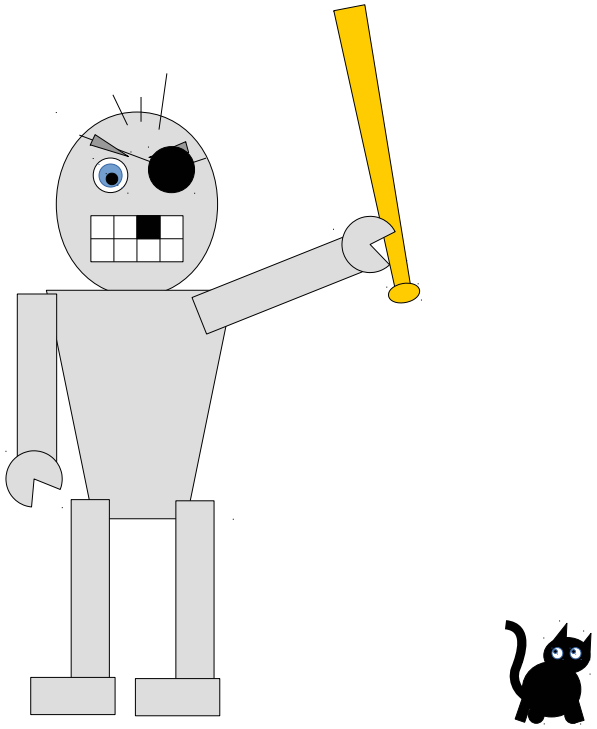
Machine learning: what is it?

EVIL ROBOT...



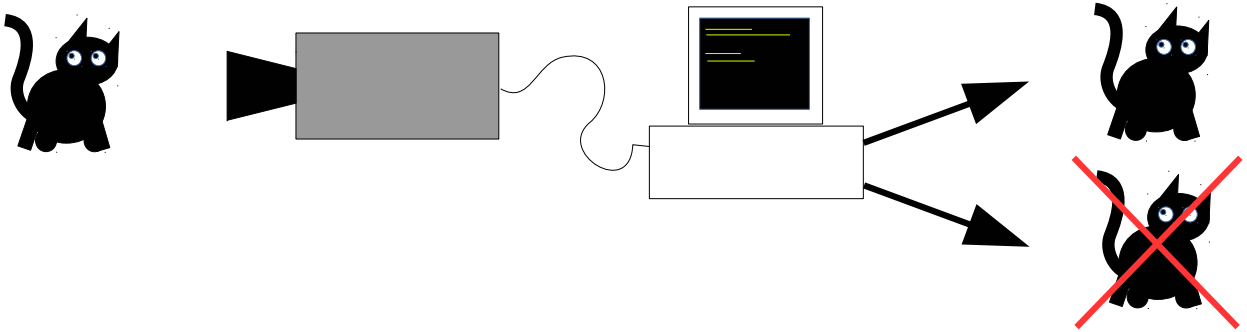
Machine learning: what is it?

EVIL ROBOT...

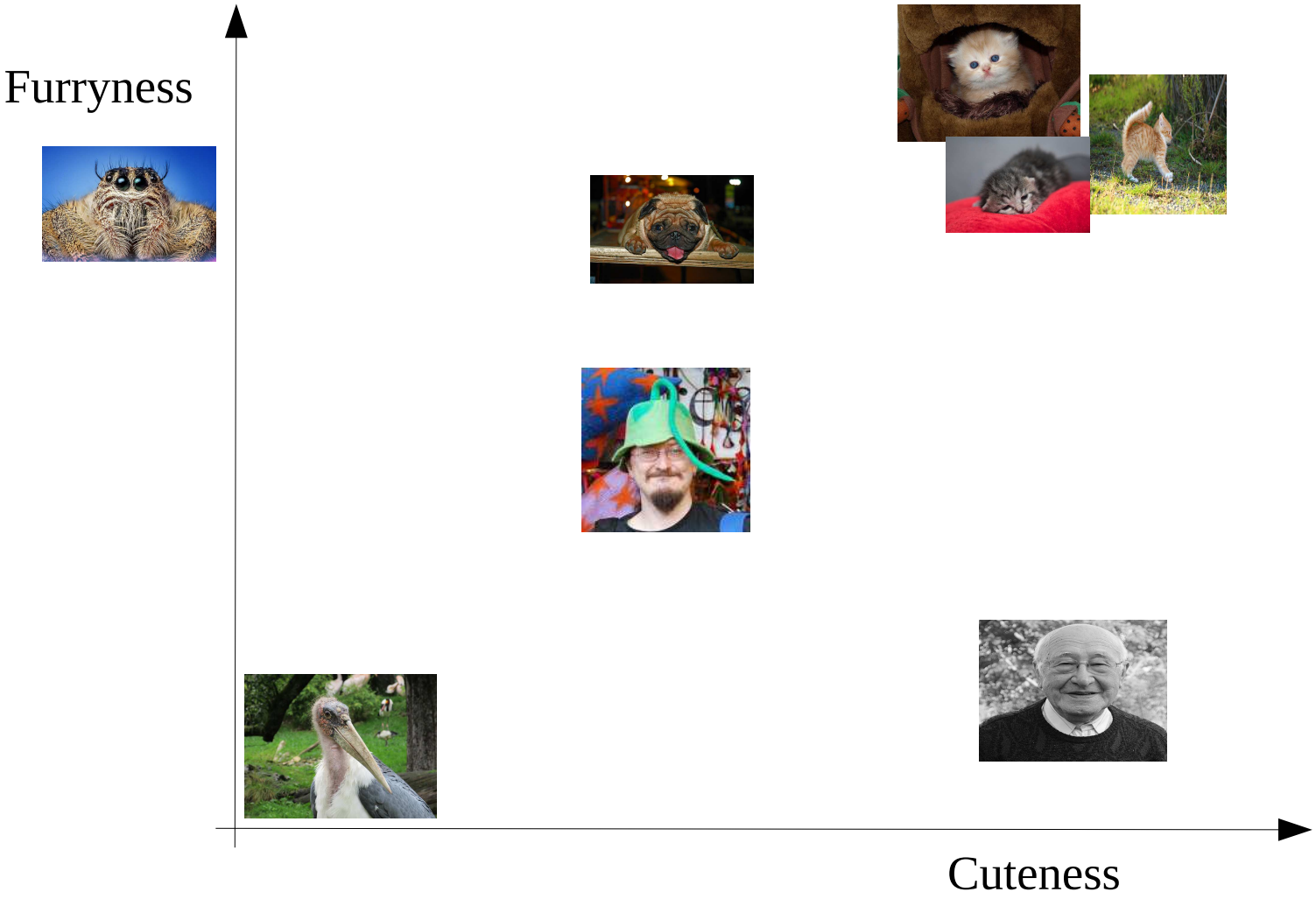


...hates kittens!!!

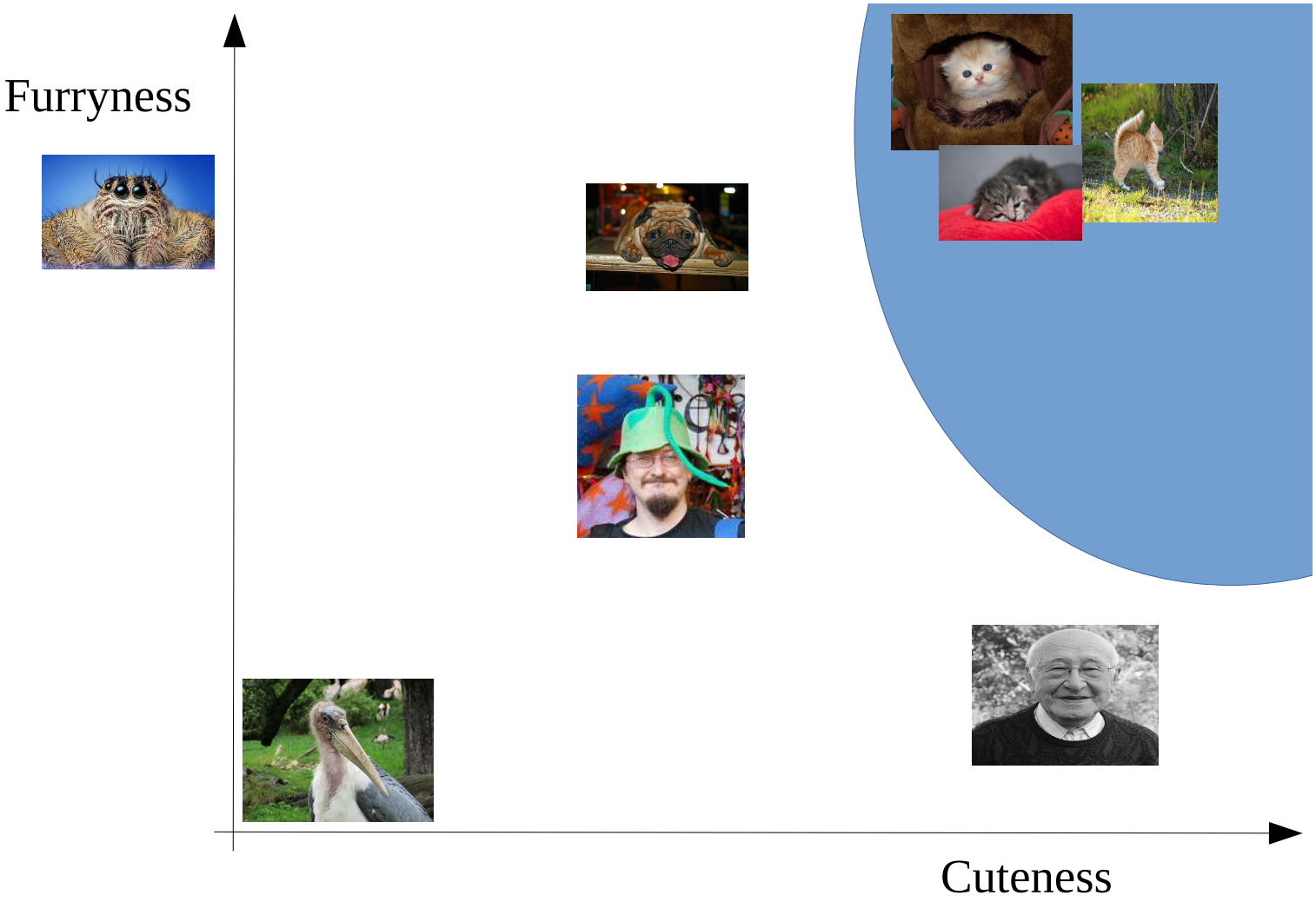
Machine learning: what is it?



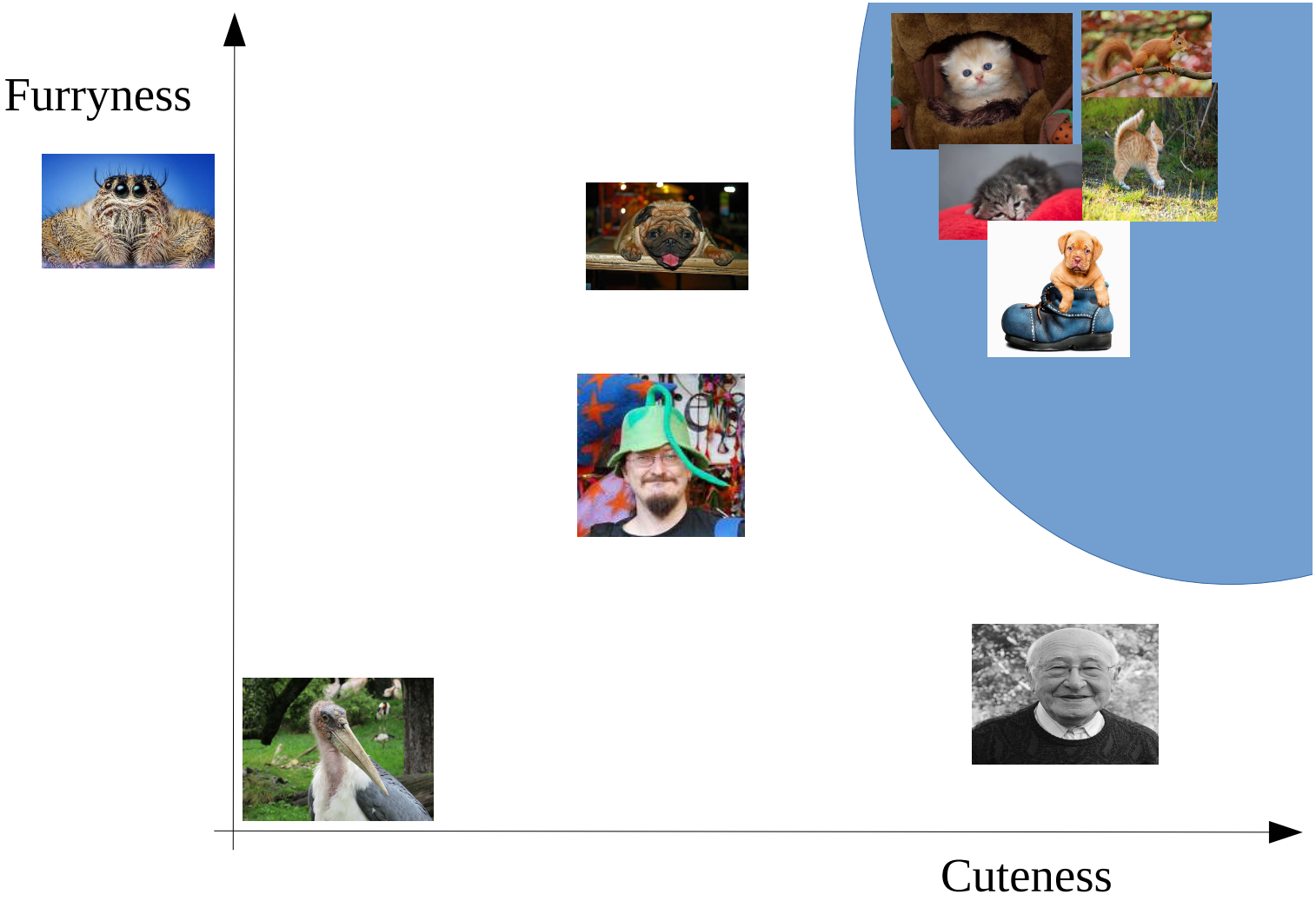
Machine learning: what is it?



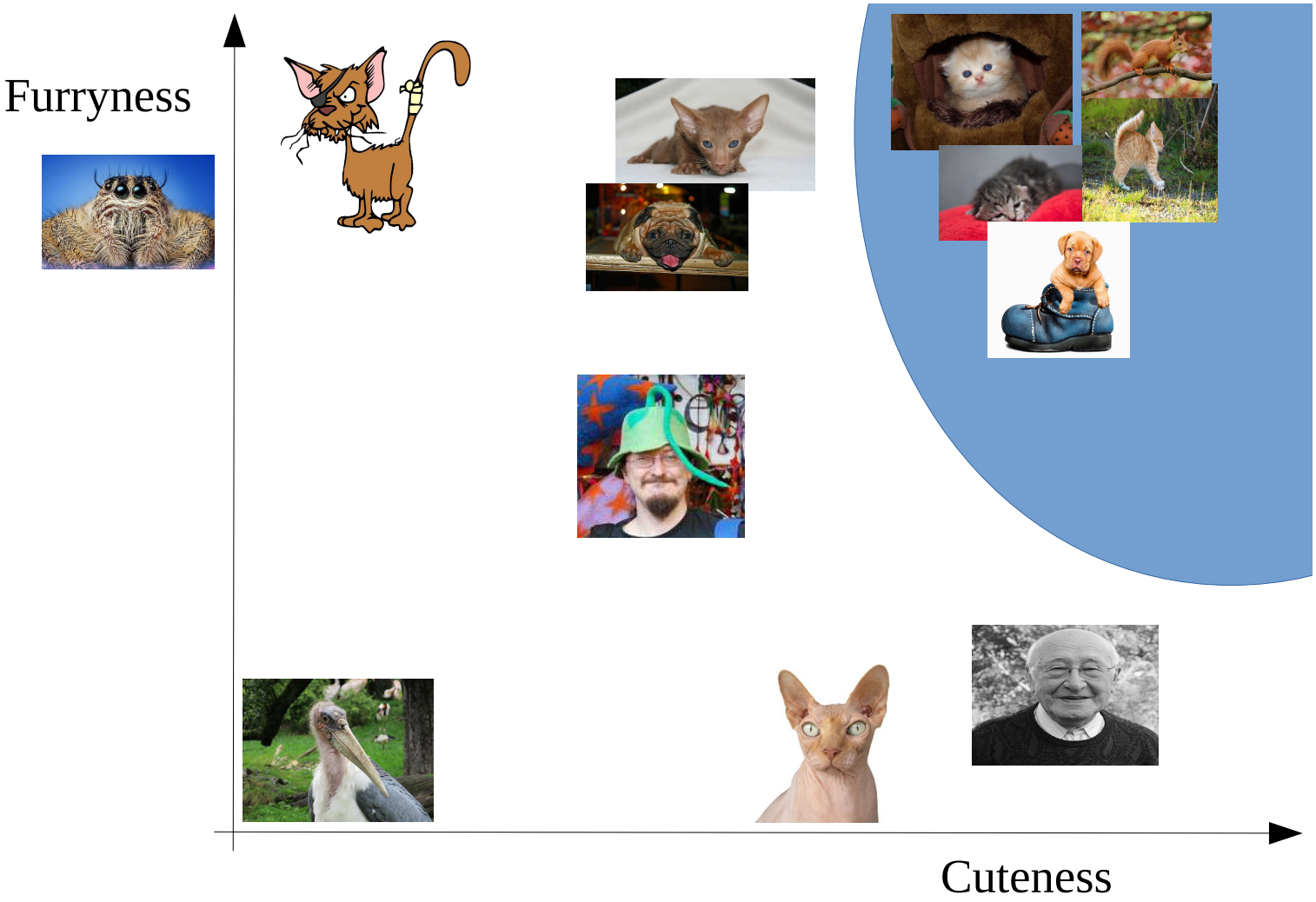
Machine learning: what is it?



Machine learning: what is it?



Machine learning: what is it?



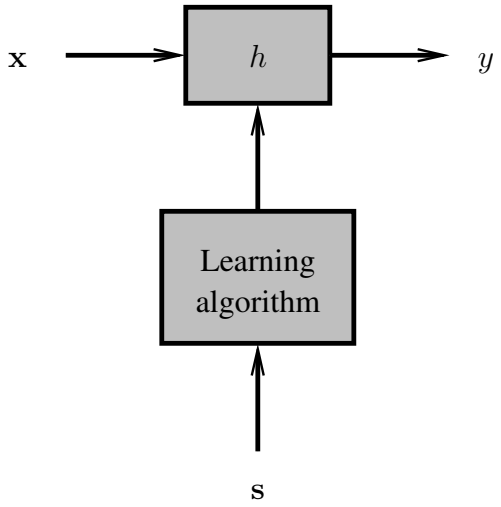
Machine learning: what is it?

I have d *features* allowing me to make vectors $\mathbf{x} = (x_1, \ldots, x_d)$ describing *instances*.

I have a set of m labelled examples

$$\mathbf{s} = ((\mathbf{x}_1, y_1), \ldots (\mathbf{x}_m, y_m))$$

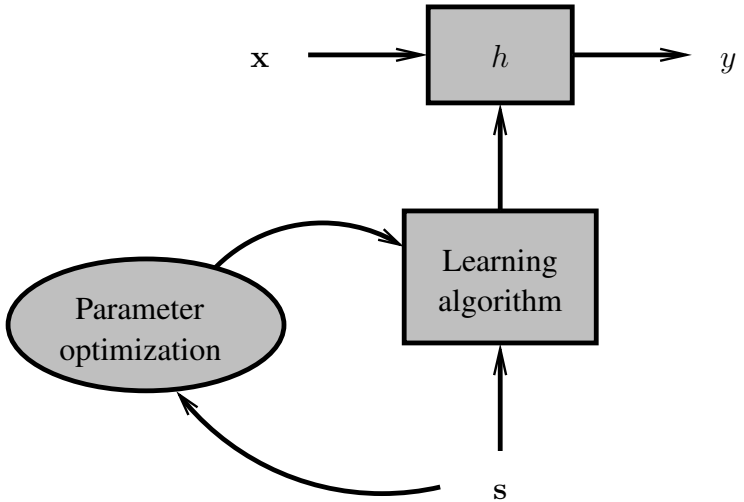
where usually y is either real (regression) or one of a finite number of categories (classification).



I want to infer a function h that can predict the values for y given $\mathbf{x}$ on *all instances*, not just the ones in $\mathbf{s}$.

Machine learning: what is it?

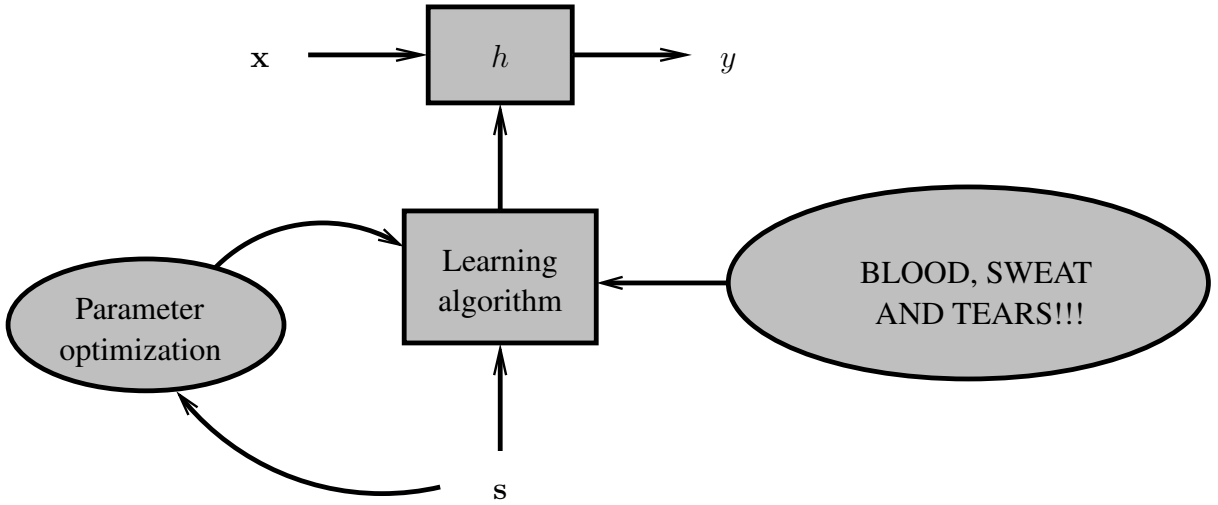
There are a couple of things missing:



Generally we need to optimize some parameters associated with the learning algorithm.

Machine learning: what is it?

There are a couple of things missing:



Generally we need to optimize some parameters associated with the learning algorithm.

Also, the process is far from automatic...

Machine learning: what is it?

So with respect to theorem proving, the key questions have been:

1. What *specific problem* do you want to solve?
2. What are the *features*?
3. How do you get the *training data*?
4. What machine learning *method* do you use?

As far as the last question is concerned:

1. It's been known for a long time that *you don't necessarily need a complicated method*. (**Reference:** Robert C Holt, "Very simple classification rules perform well on most commonly used datasets", *Machine Learning*, 1993.)
2. The chances are that *a support vector machine (SVM) is a good bet*. (**Reference:** Fernández-Delgado et al., "Do we need hundreds of classifiers to solve real world classification problems?", *Journal of Machine Learning Research*, 2014.)

Three examples of machine learning for theorem proving

In this talk we look at three representative examples of how machine learning has been applied to *automatic theorem proving (ATP)*:

1. Machine learning for solving *boolean satisfiability SAT problems* by *selecting an algorithm from a portfolio*.
2. Machine learning for *proving theorems in first-order logic (FOL)* by *selecting a good heuristic*.
3. Machine learning for *selecting good axioms* in the context of an *interactive proof assistant*.

In each case I present the underlying problem, and a brief description of the machine learning method used.

Machine learning for SAT

Given a *Boolean formula*, decide whether it is satisfiable.

There is no single "best" SAT-solver.

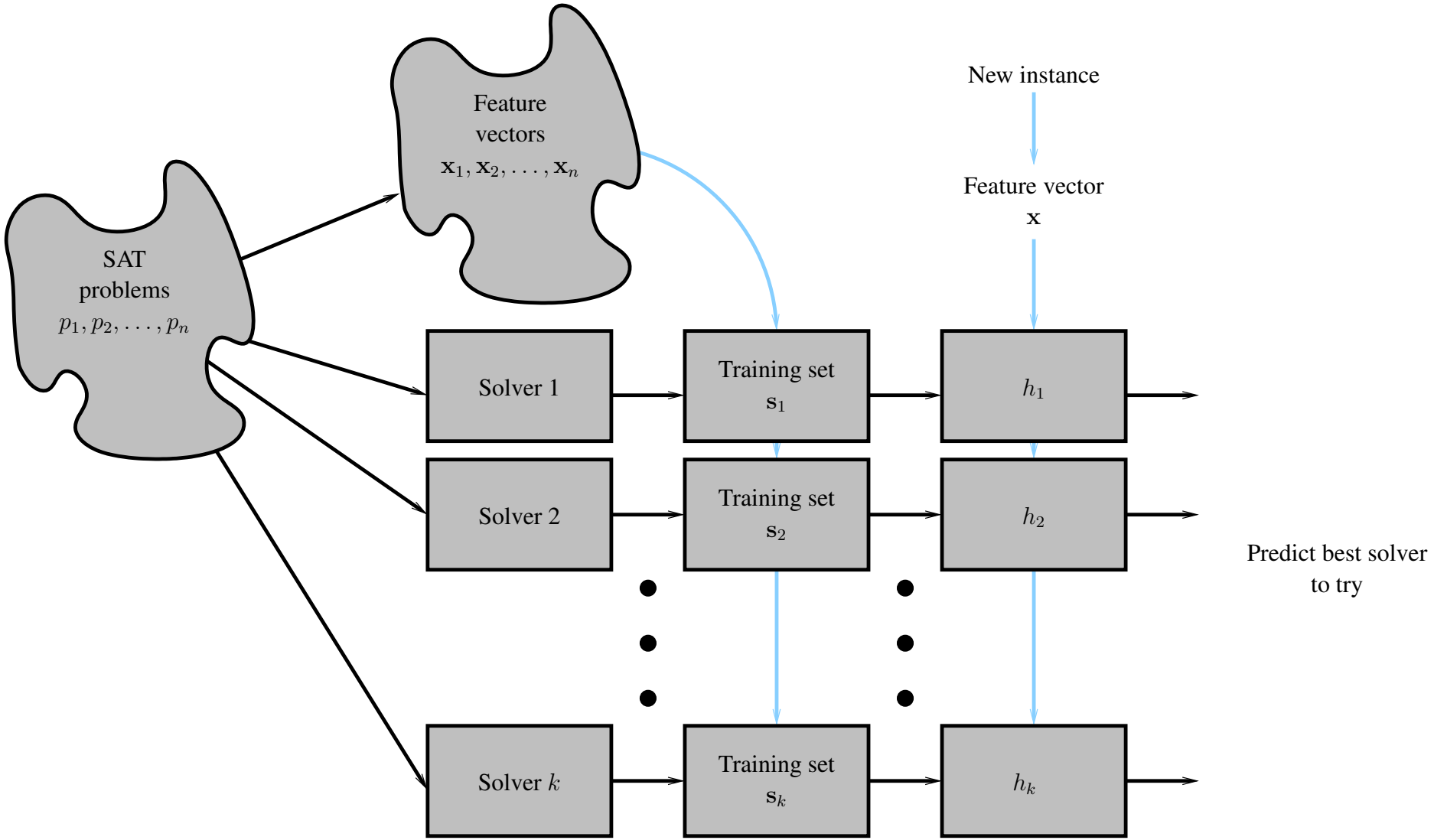
Basic machine learning approach:

1. Derive a *standard set of features* that can be used to describe any formula.
2. Apply a collection of solvers (the *portfolio*) to some training set of formulas.
3. The *running time* of a solver provides the label y.
4. For each solver, train a classifier to predict the *running time* of an algorithm *for a particular instance*.

This is known as an *empirical hardness model*.

Reference: Lin Xu et al, "SATzilla: Portfolio-based algorithm selection for SAT", *Journal of Artificial Intelligence Research*, 2008. (Actually more complex and uses a hierarchical model.)

Machine learning for SAT



Machine learning for SAT

The approach employed *48 features*, including for example:

1. The *number of clauses*.
2. The *number of variables*.
3. The *mean ratio* of *positive and negative literals* in a *clause*.
4. The *mean, minimum, maximum and entropy* of the *ratio of positive and negative occurences* of a *variable*.
5. The *number of DPLL unit propagations* computed at various *depths*.
6. And so on...

Linear regression

I have d *features* allowing me to make vectors $\mathbf{x} = (x_1, \ldots, x_d)$.

I have a set of m labelled examples

$$\mathbf{s} = ((\mathbf{x}_1, y_1), \ldots (\mathbf{x}_m, y_m)).$$

I want a function h that can predict the values for y given $\mathbf{x}$.

In the simplest scenario I use

$$h(\mathbf{x}; \mathbf{w}) = w_0 + \sum_{i=1}^{d} w_i x_i.$$

and choose the *weights* w_i to minimize

$$E(\mathbf{w}) = \sum_{i=1}^{m} (h(\mathbf{x}_i; \mathbf{w}) - y_i)^2.$$

This is *linear regression*.

Ridge regression

This can be problematic: the function h is linear, and computing $\mathbf{w}$ can be numerically problematic.

Instead introduce *basis functions* ϕ_i and use

$$h(\mathbf{x}; \mathbf{w}) = \sum_{i=1}^{d} w_i \phi_i(\mathbf{x})$$

minimizing

$$E(\mathbf{w}) = \sum_{i=1}^{m} (h(\mathbf{x}_i; \mathbf{w}) - y_i)^2 + \lambda ||\mathbf{w}||^2$$

This is *ridge regression*. The optimum $\mathbf{w}$ is

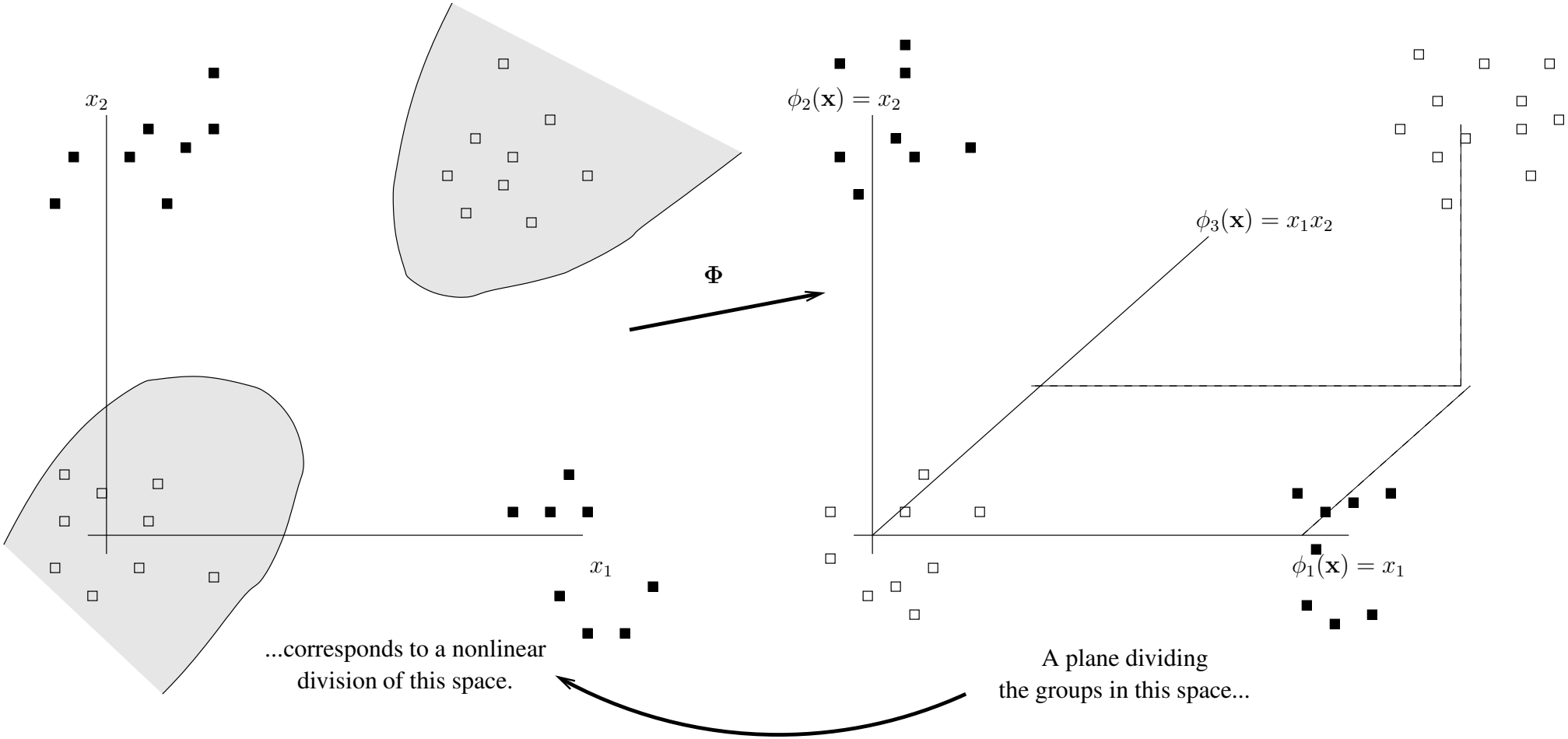
$$\mathbf{w}_{\text{opt}} = \left(\mathbf{\Phi}^T \mathbf{\Phi} + \lambda \mathbf{I}\right)^{-1} \mathbf{\Phi}^T \mathbf{y}$$

where $\Phi_{i,j} = \phi_j(\mathbf{x}_i)$.

Example: in SATzilla, we have linear basis functions $\phi_i(\mathbf{x}) = x_i$ and quadratic basis functions $\phi_{i,j}(\mathbf{x}) = x_i x_j$.

Mapping to a bigger space

Mapping to a *different space* to introduce *nonlinearity* is a common trick:



We will see this again later...

Machine learning for first-order logic

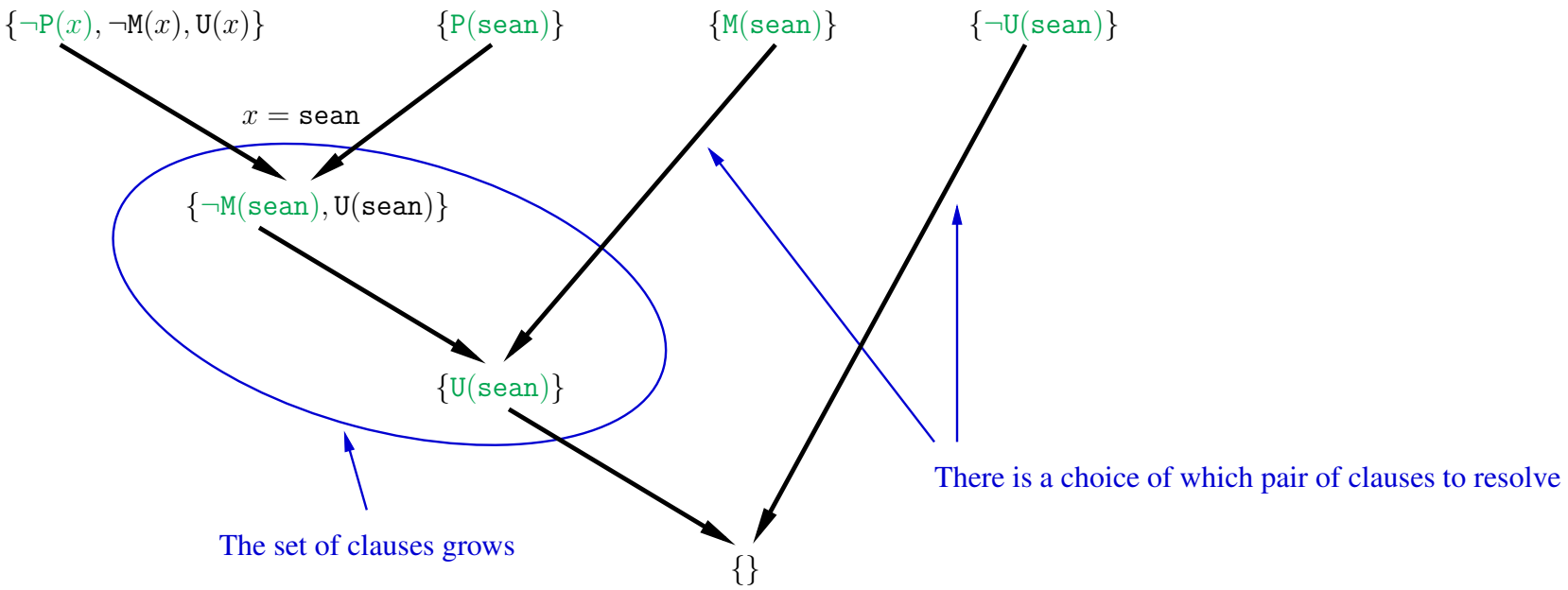
Am I **AN UNDESIRABLE**?

$$\forall x \,.\, \texttt{Pierced}(x) \wedge \texttt{Male}(x) \longrightarrow \texttt{Undesirable}(x)$$

$$\texttt{Pierced}(\texttt{sean})$$

$$\texttt{Male}(\texttt{sean})$$

Does $\texttt{Undesirable}(\texttt{sean})$ follow?



Oh dear...

Machine learning for first-order logic

The procedure has some similarities with the portfolio SAT solvers:

However this time we have a *single theorem prover* and learn to *choose a heuristic*:

1. Convert any *set of axioms* along with a *conjecture* into (up to) 53 features.
2. Train using a *library of problems*.
3. For *each problem* in the library, run the prover with *each available heuristic*.
4. This produces a *training set for each heuristic*. Labels are *whether or not the relevant heuristic is the best (fastest)*.

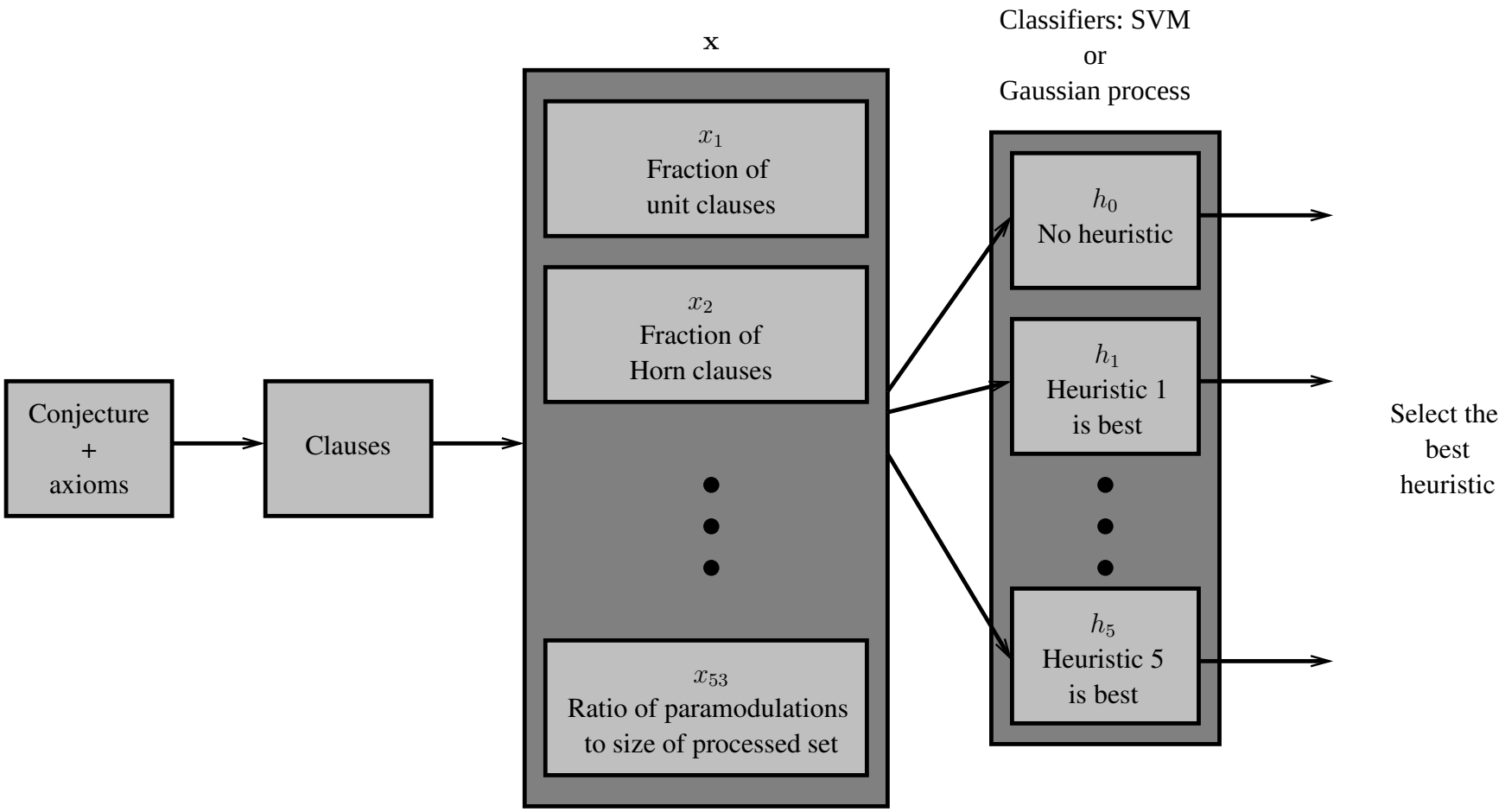
We then train a classifier per heuristic.

New problems are solved using the predicted best heuristic.

Reference: James P Bridge, Sean B Holden and Lawrence C Paulson, "Machine learning for first-order theorem proving: learning to select a good heuristic", *Journal of Automated Reasoning*, 2014.

Machine learning for first-order logic

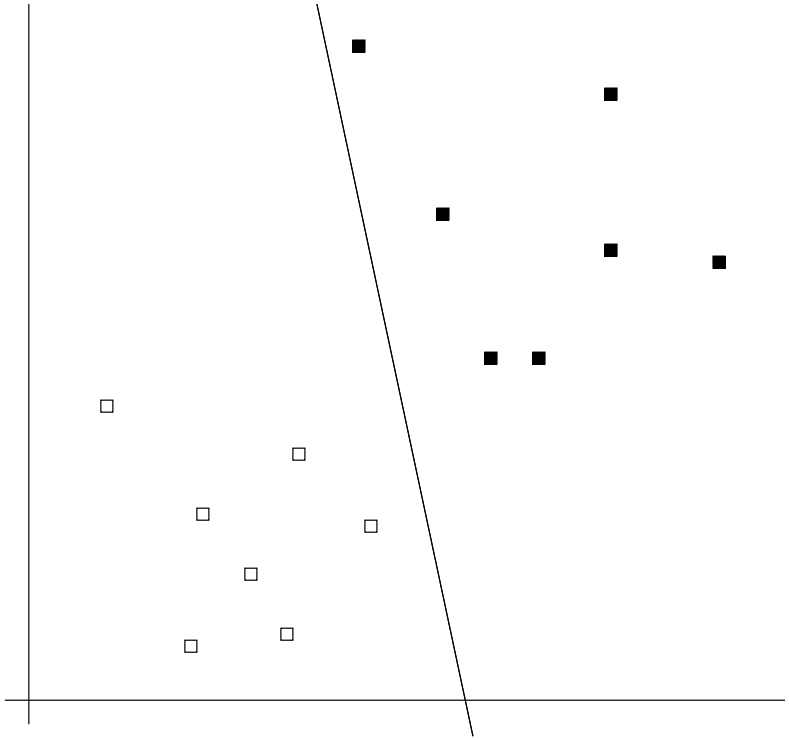
To *select a heuristic* for a *new problem*:



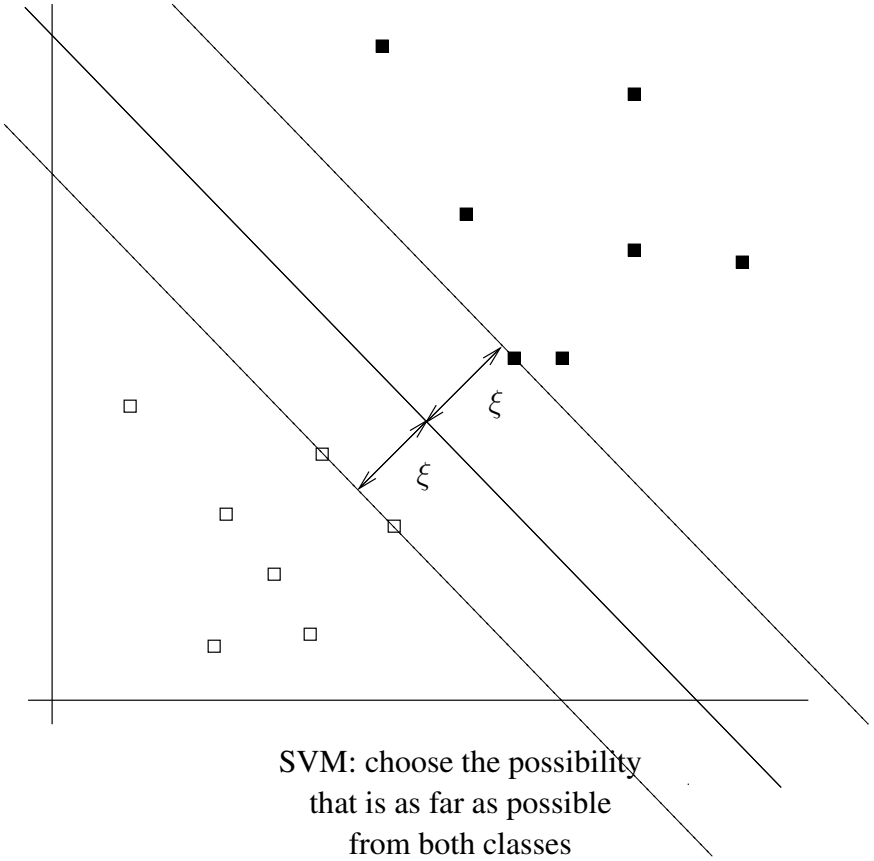
We can also *decline to attempt a proof*.

The support vector machine (SVM)

An SVM is essentially a *linear classifier* in a *new space* produced by Φ, as we saw before:



Linear classifier:
there are many ways
of dividing the classes



SVM: choose the possibility
that is as far as possible
from both classes

BUT the decision line is chosen in a specific way: we *maximize the margin*.

The support vector machine (SVM)

How do we train an SVM?

1. As previously, the basic function of interest is

$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}) + b$$

   and we classify new examples as

$$y = \text{sgn}(h(\mathbf{x})).$$

2. The *margin* for the ith example $(\mathbf{x}_i, y_i)$ is

$$M(\mathbf{x}_i) = y_i h(\mathbf{x}_i).$$

3. We therefore want to solve

$$\underset{\mathbf{w},b}{\text{argmax}} \left[\min_i y_i h(\mathbf{x}_i) \right].$$

That doesn't look straightforward...

The support vector machine (SVM)

Equivalently however:

1. Formulate as a *constrained optimization*

$$\underset{\mathbf{w},b}{\operatorname{argmin}} ||\mathbf{w}||^2 \text{ such that } y_i h(\mathbf{x}_i) \geq 1 \text{ for } i = 1, \ldots, m.$$

2. We have a *quadratic optimization with linear constraints* so standard methods apply.
3. It turns out that the solution has the form

$$\mathbf{w}_{\text{opt}} = \sum_{i=1}^{m} y_i \alpha_i \mathbf{\Phi}(\mathbf{x}_i)$$

   where the α_i are *Lagrange multipliers*.
4. So we end up with

$$y = \operatorname{sgn}\left[\sum_{i=1}^{m} y_i \alpha_i \mathbf{\Phi}^T(\mathbf{x}_i)\mathbf{\Phi}(\mathbf{x}) + b\right].$$

The support vector machine (SVM)

It turns our that the *inner product* $\mathbf{\Phi}^T(\mathbf{x}_1)\mathbf{\Phi}(\mathbf{x}_2)$ is fundamental to SVMs:

1. A *kernel* K is a function that *directly computes* the inner product
$$K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{\Phi}^T(\mathbf{x}_1)\mathbf{\Phi}(\mathbf{x}_2).$$
2. A kernel may do this *without explicitly computing the sum* implied.
3. *Mercer's theorem* characterises the K for which *there exists a corresponding function* $\mathbf{\Phi}$.
4. We generally deal with K directly. For example the *radial basis function* kernel.
$$K(\mathbf{x}_1, \mathbf{x}_2) = \exp\left(-\frac{1}{2\sigma^2}||\mathbf{x}_1 - \mathbf{x}_2||^2\right)$$

Various other refinements let us handle, for example, problems that are not *linearly separable*.

Machine learning for interactive proof assistants

Interactive theorem provers such as *Isabelle* and *Mizar* can employ ATPs:

1. Problems can be *translated to first-order* and handed on to an ATP.
2. However this can generate *large, hard problems*.
3. We want to insure that *only relevant premises* are supplied to an ATP.

Reference: Jesse Alama et al., "Premise selection for mathematics by corpus analysis and kernel methods", *Journal of Automated Reasoning*, 2014.

Starting with the *Mizar mathematical library (MML)*, construct the graph of proof dependencies:

$$\text{MML}$$
$$\downarrow$$
$$\text{Set } \Gamma \text{ of FOL formulae}$$
$$\downarrow$$
$$\text{Matrix } \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

where the *dependency matrix* $\mathbf{D}$ is defined as

$$D_{c,a} = \begin{cases} 1 & \text{if axiom } a \text{ is used to prove conjecture } c \\ 0 & \text{otherwise} \end{cases}.$$

Next, represent every formula in Γ using its symbols and subterms.

$$\text{Set } \{t_1, t_2, \ldots, t_n\} \text{ of symbols/subterms}$$

$$\downarrow$$

$$\text{Matrix } \mathbf{S} = \begin{bmatrix} 0 & 0 & 1 & 1 & \cdots & 0 \\ 1 & 1 & 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & 0 & \cdots & 1 \end{bmatrix}$$

where the *subterm matrix* $\mathbf{S}$ is defined as

$$S_{c,i} = \begin{cases} 1 & \text{if symbol/subterm } i \text{ appears in } c \\ 0 & \text{otherwise} \end{cases}.$$

The *features* for a conjecture c are just the corresponding row of $\mathbf{S}$.

The approach works as follows:

1. For every axiom $a \in \Gamma$ train a classifier

$$h_a(c) : \Gamma \to \mathbb{R}$$

   that provides an indication of *how useful a is for proving c*.

2. One way to do this is to construct for each axiom a a model

$$h_a(c) = \Pr(a \text{ is used to prove } c | \text{symbols/subterms in } c).$$

3. Conditional probabilities like this lead us into the domain of *Bayes-optimal classifiers*.

The method is mostly concerned with a form of *kernel classifier*.

However in the interest of a varied presentation we introduce a simple form of *Bayesian classification*.

Naïve Bayes

The *naïve Bayes classifier* is simple:

1. Choose the class maximizing
$$\Pr(C|\mathbf{x}) = \frac{1}{Z}\Pr(\mathbf{x}|C)\Pr(C).$$
2. We *assume* that features are conditionally independent given the class. So
$$h(\mathbf{x}) = \operatorname*{argmax}_C \Pr(C)\prod_{i=1}^{d}\Pr(x_i|C).$$
3. The probabilities are easily *estimated from the training data*.
4. In this application we want to *rank* the possible axioms. This is easy: the closer $h(\mathbf{x})$ is to 1 the more useful we expect it to be.

Point 2 is a *very strong assumption* but the algorithm can often work surprisingly well.

Machine learning for interactive proof assistants