# Learning from examples

## In which we describe agents that can improve their behavior through diligent study of their own experiences.

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

May 5, 2011

# Spis treści

# Introduction to lecture

An agent is learning if it improves its performance on future tasks after making observations about the world.

Learning can range from the trivial, as . . .
to the profound, as . . .

In this lecture we will concentrate on one class of learning problem, which seems restricted but actually has vast applicability: from a collection of input-output pairs, learn a function that predicts the output for new inputs.

# Introduction to lecture

An agent is learning if it improves its performance on future tasks after making observations about the world.

Learning can range from the trivial, as . . .
to the profound, as . . .

In this lecture we will concentrate on one class of learning problem, which seems restricted but actually has vast applicability: from a collection of input-output pairs, learn a function that predicts the output for new inputs.

# Introduction to lecture

Why would we want an agent to learn? If the design of the agent can be improved, why wouldn't the designers just program in that improvement to begin with? There are three main reasons.

- First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters.
- Second, the designers 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.
- Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family mernbers, but even the best programrners are unable to program a computer to accomplish that task, except by using learning algorithms.

# Introduction to lecture

Why would we want an agent to learn? If the design of the agent can be improved, why wouldn't the designers just program in that improvement to begin with? There are three main reasons.

- First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters.
- Second, the designers 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.
- Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms.

# Introduction to lecture

Why would we want an agent to learn? If the design of the agent can be improved, why wouldn't the designers just program in that improvement to begin with? There are three main reasons.

- First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters.
- Second, the designers 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.
- Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family mernbers, but even the best programrners are unable to program a computer to accomplish that task, except by using learning algorithms.

# Forms of learning

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which component is to be improved.
- What prior knowledge the agent already has.
- What representation is used for the data and the component.
- What feedback is available to learn from.

# Forms of learning

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which component is to be improved.
- What prior knowledge the agent already has.
- What representation is used for the data and the component.
- What feedback is available to learn from.

# Forms of learning

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which component is to be improved.
- What prior knowledge the agent already has.
- What representation is used for the data and the component.
- What feedback is available to learn from.

# Forms of learning

Any component of an agent can be improved by learning from data. The improvements, and the techniques used to make them, depend on four major factors:

- Which component is to be improved.
- What prior knowledge the agent already has.
- What representation is used for the data and the component.
- What feedback is available to learn from.

## Components to be learned

The components of agents include:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe classes of states whose achievement maximizes the agent's utility.

### Components to be learned – example

Each of these components can be leamed. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts "Brake!" the agent might learn a condition - action rule for when to brake (component l); the agent also learns every time the instructor does not shout. By seeing many camera images that it is told contain buses, it can learn to recognize them (2). By trying actions and observing the results – for example, breaking hard on a wet road – it can learn the effects of its actions (3). Then, when it receives no tip from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (4).

# Feedback to learn from

There are three types of feedback that determine the three main types of learning:

## Unsupervised learning

In unsupervised learning the agent leams pattems in the input even though no feedback is supplied. The most common unsupervised leaming task is clustering: potentially useful clusters of input examples. For example, a taxi agent might gradually develop a concept of „good traffic days" and „bad traffic days" without ever being given labeled examples of each by a teacher.

# Feedback to learn from

## Reinforcement learning

In reinforcement learning the agent leams from a series of reinforcements-rewards or punishments. For example, the lack of a tip at the end of the joumey gives the taxi agent an indication that it did something wrong. The two points for a win at the end of a chess game tells the agent it did something right. It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it.

## Supervised learning

In supervised learning the agent observes some example input-output pairs and learns a function that maps from input to output. In component I (previous slide), the inputs are percepts and the output are provided by a teacher who says „Brake!" or „Turn left". In component 2, the inputs are camera images and the outputs again come from a teacher who says „that's a bus". In 3, the theory of braking is a function from states and braking actions to stopping distance in metres. In this case the output value is available directly from the agent' s percepts (after the fact); the environment is the teacher.

The task of supervised learning is this:

**The task of supervised learning**

Given a training set of $n$ example input-output pairs

$$(x_1, y_1), (x_2, y_2), \dots (x_n, x_n),$$

where each $y_j$ was generated by an unknown function $y = f(x)$, discover a function $h$ that approximates the true function $f$.

Notice, that $x$ and $y$ can be any value; they need not be numbers. The function $h$ is a hypothesis.

**Learning**

Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set.

To measure the accuracy of a hypothesis we give it a test set of examples that are distinct from the training set. We say a hypothesis generalizes well if it correctly predicts the value of $y$ for novel examples. Sometimes function $f$ is stochastic – it is not strictly a function of $x$, and what we have to learn is a conditional probability distribution, $P(Y|x)$.
When the output $y$ is

- one of a finite set of values (such as sunny, cloudy or rainy) the learning problem is called classification, and is called Boolean or binary classification if there are only two values.
- a number (such as tomorrow's temperatureh) learning problem is called regression.

**Example – hypothesis space**

Fitting a function of a single variable to some points.

**Example**

In some cases, an analyst looking at a problem is willing to make more fine-grained distinctions about the hypothesis space, to say – even before seeing any data – not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis $h^*$ that is most probable given the data:

$$h^* = \underset{h \in H}{argmax} P(h|data),$$

which, by Bayes' rule, is equivalent to

$$h^* = \underset{h \in H}{argmax} P(data|h)P(h).$$

## Tradeoff

Why not let $H$ be the class of all Java programs, or Turing machines? After all, every computable function can be represented by some Turing machine, and that is the best we can do. Problem with this idea is that it does not take into account the computational complexity of learning. There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space. For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is in general undecidable.

Decision tree induction is one of the simplest and yet most successful forms of machine learning. We first describe the representation – the hypothesis space – and then show how to learn a good hypothesis.

# A decision tree representation

## A decision tree

A decision tree represents a function that takes as input a vector of attribute values returns a „decision" – a single output value. The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a positive example) or false (a negative example).

# A decision tree representation

**How it works**

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, $A_i$, and the branches from the node are labeled with the possible values of the attribute, $A_j = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function. The decision tree representation is natural for humans; indeed, many „How To” manuals are written entirely as a single decision tree stretching over hundreds of pages.

## Example

As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate *WillWait*. First we list the attributes that we will consider as part of the input:

- Alternate: whether there is a suitable altemative restaurant nearby.
- Bar: whether the restaurant has a comfortable bar area to wait in.
- Fri or Sat: true on Fridays and Saturdays.
- Hungry: whether we are hungry.
- Patrons: how many people are in the restaurant (values are None, Same, and Fi
- Price: the restaurant's price range ($, $$, $$$).
- Raining: whether it is raining outside.
- Reservation: whether we made a reservation.
- Type: the kind of restaurant (French, Italian, Thai, or burger).
- WaitEstimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, or $> 60$.

**Example**

A decision tree for deciding whether to wait for a table.

# Expressiveness of decision trees

**Example**

A Boolean decision tree is logically equivalent to the assertion that the goal attribute is trur if and only if the input attributes satisfy one of the paths leading to a leaf with value true. Writing this out in propositional logic, we have

$$Goal := (Path_1 \vee Path_2 \vee \dots),$$

where each *Path* is a conjunction of attribute-value tests required to follow that path. Thus, the whole expression is equivalent to disjunctive normal form, which that any function in propositional logic can be expressed as a decision tree.

# Inducing decision trees from examples

**Example**

An example for a Boolean decision tree consists of an $(x, y)$ pair, where $x$ is a vector of values for the input attributes, and $y$ is a single Boolean output value. We have training set of 12 examples

## Examples for the restaurant domain

| Example | Input Attributes | | | | | | | | | | Goal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Alt | Bar | Fri | Hun | Pat | Price | Rain | Res | Type | Est | WillWait |
| x_1 | Yes | No | No | Yes | Same | $$$ | No | Yes | French | 0-10 | Yes |
| x_2 | Yes | No | No | Yes | Full | $ | No | No | Thai | 30-60 | No |
| x_3 | No | Yes | No | No | Same | $ | No | No | Burger | 0-10 | Yes |
| x_4 | Yes | No | Yes | Yes | Full | $ | Yes | No | Thai | 10-30 | Yes |
| x_5 | Yes | No | Yes | No | Full | $$$ | No | Yes | French | >60 | No |
| x_6 | No | Yes | No | Yes | Same | $$ | Yes | Yes | Italian | 0-10 | Yes |
| x_7 | No | Yes | No | No | None | $ | Yes | No | Burger | 0-10 | No |
| x_8 | No | No | No | Yes | Same | $$ | Yes | Yes | Thai | 0-10 | Yes |
| x_9 | No | Yes | Yes | No | Full | $ | Yes | No | Burger | >60 | No |
| x_10 | Yes | Yes | Yes | Yes | Full | $$$ | No | Yes | Italian | 10-30 | No |
| x_11 | No | No | No | No | None | $ | No | No | Thai | 0-10 | No |
| x_12 | Yes | Yes | Yes | Yes | Full | $ | No | No | Burger | 30-60 | Yes |

# Inducing decision trees from examples

## Decision tree learning algorithm

We want a tree that is consistent with the examples and is as small as possible. The DECISION-TREE-LEARNING algorithm adopts a greedy divide-and-conquer strategy: always test the most important attribute first.

## Example

This test divides the problem up into smaller subproblerns that can then be solved recursively. By „most important attribule", we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

**Example – importance of the attributes**

- Type is a poor attribute.
- Patrons is a fairly important attribute.

# Inducing decision trees from examples

In general, after the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one less attribute. There are four cases to consider for these recursive problems:

1. If the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No.
2. If there are same positive and some negative examples, then choose the best attribute to split them.
3. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent.
4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description, but different classifications. This can happen because there is an error or noise in the data; because the domain is nondeterministic; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the plurality classification of the remaining examples.

## Decision tree learning algorithm

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns a tree
{
  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
    A := argmax(a in attributes: IMPORTANCE(a, examples))
    tree := a new decision tree with root test A
    for each value v_{k} of A do
      exs := {e : e in examples and e.A = v_{k}}
      subtree := DECISION-TREE-LEARNING(exs, attributes - A, examples)
      add a branch to tree with label (A = v_{k}) and subtree subtree
    return tree
}
```

## Etropy

We will use the notion of information gain, which is defined in terms of entropy, the fundamental quantity in information theory.
Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy.
A random variable with only one value – a coin that always comes up heads – has no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and this counts as "1 bit" of entropy. Consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin – if we guess heads we'll be wrong only 1% of the time – so we would like it to have an entropy measure that is close to zero, but positive.

## Etropy

We will use the notion of information gain, which is defined in terms of entropy, the fundamental quantity in information theory.
Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy.
A random variable with only one value – a coin that always comes up heads – has no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and this counts as "1 bit" of entropy. Consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin – if we guess heads we'll be wrong only 1% of the time – so we would like it to have an entropy measure that is close to zero, but positive.

## Etropy

We will use the notion of information gain, which is defined in terms of entropy, the fundamental quantity in information theory.

Entropy is a measure of the uncertainty of a random variable; acquisition of information corresponds to a reduction in entropy.

A random variable with only one value – a coin that always comes up heads – has no uncertainty and thus its entropy is defined as zero; thus, we gain no information by observing its value. A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and this counts as "1 bit" of entropy. Consider an unfair coin that comes up heads 99% of the time. Intuitively, this coin has less uncertainty than the fair coin – if we guess heads we'll be wrong only 1% of the time – so we would like it to have an entropy measure that is close to zero, but positive.

**Etropy**

In general, the entropy of a random variable $V$ with values $v_k$, each with probability $P(v_k)$, is defined as

$$Entropy : H(V) = \sum_k P(v_k)\log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k)\log_2 P(v_k).$$

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(Fair) = -(0.5\log_2 0.5 + 0.5\log_2 0.5) = 1.$$

If the coin is loaded to give 99% heads, we get

$$H(Loaded) = -(0.99\log_2 0.99 + 0.01\log_2 0.01) \approx 0.08 bits.$$

**Etropy of Boolean random variable**

Based on above-mentioned, let's define $B(q)$ as the entropy of a Boolean random variable that is true with probability $q$

$$B(q) = -(q log_2 q + (1 - q) log_2 (1 - q))$$

# Choosing attribute tests

If a training set contains $p$ positive examples and $n$ negative examples, then the entropy of the *Goal* attribute on the whole set is

$$H(Goal) = B\left(\frac{p}{p+n}\right)$$

The restaurant training set has $p = n = 6$, so the corresponding entropy is $B(0.5)$ or exactly 1 bit. A test on a single attribute $A$ might give us only part of this 1 bit. We can measure exactly how much by looking at the entropy remaining after the attribute test.

# Choosing attribute tests – entropy remaining after the attribute test

An attribute $A$ with $d$ distinct values divides the training set $E$ into subsets $E_1, \cdots, E_d$. Each subset $E_k$ has $p_k$ positive examples and $n_k$ negative examples, so if we go along that branch, we will need an additional $B(p_k/(p_k + n_k))$ bits of information to answer the question. A randomly chosen example from the training set has the $k$-th value for the attribute with probability $(p_k + n_k)/(p + n)$, so the expected entropy remaining after testing $A$ is

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right)$$

# Choosing attribute tests – information gain

The information gain from the attribute test on $A$ is the expected reduction in entropy

$$Gain(A) = B\left(\frac{p}{p+n}\right) - Remainder(A)$$

In fact $Gain(A)$ is just what we need to implement the *IMPORTANCE* function. For our example, we have

$$Gain(Patrons) = 1 - \left[\frac{2}{12}B\left(\frac{0}{2}\right) + \frac{4}{12}B\left(\frac{4}{4}\right) + \frac{6}{12}B\left(\frac{2}{6}\right)\right] \approx 0.541 \text{ bits},$$

$$Gain(Type) = 1 - \left[\frac{2}{12}B\left(\frac{1}{2}\right) + \frac{2}{12}B\left(\frac{1}{2}\right) + \frac{4}{12}B\left(\frac{2}{4}\right) + \frac{4}{12}B\left(\frac{2}{4}\right)\right] = 0 \text{ bits},$$

confirming our intuition that *Patrons* is a better attribute to split on[1].

[1] In fact, *Patrons* has the maximum gain of any of the attributes and would be chosen by the decision tree learning algorithm as the root.

# Choosing attribute tests calculation for *Patrons* atribute

$$Gain(Patrons) = B\left(\frac{p}{p+n}\right) - Remainder(Patrons)$$

$$B\left(\frac{p}{p+n}\right) = B\left(\frac{6}{12}\right) = B(0.5) = 1$$

Because atribute *Patrons* takes three values: *None*, *Same* and *Full* it divides the training set $E$ into subsets $E_{None}$, $E_{Same}$, $E_{Full}$.

$$Remainder(Patrons) = \sum_{k=None,Same,Full} \frac{p_k + n_k}{p+n} B\left(\frac{p_k}{p_k + n_k}\right)$$

## Choosing attribute tests calculation for *Patrons* atribute

For each subset we have following numper of positie and negative examples

| | *positive* | *negative* |
|---|---|---|
| $E_{None}$ | 0 | 2 |
| $E_{Same}$ | 4 | 0 |
| $E_{Full}$ | 2 | 4 |

so we have

$$Remainder(Patrons) = \sum_{k=None,Same,Full} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right) =$$

# Choosing attribute tests calculation for *Patrons* atribute

| | *positive* | *negative* |
|---|---|---|
| $E_{None}$ | 0 | 2 |
| $E_{Same}$ | 4 | 0 |
| $E_{Full}$ | 2 | 4 |

$$= \frac{p_{None} + n_{None}}{p+n} B\left(\frac{p_{None}}{p_{None} + n_{None}}\right) + \frac{p_{Same} + n_{Same}}{p+n} B\left(\frac{p_{Same}}{p_{Same} + n_{Same}}\right) +$$

$$+ \frac{p_{Full} + n_{Full}}{p+n} B\left(\frac{p_{Full}}{p_{Full} + n_{Full}}\right) =$$

$$= \frac{0+2}{6+6} B\left(\frac{0}{0+2}\right) + \frac{4+0}{6+6} B\left(\frac{4}{4+0}\right) + \frac{2+4}{6+6} B\left(\frac{2}{2+4}\right) =$$

$$= \frac{2}{12} B\left(\frac{0}{2}\right) + \frac{4}{12} B\left(\frac{4}{4}\right) + \frac{6}{12} B\left(\frac{2}{6}\right)$$

what is the same as before.

# Choosing attribute tests – what next?

After all calculation, we have that *Patrons* is the best attribute so we set it as a root node in out decision tree. From this (root) node we have three branches: *None*, *Same* and *Full* (because atribute *Patrons* takes those three values).
For branch *None* all examples returns *No* so we know what to do.
Similarly for *Same* – an answer is *Yes*. Now there is a question: how we can find attribute for e.g. *Full* branch?

# Choosing attribute tests – what next?

An answer is that algorithm is similar to previous but the set with examples should be modified. We should consider only patterns for which attribute *Patrons* takes value *Full*, i.e.

| Example | Input Attributes | | | | | | | | | Goal |
|---|---|---|---|---|---|---|---|---|---|---|
| | Alt | Bar | Fri | Hun | Price | Rain | Res | Type | Est | WillWait |
| x_2 | Yes | No | No | Yes | $ | No | No | Thai | 30-60 | No |
| x_4 | Yes | No | Yes | Yes | $ | Yes | No | Thai | 10-30 | Yes |
| x_5 | Yes | No | Yes | No | $$$ | No | Yes | French | >60 | No |
| x_9 | No | Yes | Yes | No | $ | Yes | No | Burger | >60 | No |
| x_10 | Yes | Yes | Yes | Yes | $$$ | No | Yes | Italian | 10-30 | No |
| x_12 | Yes | Yes | Yes | Yes | $ | No | No | Burger | 30-60 | Yes |