Lecture 1

Cezar Ionescu

Administrative matters

- course materials on GitHub
  - lectures notes will generally be available after the lecture, though drafts may appear before
  - exercises will also be posted on GitHub
- register for CATS points
  - assessment: homework to be handed in a the next meeting (or sent to me by email, or to PPWeekly)
  - every piece of homework is pass or fail
- course takes place Saturdays 10:00-12:30 at Ewert House
  - exceptions: **no class** on the **4th of May** and **1st of June (TBC)**!
- main text: /Machine Learning/, Tom Mitchell, 1997
  - there are a couple of copies available in the ContEd library
  - you can buy it used for under 15 GBP on Amazon
  - but you should be able to complete the course using only the lecture notes

Introduction

- types of learning:
  - by rote
  - conditioning
  - from experience
  - any others?
- reasons for *machine learning*:
  - programming is hard, it would be better if computers learnt by themselves
  - to study human learning (and intelligence)
    - perhaps we could then improve our own abilities to learn and to teach
- two main approaches:
  - modelling how we think and learn, without caring about the underlying physiological mechanisms
  - modelling the underlying physiological mechanism, without caring how they lead to thinking and learning

- **Definition** (Mitchell, p. 2) A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.
  - how can we understand this mathematically?
    - for example, by representing the various elements using sets and functions
    - the most difficult to represent in this way appears to be *the task*
- **Note**
  - Sets and functions are the basic building blocks of mathematics. We assume them known.
  - The notation `f : In → Out` represents a function `f` taking inputs from the set `In` and "returning" values from the set `Out`. The notation `f(x)` refers to **the** element of `Out` that `f` returns when given the input `x` (which, therefore, must be an element of `In`, i.e., `x ∈ In`).
  - In machine learning, we frequently encounter function-like "black boxes", which, however, are not functions. The typical example is the Python "function" `Random.choice(seq)`, which takes as input a sequence (e.g., a list) and returns a randomly selected element of this sequence. Obviously, if the argument has more than one element, `Random.choice(seq)` will not be well-defined. We shall follow the generally accepted convention of using the function notation also for such black boxes. When we want to emphasise that we are not dealing with a proper function, we shall use a squiggly ⇝ arrow instead of the standard straight one →
- A preliminary mathematical representation:
  - the set of tasks: `Task`
  - the set of experiences: `Experience`
  - measure of performance: `perf : Task → ℝ`
  - machine learning system: `learn : (Task, Experience) → Task`
    - learning means improving with experience (according to 'perf):

```
for all t ∈ Task, for all e ∈ Experience, perf(t) ≤ perf(learn(t, e))
```

- *Example*: checkers learning
  - the task is to play a game: `Task = Board → Move`
    - a function in `Task` tells you how to move on the board (we assume that the board contains information about whose turn it is). We can call such a function a *strategy*
  - experience: `experience : Task → List Game`
    - we assume we have a function that takes two strategies, plays them against each other, and returns the resulting game:`play : (Play, Play) ⇝ Game`
      - note the squiggly arrow in the type of `play`!
    - the list of games is created by giving the `play` function the same argument *twice*

```
experience(t) = [play(t, t), play(t, t), ..., play(t, t)]
```

- measure of performance: `perf : (Task, List Task) → ℝ`
- we need a function `score : Game → {0, 1}`

```
perf (learner, [adv₁, ..., advₙ]) =
  sum [score(play(learner, adv₁)), ..., score(play(learner, advₙ))] / n
```

- *Example*: self-driving car
  - the task is to give steering commands based on sensor input: `Task = Sensor → Command`
  - the set of experiences: `Experience = List (Sensor, Command)`
  - performance: `perf : (Task, Itinerary) → Time`
    - `perf (learner, itinerary)` measures how long the learner drives along the given itinerary before making a mistake
- **Homework**
  - Give a similar interpretation for the handwriting recognition problem (Mitchell, page 3):
    - Task T: recognizing and classifying handwritten words within images
    - Performance measure P: percent of words correctly classified
    - Training experience E: a database of handwritten words with given classifications
  - This involves filling in
    - `Task =`
    - `Experience =`
    - `perf :`
      - something about how `perf` is computed

Concept learning

- idea: acquiring general concepts from examples
  - e.g., learn to recognise cats from images of animals
- what is a concept?
  - *nominalistic* view: the set of instances of the concept
- mathematically, we can identify a concept with a subset
  - e.g., `X` is the set of all images of animals, ‘`C ⊆ X` is the subset of images of cats
- subsets are in one-to-one correspondence with boolean-valued functions
  - `C ⊆ X` can be replaced by `c : X → Bool` such that

```
∀ x ∈ X  c(x) = 1  iff x ∈ C
```

- Mitchell uses the functional view and defines:

  - **Concept learning:** inferring a boolean-valued function from training examples of input and output

- **Exercise:** Give an interpretation of concept learning as a learning task (i.e., identify the task, the experience, and the performance measure).

- **Notation**:

  - the training data `D = {((x₁, c(x₁)) ..., (xₙ, c(xₙ))}`
  - the subset of negative training examples `D₀ = {(x, 0) | (x, 0) ∈ D}`

- the subset of positive training examples `D₁ = {(x, 1) | (x, 1) ∈ D}`

- **Example:** enjoyable day
  - we want to learn the concept `enjoyable : Day → {0, 1}`
  - days are described via *attributes*:
    * `Day = (Sky, Temp, Humidity, Wind, Water, Forecast)`
      - `Sky = {Sunny, Cloudy, Rainy}`
      - `Temp = {Warm, Cold}`
      - `Humidity = {Normal, High}`
      - `Wind = {Strong, Weak}`
      - `Water = {Warm, Cool}`
      - `Forecast = {Same, Change}`
  - training data:

Nr	Sky	Temp	Humidity	Wind	Water	Forecast	Enjoyable
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

- **Two major related problems**
  - `Day` contains 96 elements; there are 2^{96} concepts.
  - We can represent a concept by a lookup table. In general, however, we are not going to be able to represent all concepts (the space could be infinite).
  - The training data does not suffice to determine the concept we are looking for. It only fixes the values of the concept for 4 elements of its domain. Thus, there are 2^{92} remaining possibilities.
  - The two problems force us to make two decisions:
    1. Choose a representation for **some** of the concepts. We are going to have to assume that the "real" concept can be represented that way. A representable concepts is called **hypothesis**.
    2. In general, we will still have many hypothesis consistent with the training data. The second decision is how to pick one of them.
  - The assumptions under which we manage to learn the correct concept form **the inductive bias**.
- Hypothesis space for the weather example:
  - each hypothesis is described by a tuple of `(Sky*, Temp*, Humidity*, Wind*, Water*, Foreca` where `S* = S ∪ {?, ∅}`
  - notation: `s ~ s* iff s = s* or s* = ?` (s matches s*)

Let `h` be described by `(s*, t*, u*, wi*, wa*, f*)`. Then

```
h (s, t, u, wi, wa, f) = s ~ s* and t ~ t* and u ~ u* and
                         wi ~ wi* and wa ~ wa* and f ~ f*
```

`Find-S`

- `Find-S` solves problem 2 by choosing the *most specific* hypothesis that is consistent with the training data. Obviously, that implies that there exists an ordering from specific to general.
- Our hypothesis correspond to subsets. We have a natural ordering on subsets: ⊆.

```
Find-S algorithm:

-- input: training data {((x₁, c(x₁)) ..., (xₙ, c(xₙ))}
--        hypothesis set H
h = min H -- set the current hypothesis h to "the" (or "a") smallest element of H
for i in 1:n
  if c(xᵢ) = 0
    then keep h
    else if xᵢ ∈ h then keep h
                   else h = min {h' ∈ H | h ⊆ h' and xᵢ ∈ h'}
-- output: "the" (or "a" most) specific hypothesis in H consistent with the training data
```

Remarks:

- If `H` contains all possible concepts, then the result of `Find-S` is D_1.
- The result of `Find-S` can depend on arbitrary choices if the minimisation problems do not have a unique solution. Hence, the result might not be contained in `c`! A simple example:
  - `X = {a, b, c, d}, H = {∅, {a, b}, {a, c}}, D₁ = {a}`
- The choice of `H` can avoid these problems.

Fundamental property of `Find-S`

If `Find-S` works, then

```
s = Find-S (D₀, D₁, H)  implies
    D₀ ⊆ ¬s, D₁ ⊆ s, and
    for all h ∈ H, D₀ ⊆ ¬h and D₁ ⊆ h ⇒ s ⊆ h
```

Exercise

What does `Find-S (D₁, D₀, H)` do?

- Improving `Find-S`: in order to solve the two problems, we can modify the algorithm to find *all* maximally specific hypotheses consisten with the data.

```
-- input: training data {((x₁, c(x₁)) ..., (xₙ, c(xₙ))}
--        hypothesis set H
S = allMin H -- start with all smallest element of H
repeat until S no longer changes:
  for i in 1:n
    if c(xᵢ) = 0
```

```
    then eliminate from S all {s | xᵢ ∈ s}
    else for all s ∈ S
        if xᵢ ∈ s
        then keep s
        else replace s with allMin {h' ∈ H | h ⊆ h' and xᵢ ∈ h'}
-- output: all most specific hyp consistent with D
```

- **Remarks**
  - why do we need to repeat the `for` loop?
  - the algorithm terminates (why?)
- Avoiding `repeat`
  - we need to keep a record of the negative examples
  - idea: do that in the same form as the record we keep for positive examples!
  - this leads to the `Candidate-Elimination` algorithm