Lecture 1

Cezar Ionescu

11/05/2019

DEPARTMENT FOR CONTINUING EDUCATION

UNIVERSITY OF OXFORD

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

Administrative matters

- 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

Administrative matters

- course takes place Saturdays 10:00-12:30 at Ewert House
  - exceptions: **no class** on the **4th of May** and **1st of June**!

Administrative matters

- 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

Types of learning

- types of learning:
    - by rote
    - conditioning
    - from experience
    - any others?

Why “machine learning”?

- 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

Approaches to ML

- 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

- **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.

Sets and functions

Notation:

- `f : In → Out`
- `f(x)`
- `x ∈ In, f(x) ∈ Out`

Black boxes versus functions

- is `Random.choice(seq)` a function?
- we use $\rightsquigarrow$ instead of $\rightarrow$ when we need to distinguish black boxes from “real” functions

Mathematical representation

- `Task = In → Out`
- `Experience = List E`
- `perf : (Task, List Out) → ℝ`
- `learn : Experience → Task`
- for all `ex₁, ex₂, ins`, we have

```
perf (learn ex₁) ins <= perf (learn (es₁ ++ es₂)) ins
```

Example: checkers learning

- Task: `Play = Board → Move`
- Experience: `Experience = Play → List Game`
    - `play : (Play, Play) → Game`
    - the list of games is created by giving the `play` function the same argument *twice*
- Measure of performance: `perf : (Play, List Play) → ℝ`

```
perf (learner, [adv₁, ..., advₙ]) =
  prc [score(play(learner, adv₁)), ..., score(play(learner, advₙ)
```

Example: self-driving car

- Task: `Drive = Sensor → Steer`
- Experiences: `Experience = List (Sensor, Steer)`
- Measure of performance: `perf : (Drive, Itinerary) → Time`
    - `perf (learner, itinerary)` 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 =`
  - `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

Mathematical representation of concepts

- 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
```

Definition

- 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

- 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 problems

- `Day` contains 96 elements; there are 2^{96} concepts.
- We can represent a concept by a lookup table, but not if it's too big!
- The training data does not suffice to determine the concept we are looking for.

Decisions

- The two problems force us to make two decisions:
  1. How to represent **some** of the concepts (the hypotheses space)
  2. Which hypothesis to pick.

Inductive bias

- 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*, Forecast*)`, 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.
- Our hypothesis correspond to subsets. We have a natural ordering on subsets: $\subseteq$.

Find-S algorithm

```
-- input: training data {((x₁, c(x₁)) ..., (xₙ, c(xₙ))}
--        hypothesis set H
h = min H -- "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
--         consistent with D
```

Example

Work out how `Find-S` works on the weather example.

Remarks

- If H contains all possible concepts, then the result of `Find-S` is D_1.
- A bad situation for `Find-S`:
  - `X = {a, b, c, d}, H = {∅, {a, b}, {a, c}}, D₁ = {a}`
- Another bad situation:
  - `X = {a, b, c, d}, H = {∅, {a, b, c}, {a, b, d}}, D₁ = {a}, D₀ =`
- The choice of H can avoid these problems, as in the weather example.

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?

A better Find-S

```
-- 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