Learning as a Logic formulation

Viktor Schlegel

Inductive Reasoning

"Examples", "Training set"

Don't confuse with FOL notion of functions!

Hypothesis

Given a set of **observations**, find a general rule that describes them.

We assume that the observations are generated by some unknown function f.

Thus, the goal is to find a function h that is **consistent** with the observations and approximates the true f.

Applicable to new examples generated by the same function

Hypothesis

We assume h belongs to some **hypothesis space** $\mathcal{H}$, $h \in \mathcal{H}$. $\mathcal{H}$ describes some class of functions (e.g. all linear functions, all prolog programs, etc)

- h is **consistent** with a set of observations, if it applies to them all
- The problem is **realisable**, if $f \in \mathcal{H}$.

Example: Walk in the park

Attributes

Hypothesis $Walk(x) \Leftrightarrow \neg Lazy(x) \wedge \neg Temperature(x, Low) \wedge Sunny(x)$

Consistent

Class

Sunny?	**Lazy?**	**Rainy?**	**Windy?**	**Temperature?**	**Walk?**
Yes	No	Yes	No	Med	Yes
No	No	No	Yes	Med	No
Yes	Yes	No	No	High	No
Yes	No	Yes	Yes	Low	No

Example: Walk in the park

True function $Walk(x) \Leftrightarrow \neg Lazy(x) \wedge Temperature(x, High)$
$\vee \neg Lazy(x) \wedge Temperature(x, Med) \wedge \neg Windy(x)$
$\vee \neg Lazy(x) \wedge Temperature(x, Low) \wedge Sunny(x) \wedge \neg Windy(x)$

Attributes

Hypothesis $Walk(x) \Leftrightarrow \neg Lazy(x) \wedge \neg Temperature(x, Low) \wedge Sunny(x)$

Consistent

Class

Sunny?	**Lazy?**	**Rainy?**	**Windy?**	**Temperature?**	**Walk?**
Yes	No	Yes	No	Med	Yes
No	No	No	Yes	Med	No
Yes	Yes	No	No	High	No
Yes	No	Yes	Yes	Low	No

“False Negative”: Hypothesis says No, but is actually Yes

No	No	No	No	High	Yes
Yes	No	Yes	Yes	Med	No

“False Positive”: Hypothesis says Yes, but is actually No

Classification

As opposed to regression, where the goal is to assign a continuous numeric value

We will focus on **classification**, i.e. assigning an observation to one of a set of defined classes based on its **attribute values**.

Descriptions, Discrete

We call a classification problem **binary classification** if there are only two classes

Yes/No,
True/False,
Cat/Dog,
...

A **false negative** is an observation classified as negative under the hypothesis but in fact positive.

A **false positive** is an observation classified as positive under the hypothesis but in fact negative.

Logical Formulation of Learning

We will focus on problems and hypotheses that can be represented in **logical form.**

In fact, mostly DNF and Horn clauses

- The hypothesis space is represented as a disjunction of all possible hypotheses

$$\mathcal{H} = h_1 \vee h_2 \vee \ldots \vee h_n$$

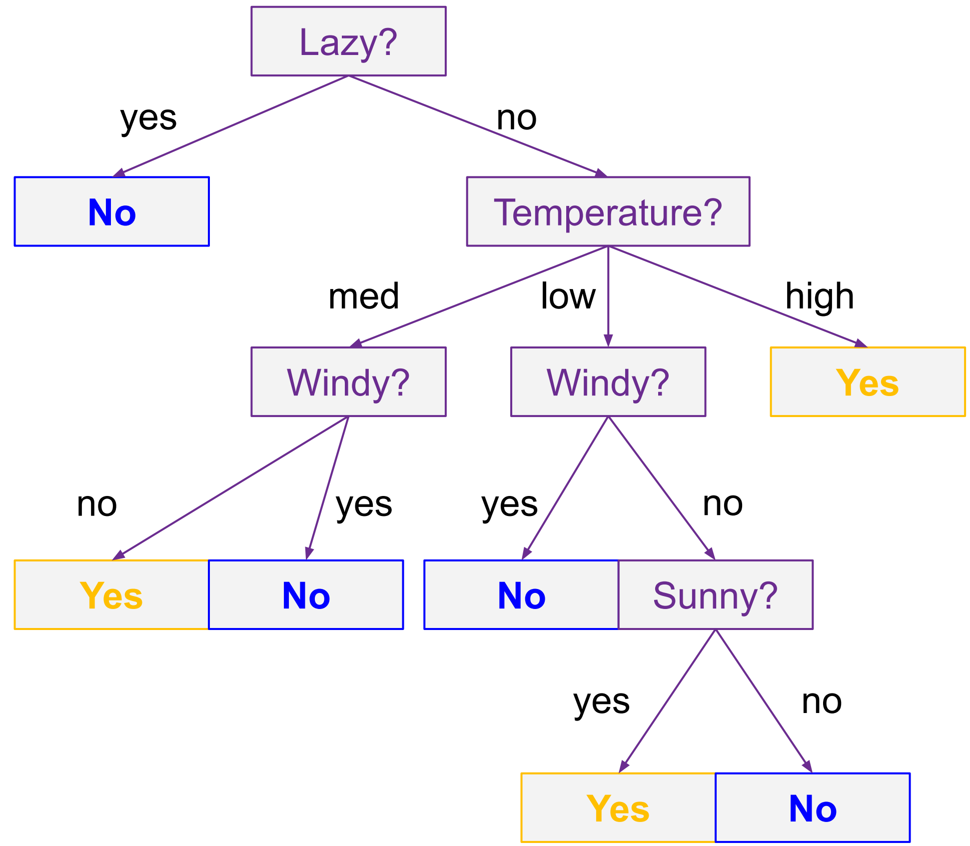
- Goal of learning: rule out hypotheses $h_{j,}$ $j \in 1..\ n$ are **inconsistent** with the observations

$$Hypothesis \wedge \underbrace{\boxed{Descriptions \models Classifications}}_{\text{Observations}}$$

The University of Manchester

Example: decision trees

$$
\begin{aligned}
Walk(x) \Leftrightarrow & \neg Lazy(x) \wedge Temperature(x, High) \\
& \vee \neg Lazy(x) \wedge Temperature(x, Med) \wedge \neg Windy(x) \\
& \vee \neg Lazy(x) \wedge Temperature(x, Low) \wedge Sunny(x) \wedge \neg Windy(x)
\end{aligned}
$$



The "walk in the park" example can be represented as a **decision tree**.

A binary decision tree has an **equivalent** logic representation.

$$Classification \Leftrightarrow \bigvee_{p \in paths} \bigwedge_{step \in p} step$$

The goal is to obtain a concise, interpretable decision tree consistent with the provided examples.

Exhaustive search for the “best” decision tree has prohibitive cost, the size of the search space is $\mathcal{O}(2^{2^n})$

Instead: Greedy, recursive algorithm to approximate the “best” solution

Some more observations...

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	True	True	False	Low	False
False	False	False	True	High	True
True	True	False	True	Med	False
False	False	False	False	Med	True
True	True	True	True	High	False
False	True	False	True	High	False
True	False	True	False	Low	True
True	False	True	True	Low	False
True	False	False	True	Low	False
True	False	False	False	Low	True
True	False	True	False	Med	True
False	False	False	False	Low	False
False	True	True	False	Med	False
True	True	True	True	Low	False
True	False	False	True	Med	False

The algorithm

Lazy?

yes

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	True	True	False	Low	False
True	True	False	True	Med	False
True	True	True	True	High	False
False	True	False	True	High	False
False	True	True	False	Med	False
True	True	True	True	Low	False

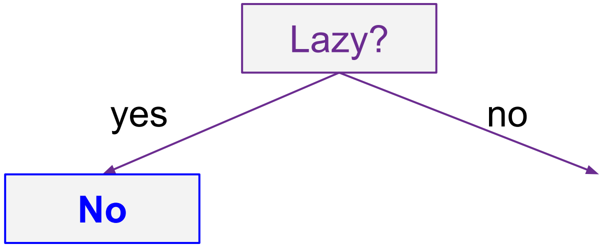
no

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	False	False	True	High	True
False	False	False	False	Med	True
True	False	True	False	Low	True
True	False	True	True	Low	False
True	False	False	True	Low	False
True	False	False	False	Low	True
True	False	True	False	Med	True
False	False	False	False	Low	False
True	False	False	True	Med	False

```
def decision_tree_learning(examples, attributes, parent_examples):
  if examples is empty: return plurality_value(parent_examples)
  elif all examples have same classification: return classification
  elif attributes is empty: return plurality_value(examples)
  else:
    attr := get_most_important_attribute(attributes, examples)
    tree := new decision tree with root attr
    new_attrs = attributes - {attr}
    for each possible value v of attr:
      new_ex := {e for e in examples and e[attr] = v}
      subtree := decision_tree_learning(new_ex, new_attrs, examples)
      add branch to tree with label v and subtree subtree
    return tree
```

The algorithm

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	True	True	False	Low	False
True	True	False	True	Med	False
True	True	True	True	High	False
False	True	False	True	High	False
False	True	True	False	Med	False
True	True	True	True	Low	False




```
def decision_tree_learning(examples, attributes, parent_examples):
  if examples is empty: return plurality_value(parent_examples)
  elif all examples have same classification: return classification
  elif attributes is empty: return plurality_value(examples)
  else:
    attr := get_most_important_attribute(attributes, examples)
    tree := new decision tree with root attr
    new_attrs = attributes - {attr}
    for each possible value v of attr:
       new_ex := {e for e in examples and e[attr] = v}
       subtree := decision_tree_learning(new_ex, new_attrs, examples)
       add branch to tree with label v and subtree subtree
    return tree
```

The algorithm

```
def decision_tree_learning(examples, attributes, parent_examples):
  if examples is empty: return plurality_value(parent_examples)
  elif all examples have same classification: return classification
  elif attributes is empty: return plurality_value(examples)
  else:
    attr := get_most_important_attribute(attributes, examples)
    tree := new decision tree with root attr
    new_attrs = attributes - {attr}
    for each possible value v of attr:
       new_ex := {e for e in examples and e[attr] = v}
       subtree := decision_tree_learning(new_ex, new_attrs, examples)
       add branch to tree with label v and subtree subtree
    return tree
```

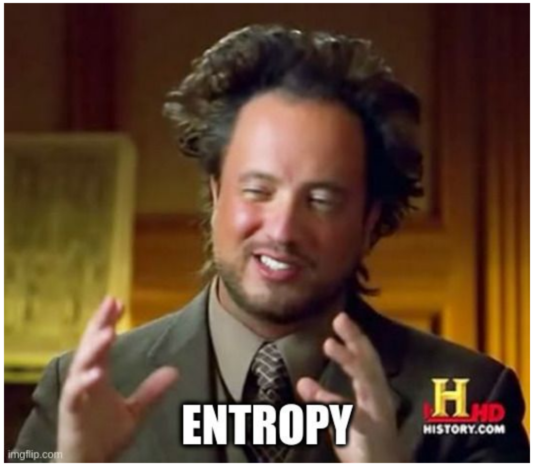
Attribute importance

```
attr := get_most_important_attribute(attributes, examples)
```

Recall: objective is to learn an “optimal” tree.

We want to test attributes that contribute most to the classification as early as possible.

How do we decide that?



Entropy describes the **uncertainty** of a random variable.

With every attribute test we want to **reduce** the entropy of the classification as much as possible.

MANCHESTER 1824
The University of Manchester

Information gain

That is, we want to answer the question "*assuming we knew the value of the attribute, how much less uncertain is the classification now?*"

$$IG(Examples, Attr) = H(Examples) - H(Examples|Attr)$$

$$H(Examples|Attr) = \sum_{v \in vals(Attr)} Pr(v) \cdot H(\{e|e \in Examples \wedge e[Attr] = v\})$$

Entropy

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	True	True	False	Low	False
False	False	False	True	High	True
True	True	False	True	Med	False
False	False	False	False	Med	True
True	True	True	True	High	False
False	True	False	True	High	False
True	False	True	False	Low	True
True	False	True	True	Low	False
True	False	False	True	Low	False
True	False	False	False	Low	True
True	False	True	False	Med	True
False	False	False	False	Low	False
False	True	True	False	Med	False
True	True	True	True	Low	False
True	False	False	True	Med	False

$$H(V) = -\sum_{k \in vals(V)} Pr(k) log_2(Pr(k))$$

$$H(Coin) = -(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{2} log_2 \frac{1}{2}) = 1$$

$$H(DoubleHeadsCoin) = -(1 log_2 1 + 0 log_2 0) = 0$$

$$H(Walk) = -(\frac{1}{3} log_2 \frac{1}{3} + \frac{2}{3} log_2 \frac{2}{3}) = 0.918$$

Entropy

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	True	True	False	Low	False
False	False	False	True	High	True
True	True	False	True	Med	False
False	False	False	False	Med	True
True	True	True	True	High	False
False	True	False	True	High	False
True	False	True	False	Low	True
True	False	True	True	Low	False
True	False	False	True	Low	False
True	False	False	False	Low	True
True	False	True	False	Med	True
False	False	False	False	Low	False
False	True	True	False	Med	False
True	True	True	True	Low	False
True	False	False	True	Med	False

$$H(Walk) = -(\frac{1}{3}log_2\frac{1}{3} + \frac{2}{3}log_2\frac{2}{3}) = 0.918$$

$$H(Examples|Attr) = \sum_{v \in vals(Attr)} Pr(v) \cdot H(\{e|e \in Examples \wedge e[Attr] = v\})$$

$$H(Walk|Lazy) = \frac{2}{5}H(Walk|Lazy = Yes) + \frac{3}{5}H(Walk|Lazy = No) \quad = 0.59$$

$$-(1log_2 1 + 0log_2 0) = 0 \qquad -(\tfrac{5}{9}log_2\tfrac{5}{9} + \tfrac{4}{9}log_2\tfrac{4}{9}) = 0.99$$

$$IG(Walk, Lazy) = 0.324$$

$$IG(Walk, Temperature) = 0.008 \qquad IG(Walk, Rainy) = 0.006$$

$$IG(Walk, Sunny) = 0 \qquad IG(Walk, Windy) = 0.16$$

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	False	False	True	High	True
False	False	False	False	Med	True
True	False	True	False	Low	True
True	False	True	True	Low	False
True	False	False	True	Low	False
True	False	False	False	Low	True
True	False	True	False	Med	True
False	False	False	False	Low	False
True	False	False	True	Med	False

Sunny	Lazy	Rainy	Windy	Temperature	Walk
False	True	True	False	Low	False
True	True	False	True	Med	False
True	True	True	True	High	False
False	True	False	True	High	False
False	True	True	False	Med	False
True	True	True	True	Low	False

The algorithm

- Greedy: First attribute to split on might be not the best one, but we cannot "undo" the decision
- Extensions:
  - Deal with regression problems by applying linear regression deeper in the tree
  - Continuous inputs: Learn to split continuous attributes into buckets
  - Pruning: Remove branches that only give insignificant information gain

How do we know we found the best hypothesis?

- To investigate how well we learned the function, we typically measure the classification performance of the hypothesis on a held-out “evaluation set”
- We won’t focus too much evaluation here, but it is a vital and necessary step to assess the learned hypothesis where the true function is unknown

Learning without knowledge

Viktor Schlegel

Hypothesis Space

Recall:

$$\mathcal{H} = h_1 \vee h_2 \vee \ldots \vee h_n$$

In this video:

- Investigate the structure of the hypothesis space
- Exploit this structure when learning

Extension

Hypothesis

$$Walk(x) \Leftrightarrow \neg Lazy(x) \wedge Temperature(x, High)$$

We call the set of all **possible** observations that are consistent with a hypothesis its **extension**. $\mathcal{E}(h)$

Hypotheses that have equivalent extensions, are **logically equivalent**.

Extension

Sunny	Lazy	Rainy	Windy	Temperature	Walk
True	False	True	False	High	True
True	False	False	False	High	True
True	False	True	True	High	True
False	False	True	False	High	True
False	False	True	True	High	True
False	False	False	True	High	True
False	False	False	False	High	True
True	False	False	True	High	True

Sunny	Lazy	Rainy	Windy	Temperature	Walk
True	True	True	True	Low	False
False	True	False	True	Low	False
True	True	True	False	Low	False
True	True	False	True	High	False
True	False	True	True	Low	False
True	True	True	False	Med	False
True	True	False	True	Med	False
True	False	False	False	Low	False
True	False	True	True	Med	False
False	True	True	True	Med	False
False	True	False	True	Med	False
False	True	True	False	High	False
False	False	False	True	Med	False
False	True	False	False	High	False
False	True	True	False	Low	False
False	True	False	True	High	False
True	True	False	False	Low	False
False	True	True	False	Med	False
True	False	False	True	Low	False
False	True	True	True	Low	False
False	False	True	True	Low	False
True	False	True	False	Med	False
True	True	False	False	High	False
True	True	True	True	High	False
True	False	True	False	Low	False
False	True	False	False	Low	False
True	True	False	False	Med	False
True	False	False	False	Med	False
True	False	False	True	Med	False
False	True	True	True	High	False
True	True	False	True	Low	False
False	False	True	True	Med	False
False	False	True	False	Low	False
False	True	False	False	Med	False
True	True	True	True	Med	False
False	False	False	False	Med	False
True	True	True	False	High	False
False	False	False	False	Low	False
False	False	False	True	Low	False
False	False	True	False	Med	False

There must be order!

We can compare hypotheses by their extensions.

If the extension of h_n is a subset of the extension of h_k, h_k is a **generalisation** of h_n. $\mathcal{E}(h_k) \supset \mathcal{E}(h_n)$

If the extension of h_k is a superset of the extension of h_n,h_n is a **specialisation** of h_k. $\mathcal{E}(h_n) \subset \mathcal{E}(h_k)$

$h_k : Walk(x) \Leftrightarrow \neg Lazy(x) \wedge Temperature(x, High)$

$\mathcal{E}(h_k)$

Sunny	Lazy	Rainy	Windy	Temperature	Walk
True	False	True	False	High	True
True	False	False	False	High	True
True	False	True	True	High	True
False	False	True	False	High	True
False	False	True	True	High	True
False	False	False	True	High	True
False	False	False	False	High	True
True	False	False	True	High	True

$h_n : Walk(x) \Leftrightarrow \neg Lazy(x) \wedge Temperature(x, High) \wedge Rainy(x)$

$\mathcal{E}(h_n)$

Sunny	Lazy	Rainy	Windy	Temperature	Walk
True	False	True	False	High	True
True	False	True	True	High	True
False	False	True	False	High	True
False	False	True	True	High	True

$\mathcal{E}(h_n) \subset \mathcal{E}(h_k)$

h_n is a **specialisation** of h_k

A simple learning algorithm

Sketch:

- pick a hypothesis, test it against all examples
- As long as the hypothesis is consistent, do nothing
- If an example is a false positive: specialise the hypothesis
  → Hypothesis says yes, but actually no
- If an example is a false negative: generalise the hypothesis
  → Hypothesis says no, but actually yes

How to generalise and specialise?

Generalisation and specialisation describe a **logical relationship** between hypotheses. Let

$$h_n : \forall x C_n(x)$$
$$h_k : \forall x C_k(x)$$

Iff: h_n is specialisation of h_k :

Iff: h_k is generalisation of h_n:

$$\forall x C_n(x) \Rightarrow C_k(x)$$

$$Walk(x) \Leftrightarrow \neg Lazy(x) \wedge \neg Temperature(x, Low) \wedge Sunny(x) \wedge Rainy(x)$$

Sunny?	Lazy?	Rainy?	Windy?	Temperature?	Walk?
Yes	No	Yes	Yes	Med	No

Need to specialise!

$$Walk(x) \Leftrightarrow \neg Lazy(x) \wedge \neg Temperature(x, Low) \wedge Sunny(x)$$

No	No	No	No	High	Yes

Need to generalise!

$$Walk(x) \Leftrightarrow \neg Lazy(x) \wedge \neg Temperature(x, Low)$$

Current-best-learning

```
def cbl(all_examples, h, seen_examples):
  if all_examples is empty: return hypothesis
  example := all_examples.pop()
  seen_examples.push(example)
  if h |= e:
    return cbl(all_examples, hypothesis, seen_examples)
  elif e is false positive for h:
    for h' in specialisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  elif e is false negative for h:
    for h' in generalisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  return None
```

- As long as the hypothesis is consistent, do nothing
- When no more examples, done

Current-best-learning

```
def cbl(all_examples, h, seen_examples):
  if all_examples is empty: return hypothesis
  example := all_examples.pop()
  seen_examples.push(example)
  if h |= e:
    return cbl(all_examples, hypothesis, seen_examples)
  elif e is false positive for h:
    for h' in specialisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  elif e is false negative for h:
    for h' in generalisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  return None
```

- If an example is a **false positive**: specialise the hypothesis
- But: consistent with all previous examples

Current-best-learning

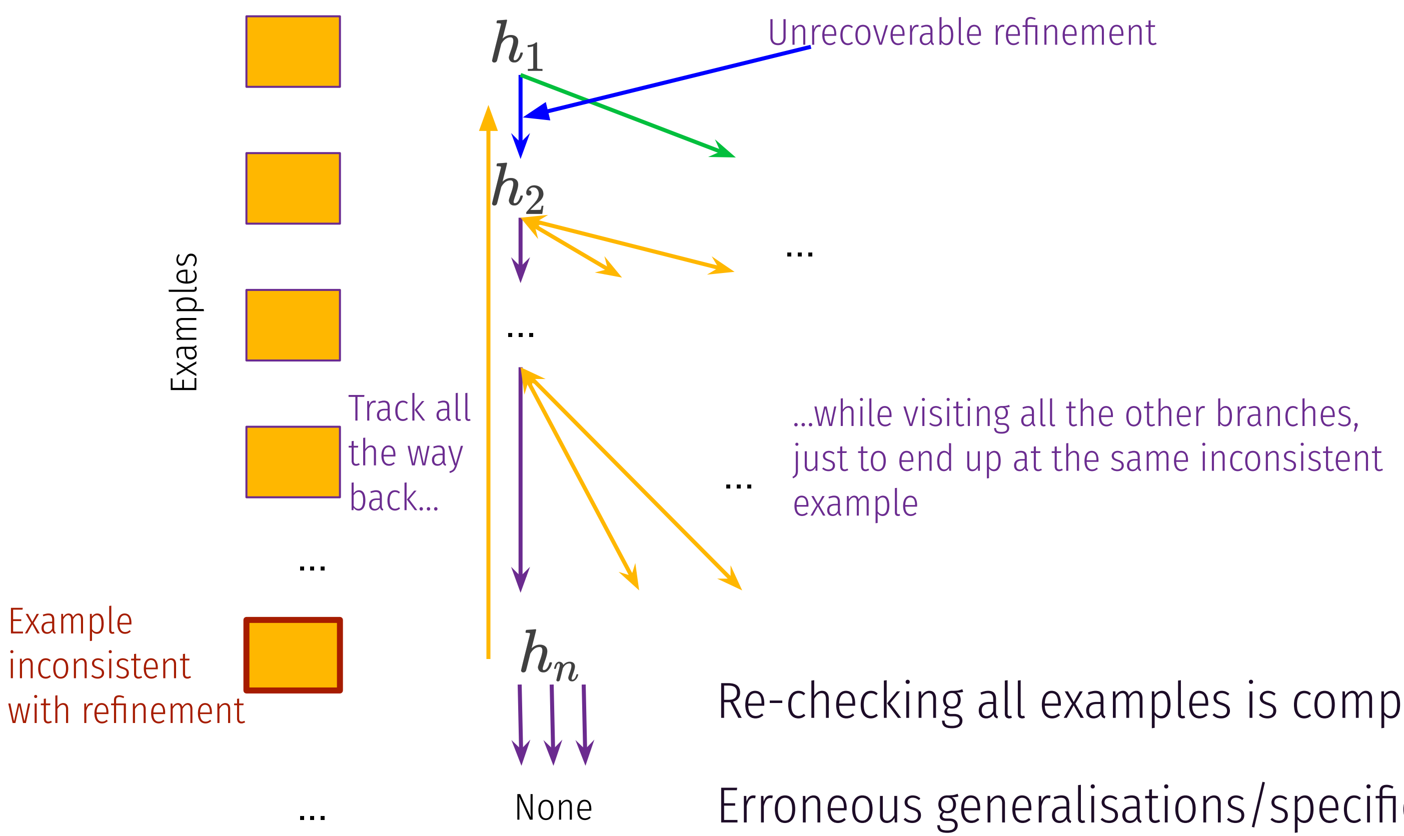
```
def cbl(all_examples, h, seen_examples):
  if all_examples is empty: return hypothesis
  example := all_examples.pop()
  seen_examples.push(example)
  if h |= e:
    return cbl(all_examples, hypothesis, seen_examples)
  elif e is false positive for h:
    for h' in specialisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  elif e is false negative for h:
    for h' in generalisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  return None
```

- If an example is a **false negative**: generalise the hypothesis
- But: consistent with all previous examples

Current-best-learning

```
def cbl(all_examples, h, seen_examples):
  if all_examples is empty: return hypothesis
  example := all_examples.pop()
  seen_examples.push(example)
  if h |= e:
    return cbl(all_examples, hypothesis, seen_examples)
  elif e is false positive for h:
    for h' in specialisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  elif e is false negative for h:
    for h' in generalisations of h if h' |= seen_examples:
        h" := cbl(all_examples, h', seen_examples)
        if h": return h"
  return None
```

- If no consistent specifications or generalisations:
  return None
- Recursion: check if recursive call returns consistent hypothesis



Re-checking all examples is computationally expensive!

Erroneous generalisations/specifications can potentially be detected way later (which results in back-tracking)!

Avoiding Backtracking

In the previous example we had to backtrack a lot, because we committed to a hypothesis, which (much later) turned out to be inconsistent with the examples.

We cannot decide in advance, whether the refinement we choose will lead to an unrecoverable situation.

Why commit to a single hypothesis?

Least commitment search

Instead of keeping track of a single hypothesis, keep track of **all hypotheses** consistent with examples seen so far (**version space**).

As new examples arrive, we refine the version space.

How do we represent the version space?

Version space boundaries

Analogy: How do we represent all (i.e. an infinite amount of) real numbers between 3 and 5? With the interval [3,5]

We represent the version space by its boundaries.

G-set: set of most general consistent hypotheses.

S-set: set of most specific consistent hypotheses.

… with all observations so far

Every hypothesis in between is consistent, every hypothesis outside isn't!

More general than some member of S-set but more specific than some member of G-set.

For all h not in G-set or S-set: h consistent $\Leftrightarrow$ h more specific than some g_k and more general than some s_k

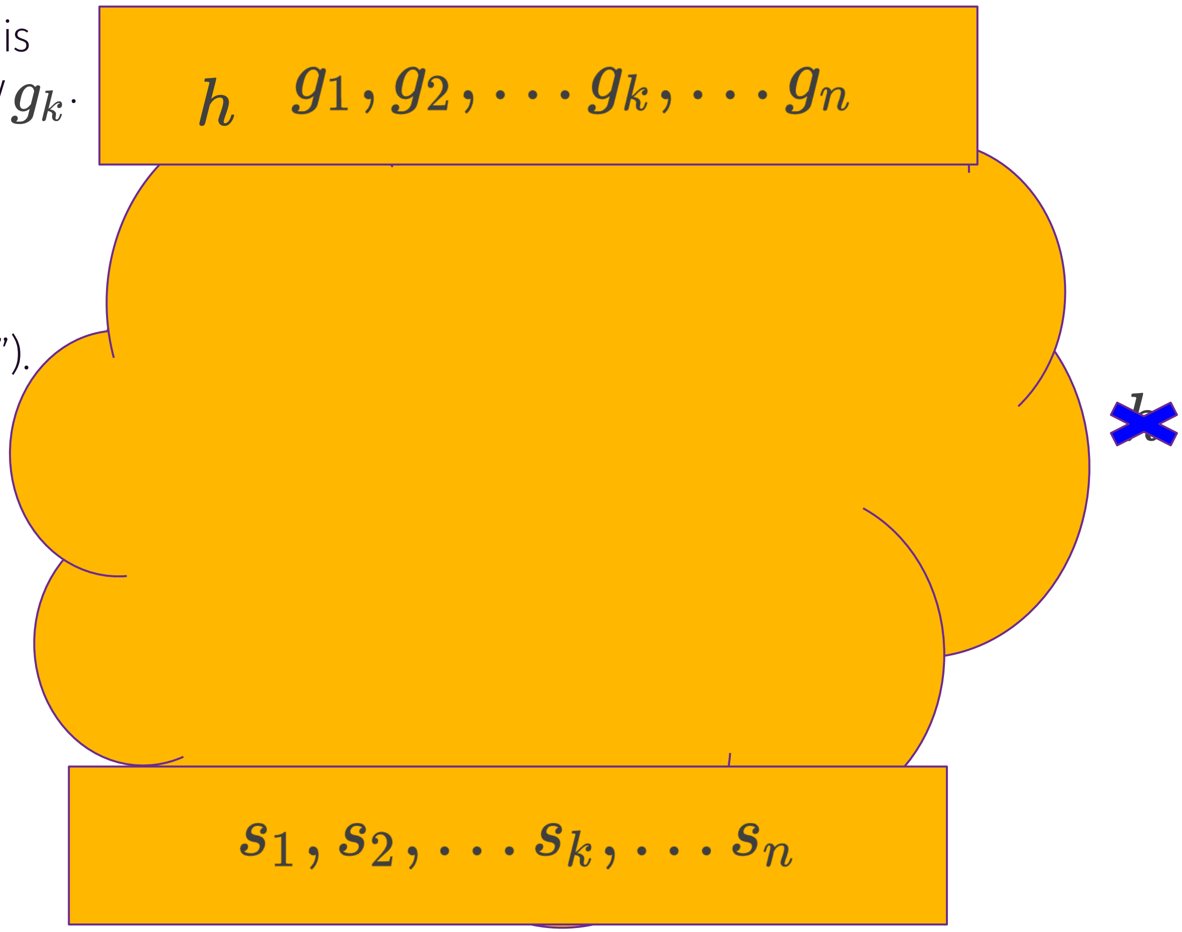
Proof

h consistent => h more specific than some g_k and more general than some s_k

Assume h consistent and is not a specialisation of any g_k.

Then h is by definition a member of the G-Set (and therefore not "in between").

Analogous for the S-Set.

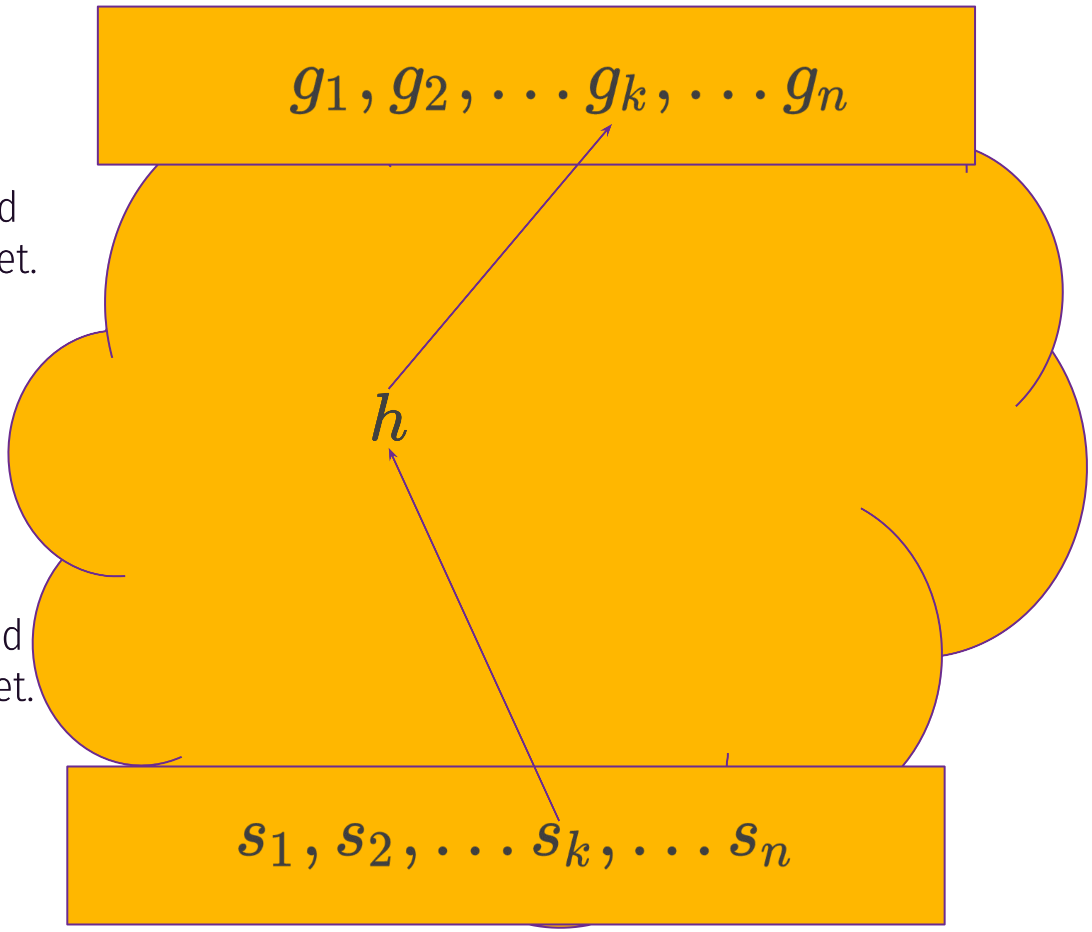


Proof

h more specific than some g_k and more general than some s_k
=> h consistent

Assume h more specific than some g_k

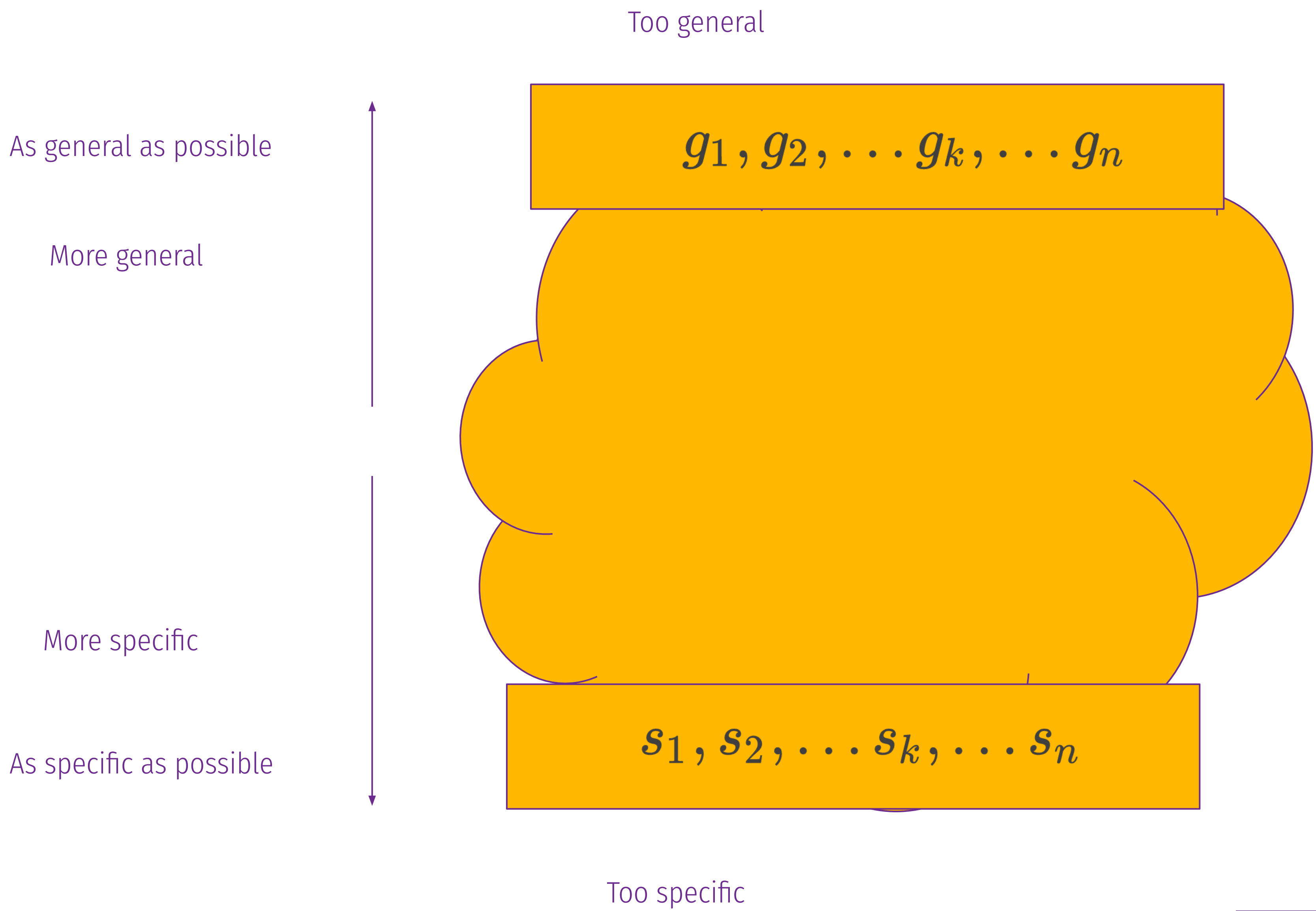
Then h must reject all negative examples rejected by all members of the G-set.



Assume h more general than some s_k

Then h must accept all positive examples accepted by all members of the S-set.

Therefore, h is consistent with all examples so far



Too general
As general as possible
$g_1, g_2, \ldots g_k, \ldots g_n$
More general
More specific
As specific as possible
$s_1, s_2, \ldots s_k, \ldots s_n$
Too specific

Version Space Learning

```
def vsl(examples):
  s_set, g_set := initialise()
  For e in examples:
    if not s_set or not g_set: return None
    for s in s_set:
      if e is false positive for s:
        s_set.pop(s)
      elif e is false negative for s:
        s_set.pop(s)
        s_set.update(ss for ss in generalisations(s, g_set))
    for g in g_set:
      if e is false positive for g:
        g_set.pop(g)
        g_set.update(gs for gs in specifications(g, s_set))
      elif e is false negative for s:
        g_set.pop(g)
  return compute_version_space(s_set, g_set)
```

Version Space Learning

```
def vsl(examples):
  s_set, g_set := initialise()
  For e in examples:
    if not s_set or not g_set: return None
    for s in s_set:
      if e is false positive for s:
        s_set.pop(s)
      elif e is false negative for s:
        s_set.pop(s)
        s_set.update(ss for ss in generalisations(s, g_set))
    for g in g_set:
      if e is false positive for g:
        g_set.pop(g)
        g_set.update(gs for gs in specifications(g, s_set))
      elif e is false negative for s:
        g_set.pop(g)
  return compute_version_space(s_set, g_set)
```

All immediate generalisations of s that are not too general, i.e. more specific than some member of g_set

Version Space Learning

```
def vsl(examples):
  s_set, g_set := initialise()
  For e in examples:
    if not s_set or not g_set: return None
    for s in s_set:
      if e is false positive for s:
        s_set.pop(s)
      elif e is false negative for s:
        s_set.pop(s)
        s_set.update(ss for ss in generalisations(s, g_set))
    for g in g_set:
      if e is false positive for g:
        g_set.pop(g)
        g_set.update(gs for gs in specifications(g, s_set))
      elif e is false negative for s:
        g_set.pop(g)
  return compute_version_space(s_set, g_set)
```

All immediate specifications of g that are not too specific, i.e. more general than some member of g_set

Version Space Learning

```
def vsl(examples):
  s_set, g_set := initialise()
  For e in examples:
    if not s_set or not g_set: return None
    for s in s_set:
      if e is false positive for s:
        s_set.pop(s)
      elif e is false negative for s:
        s_set.pop(s)
        s_set.update(ss for ss in generalisations(s, g_set))
    for g in g_set:
      if e is false positive for g:
        g_set.pop(g)
        g_set.update(gs for gs in specifications(g, s_set))
      elif e is false negative for s:
        g_set.pop(g)
  return compute_version_space(s_set, g_set)
```

Everything “in between” the S-Set and G-Set

Takeaways

- The hypothesis space has an **innate structure** and hypotheses can be ordered with regard to the generalisation/specialisation relation
- This order can be used to speed up the search for a hypothesis over brute-force search
- The algorithms are prone to noise, i.e. examples with erroneous classifications
- We didn't make any use of existing knowledge so far!