17.3 Policy Iteration

- Beginning from some initial policy $\pi_0$ alternate:
  - **Policy evaluation**: given a policy $\pi_i$, calculate $U_i = U^{\pi_i}$, the utility of each state if $\pi_i$ were to be executed
  - **Policy improvement**: Calculate the new MEU policy $\pi_{i+1}$, using one-step look-ahead based on $U_i$ (Equation (*))

- The algorithm terminates when the policy improvement step yields no change in utilities
- At this point, we know that the utility function $U_i$ is a fixed point of the Bellman update and a solution to the Bellman equations, so $\pi_i$ must be an optimal policy
- Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate

Because at the $i$th iteration the policy $\pi_i$ specifies the action $\pi_i(s)$ in state $s$, there is no need to maximize over actions in policy iteration

We have a simplified version of the Bellman equation:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi_i(s)) U_i(s')$$

- Now the nonlinear max has been removed, and we have linear equations
- A system of linear equations with $n$ equations with $n$ unknowns can be solved exactly in time $O(n^3)$ by standard linear algebra methods
- Instead of using a cubic amount of time to reach the exact solution, we can instead perform some number simplified value iteration steps to give a reasonably good approximation of the utilities
18 LEARNING FROM EXAMPLES

- An intelligent agent may have to learn, for instance, the following components:
  - A direct mapping from conditions on the current state to actions
  - A means to infer relevant properties of the world from the percept sequence
  - Information about the way the world evolves and about the results of possible actions the agent can take
  - Utility information indicating the desirability of world states
  - Action-value information indicating the desirability of actions
  - Goals that describe classes of states whose achievement maximizes the agent’s utility

- The type of feedback available for learning determines the nature of the learning problem that the agent faces
  - **Supervised learning** involves learning a function from examples of its inputs and outputs
  - **Unsupervised learning** involves learning patterns in the input when no specific output values are supplied
  - In **reinforcement learning** the agent must learn from reinforcement (reward, punishment, less exact feedback than in supervised learning)

- The representation of the learned information plays an important role in determining how the learning algorithm must work
18.2 Supervised Learning

- In deterministic supervised learning, the aim is to recover the unknown function $f$ given examples $(x, f(x))$, where $x$ is the input (vector).
- In pure inductive inference (or induction), the result is a hypothesis $h$, which is a function that approximates $f$.
- A good hypothesis will generalize well – will predict unseen instances correctly.
- The hypothesis is chosen from a hypothesis space $H$.
- For example, when both $x$ and $f(x)$ are real numbers, then $H$ can be, e.g., the set of polynomials of degree at most $k$: $3x^2 + 2$, $x^{17} - 4x^3$, ...

- A consistent hypothesis agrees with all the data.
- How do we choose from among multiple consistent hypotheses?
- Occam’s (Ockham’s) razor: prefer the simplest hypothesis consistent with the data.
- How to define simplicity?
The learning problem is called **realizable** if the hypothesis space contains the true function; otherwise it is **unrealizable**.

On the other hand, in the name of better generalization ability it may be sensible to trade off exactness of fitting to simplicity of the hypothesis.

In other words, it may be sensible to be content with a hypothesis fitting the data less perfectly as long as it is simple.

The hypothesis space needs to be restricted so that finding a hypothesis that fits the data stays computationally efficient.

Machine learning concentrates on learning relatively simple knowledge representations.

---

**18.3 Learning Decision Trees**

A **decision tree** takes as input an object or situation described by a set of attributes.

It returns a decision — the predicted output value for the input.

If the output values are discrete, then the decision tree **classifies** the inputs.

Learning a continuous function is called **regression**.

Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with possible values of the test.

Each leaf node in the tree specifies the value to be returned if the leaf is reached.

To process an input, it is directed from the root of the tree through internal nodes to a leaf, which determines the output value.
A decision tree (of reasonable size) is an easy to comprehend way of representing knowledge. Important in practice, heuristically learnable.

The previous decision tree corresponds to the goal predicate $\text{WillWait} - \text{whether to wait for a table in a restaurant}$.

The goal predicate can be seen as an assertion of the form

$v: \text{WillWait}(s) \iff (p_1(s) \lor \cdots \lor p_n(s))$,

where each $p_i(s)$ is a conjunction of tests corresponding to a path from the root of the tree to a leaf with a positive outcome.

An exponentially large decision tree can express any Boolean function.
Typically, decision trees can represent many functions with much smaller trees.

For some kinds of functions this, however, is a real problem, e.g., xor and maj need exponentially large decision trees.

Decision trees, like any other knowledge representation, are good for some kinds of functions and bad for others.

Consider the set of all Boolean functions on \( n \) attributes.

How many different functions are in this set?

The truth table has \( 2^n \) rows, so there are \( 2^{2^n} \) different functions.

For example, when \( n = 6 \Rightarrow 2^6 > 18 \times 10^{18}, n = 10 \Rightarrow 2^{1024} > 10^{308}, \) and \( n = 20 \Rightarrow > 10^{300} \).

We will need some ingenious algorithms to find consistent hypotheses in such a large space.

---

Top-down induction of decision trees

The input to the algorithm is a training set, which consists of examples \((X, y)\), where \( X \) is a vector of input attribute values and \( y \) is the single output value (class value) attached to them.

We could simply construct a consistent decision tree that has one path from the root to a leaf for each example.

Then we would be able to classify all training examples correctly, but the tree would not be able to generalize at all.

Applying Occam’s razor, we should find the smallest decision tree that is consistent with the examples.

Unfortunately, for any reasonable definition of “smallest,” finding the smallest tree is an intractable problem.
Successful decision tree learning algorithms are based on simple heuristics and do a good job of finding a smallish tree. The basic idea is to test the most important attribute first. Because the aim is to classify instances, "most important" attribute is the one that makes the most difference to the classification of an example. Actual decision tree construction happens with a recursive algorithm:

- First the most important attribute is chosen to the root of the tree,
- the training data is divided according to the values of the chosen attribute, and
- (sub)tree construction continues using the same idea.

Tree growConsTree( Attrs A, Exs S )
{
  if ( all examples in S have class C )
    return an one-leaf tree labeled by C;
  else
    select an attribute α from A;
    partition S into S₁, ..., Sₖ by the value of α;
    for ( i = 1; i <= k; i++ )
      Tᵢ = growConsTree( A-{α}, Sᵢ);
    return a tree T that has α in its root and Tᵢ as its i-th subtree; }

• If there are no examples left
  ⇒ no such example has been observed, and we return a default
  value calculated from the majority classification at the node's
  parent (or the majority classification at the root)
• If there are no attributes left but still instances of several classes
  in the remaining portion of the data,
  ⇒ these examples have exactly the same description, but
  different classification
• Then we say that there is noise in the data
• Noise may follow either when the attributes do not give enough
  information to describe the situation fully, or when the domain is
  truly nondeterministic
• One simple way out of this problem is to use a majority vote

• If there are no examples left
  ⇒ no such example has been observed, and we return a default
  value calculated from the majority classification at the node's
  parent (or the majority classification at the root)
• If there are no attributes left but still instances of several classes
  in the remaining portion of the data,
  ⇒ these examples have exactly the same description, but
  different classification
• Then we say that there is noise in the data
• Noise may follow either when the attributes do not give enough
  information to describe the situation fully, or when the domain is
  truly nondeterministic
• One simple way out of this problem is to use a majority vote

Choosing attribute tests

• The idea is to pick the attribute that goes as far as possible
toward providing an exact classification of the examples
• A perfect attribute divides the examples into sets that contain
  only instances of one class
• A really useless attribute leaves the example sets with roughly
  the same proportion of instances of all classes as the original set
• To measure the usefulness of attributes we can use, for instance,
  the expected amount of information provided by the attribute –
  i.e., its Shannon entropy
• Information theory measures information content in bits
• One bit of information is enough to answer a yes/no question
  about which one has no idea, such as the flip of a fair coin
• In general, if the possible answers \( v_i \) have probabilities \( P(v_i) \), then the entropy \( H \) of the actual answer is:
\[
H(P(v_1), \ldots, P(v_n)) = \sum_{i=1}^{n} - P(v_i) \log_2 P(v_i)
\]

• For example, \( H(\frac{1}{2}, \frac{1}{2}) = 2(-\frac{1}{2} \log_2(\frac{1}{2})) = 1 \text{ bit} \)

• In choosing attribute tests, we want to calculate the change of the value distribution \( P(C) \) of the class attribute \( C \), if the training set \( S \) is divided into subsets according to the value of attribute \( a \):
\[
\text{Gain}_S(P(C) \mid a) = H_S(P(C)) - \text{Gain}_S(P(C) \mid a),
\]
where
\[
\text{Gain}_S(P(C) \mid a) = \sum_{S_i} \left( \frac{|S_i|}{|S|} \right) \cdot H_{S_i}(P(C)),
\]
when \( a \) divides \( S \) in subsets \( S_i \)

Let the training set \( S \) contain **14 apples** and **6 pears**

• Hence,
\[
H_S(P(C)) = H(0.7, 0.3) \approx 0.7 \times 0.515 + 0.3 \times 1.737 \approx 0.881
\]

• Suppose that attribute \( a \) divides the data s.t.
   \[
   S_1 = \{7 \text{ apples}, 3 \text{ pears}\},
   \]
   \[
   S_2 = \{7 \text{ apples}\}, \quad \text{and}
   \]
   \[
   S_3 = \{3 \text{ pears}\}
   \]
   
   then
\[
\text{Gain}_S(P(C) \mid a) = \sum_{S_i} \left( \frac{|S_i|}{|S|} \right) \cdot H_{S_i}(P(C))
\]
\[
\approx (10/20) \times H(0.7, 0.3) + 0 + 0
\]
\[
\approx \frac{1}{2} \times 0.881 \approx 0.441
\]
Assessing performance of learning algorithms

- Divide the set of examples into disjoint training set and test set
- Apply the training algorithm to the training set, generating a hypothesis $h$
- Measure the percentage of examples in the test set that are correctly classified by $h$: $h(x) = y$ for an $(x,y)$ example
- Repeat the above-mentioned steps for different sizes of training sets and different randomly selected training sets of each size
- The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set
- Plotting this function on a graph gives the learning curve

- An alternative (better) approach to testing is cross-validation

The idea in $k$-fold cross-validation is that each example serves double duty as training data and test data
- First we split the data into $k$ equal subsets
- We then perform $k$ rounds of learning; on each round $1/k$ of the data is held out as a test set and the remaining examples are used as training data
- The average test set score of the $k$ rounds should then be a better estimate than a single score
- Popular values for $k$ are 5 and 10 – enough to give an estimate that is statistically likely to be accurate, at the cost of 5 to 10 times longer computation time
- The extreme is $k = n$, also known as leave-one-out cross-validation (or jackknife)