Why Parameter Tuning?

• What we have seen so far
  – many learning algorithms for classification, regression, ...

• Many of those have parameters
  – e.g., $k$ for k nearest neighbors
  – splitting threshold in decision tree learning
  – hidden layers in neural networks
  – $C$ and epsilon for SVMs

• But what is their effect?
  – hard to tell in general
  – rules of thumb are rare
Parameter Tuning – a Naive Approach

• You probably know that approach from the exercises
  1. run classification/regression algorithm
  2. look at the results (e.g., accuracy, RMSE, …)
  3. choose different parameter settings, go to 1

• Questions:
  • when to stop?
  • how to select the next parameter setting to test?
Parameter Tuning – Avoid Overfitting!

- Recap overfitting:
  - classifiers may overadapt to training data
  - the same holds for parameter settings

- Possible danger:
  - find parameters that work well on the training set
  - but not on the test set

- Remedy:
  - use cross-validation for testing parameter settings
Parameter Tuning: Brute Force

- Try all parameter combinations that exist

- Consider, e.g., the k-NN classifier in RapidMiner
  - try 30 different distance measures
  - try all k from 1 to 100
  - use weighting or not
    → 6,000 runs of k-NN

- Plus: we use 10-fold CV for evaluating the parameter settings
  → that makes a total of 60,000 runs of k-NN

→ we need a better strategy than brute force!
Intermezzo: Beyond Parameter Tuning

• Parameter tuning is an optimization problem
• Finding optimal values for N variables

• Properties of the problem:
  – the underlying model is unknown
    • i.e., we do not know changing a variable will influence the results
  – we can tell how good a solution is when we see it
    • i.e., by running a classifier with the given parameter set
  – but looking at each solution is costly
    • e.g., 60,000 runs of k-NN

• Such problems occur quite frequently
Intermezzo: Beyond Parameter Tuning

• Related problem:
  – feature subset selection
  – cf. Data Mining 2, first lecture

• Given n features, brute force requires $2^n$ evaluations
  – for 20 features, that is already one million
    → ten million with cross validation
Intermezzo: Beyond Parameter Tuning

- Knapsack problem
  - given a maximum weight you can carry
  - and a set of items with different weight and monetary value
  - pack those items that maximize the monetary value

- Problem is NP hard
  - i.e., deterministic algorithms require an exponential amount of time
Intermezzo: Beyond Parameter Tuning

• Many optimization problems are NP hard
  – Routing problems (Traveling Salesman Problem)
  – Integer factorization
    hard enough to be used for cryptography
  – Resource use optimization
    • e.g., minimizing cutoff waste
  – Chip design
    • minimizing chip sizes
Intermezzo: Beyond Parameter Tuning

http://xkcd.com/287/
Parameter Tuning: Brute Force

• Properties of Brute Force search
  – guaranteed to find the best parameter setting
  – too slow in most practical cases

• Grid Search in RapidMiner:
  – performs a brute force search
  – with equal-width steps on non-discrete numerical attributes (e.g., 10, 20, 30, .., 100)

• Parameters with a wide range (e.g., 0.0001 to 1,000,000)
  – logarithmic steps may perform better
  – with ten equal-width steps, the first step would be 1,000
  – but what if the optimum is around 0.1?
Parameter Tuning: Heuristics

• Properties of Brute Force search
  – guaranteed to find the best parameter setting
  – too slow in most practical cases

• Needed:
  – solutions that take less time/computation
  – and often find the best parameter setting
  – or find a near-optimal parameter setting
Beyond Brute Force

https://xkcd.com/399/

03/24/14  Heiko Paulheim, Robert Meusel
Parameter Tuning: One After Another

• Given \( n \) parameters with \( m \) degrees of freedom
  – brute force takes \( m^n \) runs of the base classifier

• Simple tweak:
  1. start with default settings
  2. try all options for the first parameter
  3. try all options for the second parameter
  4. ...

• This reduces the runtime to \( n \times m \)
  – but we may miss the best solution
Parameter Tuning: Interaction Effects

• Interaction effects make parameter tuning hard
  – i.e., changing one parameter may change the optimal settings for another one

• Example: two parameters p and q, each with values 0, 1, and 2
  – the table depicts the accuracy

<table>
<thead>
<tr>
<th></th>
<th>p=0</th>
<th>p=1</th>
<th>p=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>q=0</td>
<td>0.5</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>q=1</td>
<td>0.4</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>q=2</td>
<td>0.1</td>
<td>0.2</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Parameter Tuning: Interaction Effects

• If we try to optimize one parameter by another (first p, then q)
  – we end at $p=0,q=0$ in six out of nine cases
  – on average, we investigate 2.3 solutions

<table>
<thead>
<tr>
<th></th>
<th>p=0</th>
<th>p=1</th>
<th>p=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>q=0</td>
<td>0.5</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>q=1</td>
<td>0.4</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>q=2</td>
<td>0.1</td>
<td>0.2</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Hill-Climbing Search

- a.k.a. *greedy local search*
- always search in the direction of the steepest ascend
  - "Like climbing Everest in thick fog with amnesia"

```plaintext
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
               neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```
Hill-Climbing Search

- Problem: depending on initial state, can get stuck in local maxima
Hill Climbing Search

- Given our previous problem
  - we end up at the optimum in three out of nine cases
  - but the local optimum \((p=0, q=0)\) is reached in six out of nine cases!
  - on average, we investigate 2.1 solutions

<table>
<thead>
<tr>
<th></th>
<th>p=0</th>
<th>p=1</th>
<th>p=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>q=0</td>
<td>0.5</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>q=1</td>
<td>0.4</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>q=2</td>
<td>0.1</td>
<td>0.2</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Variations of Hill Climbing Search

• Stochastic hill climbing
  – random selection among the uphill moves
  – the selection probability can vary with the steepness of the uphill move

• First-choice hill climbing
  – generating successors randomly until a better one is found, then pick that one

• Random-restart hill climbing
  – run hill climbing with different seeds
  – tries to avoid getting stuck in local maxima
Local Beam Search

- Keep track of $k$ states rather than just one
- Start with $k$ randomly generated states
- At each iteration, all the successors of all $k$ states are generated
- Select the $k$ best successors from the complete list and repeat
Simulated Annealing

- Escape local maxima by allowing “bad” moves
  - Idea: but gradually decrease their size and frequency
- Origin: metallurgical annealing

- Bouncing ball analogy:
  - Shaking hard (= high temperature)
  - Shaking less (= lower the temperature)
- If $T$ decreases slowly enough, best state is reached
Simulated Annealing

function SIMULATED-ANNEALING( problem, schedule) return a solution state
  input: problem, a problem
  schedule, a mapping from time to temperature
  local variables: current, a node.
    next, a node.
    T, a “temperature” controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ∆E ← VALUE[next] - VALUE[current]
    if ∆E > 0 then current ← next
    else current ← next only with probability $e^{∆E / T}$
Genetic Algorithms

• Inspired by evolution
• Overall idea:
  – use a population of individuals (solutions)
  – create new individuals by crossover
  – introduce random mutations
  – from each generation, keep only the best solutions (“survival of the fittest”)
• Developed in the 1970s
• John H. Holland:
  – Standard Genetic Algorithm (SGA)

Charles Darwin (1809-1882)
Genetic Algorithms

• Basic ingredients:
  – individuals: the solutions
    • parameter tuning: a parameter setting
  – a fitness function
    • parameter tuning: performance of a parameter setting
      (i.e., run learner with those parameters)
  – a crossover method
    • parameter tuning: create a new setting from two others
  – a mutation method
    • parameter tuning: change one parameter
  – survivor selection
SGA Reproduction Cycle

1. Select parents for the mating pool
   (size of mating pool = population size)
2. Shuffle the mating pool
3. For each consecutive pair apply crossover with probability $p_c$, otherwise copy parents
4. For each offspring apply mutation
   (bit-flip with probability $p_m$ independently for each bit)
5. Replace the whole population with the resulting offspring
SGA Operators: 1-point crossover

- Choose a random point on the two parents
- Split parents at this crossover point
- Create children by exchanging tails
- $P_c$ typically in range (0.6, 0.9)
SGA Operators: Mutation

• Alter each gene independently with a probability $p_m$
• $p_m$ is called the mutation rate
  – Typically between $1/\text{pop\_size}$ and $1/\text{chromosome\_length}$

Parent: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Child: 0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1
Main idea: better individuals get higher chance

- Chances proportional to fitness
- Implementation: roulette wheel technique
  
  » Assign to each individual a part of the roulette wheel
  
  » Spin the wheel n times to select n individuals

SGA Operators: Selection

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

- fitness(A) = 3
- fitness(B) = 1
- fitness(C) = 2
Crossover OR Mutation?

• Decade long debate: which one is better / necessary / main-background

• Answer (at least, rather wide agreement):
  – it depends on the problem, but
  – in general, it is good to have both
  – both have another role
  – mutation-only-EA is possible, xover-only-EA would not work
Crossover OR Mutation? (cont’d)

• Exploration: Discovering promising areas in the search space, i.e. gaining information on the problem
• Exploitation: Optimising within a promising area, i.e. using information
• There is co-operation AND competition between them
  • Crossover is explorative, it makes a *big* jump to an area somewhere “in between” two (parent) areas
  • Mutation is exploitative, it creates random *small* diversions, thereby staying near (in the area of) the parent
Crossover OR Mutation? (cont'd)

• Recall the solution space example from Hill Climbing
  – crossover makes big jumps
  – mutation makes small steps
• Only crossover can combine information from two parents
• Only mutation can introduce new information (alleles)
• Crossover does not change the allele frequencies of the population (thought experiment: 50% 0’s on first bit in the population, ?% after performing $n$ crossovers)
• To hit the optimum you often need a ‘lucky’ mutation
The Simple GA

• Has been subject of many (early) studies
  – still often used as benchmark for novel GAs

• Shows many shortcomings, e.g.
  – Representation is too restrictive
  – Mutation & crossovers only applicable for bit-string representation
  – Selection mechanism sensitive for converging populations with close fitness values
  – Generational population model (step 5 in the reproduction cycle) can be improved with explicit survivor selection
Representation Problems

- Integers are represented by a binary substring
  - e.g., the number 1024 as 10000000000

- Mutation of single bits has effects on different scales
  - e.g., 10000000000 → 10000000001 (1024 → 1025)
  - e.g., 10000000000 → 00000000000 (1024 → 0)

- Early solution:
  - gray coding

- Recent solutions:
  - use numbers in representation
  - e.g., x-over by average, mutation by small delta change

<table>
<thead>
<tr>
<th>N</th>
<th>Binary</th>
<th>Gray</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0110</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0101</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Arithmetic Crossover

- Most commonly used
- Parents: \( \langle x_1, \ldots, x_n \rangle \) and \( \langle y_1, \ldots, y_n \rangle \)
- child_1 is:

\[
a \cdot \bar{x} + (1 - a) \cdot \bar{y}
\]

- reverse for other child. e.g. with \( \alpha = 0.5 \)

\[
\begin{array}{ccccccccccccccc}
0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 \\
0.3 & 0.2 & 0.3 & 0.2 & 0.3 & 0.2 & 0.3
\end{array}
\quad
\begin{array}{ccccccccccccccc}
0.2 & 0.2 & 0.3 & 0.3 & 0.4 & 0.4 & 0.5 & 0.5 & 0.6 \\
0.2 & 0.2 & 0.3 & 0.3 & 0.4 & 0.4 & 0.5 & 0.5 & 0.6
\end{array}
\]
Further Variants in Genetic Algorithms

- Two-parent vs. multi-parent cross-over
- Rank based selection
  - instead of wheel-of-fortune implementation
  - avoids strong dominance of fitter individuals
- Tournament based selection
  - pairwise comparison of individuals
  - score of individual: number of wins
- Age-based survivor selection
  - i.e., more likely to delete older survivors
- ...
Parameter Tuning Operators in RapidMiner

- **Optimize Parameters (Evolutionary)**

  - **Parameters**
    - Max generations: 50
    - Use early stopping: off
    - Specify population size: on
    - Population size: 5
    - Keep best: on
    - Mutation type: switching_mutation
    - Selection type: tournament
    - Tournament fraction: 0.25
    - Crossover prob: 0.9
    - Use local random seed: off
    - Show convergence plot: off
Genetic Feature Subset Selection

- Feature Subset Selection
  - can also be solved by Genetic Programming

- Individuals: feature subsets
- Representation: binary
  - 1 = feature is included
  - 0 = feature is not included

- Fitness: classification performance
- Crossover: combine selections of two subsets
- Mutation: flip bits
Selecting a Learner

• So far, we have looked at finding good parameters for a learner
  – the learner was always fixed

• A similar problem is selecting a learner for the task at hand
• Again, we could go with search
• Another approach is meta learning
Selecting a Learner by Meta Learning

• Meta Learning
  – i.e., learning about learning

• Goal: learn how well a learner will perform on a given dataset
  – features: dataset characteristics, learning algorithm
  – prediction target: accuracy, RMSE, ...
Selecting a Learner by Meta Learning

- Used in the *Automatic System Construction* extension
- regression trained on
  - 90 datasets
  - 54 features

- Examples for features
  - number of instances/attributes
  - fraction of nominal/numerical attributes
  - min/max/average entropy of attributes
  - skewness of classes
  - ...

Selecting a Learner by Meta Learning

- Used in the *Automatic System Construction* extension
...and now for something completely different.

• Recap: search heuristics are good for problems where...
  – finding an optimal solution is difficult
  – evaluating a solution candidate is easy
  – the search space of possible solutions is large
• Possible solution: genetic programming

• We have encountered such problems quite frequently
• Example: learning an optimal decision tree from data
Genetic Decision Tree Learning

• e.g., GAIT (Fu et al., 2003)
  – also the source of the pictures on the following slides

• Population: candidate decision trees
  – initialization: e.g., trained on small subsets of data

• Create new decision trees by means of
  – crossover
  – mutation

• Fitness function: e.g., accuracy
Genetic Decision Tree Learning

- Crossover:

  Parent 1

  Parent 2

  Child 1

  Child 2

Subtree-to-subtree Crossover

Parent 1

Parent 2

Child 1

Child 2

Subtree-to-leaf Crossover
Genetic Decision Tree Learning

- Mutation:
Genetic Decision Tree Learning

- Feasibility Check:

![Genetic Decision Tree Diagram](image-url)
Combination of GP with other Learning Methods

• Rule Learning (“Learning Classifier Systems”), since late 70s
  – Population: set of rule sets (!)
  – Crossover: combining rules from two sets
  – Mutation: changing a rule

• Artificial Neural Networks
  – Easiest solution: fixed network layout
  – The network is then represented as an ordered set (vector) of weights e.g., [0.8, 0.2, 0.5, 0.1, 0.1, 0.2]
  – Crossover and mutation are straight forward
  – There are variants that encode different layouts as well
Wrap-Up

• Parameter tuning is important
  – many learning methods work poorly with standard parameters
  – often no global optimum, dataset dependent

• Parameter tuning has a large search space
  – trying all combinations is infeasible
  – interaction effects do not allow for one-by-one tuning
Wrap-Up

• Heuristic Methods
  – Hill climbing with variations
  – Beam search
  – Simulated Annealing
  – Genetic Programming

• Other uses of genetic programming
  – Feature subset selection
  – Model fitting
Questions?