Machine Learning: More ANNs, Genetic and Evolutionary Computation (GEC)  
Discussion: Genetic Programming

William H. Hsu  
Department of Computing and Information Sciences, KSU

KSOL course page: http://snipurl.com/v9v3  
Course web site: http://www.kddresearch.org/Courses/CIS730  
Instructor home page: http://www.cis.ksu.edu/~bhsu

Reading for Next Class:

Chapter 20, Russell and Norvig
• **Perceptron: Single Neuron Model**
  
  – *aka* Linear Threshold Unit (LTU) or Linear Threshold Gate (LTG)
  
  – Net input to unit: defined as linear combination
    \[
    \text{net} = \sum_{i=0}^{n} w_i x_i
    \]
  
  – Output of unit: threshold (activation) function on net input (threshold \( \theta = w_0 \))

• **Perceptron Networks**
  
  – Neuron is modeled using a unit connected by weighted links \( w_i \) to other units
  
  – Multi-Layer Perceptron (MLP): next lecture
Decision Surface of a Perceptron

- **Perceptron: Can Represent Some Useful Functions**
  - LTU emulation of logic gates (McCulloch and Pitts, 1943)
  - e.g., What weights represent \( g(x_1, x_2) = AND(x_1, x_2) \)? \( OR(x_1, x_2) \)? \( NOT(x) \)?

- **Some Functions Not Representable**
  - e.g., not linearly separable
  - Solution: use networks of perceptrons (LTUs)
Learning Rules for Perceptrons

- **Learning Rule ≡ Training Rule**
  - Not specific to supervised learning
  - Context: updating a model

- **Hebbian Learning Rule (Hebb, 1949)**
  - Idea: if two units are both active (“firing”), weights between them should increase
  - \( w_{ij} = w_{ij} + r_i o_j \) where \( r \) is a learning rate constant
  - Supported by neuropsychological evidence

- **Perceptron Learning Rule (Rosenblatt, 1959)**
  - Idea: when a target output value is provided for a single neuron with fixed input, it can incrementally update weights to learn to produce the output
  - Assume binary (boolean-valued) input/output units; single LTU
  - \( w_i \leftarrow w_i + \Delta w_i \)
  
  \[
  \Delta w_i = r(t - o)x_i 
  \]
  
  where \( t = c(x) \) is target output value, \( o \) is perceptron output, \( r \) is small learning rate constant (e.g., 0.1)
  - Can prove convergence if \( D \) linearly separable and \( r \) small enough
Perceptron Learning Algorithm

• **Simple Gradient Descent Algorithm**
  - Applicable to concept learning, symbolic learning (with proper representation)

• **Algorithm Train-Perceptron** \( (D \equiv \{<x, t(x) \equiv c(x)>\}) \)
  - Initialize all weights \( w_i \) to random values
  - WHILE not all examples correctly predicted DO
    - FOR each training example \( x \in D \)
      - Compute current output \( o(x) \)
      - FOR \( i = 1 \) to \( n \)
        - \( w_i \leftarrow w_i + r(t - o)x_i \) // perceptron learning rule

• **Perceptron Learnability**
  - Recall: can only learn \( h \in H \) - i.e., linearly separable (LS) functions
  - Minsky and Papert, 1969: demonstrated representational limitations
    - e.g., parity \( (n\text{-attribute XOR}: x_1 \oplus x_2 \oplus \cdots \oplus x_n) \)
    - e.g., symmetry, connectedness in visual pattern recognition
    - Influential book *Perceptrons* discouraged ANN research for ~10 years
  - NB: $64K$ question - “Can we transform learning problems into LS ones?”
Functional Definition

- \( f(x) = 1 \) if \( w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \geq \theta \), 0 otherwise
- \( \theta \): threshold value

Linearly Separable Functions

- **NB:** \( D \) is LS does not necessarily imply \( c(x) = f(x) \) is LS!
- Disjunctions: \( c(x) = x_1' \lor x_2' \lor \ldots \lor x_m' \)
- \( m \) of \( n \): \( c(x) = \text{at least 3 of (} x_1', x_2', \ldots, x_m' \text{)} \)
- Exclusive OR (XOR): \( c(x) = x_1 \oplus x_2 \)
- General DNF: \( c(x) = T_1 \lor T_2 \lor \ldots \lor T_m; T_i = l_1 \land l_2 \land \ldots \land l_k \)

Change of Representation Problem

- Can we transform non-LS problems into LS ones?
- Is this meaningful? Practical?
- Does it represent a significant fraction of real-world problems?
Perceptron Convergence Theorem
- Claim: If there exist a set of weights that are consistent with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
- Proof: well-founded ordering on search region (“wedge width” is strictly decreasing) - see Minsky and Papert, 11.2-11.3
- Caveat 1: How long will this take?
- Caveat 2: What happens if the data is not LS?

Perceptron Cycling Theorem
- Claim: If the training data is not LS the perceptron learning algorithm will eventually repeat the same set of weights and thereby enter an infinite loop
- Proof: bound on number of weight changes until repetition; induction on $n$, the dimension of the training example vector - MP, 11.10

How to Provide More Robustness, Expressivity?
- Objective 1: develop algorithm that will find closest approximation (today)
- Objective 2: develop architecture to overcome representational limitation (next lecture)
Understanding Gradient Descent for Linear Units

- Consider simpler, unthresholded linear unit:
  \[ o(\tilde{x}) = \text{net}(\tilde{x}) = \sum_{i=0}^{n} w_i x_i \]

- Objective: find “best fit” to \(D\)

Approximation Algorithm

- Quantitative objective: minimize error over training data set \(D\)
- Error function: sum squared error (SSE)
  \[ E[\tilde{w}] = \text{error}_D[\tilde{w}] = \frac{1}{2} \sum_{x \in D} (t(x) - o(x))^2 \]

How to Minimize?

- Simple optimization
- Move in direction of steepest gradient in weight-error space
  - Computed by finding tangent
  - i.e. partial derivatives (of \(E\)) with respect to weights \((w_i)\)
Gradient Descent: Derivation of Delta/LMS (Widrow-Hoff) Rule

**Definition: Gradient**

$$\nabla E[\vec{w}] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right]$$

**Modified Gradient Descent Training Rule**

$$\Delta \vec{w} = -r \nabla E[\vec{w}]$$

$$\Delta w_i = -r \frac{\partial E}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left[ \frac{1}{2} \sum_{x \in D} (t(x) - o(x))^2 \right] = \frac{1}{2} \sum_{x \in D} \frac{\partial}{\partial w_i} (t(x) - o(x))^2$$

$$= \frac{1}{2} \sum_{x \in D} \left[ 2(t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - o(x)) \right] = \sum_{x \in D} \left[ (t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - \vec{w} \cdot \vec{x}) \right]$$

$$\frac{\partial E}{\partial w_i} = \sum_{x \in D} [(t(x) - o(x))(-x_i)]$$
Algorithm Gradient-Descent \((D, r)\)

- Each training example is a pair of the form \(<x, t(x)>\), where \(x\) is the vector of input values and \(t(x)\) is the output value. \(r\) is the learning rate (e.g., 0.05)
- Initialize all weights \(w_i\) to (small) random values
- UNTIL the termination condition is met, DO
  
  Initialize each \(\Delta w_i\) to zero
  
  FOR each \(<x, t(x)>\) in \(D\), DO
    
    Input the instance \(x\) to the unit and compute the output \(o\)
    
    FOR each linear unit weight \(w_i\), DO
      
      \[\Delta w_i \leftarrow \Delta w_i + r(t - o)x_i\]
    
  RETURN final \(w\)

Mechanics of Delta Rule

- Gradient is based on a derivative
- Significance: later, will use nonlinear activation functions (aka transfer functions, squashing functions)
• **LS Concepts: Can Achieve Perfect Classification**
  – Example A: perceptron training rule converges
• **Non-LS Concepts: Can Only Approximate**
  – Example B: not LS; delta rule converges, but can’t do better than 3 correct
  – Example C: not LS; better results from delta rule
• **Weight Vector** \( w = \text{Sum of Misclassified } x \in D \)
  – Perceptron: minimize \( w \)
  – Delta Rule: minimize error \( \equiv \) distance from separator (I.e., maximize \( \frac{\partial E}{\partial \mathbf{w}} \) )
Review: Backprop, Feedforward

Intuitive Idea: Distribute Blame for Error to Previous Layers

Algorithm Train-by-Backprop \((D, r)\)
- Each training example is a pair of the form \(<x, t(x)>\), where \(x\) is the vector of input values and \(t(x)\) is the output value. \(r\) is the learning rate (e.g., 0.05)
- Initialize all weights \(w_{ij}\) to (small) random values
- UNTIL the termination condition is met, DO
  FOR each \(<x, t(x)>\) in \(D\), DO
    Input the instance \(x\) to the unit and compute the output \(o(x) = \sigma(\text{net}(x))\)
    FOR each output unit \(k\), DO
      \[\delta_k = o_k(x)(1 - o_k(x))(t_k(x) - o_k(x))\]  
    END-OUTPUT-LAYER  
    FOR each hidden unit \(j\), DO
      \[\delta_j = h_j(x)(1 - h_j(x)) \sum_{k \in \text{outputs}} v_{jk} \delta_k\]  
    END-HIDDEN-LAYER  
    Update each \(w = u_{ij} (a = h_i)\) or \(w = v_{jk} (a = o_k)\)
    \[w_{\text{start-layer, end-layer}} \leftarrow w_{\text{start-layer, end-layer}} + \Delta w_{\text{start-layer, end-layer}}\]
    \[\Delta w_{\text{start-layer, end-layer}} \leftarrow r \delta_{\text{end-layer}} a_{\text{end-layer}}\]
  END-FOR
  RETURN final \(u, v\)
Review: Derivation of Backprop

- **Recall: Gradient of Error Function**

\[ \nabla E[\bar{w}] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

- **Gradient of Sigmoid Activation Function**

\[ \frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left[ \frac{1}{2} \sum_{(\tilde{x},t(x)) \in D} (t(\tilde{x}) - o(\tilde{x}))^2 \right] = \frac{1}{2} \sum_{(\tilde{x},t(x)) \in D} \left[ \frac{\partial}{\partial w_i} (t(\tilde{x}) - o(\tilde{x}))^2 \right] \]

\[ = \frac{1}{2} \sum_{(\tilde{x},t(x)) \in D} \left[ 2(t(\tilde{x}) - o(\tilde{x})) \frac{\partial}{\partial w_i} (t(\tilde{x}) - o(\tilde{x})) \right] = \sum_{(\tilde{x},t(x)) \in D} \left[ (t(\tilde{x}) - o(\tilde{x})) \left( -\frac{\partial o(\tilde{x})}{\partial w_i} \right) \right] \]

\[ = -\sum_{(\tilde{x},t(x)) \in D} \left[ (t(\tilde{x}) - o(\tilde{x})) \frac{\partial o(\tilde{x})}{\partial \text{net}(\tilde{x})} \frac{\partial \text{net}(\tilde{x})}{\partial w_i} \right] \]

- **But We Know:**

\[ \frac{\partial o(\tilde{x})}{\partial \text{net}(\tilde{x})} = \frac{\partial o(\text{net}(\tilde{x}))}{\partial \text{net}(\tilde{x})} = o(\tilde{x})(1 - o(\tilde{x})) \]

\[ \frac{\partial \text{net}(\tilde{x})}{\partial w_i} = \frac{\partial (\bar{w} \cdot \tilde{x})}{\partial w_i} = x_i \]

- **So:**

\[ \frac{\partial E}{\partial w_i} = -\sum_{(\tilde{x},t(x)) \in D} \left[ (t(\tilde{x}) - o(\tilde{x})) \cdot o(\tilde{x})(1 - o(\tilde{x})) \cdot x_i \right] \]
Simple Genetic Algorithm (SGA)

Algorithm Simple-Genetic-Algorithm (Fitness, Fitness-Threshold, p, r, m)

// p: population size; r: replacement rate (aka generation gap width), m: string size

* P ← p random hypotheses // initialize population
* FOR each h in P DO f[h] ← Fitness(h) // evaluate Fitness: hypothesis → R
* WHILE (Max(f) < Fitness-Threshold) DO
  ⇒ 1. Select: Probabilistically select (1 - r)p members of P to add to P_s

  \[ P(h_i) = \frac{f[h_i]}{\sum_{j=1}^{p} f[h_j]} \]

  ⇒ 2. Crossover:
  ◆ Probabilistically select (r · p)/2 pairs of hypotheses from P
  ◆ FOR each pair <h_1, h_2> DO
    \[ P_s += \text{Crossover} (<h_1, h_2>) \] // P_s[t+1] = P_s[t] + <offspring_1, offspring_2>

  ⇒ 3. Mutate: Invert a randomly selected bit in m · p random members of P_s

  ⇒ 4. Update: P ← P_s

  ⇒ 5. Evaluate: FOR each h in P DO f[h] ← Fitness(h)

* RETURN the hypothesis h in P that has maximum fitness f[h]
GA-Based Inductive Learning (GABIL)

- GABIL System [Dejong et al., 1993]
  - Given: concept learning problem and examples
  - Learn: disjunctive set of propositional rules
  - Goal: results competitive with those for current decision tree learning algorithms (e.g., C4.5)

- Fitness Function: $Fitness(h) = (Correct(h))^2$

- Representation
  - Rules: IF $a_1 = T \land a_2 = F$ THEN $c = T$; IF $a_2 = T$ THEN $c = F$

- Genetic Operators
  - Want variable-length rule sets
  - Want only well-formed bit string hypotheses
**Crossover:**

**Variable-Length Bit Strings**

- **Basic Representation**
  - Start with
    
    |   |   |   |   |
    |---|---|---|---|
    | 0 | 1 | 1 | 1 |
    | a | a | c | a | a | c |
    | 1 | 0 | 1 | 1 | 0 | 0 |
    | h | h | 1 | 2 | 0 | 1 |
    | 1 | 0 | 1 | 0 | 1 | 0 |
  - Idea: allow crossover to produce variable-length offspring

- **Procedure**
  1. Choose crossover points for \( h_1 \), e.g., after bits 1, 8
  2. Now restrict crossover points in \( h_2 \) to those that produce bitstrings with well-defined semantics, e.g., \(<1, 3>, <1, 8>, <6, 8>\)

- **Example**
  1. Suppose we choose \(<1, 3>\)
  2. Result

    |   |   |   |
    |---|---|---|
    | 1 | 1 | 0 |
    | h | h | 3 |
    | 4 | 0 | 0 |
    | 01 | 1 | 1 |
    | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
GABIL EXTENSIONS

- New Genetic Operators
  - Applied probabilistically
  - 1. **AddAlternative**: generalize constraint on $a_i$ by changing a 0 to a 1
  - 2. **DropCondition**: generalize constraint on $a_i$ by changing every 0 to a 1

- New Field
  - Add fields to bit string to decide whether to allow the above operators

\[
\begin{array}{ccccccccc}
   & a_1 & a_2 & c & a_1 & a_2 & c & AA & DC \\
\hline
01 & 11 & 0 & 10 & 01 & 0 & 1 & 0 & 0
\end{array}
\]

- So now the learning strategy also evolves!

- *aka genetic wrapper*
GABIL Results

- Classification Accuracy
  - Compared to symbolic rule/tree learning methods
    - C4.5 [Quinlan, 1993]
    - ID5R
    - AQ14 [Michalski, 1986]
  - Performance of GABIL comparable
    - Average performance on a set of 12 synthetic problems: 92.1% test accuracy
    - Symbolic learning methods ranged from 91.2% to 96.6%

- Effect of Generalization Operators
  - Result above is for GABIL without AA and DC
  - Average test set accuracy on 12 synthetic problems with AA and DC: 95.2%
Problem

- How to characterize evolution of population in GA?

Goal

- Identify basic building block of GAs
- Describe family of individuals

Definition: Schema

- String containing 0, 1, * (“don’t care”)
- Typical schema: 10**0*
- Instances of above schema: 101101, 100000, …

Solution Approach

- Characterize population by number of instances representing each possible schema

\[ m(s, t) = \text{number of instances of schema } s \text{ in population at time } t \]
Selection and Building Blocks

- Restricted Case: Selection Only
  - $\bar{f}(t)$ ≡ average fitness of population at time $t$
  - $m(s, t)$ ≡ number of instances of schema $s$ in population at time $t$
  - $\hat{u}(s, t)$ ≡ average fitness of instances of schema $s$ at time $t$

- Quantities of Interest
  - Probability of selecting $h$ in one selection step
    \[ P(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)} \]
  - Probability of selecting an instance of $s$ in one selection step
    \[ P(h \in s) = \sum_{h \in (s \cap p_i)} \frac{f(h)}{n \cdot f(t)} \cdot \hat{u}(s, t) \cdot m(s, t) \]
  - Expected number of instances of $s$ after $n$ selections
    \[ E[m(s, t+1)] = \frac{\hat{u}(s, t)}{\bar{f}(t)} \cdot m(s, t) \]
### Schema Theorem

- **Theorem**

\[
E[m(s, t + 1)] \geq \frac{\hat{u}(s, t)}{f(t)} \cdot m(s, t) \cdot \left(1 - p_c \frac{d_s}{l-1}\right) \cdot (1 - p_m)^{o(s)}
\]

- **m(s, t)** $\equiv$ number of instances of schema $s$ in population at time $t$
- **$f(t)$** $\equiv$ average fitness of population at time $t$
- **$\hat{u}(s, t)$** $\equiv$ average fitness of instances of schema $s$ at time $t$
- **$p_c$** $\equiv$ probability of single point crossover operator
- **$p_m$** $\equiv$ probability of mutation operator
- **$l$** $\equiv$ length of individual bit strings
- **$o(s)$** $\equiv$ number of defined (non "*") bits in $s$
- **$d(s)$** $\equiv$ distance between rightmost, leftmost defined bits in $s$

- **Intuitive Meaning**

"The expected number of instances of a schema in the population tends toward its relative fitness"

A fundamental theorem of GA analysis and design
**Terminology**

- **Evolutionary Computation (EC): Models Based on Natural Selection**
- **Genetic Algorithm (GA) Concepts**
  - **Individual**: single entity of model (corresponds to hypothesis)
  - **Population**: collection of entities in competition for survival
  - **Generation**: single application of selection and crossover operations
  - **Schema aka building block**: descriptor of GA population (e.g., 10**0*)
  - **Schema theorem**: representation of schema proportional to its relative fitness
- **Simple Genetic Algorithm (SGA) Steps**
  - **Selection**
    - Proportionate reproduction (*aka* roulette wheel): \( P(\text{individual}) \propto f(\text{individual}) \)
    - Tournament: let individuals compete in pairs or tuples; eliminate unfit ones
  - **Crossover**
    - Single-point: 11101001000 \( \times \) 00001010101 \( \rightarrow \) \{ 11101010101, 00001001000 \}
    - Two-point: 11101001000 \( \times \) 00001010101 \( \rightarrow \) \{ 11001011000, 00101000101 \}
    - Uniform: 11101001000 \( \times \) 00001010101 \( \rightarrow \) \{ 10001000100, 01101011001 \}
  - **Mutation**: single-point (“bit flip”) multi-point
Lecture Outline

- **Readings / Viewings**
  - View GP videos 1-3
    - GP1 – *Genetic Programming: The Video*
    - GP2 – *Genetic Programming: The Next Generation*
    - GP3 – *Genetic Programming: Human-Competitive*
  - Suggested: Chapters 1-5, Koza

- **Previously**
  - Genetic and evolutionary computation (GEC)
  - Generational vs. steady-state GAs; relation to simulated annealing, MCMC
  - Schema theory and GA engineering overview

- **Today: GP Discussions**
  - Code bloat and potential mitigants: types, OOP, parsimony, optimization, reuse
  - Genetic programming vs. human programming: similarities, differences

- **Thursday: Course Review**
Flowchart for Genetic Programming

- **Bin = 0**
  - Create Initial Random Population
  - **Termination Criteria Reached?**
    - **Yes**
      - **Stop**
    - **No**
      - Execute Fitness of Each Individual in Population
        - **Indiv. = 0**
          - **Bin = Bin + 1**
        - **Indiv. = Indiv. + 1**
  - **Select Genetic Operation Probabilistically**
    - **Selection**
      - Select One Individual Based on Fitness
      - Perform Reproduction
      - Copy Into New Population
      - **Indiv. = Indiv. + 1**
    - **Mutation**
      - Select One Individual Based on Fitness
      - Perform Mutation
      - Insert Mutant Into New Population
      - **Indiv. = Indiv. + 2**
    - **Crossover**
      - Select Two Individuals Based on Fitness
      - Perform Crossover
      - Insert Two Offspring Into New Population
      - **Indiv. = Indiv. + 1**

Adapted from The Genetic Programming Notebook © 2002 Jaime J. Fernandez
http://www.geneticprogramming.com
Structural Crossover

Crossover Operation with Different Parents

Parents

Children

Crossover Operation with Identical Parents

Parents

Children

Adapted from The Genetic Programming Notebook © 2002 Jaime J. Fernandez
http://www.geneticprogramming.com
Structural Mutation

Mutation

Original Individual

Mutated Individuals
Genetic Programming: The Next Generation
(Synopsis and Discussion)

- Automatically-Defined Functions (ADFs)
  - *aka* macros, anonymous inline functions, subroutines
  - Basic method of software reuse

- Questions for Discussion
  - *What are advantages, disadvantages of learning anonymous functions?*
  - *How are GP ADFs similar to and different from human-produced functions?*

- Exploiting Advantages
  - Reuse
  - Innovation

- Mitigating Disadvantages
  - Potential lack of meaning – semantic clarity issue (and topic of debate)
  - Redundancy

Accelerated bloat – scalability issue
Code Bloat [1]:
Problem Definition

- **Definition**
  - Increase in program size not commensurate with increase in functionality (possibly as function of problem size)
  - Compare: structural criteria for overfitting, overtraining

- **Scalability Issue**
  - Large GPs will have this problem
  - Discussion: *When do we expect large GPs?*
  - Machine learning: large, complex data sets
  - Optimization, control, decision making / DSS: complex problem

- **What Does It Look Like?**

- **What Can We Do About It?**
  - ADFs
  - Advanced reuse techniques from software engineering: e.g., design patterns
  - Functional, object-oriented design; theory of types
  - Controlling size: parsimony (MDL-like), optimization (cf. compiler)
Code Bloat [2]: Mitigants

- Automatically Defined Functions
- Types
  - Ensure
    - Compatibility of functions created
    - Soundness of functions themselves
  - Define: abstract data types (ADTs) – object-oriented programming
  - Behavioral subtyping – still “future work” in GP
  - Generics (cf. C++ templates)
  - Polymorphism
- Advanced Reuse Techniques
  - Design patterns
  - Workflow models
  - Inheritance, reusable classes
Code Bloat [3]: More Mitigants

- Parsimony (cf. Minimum Description Length)
  - Penalize code bloat
  - Inverse fitness = loss + cost of code (evaluation)
  - May include terminals

- Target Language Optimization
  - Rewriting of constants
  - Memoization
  - Loop unrolling
  - Loop-invariant code motion
16 Criteria for Automatic Program Synthesis by Computational Intelligence

1. **Specification**: starts with *what needs to be done*
2. **Procedural representation**: tells us how to do it
3. **Algorithm implementation**: produces a computer program
4. **Automatic determination of program size**
5. **Code reuse**
6. **Parametric reuse**
7. **Internal storage**
8. **Iteration** (while / for), recursion
9. **Self-organization of hierarchies**
10. **Automatic determination of architecture**
11. **Wide range of programming constructs**
12. **Well-defined**
13. **Problem independent**
Genetic Programming 3
(Synopsis and Discussion [2])

- 16 Criteria for Automatic Program Synthesis …
  - 14. Generalization: wide applicability
  - 15. Scalability
  - 16. Human-competitiveness

- Current Bugbears: Generalization, Scalability

- Discussion: Human Competitiveness?
More Food for Thought and Research Resources

- Discussion: Future of GP
- Current Applications
- Conferences
  - GECCO: ICGA + ICEC + GP
  - GEC
  - EuroGP
- Journals
  - Evolutionary Computation Journal (ECJ)
  - Genetic Programming and Evolvable Machines (GPEM)