🐜 Ant Colony Optimization (ACO)

What is ACO?

Ant Colony Optimization (ACO) is a **bio-inspired metaheuristic** based on the **foraging behavior of ants**. Real ants deposit pheromones to mark favorable paths to food, and over time, the strongest pheromone trails guide other ants to the optimal path. ACO translates this idea into a computational algorithm used to solve combinatorial optimization problems such as the **Traveling Salesman Problem**, **feature selection**, and **routing**.

How ACO Works

In the context of **feature selection**:

1. **Ants = candidate solutions**: Each ant selects a subset of features from the available pool.
2. **Pheromones = memory of good solutions**: Features that appear in better subsets are reinforced with higher pheromone values.
3. **Heuristic = local guidance**: Features that are individually good (based on mutual information) are favored.
4. **Probabilistic Selection**: Ants choose features based on the combination of pheromone levels and heuristic information.
5. **Evaporation**: Pheromones decay over time to avoid convergence to a local optimum.

🏋️ Training Process

1. **Initialize**

- Calculate **mutual information (MI)** for each feature.
- Initialize **pheromones** randomly between 0.1 and 1.0.
- Normalize MI scores and use them as **heuristic values**.

2. **Iterations**

For a specified number of iterations (max_iter):

- For each ant:
  - Select a subset of features probabilistically using the select_features() function.
  - Evaluate the subset using a **train-test split** and an ML model (model.fit()).

    - Save the best feature subset found so far.
- Update **pheromones** based on the fitness (accuracy) of the ants' solutions.
- Apply **evaporation** to prevent unlimited accumulation.

3. **Final Output**

- Return the **best-performing feature subset** and its corresponding **accuracy score**.

Parameters and Their Roles

Parameter	Role
columns	List of all feature names.
max iter	Number of algorithm iterations (generations).
num ants	Number of ants (solutions) per iteration.
num features	Number of features to select per ant (subset size).
alpha	Importance of **pheromone** in selection (0 = ignore pheromone).
beta	Importance of **heuristic (MI)** in selection.
evaporation rate	Controls how quickly old pheromones fade (prevents overfitting).

⚠ Limitations & Drawbacks of ACO

1. **Computationally Expensive**:
    - Evaluating many feature subsets per iteration is costly, especially with large datasets and high-dimensional features.
2. **Parameter Sensitivity**:
    - The performance of ACO is **highly dependent** on the choice of hyperparameters (alpha, beta, evaporation_rate, etc.).
3. **Stochastic Nature**:
    - Results can vary between runs; not always guaranteed to converge to a global optimum.

4. **Overfitting Risk**:
    - Without proper regularization or validation, ACO can overfit the feature selection to the training set.
5. **Slow Convergence**:
    - It might take many iterations to converge to a good solution, especially in a large search space.

🎵 Harmony Search (HS)

What is Harmony Search?

Harmony Search (HS) is a music-inspired metaheuristic algorithm that mimics the improvisation process of musicians. Just as musicians try different note combinations to find the best harmony, HS tries different combinations of variables (in this case, feature subsets) to find the optimal solution.

HS is particularly suitable for combinatorial optimization problems, such as feature selection and dimensionality reduction, due to its balance between exploitation (using existing knowledge) and exploration (trying new solutions).

How it Works:

Music Concept	Optimization Equivalent
Musician	Decision variable (feature index)
Note	Value (feature selected)
Harmony Memory (HM)	Set of existing solutions
Harmony Memory Size	Number of stored solutions (harmonies)
Harmony Memory Considering Rate (HMCR)	Probability of choosing from memory
Pitch Adjustment Rate (PAR)	Probability of tweaking a chosen feature
Improvisation	Generation of a new candidate solution

T**raining Process in Your Code**

1. **Data Preparation**

- X: Feature matrix (21 binary features).
- y: Target vector (binary classification: 0 = no diabetes, 1 = diabetes).

2. **Initialize Harmony Memory**

- Generate HMS number of random feature subsets, each of target_dimension size.
- Store these initial harmonies in a list.

3. **Evaluate Fitness**

- For each harmony (subset of features), calculate fitness using **logistic regression accuracy** on a validation set (via train_test_split).

4. **Iterative Search**

For max_iter iterations:

- Generate a **new harmony**:
  - For each feature in the subset:
    - With probability HMCR, choose from existing memory.
      - With probability PAR, apply **pitch adjustment** (tweak index ±1).
    - Otherwise, select a new random feature.
- Ensure the new harmony contains unique features.
- Evaluate its fitness.
- **Replace the worst-performing harmony** in memory if the new one performs better.

5. **Return Best Harmony**

- After all iterations, return the feature subset with the highest validation accuracy.

Parameters and Their Roles

Parameter	Description
HMS	Number of solutions (harmonies) stored in memory.
HMCR	Probability of selecting features from memory rather than randomly. High HMCR encourages exploitation.
PAR	Probability of adjusting (mutating) a selected feature. Introduces diversity.
target_dimension (e.g., 2 or 3)	Number of features to select. Controls dimensionality reduction level.
max_iter	Number of iterations (improvisations). Controls convergence time and quality.

⚠ Limitations & Drawbacks of HS

1. **Slow Convergence**:
   - HS may require a large number of iterations to find high-quality solutions.

2. **Sensitivity to Parameters**:
   - Like most metaheuristics, performance is highly dependent on tuning HMS, HMCR, and PAR.
3. **Stochastic Behavior**:
   - Results can vary between runs due to randomness in feature selection and mutation.
4. **Scalability**:
   - Feature space grows exponentially with input size, making it slower with high-dimensional data.
5. **Simple Adjustment Mechanism**:
   - Pitch adjustment (±1) is basic and may not be ideal for more complex problems or structured features.

Genetic Algorithm (GA) for Feature Selection

The goal of this Genetic Algorithm is to identify the most relevant subset of features from a dataset to improve classification accuracy while reducing dimensionality. Logistic Regression is used as the classifier to evaluate the fitness of each feature subset.

Overview of Genetic Algorithm (GA)

A Genetic Algorithm is a metaheuristic inspired by natural selection that evolves a population of candidate solutions (feature subsets) over generations to optimize a specific objective – in this case, classification accuracy using a fixed number of features.

Individual Representation

- Each individual (chromosome) is a **binary array** of size equal to the total number of features.
- 1 indicates a selected feature, 0 indicates an unselected feature.
- A constraint is enforced: **each individual selects exactly *n* features** to maintain consistency in feature subset size.

Fitness Function

The fitness of an individual is the **classification accuracy** on a test set using the selected features. We use Logistic Regression.

Genetic Operators

Initialization

- The initial population is randomly generated.
- Each chromosome is initialized with exactly n_selected features set to 1, and the rest set to 0.

Crossover Techniques

Crossover combines two parent individuals to produce a child, allowing for the exchange of genetic material (features).

1. **One-Point Crossover**

- A single crossover point is randomly selected.
- The child inherits genes from parent 1 up to that point, and from parent 2 afterward.

2. **Two-Point Crossover**

- Two crossover points are chosen randomly.
- The child takes the first segment from parent 1, middle from parent 2, and last from parent 1 again.

3. **Uniform Crossover**

- Each gene is randomly chosen from either parent with equal probability (50%).
- Offers more gene-level mixing than point-based methods.

Post-crossover, **feature count correction** is enforced to ensure exactly n_selected features are active.

Mutation Techniques

Mutation introduces diversity and prevents premature convergence.

1. **Flip Bit Mutation**

- A randomly selected gene is flipped from 0→1 or 1→0.

2. **Random Reset Mutation**

- A gene is randomly selected and assigned a new binary value (0 or 1).

3. **Swap Mutation**

- Swaps a 1 and a 0 position, preserving the total number of selected features.

All mutation techniques invoke the feature count correction method to maintain a fixed number of selected features.

Selection Technique

Stochastic Universal Sampling (SUS)

Stochastic Universal Sampling (SUS) is a probabilistic selection method that ensures **a more evenly distributed and less biased selection** process compared to standard Roulette Wheel Selection. It is particularly useful in Genetic Algorithms to **maintain population diversity** and **prevent premature convergence**.

Core Idea

Instead of selecting individuals one-by-one (as in roulette selection), SUS places **multiple evenly spaced selection points** over the **fitness proportionate distribution** (i.e., a virtual "roulette wheel") and selects all individuals at once.

This helps avoid the issue where high-fitness individuals dominate the selection, while low-fitness individuals never get selected – a common problem in roulette selection.

How It Works – Step-by-Step

1. **Calculate Total Fitness**
   Compute the sum of all individual fitnesses:

$$F = \sum_{i=1}^{N} f_i$$

2. Normalize Fitness Scores

Compute the selection probability for each individual:

$$p_i = \frac{f_i}{F}$$

3. Calculate Distance Between Pointers

Let n be the number of individuals to select (usually equal to pop_size or half if you're selecting parents for crossover).

Set:

$$\text{distance} = \frac{1}{n}$$

4.Generate Start Point

Randomly choose a start point:

start=random.uniform(0,distance)

5.Place Pointers and Select

Place n pointers:

$$\text{pointers} = [\text{start} + i \times \text{distance for } i \text{ in } 0 \text{ to } n-1]$$

Then walk through the cumulative fitness distribution and pick the individuals where the pointer lands.

Tournament Selection

Core Idea

Tournament Selection is a widely used selection strategy in Genetic Algorithms. It selects individuals by holding "tournaments" among a few randomly chosen candidates from the population and choosing the best (or sometimes with a small probability, a worse) individual from each group.

This approach mimics competitive survival, similar to a real-world tournament — the best of each group advances to reproduction.

How It Works – Step-by-Step

1. **Set Tournament Size (k)**
   Choose k, the number of individuals to participate in each tournament. A typical choice is k = 2 or k = 3.
2. **Repeat for Each Selection Needed**
   - Randomly pick k individuals from the population (with or without replacement).
   - Evaluate their fitness scores.
   - Select the **individual with the highest fitness** from this subset to proceed.
3. **Repeat Until Enough Individuals Are Selected**
   Keep running tournaments until the desired number of individuals is selected (usually equal to the population size or parent pool size).

Roulette Wheel Selection (Fitness Proportionate Selection)

Core idea

Roulette Wheel Selection is inspired by a casino roulette wheel where each slice represents an individual in the population. The size of each slice is **proportional to the individual's fitness** — higher fitness means a bigger slice and higher chance of being selected.

Feature / Criterion	**Stochastic Universal Sampling (SUS)**	**Tournament Selection**	**Roulette Wheel Selection**
Selection Basis	Fitness proportionate (uses multiple evenly spaced pointers)	Best among k randomly chosen candidates	Fitness proportionate (probabilistic spin)
Control Parameter	None (selection proportional to fitness)	Tournament size k	None
Selection Pressure	Balanced – ensures diverse sampling of entire population	Adjustable – higher k increases pressure	Medium – proportional to fitness
Fairness	High – more consistent and deterministic selection	Moderate – stochastic but tunable via k	Low – random variation can cause strong individuals to dominate
Randomness Level	Low randomness (less bias, evenly spaced pointers)	Medium randomness (depends on participant draw)	High randomness (biased by chance and fitness scaling)
Diversity Maintenance	Good – more individuals get selected (esp. mid-fitness ones)	Good (if k is small) – weak individuals still have a chance	Poor – weak individuals rarely get picked
Speed	Fast –	Very fast –	Fast for small

	deterministic and parallelizable	simple comparison among few individuals	populations, slower for large ones
Scalability	Scales well with large populations	Scales excellently – independent of population size	Less scalable – fitness scaling becomes tricky with large populations
Fitness Scaling Sensitivity	Less sensitive (handles outliers better)	Not sensitive (relative rankings matter, not absolute fitness values)	Highly sensitive (outliers may dominate)
Implementation Complexity	Moderate – requires cumulative fitness and even pointer spacing	Simple – just random sampling and comparison	Simple – fitness normalization and cumulative selection
Determinism	More deterministic – reduces variance in selection	Stochastic but controllable	Highly stochastic
Best Use Case	When balanced exploration and exploitation are needed	When tuning selection pressure is important	When population is small and fitness is well-scaled
Drawbacks	Slightly more complex than Tournament or Roulette	May lose diversity if k is too large	Risk of premature convergence, sensitive to skewed fitness