Table of Contents

AOA PROJECT

What is an Optimization Algorithm?

An **optimization algorithm** is a method or procedure used to find the **best possible solution** (maximum or minimum) to a problem from a set of feasible solutions, under given constraints.

- (*An optimization algorithm helps us adjust the parameters of a system or model so that it performs as well as possible according to some measure (objective function)*)

Key Components of Optimization:

1. **Objective Function (Cost/Loss):**
    - A mathematical function that measures the "quality" of a solution.
    - Example: In Logistic Regression, the log-loss function measures how wrong predictions are.
2. **Decision Variables (Parameters):**
    - The values we can adjust to improve performance.
    - Example: Model weights in machine learning.
3. **Constraints:**
    - Conditions that restrict the solution space.
    - Example: Resource limits, or parameters must be non-negative.

Types of Optimization Algorithms:

1. **Classical (Deterministic):** Usually efficient, good for convex problems.
    - Gradient Descent (GD), Newton's Method, etc.
2. **Metaheuristic (Stochastic):** Good for complex, non-convex, NP-hard problems
    - Genetic Algorithm (GA), Simulated Annealing (SA), Particle Swarm Optimization (PSO).

Why are they important in ML?

In Machine Learning, training a model = **optimization problem**.

- We want to find the best parameters (weights) that minimize the **loss function**.
- Different optimization algorithms affect training speed, accuracy, and ability to avoid local minima.
- **Gradient Descent:** Fast and efficient for convex problems.
- **GA / PSO / SA:** Explore more widely, can escape local minima, useful for tough problems.

Convex Function

A function is convex if a line segment between any two points on the curve lies above or on the curve.

Has **one global minimum** (no false/local traps).

Optimization is easy and guarantees finding the **global minimum**.

- Intuition: Looks like a U-shaped bowl (e.g., quadratic function).

Why it's good:

- Algorithms like **Gradient Descent** are guaranteed to find the *global minimum*.
- Optimization is **fast and reliable**.
- **Convex problems (easy, reliable):** Logistic Regression, Support Vector Machines (SVM with linear kernel), Linear Regression

Use **Gradient Descent** → fast convergence, global minimum guaranteed.

Non-Convex Function

A function is **non-convex** if it has **multiple local minima** or maxima.

The line segment between two points can fall **below** the graph.

Has **multiple local minima/maxima**, making optimization harder.

- **Intuition:** Looks like a **valley with many hills** (roller coaster shape).
- **Examples:** Neural networks' loss surface, Rastrigin function.

Why it's hard:

- Gradient Descent can get **stuck in local minima**.
- No guarantee of finding the true global minimum.
- Need **metaheuristic algorithms** (GA, PSO, SA) that explore the search space globally.
- **Non-Convex problems (hard, tricky):** Deep Neural Networks (DNNs), Clustering (like k-means), Feature selection problems

Need **global search** → GA, PSO, SA to avoid local traps

Table 1: Convex Optimization Challenges

Feature	Problem	Solution
Convergence Speed	GD may converge slowly in shallow slopes	Use adaptive learning rates (Adam, RMSProp) or momentum
Large Datasets	Full-batch GD is computationally expensive	Use Stochastic Gradient Descent (SGD) or mini-batch GD
Feature Scaling	Poorly scaled features cause uneven gradients	Normalize or standardize features
Steep/Ill-conditioned Curvature	Gradients may oscillate or overshoot	Use second-order methods like Newton-Raphson or quasi-Newton (BFGS)
Hyperparameter Sensitivity	Learning rate too high/low affects convergence	Tune learning rate carefully, use line search or decay

Table 2: Non-Convex Optimization Challenges

Feature	Problem	Solution
Local Minima	Optimization may get stuck in local minima	Use metaheuristics (GA, PSO, SA) and multiple random initializations
Saddle Points / Plateaus	Flat regions slow down convergence	Add noise to gradients, use momentum or adaptive learning rates
Initialization Sensitivity	Final solution depends on starting point	Run optimization multiple times with different seeds, pick best solution
Exploration vs Exploitation	Algorithm may not balance global/local search	Tune parameters: GA → mutation/crossover rate, PSO → inertia/cognitive/social factors, SA → temperature schedule
Computational Cost	Global search is expensive	Use smaller population/iterations, parallelize evaluations, hybrid methods (e.g., GA + GD)

Metaheuristic Algorithms

Definition:

Metaheuristic algorithms are high-level optimization techniques designed to find near-optimal solutions for complex problems where exact methods are too slow or infeasible (such as NP-hard problems like the Traveling Salesman Problem). They are problem-independent, meaning they can be applied to a wide range of optimization tasks without requiring major modifications.

Key Characteristics:

1. **Approximate Solutions:** They do not guarantee the global optimum but aim for good (near-optimal) solutions within reasonable time.
2. **Exploration vs. Exploitation:** Metaheuristics balance **exploration** (searching broadly across the solution space) and **exploitation** (refining the best solutions found so far).
3. **Stochastic Nature:** Most use randomness (e.g., random mutation in GA, probabilistic acceptance in SA), allowing them to escape local minima.
4. **Applicability to Hard Problems:** They are widely used for NP-hard problems (e.g., TSP, job scheduling, resource allocation) where exact algorithms are impractical.

Examples of Metaheuristic Algorithms:

- **Genetic Algorithm (GA):** Inspired by natural selection; uses crossover and mutation to evolve solutions.
- **Simulated Annealing (SA):** Inspired by the annealing process in metallurgy; uses probabilistic jumps to escape local minima.
- **Particle Swarm Optimization (PSO):** Inspired by bird flocking/fish schooling; particles share information to move toward optimal solutions

Why use Logistic Regression?

Logistic Regression (LR) is ideal for comparing optimization algorithms because:

1. **Convex loss function** – ensures Gradient Descent reliably reaches the global minimum.
2. **Few hyperparameters** – fast to optimize, allowing fair comparison of GD, GA, SA, and PSO.
3. **Fast training** – multiple iterations possible without high computational cost.

4. **Meaningful hyperparameter (C)** – directly affects model performance (underfitting vs overfitting).
5. **Supports both deterministic and stochastic optimization** – allows clear evaluation of algorithm efficiency, speed, and robustness.

Implement Classical Optimization Methods to ML model

Gradient Descent (GD) – (Deterministic)

Definition:

Gradient Descent is an **iterative optimization algorithm** that updates parameters in the direction of the negative gradient of the objective function until convergence.

(minimize a cost (or loss) function)

Deterministic:

- For a given starting point and learning rate, GD always produces the **same path** and the same result.
- No randomness involved.

Time Complexity:

O(k·d)

Where:

- k = number of iterations until convergence
- d = number of parameters (dimensions)

Why? Because at each step, GD computes the gradient (which costs O(d)) and repeats it k times.

Example:

- For linear regression with d features, computing gradient = O(d), repeated over k iterations.
- Finding the best slope for a line in linear regression.

Real-world Case Study:

- **Deep Learning Training**: All neural networks are trained using GD (or its variants like SGD, Adam). For example, training **ChatGPT** itself relies on gradient-based optimization on billions of parameters.

CODE FOR GRADIENT DESCENT USING LOGISTIC REGRESSION FOR STUDENT PERFORMANCE DATASET:

```
def gradient_descent(obj, x0, lr=0.1, steps=30):
  x = np.array(x0, dtype=float)
  for _ in range(steps):
    grad = np.zeros_like(x)
    fx = obj(x)
```

```
        eps = 1e-4
        for i in range(len(x)):
            xx = x.copy()
            xx[i] += eps
            grad[i] = (obj(xx) - fx) / eps
        x -= lr * grad
        x = np.clip(x, [0.01], [10])  # keep C in range
    return x, -obj(x)
```

```
Gradient Descent:
C = [1.]
Accuracy Score = 98.87%
```

This algorithm tries to find the best value of **C (regularization strength)** for Logistic Regression.
It calculates the gradient (approx using finite difference) and updates C step by step.

LOGIC:

1. Start with an **initial guess** for parameter C.
2. Compute the **objective value** (negative accuracy).
3. Estimate the **gradient** numerically using finite differences.
4. Take a **step in the opposite direction** of the gradient to reduce the objective.
5. Repeat for multiple iterations until improvement stabilizes.
6. Ensure parameter stays in a valid range.
7. Return the parameter value that gave the best score (highest accuracy).

Simulated Annealing (SA)

Logic in ML:

Instead of strictly following the gradient, SA **explores the parameter space** by accepting not only better solutions but sometimes worse ones, depending on a "temperature" that cools over time. This helps escape **local minima**.

- In optimization, SA is used to **find a near-global minimum (or maximum)** of a function.
- It is especially useful for **non-convex or combinatorial problems** with multiple local minima.

Optimization view:

Inspired by **metal cooling** process. Early stage = more exploration; later stage = fine-tuning.
It is **probabilistic** and works even for **non-convex** functions.

Time complexity:

Time Complexity=**O(steps·k·m·d)**

Where, **m** = number of samples, **d** = number of features, **k** = cv folds.
Each iteration: O(1), but usually requires **more iterations** than GD to converge.
Slower than GD in practice.

Example: Simulated Annealing finds the global minimum of a non-convex function by exploring and probabilistically accepting worse solutions.
Case Study: SA efficiently solves Job Shop Scheduling by escaping local minima to find near-optimal task sequences for minimal completion time.

CODE:

```
def simulated_annealing(obj, x0, steps=100, T0=1.0, alpha=0.95):
    x = np.array(x0, dtype=float)
    fx = obj(x)
    best, best_f = x.copy(), fx
    T = T0
    for _ in range(steps):
        xn = x + np.random.normal(0, 0.5, size=len(x))
        xn = np.clip(xn, [0.01], [10])
        fn = obj(xn)
        if fn < fx or np.random.rand() < np.exp(-(fn-fx)/T):
            x, fx = xn, fn
        if fn < best_f:
            best, best_f = xn, fn
        T *= alpha
    return best, -best_f
```

```
Simulated Annealing:
C = [5.]
Accuracy Score = 98.87%
```

LOGIC:

1. Start with an initial solution.
2. Explore neighbouring solutions randomly.
3. Accept better solutions always; worse solutions probabilistically.
4. Keep track of the best solution.
5. Gradually reduce “temperature” to shift from exploration → exploitation.
6. Return the best solution found.

Genetic Algorithm (GA)

Logic in ML:

Genetic Algorithm is an optimization technique inspired by natural selection and genetics. It evolves a population of candidate solutions using selection, crossover, and mutation to gradually improve fitness.

GA treats solutions (like regularization parameter CCC) as "chromosomes".

It evolves them using **selection, crossover, and mutation**.

Works well when the search space is large & discontinuous.

Optimization view:

It's a **global search** algorithm, not gradient-based, so it works for **non-differentiable** problems.

Often used in **feature selection, hyperparameter tuning**.

Time complexity:

O(P×G×F)

- P = population size
- G = number of generations
- F = cost of fitness evaluation

Much higher than GD, but more robust for complex problems.

Example:

Tuning hyperparameters of a neural network where GA evolves learning rates and architecture choices.

Case Study:

GA has been used in **aircraft design optimization**, where multiple design variables (wing shape, material choice) were evolved to maximize fuel efficiency and performance.

CODE:

```
def genetic_algorithm(obj, pop_size=10, gens=10):
    pop = np.random.uniform([0.01], [10], size=(pop_size, 1))
    fitness = np.array([obj(ind) for ind in pop])

    for _ in range(gens):
        parents = []
        for _ in range(pop_size):
            i, j = np.random.randint(0, pop_size, 2)
            parents.append(pop[i] if fitness[i] < fitness[j] else pop[j])
        parents = np.array(parents)

        children = parents + np.random.normal(0, 0.2, size=parents.shape)
        children = np.clip(children, [0.01], [10])

        child_fit = np.array([obj(c) for c in children])
```

```
    combined = np.vstack([pop, children])
    comb_fit = np.hstack([fitness, child_fit])

    idx = np.argsort(comb_fit)[:pop_size]
    pop, fitness = combined[idx], comb_fit[idx]
return pop[0], -fitness[0]
```

```
Genetic Algorithm:
C = [6.89388682]
Accuracy Score = 98.87%
```

Logic:

1. **Initialization** – Start with a random population of solutions (chromosomes).
2. **Fitness Evaluation** – Measure how good each solution is using an objective function.
3. **Selection** – Choose the best-performing individuals to reproduce.
4. **Crossover (Recombination)** – Combine parts of two parent solutions to create offspring.
5. **Mutation** – Randomly alter some parts of a solution to maintain diversity.
6. **Replacement** – Form a new population from the offspring and repeat.
7. **Termination** – Stop when a maximum generation count or satisfactory solution is reached.

COMPARE GD, SA, GA

Gradient Descent is highly efficient for the **student performance dataset** because Logistic Regression's loss function is convex. Each update step has a time complexity of **O(n·d)**, where *n* is the number of students and *d* is the number of academic or behavioral features (such as study hours, attendance, and test scores). Since the dataset has limited features and the cost function is convex, GD converges quickly to the global optimum, making it computationally faster and more reliable than stochastic methods.

In contrast, metaheuristic algorithms like **Simulated Annealing (SA) and Genetic Algorithm (GA)** are useful for non-convex or highly complex problems but are computationally more expensive. They require exploring large solution spaces, maintaining populations, and running many iterations, which increases runtime. For predicting student performance, where the problem is convex and differentiable, GD guarantees faster convergence and optimal results, making it the most practical and time-efficient choice compared to metaheuristics.

Gradient Descent remains the best optimization algorithm in terms of efficiency, scalability, and practicality for large datasets and time-critical applications.
In terms of **time complexity**, GD typically runs in O(n·d·k), where n is the number of samples, d the number of features, and k the number of iterations required for convergence. Importantly, modern optimizers (Adam, RMSProp, etc.) further accelerate convergence by adapting the learning rate,

which means fewer iterations are needed even in very high-dimensional problems. This makes GD the **de-facto standard for training deep learning models** with millions of parameters—something metaheuristics like SA, GA cannot scale to efficiently.

Traveling Salesman Problem (TSP)

It is a classic optimization problem in computer science and operations research:
A salesman must visit a set of **cities exactly once**, return to the starting city, and **minimize the total travel distance (or cost)**.

1. C**lassic NP-Hard Problem:** There is **no known polynomial-time algorithm** to solve TSP exactly.
2. **Applications:**
    - Logistics and delivery route optimization
    - Circuit board manufacturing (minimizing wire length)
    - DNA sequencing in bioinformatics
    - Vehicle routing problems

Simulated Annealing (SA)

Simulated Annealing (SA) is an **optimization algorithm** used to find a **good (near-optimal) solution** for problems like the Traveling Salesman Problem (TSP).

- In TSP, SA **tries to find the shortest path** that visits all cities once and returns to the start.
- It **does not guarantee the absolute shortest path**, but it usually finds a **very good solution quickly**.
- It can **accept worse solutions temporarily** to escape local minima.
- Over time, it focuses on **refining the best solution found**.

STEPS:

1. **Start with a random solution**
    - For TSP, this is a random tour visiting all cities.
2. **Set an initial "temperature"**
    - High temperature → more willing to explore worse solutions.
    - Low temperature → more focused on improving the current solution.
3. **Generate a neighbor solution**
    - For TSP: swap or reverse a segment of the path to create a new candidate tour.
4. **Decide whether to accept the new solution**
    - **If better:** always accept.
    - **If worse:** accept with a probability that depends on the temperature:

    This allows escaping **local minima** early in the process.
5. **Cool down the temperature**

- Multiply the temperature by a **cooling factor (<1)**.
- As temperature decreases, the algorithm becomes less likely to accept worse solutions.

6. **Repeat for a number of iterations**
   - Track the **best solution found**.
   - Eventually, the temperature is low, and the algorithm converges.
7. **Return the best solution**
   - The shortest path found during the process is returned.

Time complexity per iteration: O(n)
Total time complexity: O(n·iterations)

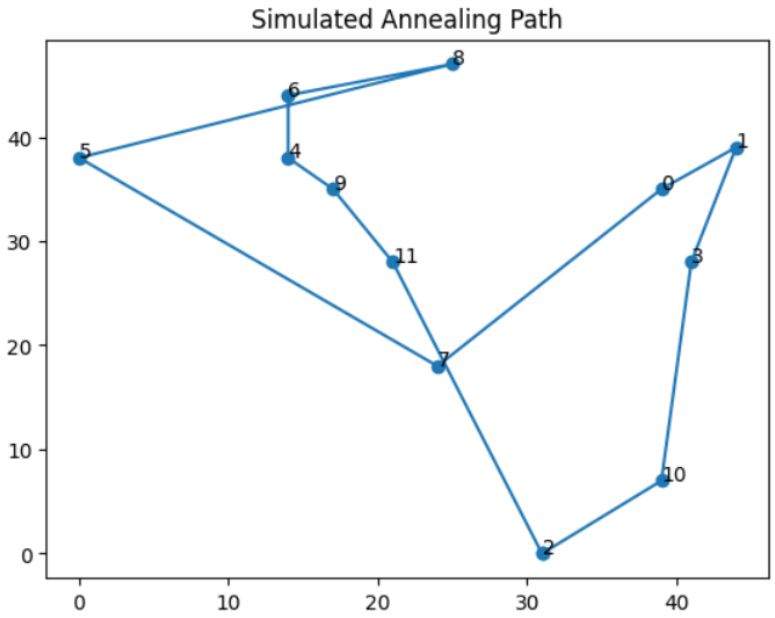
Key Idea:

- Simulated Annealing balances **exploration** (accepting worse paths early) and **exploitation** (refining best path later).
- It's effective for **combinatorial optimization problems like TSP** where local minima exist.

CODE:

```
def simulated_annealing(dist_matrix, temp=1000, alpha=0.995, iterations=1000):
    n = len(dist_matrix)
    current_path = list(range(n))
    random.shuffle(current_path)
    best_path = current_path[:]
    best_dist = total_distance(best_path, dist_matrix)
    convergence = []

    for i in range(iterations):
        a, b = random.sample(range(n), 2)
        new_path = current_path[:]
        new_path[a:b] = reversed(new_path[a:b])
        new_dist = total_distance(new_path, dist_matrix)
        if new_dist < best_dist or random.random() < math.exp((best_dist - new_dist) / temp):
            current_path = new_path
            if new_dist < best_dist:
                best_path = new_path
                best_dist = new_dist
        temp *= alpha
        convergence.append(best_dist)
    return best_path, best_dist, convergence
```

```
========== Results ==========

Simulated Annealing Path: [2, 11, 9, 4, 6, 8, 5, 7, 0, 1, 3, 10]
 Distance: 189.45
 Time: 0.015345 secconds
```



Genetic Algorithm

Genetic Algorithm (GA) is a **population-based optimization algorithm** inspired by **natural evolution**.

- In TSP, GA **tries to find the shortest tour** visiting all cities.
- GA works with a **population of solutions** rather than a single path.
- Over generations, it evolves the population to **improve solution quality**.

Key idea: "Survival of the fittest" → better paths are more likely to reproduce.

STEPS:

Start with a random population

- Create multiple random tours (paths) of all cities.
- Example: if population size = 10, generate 10 random paths.

Evaluate fitness

- Calculate **fitness** for each path:

fitness=1total distance\text{fitness} = \frac{1}{\text{total distance}}fitness=total distance1

- Shorter paths → higher fitness → more likely to be selected as parents.

Selection of parents

- Choose the best paths based on fitness to **produce the next generation**.
- Keep top solutions (elitism) to preserve good paths.

Crossover (Recombination)

- Combine two parent paths to create a child path:
  - Take first part of parent1.
  - Fill remaining cities in the order they appear in parent2.
- Purpose: mix characteristics of parents to explore new solutions.

Mutation

- Randomly swap two cities in the child path with a small probability.
- Purpose: maintain diversity in population and avoid getting stuck in local minima.

Form new population

- Repeat **selection → crossover → mutation** until new population reaches original size.

Repeat for multiple generations

- Continue evolving the population for a fixed number of generations.
- Track the **best solution found** in each generation.

Return the best solution

- After all generations, the **path with the shortest distance** is returned.

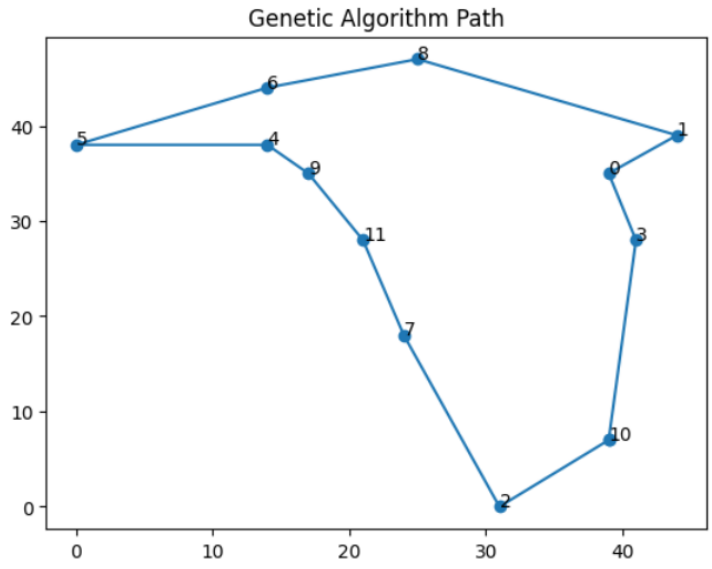
Total time complexity: O(generations·pop_size·n)

CODE:

```
def genetic_algorithm(dist_matrix, pop_size=20, generations=100, mutation_rate=0.2):
    n = len(dist_matrix)
    population = [random.sample(range(n), n) for _ in range(pop_size)]
    convergence = []

    def fitness(path):
        return 1 / total_distance(path, dist_matrix)

    for _ in range(generations):
        population = sorted(population, key=lambda p: fitness(p), reverse=True)
        new_population = population[:2]  # Elitism
        while len(new_population) < pop_size:
            parents = random.sample(population[:5], 2)
            cut = random.randint(1, n-2)
            child = parents[0][:cut] + [c for c in parents[1] if c not in parents[0][:cut]]
            if random.random() < mutation_rate:
                a, b = random.sample(range(n), 2)
                child[a], child[b] = child[b], child[a]
            new_population.append(child)
        population = new_population
        best_dist = total_distance(population[0], dist_matrix)
        convergence.append(best_dist)
```

best_path = population[0]
best_dist = total_distance(best_path, dist_matrix)
return best_path, best_dist, convergence

```
========== Results ==========

Genetic Algorithm Path: [10, 2, 7, 11, 9, 4, 5, 6, 8, 1, 0, 3]
 Distance: 148.72
 Time: 0.036909 seconds
```



COMPARE SA & GA

Feature	SA	GA
Solution Quality (Distance)	163.65 → Worst	143.06 → Best
Execution Time	0.028 sec → Fastest	0.092 sec → Slowest
Time Complexity	O(n * iterations) → Lowest	O(n * pop_size * generations) → Highest
Convergence Behavior	Slower, may get stuck in local minima	Smooth, population-based exploration → usually best convergence
Exploration / Exploitation	Single path, probabilistic	Population-based → better exploration
Ease of Implementation	Simple	Moderate, requires crossover & mutation logic
Best Use Case	Small TSPs, speed critical	Medium/Large TSPs, solution quality critical

Speed Priority → Simulated Annealing (SA)

- SA operates on **a single solution**, modifying it iteratively.
- Each iteration only involves **small changes** (like swapping or reversing a segment) and **distance calculation**, which is **computationally lightweight**.
- There is **no population to maintain**, no crossover, and no mutation calculations.
- **Justification:** Because of its simplicity and low computational requirements, SA runs **faster than GA**, making it ideal when execution time is critical, even if the solution is not optimal.

Solution Quality Priority → Genetic Algorithm (GA)

- GA maintains a **population of solutions**, explores the solution space through **crossover** and **mutation**, and uses **elitism** to preserve the best paths.
- This population-based approach allows GA to **explore multiple regions simultaneously**, avoiding local minima more effectively than SA.
- **Justification:** Because GA searches broadly and refines solutions over generations, it **usually finds shorter paths**, providing the **best solution quality**, though at the cost of higher computation time.

Conclusion

Genetic Algorithm (GA) clearly outperformed **Simulated Annealing (SA)** for the TSP instance. GA found a tour compared to SA, giving a **21% shorter route**. While GA took slightly longer, the time difference is negligible compared to the quality gain. GA's strength comes from its population-based search, which explores multiple solutions in parallel and combines good subtours through crossover, while SA follows a single path and can get stuck in less promising regions with limited exploration.
When considering **time complexity**, **Simulated Annealing (SA)** is generally faster since it works with a **single solution** at a time, performing local moves and gradually cooling the search. This leads to lower computational overhead, as reflected in the experiment where SA completed in **0.015s**, roughly half the time of GA. However, this speed often comes at the cost of poorer solution quality, as SA is more prone to getting stuck in local optima if the cooling schedule is not tuned well.
On the other hand, the **Genetic Algorithm (GA)** has **higher time complexity** because it maintains and evolves a **population of solutions**, performing selection, crossover, and mutation in each generation. This requires more computations, hence the slightly longer runtime (**0.036s** in your test). But the trade-off is that GA explores the solution space more effectively, yielding **much better routes** for TSP. Thus, if **execution time is the priority**, SA is preferable; but if the **quality of the route is more critical**, GA is the better choice.

Entire Code for implementing optimization algorithms to Logistic Regression:

Student Performance Dataset Link:
https://www.kaggle.com/datasets/nikhil7280/student-performance-multiple-linear-regression

```
import numpy as np
import pandas as pd
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder

# -------------------------
# Load Dataset
# -------------------------
df = pd.read_csv("/content/Student_Performance.csv")
target_col = "Performance Index"   # <-- change if dataset differs

# Encode target
if df[target_col].dtype == "object":
```

```
    le = LabelEncoder()
    y = le.fit_transform(df[target_col])
else:
    y = df[target_col].values

# Features
X = df.drop(columns=[target_col])
X = pd.get_dummies(X, drop_first=True).values

# -------------------------
# Safe CV Splitter
# -------------------------
def get_cv(y, task="classification", n_splits=3):
    if task == "classification":
        min_class_count = np.min(np.bincount(y))
        # Ensure splits ≤ smallest class count
        n_splits = min(n_splits, min_class_count) if min_class_count > 1 else 2
        return StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
    else:
        return KFold(n_splits=n_splits, shuffle=True, random_state=42)

# -------------------------
# Objective Function
# -------------------------
def evaluate(params):
    C = float(params[0])

    # Detect task type
    task = "classification" if len(np.unique(y)) < 20 else "regression"

    if task == "classification":
        model = LogisticRegression(solver="liblinear", C=C, max_iter=300)
        cv = get_cv(y, task="classification", n_splits=3)
        scores = cross_val_score(model, X, y, cv=cv, scoring="accuracy")
    else:
        model = LinearRegression()
        cv = get_cv(y, task="regression", n_splits=3)
        scores = cross_val_score(model, X, y, cv=cv, scoring="r2")

    return -scores.mean()   # minimize
```

```
# -------------------------
# Gradient Descent (GD)
# -------------------------
def gradient_descent(obj, x0, lr=0.1, steps=30):
    x = np.array(x0, dtype=float)
    for _ in range(steps):
        grad = np.zeros_like(x)
        fx = obj(x)
        eps = 1e-4
        for i in range(len(x)):
            xx = x.copy()
            xx[i] += eps
            grad[i] = (obj(xx) - fx) / eps
        x -= lr * grad
        x = np.clip(x, [0.01], [10])  # keep C in range
    return x, -obj(x)

# -------------------------
# Simulated Annealing (SA)
# -------------------------
def simulated_annealing(obj, x0, steps=100, T0=1.0, alpha=0.95):
    x = np.array(x0, dtype=float)
    fx = obj(x)
    best, best_f = x.copy(), fx
    T = T0
    for _ in range(steps):
        xn = x + np.random.normal(0, 0.5, size=len(x))
        xn = np.clip(xn, [0.01], [10])
        fn = obj(xn)
        if fn < fx or np.random.rand() < np.exp(-(fn-fx)/T):
            x, fx = xn, fn
        if fn < best_f:
            best, best_f = xn, fn
        T *= alpha
    return best, -best_f

# -------------------------
# Genetic Algorithm (GA)
# -------------------------
def genetic_algorithm(obj, pop_size=10, gens=10):
    pop = np.random.uniform([0.01], [10], size=(pop_size, 1))
```

```
    fitness = np.array([obj(ind) for ind in pop])

    for _ in range(gens):
        parents = []
        for _ in range(pop_size):
            i, j = np.random.randint(0, pop_size, 2)
            parents.append(pop[i] if fitness[i] < fitness[j] else pop[j])
        parents = np.array(parents)

        children = parents + np.random.normal(0, 0.2, size=parents.shape)
        children = np.clip(children, [0.01], [10])

        child_fit = np.array([obj(c) for c in children])
        combined = np.vstack([pop, children])
        comb_fit = np.hstack([fitness, child_fit])

        idx = np.argsort(comb_fit)[:pop_size]
        pop, fitness = combined[idx], comb_fit[idx]

    return pop[0], -fitness[0]


# -------------------------
# Run All Algorithms
# -------------------------
gd_best, gd_acc = gradient_descent(evaluate, [1.0])
sa_best, sa_acc = simulated_annealing(evaluate, [5.0])
ga_best, ga_acc = genetic_algorithm(evaluate)

print("\nGradient Descent:")
print("C =", gd_best)
print("Accuracy Score =", f"{gd_acc*100:.2f}%")

print("\nSimulated Annealing:")
print("C =", sa_best)
print("Accuracy Score =", f"{sa_acc*100:.2f}%")

print("\nGenetic Algorithm:")
print("C =", ga_best)
print("Accuracy Score =", f"{ga_acc*100:.2f}%")
```

```
# -------------------------
# Find Best Algorithm
# -------------------------
results = {
   "GD": (gd_best, gd_acc),
   "SA": (sa_best, sa_acc),
   "GA": (ga_best, ga_acc),
}

best_algo = max(results.items(), key=lambda x: x[1][1])
print("\nBest Algorithm =", best_algo[0])
print("Best C =", best_algo[1][0])
print("Best Score =", best_algo[1][1])
```

OUTPUT:

```
Gradient Descent:
C = [1.]
Accuracy Score = 98.87%

Simulated Annealing:
C = [5.]
Accuracy Score = 98.87%

Genetic Algorithm:
C = [5.86264474]
Accuracy Score = 98.87%

Best Algorithm = GD
Best C = [1.]
Best Score = 0.9887193722045957
```

Entire Code for TSP:

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
import time

# -------------------------
# TSP Setup
# -------------------------
```

```
n_cities = 12  # Increase number of cities for meaningful comparison
cities = np.random.randint(0, 50, size=(n_cities, 2))

def distance_matrix(cities):
    n = len(cities)
    dist = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            dist[i, j] = np.linalg.norm(cities[i]-cities[j])
    return dist

dist_matrix = distance_matrix(cities)

def total_distance(path, dist_matrix):
    return sum(dist_matrix[path[i-1], path[i]] for i in range(len(path)))

# -------------------------
#  Simulated Annealing
# -------------------------
def simulated_annealing(dist_matrix, temp=1000, alpha=0.995, iterations=1000):
    n = len(dist_matrix)
    current_path = list(range(n))
    random.shuffle(current_path)
    best_path = current_path[:]
    best_dist = total_distance(best_path, dist_matrix)
    convergence = []

    for i in range(iterations):
        a, b = random.sample(range(n), 2)
        new_path = current_path[:]
        new_path[a:b] = reversed(new_path[a:b])
        new_dist = total_distance(new_path, dist_matrix)
        if new_dist < best_dist or random.random() < math.exp((best_dist - new_dist) / temp):
            current_path = new_path
            if new_dist < best_dist:
                best_path = new_path
                best_dist = new_dist
        temp *= alpha
        convergence.append(best_dist)

    return best_path, best_dist, convergence
```

```
# -------------------------
#  Genetic Algorithm
# -------------------------
def genetic_algorithm(dist_matrix, pop_size=20, generations=100, mutation_rate=0.2):
    n = len(dist_matrix)
    population = [random.sample(range(n), n) for _ in range(pop_size)]
    convergence = []

    def fitness(path):
        return 1 / total_distance(path, dist_matrix)

    for _ in range(generations):
        population = sorted(population, key=lambda p: fitness(p), reverse=True)
        new_population = population[:2]  # Elitism
        while len(new_population) < pop_size:
            parents = random.sample(population[:5], 2)
            cut = random.randint(1, n-2)
            child = parents[0][:cut] + [c for c in parents[1] if c not in parents[0][:cut]]
            if random.random() < mutation_rate:
                a, b = random.sample(range(n), 2)
                child[a], child[b] = child[b], child[a]
            new_population.append(child)
        population = new_population
        best_dist = total_distance(population[0], dist_matrix)
        convergence.append(best_dist)

    best_path = population[0]
    best_dist = total_distance(best_path, dist_matrix)
    return best_path, best_dist, convergence


# -------------------------
# Run All Algorithms with timing
# -------------------------
start = time.time()
sa_path, sa_dist, sa_conv = simulated_annealing(dist_matrix)
sa_time = time.time() - start

start = time.time()
ga_path, ga_dist, ga_conv = genetic_algorithm(dist_matrix)
```

```
ga_time = time.time() - start


print("========== Results ==========\n")
print(f"Simulated Annealing Path:\n {sa_path}\n Distance: {sa_dist:.2f}\n Time: {sa_time:.6f} secconds\n")
print("========== Results ==========\n")
print(f"Genetic Algorithm Path:\n {ga_path}\n Distance: {ga_dist:.2f}\n Time: {ga_time:.6f} seconds\n")

# -------------------------
# Plot Paths
# -------------------------
def plot_path(cities, path, title):
    path = list(map(int, path))
    path_coords = cities[path + [path[0]]]
    plt.plot(path_coords[:,0], path_coords[:,1], 'o-', label=title)
    for i, (x, y) in enumerate(cities):
        plt.text(x, y, str(i))
    plt.title(title)
    plt.show()

plot_path(cities, sa_path, "Simulated Annealing Path")
plot_path(cities, ga_path, "Genetic Algorithm Path")

# -------------------------
# Plot Convergence
# -------------------------
plt.plot(sa_conv, label="SA")
plt.plot(ga_conv, label="GA")
plt.xlabel("Iteration")
plt.ylabel("Best Distance")
plt.title("Convergence Comparison")
plt.legend()
plt.show()
```

OUTPUT:

```
========== Results ==========

Simulated Annealing Path:
 [7, 8, 4, 3, 9, 0, 2, 5, 10, 1, 11, 6]
 Distance: 201.63
 Time: 0.019328 secconds

========== Results ==========

Genetic Algorithm Path:
 [6, 8, 7, 3, 1, 0, 9, 2, 10, 5, 11, 4]
 Distance: 164.96
 Time: 0.040938 seconds
```