```
!pip install genetic_algorithm
```

```
Collecting genetic_algorithm
  Downloading genetic_algorithm-1.0.0.tar.gz (6.9 kB)
  Preparing metadata (setup.py) ... ent already satisfied: numpy in
/usr/local/lib/python3.11/dist-packages (from genetic_algorithm)
(2.0.2)
Requirement already satisfied: pandas in
/usr/local/lib/python3.11/dist-packages (from genetic_algorithm)
(2.2.2)
Requirement already satisfied: scipy in
/usr/local/lib/python3.11/dist-packages (from genetic_algorithm)
(1.14.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.11/dist-packages (from pandas-
>genetic_algorithm) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.11/dist-packages (from pandas-
>genetic_algorithm) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in
/usr/local/lib/python3.11/dist-packages (from pandas-
>genetic_algorithm) (2025.1)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2-
>pandas->genetic_algorithm) (1.17.0)
Building wheels for collected packages: genetic_algorithm
  Building wheel for genetic_algorithm (setup.py) ... :
filename=genetic_algorithm-1.0.0-py3-none-any.whl size=7614
sha256=011b1156c78d3f533dbbd350566de931f32684183e094e4ac1dbf7bb68c54cc
2
  Stored in directory:
/root/.cache/pip/wheels/22/ca/e8/e5be5f6cf6868badb376ac1ecb29beb47e980
3a0a14827c72a
Successfully built genetic_algorithm
Installing collected packages: genetic_algorithm
Successfully installed genetic_algorithm-1.0.0
```

```
import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Sample data (features and target)
X = np.random.rand(100, 5)  # Example features
y = np.random.rand(100)  # Example target

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

```python
# Define the neural network model
nn_model = MLPRegressor(hidden_layer_sizes=(100, 50),
activation='relu', solver='adam', random_state=42)

# Define a GeneticAlgorithm class with an optimize method
class GeneticAlgorithm:
    def __init__(self, fitness_function=None, parameter_ranges=None,
population_size=50, crossover_rate=0.8, mutation_rate=0.05,
generations=20):
        # If initialized with parameter values (for use in fitness
function)
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate

        # If initialized for optimization
        self.fitness_function = fitness_function
        self.parameter_ranges = parameter_ranges
        self.generations = generations

    def optimize(self):
        """Optimize parameters using genetic algorithm"""
        # Initialize population randomly within parameter ranges
        population = []
        for _ in range(self.population_size):
            individual = {}
            for param, (min_val, max_val) in
self.parameter_ranges.items():
                if param == 'population_size':
                    # For integer parameters
                    individual[param] = np.random.randint(min_val,
max_val + 1)
                else:
                    # For float parameters
                    individual[param] = np.random.uniform(min_val,
max_val)
            population.append(individual)

        best_individual = None
        best_fitness = float('inf')  # We're minimizing MSE

        # Run for specified number of generations
        for generation in range(self.generations):
            # Evaluate fitness for each individual
            fitness_scores = []
            for individual in population:
                params = (individual['population_size'],
                          individual['crossover_rate'],
                          individual['mutation_rate'])
                fitness = self.fitness_function(params)
```

```
                fitness_scores.append(fitness)

                # Track best individual
                if fitness < best_fitness:
                    best_fitness = fitness
                    best_individual = individual.copy()

            # Print progress
            print(f"Generation {generation+1}/{self.generations}, Best
MSE: {best_fitness:.6f}")

            # Create new population
            new_population = []

            # Elitism: Keep the best individual
            new_population.append(best_individual)

            # Tournament selection and crossover
            while len(new_population) < self.population_size:
                # Tournament selection
                parent1 = self._tournament_selection(population,
fitness_scores)
                parent2 = self._tournament_selection(population,
fitness_scores)

                # Crossover
                if np.random.random() < self.crossover_rate:
                    child = self._crossover(parent1, parent2)
                else:
                    child = parent1.copy()

                # Mutation
                child = self._mutation(child)

                new_population.append(child)

            # Update population
            population = new_population

        return best_individual

    def _tournament_selection(self, population, fitness_scores,
tournament_size=3):
        """Select individual using tournament selection"""
        indices = np.random.choice(len(population), tournament_size,
replace=False)
        tournament_fitness = [fitness_scores[i] for i in indices]
        best_idx = indices[np.argmin(tournament_fitness)]  # Minimize
MSE
        return population[best_idx].copy()
```

```
    def _crossover(self, parent1, parent2):
        """Perform crossover between two parents"""
        child = {}
        for param in parent1.keys():
            # 50% chance of inheriting from each parent
            if np.random.random() < 0.5:
                child[param] = parent1[param]
            else:
                child[param] = parent2[param]
        return child

    def _mutation(self, individual):
        """Apply mutation to an individual"""
        mutated = individual.copy()
        for param, (min_val, max_val) in
self.parameter_ranges.items():
            # Apply mutation with probability self.mutation_rate
            if np.random.random() < self.mutation_rate:
                if param == 'population_size':
                    # For integer parameters
                    mutated[param] = np.random.randint(min_val,
max_val + 1)
                else:
                    # For float parameters, small perturbation
                    delta = (max_val - min_val) * 0.1  # 10% of range
                    mutated[param] += np.random.uniform(-delta, delta)
                    # Keep within bounds
                    mutated[param] = max(min_val, min(max_val,
mutated[param]))
        return mutated

# Define the fitness function for the genetic algorithm
def fitness_function(params):
    # Unpack genetic algorithm parameters
    population_size, crossover_rate, mutation_rate = params

    # Create the genetic algorithm object
    ga = GeneticAlgorithm(population_size=int(population_size),
crossover_rate=crossover_rate, mutation_rate=mutation_rate)

    # Train the neural network model
    nn_model.fit(X_train, y_train)

    # Evaluate the model
    y_pred = nn_model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)

    return mse
```

```python
# Define the parameter ranges for the genetic algorithm
parameter_ranges = {
    'population_size': (50, 100),
    'crossover_rate': (0.6, 0.9),
    'mutation_rate': (0.01, 0.1)
}

# Create the genetic algorithm object
ga = GeneticAlgorithm(fitness_function=fitness_function,
parameter_ranges=parameter_ranges, generations=10)

# Optimize the genetic algorithm parameters
best_params = ga.optimize()
print("Best Parameters:", best_params)

# Train the neural network with optimized GA parameters
final_ga = GeneticAlgorithm(
    population_size=int(best_params['population_size']),
    crossover_rate=best_params['crossover_rate'],
    mutation_rate=best_params['mutation_rate']
)

# Train the final model with the best parameters
nn_model.fit(X_train, y_train)

# Evaluate and print final results
y_pred = nn_model.predict(X_test)
final_mse = mean_squared_error(y_test, y_pred)
print(f"Final Model MSE: {final_mse:.6f}")
```

```
Generation 1/10, Best MSE: 0.072562
Generation 2/10, Best MSE: 0.072562
Generation 3/10, Best MSE: 0.072562
Generation 4/10, Best MSE: 0.072562
Generation 5/10, Best MSE: 0.072562
Generation 6/10, Best MSE: 0.072562
Generation 7/10, Best MSE: 0.072562
Generation 8/10, Best MSE: 0.072562
Generation 9/10, Best MSE: 0.072562
Generation 10/10, Best MSE: 0.072562
Best Parameters: {'population_size': 56, 'crossover_rate':
0.7856294212392513, 'mutation_rate': 0.05654018449040655 6}
Final Model MSE: 0.072562
```