Evaluation of Factor Models using genetic algorithms
Introduction

The following implementation illustrates the use of Genetic Algorithms (GAs) to choose the factors via regression analysis. The objective is to select the best set of factors that most accurately describe a target variable based on R^2 (coefficient of determination) as the fitness measure.

Factor Selection in Finance

In econometrics and quantitative finance, one typically has data sets with hundreds of potential explanatory variables (factors) but needs to identify which subset is most influential for a target variable (e.g., stock returns). Classical approaches like stepwise regression are computationally infeasible and may miss the optimal combinations.

Challenge: Out of 117 potential factors, find the best set of exactly 3/4/5 factors that results in the highest R^2 of a regression model for monthly excess returns.

Genetic Algorithm for Factor Selection

In this implementation, the biological concepts are mapped as follows:

Concept	**Implementation Mapping**
Individual/Chromosome	A set of Factors (e.g., ['SEAS_16_20AN', 'RET_9_1', 'RMAX5_21D'])
Gene	One factor from the data
Population	A set of different factor combinations
Fitness Score	R^2 value of OLS regression on those factors
Selection	Select factor combinations with higher R^2 values
Crossover	Combine factors from two parents
Mutation	Randomly replace one factor in a combination

Algorithm Components

1. Data Loading and Preprocessing

def load_data():
 Selecting query(new).xlsx or 1000Lines_Clusters_Mappings.xlsm dataset happens here. This function also cleans column names, selects numeric factors and defines target variable (monthly excess returns)

Purpose: Prepares the financial dataset by:
- Loading factor data (independent variables)
- Identifying the target variable (dependent variable)
- Cleaning and normalizing column names
- Filtering to numeric factors only

2. Fitness Function

def get_fitness_with_pvals(chrom, X_all, y_all, target_col):
This function evaluates a chromosome using OLS regression and returns R^2 score and stores p-values.

The Heart of the Algorithm: This function determines how good a factor combination is by:
- Running an OLS regression
- Calculating R^2 (explained variance ratio)
- Higher R^2 = better fitness = greater chance of survival and reproduction
- Also tracks statistical significance (p-values) for analysis

Fitness Scoring Logic:
- Perfect Score**: $R^2 = 1.0$ (explains 100% of variance explained)
- Good Score**: $R^2 > 0.5$ (explains >50% of variance explained)
- Poor Score**: $R^2 < 0.1$ (explains <10% of variance explained)
- Invalid Score**: -999 (not enough data or errors)

3. Population Initialization

def create_unique_population(genes, pop_size):
  Produces first generation of unique factor combinations

Starting Point: Produces the first generation by:
- Selecting 3 factors randomly from all factors
- Ensuring each combination is unique (no duplicates)
- Producing enough combinations to fill the population

Example Initial Population (if factors are 'SALE_BEV', 'RET_1_0', 'LNOA_GR1A', 'OPEX_AT', 'RET_60_12', 'RMAX5_21D'):
Individual 1: ['SALE_BEV', 'LNOA_GR1A', 'RMAX5_21D']
Individual 2: ['RET_1_0', 'OPEX_AT', 'RET_60_12']
Individual 3: ['SALE_BEV', 'RET_1_0', 'OPEX_AT']
Individual 4: ['LNOA_GR1A', 'RET_60_12', 'RMAX5_21D']
…

4. Selection Operator: Roulette Wheel Selection

def roulette_selection(pop, X_all, y_all, target_col):
   Fitness-proportionate selection
   Higher R^2 = higher probability of being chosen as parent

Survival of the Fittest: Simulates natural selection by:
- Giving combinations with higher R^2 more chance to reproduce
- Still allowing bad combinations with some chance (maintains diversity)
- Using probability proportionate to fitness values

Selection Probability Example:

Combination A: R² = 0.8 → 40% selection chance of being selected
Combination B: R² = 0.6 → 30% selection chance of being selected
Combination C: R² = 0.4 → 20% chance of selection
Combination D: R² = 0.2 → 10% chance of selection

5. Crossover Operator: Uniform Crossover

def uniform_crossover(p1, p2):
This combines factors from two parent chromosomes and creates offspring with mixed characteristics.

Reproduction Process: Generates new factor combinations by:
- Taking factors randomly from both parents
- Avoiding duplicate factors in the offspring
- Forming potentially better combinations than both parents

Crossover Example:

Parent 1: [‘SALE_BEV', 'RET_1_0', ‘LNOA_GR1A’]
Parent 2: ['OPEX_AT', 'RET_60_12', 'RMAX5_21D']

Possible Offspring: [‘SALE_BEV', 'RET_60_12', ‘LNOA_GR1A’]
(inherits ‘SALE_BEV' & ‘LNOA_GR1A’ from Parent 1, 'RET_60_12' from Parent 2)

6. Mutation Operator

def mutate(chrom, usable, rate=0.2):
Randomly replacing one factor with another different factor to maintain genetic diversity

Genetic Diversity: Avoids algorithm from getting stuck:
- Add new factors (20% chance) randomly
- Substitute one factor with a completely different one
- Explore factor combinations that might not happen from crossover alone

Mutation Example:

Before Mutation: [SALE_BEV', 'RET_1_0', ‘LNOA_GR1A’]
After Mutation: [SALE_BEV', ‘LNOA_GR1A', ‘LNOA_GR1A’] ('RET_1_0' → ‘LNOA_GR1A’)

7. Evolution Process

Evolve the algorithm across generations by implementing these steps:

Generation Loop:

1. Evaluation: Calculate R^2 for all factor combinations
2. Ranking: Rank combinations by fitness (R^2 score)
3. Elitism: Keep best combinations for next generation
4. Selection: Choose parents based on roulette wheel selection
5. Reproduction: Produce offspring by crossover
6. Mutation: Apply random mutations for diversity
7. Replacement: Form new generation

Convergence Criteria:
- Target Reached $R^2 > 0.999$ (nearly perfect fit)
- Maximum Generations: Hit predefined number
- Stagnation: No improvement after numerous generations

Key Features of This Implementation

1. Multiple Configuration Testing

- Tests different population sizes (20, 50)
- Tests different generation counts (20, 50)
- Runs multiple experiments per configuration

Compares performance across settings

2. Caching System

```
fitness_cache = {}
        Avoids recalculating same combinations
```

- Stores calculated R^2 values and speeds up algorithm when the same combinations reappears. Particularly useful in later generations with elitism

3. Statistical Analysis
- Tracks p-values for statistical significance
- Calculates most chosen factors by frequency
- Provides complete OLS regression summaries

4. Visualization Components

For each Population size, Generation combination

Dashboard 1 (Performance Analysis)
- Convergence Plot - Line chart showing R^2 progression across generations for each run
- Performance Statistics - Histogram of final R^2 distribution with mean/median lines
- Learning Curve Analysis - Line chart with mean $R^2 \pm$ std deviation across generations
- Configuration Summary - Text summary box with key metrics

Dashboard 2 (Factor Analysis)
- Factor Importance Analysis - Bar chart showing factor selection frequency
- Factor Performance Boxplot - Box plots of R^2 scores for top factors
- Factor P-value Analysis - Bar chart of average p-values with significance thresholds
- Optimization Efficiency - Bar chart of improvements per run

Dashboard 3 (OLS Regression Analysis Dashboard)
- Regression Coefficients - Horizontal bar chart with significance markers
- Residuals vs Fitted - Scatter plot for residual analysis
- Q-Q Plot - Quantile-quantile plot for normality check
- Model Statistics - Text box with regression statistics

Cross-Configuration Comparison Dashboards
Dashboard 1 (Performance Comparison)
- Configuration Performance Comparison - Bar chart with error bars comparing all configs
- Average vs Best Performance - Grouped bar chart showing average vs best R^2
- Population Size Impact - Box plots showing performance by population size
- Generation Count Impact - Box plots showing performance by generation count

Dashboard 2 (Other Analysis)
- Performance Heatmap - 2D heatmap of population size vs generations
- Top Most Consistent Factors - Bar chart of factors selected across configs
- Convergence Speed Analysis - Bar chart of generations to reach 90% performance
- Overall Summary Statistics - Text summary box

Interactive Plotly Dashboard:
- Configuration Performance - Interactive bar chart with error bars
- Population vs Generations Impact - 3D scatter plot
- Convergence Comparison - Interactive line chart with multiple series

Real-World Application Example

Scenario: Have a data set of 117 financial factors and want to build a model of predicting monthly stock returns.

Traditional Approach Problems:
- Testing all combinations of 3 factors = C(117, 3)
- Computationally expensive to test all
- Can overlook interaction effects

GA Approach Benefits:
- Intelligently searches the solution space
- Finds near-optimal solutions in reasonable time
- Explores diverse factor combinations
- Provides statistical validation

Example Results:

Best Factor Combination Found:
- 'GROSS PROFIT CHANGE 3YR (GP_GR3A)’
- 'OPERATING EARNINGS TO EQUITY CHANGE 3YR (OPE_GR3A)’
- 'HIRING RATE (EMP_GR1)'

$R^2 = 0.647$ (64.7% of variance explained)

Performance Interpretation

Convergence Patterns:
- Fast Convergence: Algorithm converges on good solutions quickly (good)
- Premature Convergence: Stops improving too early (bad - increase mutation)
- Slow Convergence: Improves slowly over many generations (normal)
- No Convergence: Fitness does not increase (bad - change parameters)

Factor Selection Insights:
- Often Selected Factors: Consistently relevant across runs
- High R^2 Factors: Good predictive ability
- Low P-values: Statistically significant relationship

Parameter Tuning Guidelines

Population Size:
- Small (20): Quicker execution, might miss perfect solutions
- Large (100+): More exploration, slower execution

Generations:
- Few (20): Fast results, might not converge
- Many (100+): Improved solutions, increased runtime
- Stop Early: If $R^2 > 0.99$ or no improvement

Mutation Rate:
- Low (0.1)- faster convergence, less exploration
- High (0.3)- more exploration, slower convergence
- In this code (0.2): good compromise for most problems

Strengths of This GA Approach

1. Global Search: Avoids local optima that plague greedy methods
2. Parallel Evaluation: Can evaluate several solutions simultaneously
3. Flexibility* Easy to modify for different objectives or constraints
4. Robustness: Works well even for noisy or incomplete data
5. Interpretability: Provides insight into factor importance

Considerations

1. No Guarantee of Global Optimum: Heuristic approach
2. Parameter Sensitivity: Results depend on GA parameters
3. Computational Cost: Still requires many regression evaluations

4. Random Element: Results may vary between runs

This code is a good foundation for factor selection problems and could be adapted to other regression and optimization problems in quantitative finance and beyond.