STAT 243 Final Project
Genetic Algorithm for Variable Selection

Jiarong Zhou Fangyuan Li Hannah Neumann

URL: https://github.berkeley.edu/fangyuan-li/STAT243-Final-Project

Introduction

This project presents a Genetic Algorithm (GA) designed for efficient variable selection in linear regression and Generalized Linear Models (GLMs). Our GA addresses the challenges of traditional variable selection methods, like computational inefficiency and limited scalability, by offering a robust and flexible solution.

Algorithm Design

Our GA optimizes variable selection through:

- **Population Initialization**: Generating initial variable subsets.
- **Fitness Function**: Utilizing AIC, BIC, Adjusted R2, or custom metrics.
- **Genetic Operators**: Implementing selection, crossover, mutation, and optional custom operators to evolve solutions.
- **Termination Criteria**: Running for a predefined number of generations or until convergence.

Integration with Regression Models

The GA's compatibility with statsmodels and scikit-learn models underscores its versatility in statistical analysis.

Testing on Diverse Datasets

To validate the effectiveness of our GA, we conducted tests on four distinct datasets in addition to simulated data. This approach allowed us to assess the algorithm's performance across varying data characteristics, different types of regression models, and complexities.

- baseball data
- wine data
- admission data
- date fruit data (multi-classification)
- simulated data

```
from sklearn.linear_model import lasso_path, LinearRegression,
 ↪  BayesianRidge, Ridge, LogisticRegression
import statsmodels.api as sm
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import GA.GA as GA
from sklearn import linear_model

# Load Data
file_path = 'data/baseball.dat'
try:
    data = pd.read_csv(file_path, delim_whitespace=True)
except Exception as e:
    print("Error loading file:", e)
    exit()

# Data Preparation
target_variable = 'salary'
variables = [col for col in data.columns if col != target_variable]
X = data[variables]
y = data[target_variable]
```

LASSO Model Selection

To benchmark the performance of our Genetic Algorithm, we started with LASSO model selection. The LASSO algorithm computed a regularization path and was evaluated using AIC across different alpha values. This provided insight into the trade-off between model complexity and fit. Normalization is a crucial step here, ensuring that all features contribute equally to the model.

```
# Normalize data for LASSO
X_norm = X / X.std(axis=0)
y_norm = y / y.std(axis=0)

# LASSO Model Selection
print("Computing regularization path using LASSO...")
alphas_lasso, coefs_lasso, _ = lasso_path(X_norm, y_norm, eps=0.001)

lin_model_sel_var_AIC = np.zeros(coefs_lasso.shape[1])
for i in range(coefs_lasso.shape[1]):
    selected_variables = np.array(variables)[coefs_lasso[:, i] != 0]
    if len(selected_variables) > 0:
        model = LinearRegression().fit(X[selected_variables], y)
        prediction = model.predict(X[selected_variables])
        lin_model_sel_var_AIC[i] = GA.calculate_aic([y, prediction,
 ↪  len(selected_variables)])
    else:
        lin_model_sel_var_AIC[i] = np.inf

best_lasso_index = np.argmin(lin_model_sel_var_AIC)
selected_variables_best_AIC = np.array(variables)[coefs_lasso[:,
 ↪  best_lasso_index] != 0]
lin_model_sel_var_best_AIC =
 ↪  LinearRegression().fit(X[selected_variables_best_AIC], y)

print('Lasso: Selected Variables for the Model with the best AIC:',
 ↪  selected_variables_best_AIC)
```

```
Computing regularization path using LASSO...
Lasso: Selected Variables for the Model with the best AIC: ['obp' 'runs' 'homeruns' 'rbis' '
 'soserrors' 'sbsruns']
```

Greedy Algorithm Implementation

We further explored variable selection using a Greedy algorithm, which incrementally built a model by adding variables that most improved the AIC. This method served as another comparative model to gauge the GA's performance.

```python
# Greedy Algorithm Implementation
print("Running Greedy algorithm for variable selection...")
X_new = sm.add_constant(X)
greedy_aic = np.zeros(X_new.shape[1] - 1)
greedy_var_to_keep = []

for i in range(X_new.shape[1] - 1):
    fitted_model = sm.OLS(y, X_new).fit()
    var_max_pval = np.argmax(fitted_model.pvalues)
    greedy_var_to_keep.append(X_new.columns.tolist())

    prediction = fitted_model.predict(X_new)
    greedy_aic[i] = GA.calculate_aic([y, prediction, len(X_new.columns)
 ↪  - 1])

    X_new = X_new.drop(X_new.columns[var_max_pval], axis=1)

best_greedy_index = np.argmin(greedy_aic)
print('Greedy: Selected Variables for the Model with the best AIC:',
 ↪  greedy_var_to_keep[best_greedy_index])
```

```
Running Greedy algorithm for variable selection...
Greedy: Selected Variables for the Model with the best AIC: ['runs', 'hits', 'rbis', 'walks'
```

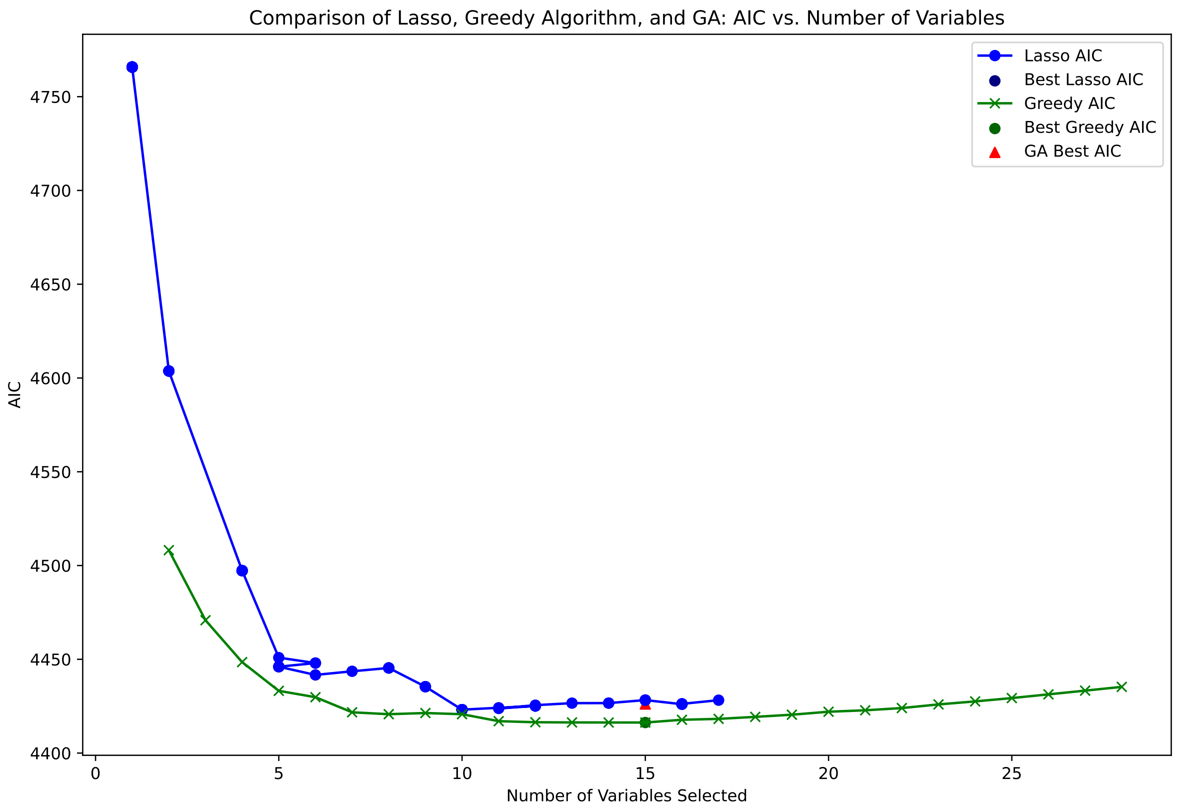
Visual comparison of the different methods

To visually compare the performance of LASSO, the Greedy algorithm, and the GA, we plotted the AIC against the number of variables selected by each method.

```python
# Genetic Algorithm with sm.OLS
print("Running GA with sm.OLS...")
ga_ols = GA.GeneticAlgorithm(num_generations=2000, population_size=2000,
 ↪  selection_method='tournament', mutation_rate=0.02)
ga_ols.fit(data, target_variable, FUN=sm.OLS)
selected_variables_ols, best_fitness_ols = ga_ols.run()
print("GA(sm.OLS): Selected Variables:", selected_variables_ols)
print("Best Fitness (AIC):", best_fitness_ols)

# Genetic Algorithm with Linear Regression
print("Running GA with Linear Regression...")
```

```python
ga_lr = GA.GeneticAlgorithm(num_generations=2000, population_size=2000,
 ↪  selection_method='tournament', mutation_rate=0.03)
ga_lr.fit(data, target_variable, FUN=LinearRegression)
selected_variables_lr, best_fitness_lr = ga_lr.run()
print("GA(Linear Regression): Selected Variables:",
 ↪  selected_variables_lr)
print("Best Fitness (AIC):", best_fitness_lr)

# Plotting for Comparison
num_variables_selected_lasso = np.sum(coefs_lasso != 0, axis=0)
num_variables_selected_greedy = [len(vars) for vars in
 ↪  greedy_var_to_keep]
num_variables_selected_ga = len(selected_variables_ols)

plt.figure(figsize=(12, 8))
plt.plot(num_variables_selected_lasso, lin_model_sel_var_AIC,
 ↪  marker='o', color='blue', label='Lasso AIC')
plt.scatter(num_variables_selected_lasso[best_lasso_index],
 ↪  lin_model_sel_var_AIC[best_lasso_index], color='navy', label='Best
 ↪  Lasso AIC')
plt.plot(num_variables_selected_greedy, greedy_aic, marker='x',
 ↪  color='green', label='Greedy AIC')
plt.scatter(num_variables_selected_greedy[best_greedy_index],
 ↪  greedy_aic[best_greedy_index], color='darkgreen', label='Best Greedy
 ↪  AIC')
plt.scatter(num_variables_selected_ga, best_fitness_ols, marker='^',
 ↪  color='red', label='GA Best AIC')
plt.xlabel('Number of Variables Selected')
plt.ylabel('AIC')
plt.title('Comparison of Lasso, Greedy Algorithm, and GA: AIC vs. Number
 ↪  of Variables')
plt.legend()
plt.show()
```

```
Running GA with sm.OLS...
GA(sm.OLS): Selected Variables: ['obp', 'runs', 'hits', 'doubles', 'homeruns', 'rbis', 'sos'
Best Fitness (AIC): 4426.151424041115
Running GA with Linear Regression...
GA(Linear Regression): Selected Variables: ['average', 'hits', 'rbis', 'walks', 'sos', 'sbs'
Best Fitness (AIC): 4427.054181698666
```



Genetic Algorithm with Various Regression Models and Generalized Linear Models

The core of our testing involved running the GA with different regression functions:

- sm.OLS: The GA, when coupled with ordinary least squares regression from statsmodels, selected variables and computed the best fitness (AIC), demonstrating the algorithm's adaptability to traditional statistical methods.
- Linear Regression: Applying the GA with scikit-learn's Linear Regression revealed its ability to work seamlessly with machine learning frameworks.
- Bayesian Ridge and Ridge Regression: The GA was also tested with Bayesian Ridge and Ridge Regression models, highlighting its versatility in handling regularized regression methods.
- Logistic Regression: The GA was applied on multi-classification dataset using scikit-learn's Logistic Regression model.

```
# Genetic algorithm with Bayesian Ridge Regression
print("Running GA with Bayesian Ridge Regression...")
ga_br = GA.GeneticAlgorithm(num_generations=1000, population_size=500,
 ↪  selection_method='tournament', mutation_rate=0.02)
ga_br.fit(data, target_variable, FUN=linear_model.BayesianRidge)
selected_variables_br, best_fitness_br = ga_br.run()
print("GA(Bayesian Ridge): Selected Variables:", selected_variables_br)
print("Best Fitness (AIC):", best_fitness_br)

# Genetic algorithm with Ridge Regression
ga_ridge = GA.GeneticAlgorithm(num_generations=1000,
 ↪  population_size=500, selection_method='tournament',
 ↪  mutation_rate=0.02)
ga_ridge.fit(data, target_variable, FUN=linear_model.Ridge(alpha=0.5))
selected_variables_ridge, best_fitness_ridge = ga_ridge.run()
print("GA(Ridge): Selected Variables:", selected_variables_ridge)
print("Best Fitness (AIC):", best_fitness_ridge)

# Genetic algorithm with Generalized Linear Models (GLM)
print("Running GA with Generalized Linear Models...")
ga_glm = GA.GeneticAlgorithm(num_generations=1000, population_size=500,
 ↪  selection_method='tournament', mutation_rate=0.02)
ga_glm.fit(data, target_variable, FUN=sm.GLM,
 ↪  family=sm.families.Gaussian())
selected_variables_glm, best_fitness_glm = ga_glm.run()
print("GA(GLM): Selected Variables:", selected_variables_glm)
print("Best Fitness (AIC):", best_fitness_glm)
```

```
Running GA with Bayesian Ridge Regression...
GA(Bayesian Ridge): Selected Variables: ['obp', 'runs', 'doubles', 'homeruns', 'walks', 'sos
Best Fitness (AIC): 4440.693510050216
GA(Ridge): Selected Variables: ['runs', 'doubles', 'triples', 'rbis', 'sos', 'errors', 'free
Best Fitness (AIC): 4427.433521934424
Running GA with Generalized Linear Models...
GA(GLM): Selected Variables: ['doubles', 'homeruns', 'rbis', 'sos', 'errors', 'freeagent', '
Best Fitness (AIC): 4430.602493661757
```

Genetic Algorithm with Logistic Regression Model

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

file_path = 'data/Date_Fruit_Datasets.xlsx'
try:
    data = pd.read_excel(file_path)
except Exception as e:
    print("Error loading file with comma as delimiter:", e)

# Data Preparation
scaler = StandardScaler()
label_encoder = LabelEncoder()
target_variable = 'Class'
data[target_variable] =
 ↪  label_encoder.fit_transform(data[target_variable])
data[data.columns[:-1]] = scaler.fit_transform(data[data.columns[:-1]])

# Genetic Algorithm with Logistic Regression Model
ga_logit = GA.GeneticAlgorithm(num_generations=1000,
 ↪  population_size=500, selection_method='tournament',
 ↪  mutation_rate=0.02)
ga_logit.fit(data, target_variable,
 ↪  FUN=linear_model.LogisticRegression(max_iter=1000))
selected_variables_logit, best_fitness_logit = ga_logit.run()

print("Selected Variables:", selected_variables_logit)
print("Best Fitness (AIC):", best_fitness_logit)
```

```
Selected Variables: ['MAJOR_AXIS', 'ECCENTRICITY', 'ASPECT_RATIO', 'ROUNDNESS', 'COMPACTNESS
Best Fitness (AIC): -409.5175136560569
```

Genetic Algorithm with Custom Genetic Operators

Our algorithm can also allow user-defined genetic operators besides the basic crossover and mutation. Users can change the functional of crossover operators and mutation operators, as well as pass in addtional well-defined operators. The operators shall operate on one or two parents to generate offsprings.

Below is an examply using self-defined `unifrom_crossover` and `cauchy_mutation`.

```
def uniform_crossover(parent1, parent2, probability=0.5):
    """
    Apply uniform crossover to two binary parents.

    Parameters:
    - parent1: The first parent.
    - parent2: The second parent.
    - probability: Probability of selecting a gene from the first
 ↪  parent.

    Returns:
    - Two offspring generated by uniform crossover.
    """
    # Ensure both parents have the same length
    assert len(parent1) == len(parent2), "Parents must have the same
     ↪  length"

    # Generate a mask of random values with the same length as the
     ↪  parents
    mask = np.random.rand(len(parent1)) < probability

    # Create offspring by selecting genes based on the mask
    offspring1 = np.where(mask, parent1, parent2)
    offspring2 = np.where(mask, parent2, parent1)

    return offspring1, offspring2

def cauchy_mutation(individual, scale=0.1, alpha=0.5):
    """
    Apply Cauchy mutation to an individual.

    Parameters:
    - individual: The individual to be mutated.
    - scale: The scale parameter controlling the spread of the Cauchy
 ↪  distribution.
    - alpha: The location parameter of the Cauchy distribution.

    Returns:
    - Mutated individual.
    """
    mutation = scale * np.tan(np.pi * (np.random.rand(len(individual)) -
 ↪  0.5))
    mutated_individual = individual + alpha * mutation
    mutated_individual = abs(np.round(mutated_individual))
    return np.where(mutated_individual>1, 1, mutated_individual)

# Genetic Algorithm with Custom Genetic Operator Function
print("Running GA with two custom genetic operator functions...")
ga_opr_custom = GA.GeneticAlgorithm(num_generations=1000,
 ↪  population_size=500, selection_method='tournament',
 ↪  mutation_rate=0.02, opr_fun=[uniform_crossover, cauchy_mutation],
 ↪  opr_para=[2,1])
ga_opr_custom.fit(data, target_variable, FUN=sm.OLS)
selected_variables_opr_custom, best_fitness_opr_custom =
 ↪  ga_opr_custom.run()
print("GA(Custom operators): Selected Variables:",
 ↪  selected_variables_opr_custom)
print("Best Fitness (Custom operators):", best_fitness_opr_custom)
```

```
Running GA with two custom genetic operator functions...
GA(Custom operators): Selected Variables: ['AREA', 'EQDIASQ', 'SOLIDITY', 'CONVEX_AREA', 'EX
Best Fitness (Custom operators): 373.62927686983534
```

Results

Dataset	Lasso Variables	Lasso AIC	Greedy Variables	Greedy AIC	GA Variables	GA AIC
Baseball	10	4423.1434	20	4422.8136	15	4421.6699
Admission	4	-2171.6416	7	-2226.4030	6	-2218.0829
Wine	1	-730.1851	7	-1380.7885	9	-1378.2456
Simulated	10	-714.2309	10	-714.2309	11	-406.4462
Date					18	-456.0749
Fruit						

Conclusion from Testing

The testing phase successfully validated the GA's effectiveness in variable selection across different scenarios and models.

Conclusion

The Genetic Algorithm (GA) can be widely applied for efficient variable selection in linear regression and Generalized Linear Models (GLMs). Our GA effectively addresses the challenges of traditional variable selection methods, offering a robust and flexible solution.

Contributions

- **Fangyuan**: Led the algorithm design and implementation, integrating the GA with various regression models.Gathering of test data and implementation of felxible genetic operators.
- **Hannah**: Focused on testing and documentation, ensuring the reliability and usability of the GA. Implemented AIC alternatives and tested GA against Lasso and the greedy algorithm.
- **Jiarong**: Implemented selction function, focused on pytests and writing the final report.