Assignment-8: Genetic Algorithm

Rezwan-Ul-Alam (ID: 2011659042)
Md. Nur Alam Jowel (ID: 2012355042)
Raian Ruku (ID: 2013409642)

Email addresses:

rezwan.alam1@northsouth.edu
alam.jowel@northsouth.edu
raian.ruku@northsouth.edu

I. Introduction

In this assignment, we implemented and compared three machine learning models such as Decision Tree, Support Vector Classifier (SVC), and Logistic Regression using genetic algorithms for hyperparameter tuning. The objective was to classify cars into different categories based on various features such as price, maintenance, number of doors, capacity, size of luggage boot, and estimated safety.

II. Methodology

A. *Data Preprocessing and Scaling*

The dataset was first loaded and examined for any issues. This included checking for null values, duplicate rows, and performing label encoding for categorical variables. The dataset consisted of the following columns: 'Price', 'Maintenance', 'Number of Doors', 'Capacity', 'Size of Luggage Boot', 'Estimated Safety', and 'Classes'.

B. *Data Loading*

First we load the dataset and see the shape of the data.

```
data = pd.read_csv('/content/gdrive/MyDrive/data/car.csv', header = None)
print("Shape of the Data: ", data.shape)
```

C. *Renaming Columns and Data Cleaning*

```
data.columns = ['Price', 'Maintenance', 'Number of Doors', 'Capacity',
'Size of Luggage Boot', 'Estimated Safety', 'Classes']
data = data.drop_duplicates()
```

D. *Label Encoding*

A label encoder is a preprocessing techniques commonly used in machine learning to convert categorical data into numerical format. When dealing with categorical data many machine learning algorithm requires numerical input. Label encoders converts these categorical values into numerical labels. In our case we use label encoding.

Categorical variables were encoded as follows:

```
data.classes.replace(('unacc', 'acc', 'good', 'vgood'), (0, 1, 2, 3), inplace = True)
data['Size␣of␣Luggage␣Boot'].replace(('small', 'med', 'big'), (0, 1, 2),
inplace = True)
data['Estimated␣Safety'].replace(('low', 'med', 'high'), (0, 1, 2), inplace = True)
data['Price'].replace(('low', 'med', 'high', 'vhigh'), (0, 1, 2, 3), inplace = True)
data['Maintenance'].replace(('low', 'med', 'high', 'vhigh'), (0, 1, 2, 3),
inplace = True)
data['Number␣of␣Doors'].replace('5more', 5, inplace = True)
data['Capacity'].replace('more', 5, inplace = True)
```

*E. **Train-Test Split***

As per the instruction, the first 70% of the data was used for training, and 15% of the data was used for validation and last 15% of the data is used for testing. The code is given below:

```
X = data.iloc[:, :6]
y = data.iloc[:, 6]
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,
random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
random_state=42)
```

*F. **Standard Scaling***

The StandardScaler in scikit-learn is used for standardizing features by removing the mean and scaling them to unit variance.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

III. Genetic Algorithm for Hyperparameter Tuning

A genetic algorithm was employed to optimize hyperparameters for each model.

*A. **Decision Tree***

Hyperparameters and Initialization

The hyperparameters for the Decision Tree included 'max_depth', 'min_samples_split', and 'min_samples_leaf'. The population was initialized with random values within specified ranges. Here is the code for hyperparameters and initializations.

```
hyperparameter_space = {
    'max_depth': [3, 10],
    'min_samples_split': [2, 10],
    'min_samples_leaf': [1, 5]
}
def initialize_population(size):
    population = []
    for _ in range(size):
        chromosome = {
            'max_depth': np.random.randint(hyperparameter_space['max_depth'][0],
            hyperparameter_space['max_depth'][1]),
            'min_samples_split': np.random.randint
            (hyperparameter_space['min_samples_split'][0],
            hyperparameter_space['min_samples_split'][1]),
            'min_samples_leaf': np.random.randint
            (hyperparameter_space['min_samples_leaf'][0],
            hyperparameter_space['min_samples_leaf'][1])
        }
        population.append(chromosome)
    return population
```

Fitness Function and Evolutionary Process

The fitness function measured model accuracy on the validation set. The genetic algorithm then performed selection (selecting the top-performing chromosomes to create the next generation.), crossover (combining parts of two parents to create offspring.), and mutation (introducing random changes to chromosomes to maintain genetic diversity) over 10 generations to find the best hyperparameters. Here is the code in the next page:

```
def fitness_function(chromosome, X_train, y_train, X_val, y_val):
    model = DecisionTreeClassifier(
        max_depth=chromosome['max_depth'],
        min_samples_split=chromosome['min_samples_split'],
        min_samples_leaf=chromosome['min_samples_leaf']
    )
    model.fit(X_train, y_train)
    y_pred = model.predict(X_val)
    return accuracy_score(y_val, y_pred)
def selection(population, fitnesses):
    sorted_population = [chromosome for _,
    chromosome in sorted(zip(fitnesses, population), key=lambda x: x[0], reverse=True)]
    return sorted_population[:len(population)//2]
def crossover(parent1, parent2):
    child1 = {
        'max_depth': parent1['max_depth'],
        'min_samples_split': parent2['min_samples_split'],
        'min_samples_leaf': parent1['min_samples_leaf']
    }
    child2 = {
        'max_depth': parent2['max_depth'],
        'min_samples_split': parent1['min_samples_split'],
        'min_samples_leaf': parent2['min_samples_leaf']
    }
    return child1, child2
def mutate(chromosome):
    if np.random.rand() < 0.1:
        chromosome['max_depth'] = np.random.randint
        (hyperparameter_space['max_depth'][0], hyperparameter_space['max_depth'][1])
    if np.random.rand() < 0.1:
        chromosome['min_samples_split'] = np.random.randint
        (hyperparameter_space['min_samples_split'][0],
        hyperparameter_space['min_samples_split'][1])
    if np.random.rand() < 0.1:
        chromosome['min_samples_leaf'] = np.random.randint
        (hyperparameter_space['min_samples_leaf'][0],
        hyperparameter_space['min_samples_leaf'][1])
    return chromosome
def genetic_algorithm(X_train, y_train, X_val, y_val, population_size=10,
generations=10):
    population = initialize_population(population_size)
    for generation in range(generations):
        fitnesses = [fitness_function(chromosome, X_train, y_train, X_val, y_val)
        for chromosome in population]
        print(f"Generation␣{generation},␣Best␣Fitness:␣{max(fitnesses)}")
        population = selection(population, fitnesses)
        next_population = []
        while len(next_population) < population_size:
            parent1, parent2 = np.random.choice(population, 2)
            child1, child2 = crossover(parent1, parent2)
            next_population.append(mutate(child1))
            next_population.append(mutate(child2))
        population = next_population
    best_chromosome = max(population, key=lambda
    c: fitness_function(c, X_train, y_train, X_val, y_val))
    return best_chromosome
```

B. Support Vector Classifier (SVC)

Hyperparameters and Initialization

For the SVC model, hyperparameters included 'C', 'kernel', 'degree', and 'gamma'. The population was similarly initialized with random values within specified ranges. Here is the code for hyperparameter and initializations.

```
hyperparameter_space = {
    'C': [0.01, 10],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': [2, 5],
    'gamma': ['scale', 'auto']
}

def initialize_population(size):
    population = []
    for _ in range(size):
        chromosome = {
            'C': np.random.uniform(hyperparameter_space['C'][0],
            hyperparameter_space['C'][1]),
            'kernel': np.random.choice(hyperparameter_space['kernel']),
            'degree': np.random.randint(hyperparameter_space['degree'][0],
            hyperparameter_space['degree'][1]),
            'gamma': np.random.choice(hyperparameter_space['gamma'])
        }
        population.append(chromosome)
    return population
```

Fitness Function and Evolutionary Process

The fitness function measured model accuracy on the validation set. The genetic algorithm then performed selection (selecting the top-performing chromosomes to create the next generation.), crossover (combining parts of two parents to create offspring.), and mutation (introducing random changes to chromosomes to maintain genetic diversity) over 10 generations to find the best hyperparameters. Here is the code:

```
def selection(population, fitnesses):
    sorted_population = [chromosome for _,
    chromosome in sorted(zip(fitnesses, population), key=lambda x: x[0], reverse=True)]
    return sorted_population[:len(population)//2]
def crossover(parent1, parent2):
    child1 = {
        'C': parent1['C'],
        'kernel': parent2['kernel'],
        'degree': parent1['degree'],
        'gamma': parent2['gamma']
    }
    child2 = {
        'C': parent2['C'],
        'kernel': parent1['kernel'],
        'degree': parent2['degree'],
        'gamma': parent1['gamma']
    }
    return child1, child2

def mutate(chromosome):
    if np.random.rand() < 0.1:
        chromosome['C'] = np.random.uniform
        (hyperparameter_space['C'][0], hyperparameter_space['C'][1])
    if np.random.rand() < 0.1:
        chromosome['kernel'] = np.random.choice
        (hyperparameter_space['kernel'])
    if np.random.rand() < 0.1:
        chromosome['degree'] = np.random.randint
        (hyperparameter_space['degree'][0], hyperparameter_space['degree'][1])
    if np.random.rand() < 0.1:
        chromosome['gamma'] = np.random.choice(hyperparameter_space['gamma'])
    return chromosome

def genetic_algorithm(X_train, y_train, X_val, y_val, population_size=10,
generations=10):
    population = initialize_population(population_size)
    for generation in range(generations):
        fitnesses = [fitness_function(chromosome, X_train, y_train, X_val, y_val)
        for chromosome in population]
        print(f"Generation␣{generation},␣Best␣Fitness:␣{max(fitnesses)}")
        population = selection(population, fitnesses)
        next_population = []
        while len(next_population) < population_size:
            parent1, parent2 = np.random.choice(population, 2)
            child1, child2 = crossover(parent1, parent2)
            next_population.append(mutate(child1))
            next_population.append(mutate(child2))
        population = next_population
    best_chromosome = max(population, key=lambda
    c: fitness_function(c, X_train, y_train, X_val, y_val))
    return best_chromosome
```

C. *Logistic Regression*

Hyperparameters and Initialization

The Logistic Regression model's hyperparameters included 'C' and 'max_iter'. The genetic algorithm was used to optimize these parameters. Here is the code for hyperparameter and initializations.

```
hyperparameter_space = {
    'C': [0.01, 10],
    'max_iter': [100, 500]
}
def initialize_population(size):
    population = []
    for _ in range(size):
        chromosome = {
            'C': np.random.uniform(hyperparameter_space['C'][0],
            hyperparameter_space['C'][1]),
            'max_iter': np.random.randint(hyperparameter_space['max_iter'][0],
            hyperparameter_space['max_iter'][1])
        }
        population.append(chromosome)
    return population
```

Fitness Function and Evolutionary Process

The fitness function measured model accuracy on the validation set. The genetic algorithm then performed selection (selecting the top-performing chromosomes to create the next generation.), crossover (combining parts of two parents to create offspring.), and mutation (introducing random changes to chromosomes to maintain genetic diversity) over 10 generations to find the best hyperparameters. Here is the code:

```
def fitness_function(chromosome, X_train, y_train, X_val, y_val):
    model = LogisticRegression(
        C=chromosome['C'],
        max_iter=chromosome['max_iter'],
        solver='lbfgs',
        multi_class='auto'  # detect multi-class classification automatically
    )
    model.fit(X_train, y_train)
    y_pred = model.predict(X_val)
    return accuracy_score(y_val, y_pred)
def selection(population, fitnesses):
    sorted_population = [chromosome for _,
    chromosome in sorted(zip(fitnesses, population), key=lambda x: x[0], reverse=True)]
    return sorted_population[:len(population)//2]
def crossover(parent1, parent2):
    child1 = {
        'C': parent1['C'],
        'max_iter': parent2['max_iter']
    }
    child2 = {
        'C': parent2['C'],
        'max_iter': parent1['max_iter']
    }
    return child1, child2
def mutate(chromosome):
    if np.random.rand() < 0.1:
        chromosome['C'] = np.random.uniform(hyperparameter_space['C'][0],
        hyperparameter_space['C'][1])
    if np.random.rand() < 0.1:
        chromosome['max_iter'] = np.random.randint(hyperparameter_space['max_iter'][0],
        hyperparameter_space['max_iter'][1])
    return chromosome

def genetic_algorithm(X_train, y_train, X_val, y_val, population_size=10,
generations=10):
    population = initialize_population(population_size)
    for generation in range(generations):
        fitnesses = [fitness_function(chromosome, X_train, y_train, X_val, y_val)
        for chromosome in population]
        print(f"Generation␣{generation},␣Best␣Fitness:␣{max(fitnesses)}")
        population = selection(population, fitnesses)
        next_population = []
        while len(next_population) < population_size:
            parent1, parent2 = np.random.choice(population, 2)
            child1, child2 = crossover(parent1, parent2)
            next_population.append(mutate(child1))
            next_population.append(mutate(child2))
        population = next_population
    best_chromosome = max(population, key=lambda
    c: fitness_function(c, X_train, y_train, X_val, y_val))
    return best_chromosome
```

IV. Evaluation

Evaluation Metrics

The models were evaluated using accuracy, precision, recall, F1-score, and confusion matrix. Custom functions were written to calculate these metrics.

- **Accuracy:** accuracy is the ratio of the number of correctly classified instances (true positives and true negatives) to the total number of instances in the dataset.
- **Precision:**Precision measures the proportion of true positive predictions among all positive predictions made by the classifier. It answers the question: "Out of all the instances predicted as positive, how many are actually positive?"
- **Recall:** Recall measures the proportion of true positive predictions among all actual positive instances in the dataset. It answers the question: "Out of all the actual positive instances, how many were correctly predicted as positive?"
- **F1-score:** The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall. F1-score reaches its best value at 1 and worst at 0.
- **Confusion matrix** A confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. It presents the count of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions made by the classifier.

  Calculation process:: *here for calculating accuracy, precision, recall, F1-score and confusion matrix we use our own created function. In below we provide all the created function with necessary equation.*

  NOTE here,

  TP (True positive): Number of correctly predicted positive instances.
  TN (True negative): Number of correctly predicted negative instances.
  FP (False positive): Number of incorrectly predicted positive instances.
  FN (False negative): Number of incorrectly predicted negative instances.

*1) **Accuracy [**$(TP+TN)/(TP+TN+FP+FN)$**]** :* Here the function calculate_accuracy(y_true, y_pred) essentially computes the accuracy metric based on the provided true labels and predicted labels. It counts the number of correct predictions and divides it by the total number of predictions to obtain the accuracy value.

```
  def calculate_accuracy(y_true, y_pred):
.       correct_predictions = sum(y_true == y_pred)
.       total_predictions = len(y_true)
.       accuracy = correct_predictions / total_predictions
.       return accuracy
```

*2) **Precision [**$TP/(TP+FP)$**]** :* Here, this function calculate_precision(y_true, y_pred, label):takes the true labels, predicted labels, and a specific label for which precision is calculated. It then computes and returns the precision value for that label. If there are no predicted positives, it returns 0 to avoid division by zero.

```
def calculate_precision(y_true, y_pred, label):
    true_positives = sum((y_true == label) & (y_pred == label))
    predicted_positives = sum(y_pred == label)
    if predicted_positives > 0:
      precision = true_positives / predicted_positives
    else :0
    return precision
```

*3) **Recall [**$TP/(TP+FN)$ **]:*** Here, The function calculate_recall(y_true, y_pred, label): essentially computes the recall metric for a given label based on the true and predicted labels provided. It ensures consistence by handling the scenario where there are no actual positive instances for the specified label.

```
def calculate_recall(y_true, y_pred, label):
    true_positives = sum((y_true == label) & (y_pred == label))
    actual_positives = sum(y_true == label)
    if actual_positives > 0:
      recall = true_positives / actual_positives
    else :0
    return recall
```

*4) **F1-Score [**$2*(precision*recall)/(precicion+recall)$**] :*** Here, the function calculate_f1_score(precision, recall): essentially computes the F1-score metric based on the provided precision and recall values. It ensures consistence by handling the scenario where both precision and recall are zero to avoid division by zero.

```
def calculate_f1_score(precision, recall):
    if (precision + recall) > 0 :
      f1_score = 2 * (precision * recall) / (precision + recall)
    else :0
    return f1_score
```

*5) **Accuracy [**$(TP+TN)/(TP+TN+FP+FN)$**] :***

```
  def calculate_accuracy(y_true, y_pred):
      correct_predictions = sum(y_true == y_pred)
      total_predictions = len(y_true)
      accuracy = correct_predictions / total_predictions
      return accuracy
```

V. Results

A. *Decision Tree*

Best Hyperparameters:

Code:

```
best_params = genetic_algorithm(X_train_scaled, y_train, X_val_scaled, y_val)
print("Best hyperparameters: ", best_params)

output:{'max_depth': 9, 'min_samples_split': 4, 'min_samples_leaf': 1}
```

B. *Support Vector Classifier (SVC)*

Best Hyperparameters:

Code:

```
best_params = genetic_algorithm(X_train_scaled, y_train, X_val_scaled, y_val)
print("Best hyperparameters: ", best_params)

Output: {'C': 2.173098547427915, 'kernel': 'rbf', 'degree': 4, 'gamma': 'scale'}
```

C. *Logistic Regression*

Best Hyperparameters:

Code:

```
best_params = genetic_algorithm(X_train_scaled, y_train, X_val_scaled, y_val)
print("Best hyperparameters: ", best_params)

Output: {'C': 3.640320489274689, 'max_iter': 457}
```

Here is the Evaluation metrics Table For all three classifier:

Classifier	Mertic	overall	Class 0	Class 1	Class 2	Class 3	Confusion Matrix
Decision Tree	Accuracy	0.95					[177, 3, 0, 0]
	Precision		0.99	0,94	0.61	0.81	[1, 55, 4, 1]
	Recall		0.98	0.90	0.88	0.9	[0, 0, 8, 1]
	F1 Score		0.98	0.92	0.72	0.85	[0, 0, 1, 9]
Support Vector Classifier	Accuracy	0.97					[179, 1, 0, 0]
	Precision		0.98	0.96	0.75	1.0	[2, 56, 3, 0]
	Recall		0.99	0.91	1.0	0.9	[0, 0, 9, 0]
	F1 Score		0.99	0.94	0.85	0.94	[0, 1, 0, 9]
Logistic Regression	Accuracy	0.84					[167, 11, 2, 0]
	Precision		0.89	0.70	0.57	0.8	[19, 40, 1, 1]
	Recall		0.92	0.65	0.44	0.8	[19, 40, 1, 1]
	F1 Score		0.91	0.67	0.5	0.80	[0, 2, 0, 8]

VI. Discussion

A. *Accuracy:*

The SVC model achieved the highest GA-tuned accuracy (0.97), followed by the Decision Tree (0.95) and Logistic Regression (0.84).

B. *Precision:*

The SVC model showed the highest precision across most classes, with particularly notable performance for class 3 (1.0) and class 0 (0.98). The Decision Tree also performed well, especially for class 0 (0.99), but had a lower precision for class 2 (0.61). Logistic Regression had lower precision overall, particularly for class 2 (0.57).

C. *Recall:*

SVC had the highest recall overall, particularly excelling in class 2 (1.0) and class 0 (0.99). The Decision Tree showed strong recall for class 2 (0.88) and class 0 (0.98), but lower for class 1 (0.90). Logistic Regression had the lowest recall for class 2 (0.44), indicating it missed many true instances of this class.

D. *F1 Scores:*

The SVC model had consistently high F1 scores, indicating a balanced performance between precision and recall, with class 3 achieving 0.94 and class 0 achieving 0.99. The Decision Tree had high F1 scores for most classes but was lower for class 2 (0.72). Logistic Regression's F1 scores were notably lower for class 2 (0.50), reflecting its struggle with this class.

E. *Confusion Matrix:*

The SVC confusion matrix indicated fewer misclassifications overall, particularly excelling in correctly classifying instances of class 2 and class 0. The Decision Tree had more misclassifications, especially for class 2. Logistic Regression showed higher misclassification rates, particularly in class 2 and class 1.