In [1]:

```python
# Graphs
# Breadth First Search --> Analogous to Level orders traversal
def bfs(graph,start):
    visited=set()
    queue=[start]
    visited.add(start)
    while queue:
        vertex=queue.pop(0)
        print(vertex,end=" ")
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Depth First Search --> Analogous to inorder,preorder and postorder traversals
def dfs(graph,start):
    visited=set()
    stack=[start]
    while stack:
        vertex=stack.pop()
        if vertex not in visited:
            print(vertex,end=" ")
            visited.add(vertex)
            stack.extend(reversed(graph[vertex]))

# Example usage
graph={"A":["B","C"],"B":["A","D","E"],"C":["A","F"],
       "D":["B"],"E":["B","F"],"F":["C","E"]}
start_vertex="A"
print("Breadth First Traversal: ",end="")
bfs(graph,start_vertex)
print()
print("Depth First Traversal: ",end="")
dfs(graph,start_vertex)
```

```
Breadth First Traversal: A B C D E F
Depth First Traversal: A B D E F C
```

In [2]:
```python
# Best First Search
from queue import PriorityQueue

# Graph represented as an adjacency list
graph = {
    0: [(1, 1), (2, 2), (3, 3)],
    1: [(4, 4)],
    2: [(5, 5)],
    3: [(6, 6)],
    4: [(7, 3)],
    5: [(7, 2)],
    6: [(7, 1)],
    7: []
}

def best_first_search(source, target):
    visited = set()
    pq = PriorityQueue()  # Priority queue to explore nodes by lowest cost first
    pq.put((0, source))   # Start with the source node (priority, node)
    while not pq.empty():
        cost, node = pq.get()  # Get node with the lowest cost
        if node in visited:
            continue
        print(node, end=" ")   # Print the current node
        visited.add(node)
        if node == target:     # Stop if the target is reached
            break
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                pq.put((weight, neighbor))  # Add neighbors to the queue with their cost

# Run Best First Search
source = 0
target = 7
best_first_search(source, target)
```

```
0 1 2 3 4 7
```

In [3]:
```python
# A* Search Algorithm
from queue import PriorityQueue

def a_star(graph, heuristics, start, goal):
    pq = PriorityQueue()  # Priority queue for A* (min-heap based on f-cost)
    pq.put((0, start))  # Start node with f-cost 0
    came_from = {start: None}  # Track the path (parent nodes)
    g_cost = {start: 0}  # Cost from start to the current node (g-cost)

    while not pq.empty():
        current_f_cost, current_node = pq.get()

        if current_node == goal:  # Goal reached
            path = []
            while current_node:
                path.append(current_node)
                current_node = came_from[current_node]
            return path[::-1]  # Return reversed path from start to goal

        # Explore neighbors
        for neighbor, cost in graph[current_node]:
            new_g_cost = g_cost[current_node] + cost
            if neighbor not in g_cost or new_g_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_g_cost
                f_cost = new_g_cost + heuristics[neighbor]  # f(n) = g(n) + h(n)
                pq.put((f_cost, neighbor))
                came_from[neighbor] = current_node

    return None  # No path found

# Graph (Adjacency List)
graph = {
   'A':[('B',2),('E',3)],
   'B':[('C',1),('G',9)],
   'C':None,
   'E':[('D',6)],
   'D':[('G',1)]
}

# Heuristic (h-cost) for each node (estimated cost to goal)
heuristics = {
```

```python
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }

# Run A* search
start = 'A'
goal = 'G'
path = a_star(graph, heuristics, start, goal)
print("Path found:", path)
```

```
Path found: ['A', 'E', 'D', 'G']
```

In [4]:
```python
#AO* Algorithm
def calculate_cost(H, condition, weight=1):
    total_cost = 0
    # Calculate AND conditions cost
    if 'AND' in condition:
        total_cost += sum(H[node] + weight for node in condition['AND'])

    # Calculate OR conditions cost (minimum of all OR nodes)
    if 'OR' in condition:
        or_cost = min(H[node] + weight for node in condition['OR'])
        total_cost += or_cost

    return total_cost

def find_shortest_path(start, H, conditions, weight=1):
    path = start
    if start in conditions:
        condition = conditions[start]
        # Calculate the cost directly while finding the path
        cost = calculate_cost(H, condition, weight)
        H[start] = cost  # Update heuristic for the node

        # Process OR paths
        if 'OR' in condition:
            next_node = condition['OR'][0]  # Take the first OR node
            path += f' <-- {find_shortest_path(next_node,H,conditions,weight)}'

        # Process AND paths
        if 'AND' in condition:
            and_nodes = condition['AND']
            path += f' <-- (AND: {", ".join(and_nodes)})'
            for and_node in and_nodes:
                path += f' + {find_shortest_path(and_node,H,conditions,weight)}'

    return path.strip()

# Heuristic values
H = {'A': -1, 'B': 4, 'C': 2, 'D': 3, 'E': 6,
     'F': 8, 'G': 2, 'H': 0, 'I': 0, 'J': 0}
# Conditions representing the graph structure (AND/OR)
conditions = {
```

```
    'A': {'OR': ['B'], 'AND': ['C', 'D']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': ['H', 'I']},
    'D': {'OR': ['J']}
}

# Weight for cost calculation
weight = 1

# Shortest Path Calculation
print('Shortest Path:')
print(find_shortest_path('A', H, conditions, weight))
```

```
Shortest Path:
A <-- B <-- E <-- (AND: C, D) + C <-- G <-- (AND: H, I) + H + I + D <-- J
```

In [51]:

```
#Part A: Exploratory Data Analysis (EDA) using Python
#Step 1: Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
```

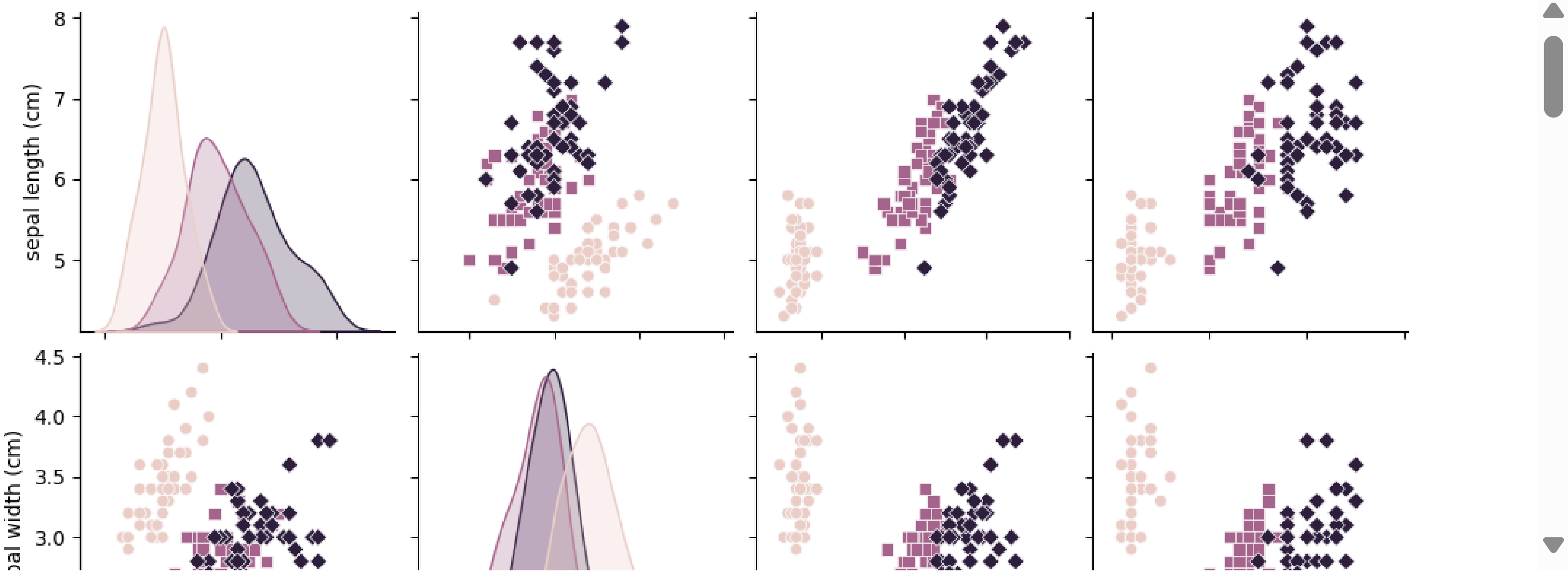
In [52]:
```
#Step 2: Load Dataset
# Load the Iris dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris. feature_names)
data['target' ] = iris.target
print(data.head())
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
3       0
4       0
```

In [53]:
```
#Step 3: Data Overview
# Display basic information about the dataset
print(data.info())
print(data.describe())

# Check for missing values
print(data.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int32  
dtypes: float64(4), int32(1)
memory usage: 5.4 KB
None
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000   
mean            5.843333          3.057333           3.758000   
std             0.828066          0.435866           1.765298   
min             4.300000          2.000000           1.000000   
25%             5.100000          2.800000           1.600000   
50%             5.800000          3.000000           4.350000   
75%             6.400000          3.300000           5.100000   
max             7.900000          4.400000           6.900000   

       petal width (cm)      target  
count        150.000000  150.000000  
mean           1.199333    1.000000  
std            0.762238    0.819232  
min            0.100000    0.000000  
25%            0.300000    0.000000  
50%            1.300000    1.000000  
75%            1.800000    2.000000  
max            2.500000    2.000000  
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
target               0
dtype: int64
```

In [54]:
```
#Step 4: Data Visualization
# Pairplot to visualize relationships between features
sns.pairplot(data, hue='target', markers=["o", "s", "D"])
plt.show()

# Boxplot to visualize the distribution of features
plt.figure(figsize=(12, 8))
sns.boxplot(data=data)
plt.show()

# Correlation Heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(data.corr(), annot=True, cmap='coolwarm')
plt.show()
```



In [9]:
```python
#Part B: Model Building in Python
#Step 1: Import Libraries
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix
```

In [10]:
```python
#Step 2: Split Dataset
# Define features and target
X = data.drop('target', axis=1)
y = data['target' ]

# Split the dataset 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)
```

In [11]:
```python
#Step 3: Preprocess Data
# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [12]:
```python
#Step 4: Train Model
# Initialize the RandomForestClassifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model
clf.fit(X_train_scaled, y_train)
```

Out[12]:
```
▾ RandomForestClassifier
RandomForestClassifier(random_state=42)
```

In [13]:

```python
#Step 5: Evaluate Model
# Make predictions
y_pred = clf.predict(X_test_scaled)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

In [14]:
```
#Step 6: Make Predictions
# Making predictions on new data
new_data = np.array([[5.0, 3.6, 1.4, 0.2]])
new_data_scaled = scaler.transform(new_data)
prediction = clf.predict(new_data_scaled)
predicted_class = iris. target_names [prediction]
print(f"Predicted class for the new data: {predicted_class}")
```

```
Predicted class for the new data: ['setosa']
```

```
C:\Users\subha\anaconda3\Lib\site-packages\sklearn\base.py:464: UserWarning: X does not have valid feature names, but
StandardScaler was fitted with feature names
  warnings.warn(
```

In [15]:
```python
#### Binary Classification
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Load the dataset
data = load_breast_cancer()
X = data.data
y = data.target

# Convert to a DataFrame for better visualization (optional)
df = pd.DataFrame(X, columns=data.feature_names)
df['target'] = y

# Data Preprocessing: Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

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

# Train a classification model (Logistic Regression)
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Print the evaluation results
print(f"Accuracy: {accuracy:.2f}")
print("Confusion Matrix:")
```

```python
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

```
Accuracy: 0.97
Confusion Matrix:
[[41  2]
 [ 1 70]]
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.95      0.96        43
           1       0.97      0.99      0.98        71

    accuracy                           0.97       114
   macro avg       0.97      0.97      0.97       114
weighted avg       0.97      0.97      0.97       114
```

In [16]:

```python
### Multi CLassification
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Convert to DataFrame for better readability
df = pd.DataFrame(X, columns=iris.feature_names)
df['target'] = y
print(df.head())

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Initialize the RandomForestClassifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Evaluate the model
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

```python
# Making predictions on new data
new_data = np.array([[5.0, 3.6, 1.4, 0.2]])
new_data_scaled = scaler.transform(new_data)
prediction = clf.predict(new_data_scaled)
predicted_class = iris.target_names[prediction]
print(f"Predicted class for the new data: {predicted_class[0]}")
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
3       0
4       0
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Predicted class for the new data: setosa
```

In [18]:
```python
###Simple Linear Regression
# import all the lib
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

In [19]:
```python
# read the dataset using pandas
data=pd.read_csv('Salary_Data.csv')
```

In [20]:
```python
# This displays the top 5 rows of the data
data.head(5)
```

Out[20]:

	YearsExperience	Salary
0	1.1	39343.0
1	1.3	46205.0
2	1.5	37731.0
3	2.0	43525.0
4	2.2	39891.0

In [21]:
```python
# Provides some information regarding the columns in the data
data.info()
```

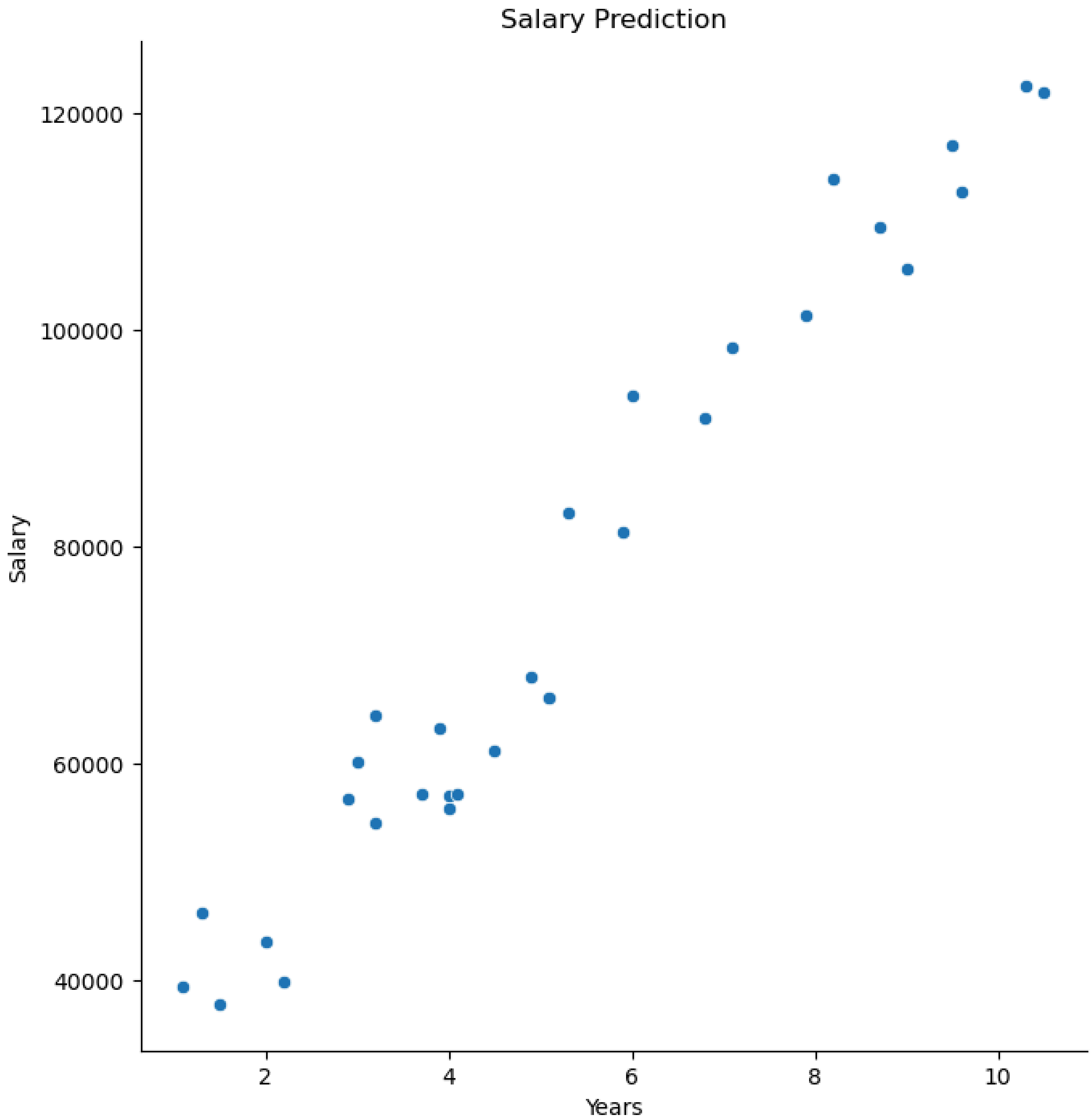
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30 entries, 0 to 29
Data columns (total 2 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   YearsExperience  30 non-null     float64
 1   Salary           30 non-null     float64
dtypes: float64(2)
memory usage: 612.0 bytes
```

In [23]:
```
# This describes the basic stat behind the dataset used
data.describe().T
```

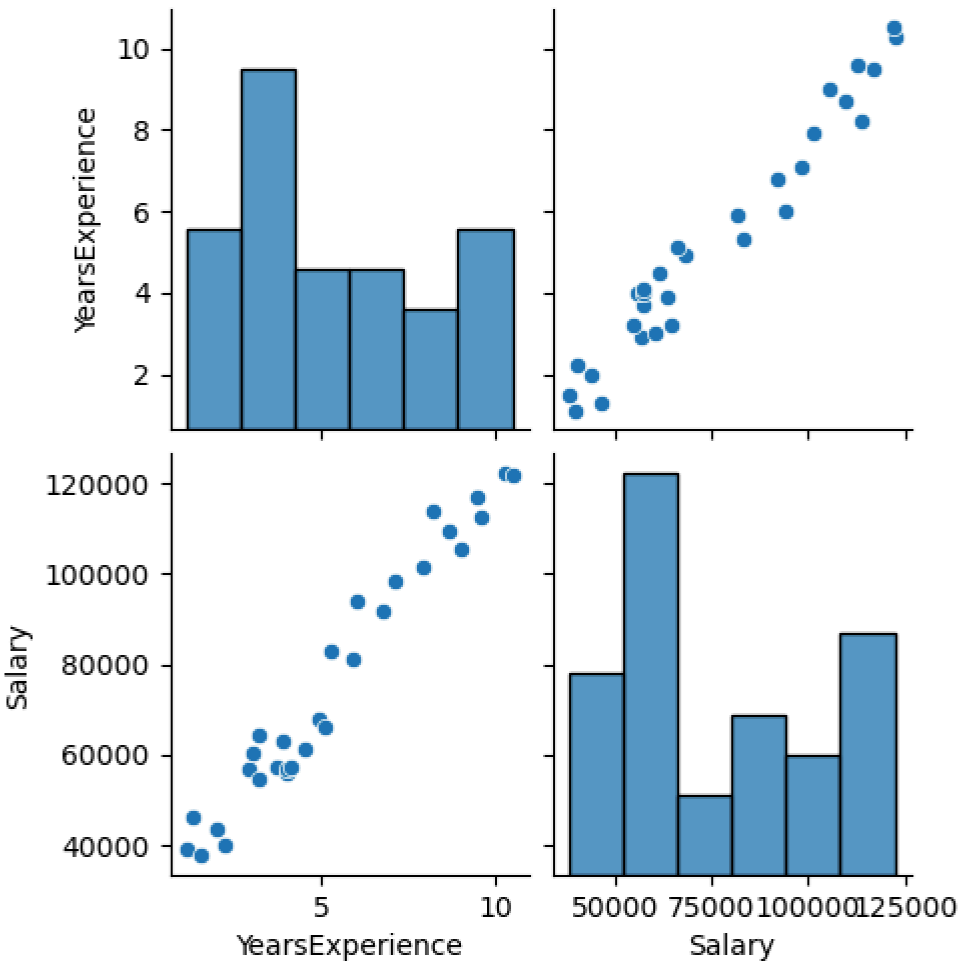
Out[23]:

	count	mean	std	min	25%	50%	75%	max
YearsExperience	30.0	5.313333	2.837888	1.1	3.20	4.7	7.70	10.5
Salary	30.0	76003.000000	27414.429785	37731.0	56720.75	65237.0	100544.75	122391.0

In [25]:
```
sns.pairplot(data,x_vars=['YearsExperience'],y_vars=['Salary'],height=7,kind='scatter')
plt.xlabel('Years')
plt.ylabel('Salary')
plt.title('Salary Prediction')
plt.show()
```


Salary Prediction
Salary
120000
100000
80000
60000
40000
2
4
6
8
10
Years

In [26]:
```
sns.pairplot(data)
```
Out[26]: <seaborn.axisgrid.PairGrid at 0x1fa0646c4d0>



In [27]:
```
# Cooking the data
X=data['YearsExperience']
X.head()
```

Out[27]:
```
0    1.1
1    1.3
2    1.5
3    2.0
4    2.2
Name: YearsExperience, dtype: float64
```

In [28]:
```
# Cooking the data
y=data['Salary']
y.head()
```

Out[28]:
```
0    39343.0
1    46205.0
2    37731.0
3    43525.0
4    39891.0
Name: Salary, dtype: float64
```

In [29]:
```
# Import segragating data for train and test
from sklearn.model_selection import train_test_split
```

In [31]:
```
# Split the data for train and test
X_train, X_test, y_train, y_test = train_test_split(X,y, train_size=0.7,random_state=10)
```

In [32]:
```
# Create new axis for x column
X_train = X_train[:,np.newaxis]
X_test = X_test[:,np.newaxis]
```

```
C:\Users\subha\AppData\Local\Temp\ipykernel_18316\67130142.py:2: FutureWarning: Support for multi-dimensional indexin
g (e.g. `obj[:, None]`) is deprecated and will be removed in a future version.  Convert to a numpy array before index
ing instead.
  X_train = X_train[:,np.newaxis]
C:\Users\subha\AppData\Local\Temp\ipykernel_18316\67130142.py:3: FutureWarning: Support for multi-dimensional indexin
g (e.g. `obj[:, None]`) is deprecated and will be removed in a future version.  Convert to a numpy array before index
ing instead.
  X_test = X_test[:,np.newaxis]
```

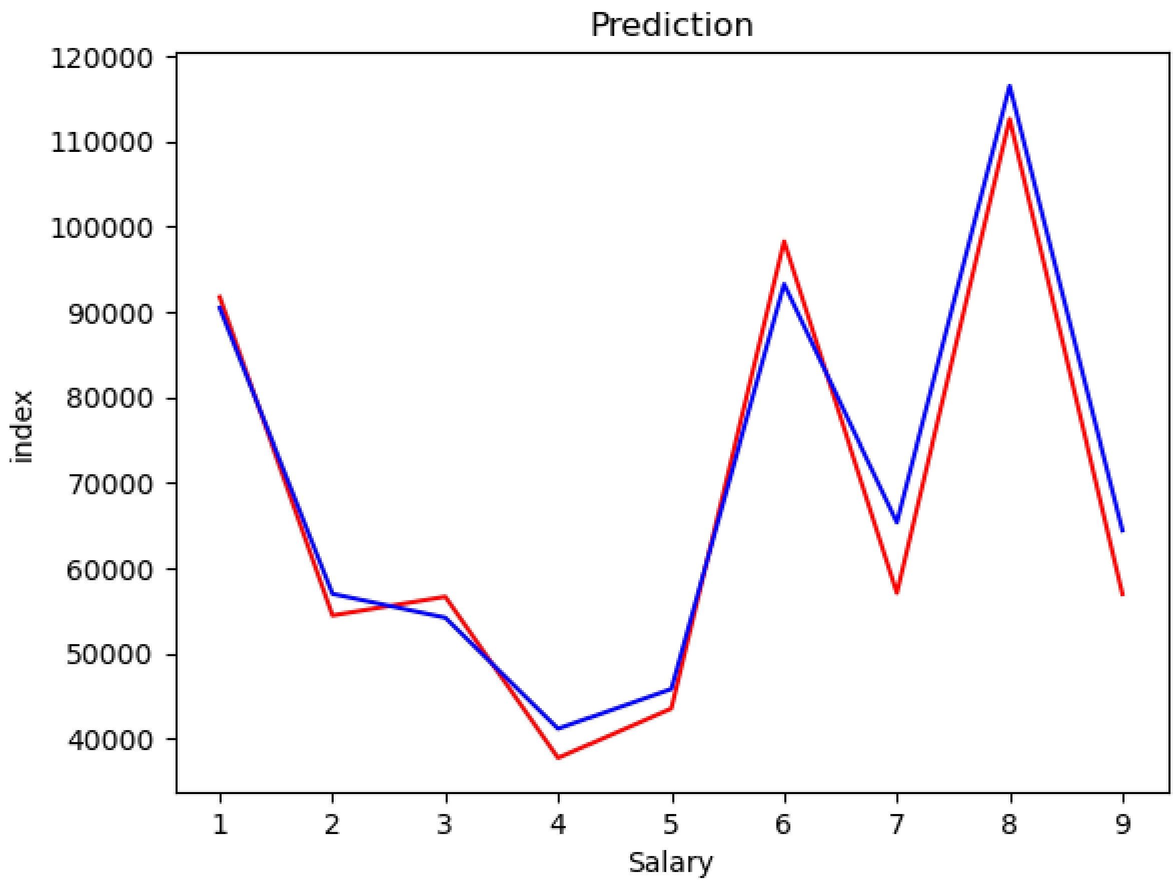
In [33]:
```
# Importing Linear Regression Model form scikit learn
from sklearn.linear_model import LinearRegression
```

In [34]:
```
# Fitting the model
lr=LinearRegression()
lr.fit(X_train,y_train)
```

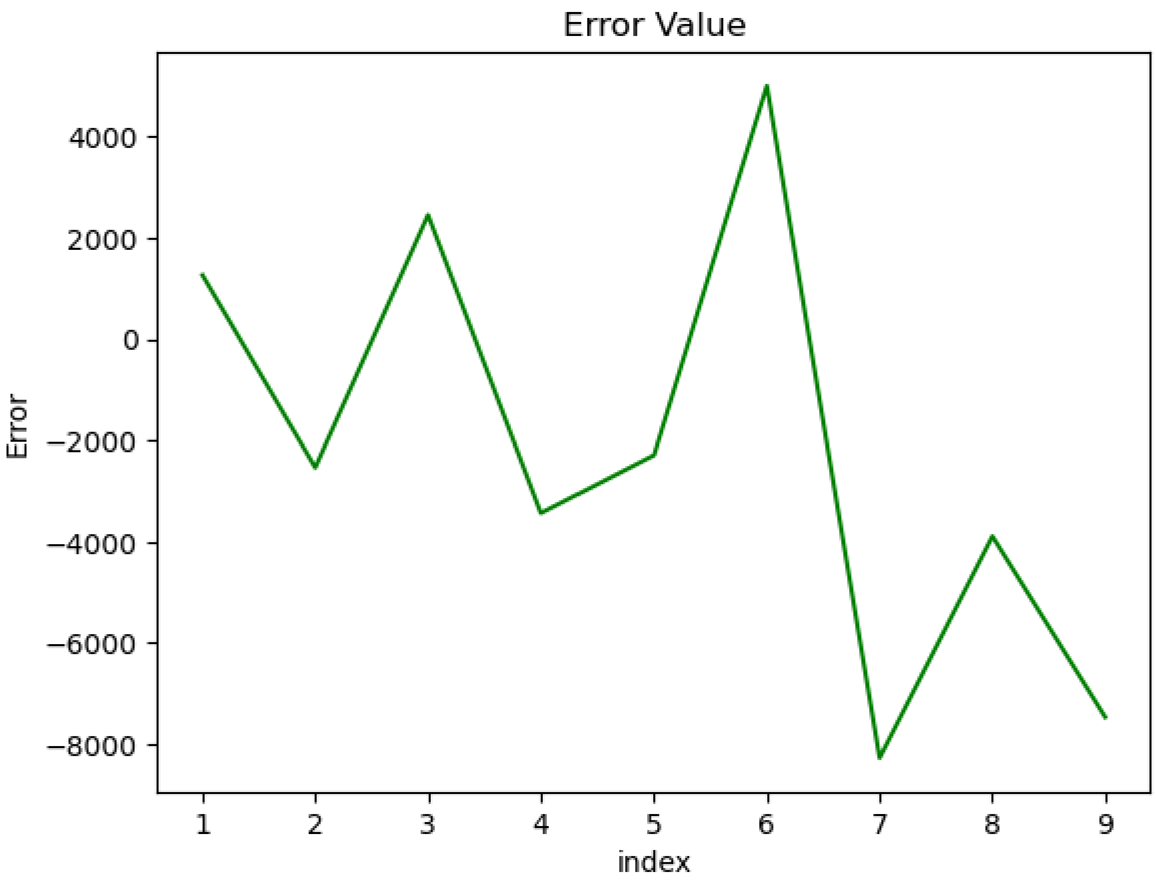
Out[34]:
```
▾ LinearRegression

LinearRegression()
```

In [35]:
```
# Predicting the salaries for the Test values
y_pred=lr.predict(X_test)
```

In [36]:
```
# Plotting the actual and predicted values
c = [i for i in range (1,len(y_test)+1,1)]
plt.plot(c,y_test, color='r',linestyle='-')
plt.plot(c,y_pred, color='b',linestyle='-')
plt.xlabel('Salary')
plt.ylabel('index' )
plt.title('Prediction')
plt.show()
```



In [37]:
```
# plotting the error
c = [i for i in range(1,len(y_test)+1,1)]
plt.plot(c,y_test-y_pred, color='green' , linestyle='-')
plt.xlabel('index')
plt.ylabel('Error')
plt.title('Error Value' )
plt.show()
```



In [38]:
```
# Importing metrics for the evaluation of the model
from sklearn.metrics import r2_score,mean_squared_error
```

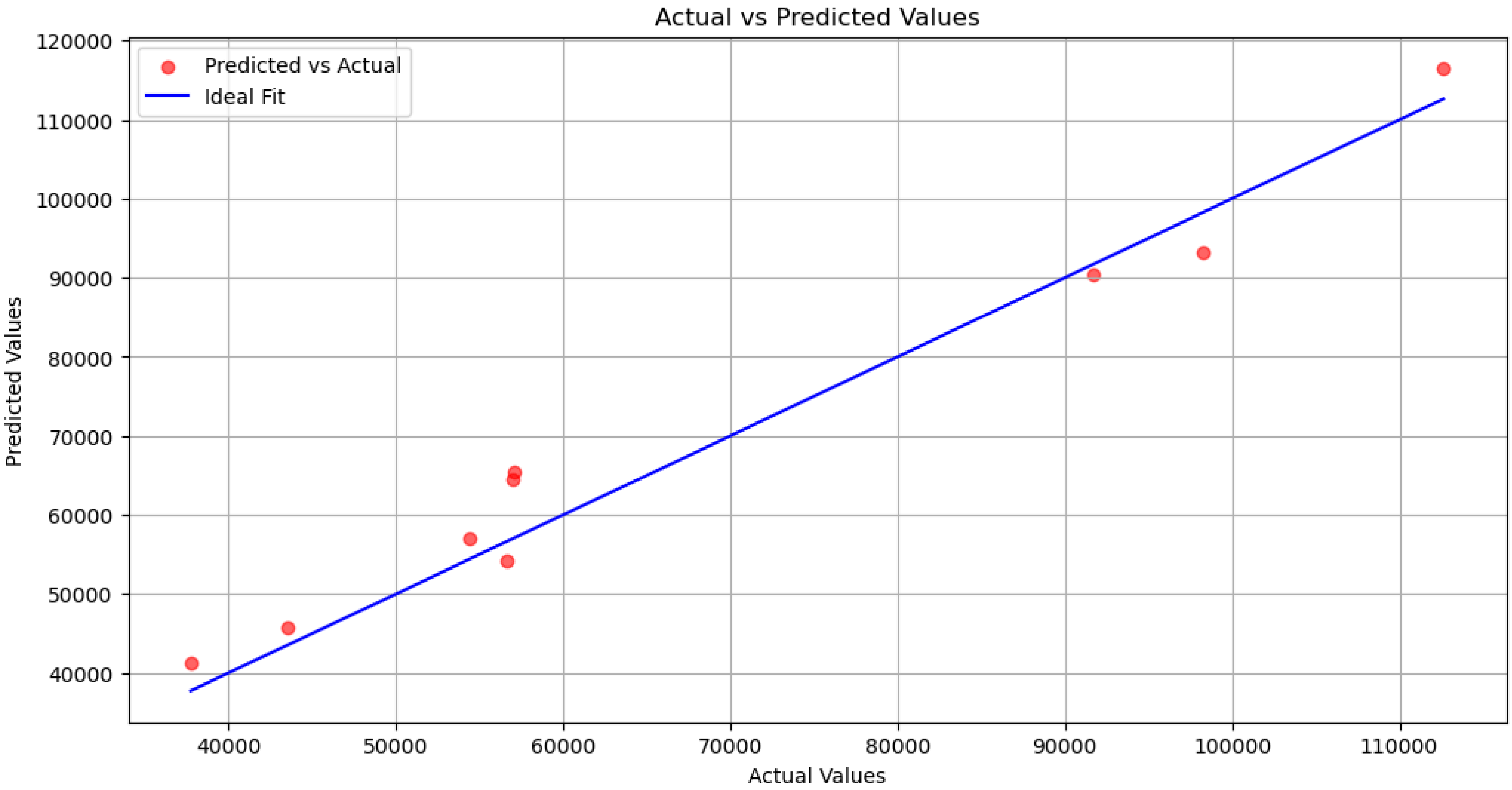
In [39]:
```
# Calculate the mean square error
mse=mean_squared_error(y_test,y_pred)
```

In [40]:
```
# Calculate the R square value
rsq=r2_score(y_test,y_pred)
```

In [41]:
```
print('Mean Squared Error: ',mse)
print('R square: ',rsq)
```

```
Mean Squared Error:  21713548.637118638
R square:  0.9647278344670828
```

In [56]:
```
# Enhanced plot for actual and predicted values
plt.figure(figsize=(12, 6))
plt.scatter(y_test, y_pred, color='red', label='Predicted vs Actual', alpha=0.6)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='blue',label='Ideal Fit')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values' )
plt.title('Actual vs Predicted Values' )
plt.legend()
plt.grid(True)
plt.show()
```



In [46]:

```
# Intecept and coeff of the line
print('Intercept of the model:',lr.intercept_)
print('Coefficient of the line:',lr.coef_)
```

```
Intercept of the model: 27206.42890292858
Coefficient of the line: [9303.95933197]
```

In [17]:

```python
### Logistic Regression
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score

# Load the dataset
dataset = pd.read_csv('data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

# Display the first 10 rows of the dataset
print(dataset.head(10))

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)

# Feature Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Initialize the Logistic Regression model
classifier = LogisticRegression(random_state=0, max_iter=100)
classifier.fit(X_train, y_train)

# Predict on the test set
y_pred = classifier.predict(X_test)

# Display the results (confusion matrix and accuracy)
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

```
   SNo       X_1       X_2   y
0    0 -0.869144  0.389310  0.0
1    1 -0.993467 -0.610591  0.0
2    2 -0.834064  0.239236  0.0
3    3 -0.136471  0.632003  1.0
4    4  0.403887  0.310784  1.0
5    5 -0.569309 -0.246681  0.0
6    6 -0.109982  0.930917  1.0
7    7  0.288994 -0.532689  1.0
8    8  0.319782  0.664582  1.0
9    9  0.558686 -0.621185  1.0
Confusion Matrix:
[[ 8  1]
 [ 3 18]]
Accuracy: 0.87
```