## Importing Essential Libraries

In [17]:
```
# Importing the necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_curve, auc
```

## Reading Data from CSV

Reading the Dataset into datframe and splitting the features and Output label.

In [40]:
```
#Load and preprocess the dataset
data = pd.read_csv("diabetes.csv")

# Splitting the dataset into features and target variable
X = data.iloc[:, :-1]
y = data.iloc[:, -1]
```

## Exploratory Data Analysis

### Data Description

In [19]:
```
# Display the first few rows of the DataFrame to get an overview of the data
data.head()
```

Out[19]:

| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| **1** | 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| **2** | 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |
| **3** | 1 | 89 | 66 | 23 | 94 | 28.1 | 0.167 | 21 | 0 |
| **4** | 0 | 137 | 40 | 35 | 168 | 43.1 | 2.288 | 33 | 1 |

In [21]:
```
# Summary statistics of the dataset
data.describe()
```

Out[21]:

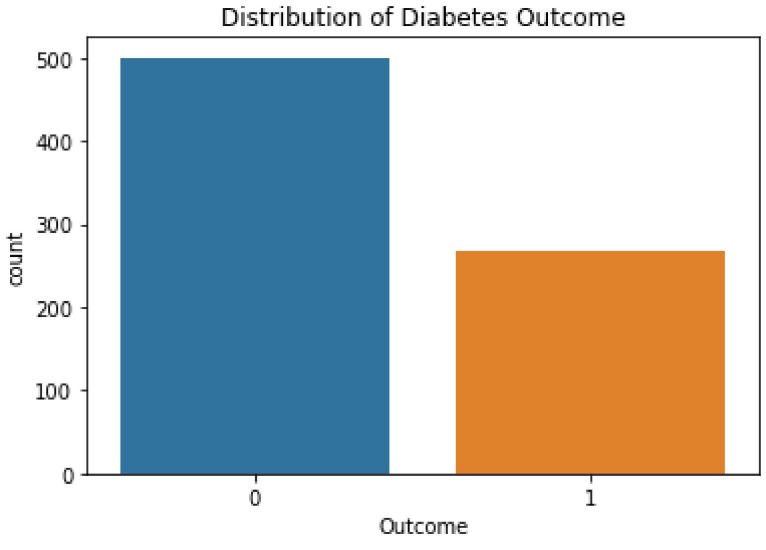
| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| **count** | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 |
| **mean** | 3.845052 | 120.894531 | 69.105469 | 20.536458 | 79.799479 | 31.992578 | 0.471876 | 33.240885 | 0.348958 |
| **std** | 3.369578 | 31.972618 | 19.355807 | 15.952218 | 115.244002 | 7.884160 | 0.331329 | 11.760232 | 0.476951 |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.078000 | 21.000000 | 0.000000 |
| **25%** | 1.000000 | 99.000000 | 62.000000 | 0.000000 | 0.000000 | 27.300000 | 0.243750 | 24.000000 | 0.000000 |
| **50%** | 3.000000 | 117.000000 | 72.000000 | 23.000000 | 30.500000 | 32.000000 | 0.372500 | 29.000000 | 0.000000 |
| **75%** | 6.000000 | 140.250000 | 80.000000 | 32.000000 | 127.250000 | 36.600000 | 0.626250 | 41.000000 | 1.000000 |
| **max** | 17.000000 | 199.000000 | 122.000000 | 99.000000 | 846.000000 | 67.100000 | 2.420000 | 81.000000 | 1.000000 |

### Data Visualization

#### Distribution of the outcome variable

Using a count plot (bar plot) with the sns.countplot() function, we visualize the distribution of the outcome variable. This allows us to understand the balance between positive and negative diabetes cases in the dataset.

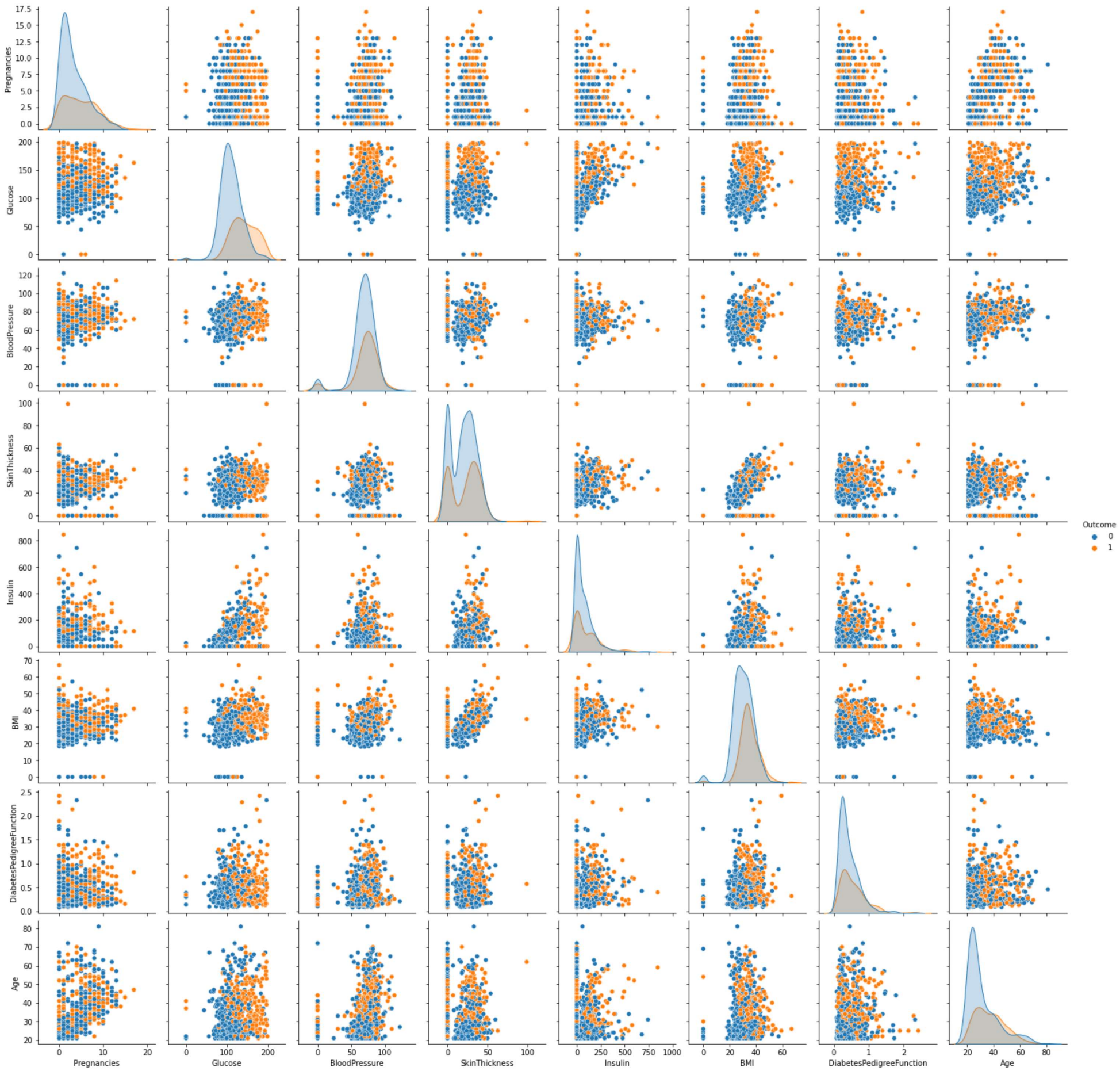
In [22]:
```
# Distribution of the Outcome variable
sns.countplot(x='Outcome', data=data)
plt.title('Distribution of Diabetes Outcome')
plt.show()
```




**Pairwise scatter plot**

With a scatter plot matrix created by sns.pairplot(), we can examine the relationships between pairs of numeric attributes. Each scatter plot shows the relationship between two variables, and the plots are colored by the outcome variable. This visualization can reveal any noticeable patterns or correlations between attributes.

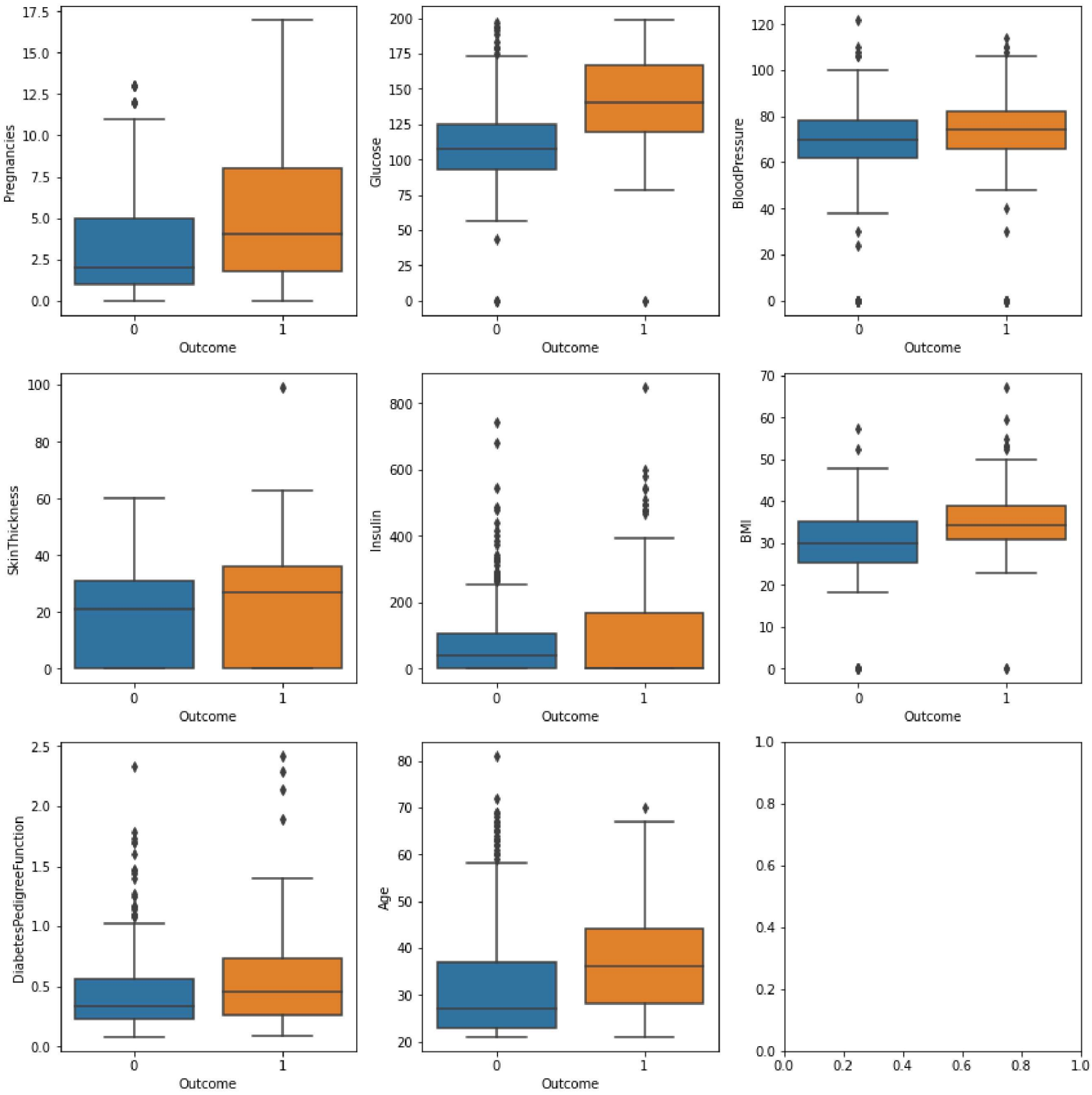
In [23]:
```
# Pairwise scatter plot of all numeric attributes colored by Outcome
sns.pairplot(data=data, hue='Outcome')
plt.show()
```



### Box plots

A box plot (sns.boxplot()) for each attribute, grouped by the outcome variable, helps us understand the distribution of attribute values within each outcome category. It provides information about the median, quartiles, and potential outliers or extreme values.

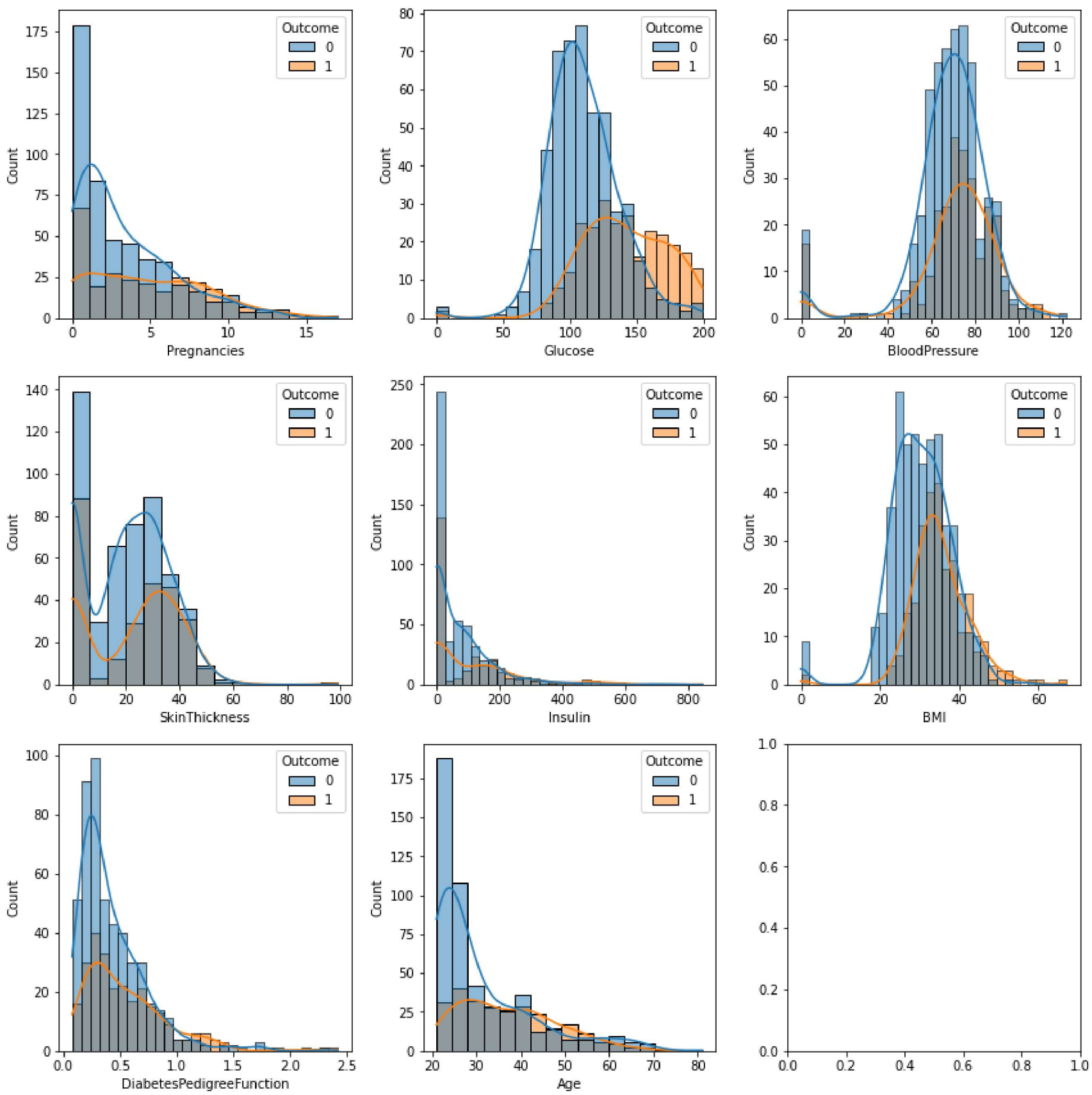
In [24]:
```
# Box plot for each attribute colored by Outcome
fig, axes = plt.subplots(3, 3, figsize=(12, 12))
for i, ax in enumerate(axes.flatten()):
    if i < len(data.columns) - 1:
        sns.boxplot(x='Outcome', y=data.columns[i], data=data, ax=ax)
plt.tight_layout()
plt.show()
```




### Histograms

Using sns.histplot(), we create histograms for each attribute, colored by the outcome variable. Histograms show the distribution of values for a single attribute, allowing us to observe any differences in distribution between positive and negative diabetes cases.

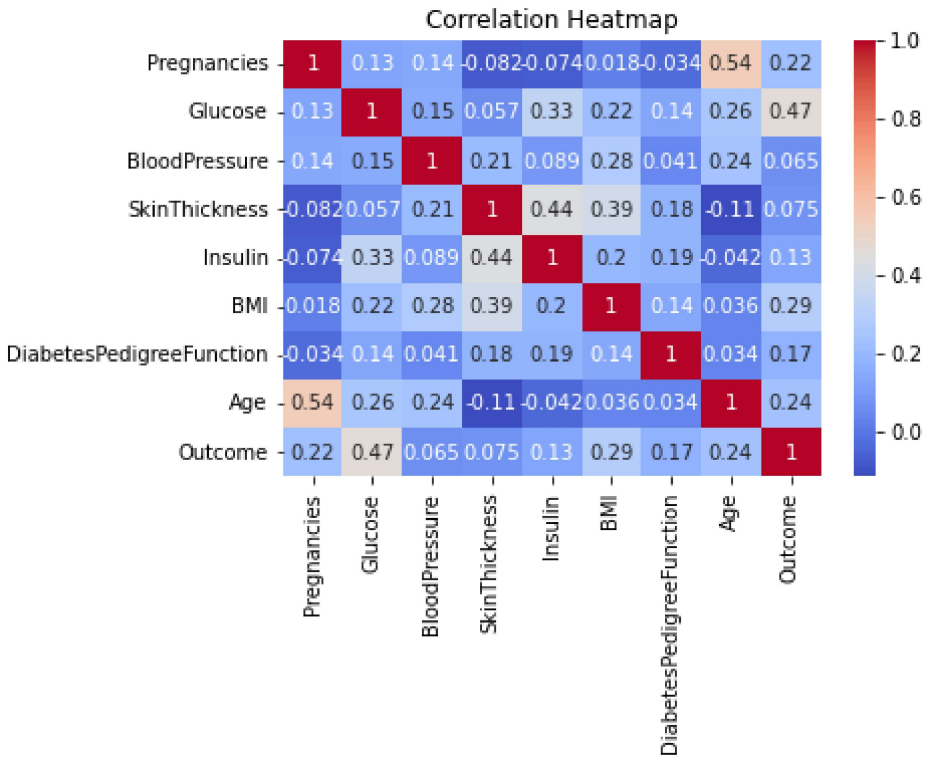
In [25]:
```
# Histograms of each attribute colored by Outcome
fig, axes = plt.subplots(3, 3, figsize=(12, 12))
for i, ax in enumerate(axes.flatten()):
    if i < len(data.columns) - 1:
        sns.histplot(data=data, x=data.columns[i], hue='Outcome', ax=ax, kde=True)
plt.tight_layout()
plt.show()
```




**Correlation heatmap**

A correlation heatmap (sns.heatmap()) displays the correlation coefficients between all pairs of numeric attributes in the dataset. This helps identify any strong positive or negative correlations between attributes. The heatmap is annotated with the correlation values for easy interpretation.

In [26]:
```
# Correlation heatmap
correlation = data.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```




## Train Test Splitting

In [41]:
```
# Splitting the 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)
```

## Features Scaling (Dataset Treatment and Normalization)

Feature scaling is used to bring all the features of a dataset onto a similar scale or range. It is a common preprocessing step in data analysis and machine learning tasks.

In [28]:
```
# Scaling the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

### Models Configuration and Hyperparamteres Selection

### Support Vectors Machine

**Why SVM** SVM model was chosen based on its ability to handle non-linearly separable data, its versatility with different kernel functions, and its effectiveness in high-dimensional spaces.

### Selected Hyperparameters Justification

**Kernel - RBF** The RBF (Radial Basis Function) kernel is a popular choice for SVM models, especially when dealing with non-linearly separable data. It allows the model to capture complex relationships between features by mapping them into a higher-dimensional space. The RBF kernel is widely used due to its ability to handle a variety of data distributions and achieve good generalization performance.

**C - 1.0** The C hyperparameter controls the regularization strength in SVM models. A smaller C value allows for a wider margin and can improve generalization by avoiding overfitting. The default value of C is 1.0, which is often considered a reasonable starting point. It balances the trade-off between fitting the training data and maintaining a wider margin.

**Gamma - 'scale'** The gamma hyperparameter controls the influence of a single training example. A lower gamma value results in a smoother decision boundary with a broader influence of training examples, which can help prevent overfitting and improve generalization. Using 'scale' as the gamma value scales the value based on the inverse of the number of features and the variance of the data. It allows the algorithm to adapt to the dataset's characteristics automatically.

**Degree - 3** The degree hyperparameter is only relevant when using a polynomial kernel. It determines the complexity of the decision boundary by specifying the degree of the polynomial used in the kernel function. The default value of degree is 3, which is often a good starting point for capturing moderately complex relationships between features without excessively increasing model complexity.

**Random State - 42** The random_state hyperparameter sets the random seed used by the SVM model. It ensures reproducibility of the results. Choosing a specific random_state value (such as 42) allows for consistent results each time the model is run, which is useful for reproducibility and result comparison purposes.

In [29]:
```
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale', degree=3, random_state=42)
```

### Ensembles Models

**Why Random Forest** Random Forest is a popular choice for the diabetes dataset due to its ability to handle non-linear relationships and its robustness to outliers and noisy data. By aggregating predictions from multiple decision trees, Random Forest can capture complex interactions and patterns in the data. It also provides insights into feature importance, aiding in feature selection and understanding the data. Moreover, Random Forest performs well in high-dimensional spaces by randomly selecting features, reducing the risk of overfitting. Additionally, it can handle imbalanced classes and adjust class weights for better predictions. However, it's important to note that the choice of algorithm depends on various factors, and experimentation is necessary to determine the most suitable approach for a given problem

### Selected Hyperparameter Justification

**n_estimators - 100** The n_estimators hyperparameter represents the number of decision trees in the random forest. Increasing the number of estimators can improve the model's performance by reducing overfitting and increasing the robustness of predictions. However, there is a trade-off between model performance and computational complexity. Too many estimators can increase training time and memory requirements. In this case, we chose 100 as a reasonably large number of estimators to provide a balance between model performance and computational efficiency. The exact value can be adjusted based on the size of the dataset and available computational resources.

In [30]:
```
ensemble_model = RandomForestClassifier(n_estimators=100, random_state=42)
```

## Models Training

In [31]:
```
# Fitting the SVM model to the training data
svm_model.fit(X_train_scaled, y_train)
```

Out[31]:
```
SVC(random_state=42)
```

In [32]:
```
# Fitting the ensemble model to the training data
ensemble_model.fit(X_train, y_train)
```

Out[32]:
```
RandomForestClassifier(random_state=42)
```

## Support Vector Machine Evaluation

In [33]:
```
# Applying the SVM model to the testing data
y_pred_svm = svm_model.predict(X_test_scaled)

# Evaluating the SVM model
accuracy_svm = accuracy_score(y_test, y_pred_svm)
report_svm = classification_report(y_test, y_pred_svm)
cm_svm = confusion_matrix(y_test, y_pred_svm)
```

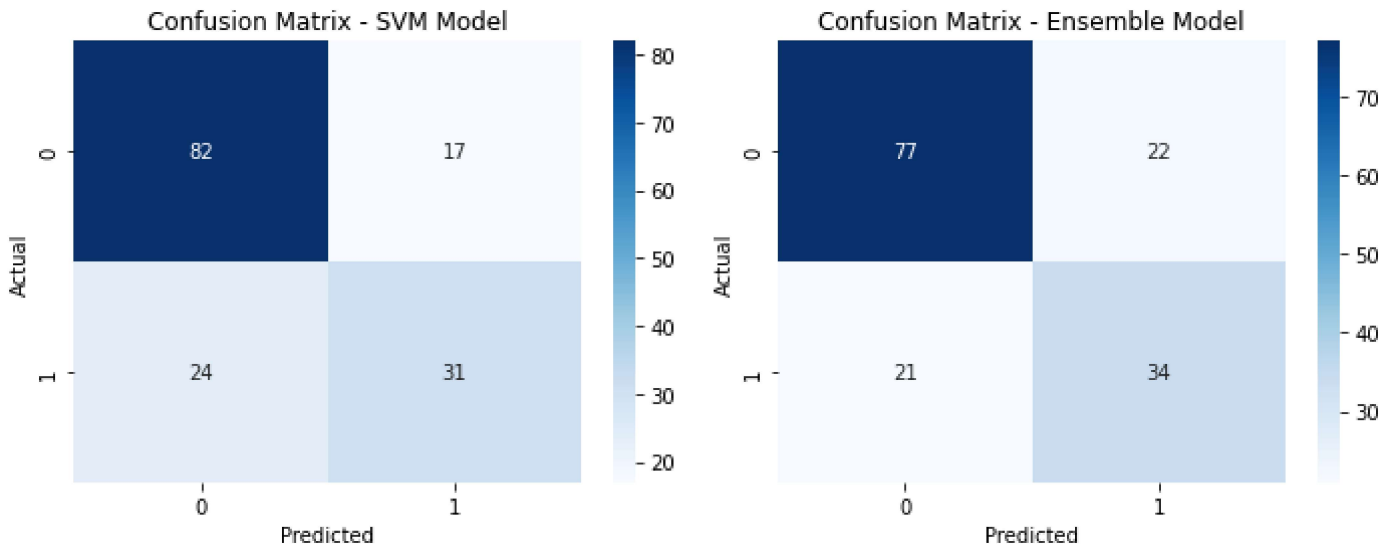
## Ensembles Model Evaluation

In [34]:
```
# Applying the ensemble model to the testing data
y_pred_ensemble = ensemble_model.predict(X_test)

# Evaluating the ensemble model
accuracy_ensemble = accuracy_score(y_test, y_pred_ensemble)
report_ensemble = classification_report(y_test, y_pred_ensemble)
cm_ensemble = confusion_matrix(y_test, y_pred_ensemble)
```

## Results Comparion and Visualization

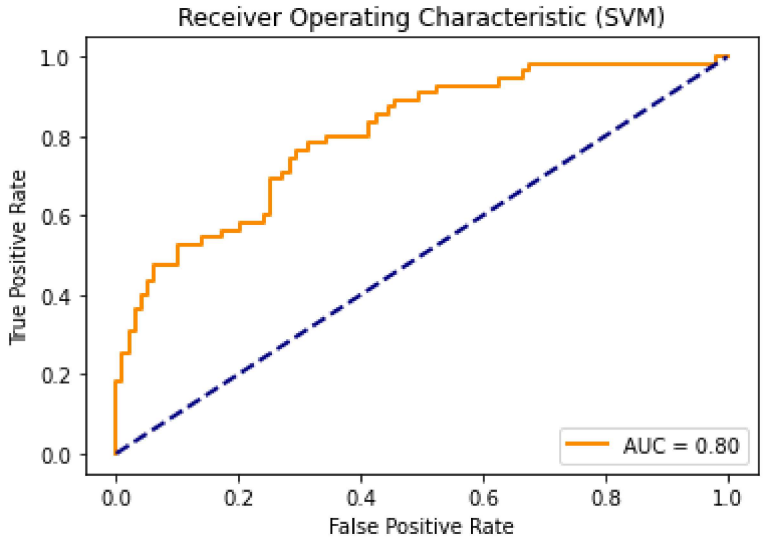
#### Confusion Matrix

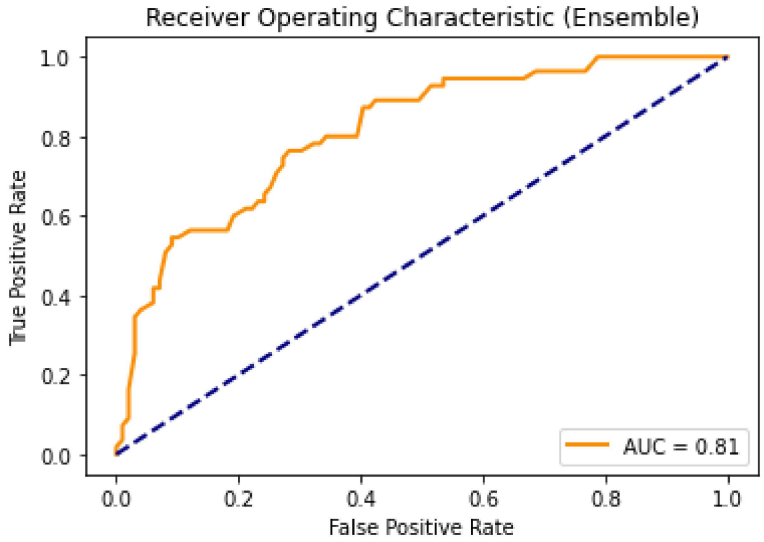
In [35]:
```
# Plotting the confusion matrices
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
sns.heatmap(cm_svm, annot=True, cmap='Blues', fmt='g')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - SVM Model")

plt.subplot(1, 2, 2)
sns.heatmap(cm_ensemble, annot=True, cmap='Blues', fmt='g')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Ensemble Model")
plt.tight_layout()
plt.show()
```




**ROC curve**

In [36]:
```
# Plotting the ROC curve and calculating the AUC for SVM model
svm_prob = svm_model.decision_function(X_test_scaled)
fpr_svm, tpr_svm, _ = roc_curve(y_test, svm_prob)
roc_auc_svm = auc(fpr_svm, tpr_svm)

plt.figure()
plt.plot(fpr_svm, tpr_svm, color='darkorange', lw=2, label=f'AUC = {roc_auc_svm:.2f}')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (SVM)')
plt.legend(loc="lower right")
plt.show()

# Plotting the ROC curve and calculating the AUC for Ensemble model
ensemble_prob = ensemble_model.predict_proba(X_test)[:, 1]
fpr_ensemble, tpr_ensemble, _ = roc_curve(y_test, ensemble_prob)
roc_auc_ensemble = auc(fpr_ensemble, tpr_ensemble)

plt.figure()
plt.plot(fpr_ensemble, tpr_ensemble, color='darkorange', lw=2, label=f'AUC = {roc_auc_ensemble:.2f}')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (Ensemble)')
plt.legend(loc="lower right")
plt.show()
```



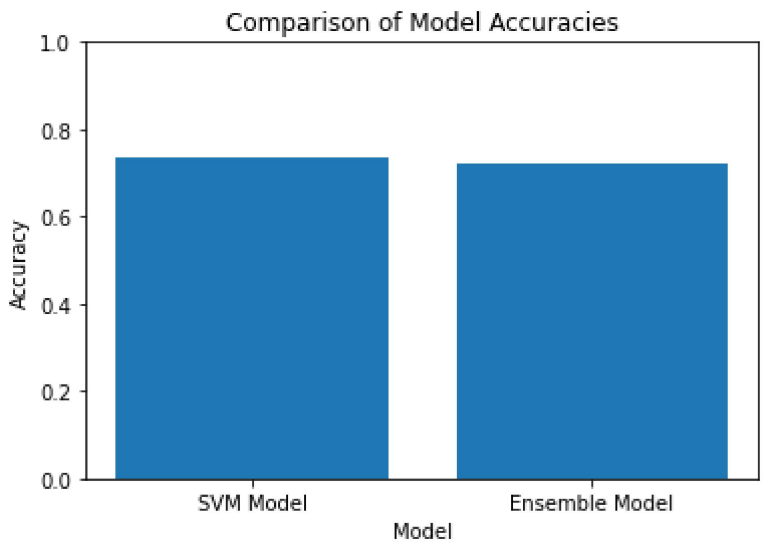




**Classification Report and Accuracy Comparison**

In [43]:
```
print("SVM Model Performance:")
print(f"Accuracy: {accuracy_svm:.4f}")
print("Classification Report:")
print(report_svm)
print("_____\n\n")
print("Ensemble Model Performance:")
print(f"Accuracy: {accuracy_ensemble:.4f}")
print("Classification Report:")
print(report_ensemble)
```

```
SVM Model Performance:
Accuracy: 0.7338
Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.83      0.80        99
           1       0.65      0.56      0.60        55

    accuracy                           0.73       154
   macro avg       0.71      0.70      0.70       154
weighted avg       0.73      0.73      0.73       154


_____


Ensemble Model Performance:
Accuracy: 0.7208
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.78      0.78        99
           1       0.61      0.62      0.61        55

    accuracy                           0.72       154
   macro avg       0.70      0.70      0.70       154
weighted avg       0.72      0.72      0.72       154
```

In [39]:
```
# Create a bar chart
models = ['SVM Model', 'Ensemble Model']
accuracies = [accuracy_svm, accuracy_ensemble]

plt.bar(models, accuracies)
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Comparison of Model Accuracies')
plt.ylim(0, 1)  # Set the y-axis limit

# Display the chart
plt.show()
```




# ADVANTAGES AND DISADVANTAGES

**Advantages of SVM**

- Effective in high-dimensional spaces
- Versatility with kernels
- Control over regularization
- Robust to outliers

**Disadvantages of SVM**

- Sensitivity to parameter tuning
- Computationally expensive
- Difficulty handling noisy datasets

**Advantages of ensemble methods (Random Forest)**

```
Improved accuracy and robustness
Handles high-dimensional data
Provides feature importance
Handles missing values and outliers
```

**Disadvantages of ensemble methods (Random Forest)**

```
Lack of interpretability
Computationally expensive
Biased towards features with more categories
```

# CONCLUSION

In conclusion, both the SVM model and the ensemble model demonstrate similar performance on the given dataset. The SVM model achieves a slightly higher accuracy, but the ensemble model offers a comparable performance in terms of precision, recall, and F1-score. Further analysis, such as examining additional evaluation metrics, comparing feature importances, and considering the specific requirements of the problem, can provide deeper insights into the models' advantages and disadvantages.