Classification and prediction of diabetes disease

Support machine vector

Name : Sai Krishna Vavilli

Student id : 23022047

Github link : https://github.com/SaiKrishna200120/Applied-Data-science-2.git

Dataset link: https://www.kaggle.com/datasets/mathchi/diabetes-data-set

Table of Contents :

Abstract

Support Vector Machines (SVM) are a powerful class of supervised learning algorithms widely used for classification tasks, particularly when the data is high-dimensional or non-linearly separable. This tutorial focuses on the application of SVM using the Diabetes dataset, a well-known dataset used to predict the presence of diabetes based on various clinical measurements. The tutorial covers the fundamentals of SVM, including the concept of hyperplanes, margins, and support vectors, and demonstrates how to apply SVM using both linear and non-linear kernels, such as the Radial Basis Function (RBF) and Sigmoid kernels. By pre-processing the dataset, splitting it into training and test sets, and implementing different SVM kernels, we explore how each kernel performs in terms of classification accuracy and model evaluation. The tutorial provides a step-by-step guide for implementing SVM, interpreting results, and comparing the effectiveness of different kernels in solving real-world classification problems. The goal is to equip readers with the knowledge necessary to apply SVM to similar tasks and understand the impact of kernel choice on model performance.

Introduction to SVM

Support Vector Machines (SVM) are a class of supervised learning algorithms widely used for classification tasks. They are particularly powerful in high-dimensional spaces and are well-suited for binary classification problems, like the task of classifying diabetes diagnoses.

Main concepts in SVM

Hyperplane: A hyperplane is the decision boundary that separates different classes in the feature space. For example, in a 2D space, the hyperplane is a line, whereas in higher dimensions, it is a plane or hyperplane.

Margin: The margin is the distance between the hyperplane and the closest data points, known as support vectors. SVM works to maximize this margin to improve the separation between the classes.

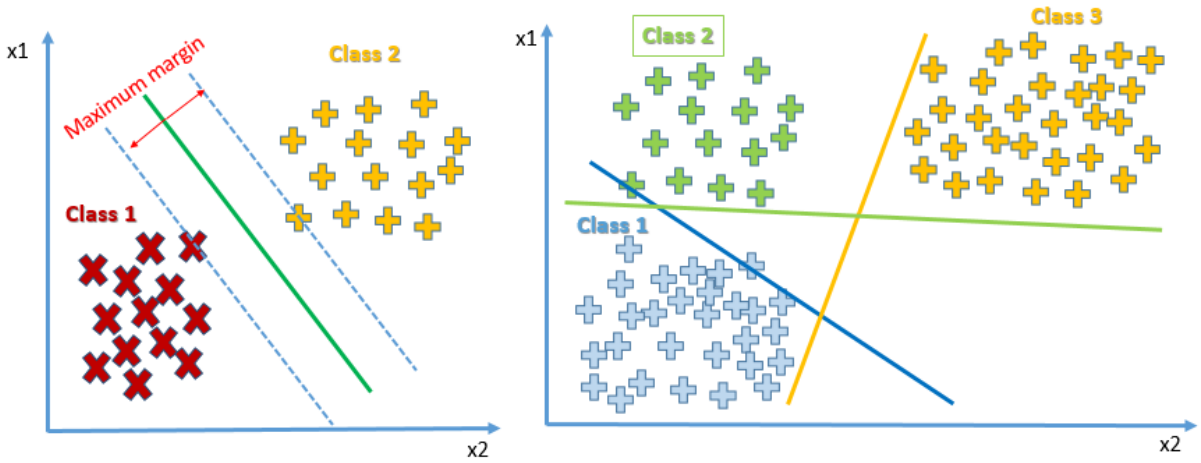
Support Vectors: Support vectors are the closest data points to the hyperplane. They are pivotal in defining the hyperplane and are used to determine the optimal decision boundary.

Linear vs Non-Linear SVM:

- **Linear SVM** is used when the data is linearly separable (i.e., it can be separated by a straight line or hyperplane).
- **Non-Linear SVM**: When data is not linearly separable, SVM uses the kernel trick to map the data into a higher-dimensional space, where it becomes linearly separable.

Kernel Trick: The kernel trick allows SVM to efficiently handle non-linear decision boundaries. Common kernels include:

- **Linear Kernel**: For linearly separable data.
- **Polynomial Kernel**: For data separable by a polynomial decision boundary.
- **Radial Basis Function (RBF) Kernel**: For non-linearly separable data in the original space.



SVM classifies its hyperspace into zones(category) using a hyperplane to separate the data. when a future data point lies in a specific

hyperspace zone, it is automatically classified into the corresponding category.

Objective:

The objective is to demonstrate the application of Support Vector Machines (SVM) on the Diabetes dataset for binary classification, predicting whether a given individual has diabetes or not based on clinical data.

Why SVM?

Support Vector Machines (SVM) are well-suited for the Diabetes dataset due to their ability to handle high-dimensional and non-linearly separable data. SVM maximizes the margin between classes, reducing overfitting and ensuring robust generalization. With its flexibility to apply various kernels (like RBF or Sigmoid), SVM can effectively classify binary outcomes in medical datasets like diabetes prediction.

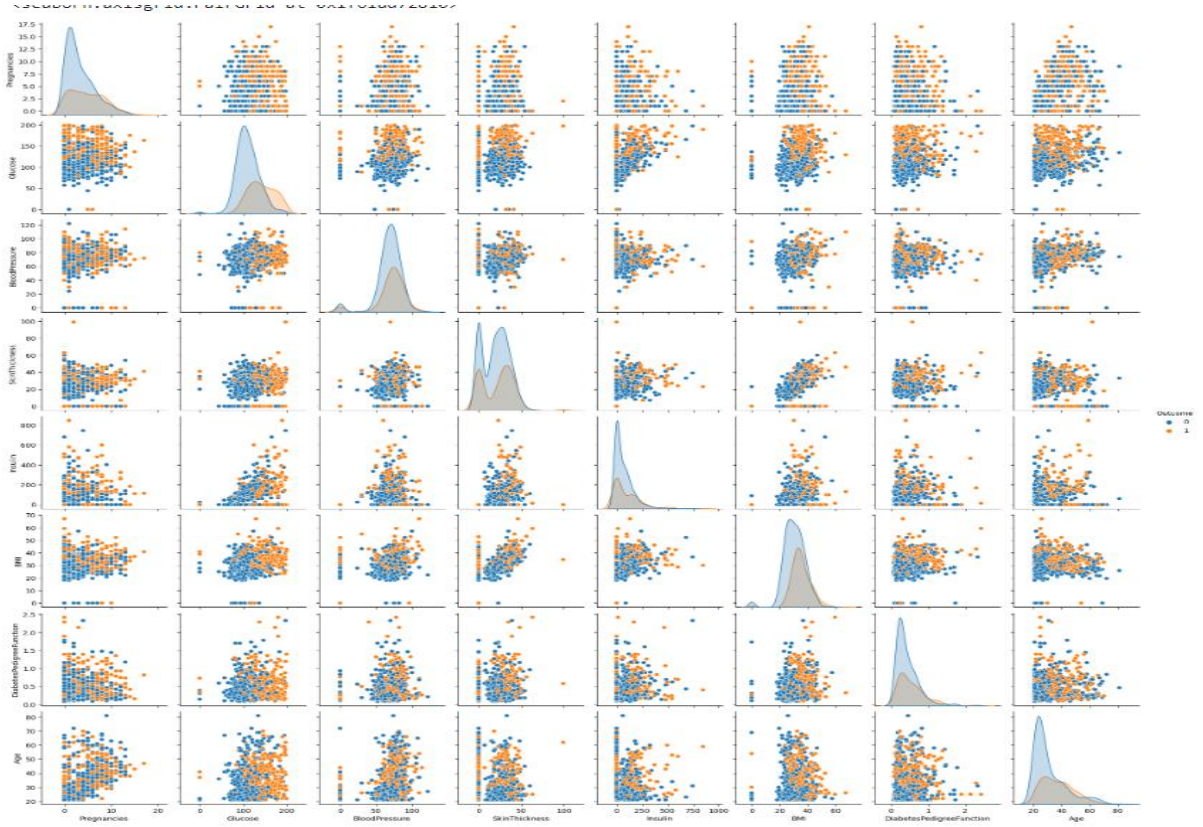
Diabetes Dataset Overview:

The Diabetes dataset is often used to demonstrate classification techniques. It contains diagnostic data for predicting whether a patient has diabetes. The dataset includes the following features:

- **Pregnancies**: Number of times the patient has been pregnant.
- **Glucose**: Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
- **BloodPressure**: Diastolic blood pressure (mm Hg).
- **SkinThickness**: Triceps skinfold thickness (mm).
- **Insulin**: 2-Hour serum insulin (mu U/ml).
- **BMI**: Body mass index (weight in kg / height in m^2).
- **DiabetesPedigreeFunction**: Diabetes pedigree function.
- **Age**: Age of the patient.

- **Outcome**: Target variable indicating whether the patient has diabetes (1 = yes, 0 = no)(Boolean)(classifier feature)

Explorative data analysis



Exploratory Data Analysis (EDA) reveals patterns, correlations, and data distribution.

Data visualization using PCA

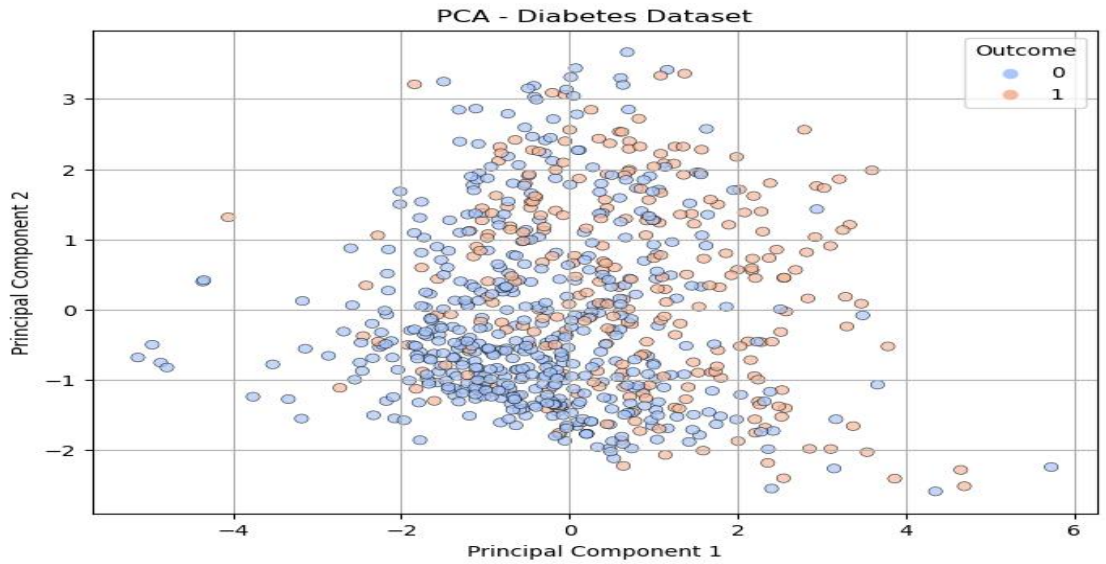
What is PCA?

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms high-dimensional data into lower dimensions while retaining the most important features, improving visualization and model performance.

```python
# Step 1: Standardize the data
scaler_pca = StandardScaler()
scaled_data = scaler_pca.fit_transform(X)

# Step 2: Perform PCA
pca = PCA(n_components=2)
pca_result = pca.fit_transform(scaled_data)

# Step 3: Plot the first two principal components with different colors for each outcome class
plt.figure(figsize=(8,6))
sns.scatterplot(x=pca_result[:, 0], y=pca_result[:, 1], hue=Y, palette='coolwarm', edgecolor='k', alpha=0.7)
plt.title('PCA - Diabetes Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True)
plt.show()
```



This is a 2D PCA of the Diabetes dataset, where a line divides the outcomes, which we call a hyperplane in higher dimensions. The hyperplane separates the data into distinct zones, so if a new point is plotted within a specific zone, it is automatically classified based on the training data. However, in this example, a line alone is not sufficient to classify the data neatly, so higher dimensions are needed for better separation and classification.

In this section, provide an overview of Support Vector Machines (SVM) and explain why they are well-suited for classification tasks, especially in high-dimensional spaces.

Preparing the Data

Loading the Data

Loading the data refers to the process of importing the dataset into your programming environment, making it accessible for analysis and model training

```
# loading the diabetes dataset to a pandas DataFrame
diabetes_dataset = pd.read_csv('diabetes.csv')
diabetes_dataset.head()
```

	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

Data cleaning

Data cleaning is an essential step to ensure that the dataset is accurate and free of errors. This step involves handling missing or null values, removing duplicates, and fixing inconsistent data entries. Cleaning the data is crucial because any errors or inconsistencies in the dataset can lead to poor model performance.

```
diabetes_dataset.isnull().sum()
```

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Data Preprocessing

Data pre-processing is essential to ensure that the data is clean, consistent, and ready for machine learning models. In this case, StandardScaler is used to standardize the features of the dataset. Standardization is particularly important for models like Support Vector Machines (SVM), which are sensitive to the scale of the input features. Without scaling, features with larger numerical ranges could disproportionately influence the model's decision boundary, leading to suboptimal performance.

The StandardScaler normalizes the features by transforming them to have a mean of 0 and a standard deviation of 1. This ensures that each feature contributes equally to the model, improving its efficiency and accuracy.

Furthermore, splitting the dataset into training and testing sets allows for proper model evaluation. The training set is used to train the model, while the test set is used to assess its ability to generalize to unseen data, helping to avoid overfitting.

```
scaler = StandardScaler()
scaler.fit(X)
```

```
StandardScaler()
```

```
standardized_data = scaler.transform(X)
```

```
X = standardized_data
Y = diabetes_dataset['Outcome']
```

```
0]: X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.2, stratify=Y, random_state=2)
```

Implementing SVM with Linear Kernel

```
classifier_linear = svm.SVC(kernel='linear')
# training the support vector Machine Classifier
classifier_linear.fit(X_train, Y_train)
# accuracy score on the training data
X_train_prediction_linear = classifier_linear.predict(X_train)
training_data_accuracy_linear = accuracy_score(X_train_prediction_linear, Y_train)
print('Accuracy score of the training data using linear kernel : ', training_data_accuracy_linear*100,'%')
```

```
Accuracy score of the training data using linear kernel :  78.66449511400651 %
```

This code trains a Support Vector Machine (SVM) classifier with a linear kernel using the SVC class from the sklearn.svm module. The classifier is first trained on the training dataset (X_train, Y_train) using the fit() method. Then, predictions are made on the training data (X_train) using the predict() method, and the accuracy score is calculated by comparing predicted labels (X_train_prediction_linear) with the true labels (Y_train). The result is printed as the training accuracy. The same process is repeated for the test dataset (X_test, Y_test) to evaluate the classifier's performance on unseen data, and the test accuracy is calculated and stored in test_data_accuracy_linear.

Implementing SVM with Non-Linear Kernels (poly)

```
classifier_poly = svm.SVC(kernel='poly')
#training the support vector Machine Classifier
classifier_poly.fit(X_train, Y_train)
# accuracy score on the training data
X_train_prediction_poly = classifier_poly.predict(X_train)
training_data_accuracy_poly = accuracy_score(X_train_prediction_poly, Y_train)
print('Accuracy score of the training data using poly kernel : ', training_data_accuracy_poly*100,'%')
# accuracy score on the test data
X_test_prediction_poly = classifier_poly.predict(X_test)
test_data_accuracy_poly = accuracy_score(X_test_prediction_poly, Y_test)
print('Accuracy score of the test data using poly kernel : ', test_data_accuracy_poly*100,'%')
```

```
Accuracy score of the training data using poly kernel :  80.29315960912052 %
Accuracy score of the test data using poly kernel :  70.77922077922078 %
```

This code trains a Support Vector Machine (SVM) classifier using a polynomial kernel (kernel='poly'). First, the classifier is trained on the training dataset (X_train, Y_train) using the fit() method. The model then makes predictions on the training data (X_train) using predict(), and the accuracy score is calculated by comparing the predicted values (X_train_prediction_poly) with the true labels (Y_train). The training accuracy is printed. Next, the classifier predicts the outcomes for the test dataset (X_test), and the test accuracy is calculated by comparing predictions (X_test_prediction_poly) with the true test labels (Y_test), printing the result.

Implementing SVM with Non-Linear Kernels (RBF)

```
classifier_rbf = svm.SVC(kernel='rbf')
#training the support vector Machine Classifier
classifier_rbf.fit(X_train, Y_train)
# accuracy score on the training data
X_train_prediction_rbf = classifier_rbf.predict(X_train)
training_data_accuracy_rbf = accuracy_score(X_train_prediction_rbf, Y_train)
print('Accuracy score of the training data using rbf kernel : ', training_data_accuracy_rbf*100,'%')
# accuracy score on the test data
X_test_prediction_rbf = classifier_rbf.predict(X_test)
test_data_accuracy_rbf = accuracy_score(X_test_prediction_rbf, Y_test)
print('Accuracy score of the test data using rbf kernel: ', test_data_accuracy_rbf*100,'%')
```

```
Accuracy score of the training data using rbf kernel :  82.89902280130293 %
Accuracy score of the test data using rbf kernel:  72.72727272727273 %
```

This code trains a Support Vector Machine (SVM) classifier using the Radial Basis Function (RBF) kernel (kernel='rbf'). The fit() method is applied to the training data (X_train, Y_train), where the model

learns the decision boundary. After training, the classifier predicts the outcomes for the training data (X_train) using predict(). The accuracy score is then computed by comparing these predictions (X_train_prediction_rbf) with the actual labels (Y_train) and is printed as the training accuracy. The same procedure is repeated for the test data (X_test), where the model's ability to generalize is evaluated by comparing the predicted test labels (X_test_prediction_rbf) to the true labels (Y_test), and the test accuracy is displayed.

Implementing SVM with Non-Linear Kernels (sigmoid)

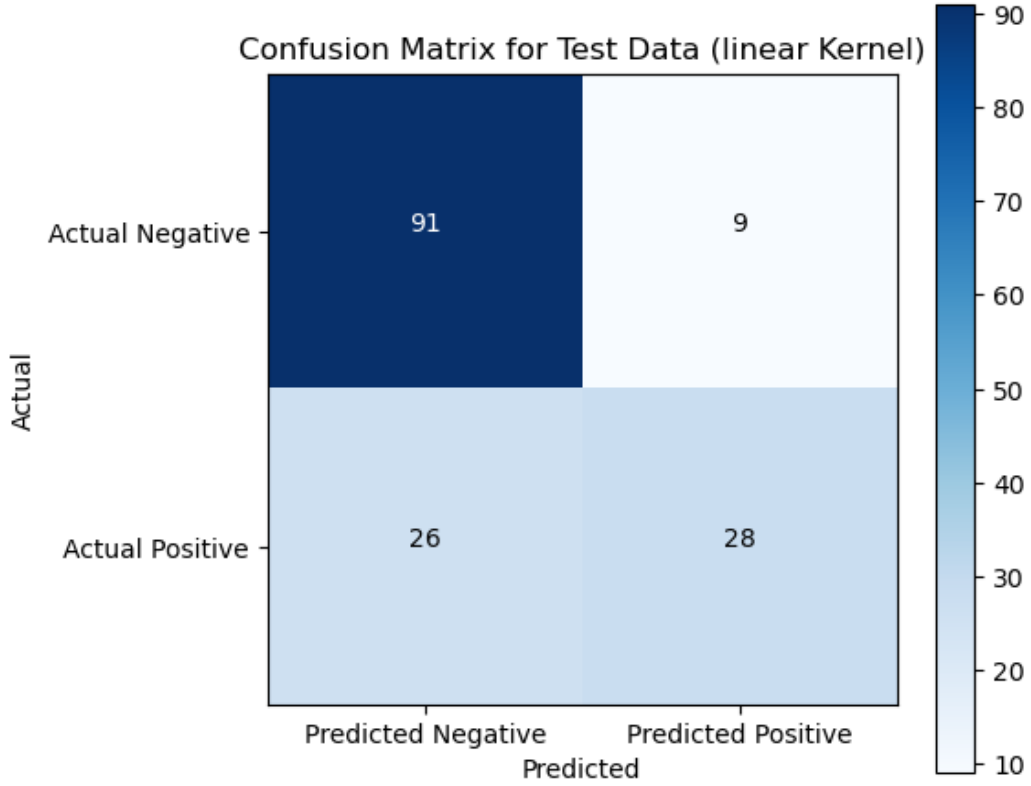
```
classifier_sigmoid = svm.SVC(kernel='sigmoid')
#training the support vector Machine Classifier
classifier_sigmoid.fit(X_train, Y_train)
# accuracy score on the training data
X_train_prediction_sigmoid = classifier_sigmoid.predict(X_train)
training_data_accuracy_sigmoid = accuracy_score(X_train_prediction_sigmoid, Y_train)
print('Accuracy score of the training data using sigmoid kernel : ', training_data_accuracy_sigmoid*100,'%')
# accuracy score on the test data
X_test_prediction_sigmoid = classifier_sigmoid.predict(X_test)
test_data_accuracy_sigmoid = accuracy_score(X_test_prediction_sigmoid, Y_test)
print('Accuracy score of the test data using sigmoid kernel : ', test_data_accuracy_sigmoid*100,'%')
```

```
Accuracy score of the training data using sigmoid kernel :  71.00977198697068 %
Accuracy score of the test data using sigmoid kernel :  75.32467532467533 %
```

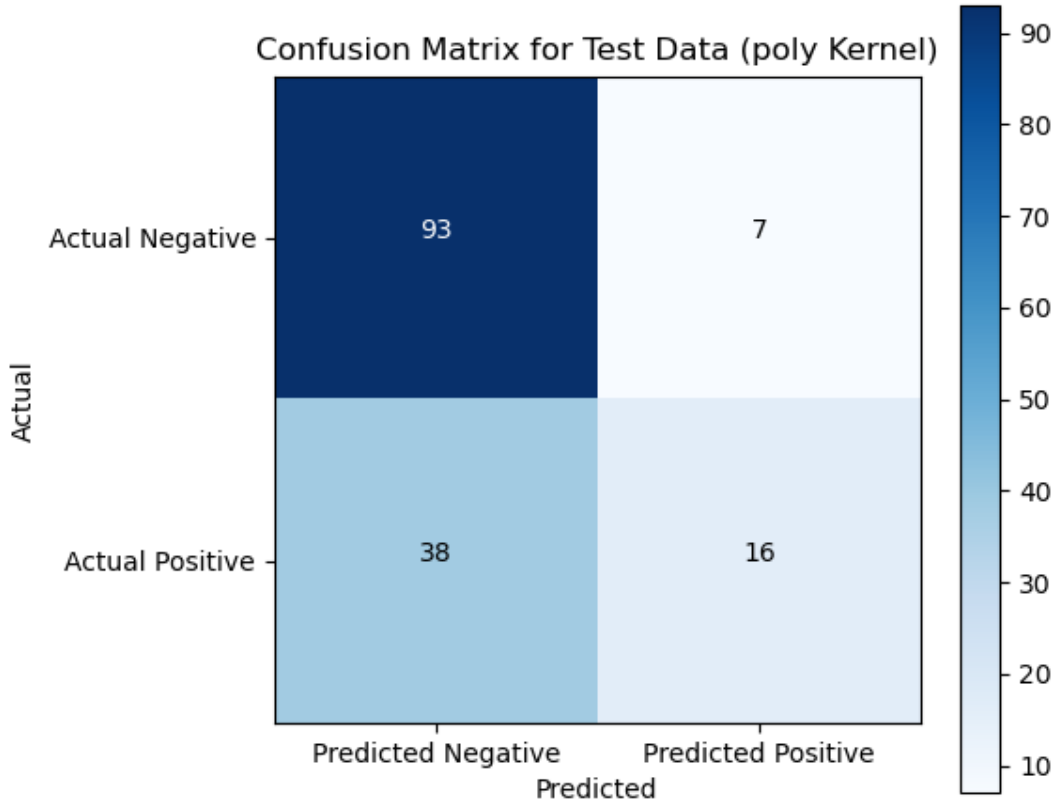
This code trains a Support Vector Machine (SVM) classifier using the sigmoid kernel (kernel='sigmoid'). The classifier is first trained on the training dataset (X_train, Y_train) using the fit() method. After training, predictions are made on the training data (X_train) using predict(), and the training accuracy is calculated by comparing the predicted values (X_train_prediction_sigmoid) to the true labels (Y_train). The result is printed as the training accuracy. The model then predicts outcomes for the test dataset (X_test), and the test accuracy is computed by comparing predicted labels (X_test_prediction_sigmoid) to the true test labels (Y_test), with the result printed as the test accuracy.

Evaluating the Models

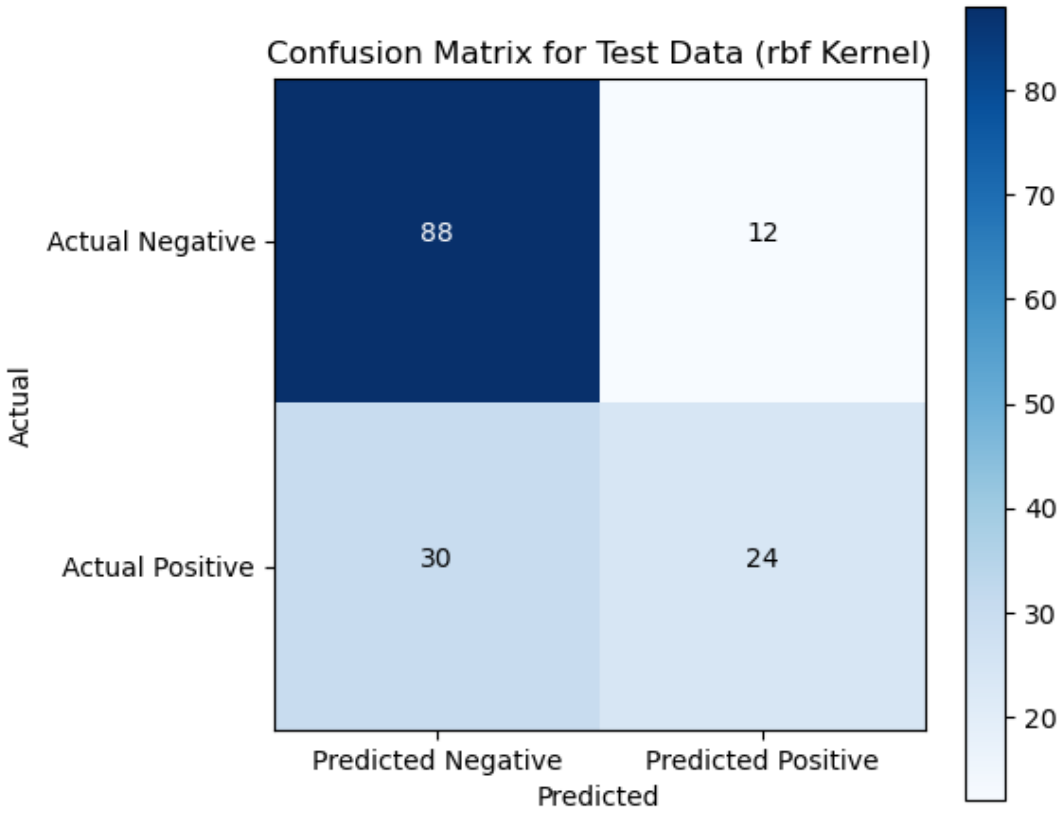
After training the models with different kernels, it's essential to evaluate them using metrics like accuracy**,** f1-score, and the confusion matrix. Comparing the confusion matrices and classification reports will give insights into which kernel performs best for this particular dataset.



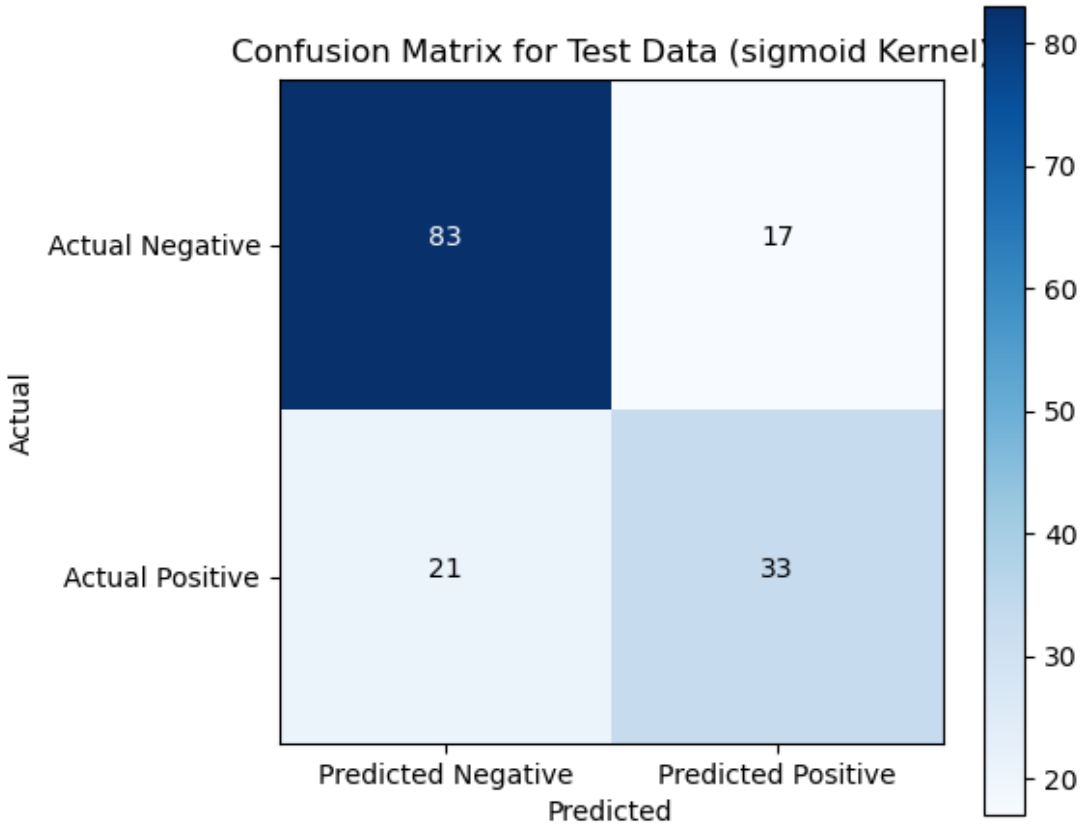

```
F1 Score: 0.62
```




```
F1 Score: 0.42
```




```
F1 Score: 0.53
```




```
F1 Score: 0.63
```

Best model is with respect to f1 score is sigmoid kernel of svm

Conclusion

In this tutorial, we implemented the Support Vector Machine (SVM) algorithm to classify diabetes cases using the Diabetes dataset. We demonstrated how to apply both linear , poly and non-linear kernels (RBF, Sigmoid) to the data and evaluated their performance. The linear SVM works well for linearly separable data, while the non-linear kernels can handle more complex data distributions.

- **Linear kernel**: Works best when the data is linearly separable.
- **Poly kernel**: Works best when the data is non-linearly separable.
- **RBF kernel**: Effective for highly non-linear data.
- **Sigmoid kernel**: Mimics the behavior of neural networks but can be sensitive to hyperparameters.

Through this tutorial, you should now understand how SVM works, how to apply it to real-world datasets, and how to evaluate its performance.

References

- Vapnik, V. (1995). The Nature of Statistical Learning Theory. Springer.
- Pedregosa, F., et al. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825-2830.