PW SKILLS

Lesson Plan

Linear SVM

Topics Covered

- Linear SVM Classification
- Paper Link - for detailed explanation
- Soft Margin Classification
- Practical Implementation - classifier
- Nonlinear SVM Classification
- Polynomial Kernel
- Gaussian (RBF) Kernel
- Data Leakage
- Practical Implementation - non-linear classifier
- SVM Regression
- Practical Implementation - regression

Linear SVM Classification

- Support Vector Machines (SVMs) are supervised learning models used for classification and regression tasks.
- They are particularly effective in high-dimensional spaces and when the number of features exceeds the number of samples.
- SVMs aim to find the optimal hyperplane that best separates different classes in the feature space.
- In SVM classification, a linear decision boundary is a hyperplane that separates data points of different classes.
- For a binary classification problem, the decision boundary divides the feature space into two regions, each corresponding to a different class.
- Mathematically, the decision boundary is represented as a linear equation: $\mathbf{w}^T\mathbf{x} + b = 0,$
- The Maximum Margin Hyperplane is the hyperplane that maximizes the margin, i.e., the distance between the hyperplane and the nearest data points (support vectors) of each class.
- SVM aims to find the MMH because it generalizes better to unseen data and is less sensitive to noise.
- The MMH is determined by the support vectors, which are the data points closest to the decision boundary.
- Margin is the distance between the decision boundary and the nearest data points (support vectors).
- Larger margins indicate better generalization ability of the classifier.
- Support vectors are the data points that lie closest to the decision boundary and have a non-zero contribution to defining the margin.
- These points are crucial in determining the MMH and are used to define the decision boundary.

Paper to Demonstrate Working of SVM

SVM Paper- **Link**

Soft Margin Classification

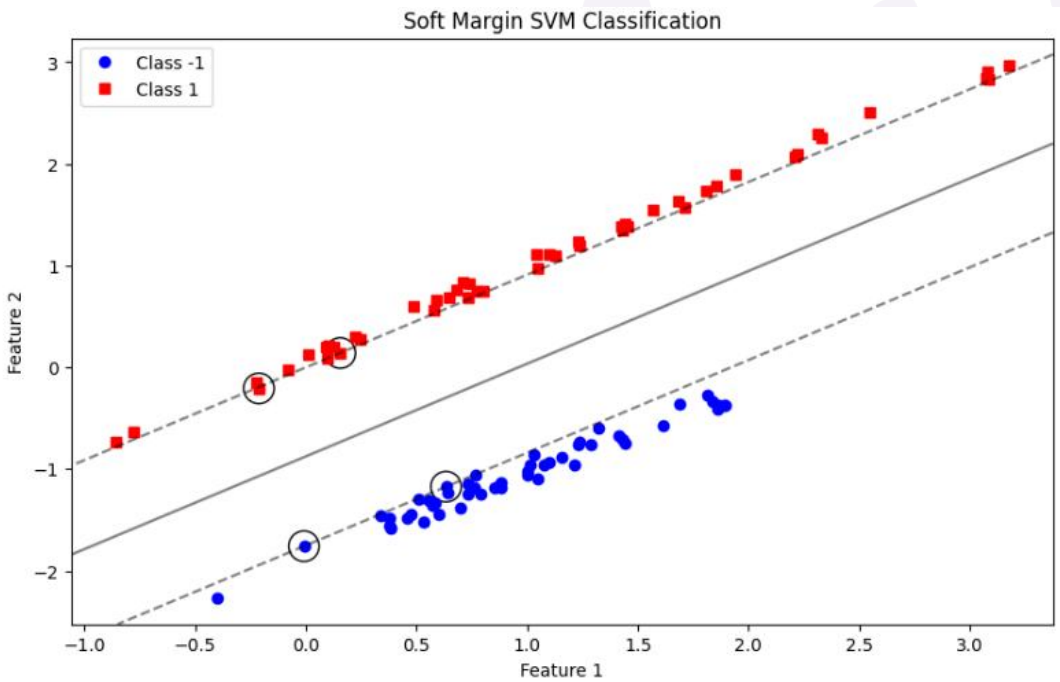
- In real-world scenarios, data is often not perfectly separable by a hyperplane.
- Soft Margin Classification relaxes the strict requirement of having a clear margin between classes.
- It allows for some misclassification to accommodate noisy or overlapping data points.
- Soft Margins introduce the concept of a "margin of tolerance" around the decision boundary.
- This margin allows for misclassified points within a certain threshold, rather than strictly enforcing all points to be correctly classified.

Code to demonstrate soft margin classification

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

# Generate non-linearly separable data

X, y = datasets.make_classification(n_samples=100, n_features=2,
n_informative=2, n_redundant=0, n_clusters_per_class=1,
random_state=42)
y = np.where(y == 0, -1, 1)  # Convert labels to -1 and 1

# Create a pipeline for SVM classifier with soft margin (C=1)
svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="linear", C=1, random_state=42))
])

# Fit the classifier
svm_clf.fit(X, y)

# Plot the decision boundary
plt.figure(figsize=(10, 6))
plt.plot(X[:, 0][y==-1], X[:, 1][y==-1], "bo", label="Class -1")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "rs", label="Class 1")

# Plot decision boundary
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = svm_clf.decision_function(xy).reshape(XX.shape)
```

```
# Plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])

# Highlight support vectors
ax.scatter(svm_clf.named_steps['scaler'].inverse_transform(svm_cl
f.named_steps['svm_clf'].support_vectors_)[:, 0],
           svm_clf.named_steps['scaler'].inverse_transform(svm_cl
f.named_steps['svm_clf'].support_vectors_)[:, 1],
  s=300, linewidth=1, facecolors='none', edgecolors='k')

plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend(loc="upper left")
plt.title("Soft Margin SVM Classification")
plt.show()
```

Output:

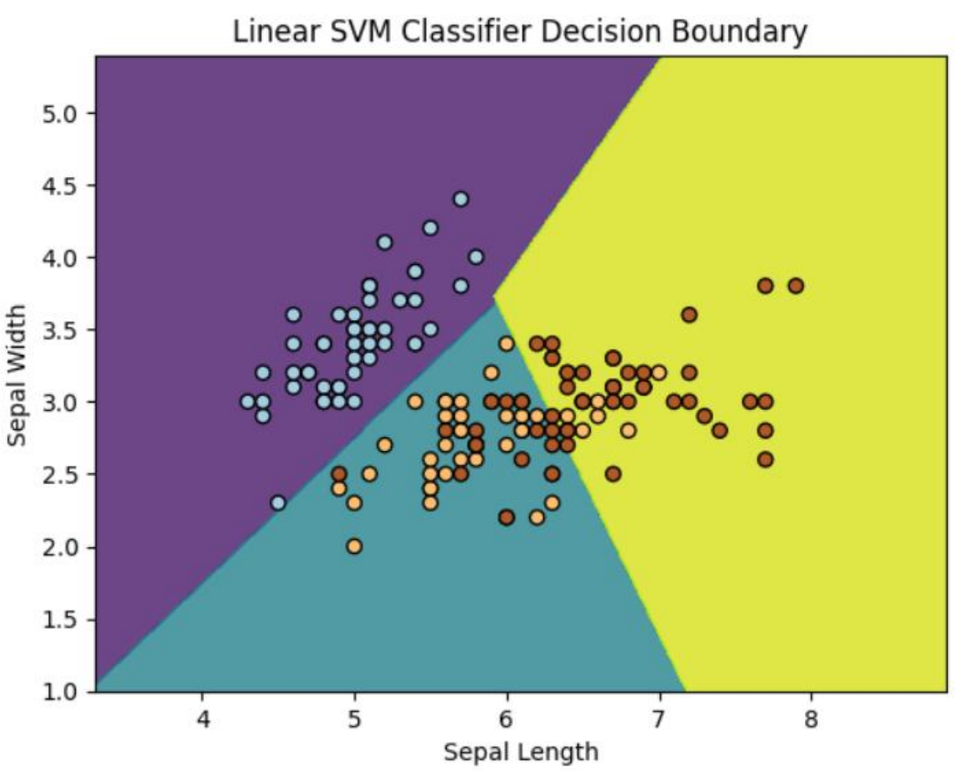


Python Code to Perform Linear SVM Classification

```
# Importing libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# Step 2: Load the Iris dataset
iris = datasets.load_iris()
```

```
X = iris.data[:, :2]  # Using only the first two features for
visualization purposes
y = iris.target

# Step 3: No preprocessing required

# Step 4: Split 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)

# Step 5: Train the Linear SVM Classifier
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)

# Step 6: Make predictions
y_pred = svm_classifier.predict(X_test)

# Step 7: Evaluate the model
accuracy = np.mean(y_pred == y_test)
print("Accuracy:", accuracy)

# Step 8: Visualize the Decision Boundary
# Create a meshgrid to plot decision boundaries
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
np.arange(y_min, y_max, 0.01))
Z = svm_classifier.predict(np.c_[xx.ravel(), yy.ravel()])

# Plot decision boundary
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.8)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
cmap=plt.cm.Paired)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Linear SVM Classifier Decision Boundary')
plt.show()
```

Output:



Nonlinear SVM Classification

- In many real-world datasets, classes are not linearly separable, meaning a straight line cannot effectively separate the data points of different classes.
- Linear classifiers like Perceptrons and Linear SVMs may fail to capture complex patterns present in the data when the decision boundary needs to be nonlinear.
- Nonlinear decision boundaries are crucial for accurately classifying such datasets, where classes may exhibit intricate structures or overlap.

- **Kernel Trick:**
  - The kernel trick is a fundamental concept in SVMs that allows them to efficiently handle nonlinear classification tasks without explicitly mapping the data into a higher-dimensional space.
  - Instead of directly transforming the input features into a higher-dimensional space, the kernel trick computes the dot product (similarity) between the transformed feature vectors implicitly.
  - Kernels are functions that compute the dot product between two vectors in the transformed space without explicitly computing the transformation. Popular kernels include the polynomial kernel, Gaussian (RBF) kernel, and sigmoid kernel.

- **Mapping to Higher-Dimensional Space:**
  - In SVMs, the idea of mapping to a higher-dimensional space is essential for achieving nonlinear decision boundaries.
  - Consider a dataset that is not linearly separable in its original feature space. By applying a nonlinear transformation (e.g., polynomial, Gaussian), the data can be mapped into a higher-dimensional space where it becomes linearly separable.
  - The decision boundary in this higher-dimensional space is a hyperplane that effectively separates the classes.
  - Despite the theoretical concept of mapping to higher-dimensional space, the kernel trick allows SVMs to implicitly compute the decision boundary in this transformed space without actually performing the expensive computation of explicitly transforming the data.

Polynomial Kernel

- The polynomial kernel computes the dot product between two vectors in a higher-dimensional space using polynomial functions.
- It has a parameter d which represents the degree of the polynomial.
- The decision boundary becomes more flexible with higher polynomial degrees, but excessively high degrees may lead to overfitting.

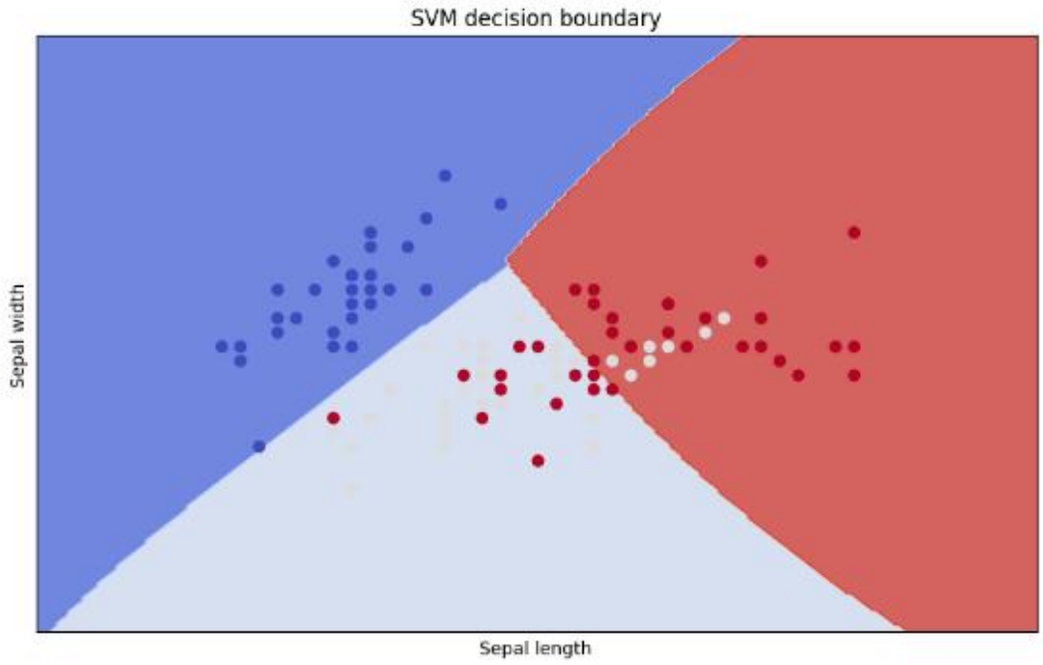
Gaussian (RBF) Kernel

- The Gaussian kernel, also known as the Radial Basis Function (RBF) kernel, transforms the data into an infinite-dimensional space.
- It measures the similarity between two samples based on the Gaussian distribution of their features.
- The kernel function is parameterized by ‘gamma’, which determines the influence of each training example on the decision boundary.

Code to Implement Non-linear SVM Classifier

```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data[:, :2]  # We'll only use the first two features for
visualization purposes
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Create a nonlinear SVM classifier with RBF kernel
svm_classifier = SVC(kernel='rbf', gamma='scale', C=1.0)

# Train the classifier
svm_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Plot decision boundary
def plot_decision_boundary(X, y, classifier):
    h = .02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)

 # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.coolwarm)
    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())
    plt.title("SVM decision boundary")

# Plot the decision boundary
plt.figure(figsize=(10, 6))
plot_decision_boundary(X_train, y_train, svm_classifier)
plt.show()
```

Output:



Introduction to Data Leakage

- Data leakage refers to the inadvertent inclusion of information in the training data that would not be available at the time of prediction.
- **Impact on SVM Training:**
  - Bias in Model Evaluation: Data leakage can lead to overly optimistic performance estimates during model evaluation, as the model may learn patterns that do not generalize to unseen data.
  - False Sense of Model Performance: Models trained on leaked data may perform well on the training and validation sets but fail to generalize to real-world data, leading to poor performance in production environments.
  - Overfitting: Models trained on leaked data are prone to overfitting, as they may capture noise or spurious correlations present in the training data.
- **Techniques to Avoid Data Leakage:**
  - Split the dataset into distinct training, validation, and testing sets.
  - Ensure that information leakage does not occur between these sets by strictly maintaining their separation.
  - Utilize techniques such as k-fold cross-validation to assess model performance while preventing leakage between folds.
  - Use evaluation metrics that are robust to data leakage, such as precision, recall, and F1 score.
  - Leverage domain expertise to identify potential sources of leakage and implement appropriate safeguards.
  - Understand the context of the data and the relationship between features to prevent unintentional leakage.

SVM Regression

- Support Vector Machine (SVM) is primarily known as a classification algorithm, but it can also be used for regression tasks.
- In SVM regression, instead of predicting discrete class labels, it predicts continuous values.
- The goal remains the same: to find the optimal hyperplane that best separates the data points.
- However, in regression, the hyperplane is used to predict the target variable's value rather than class labels.
- In SVM regression, the loss function used is typically the epsilon-insensitive loss function.
- This loss function allows for a margin of error, denoted by ε (epsilon), within which no penalty is applied to predictions. The loss function penalizes errors outside this margin.
- The epsilon-insensitive loss function is defined as:

$$L_\varepsilon(y, \hat{y}) = \begin{cases} 0 & \text{if } |y - \hat{y}| \leq \varepsilon \\ |y - \hat{y}| - \varepsilon & \text{if } |y - \hat{y}| > \varepsilon \end{cases}$$

Epsilon-insensitive loss

- The epsilon-insensitive loss function introduces a tolerance level ε, which allows the model to be less sensitive to errors within a certain range.
- This is particularly useful in regression tasks where minor deviations from the true target value might be acceptable.
- By using the epsilon-insensitive loss function, SVM regression focuses on capturing the overall trend of the data while allowing for some deviations within the specified margin ε. This makes the model more robust to outliers and noise in the data.

Practical Implementation

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error

# Load the diabetes dataset
diabetes = datasets.load_diabetes()

# Use only one feature for demonstration purposes (feature index
2)
X = diabetes.data[:, np.newaxis, 2]
y = diabetes.target

# Split 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)
```

```
# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize and train the SVR model
svr = SVR(kernel='rbf')  # RBF kernel is commonly used for SVR
svr.fit(X_train_scaled, y_train)

# Make predictions on the test set
y_pred = svr.predict(X_test_scaled)

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)

# Plot the results
plt.scatter(X_test, y_test, color='black', label='Actual')
plt.scatter(X_test, y_pred, color='red', label='Predicted')
plt.title('SVR - Diabetes Dataset')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.legend()
plt.show()
```

Output: