Machine learning tutorial

Individual assignment

Name: Vishakanan Sivarajah

ID No: 23090811

Support Vector Machines (SVM) and the Kernel Trick: Unlocking Non-Linear Patterns in Machine Learning

---

Introduction

The Support Vector Machines (SVM) algorithm in machine learning is among the best techniques applied to classification problems. Their theoretical strength, versatility, and applicability to both linearly and non-linearly separable data has made them attractive (Cortes, Vapnik and Saitta, 1995). This tutorial introduces one to some basic concepts of SVMs, the importance of the kernel trick, and their implementation in Python through the Iris dataset.

Once this is clear, we are particularly interested in understanding how the kernel trick turns SVM from a linear classifier to a strong non-linear one handling complex patterns (Schölkopf and Smola, 2001). By the end of the tutorial, one surely understands how SVMs work and implements and visualizes their work with examples utilizing both linear and non-linear kernels.

Why the Iris Dataset?

We demonstrate action SVMs by using the well-acquainted Iris dataset. The dataset is used even for teaching machine learning due to its simplicity, clarity, and relevance to real life (Fisher, 1936). The database consists of 150 observations of iris flowers with 4 measured features, namely sepal length, sepal width, petal length, and petal width. Every observation or record is labeled as belonging to one of three species, namely Setosa, Versicolor, or Virginica.

To binary classification and visualization, we are narrowing down to only two classes: Setosa and Versicolor. This simplification helps in plotting the results in two dimensions and makes the concept of SVM decision boundaries easier to grasp.

Below is a statistical summary of the filtered dataset (Setosa and Versicolor):

- Sepal Length: Mean = 5.47 cm, Std Dev = 0.64
- Sepal Width: Mean = 3.10 cm, Std Dev = 0.48
- Petal Length: Mean = 2.86 cm, Std Dev = 1.45
- Petal Width: Mean = 0.79 cm, Std Dev = 0.57

These values show how petal dimensions vary significantly, which is key to distinguishing between the two classes.

Understanding Support Vector Machines (SVM)

SVM is a supervised learning algorithm that aims to find the best possible decision boundary between different classes of data. Unlike traditional classifiers, which may focus on reducing errors, SVM attempts to maximize the margin between the closest data points of the two classes. These closest points are known as support vectors (Cortes, Vapnik and Saitta, 1995)

Imagine two groups of points on a plane. You can draw several lines to separate them, but only one line maximizes the distance to the nearest point on each side. That line is the optimal hyperplane. Mathematically, the goal is to solve an optimization problem:

$$\min \frac{1}{2}\|w\|^2 \quad \text{subject to } y_i(w \cdot x_i + b) \geq 1$$

This results in a hyperplane that is not only separating the classes but doing so with the maximum possible margin.

SVM Optimization and Slack Variables

In real-world datasets, perfect separation isn't always possible due to noise or overlapping data points. This is where **soft margin SVM** comes in. It introduces slack variables ξi\xi_iξi to allow some misclassifications while still optimizing the margin:

$$\min \frac{1}{2}\|w\|^2 + C\sum \xi_i \quad \text{subject to } y_i(w \cdot x_i + b) \geq 1 - \xi_i$$

Here, CCC is a regularization parameter that controls the trade-off between achieving a low error on training data and maintaining a large margin (Schölkopf and Smola, 2001)

The Limitations of Linearity

In many real-world scenarios, data is not linearly separable. Consider a situation where one class forms a circle, and another class lies inside it. No straight line can separate these two classes. This is a classic example of the limitation of linear models like Logistic Regression or even Linear SVMs.

This limitation necessitates a transformation of data into a higher-dimensional space where a linear separator might exist. However, performing this transformation explicitly is computationally expensive and not always practical (Schölkopf and Smola, 2001)

The Kernel Trick

The kernel trick is the answer for the non-linearity problem. With this trick, SVMs can work in a high-dimensional space of features without having to compute the transformation explicitly. In this respect, a kernel function calls the dot product of two vectors in the transformed space, providing SVMs with an efficient way of forming non-linear boundaries (Cortes, Vapnik and Saitta, 1995)

Common kernel functions include:

- Linear Kernel: $K(x_i, x_j) = x_i^T x_j$
- Polynomial Kernel: $K(x_i, x_j) = (x_i^T x_j + c)^d$
- RBF (Gaussian) Kernel: $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$
- Sigmoid Kernel: $K(x_i, x_j) = \tanh(\alpha x_i^T x_j + c)$

Among these, the RBF kernel is most used because of its ability to handle complex data distributions (Schölkopf and Smola, 2001)

Choosing the Right Kernel

Kernel	Use Case	Pros	Cons
Linear	Text, high-dimensional data	Fast, interpretable	Poor with complex patterns
Polynomial	Detecting feature interactions	Captures combinations	May overfit easily
RBF	General non-linear problems	Flexible and powerful	Needs tuning (`gamma`)
Sigmoid	Inspired by neural nets	Smooth decision surfaces	Less commonly used

---

Hyperparameter Tuning in SVM

Two critical hyperparameters are:

- **C**: Controls trade-off between margin size and misclassification.
- **Gamma**: In RBF, defines how far the influence of a single training example reaches.

```
from sklearn.model_selection import GridSearchCV
params = {'C': [0.1, 1, 10], 'gamma': [0.1, 1, 10]}
grid = GridSearchCV(SVC(kernel='rbf'), params, cv=5)
grid.fit(X_train, y_train)
print(grid.best_params_)
```

Workflow Diagram with Explanation:

The flowchart represents the fundamental workflow of Support Vector Machines (SVMs) with the kernel trick involved. It begins with input of the feature values that flow through the kernel function, implicitly mapping the data into some high-dimensional space. In that transformed space, the algorithm finds the optimal hyperplane that best separates the classes. This hyperplane is then used for predicting new or unseen data to ensure that SVM is suitably equipped to tackle not just linear but also highly complex non-linear classification problems.

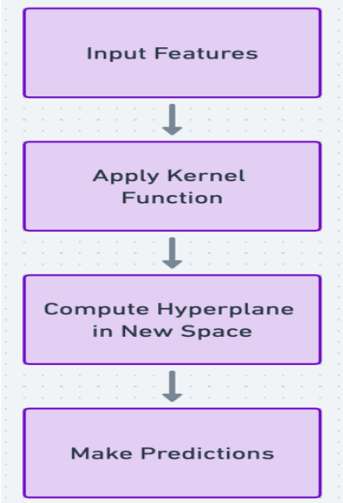

Figure 1 Workflow of SVM using the kernel trick.

Section: Step-by-Step Code Walkthrough – SVM with Kernel Trick

Now that we've covered the theory, let's walk through the **code step by step** to see how this works in practice. We'll use Python with scikit-learn, matplotlib, and seaborn, all run in **Google Colab** for accessibility and reproducibility.

Step 1: Importing Required Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix, classification_report
```

Figure 2 Showing importing required Libraries

These libraries help us load the data, train SVMs, scale features, and evaluate model performance.

Step 2: Load and Explore the Iris Dataset

```
# Load the dataset
iris = datasets.load_iris()
X = iris.data[:, :2]  # Use only first two features for 2D visualization
y = iris.target

# Filter for binary classification (Setosa vs Versicolor)
binary_filter = y < 2
X = X[binary_filter]
y = y[binary_filter]

# Preview the data
df = pd.DataFrame(X, columns=["Sepal Length", "Sepal Width"])
df["Target"] = y
df.describe()
```

Output:

	Sepal Length	Sepal Width	Target
count	100.000000	100.000000	100.000000
mean	5.471000	3.099000	0.500000
std	0.641698	0.478739	0.502519
min	4.300000	2.000000	0.000000
25%	5.000000	2.800000	0.000000
50%	5.400000	3.050000	0.500000
75%	5.900000	3.400000	1.000000
max	7.000000	4.400000	1.000000

Figure 3 Showing details about Iris Data.

Step 3: Preprocessing – Standardize Features

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Figure 4 Shows code for Preprocessing – Standardize Features

Standardizing helps ensure all features are on the same scale, which is critical for SVM performance.

Step 4: Split into Train and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42)
```

Figure 5 Shows Data splitting in test and training

We keep 80% for training and 20% for testing.

Step 5: Train Linear and RBF SVMs

Now we will train two Support Vector Machine (SVM) models: one using a **linear kernel** and the other using an **RBF (Radial Basis Function) kernel** with a specified `gamma` value of 0.5. Both models are fitted on the training data (`X_train`, `y_train`) to learn decision boundaries.

```
svm_linear = SVC(kernel='linear')
svm_linear.fit(X_train, y_train)

svm_rbf = SVC(kernel='rbf', gamma=0.5)
svm_rbf.fit(X_train, y_train)
```

Figure 6 Code for both model Training

Step 6: Visualize Decision Boundaries

Now in Below code we create a mesh grid to visualize the decision boundaries of both the linear and RBF SVM classifiers. It then predicts class labels over the grid and plots two subplots: one for the linear kernel and one for the RBF kernel. Each subplot shows the decision regions with the original data points overlaid.

```
xx, yy = np.meshgrid(
    np.linspace(X_scaled[:, 0].min() - 1, X_scaled[:, 0].max() + 1, 500),
    np.linspace(X_scaled[:, 1].min() - 1, X_scaled[:, 1].max() + 1, 500)
)
grid = np.c_[xx.ravel(), yy.ravel()]

Z_linear = svm_linear.predict(grid).reshape(xx.shape)
Z_rbf = svm_rbf.predict(grid).reshape(xx.shape)

plt.figure(figsize=(12, 5))

# Linear Kernel Plot
plt.subplot(1, 2, 1)
plt.contourf(xx, yy, Z_linear, alpha=0.3)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y, edgecolors='k')
plt.title("Linear SVM Decision Boundary")

# RBF Kernel Plot
plt.subplot(1, 2, 2)
plt.contourf(xx, yy, Z_rbf, alpha=0.3)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y, edgecolors='k')
plt.title("RBF Kernel SVM Decision Boundary")

plt.tight_layout()
plt.show()
```

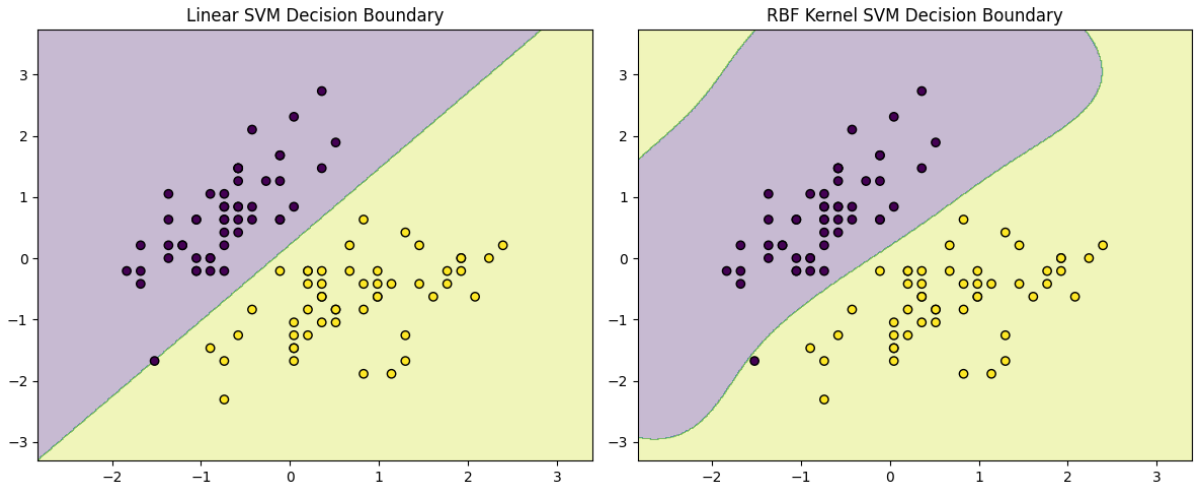

Figure 7 Decision boundary plot comparing Linear and RBF SVM.

Step 7: Evaluate with Accuracy, Precision, Recall

Now finally we will code to evaluates the RBF SVM model's performance by predicting labels for the test set. It then calculates and prints key evaluation metrics: **accuracy**, **precision**, **recall**, and a **full classification report**. These metrics help assess how well the model has classified the two iris flower classes.

```
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00         8

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```

Figure 8 Output from evaluation metrics showing perfect scores.

Step 8: Confusion Matrix

This code generates a **confusion matrix** to visualize the RBF SVM model's classification performance. It uses `seaborn.heatmap()` to display the matrix with class labels ("Setosa" and "Versicolor") on both axes. The matrix clearly shows how many predictions were correct or incorrect, with annotation and color shading for clarity.

```
conf_matrix = confusion_matrix(y_test, y_pred_rbf)

plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=["Setosa", "Versicolor"],
            yticklabels=["Setosa", "Versicolor"])
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix: RBF SVM on Iris")
plt.tight_layout()
plt.show()
```

Output:

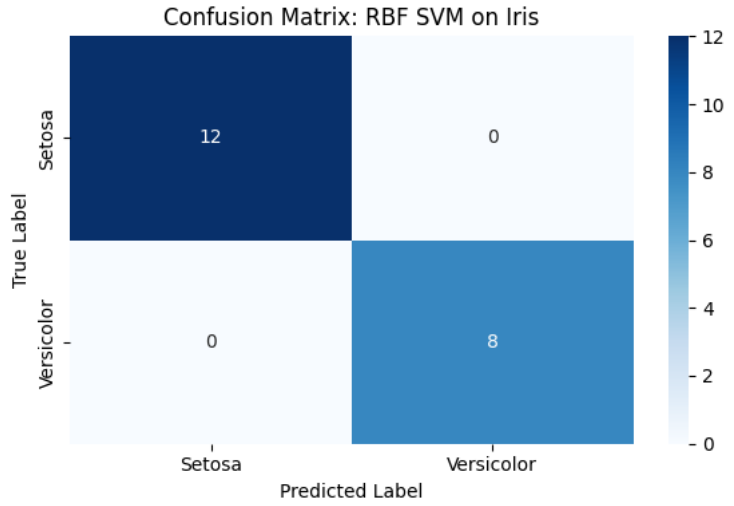

Figure 9 Confusion matrix showing 100% correct classification.

Common Misconceptions

- Linear kernels aren't bad — they are ideal for simple, separable data.
- More complexity isn't always better — high `gamma` can overfit.
- SVMs work for multi-class problems using one-vs-one or one-vs-rest strategy.

Real-World Applications

- **Text classification** (Joachims, 1998)
- **Image recognition**
- **Bioinformatics** (Brown et al., 2000)
- **Face recognition systems**

References

- Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273–297.
- Scholkopf, B., & Smola, A. J. (2001). *Learning with Kernels*. MIT Press.
- Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*.
- Joachims, T. (1998). *Text categorization with support vector machines*. ECML.
- Brown, M. P. S., et al. (2000). *Gene expression classification with SVMs*. *PNAS*.

GitHub Repository

You can visit the below links for more information:

Item	Link (Placeholder)
Github Link	https://github.com/Vishakanan/svm-kernel-trick-tutorial.git
Jupyter Notebook	https://github.com/Vishakanan/svm-kernel-trick-tutorial/blob/main/svm_kernel_tutorial_final.ipynb
README.md	https://github.com/Vishakanan/svm-kernel-trick-tutorial/blob/main/README.md
LICENSE (MIT)	https://github.com/Vishakanan/svm-kernel-trick-tutorial/blob/main/LICENSE