Minor in AI

Support Vector Machines

April 23, 2025

1 The Messy Fruit Spillover

Imagine you bought apples and oranges from the market, and a naughty kid scatters them all over your kitchen floor. Now you need to **classify** them automatically. This real-world problem introduces us to **Support Vector Machines (SVM)** - a smart way to separate different categories using mathematical boundaries.

Key Problem

How do we create the best possible boundary to separate apples (red points) from oranges (orange points) when they're mixed together?

2 The SVM Toolkit

2.1 Three Magic Components

To understand how Support Vector Machines work, you must first meet its three core concepts — like ingredients in a magic recipe for classification:

- **Hyperplane**: Think of the hyperplane as an imaginary knife slicing through your kitchen floor full of scattered fruits. It's a decision boundary that separates one class (like apples) from another (like oranges). In 2D space, this is simply a line. In 3D, it becomes a plane, and in even higher dimensions, it's called a "hyper"plane.
- **Margins**: Margins are like the safety lanes around the knife. Imagine drawing two more lines on either side of the knife, forming a no-fruit zone in between. These margins are there to give the classifier some breathing room. The wider the lane, the more confident we are that fruits on either side really belong to their respective classes.
- **Support Vectors**: These are the special fruits that touch the safety lanes. They lie closest to the hyperplane and help define where it should be placed. In fact, the hyperplane's exact position is calculated based solely on these fruits. Without them, the boundary wouldn't know where to stand!

These three components — the hyperplane, the margins, and the support vectors — form the backbone of Support Vector Machines. They work together to draw the best possible boundary between classes, especially when the fruit salad of data gets messy.

2.2 Why Wider Margins?

In the real world, data isn't clean. It's noisy, imbalanced, and unpredictable — just like when kids throw your basket of apples and oranges all over the place. That's why wider margins are not just a design choice — they are a smart strategy.

- **More robust to new fruits**: Wider margins give a buffer zone. So even if a new apple doesn't fall exactly where the old ones did, the model can still correctly classify it. It's like giving the classifier some flexibility rather than forcing it to be overly precise.

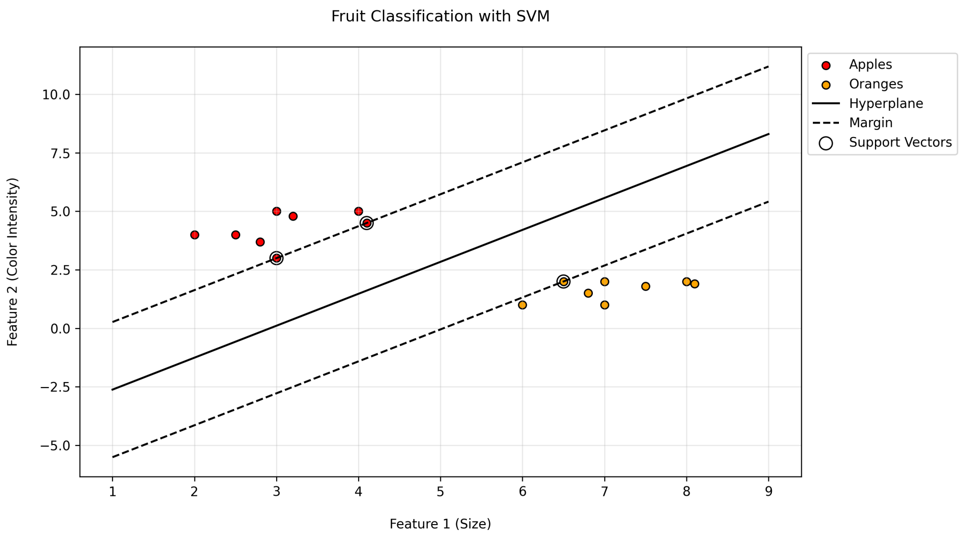


Figure 1: Visualizing SVM components (Source: Lecture Demonstration)

- **Less likely to make mistakes**: If the hyperplane is tightly packed between two classes, even a small shift in a new fruit's position could lead to a misclassification. Wider margins reduce this risk, just like having extra space on the road reduces accidents.
- **Better at handling messy real-world data**: Real datasets rarely have a clean line of separation. There may be overlapping features, noise, or outliers. Wider margins allow the model to generalize better and avoid getting misled by individual outliers that do not represent the overall trend.

In essence, wider margins are like giving your classifier a comfy pair of shoes: it walks better, stumbles less, and handles rough terrain with ease. The elegance of SVM lies in maximizing this margin while balancing misclassification to ensure reliable predictions.

3 Coding the Fruit Classifier

```python
# --- 1. Import essential libraries ---
import numpy as np                          # Numerical computing
import matplotlib.pyplot as plt             # Plotting library
from sklearn.svm import SVC                 # Support Vector Classifier

# --- 2. Define our fruit positions (features) ---
# Each point is [x, y] coordinate on the    floor
red_points    = np.array([[2, 4],         # Apple 1
                          [3, 3],         # Apple 2
                          [4, 5],         # Apple 3
                          [3, 5]])        # Apple 4
orange_points = np.array([[6, 1],         # Orange 1
                          [7, 2],         # Orange 2
                          [8, 2],         # Orange 3
                          [7, 1]])        # Orange 4

# --- 3. Combine features into single dataset X ---
# np.vstack stacks the two arrays vertically
```

```python
X = np.vstack((red_points, orange_points))
# X now has shape (8, 2): 8 samples, each with 2 features

# --- 4. Create label vector y ---
# We assign 0 to apples and 1 to oranges
y = np.array([0] * len(red_points)   +   # four zeros for apples
             [1] * len(orange_points))   # four ones  for oranges

# --- 5. Instantiate the SVM classifier ---
# kernel='linear'     learn a straightline boundary
# C=1                  regularization parameter:
#                     higher C     less margin violation allowed,
#                     lower C      softer margin (more allowance for misclassification)
model = SVC(kernel='linear', C=1)

# --- 6. Train (fit) the model to our data ---
# The SVM will find the hyperplane that maximizes margin
model.fit(X, y)

# --- 7. (Optional) Inspect the learned parameters ---
# model.coef_            weights vector w defining the hyperplane
# model.intercept_       bias term b
# model.support_vectors_     the actual points touching the margin
print("Weights (w):", model.coef_)
print("Bias (b):",   model.intercept_)
print("Support vectors:\n", model.support_vectors_)
```

Code Breakdown

- `import numpy as np / matplotlib.pyplot as plt`: Set up numerical arrays and plotting utilities.
- `from sklearn.svm import SVC`: Brings in the Support Vector Classifier.
- `red_points / orange_points`: Each row is an *(x, y)* coordinate of a fruit on the floor.
- `X = np.vstack(...)`: Combines apple and orange coordinates into a single *feature matrix* of shape (8, 2).
- `y = np.array([...])`: Creates a *label vector* of length 8: zeros for apples, ones for oranges.
- `model = SVC(kernel='linear', C=1)`: Initializes an SVM that will learn a straight-line decision boundary. `C` controls the trade-off between wide margins and training error.
- `model.fit(X, y)`: Finds the optimal hyperplane by solving a constrained optimization problem.
- `model.coef_ / model.intercept_`: After fitting, $\mathbf{w}$ and b define the equation $\mathbf{w}^\top \mathbf{x} + b = 0$.
- `model.support_vectors_`: Lists the data points that lie exactly on the margins—our "support vectors."

4 Advanced Classification with Kernels

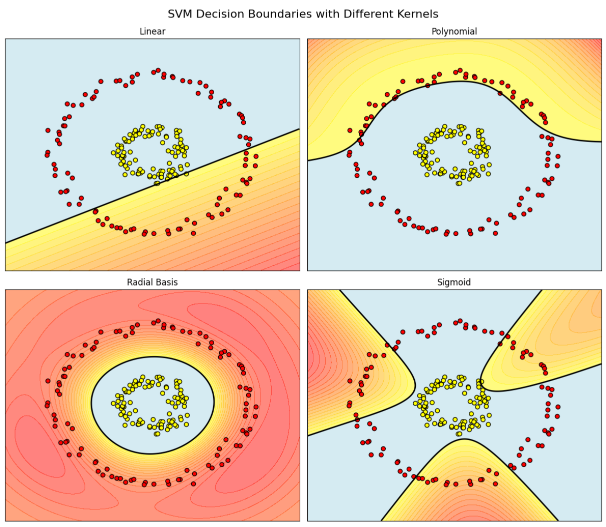
In many real-world scenarios, fruits (data points) don't lie neatly on opposite sides of a straight line. They may form concentric rings, clusters, or highly intermingled patterns. To handle such *non-linear* arrangements, SVM uses **kernels**—functions that implicitly map data into a higher-dimensional space where a simple hyperplane can separate the classes. Think of it as giving your classifier "X-ray vision" to see hidden structure.

4.1 The Kernel Superpowers

Kernel	**Use Case**	**Visualization**
Linear	Straight line separation	Basic fruit separation
Polynomial	Curved boundaries	Ripe vs unripe fruits
RBF (Radial Basis)	Complex patterns	Mixed fruit shapes
Sigmoid	Neural network prep	Advanced classifications

Choosing a Kernel:

- **Linear:** Fastest, fewest parameters. Use when you suspect a nearly straight boundary or when number of features $\gg$ number of samples.



- **Polynomial:** Introduce curvature. The degree (d) controls complexity—degree 2 or 3 is usually enough to avoid overfitting.
- **RBF:** Default for many applications. The γ parameter adjusts the "width" of each data point's influence—small γ = smoother boundary, large γ = tighter fit (risk of overfitting).
- **Sigmoid:** Less common, but bridges to neural network behavior. Two parameters (γ and intercept) govern its shape.

4.2 Example: RBF Kernel on Circular Fruit Data

```python
# 1. Generate non-linear (circular) data: fruits in rings
from sklearn import datasets
from sklearn.svm import SVC
import matplotlib.pyplot as plt

X, y = datasets.make_circles(n_samples=200,
                             factor=0.3,
                             noise=0.05,
                             random_state=42)
# 2. Instantiate  R B F kernel  SVM
#    gamma=1.0     controls how far influence of a single point reaches
model = SVC(kernel='rbf', gamma=1.0, C=1.0)

# 3. Train the model
model.fit(X, y)

# 4. Visualize decision regions
xx, yy = np.meshgrid(np.linspace(-1.5, 1.5, 300),
                     np.linspace(-1.5, 1.5, 300))
Z = model.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z > 0, alpha=0.3, cmap='autumn')
plt.contour(xx, yy, Z, levels=[0], colors='k', linewidths=2)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='autumn', edgecolors='k')
plt.title('SVM with RBF Kernel: Circular Fruit Clusters')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

Kernel Code Breakdown

- `datasets.make_circles`: Creates two concentric circles—perfect to test non-linear separators.
- `kernel='rbf'`: Chooses the radial basis function kernel.
- `gamma`: Defines the "reach" of each training sample. $\gamma \uparrow \rightarrow$ more complex boundary (risk: overfit). $\gamma \downarrow \rightarrow$ smoother boundary (risk: underfit).
- `C`: Regularization parameter as before: large C = narrow margin, small C = wider margin.
- `decision_function`: Computes distance of points to the hyperplane $\rightarrow$ used to plot contours.

With these kernel techniques, your SVM classifier can slice through even the most tangled fruit arrangements—be it concentric apples and oranges or chaotic fruit salads¡“

5 Key Takeaways

SVM Cheat Sheet

- **Always look for the *widest possible margin***
  A wider margin reduces overfitting and makes the classifier robust to noise.
- **Support vectors are your *boundary-defining heroes***
  Only points on or within the margin influence the final hyperplane. Inspect these to understand difficult cases.
- **Choose kernels based on data complexity**
  - `linear`: Fast and effective for linearly separable data.
  - `poly` (degree 2 or 3): Adds moderate curvature without overfitting.
  - `rbf`: Ideal for complex clusters like concentric circles.
  - `sigmoid`: Mimics a shallow neural network; use sparingly.
- **RBF kernel works like a *3D fruit sorter***
  It lifts tangled 2D patterns into higher dimensions, turning interlocked rings of apples and oranges into separable layers.

Food for Thought

Next time you see mixed fruits, imagine how SVM would separate them! What kernel would you use for banana-shaped clusters?