Support Vector Machines: Kernel machines

Optimal Separation

- SVMs aim to find the optimal hyperplane that best separates different classes in a feature space
  - Support vectors: Data points closest to the decision boundary
  - Hyperplane: The decision boundary that separates different classes
  - Margin: The distance between the hyperplane and the nearest data points
  - Margin Maximization: The goal is to create the widest possible margin between classes

Optimal Separation Strategies

- Linear Separability:
  - When classes can be perfectly separated by a linear boundary
  - Maximizing the geometric margin between classes
  - Minimizing classification error
- Soft Margin Classification:
  - Allows for some misclassification to handle non-linearly separable data
  - Introduces slack variables to manage classification errors
  - Balances margin width and classification accuracy

Kernels in SVM :The kernel trick & SVM algorithm

SVM is a supervised learning algorithm used for classification and regression tasks. It works by finding the hyperplane that maximally separates the classes in the feature space.

In non-linear classification problems, the classes are not separable by a linear hyperplane. To solve such problems, SVM uses the "kernel trick". The kernel trick is a mathematical technique that allows SVM to operate in a higher-dimensional space without explicitly mapping the data to that space. This is done by using a kernel function that computes the dot product of two vectors in the higher-dimensional space.

Kernel Functions

A kernel function is a mathematical function that computes the dot product of two vectors in the higher-dimensional space. Some common kernel functions are:

1. *Linear Kernel*: $K(x, y) = x^T y$
2. *Polynomial Kernel*: $K(x, y) = (x^T y + c)^d$
3. *Radial Basis Function (RBF) Kernel*: $K(x, y) = \exp(-\gamma||x - y||^2)$
4. *Sigmoid Kernel*: $K(x, y) = \tanh(\alpha x^T y + \beta)$

Optimization Problem in SVM

The goal of SVM is to find the hyperplane that maximally separates the classes in the feature space. This can be formulated as an optimization problem:

Maximize: Margin (distance between the hyperplane and the nearest data points)

Subject to: Constraints (data points are classified correctly)

Lagrange Multipliers

To solve this optimization problem, we use a technique called Lagrange multipliers. The basic idea is to introduce a new variable, called the Lagrange multiplier, which enforces the constraints.

In this case, we have a constraint for each data point:

$\sum_{i=1}^n y_i (w^T x_i + b) \geq 1$

where:

- y_i is the label of the i-th data point (+1 or -1)
- w is the weight vector
- x_i is the i-th data point
- b is the bias term

We introduce a Lagrange multiplier α_i for each constraint:
$L(w, b, \alpha) = \sum_{i=1}^n \alpha_i (y_i (w^T x_i + b) - 1)$
The Lagrange multiplier α_i can be thought of as a "penalty" term that enforces the constraint. If the constraint is satisfied, α_i is zero. If the constraint is not satisfied, α_i is non-zero and the penalty term is added to the objective function.
Now, we use the kernel trick to transform the data into a higher-dimensional space. We define a kernel function $K(x, y)$ that computes the dot product of two vectors in the higher-dimensional space:

$$K(x, y) = \varphi(x)^T \varphi(y)$$

where $\varphi(x)$ is the mapping function that transforms the data into the higher-dimensional space.

Derivation of Discriminant Function

Using the Lagrange multipliers and the kernel trick, we can derive the discriminant function as follows:

1. Compute the kernel matrix K, where $K_{ij} = K(x_i, x_j)$
2. Compute the weight vector w by solving the optimization problem:

$$w = \mathrm{argmax}_w \sum_{i=1}^n \alpha_i y_i K(x_i, x) - \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

The solution to this optimization problem is:

$$w = \sum_{i=1}^n \alpha_i y_i \varphi(x_i)$$

3. Compute the bias term b:

$$b = \sum_{i=1}^n \alpha_i y_i$$

4. The discriminant function is then given by:

$f(x) = \sum_{i=1}^n \alpha_i y_i K(x_i, x) + b$

This is the final discriminant function that is used to classify new data points.

Extensions to SVM

- Multi-class SVM (Multi-class kernel machines)
- Regression with SVMs (kernel machines for regression)

Optimization Techniques

Least Squares Optimization

- Principles:
  - Minimizing sum of squared errors
  - Finding best-fit parameters
- Applications:
  - Linear regression
  - Parameter estimation in machine learning models

Conjugate Gradient Method

- Iterative Optimization Algorithm
- Efficient for solving large-scale optimization problems
- Faster convergence compared to standard gradient descent
- Handles non-linear optimization challenges

Search Techniques

Exploration vs Exploitation

- Exploration:
  - Discovering new potential solutions
  - Investigating unknown regions of search space
- Exploitation:
  - Refining known good solutions

    - Intensifying search around promising regions

Simulated Annealing

- Probabilistic Optimization Technique
- Inspired by metallurgical annealing process
- Key Characteristics:
    - Allows accepting worse solutions with decreasing probability
    - Escapes local optima
    - Gradually reduces "temperature" to focus search

Search Strategy Components

- Initial solution generation
- Neighborhood definition
- Acceptance probability
- Cooling schedule

Simulated Annealing (SA) algorithm:

(Please refer https://github.com/bnsreenu/python_for_microscopists/blob/master/319_what_is_simulated_annealing.ipynb)

Simulated Annealing is a stochastic optimization algorithm inspired by the annealing process in metallurgy. It is used to find the global optimum of a function.

Parameters
- `x`: Initial solution
- `T`: Initial temperature
- `T_min`: Minimum temperature
- `alpha`: Cooling rate (0 < alpha < 1)
- `N`: Number of iterations

- `f(x)`: Objective function to be minimized

Algorithm
1. Initialize `x`, `T`, `T_min`, `alpha`, and `N`.
2. Evaluate `f(x)` and store the best solution `x_best` and its corresponding objective function value `f_best`.
3. For `i = 1` to `N`:
a. Generate a new solution `x_new` by applying a small perturbation to `x`.
b. Evaluate `f(x_new)`.
c. Calculate the difference `delta = f(x_new) - f(x)`.
d. If `delta < 0`, accept `x_new` as the new solution and update `x_best` and `f_best` if necessary.
e. If `delta >= 0`, accept `x_new` with probability `exp(-delta/T)`.
f. Update `T` using the cooling schedule `T = alpha * T`.
4. Return `x_best` and `f_best`.

Cooling Schedules
- `T = alpha * T` (exponential cooling)
- `T = T0 / (1 + beta * i)` (linear cooling)

Neighborhood Function
The neighborhood function generates a new solution `x_new` by applying a small perturbation to `x`. Common neighborhood functions include:

- Random mutation: `x_new = x + epsilon * randn()`
- Random walk: `x_new = x + epsilon * randn() * step_size`

Example Code
Here is an example implementation of the Simulated Annealing algorithm in Python:
```
import numpy as np

def simulated_annealing(x0, T, T_min, alpha, N, f):
    x_best = x0
```

```
    f_best = f(x0)
    x = x0
    for i in range(N):
        x_new = x + np.random.randn() * 0.1
        f_new = f(x_new)
        delta = f_new - f(x)
        if delta < 0:
            x = x_new
            if f_new < f_best:
                x_best = x_new
                f_best = f_new
        else:
            prob = np.exp(-delta / T)
            if np.random.rand() < prob:
                x = x_new
        T = alpha * T
    return x_best, f_best

Example usage
def f(x):
    return x**2 + 10 * np.sin(x)

x0 = 1.0
T = 100.0
T_min = 1.0
alpha = 0.9
N = 1000

x_best, f_best = simulated_annealing(x0, T, T_min, alpha, N, f)
print("Best solution:", x_best)
print("Best objective function value:", f_best)
```