CSCI 3485 Lab 1

Rafael Almeida

September 6, 2024

1 Report

GitHub Repository

1.1 Finding W

To find the vector w, I implemented the Perceptron algorithm introduced in class. Firstly, I define w with all 0s since the initial weights can be random. I then proceed to continuously check if the predicted class $\hat{y}$ given $w\top x^i$ where x^i is my current point equals the label y^i. In the case that $y^i > \hat{y}$, $w = w + x^i$. If $y^i < \hat{y}$, $w = w - x^i$. If $y^i = \hat{y}$ for all i, then I have found the optimal decision boundary.

1.2 Plotting W

I could not find a way to plot an equation with `matplotlib` so I used the `np.linspace` to create range of x_1 values. I then modified the expression $w_0 + w_1x_1 + w_2x_2 = 0$ to the format $x_2 = \frac{-(w_0+w_1x_1)}{w_2}$, isolating x_2. I computed the x_2 values and plotted them with the x_1 values using the `plt.plot` method.

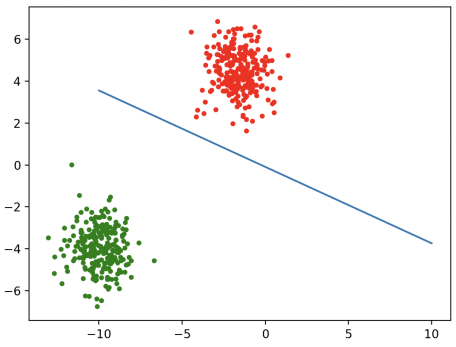

Figure 1: Green: $y = 1$. Red: $y = -1$.

1.3 Design Choices

The way I coded the algorithm is pretty straightforward and follows closely the description shown in class. The one difference is that instead of proceeding to check subsequent points whenever an inconsistency is encountered, I restart by checking the first points again. I decided to do it this way because it would allow me to make clever usage of the `Python for-else` statement, making the code more elegant. I timed my approach against the approach that more closely matches the pseudocode presented in class and saw no significant difference. Additionally, as required, I implemented a basic function `activation` and a `get_points_from_weights` that returns the x_1 and x_2 values to plot the decision boundary. Moreover, I decided to separately plot the initial scatter points to give them different colors.

2 Code

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_blobs


def activation(n: float) -> int:
    """Returns 1 if n is positive or zero, -1
    otherwise.
    """

    if n >= 0:
        return 1
    else:
        return -1


def perceptron(
    X: np.ndarray, Y: np.ndarray
) -> list[float]:
    """Trains a perceptron on the given data and returns
    the weights.
    """

    X = np.hstack((np.ones((N, 1)), X))
    w = [0 for _ in range(len(X[0]))]

    while True:
        for i in range(len(X)):
            inner_product = np.inner(w, X[i])

            y_hat = activation(inner_product)
            if Y[i] > y_hat:
                w = np.add(w, X[i])
            elif Y[i] < y_hat:
                w = np.subtract(w, X[i])
            else:
                continue

            break

        else:
            break
```

```
    return w


def get_points_from_weights(
    w: list[float],
) -> tuple[np.ndarray, np.ndarray]:
    """Creates x_1 and x_2 coordinates given weights that
    describe a decision boundary from a linear classification.
    """

    x_1 = np.linspace(-10, 10, 1000)
    x_2 = -(w[0] + w[1] * x_1) / w[2]

    return x_1, x_2


N = 500
D = 2
X, Y = make_blobs(
    n_samples=N, centers=2, n_features=D, random_state=1
)
Y = 2 * (Y - 0.5)

below = X[Y == -1]
above = X[Y == 1]
plt.scatter(below[:, 0], below[:, 1], c="r", s=10)
plt.scatter(above[:, 0], above[:, 1], c="g", s=10)

w = perceptron(X, Y)
x_1, x_2 = get_points_from_weights(w)
plt.plot(x_1, x_2)

plt.show()
```