SupportVectorMachinePython

November 10, 2023

1 State Vector Machine (SVM) Implementation

Importing necessary libraries:

- pandas for handling datasets
- matplotlib for data visualization
- NumPy for mathematical operations
- sklearn.utils -> shuffle for shuffling data
- sklearn.model_selection -> train_test_splitfor splitting datasets into a training and a testing set
- sklearn.metrics -> accuracy_score to measure the accuracy of the model

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Data pre-processing and visualization Since the Iris dataset has three classes, we will remove one of the classes. This leaves us with a binary class classification problem.

We load a dataset and remove the identifier column. Then extract the target column from the DataFrame, then remove rows with indices from 100 to 149, which contain the 'virginica' class.

Sets automatically ensure that only unique values are stored. For each value encountered in the "target" column, add that value to the "s" set.

[2]:
```
df = pd.read_csv('./datasets/iris2.csv')
df = df.drop(['Id'],axis=1) # Delete ID column
target = df['Species']
s = set()
for val in target:
    s.add(val) # Add unique values
s = list(s)
rows = list(range(100,150)) # Select virginica rows
df = df.drop(df.index[rows]) # Delete rows
```

Assign 'SepalLengthCm' and 'PetalLengthCm' to $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. slice the first 50 rows of $\boldsymbol{x}$ and $\boldsymbol{y}$, which have the Setosa label, then slice the remaining 50 rows which contain the Versicolor

label.

We initialize a new figure for the plot and create a scatter plot for the Setosa species using the sepal length (***setosa_x***) on the x-axis and petal length (***setosa_y***) on the y-axis. The data points are marked with a plus sign and are displayed in green.

Also create a scatter plot for the Versicolor species, sepal length (***versicolor_x***) on the x-axis and petal length (***versicolor_y***) on the y-axis. Mark with an underscore and display in red.

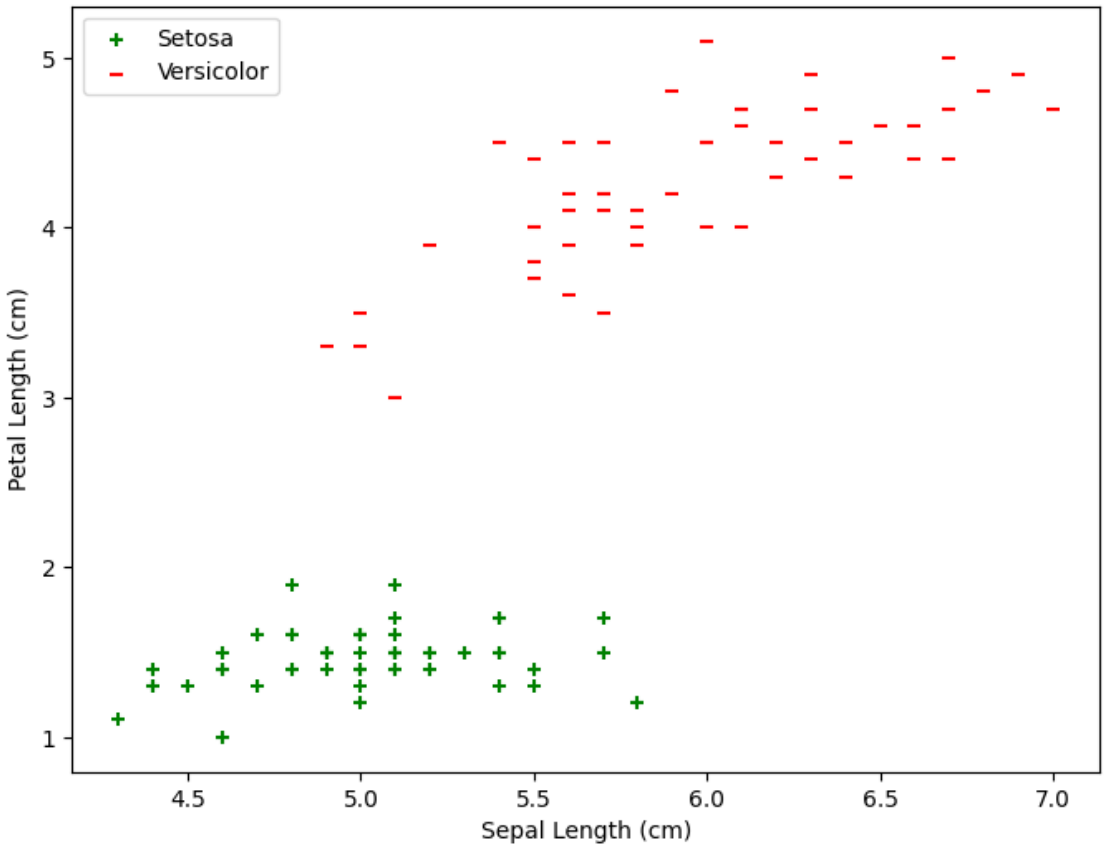
Finally, display the scatter plot.

[3]:
```
x = df['SepalLengthCm']
y = df['PetalLengthCm']

setosa_x = x[:50] # Separate the data for Setosa and Versicolor species
setosa_y = y[:50]

versicolor_x = x[50:]
versicolor_y = y[50:]

plt.figure(figsize=(8,6))
plt.scatter(setosa_x, setosa_y, marker='+', color='green', label='Setosa')
plt.scatter(versicolor_x, versicolor_y, marker='_', color='red',␣
 ↪label='Versicolor')
plt.xlabel('Sepal Length (cm)')  # X-axis label
plt.ylabel('Petal Length (cm)')  # Y-axis label
plt.legend()  # Display the legend to distinguish Setosa and Versicolor
plt.show()
```



The scatter plot displays individual data points for each flower in the dataset. The data points for Setosa and Versicolor are separated into two distinct clusters on the plot.

Observe that there is a clear separation between Setosa and Versicolor based on their sepal and petal measurements.

Training the SVM Now it's time for training. We extract the required features and split it into training and testing data.

First, drop *SepalWidthCm* and *PetalWidthCm*. The remaining features are sepal length and petal length, which we use for classification. Then create a list $\boldsymbol{Y}$ based on the *Species* column, if the species is 'Iris-setosa,' it appends -1 to $\boldsymbol{Y}$, and if it's any other species, it appends 1.

Since the *'Species'* column has been transformed into the binary labels in $\boldsymbol{Y}$ after extracting the target label, we drop it from the dataset.

In order to avoid any potential bias in the original order of the data, shuffle $\boldsymbol{X}$ and $\boldsymbol{Y}$ randomly.

The dataset is split into training and test sets. 80% of the data is used for training and 20% is used for testing. Afterwards, we convert them to NumPy arrays and reshape the ***y_train*** and ***y_test*** arrays to ensure they have the shape (n, 1).

Basically, when reshaping an array to have the shape (n, 1), (where 'n' is the number of samples), we are converting a 1-dimensional array into a 2-dimensional column vector. The '1' in (n, 1) means that we have one column of data, and 'n' rows.

> **Note:** the **'- 1'** in the reshape function tells NumPy to automatically calculate the number of rows based on the length of the input array.

[4]:
```
## Drop rest of the features and extract the target values
df = df.drop(['SepalWidthCm','PetalWidthCm'],axis=1)
Y = []
target = df['Species']
for val in target:
    if(val == 'Iris-setosa'):
        Y.append(-1)
    else:
        Y.append(1)
df = df.drop(['Species'],axis=1)
X = df.values.tolist()
## Shuffle and split the data into training and test set
X, Y = shuffle(X,Y)
x_train = []
y_train = []
x_test = []
y_test = []

x_train, x_test, y_train, y_test = train_test_split(X, Y, train_size=0.8)

x_train = np.array(x_train)
y_train = np.array(y_train)
x_test = np.array(x_test)
y_test = np.array(y_test)

y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
```

α is the learning rate, which determines the step size for updating the weight vectors ***w1*** and ***w2*** during each iteration of the gradient descent.

epochs is the number of training iterations, which sets an upper limit on how many times the SVM updates its weight vectors.

The regularization parameter is set to **'1 / epochs'**. As the number of epochs increases, the regularization value decreases

We begin by extracting *train_f1* and *train_f2*, from the training data: ***x_train[:, 0]*** extracts all the values from the first column *(feature 0)* of the x_train array. This creates a new one-dimensional array which contains the values of the first feature column for all training samples.

Similarly, ***x_train[:, 1]*** extracts all the values from the second column to create another one-dimensional array containing the values of the second feature.

These features are reshaped into column vectors to make them compatible with the SVM implementation.

Zero initialization provides a good starting point for optimization algorithms like gradient descent, hence, we initialize ***w1*** and ***w2*** with **np.zeros((80,1))** *(two-dimensional column vectors with 80 rows and 1 column)*

We enter the training loop with a maximum of 10,000 epochs. Inside the loop, calculate the decision function ***y*** as a linear combination of the features and weight vectors: $y = w1$ train_f1 + w2 * train_f2. *Then calculate the product of* **y*** and the training labels.

For each value in ***prod***, check if it's greater than or equal to 1. If it is,set the cost to 0, and update the weight vectors by subtracting the regularization term.

If ***prod*** is less than 1, calculate the cost as ***1 - val*** and update the weight vectors using gradient descent.

We use ***count*** to keep track of the training data index, and increase the ***epochs*** counter for each complete pass through the training data.

[5]:
```
# Support Vector Machine
train_f1 = x_train[:,0]
train_f2 = x_train[:,1]

train_f1 = train_f1.reshape(-1,1)
train_f2 = train_f2.reshape(-1,1)

w1 = np.zeros((80,1))
w2 = np.zeros((80,1))
# '-1' = automatically calculate the number of rows

epochs = 1
alpha = 0.0001

while(epochs < 10000):
    y = w1 * train_f1 + w2 * train_f2
    prod = y * y_train
    #print(epochs)
    count = 0
    for val in prod:
        if(val >= 1):
            cost = 0
            w1 = w1 - alpha * (2 * 1/epochs * w1)
            w2 = w2 - alpha * (2 * 1/epochs * w2)

        else:
            cost = 1 - val
            w1 = w1 + alpha * (train_f1[count] * y_train[count] - 2 * 1/epochs␣
 ↪* w1)
```

```
            w2 = w2 + alpha * (train_f2[count] * y_train[count] - 2 * 1/epochs␣
↪* w2)
        count += 1
    epochs += 1
```

Visualizing Results We now clip the weights as the test data contains only 20 data points. We extract the features from the test data and predict the values. We obtain the predictions and compare it with the actual values and print the accuracy of our model.

We remove elements 20 to 79 from the weight vectors so only 20 support vectors are retained, then reshape them to column vectors. Next, extract test data features, similarly to the training data, and reshape them to column vectors.

We then proceed to make predictions using the clipped and reshaped weight vectors and the test features.

Predictions are made based on the decision values. If a value in ***y_pred*** is greater than 1, append a 1 to the ***predictions*** list; otherwise, append -1.

For the accuracy score, we compare the predicted labels with the true labels from the test dataset, and print the result as a percentage.

[6]:
```
## Clip the weights
index = list(range(20,80))
w1 = np.delete(w1,index)
w2 = np.delete(w2,index)

w1 = w1.reshape(20,1)
w2 = w2.reshape(20,1)
## Extract the test data features
test_f1 = x_test[:,0]
test_f2 = x_test[:,1]

test_f1 = test_f1.reshape(20,1)
test_f2 = test_f2.reshape(20,1)
## Predict
y_pred = w1 * test_f1 + w2 * test_f2
predictions = []
for val in y_pred:
    if(val > 1):
        predictions.append(1)
    else:
        predictions.append(-1)

# Print accuracy
acc_text = f'Accuracy: {accuracy_score(y_test, predictions) * 100:.2f}%'
print(acc_text)
```

```
Accuracy: 100.00%
```

To better visualize the svm implementation and the results we can plot the data points and include the calculated decision boundary.

We reuse our previous scatter plot for the Setosa and Versicolor species using the Sepal Length and Petal Length. To this we generate points along the X-axis using ***np.linspace*** and calculate the corresponding Y-axis values based on the decision boundary.

The decision boundary is shown as a blue line with a solid **(-)** linestyle.

x_decision_boundary is an array of 20 equally spaced points between the minimum and maximum values of ***train_f1***. The decision_boundary values are calculated based on the weights ***w1*** and ***w2***.

$(w_1^2+w_2^2)$ is the squared magnitude of the weight vector. We then get the square root of the squared magnitude, which gives us the Euclidean norm or the length of the weight vector. $(\sqrt{w_1^2+w_2^2})$

$(\frac{1}{\sqrt{w_1^2+w_2^2}})$ is the reciprocal of the length of the weight vector. In other words, a measure of the distance from the origin to the hyperplane defined by the weight vector. In other, other words, the margin.

We calculate the upper and lower margins by adding and subtracting the margin distance from the decision boundary, respectively.

[10]:
```
# Plotting the decision boundary with automatic margin calculation
plt.figure(figsize=(8, 6))
plt.scatter(setosa_x, setosa_y, marker='+', color='green', label='Setosa')
plt.scatter(versicolor_x, versicolor_y, marker='_', color='red',␣
 ↪label='Versicolor')
plt.xlabel('Sepal Length (cm)')  # X-axis label
plt.ylabel('Petal Length (cm)')  # Y-axis label
plt.legend()

# Calculate decision boundary
x_decision_boundary = np.linspace(min(train_f1), max(train_f1), 20)
y_decision_boundary = -(w1/w2) * x_decision_boundary.reshape(-1, 1)  # Reshape␣
 ↪x_decision_boundary

# Calculate margin distance
margin_distance = 1 / np.sqrt(w1**2 + w2**2)

# Calculate margins
y_margin_upper = y_decision_boundary + margin_distance
y_margin_lower = y_decision_boundary - margin_distance

# Plot decision boundary
plt.plot(x_decision_boundary, y_decision_boundary, linestyle='-', color='blue',␣
 ↪label='Decision Boundary')

# Plot margins
```

```
plt.plot(x_decision_boundary, y_margin_upper, linestyle='--', color='gray',␣
 ↪label='Upper Margin')
plt.plot(x_decision_boundary, y_margin_lower, linestyle='--', color='gray',␣
 ↪label='Lower Margin')

# Add accuracy percentage as text annotation
plt.text(plt.xlim()[0], plt.ylim()[1], acc_text, ha='left', va='bottom',␣
 ↪backgroundcolor='white')
plt.show()
```