Support Vector Machine — Introduction to Machine Learning Algorithms

SVM model from scratch

Rohith Gandhi Follow

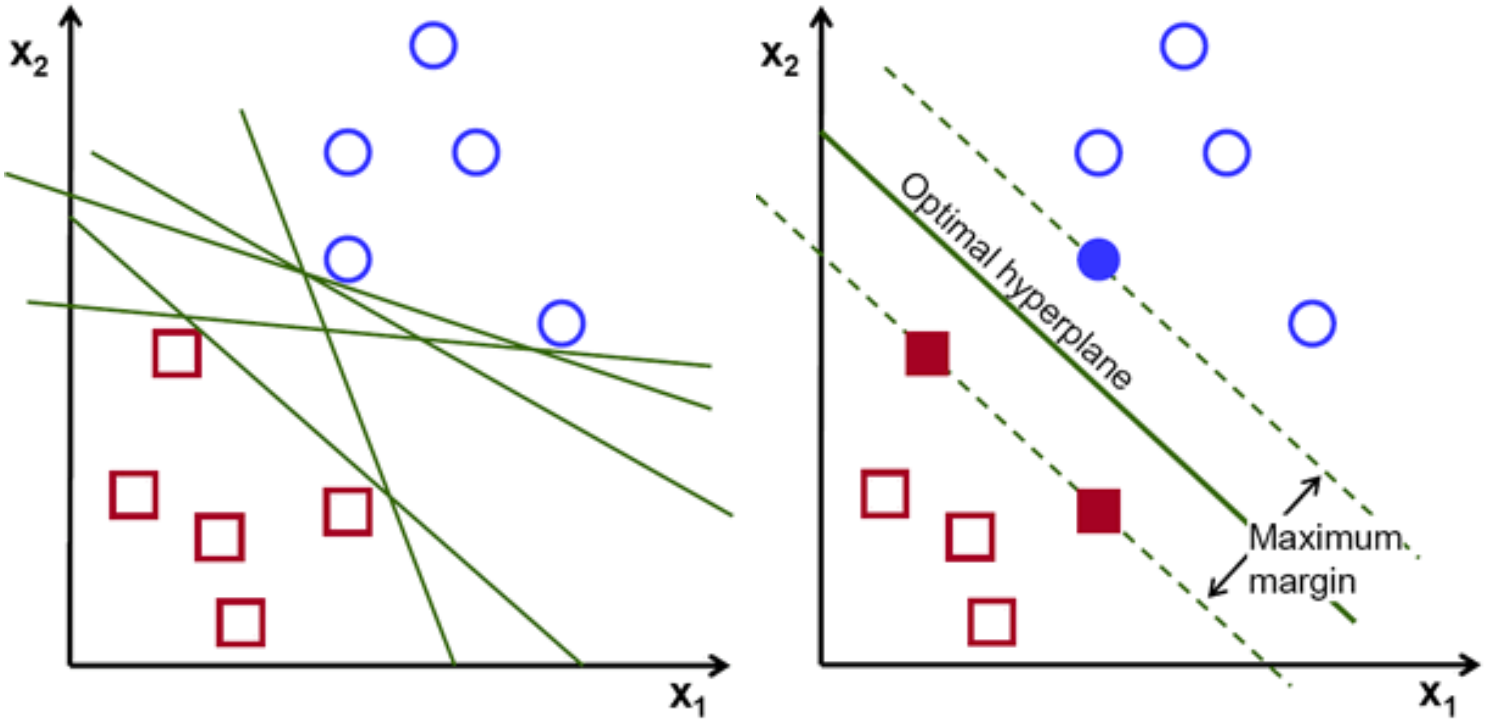
Jun 7, 2018 · 5 min read

Introduction

I guess by now you would've accustomed yourself with linear regression and logistic regression algorithms. If not, I suggest you have a look at them before moving on to support vector machine. Support vector machine is another simple algorithm that every machine learning expert should have in his/her arsenal. Support vector machine is highly preferred by many as it produces significant accuracy with less computation power. Support Vector Machine, abbreviated as SVM can be used for both regression and classification tasks. But, it is widely used in classification objectives.

What is Support Vector Machine?

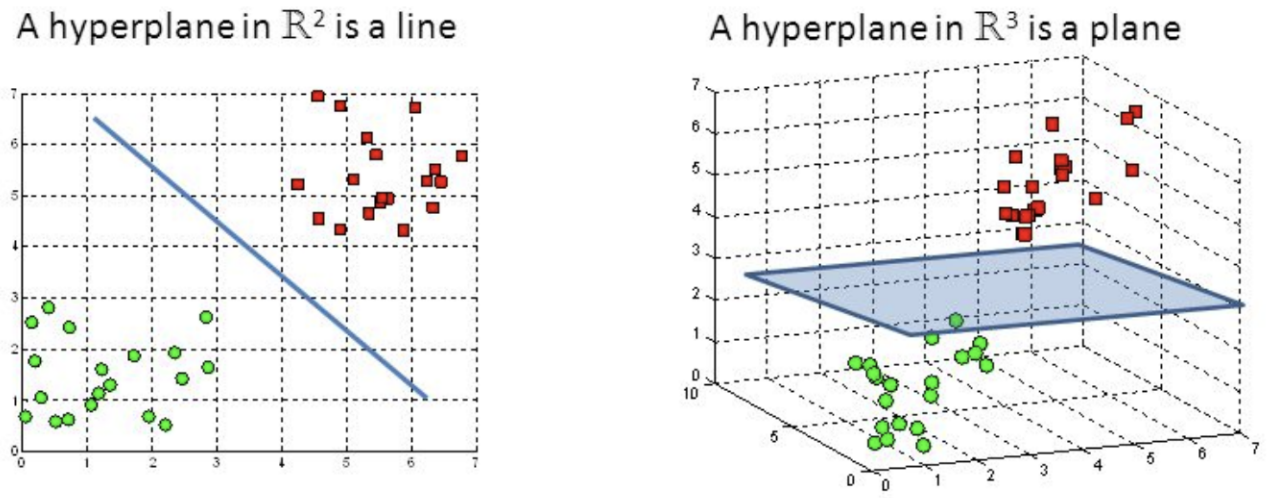
The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.



Possible hyperplanes

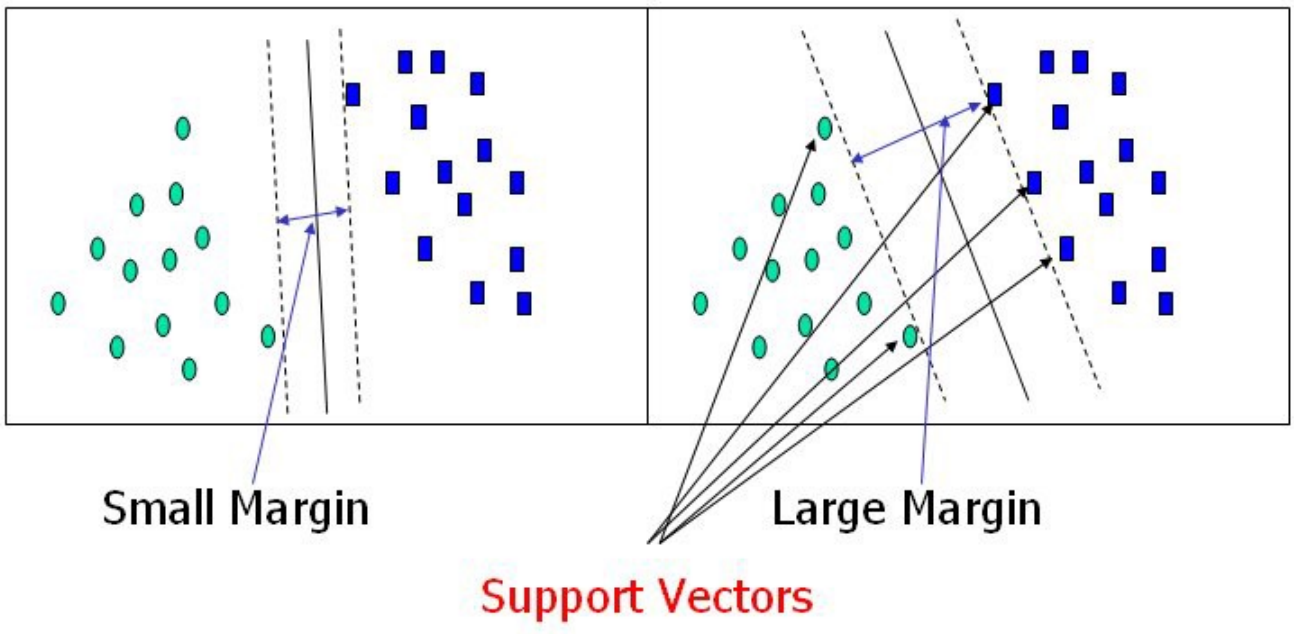
To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

Hyperplanes and Support Vectors



Hyperplanes in 2D and 3D feature space

Hyperplanes are decision boundaries that help classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.



Support Vectors

Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

Large Margin Intuition

In logistic regression, we take the output of the linear function and squash the value within the range of [0,1] using the sigmoid function. If the squashed value is greater than a threshold value(0.5) we assign it a label 1, else we assign it a label 0. In SVM, we take the output of the linear function and if that output is greater than 1, we identify it with one class and if the output is -1, we identify is with another class. Since the threshold values are changed to 1 and -1 in SVM, we obtain this reinforcement range of values([-1,1]) which acts as margin.

Cost Function and Gradient Updates

In the SVM algorithm, we are looking to maximize the margin between the data points and the hyperplane. The loss function that helps maximize the margin is hinge loss.

$$c(x, y, f(x)) = \begin{cases} 0, & \text{if } y * f(x) \geq 1 \\ 1 - y * f(x), & \text{else} \end{cases} \qquad c(x, y, f(x)) = (1 - y * f(x))_+$$

Hinge loss function (function on left can be represented as a function on the right)

The cost is 0 if the predicted value and the actual value are of the same sign. If they are not, we then calculate the loss value. We also add a regularization parameter the cost function. The objective of the regularization parameter is to balance the margin maximization and loss. After adding the regularization parameter, the cost functions looks as below.

$$min_w \lambda \parallel w \parallel^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)_+$$

Loss function for SVM

Now that we have the loss function, we take partial derivatives with respect to the weights to find the gradients. Using the gradients, we can update our weights.

$$\frac{\delta}{\delta w_k} \lambda \parallel w \parallel^2 = 2\lambda w_k$$

$$\frac{\delta}{\delta w_k} \left(1 - y_i \langle x_i, w \rangle\right)_+ = \begin{cases} 0, & \text{if } y_i \langle x_i, w \rangle \geq 1 \\ -y_i x_{ik}, & \text{else} \end{cases}$$

Gradients

When there is no misclassification, i.e our model correctly predicts the class of our data point, we only have to update the gradient from the regularization parameter.

$$w = w - \alpha \cdot (2\lambda w)$$

Gradient Update — No misclassification

When there is a misclassification, i.e our model make a mistake on the prediction of the class of our data point, we include the loss along with the regularization parameter to perform gradient update.

$$w = w + \alpha \cdot (y_i \cdot x_i - 2\lambda w)$$

Gradient Update — Misclassification

SVM Implementation in Python

The dataset we will be using to implement our SVM algorithm is the Iris dataset. You can download it from this link.

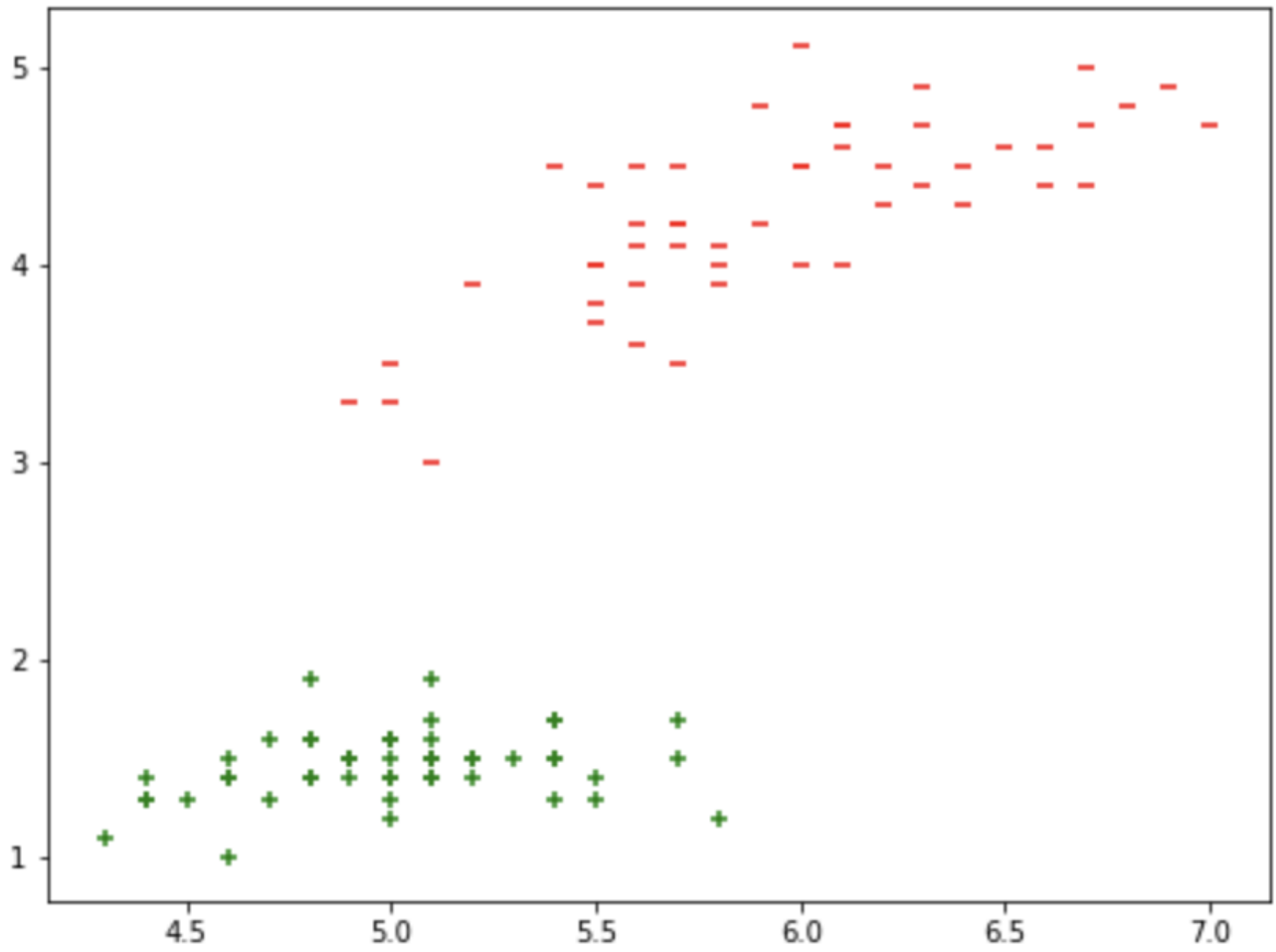
```python
import pandas as pd

df = pd.read_csv('/Users/rohith/Documents/Datasets/Iris_dataset/iris.csv')
df = df.drop(['Id'],axis=1)
target = df['Species']
s = set()
for val in target:
    s.add(val)
s = list(s)
rows = list(range(100,150))
df = df.drop(df.index[rows])

```

svm_1.py hosted with ❤ by GitHub view raw

Since the Iris dataset has three classes, we will remove one of the classes. This leaves us with a binary class classification problem.

```python
import matplotlib.pyplot as plt

x = df['SepalLengthCm']
y = df['PetalLengthCm']

setosa_x = x[:50]
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')
plt.scatter(versicolor_x,versicolor_y,marker='_',color='red')
plt.show()
```

svm_2.py hosted with ❤ by **GitHub** **view raw**



Visualizing data points

Also, there are four features available for us to use. We will be using only two features, i.e Sepal length and Petal length. We take these two features and plot them to visualize. From the above graph, you can infer that a linear line can be used to separate the data points.

```
from sklearn.utils import shuffle
from sklearn.cross_validation import train_test_split
import numpy as np
## 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=0.9)

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(90,1)
y_test = y_test.reshape(10,1)
```

svm_3.py hosted with ❤ by **GitHub** **view raw**

We extract the required features and split it into training and testing data. 90% of the data is used for training and the rest 10% is used for testing. Let's now build our SVM model using the numpy library.

```
## Support Vector Machine
import numpy as np

train_f1 = x_train[:,0]
train_f2 = x_train[:,1]

train_f1 = train_f1.reshape(90,1)
train_f2 = train_f2.reshape(90,1)

w1 = np.zeros((90,1))
w2 = np.zeros((90,1))

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
```

svm_4.py hosted with ❤ by **GitHub** **view raw**

α(0.0001) is the learning rate and the regularization parameter λ is set to 1/epochs. Therefore, the regularizing value reduces the number of epochs increases.

```
from sklearn.metrics import accuracy_score

## Clip the weights
index = list(range(10,90))
w1 = np.delete(w1,index)
w2 = np.delete(w2,index)

w1 = w1.reshape(10,1)
w2 = w2.reshape(10,1)
## Extract the test data features
test_f1 = x_test[:,0]
test_f2 = x_test[:,1]

test_f1 = test_f1.reshape(10,1)
test_f2 = test_f2.reshape(10,1)
## 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_score(y_test,predictions))
```

svm_5.py hosted with ❤ by **GitHub** **view raw**

We now clip the weights as the test data contains only 10 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.

```
Accuracy: 1.0
```

Accuracy of our SVM model

There is another simple way to implement the SVM algorithm. We can use the Scikit learn library and just call the related functions to implement the SVM model. The number of lines of code reduces significantly too few lines.

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

```
from sklearn.metrics import accuracy_score

clf = SVC(kernel='linear')
clf.fit(x_train,y_train)
y_pred = clf.predict(x_test)
print(accuracy_score(y_test,y_pred))
```

svm_6.py hosted with ❤ by **GitHub** **view raw**

Conclusion

Support vector machine is an elegant and powerful algorithm. Use it wisely :)

Machine Learning