Support Vector Machines

AcadView

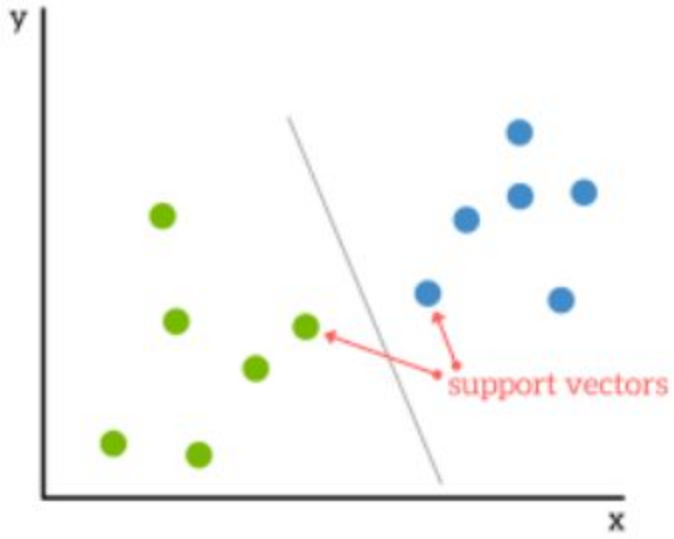
June 6, 2018

---

1 Overview

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems.

SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown in the image below.



Support Vectors Support

vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.

What is a hyperplane?

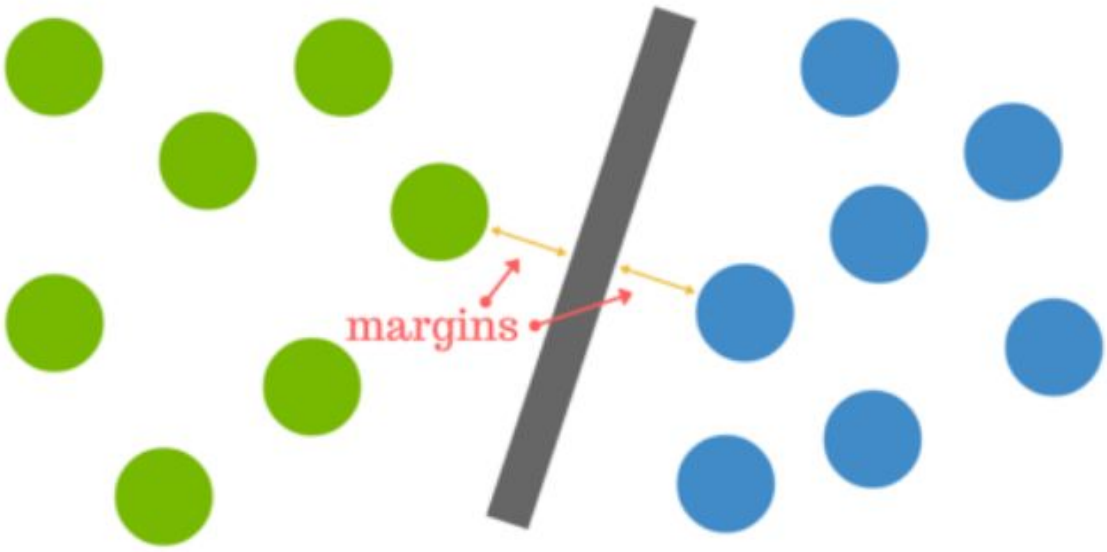
As a simple example, for a classification task with only two features (like the image above), you can think of a hyperplane as a line that linearly separates and classifies a set of data.

Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it.

So when new testing data is added, whatever side of the hyperplane it lands will decide the class that we assign to it.

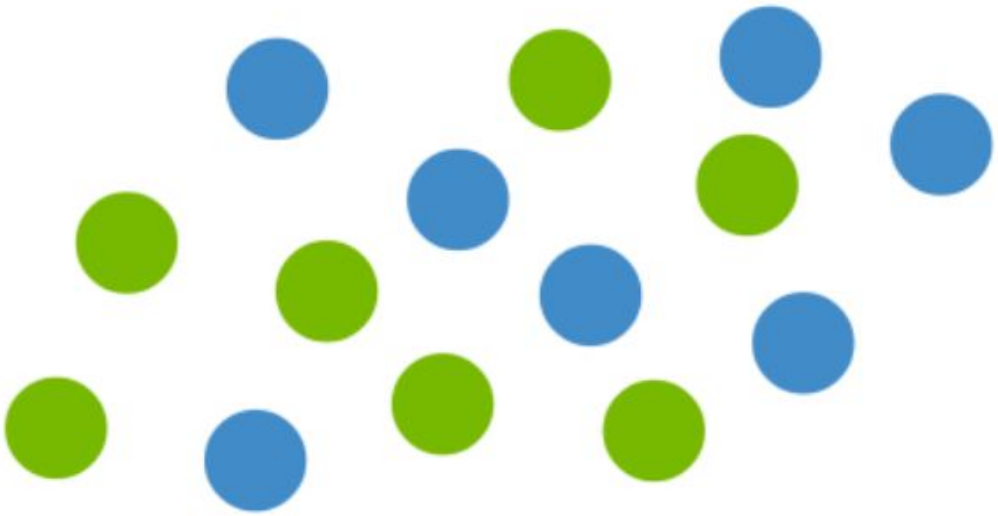
How do we find the right hyperplane?

Or in other words, how do we best segregate the two classes within the data? The distance between the hyperplane and the nearest data point from either set is known as the margin. The goal is to choose a hyperplane with the greatest possible margin between the hyperplane and any point within the training set, giving a greater chance of new data being classified correctly.

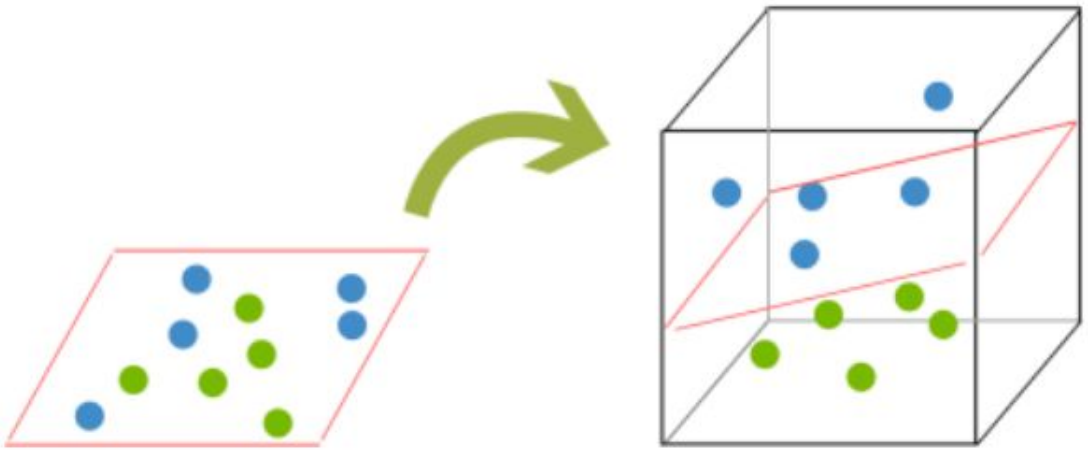


But what happens when there is no clear hyperplane?

This is where it can get tricky. Data is rarely ever as clean as our simple example above. A dataset will often look more like the jumbled balls below which represent a linearly non separable dataset.



In order to classify a dataset like the one above it's necessary to move away from a 2d view of the data to a 3d view. Explaining this is easiest with another simplified example. Imagine that our two sets of colored balls above are sitting on a sheet and this sheet is lifted suddenly, launching the balls into the air. While the balls are up in the air, you use the sheet to separate them. This 'lifting' of the balls represents the mapping of data into a higher dimension. This is known as kernelling.



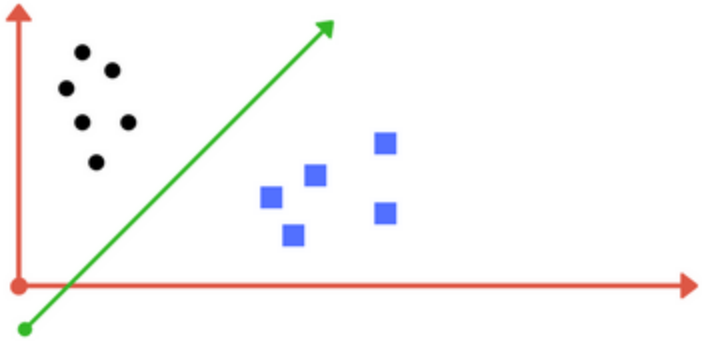
2 Support Vector Machines

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new

examples. In two dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

Suppose you are given plot of two label classes on graph as shown in image (A). Can you decide a separating line for the classes?

You might have come up with something similar to following image (image B).



It fairly separates the two classes. Any point that is left of line falls into black circle class and on right falls into blue square class. **Separation of classes.** That's what SVM does. It finds out a line/ hyper-plane (in multidimensional space that separate outs classes). Shortly, we shall discuss why I wrote multidimensional space.

2.1 The Maximal-Margin Classifier

The Maximal-Margin Classifier is a hypothetical classifier that best explains how SVM works in practice. The numeric input variables (x) in your data (the columns) form an n-dimensional space. For example, if you had two input variables, this would form a two-dimensional space.

A hyperplane is a line that splits the input variable space. In SVM, a hyperplane is selected to best separate the points in the input variable space by their class, either class 0

or class 1. In two-dimensions you can visualize this as a line and let's assume that all of our input points can be completely separated by this line. For example:

$B0+(B1 * X1) + (B2 * X2) = 0$

Where the coefficients (B1 and B2) that determine the slope of the line and the intercept (B0) are found by the learning algorithm, and X1 and X2 are the two input variables.

You can make classifications using this line. By plugging in input values into the line equation, you can calculate whether a new point is above or below the line.

• Above the line, the equation returns a value greater than 0 and the point belongs to the first class (class 0).

• Below the line, the equation returns a value less than 0 and the point belongs to the second class (class 1).

• A value close to the line returns a value close to zero and the point may be difficult to classify.

• If the magnitude of the value is large, the model may have more confidence in the prediction.

• The distance between the line and the closest data points is referred to as the margin. The best or optimal line that can separate the two classes is the line that as the largest margin. This is called the Maximal-Margin hyperplane.

The margin is calculated as the perpendicular distance from the line to only the closest points. Only these points are relevant in defining the line and in the construction of the classifier. These points are called the support vectors. They support or define the hyperplane.

The hyperplane is learned from training data using an optimization procedure that maximizes the margin.

2.2 Soft Margin Classifier

In practice, real data is messy and cannot be separated perfectly with a hyperplane.

The constraint of maximizing the margin of the line that separates the classes must be relaxed. This is often called the soft margin classifier. This change allows some points in the training data to violate the separating line.

An additional set of coefficients are introduced that give the margin wiggle room in each dimension. These coefficients are sometimes called slack variables. This increases the complexity of the model as there are more parameters for the model to fit to the data to provide this complexity.

A tuning parameter is introduced called simply C that defines the magnitude of the wiggle allowed across all dimensions. The C parameters defines the amount of violation of the margin allowed. A C=0 is no violation and we are back to the inflexible Maximal- Margin Classifier described above. The larger the value of C the more violations of the hyperplane are permitted.

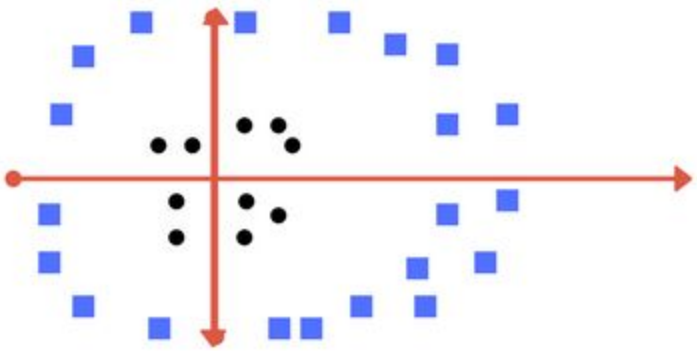
During the learning of the hyperplane from data, all training instances that lie within the distance of the margin will affect the placement of the hyperplane and are referred to as support vectors. And as C affects the number of instances that are allowed to fall within the margin, C influences the number of support vectors used by the model.

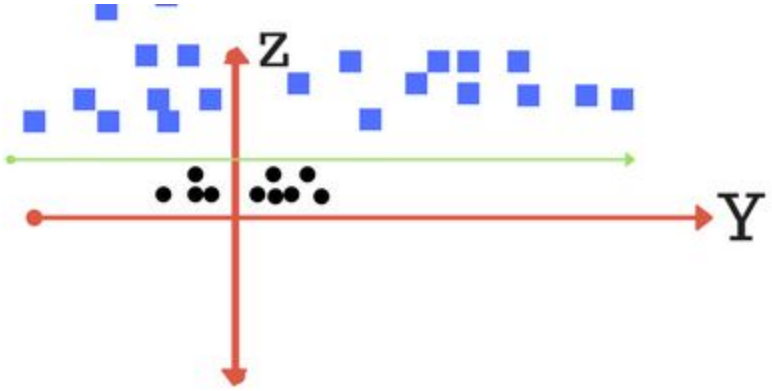
• The smaller the value of C, the more sensitive the algorithm is to the training data (higher variance and lower bias).

• The larger the value of C, the less sensitive the algorithm is to the training data (lower variance and higher bias).

3 Kernels

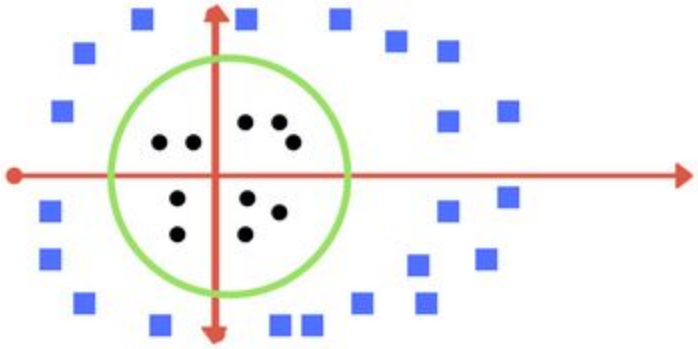
Now consider what if we had data as shown in image below? Clearly, there is no line that can separate the two classes in this x-y plane. So what do we do? We apply transformation and add one more dimension as we call it z-axis. Let's assume value of points on z plane, w = x + y. In this case we can manipulate it as distance of point from z-origin. Now if we plot in z-axis, a clear separation is visible and a line can be drawn.





IMG: Plot of zy axis. A separation can be made here.

When we transform back this line to original plane, it maps to circular boundary as shown in image below. These transformations are called **kernels.**



Transforming back to x-y plane, a line transforms to circle.

Thankfully, you don't have to guess/ derive the transformation every time for your data set. The sklearn library's SVM implementation provides it inbuilt.

The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra. This is where the kernel plays role.

For linear kernel the equation for prediction for a new input using the dot product between the input (x) and each support vector (xi) is calculated as follows:

f(x) = B(0) + sum(ai * (x,xi))

This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients B0 and ai (for each input) must be estimated from the training data by the learning algorithm.

The polynomial kernel can be written as :

K(x,xi) = 1 + sum(x * xi)^d

and exponential as

K(x,xi) = exp(-gamma * sum((x — xi²))

There is another more commonly used kernel called RBF kernel:

$$k(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

Polynomial and exponential kernels calculates separation line in higher dimension. This is called kernel trick.

4 Implementing Kernel SVM with Scikit-Learn

Implementing Kernel SVM with Scikit-Learn is similar to the simple SVM. In this section, we will use the famous iris dataset to predict the category to which a plant belongs based on four attributes: sepal-width, sepal-length, petal-width and petal-length.

The dataset can be downloaded from the following link:

https://archive.ics.uci.edu/ml/datasets/iris4

The rest of the steps are typical machine learning steps and need very little explanation until we reach the part where we train our Kernel SVM.

Importing Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Importing the Dataset

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

# Assign colum names to the dataset
colnames = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

# Read dataset to pandas dataframe
irisdata = pd.read_csv(url, names=colnames)
```

Preprocessing

```
X = irisdata.drop('Class', axis=1)
y = irisdata['Class']
```

Train Test Split

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20)
```

Training the Algorithm To train the kernel SVM, we use the same SVC class of the Scikit-Learn's svm library. The difference lies in the value for the kernel parameter of the SVC class. In the case of the simple SVM we used "linear" as the value for the kernel parameter. However, for kernel SVM you can use Gaussian, polynomial, sigmoid, or computable kernel. We will implement polynomial, Gaussian, and sigmoid kernels to see which one works better for our problem.

4.1 Polynomial Kernel

In the case of polynomial kernel, you also have to pass a value for the degree parameter of the SVC class. This basically is the degree of the polynomial. Take a look at how we can use a polynomial kernel to implement kernel SVM:

```
from sklearn.svm import SVC
svclassifier = SVC(kernel='poly', degree=8)
svclassifier.fit(X_train, y_train)
```

Making Predictions

Now once we have trained the algorithm, the next step is to make predictions on the test data. Execute the following script to do so:

```
y_pred = svclassifier.predict(X_test)
```

Evaluating the Algorithm

As usual, the final step of any machine learning algorithm is to make evaluations for polynomial kernel. Execute the following script:

```
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

The output for the kernel SVM using polynomial kernel looks like this:

```
[[11  0  0]
 [ 0 12  1]
 [ 0  0  6]]
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        11
Iris-versicolor       1.00      0.92      0.96        13
 Iris-virginica       0.86      1.00      0.92         6

    avg / total       0.97      0.97      0.97        30
```

Now let's repeat the same steps for Gaussian and sigmoid kernels.

4.2 Gaussian Kernel

Take a look at how we can use polynomial kernel to implement kernel SVM:

```
from sklearn.svm import SVC
svclassifier = SVC(kernel='rbf')
svclassifier.fit(X_train, y_train)
```

To use Gaussian kernel, you have to specify 'rbf' as value for the Kernel parameter of the SVC class.

Prediction and Evaluation

```
y_pred = svclassifier.predict(X_test)
```

```
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

The output of the Kernel SVM with Gaussian kernel looks like this:

```
[[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        11
Iris-versicolor       1.00      1.00      1.00        13
 Iris-virginica       1.00      1.00      1.00         6

    avg / total       1.00      1.00      1.00        30
```

4.3 Sigmoid Kernel

Finally, let's use a sigmoid kernel for implementing Kernel SVM. Take a look at the following script:

```
from sklearn.svm import SVC
svclassifier = SVC(kernel='sigmoid')
svclassifier.fit(X_train, y_train)
```

To use the sigmoid kernel, you have to specify 'sigmoid' as value for the kernel parameter of the SVC class.

Prediction and Evaluation

```
y_pred = svclassifier.predict(X_test)
```

```
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

The output of the Kernel SVM with Sigmoid kernel looks like this:

```
[[ 0  0 11]
 [ 0  0 13]
 [ 0  0  6]]
                precision    recall  f1-score   support

    Iris-setosa       0.00      0.00      0.00        11
Iris-versicolor       0.00      0.00      0.00        13
 Iris-virginica       0.20      1.00      0.33         6

    avg / total       0.04      0.20      0.07        30
```

4.4 Comparison of Kernel Performance

If we compare the performance of the different types of kernels we can clearly see that the sigmoid kernel performs the worst. This is due to the reason that sigmoid function returns two values, 0 and 1, therefore it is more suitable for binary classification problems. However, in our case we had three output classes.

Amongst the Gaussian kernel and polynomial kernel, we can see that Gaussian kernel achieved a perfect 100% prediction rate while polynomial kernel misclassified one instance. Therefore the Gaussian kernel performed slightly better. However, there is no hard and fast rule as to which kernel performs best in every scenario. It is all about testing all the kernels and selecting the one with the best results on your test dataset.