Data Analysis and Machine Learning: Support Vector Machines

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Nov 4, 2018

Support Vector Machines, overarching aims

A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex but small-sized or medium-sized datasets.

The case with two well-separated classes only can be understood in an intuitive way in terms of lines in a two-dimensional space separating the two classes (see figure below).

The basic mathematics behind the SVM is however less familiar to most of us. It relies on the definition of hyperplanes and the definition of a **margin** which separates classes (in case of classification problems) of variables. It is also used for regression problems.

With SVMs we distinguish between hard margin and soft margins. The latter introduces a so-called softening parameter to be discussed below. We distringuish also between linear and non-linear approaches. The latter are the most frequent ones since it is rather unlikely that we can separate classes easily by say straight lines.

Strength and weakness

When we implement a linear support vector machine, the main parameter is the constant C. Small values of C mean simple models. These models are fast to train and also fast to predict and scale to very large data sets and work well with sparse data. Linear support vector machines make it easy to understand how a prediction is made, however it is often not easy to understand why coefficients are the way they are. These models work also well in higer dimensions.

Hyperplanes and all that

The theory behind support vector machines (SVM hereafter) is based on the mathematical description of so-called hyperplanes. Let us start with a two-dimensional case. This will also allow us to introduce our first SVM examples. These will be tailored to the case of two specific classes, as displayed in the figure here.

We assume here that our data set can be well separated into two domains, where a straight line does the job in the separating the two classes. Here the two classes are represented by either crosses or circles.

What is a hyperplane

The aim of the SVM algorithm is to find a hyperplane in an n-dimensional space, where n is the number of features that distinctly classifies the data points.

In an n-dimensional space, a hyperplane is what we call an affine subspace of dimension of $n-1$. As an example, in two dimension, a hyperplane is simply as straight line while in three dimensions it is a two-dimensional subspace, or stated simply, a plane.

In two dimensions, with the variables x_1 and x_2, the hyperplane is defined as

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0,$$

In an n-dimensional space we have

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n = 0,$$

With $\hat{x} = [x_1, x_2, \ldots, x_n]$, if the above condition is not met and

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n < 0,$$

we say that $\hat{x}$ lies on one of the sides of the hyperplane and if

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n > 0,$$

then $\hat{x}$ lies on the other side.

The two-dimensional case

Let us try to develop our intuition about SVMs by limiting ourselves to a two-dimensional plane. To separate the two classes of data points, there are many possible lines (hyperplanes if you prefer a more strict naming) 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.

What a linear classifier attempts to accomplish is to split the feature space into two half spaces by placing a hyperplane between the data points. This

hyperplane will be our decision boundary. All points on one side of the plane will belong to class one and all points on the other side of the plane will belong to the second class two.

Unfortunately there are many ways in which we can place a hyperplane to divide the data. Below is an example of two candidate hyperplanes for our data sample.

Getting into the details

Let us define the function

$$f(x) = \beta_0 + \beta_1 x = 0,$$

as the function that determines the line L that separates two classes (our two features), see the figur here.

Define a vector $\hat{\beta} : \{\beta_0, \beta_1\}$. Let us label the values of $\hat{\beta}$ that satisfy this constraint as $\overline{\beta}$.

Any two points x_1 and x_2 on the line L will satisfy $\hat{\beta}(x_1 - x_2) = 0$. We normalize the solution and define

$$\overline{\beta} = \frac{\hat{\beta}}{||\hat{\beta}||},$$

which is vector normal to the line L.

The signed distance from a point x_0 on L to any point x is then

$$\overline{\beta}(x - x_0) = \frac{\beta_1 x + \beta_0}{||\hat{\beta}||}.$$

First attempt at a minimization approach

How do we find the parameters β_0 and β_0? What we could do is to define a cost function which now contains the set of all misclassified points M and attempt to minimize this function

$$C(\beta_0, \beta_1) = -\sum_{i \in M} y_i(\beta_1 x_1 + \beta_0).$$

We could now for example define all values $y_i = 1$ as misclassified in case we have $\beta_1 x_i + \beta_0 < 0$ and the opposite if we have $y_i = -1$. Taking the derivatives gives us

$$\frac{\partial C}{\partial \beta_0} = -\sum_{i \in M} y_i,$$

and

$$\frac{\partial C}{\partial \beta_1} = -\sum_{i \in M} y_i x_i.$$

Solving the equations

We can now use the Newton-Raphson method or gradient descent to solve the equations

$$\beta_0 \leftarrow \beta_0 + \eta \frac{\partial C}{\partial \beta_0},$$

and

$$\beta_1 \leftarrow \beta_1 + \eta \frac{\partial C}{\partial \beta_1},$$

where η is our by now well-known learning rate. There are however problems with this approach, although it looks pretty straightforward to implement. In case we separate our data into two distinct classes, we may up with many possible lines, as indicated in the figure and shown by running the following program. For small gaps between the entries, we may also end up needing many iterations before the solutions converge and if the data cannot be separated properly into two distinct classes, we may not experience a converge at all.

A better approach

A better approach is rather to try to define a large margin between the two classes (if they are well separated from the beginning).

Thus, we wish to find a margin M with $\hat{\beta}$ normalized to $||\hat{\beta}|| = 1$ subject to the condition

$$y_i(\beta_0 + \beta_1 x_i) \geq M \forall i = 1, 2, \dots, n.$$

All points are thus at a signed distance from the decision boundary defined by the line L. The parameters β_0 and β_1 define this line.

We seek thus the largest value M defined by

$$\frac{1}{||\hat{\beta}||} y_i(\beta_0 + \beta_1 x_) \geq M \forall i = 1, 2, \dots, n,$$

or just

$$y_i(\beta_0 + \beta_1 x_i) \geq M||\hat{\beta}|| \forall i = 1, 2, \dots, n.$$

If we scale the equation so that $||\hat{\beta}|| = 1/M$, we have to find the minimum of $\hat{\beta}^T\hat{\beta}$ subject to the condition

$$y_i(\beta_0 + \beta_1 x_i) \geq 1 \forall i = 1, 2, \dots, n.$$

A quick reminder on Lagrangian multipliers

Withe the above constraint, we introduce now the calculus of Lagrangian multiplier. In order to solve the above problem, we define the following Lagrangian function to be minimized

$$L(\lambda) = \frac{1}{2}\hat{\beta}^T\hat{\beta} - \sum_{i=1}^{n} \lambda_i \left[y_i(\beta_0 + \beta_1 x_i) - 1 \right],$$

where λ_i is a so-called Lagrange multiplier subject to the condition $\lambda_i \geq 0$.

Taking the derivatives with respect to β_0 and β_1 we obtain

$$\frac{\partial L}{\partial \beta_0} = -\sum_i \lambda_i y_i = 0,$$

and

$$\frac{\partial L}{\partial \beta_1} = \beta_1 - \sum_i \lambda_i y_i x_i = 0.$$

Inserting these constraints into the equation for L we obtain

$$L(\lambda) = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i \lambda_j y_i y_j x_i^T x_j,$$

subject to the constraints $\lambda_i \geq 0$ and $\sum_i \lambda_i y_i = 0$. We must in addition satisfy the Koriush-Kuhn-Tucker (KKT) condition

$$\lambda_i \left[y_i(\beta_0 + \beta_1 x_i) - 1 \right] \forall i = 1, 2, \ldots, n.$$

1. If $\lambda_i > 0$, then $y_i(\beta_0 + \beta_1 x_i) = 1$ and we say that x_i is on the boundary of the slab.
2. If $y_i(\beta_0 + \beta_1 x_i) > 1$, we say x_i is not on the boundary and we set $\lambda_i = 0$.

Examples with kernels

```
from IPython.display import set_matplotlib_formats, display
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import mglearn
from cycler import cycler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.datasets import make_blobs


X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.show()

from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

```
# add the squared first feature
X_new = np.hstack([X, X[:, 1:] ** 2])


from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# visualize in 3D
ax = Axes3D(figure, elev=-152, azim=-26)
# plot first all the points with y==0, then all with y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60, edgecolor='k')
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60, edgecolor='k')
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")

linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

# show linear decision boundary
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60, edgecolor='k')
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60, edgecolor='k')

ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")

ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# plot support vectors
sv = svm.support_vectors_
# class labels of support vectors are given by the sign of the dual coefficients
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

fig, axes = plt.subplots(3, 3, figsize=(15, 10))
```

```
for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
                  ncol=4, loc=(.9, 1.2))
```