Chapter8-Support-Vector-Machines

2023 年 10 月 11 日

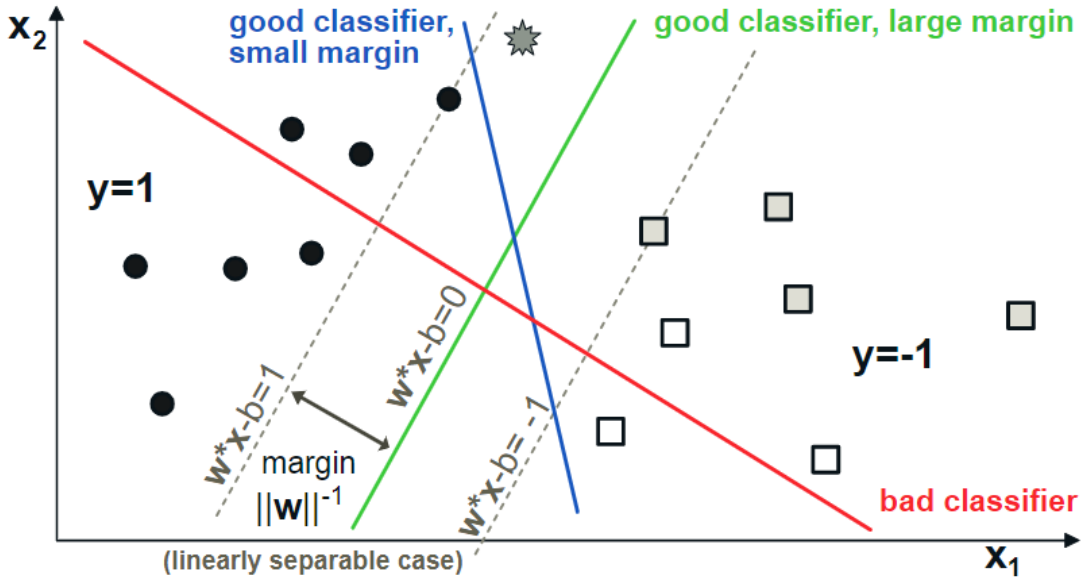
1 Support Vector Machines

The origins of SVMs are old (go back to Vapnik and Lerner (1963)). However, their modern treatment was initiated in Boser et al. (1992) and Cortes and Vapnik (1995) (binary classification) and Drucker et al. (1997) (regression). We refer to these books for an exhaustive bibliography.

1.1 SVM for Classification

Let's consider a simple case for binary classification. In the following figure (8.1), the goal is to find a model that correctly classifies points.

A model consists of two weights $\boldsymbol{w} = (w_1, w_2)$ that load on the variables and create a natural linear separation in the plane.



- The red line: a bad classifier (not discriminating different marks)
- The blue line: good

- The green line: good, but with larger margin

What about the grey star? - Given its location, should be a circle - Blue line fails to recognize it - Grey dotted lines: "margins"

The two margins are computed as the parallel lines that *maximize* the distance between the model and the *closest* points that are correctly classified (on both sides). These points are called **support vectors**, which justifies the name of the technique.

The core idea of SVMs is to **maximize the margin**, under the constraint that the classifier does not make any mistake (tries to find the most robust model).

Formally, if we numerically define circles as $+1$ and squares as -1, a good linear model is expected to satisfy

$$\begin{cases} \sum_{k=1}^{K} w_k x_{i,k} + b \geq +1 & \text{when } y_i = +1 \\ \sum_{k=1}^{K} w_k x_{i,k} + b \leq -1 & \text{when } y_i = -1 \end{cases}$$

which can be summarized in compact form as $y_i \times \left(\sum_{k=1}^{K} w_k x_{i,k} + b \right) \geq 1$.

Now, the margin between a green model and a support vector on the dashed grey line is equal to $||\boldsymbol{w}||^{-1} = \left(\sum_{k=1}^{K} w_k^2 \right)^{-1/2}$. - This value comes from the fact that the distance between a point (x_0, y_0) and a line parametrized by $ax + by + c = 0$ is $d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$. Thus, the final problem is

$$\underset{\boldsymbol{w}, b}{\operatorname{argmin}} \frac{1}{2} ||\boldsymbol{w}||^2 \quad \text{s.t. } y_i \left(\sum_{k=1}^{K} w_k x_{i,k} + b \right) \geq 1$$

The dual form of this program (see chapter 5 in Boyd and Vandenberghe (2004)) is

$$L(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2} ||\mathbf{w}||^2 + \sum_{i=1}^{I} \lambda_i \left(y_i \left(\sum_{k=1}^{K} w_k x_{i,k} + b \right) - 1 \right)$$

where either $\lambda_i = 0$ or $y_i \left(\sum_{k=1}^{K} w_k x_{i,k} + b \right) = 1$. Thus, **only some points** will matter in the solution (the so-called **support vectors**).

The first order conditions:

$$\frac{\partial L}{\partial \boldsymbol{w}} L(\boldsymbol{w}, b, \boldsymbol{\lambda}) = \mathbf{0}, \quad \frac{\partial L}{\partial b} L(\boldsymbol{w}, b, \boldsymbol{\lambda}) = 0$$

where the first condition leads to

$$\boldsymbol{w}^* = \sum_{i=1}^{I} \lambda_i u_i \boldsymbol{x}_i$$

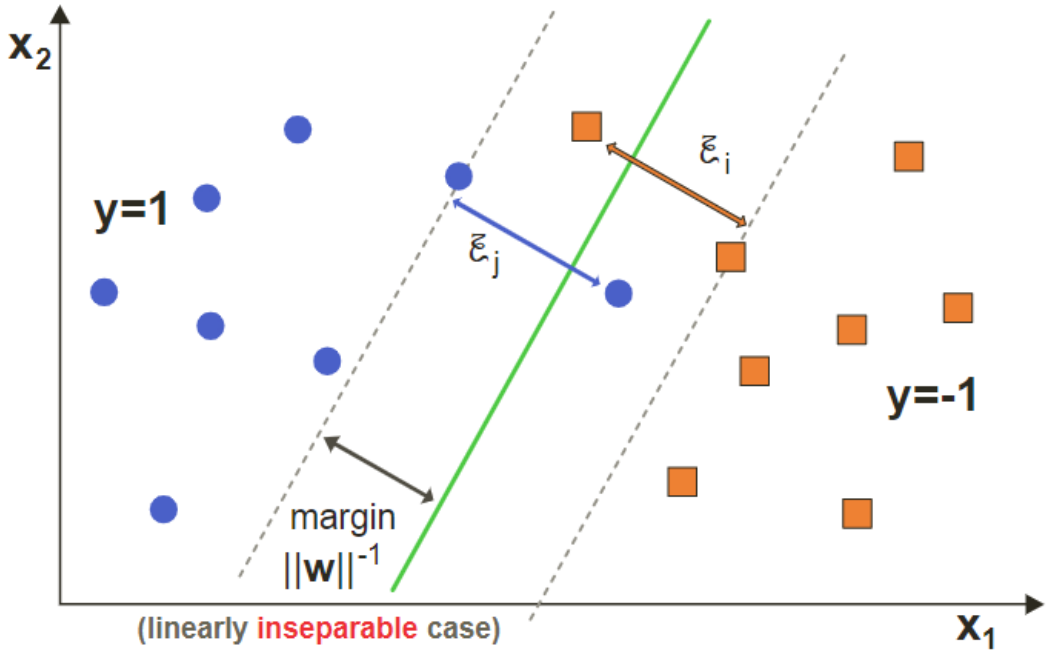
The solution is indeed a linear form of the features, but only some points are taken into account.

Naturally, this problem becomes *infeasible* whenever the condition cannot be satisfied (a simple line cannot perfectly separate the labels), i.e., logically, *not linearly separable*. This complicates

the process and it's possible to resort to a trick by **adding correction variables** that allow the conditions to be met:

$$\begin{cases} \sum_{k=1}^{K} w_k x_{i,k} + b \geq +1 - \xi_i & \text{when } y_i = +1 \\ \sum_{k=1}^{K} w_k x_{i,k} + b \leq -1 + \xi_i & \text{when } y_i = -1, \end{cases}$$

where the ξ_i are positive so-called **slack variables** that make the conditions feasible and illustrated by the following figure:



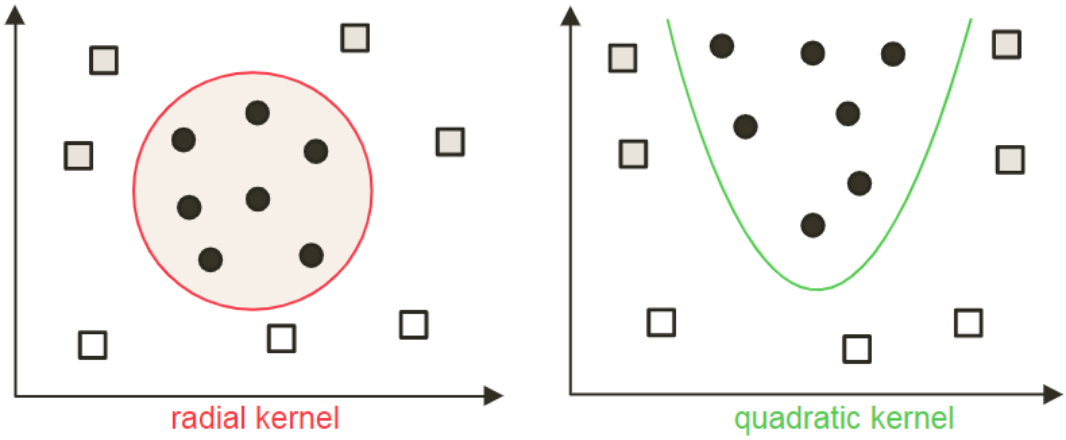
The optimization program then becomes

$$\underset{\boldsymbol{w},b,\boldsymbol{\xi}}{\text{argmin}} \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{I}\xi_i \quad \text{s.t.} \quad \left\{ y_i \left(\sum_{k=1}^{K} w_k \phi(x_{i,k}) + b \right) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0, \ \forall i \right\},$$

where the parameter $C > 0$ tunes the cost of mis-classification: as C increases, errors become more penalizing.

The program can also be generalized to *non-linear* models with kernel ϕ applied to the input points $x_{i,k}$. The following figures show non-linear kernels can help cope with patterns more complex than straight lines.

Once the weights $\boldsymbol{w}$ and bias b are set via training, a prediction for a new vector $\boldsymbol{x}_j$ is simply made by $\sum_{k=1}^{K} w_k \phi(x_{j,k}) + b$ and choosing the class based on the sign.



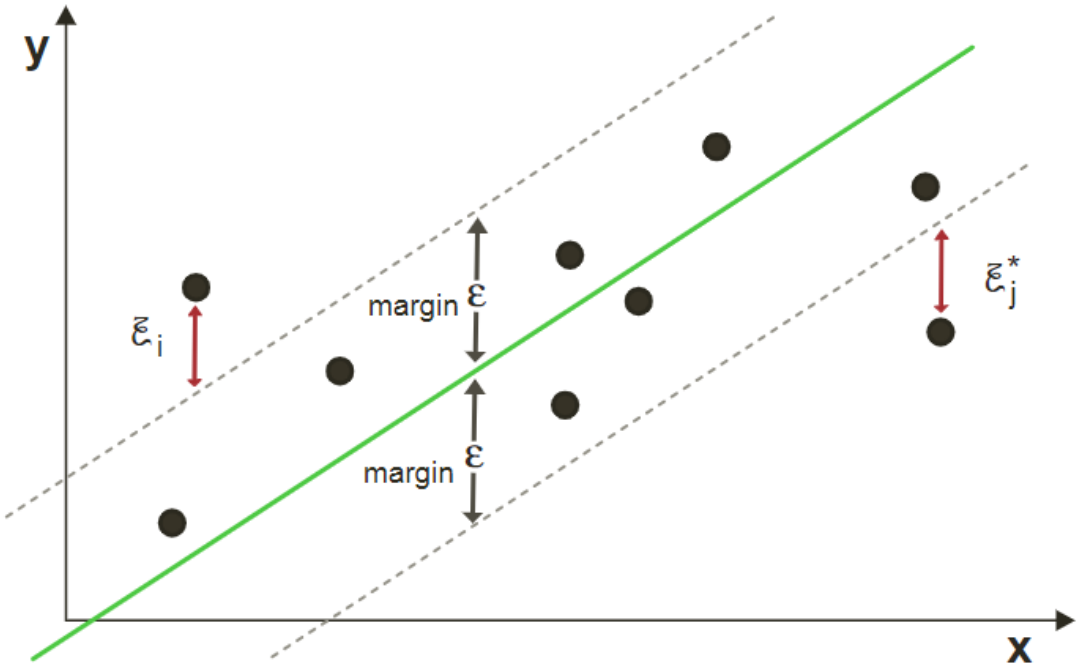
1.2 SVM for Regression

The ideas of SVM for classification can be transposed to regression but the role of the margin is different. One general formulation is

$$\begin{aligned}
\underset{\boldsymbol{w},b,\boldsymbol{\xi}}{\text{argmin}} \ & \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{I}(\xi_i + \xi_i^*) \\
\text{s.t.} \ & \sum_{k=1}^{K} w_k\phi(x_{i,k}) + b - y_i \leq \varepsilon + \xi_i \\
& y_i - \sum_{k=1}^{K} w_k\phi(x_{i,k}) - b \leq \varepsilon + \xi_i^* \\
& \xi_i, \xi_i^* \geq 0, \ \forall i
\end{aligned}$$

as illustrated in the following figure. The user specifies a **margin** ε and the model will try to find the linear (or kernel transformation) relationship between the labels y_i and the input $\boldsymbol{x}_i$. Just as in the classification task, if the data points are inside the 'strip', the slack variables ξ_i and ξ_i^* are 0.

When the points violate the threshold, the objective function is penalized (by ξ_i and ξ_i^*), and setting a large ε leaves room for more error. Once the model has been trained, a prediction for $\boldsymbol{x}_j$ is simply $\sum_{k=1}^{K} w_k\phi(x_{j,k}) + b$.



The algorithm - Minimizes the sum of squared weights $||\boldsymbol{w}||^2$ subject to the error being small enough. - The "opposite" of the penalized linear regressions which seek to minimize the error, subject to the weights being small enough.

One may refer to Chang and Lin (2011) for more details on the SVM zoo. Another reference library coded by `C` and `C++` IS `LIBSVM`.

1.3 Practice

For the sake of consistency we will use `scikit-learn`'s implementation of SVM in the following codes. In the implementation of `LIBSVM`, the package requires to specify the label and features separately. For this reason, we recycle the variables used for the boosted trees. Moreover, the training being slow, we perform it on a subsample of these sets (first thousand instances).

[ ]:
```
# Prepare data as in Chapter 6
import pandas as pd

data_ml = pd.read_pickle('./data/data_ml.pkl')
X = data_ml[data_ml.columns[2:95]]
y = data_ml['R1M_Usd']
```

```
separation_date = pd.to_datetime('2014-01-15')
training_sample = data_ml[data_ml['date'] < separation_date]
test_sample = data_ml[data_ml['date'] > separation_date]

r1m_use_quantiles = (training_sample['R1M_Usd'].quantile(0.2),␣
 ↪training_sample['R1M_Usd'].quantile(0.8))
features_short = ["Div_Yld", "Eps", "Mkt_Cap_12M_Usd", "Mom_11M_Usd", "Ocf",␣
 ↪"Pb", "Vol1Y_Usd"]

training_samples_svm = training_sample[(training_sample['R1M_Usd'] <␣
 ↪r1m_use_quantiles[0]) | (training_sample['R1M_Usd'] > r1m_use_quantiles[1])].
 ↪reset_index().drop(columns = 'index')

training_features_svm = training_samples_svm[features_short][:1000]
training_label_svm = training_samples_svm['R1M_Usd'][:1000]
test_features_svm = test_sample[features_short]
```

[ ]:
```
from sklearn.svm import SVR

fit_svm = SVR(kernel='rbf', C=0.1, epsilon=0.1, gamma=0.5)
fit_svm.fit(training_features_svm, training_label_svm)
```

[ ]:
```
SVR(C=0.1, gamma=0.5)
```

[ ]:
```
import numpy as np

np.mean(np.power(fit_svm.predict(test_features_svm) - test_sample['R1M_Usd'],␣
 ↪2))
```

[ ]:
```
0.037202909639902 71
```

[ ]:
```
np.mean(fit_svm.predict(test_features_svm) * test_sample['R1M_Usd'] > 0)
```

[ ]:
```
0.5270339562443026
```

The results are slightly better than those of the boosted trees. All parameters are completely arbitrary, especially the choice of the kernel. We finally turn to a classification example.

[ ]:
```
from sklearn.svm import SVC

training_label_svm_C = training_samples_svm['R1M_Usd_C'][:1000]

fit_svm_C = SVC(kernel='sigmoid', gamma=0.5, coef0=0.3, C=0.2)
fit_svm_C.fit(training_features_svm, training_label_svm_C)
```

[ ]:
```
SVC(C=0.2, coef0=0.3, gamma=0.5, kernel='sigmoid')
```

[ ]:
```
np.mean(fit_svm_C.predict(test_features_svm) == test_sample['R1M_Usd_C'])
```

[ ]:
```
0.49628247493163175
```

Both the small training sample and the arbitrariness in our choice of the parameters may explain why the predictive accuracy is so poor.

1.4 Coding Exercises

1. From the simple example shown above, extend SVM models to other kernels and discuss the impact on the fit.

[ ]:
```
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
def evaluate_kernel_fit(kernel,
                        training_features_svm, training_label_svm,
                        test_features_svm, test_label_svm):
    # Fit a SVM for regression with the specified kernel
    fit_svm = SVR(kernel=kernel, C=0.1, epsilon=0.1, gamma=0.5)
    fit_svm.fit(training_features_svm, training_label_svm)

    mse = np.mean(np.power(fit_svm.predict(test_features_svm) - test_label_svm,␣
 ↪2))
    hit_ratio = np.mean(fit_svm.predict(test_features_svm) * test_label_svm > 0)
    return mse, hit_ratio
```

[ ]:
```
fit_results = {kernel: evaluate_kernel_fit(kernel, training_features_svm,
               training_label_svm, test_features_svm, test_sample['R1M_Usd'])
               for kernel in kernels}
fit_results
```

[ ]:
```
{'linear': (0.03705349279047895, 0.5292701686417502),
 'poly': (0.037330909903200316, 0.5269769826800365),
 'rbf': (0.03720290963990271, 0.5270339562443026),
 'sigmoid': (0.2903700864598458, 0.49175307657247036)}
```

The first two kernels yield the best fit, while the last one should be avoided. Note that apart from the linear kernel, all other options require parameters. We have used the default ones, which may explain the poor performance of some nonlinear kernels.

2. Train a vanilla SVM model with labels being the 12-month forward (i.e., future) return and evaluate it on the testing sample. Do the same with a simple random forest. Compare.

[ ]:
```
training_label_svm_12 = training_samples_svm['R12M_Usd'][:1000]
evaluate_kernel_fit('linear', training_features_svm, training_label_svm_12,
                    test_features_svm, test_sample['R12M_Usd'])
```

[ ]:
```
(0.24691967843508394, 0.48504443938012765)
```

The vanilla model seems unsatisfying when dealing with 12-month forward return. Let's try a simple random forest.

[ ]:
```
from sklearn.ensemble import RandomForestRegressor

fit_rf_12 = RandomForestRegressor(n_estimators=100)
fit_rf_12.fit(training_features_svm, training_label_svm_12)

mse_rf = np.mean(np.power(fit_rf_12.predict(test_features_svm) -␣
 ↪test_sample['R12M_Usd'], 2))
hit_ratio_rf = np.mean(fit_rf_12.predict(test_features_svm) *␣
 ↪test_sample['R12M_Usd'] > 0)
mse_rf, hit_ratio_rf
```

[ ]:
```
(0.3208760528389372, 0.48817798541476753)
```

The MSE from the simple random forest seems to be more unsatisfying than a vanilla SVM with linear kernel. However, the hit ratio does improve a little. It's clear that the small training sample, the arbitrariness in the choice of the parameters, and the forecasting horizon could explain the poor performances of both models.