Chapter 5. Support Vector Machines

A *support vector machine* (SVM) is a powerful and versatile machine learning model, capable of performing linear or nonlinear classification, regression, and even novelty detection. SVMs shine with small to medium-sized nonlinear datasets (i.e., hundreds to thousands of instances), specially for classification tasks. However, they don't scale very well to very large datasets, as you will see.

Overview:

- LinearSVM
  - Hard margin vs Soft margin
- Non-Linear SVM
- SVM Regression
- Under the Hood

Linear SVM Classification

Large Margin Classifier: an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes.

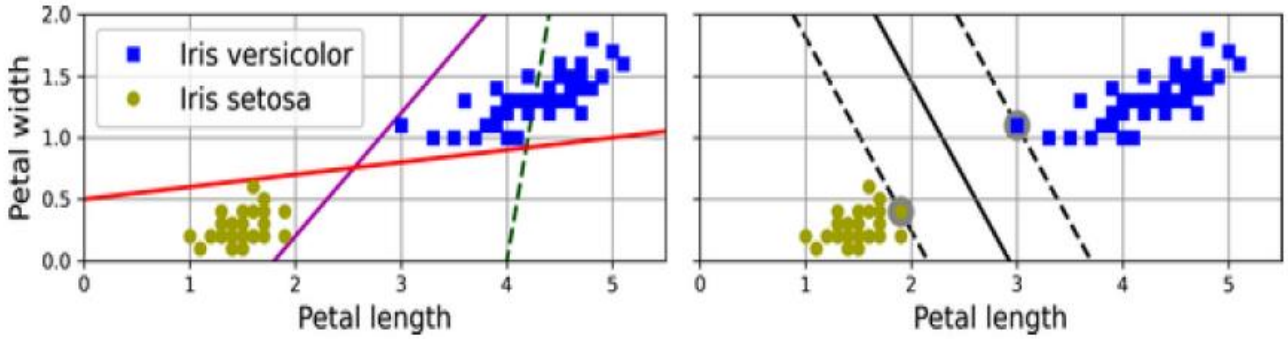


Figure 5-1. Large margin classification

- The two classes can clearly be separated easily with a straight line (they are *linearly separable*).
- The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly.
- The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances.

- In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible.

- Notice that adding more training instances "off the street" will not affect the decision boundary at all: it is fully determined (or "supported") by the instances located on the edge of the street. These instances are called the **support vectors** (they are circled in Figure 5-1).

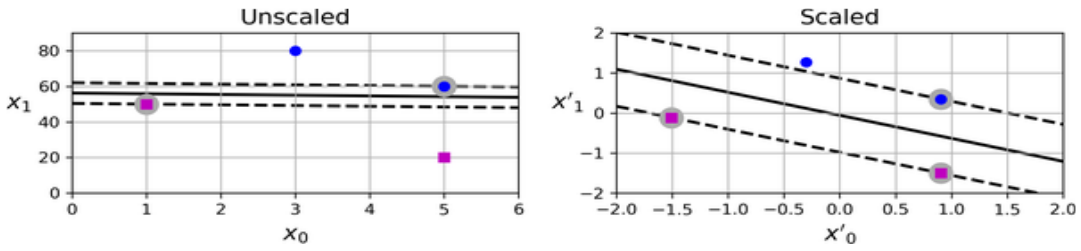


Figure 5-2. Sensitivity to feature scales

- SVMs are sensitive to the feature scales, as you can see in Figure 5-2. In the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal.
- After feature scaling (e.g., using Scikit-Learn's StandardScaler), the decision boundary in the right plot looks much better.

Hard Margin Classification: If we Strictly impose that all instances must be off the streets and on the right Side

Issues:

1- It only works if the data is linearly separable
2- It is sensitive to outliers

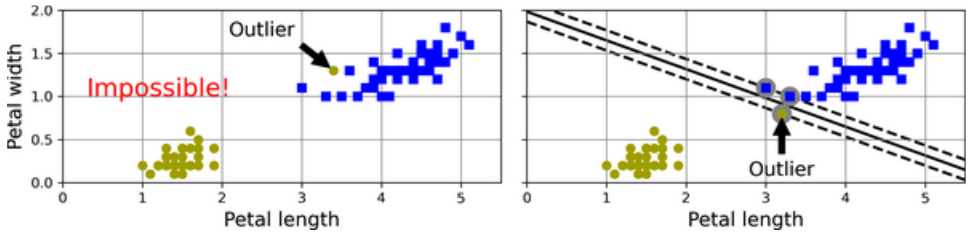


Figure 5-3. Hard margin sensitivity to outliers

- On the left, it is impossible to find a hard margin
- On the right, the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier.
- The model will probably not generalize as well.

Soft margin Classification

To avoid these issues, we need to use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side).

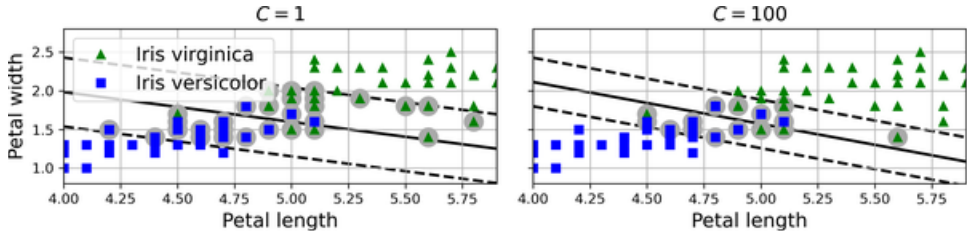


Figure 5-4. Large margin (left) versus fewer margin violations (right)

- **C** is one of SVM Hyperparameters.
- If we set it to a low value, then you end up with the model on the left of Figure 5-4.
- With a high value, you get the model on the right.
- reducing C makes the street larger, but it also leads to more margin violations. In other words, reducing C results in more instances supporting the street, so there's less risk of overfitting.
- But if you reduce it too much, then the model ends up underfitting.

TIP: If your SVM model is overfitting, you can try regularizing it by reducing C.

The following Scikit-Learn code loads the iris dataset and trains a linear SVM classifier to detect *Iris virginica* flowers. The pipeline first scales the features, then uses a LinearSVC with C=1:

```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2)  # Iris virginica

svm_clf = make_pipeline(StandardScaler(),
                        LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)
```

```
>>> X_new = [[5.5, 1.7], [5.0, 1.5]]
>>> svm_clf.predict(X_new)
array([ True, False])
```

The first plant is classified as an Iris virginica, while the second is not.

Let's look at the scores that the SVM used to make these predictions. These measure the signed distance between each instance and the decision boundary.

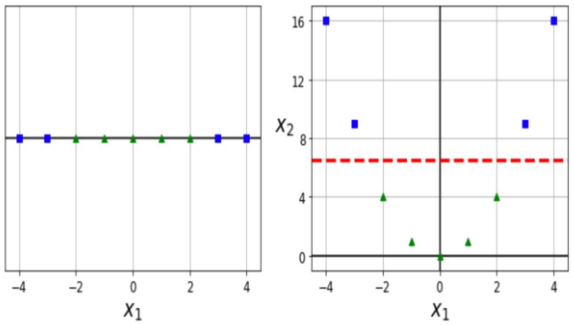
```
>>> svm_clf.decision_function(X_new)
array([ 0.66163411, -0.22036063])
```

- Unlike LogisticRegression, LinearSVC doesn't have a **predict_proba()** method to estimate the class probabilities.
- if you use the **SVC** class instead of **LinearSVC**, and if you set its **probability** hyperparameter to **True**, then the model will fit an extra model at the end of training to map the SVM decision function scores to estimated probabilities.
- Under the hood, this requires using **5-fold crossvalidation** to generate out-of-sample predictions for every instance in the training set, then training a **LogisticRegression** model, so it will slow down training considerably.
- After that, the **predict_proba()** and **predict_log_proba()** methods will be available.

- LinearSVC:
  - LinearSVC(loss="hinge", C=1)
  - Accepts two loss functions: "hinge" and "squared_hinge"
  - Default loss is "squared_hinge"
  - Doesn't output support vectors (use .intercept_ & .coef_ to find support vectors in the training data)
  - Regularizes bias term too...so center the data by using StandardScaler
  - Set dual=False if training instances > number of features
- SVC:
  - SVC(kernel="linear", C=1)
  - For linear classifier use kernel="linear"
  - For hard margin classifier use C=float("inf")
- SGDClassifier:
  - SGDClassifier(loss="hinge", alpha = 1/(m*C))
  - Slow to converge, but good for online or huge datasets(out-of-core Learning)

Nonlinear SVM Classification

- Many Datasets are not even close to being linearly separable
- One approach to handle nonlinear datasets is to add more features such as polynomial features



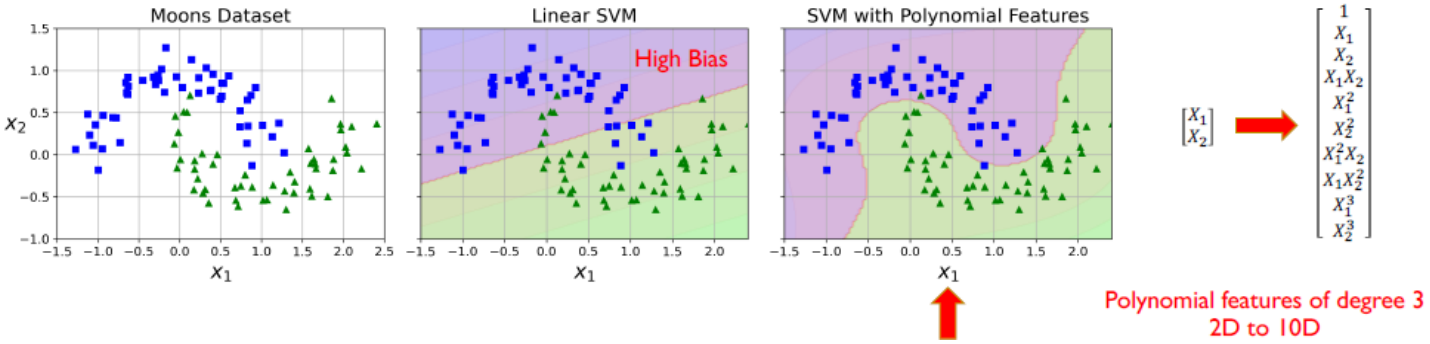
- Moons Dataset: a toy dataset for binary Classification in which the data points are shaped as two interleaving half circles
- You can generate the dataset using the makemoons() function

```
from sklearn.datasets import make_moons
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

polynomial_svm_clf = make_pipeline(
    PolynomialFeatures(degree=3),
    StandardScaler(),
    LinearSVC(C=10, max_iter=10_000, random_state=42)
)
polynomial_svm_clf.fit(X, y)
```

Polynomial kernels



Two ways to implement Polynomial Features

Use `PolynomialFeatures`, `StandardScaler`, and `LinearSVC`

Explicitly adds polynomial features

Same results without adding features

Use `SVC` with `kernel = "poly"`

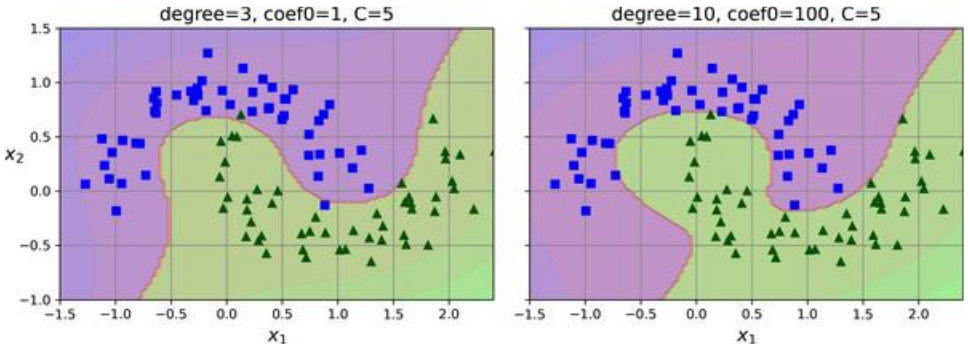
Kernel trick: adding the effect of high dimensional features without explicitly adding them.

Using SVM Kernel “poly”

```
from sklearn.svm import SVC

poly_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(kernel="poly", degree=3, coef0=1, C=5))
poly_kernel_svm_clf.fit(X, y)
```

- **(d) degree** of polynomial features, If your model is overfitting reduce the degree of polynomial kernel, conversely, if it is underfitting you can try increasing it
- **coef0(r)** is polynomial kernel Hyperparameter, Coef0 controls how much the model is influenced by high degree polynomials versus low-degree polynomials
- **C** is the soft margin Hyperparameter
- Use **grid search** for tuning hyperparameters
    - Coarse grid search first
    - Followed by finer Grid search
- Having a good sense of what each parameter actually does can also help you search in the right part of the hyperparameter space



Similarity Features

- Similarity Function, Another way to tackle nonlinear problems
- Select landmarks from training instances and evaluate the similarity of rest of the instances to landmark
- Using the Gaussian Radial Basis Function(RBF)

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma\|\mathbf{x} - \ell\|^2\right)$$

Gaussian Radial Basis Function (RBF)

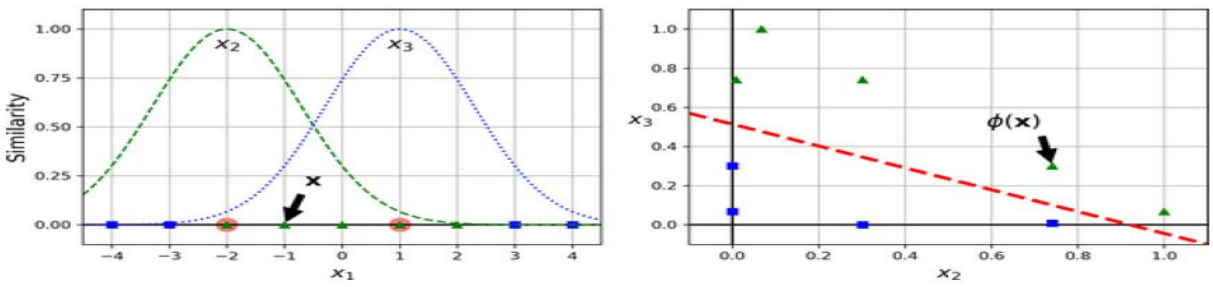


Figure 5-8. Similarity features using the Gaussian RBF

- How to select landmarks?
  - The simplest approach is to create a landmark at the location of each and every instance in the dataset
  - This creates many dimensions, increasing the chance that the transformed training set will be linearly separable
  - The downside is a training set with m instances and n Features will transform to a training set with m instances and m features(assuming you drop the features)

Gaussian RBF kernel

- Use SVM kernel

```
rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
                                   SVC(kernel="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

- Increasing **Gamma(y)** makes the bell-shaped curve narrower(fig 5-8); each instance's range of influence is smaller, the decision boundary ends up being more irregular, wiggling around individual instances
- A small **gamma** value makes the bell-shaped curve wider; the decision boundary ends up smoother
- **Y** acts like a regularization hyperparameter, reduce if overfitting and increase if underfitting
- **String kernels** are used in classifying text document or DNA sequence
- **LinearSVC** is much faster than **SVC(kernel="linear"),** especially if the training set is very large or it has many features
- You may also want to try gaussian RBF kernel, if the training set isn't too large

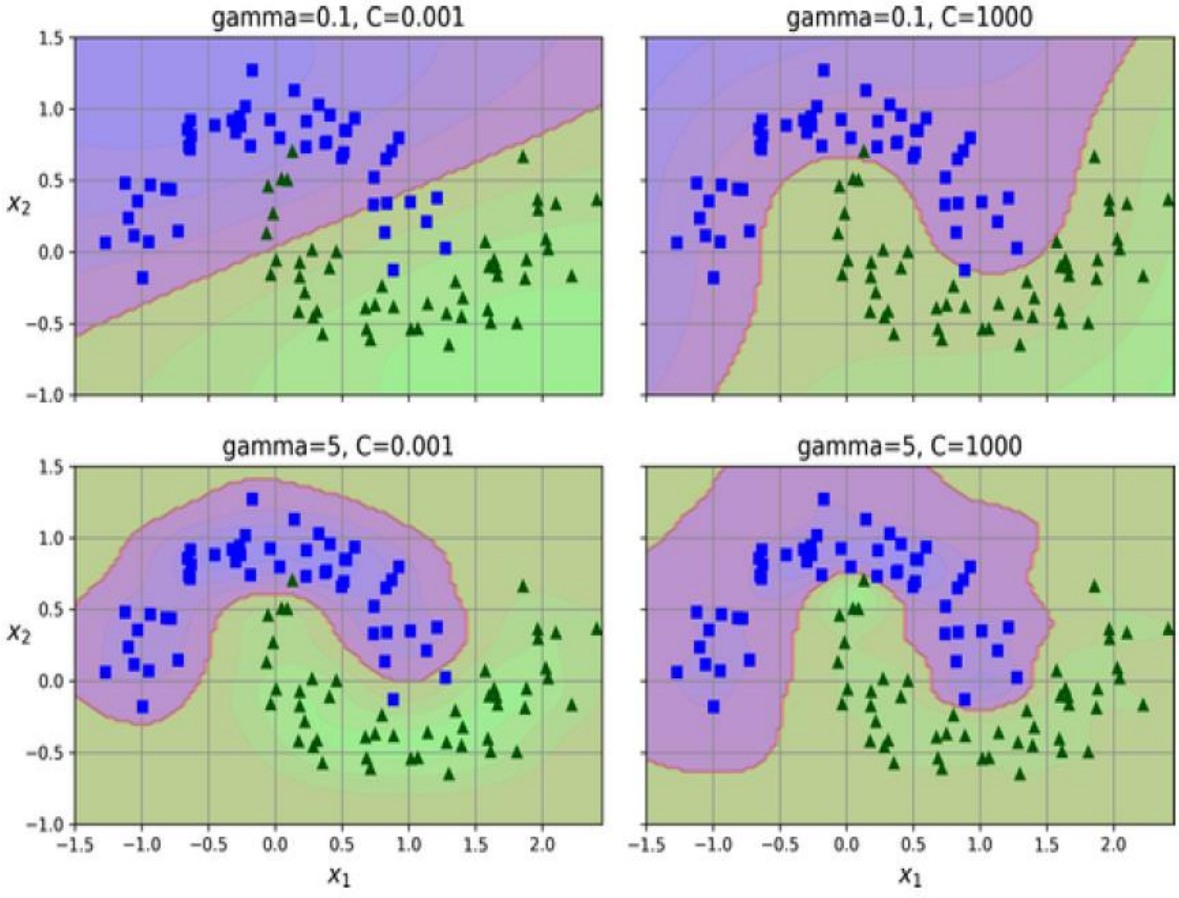


Figure 5-9. SVM classifiers using an RBF kernel

Computational Complexity

- **LinearSVC** is Based on **Liblinear** library
- Doesn't support kernel trick
- The tolerance hyperparameter **ϵ** (**tol** in scikit-learn), determines the precision of the model, higher the precision, longer the algorithm takes

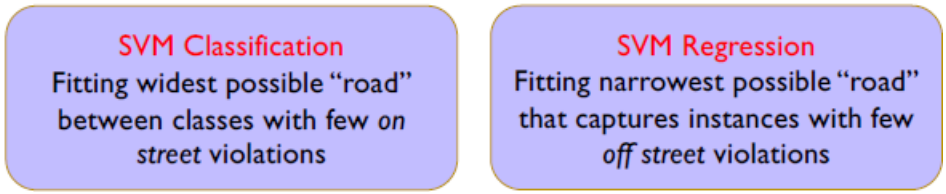
- **SVC** class is based on **libsvm** library, support the kernel trick
- When training instances gets large, it gets dreadfully slow
- Perfect for complex small or medium-sized training sets
- Scales well with no. of features, especially sparse features(has few nonzero features)

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No

SVM Regression

- SVM algorithm is versatile, can be used in both classification and regression
- Linear and non-linear regression



- The width of the street is controlled by a hyperparameter **ϵ**.
- the model is said to be **ϵ-insensitive**, adding more instances **within the margin** does not affect the model's predictions

```
from sklearn.svm import LinearSVR

X, y = [...]  # a linear dataset
svm_reg = make_pipeline(StandardScaler(),
                        LinearSVR(epsilon=0.5, random_state=42))
svm_reg.fit(X, y)
```

- to tackle nonlinear regression tasks, we use a kernelized SVM model
- example on quadratic training set
  - use SVR class with kernel ="poly"
  - soft margin via hyperparameter C

```
from sklearn.svm import SVR

X, y = [...]  # a quadratic dataset
svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1))
svm_poly_reg.fit(X, y)
```

- LinearSVC ←→ LinearSVR
  - No support vector attribute, no support for kernel
- SVC ←→ SVR
  - Support vectors, kernels, slow to train
- SVMs can also be used for outlier detection

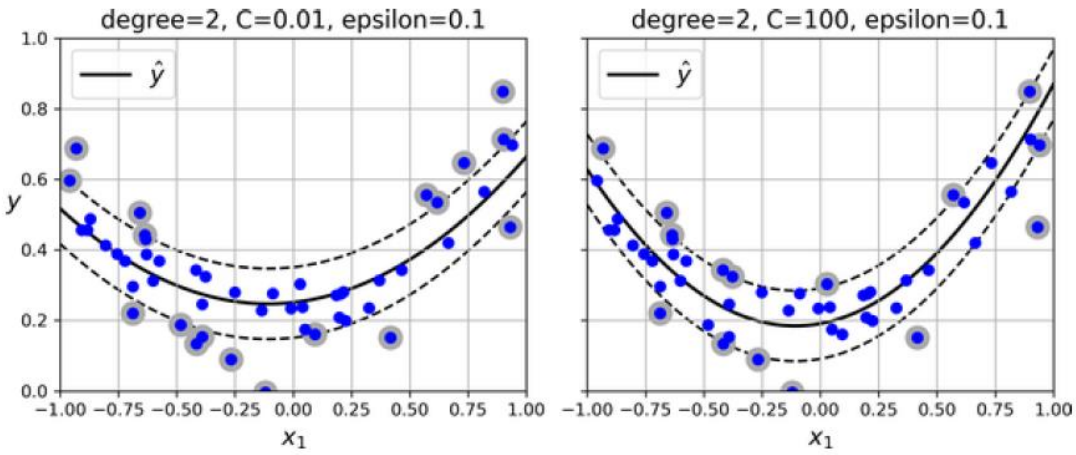


Figure 5-11. SVM regression using a second-degree polynomial kernel

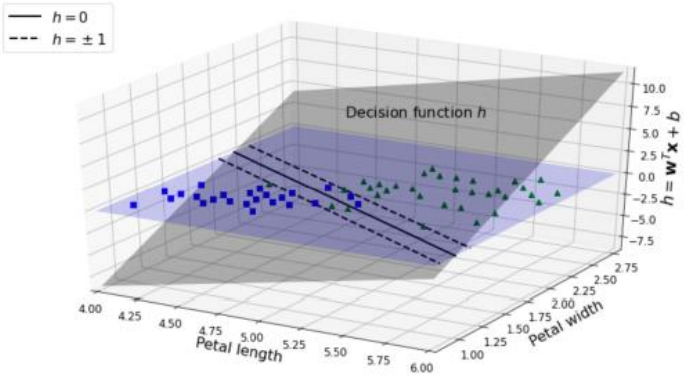
- **C** Hyperparameter works like a regularizer, if your SVM is overfitting try to regularize it by reducing **C**

Under the Hood

Decision Functions and predictions

Hyperplane $h(\boldsymbol{x}) = w^T \boldsymbol{x} + b$

Prediction $\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b \geq 0 \end{cases}$



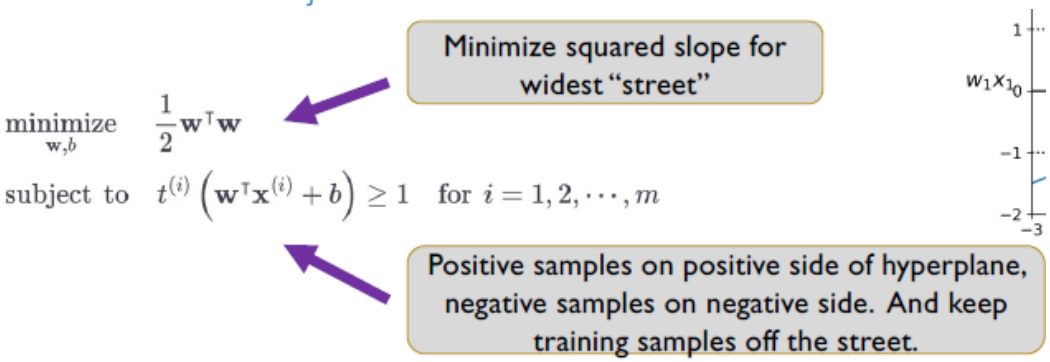
- **Decision Function** is the **hyperplane**
- **Decision boundary** is where **hyperplane** intersects **feature space**, where the **decision function** is equal to **0**
- The **dashed lines** represent the points where the decision function is equal to 1 or -1
- Training a **linear SVM** classifier means finding the values of **w** and **b** that make the margin as wide as possible while avoiding margin violations

Training Objective

- **Slope** of the Hyperplane is **||w||,** The smaller the weight vector **w**, the larger the margin
- we want to minimize **||w||** to get a large margin.
- If we want to avoid any **margin violations**, then we need the **decision function** to **> 1** for all positive instances and lower than **< -1** for all negative instances (hard margin)

- we define t = –1 for negative instances (when y = 0) and t = 1 for positive instances (when y = 1), then we can write this constraint as t (w x + b) ≥ 1 for all instances.

Hard margin Linear Classifier objective



- Optimization algorithms work much better on differential functions, that's why We are minimizing ½ $\mathbf{w}^t\,\mathbf{w}$, which is equal to ½$\|\,\mathbf{w}\,\|^2$, rather than minimizing $\|\,\mathbf{w}\,\|$. Indeed, ½$\|\,\mathbf{w}\,\|^2$ has a nice simple derivative (it is just w), while $\|\,w\,\|$ is not differentiable at w = 0.

Soft margin linear SVM classifier objective

- Add **slack** variable $\zeta^{(i)} \geq 0$ for each instance, measures how much the $\mathbf{i}^{\text{th}}$ instance is allowed to violate the margin
- There is two conflicting objectives
  - Make the **slack** variable as low as possible to reduce margin violations
  - Make ½ $\mathbf{w}^t\,\mathbf{w}$ as small as possible to increase the margin
- **C** hyperparameter acts as trade-off between these two objectives

Equation 5-2. Soft margin linear SVM classifier objective

$$\underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)}+b\right) \geq 1-\zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Quadratic programming

- SVM objective Functions are convex quadratic optimization problems
- Off-the-shelf QP solvers can be used to solve QP problems

The Dual Problem

- Primal problem ←→ Dual problem
- The dual problem typically gives the lower bound solution to the primal problem
- Both can have the same solution if the objective function is convex and the constrains are continuously differentiable and convex
- The dual problem is faster to solve than the primal one, when the number of training instances is smaller than the number of features
  - LinearSVC class solves dual problem by default
- The dual problem makes the kernel trick possible

Equation 5-3. Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \ \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)\intercal}\mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m}\alpha^{(i)} \text{subject to } \alpha^{(i)} \geq 0 \text{ for all } i = 1, 2, \ldots, m \text{ and } \sum_{i=1}^{m}\alpha$$

- Once you find the vector **α** that minimizes this equation (using a QP solver), use Equation 5-4 to compute the w and b that minimize the primal problem. In this equation, n represents the number of support vectors.

Equation 5-4. From the dual solution to the primal solution

$$\hat{\mathbf{w}} = \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \hat{\mathbf{w}}^{\intercal}\mathbf{x}^{(i)}\right)$$

Kernelized SVMs

- Second-degree polynomial function ϕ

Equation 5-5. Second-degree polynomial mapping

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\,x_1x_2 \\ {x_2}^2 \end{pmatrix}$$

- If we apply the ϕ and compute the dot product to vectors **a** and **b**

Equation 5-6. Kernel trick for a second-degree polynomial mapping

$$\phi(\mathbf{a})^{\intercal}\phi(\mathbf{b}) = \begin{pmatrix} {a_1}^2 \\ \sqrt{2}\,a_1a_2 \\ {a_2}^2 \end{pmatrix}^{\intercal}\begin{pmatrix} {b_1}^2 \\ \sqrt{2}\,b_1b_2 \\ {b_2}^2 \end{pmatrix} = {a_1}^2{b_1}^2 + 2a_1b_1a_2b_2 + {a_2}^2{b_2}^2$$

$$= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^{\intercal}\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = (\mathbf{a}^{\intercal}\mathbf{b})^2$$

- $\phi(\mathbf{a})\,\phi(\mathbf{b}) = (\mathbf{a}^t \cdot \mathbf{b})^2$
- instead of applying the **ϕ** transformation to all the training instances in the dual problem $\varphi(\mathbf{x}^{(i)})^t\,\varphi(\mathbf{x}^{(j)})$, you can replace this by the dot product of transformed vectors $(\mathbf{x}^{(i)T}\,\mathbf{x}^{(j)})^2$
- This trick makes the whole process much more computationally efficient
- **A kernel** is a function capable of computing the **dot product** of ϕ(**a**) ϕ(**b**) based only on the original vector **a** and **b**, without having to compute(or even to know about) the **transformation ϕ**

Equation 5-7. Common kernels

$$\begin{aligned} \text{Linear:} \quad & K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^{\intercal}\mathbf{b} \\ \text{Polynomial:} \quad & K(\mathbf{a}, \mathbf{b}) = (\gamma\mathbf{a}^{\intercal}\mathbf{b} + r)^d \\ \text{Gaussian RBF:} \quad & K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma\|\mathbf{a} - \mathbf{b}\|^2\right) \\ \text{Sigmoid:} \quad & K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma\mathbf{a}^{\intercal}\mathbf{b} + r\right) \end{aligned}$$

- Mercer's Theorem, If a function K(a, b) respects a few mathematical conditions called mercer's conditions (k must be continuous and symmetric in its arguments so that K(a, b) = K(b, a), etc.)
- Then $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$
- Applying the kernel trick to Equation 5-7, transform the ϕ into dot product to make computationally possible

Equation 5-8. Making predictions with a kernelized SVM

$$h_{\hat{\mathbf{w}},\hat{b}}\left(\phi\left(\mathbf{x}^{(n)}\right)\right) = \hat{\mathbf{w}}^T\phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\phi\left(\mathbf{x}^{(i)}\right)\right)^T\phi\left(\mathbf{x}^{(n)}\right) + \hat{b}$$

$$= \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\left(\phi\left(\mathbf{x}^{(i)}\right)^T\phi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b}$$

$$= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\hat{\alpha}^{(i)}t^{(i)}K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b}$$

- $\alpha \neq 0$ only for support vectors, computing the dot product of only the support vectors and the new input vector $\mathbf{x}^{(n)}$ not all the training instances

Equation 5-9. Using the kernel trick to compute the bias term

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \hat{\mathbf{w}}^T\phi\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \left(\sum_{j=1}^{m}\hat{\alpha}^{(j)}t^{(j)}\phi\left(\mathbf{x}^{(j)}\right)\right)^T\phi\left(\mathbf{x}^{(i)}\right)\right)$$

$$= \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)}>0}}^{m}\hat{\alpha}^{(j)}t^{(j)}K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)\right)$$

Online SVMs

- online learning means learning incrementally as new instances arrive
- for linear SVM classifiers, we can use Gradient Descent to implement an online SVM classifier to minimize the cost function
- Gradient Descent converges more slowly than the methods based on QP

Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}max\left(0, 1 - t^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right)\right)$$

- The first sum in the cost function will push the model to have a small weight vector **w,** leading to larger margin
- The second sum computes the total of all margin violations, an instance's margin violations is equal to 0 if it is located off street and on the correct side or else it is proportional to the distance to the correct side of the street
- Minimizing this term ensures that the model makes the margin violations as small and few as possible

Hinge Loss vs Squared Hinge loss

- Given an instance x of the positive class (i.e., with t = 1), the loss is 0 if the output s of the decision function (s = w x + b) is greater than or equal to 1. This happens when the instance is off the street and on the positive side.
- Given an instance of the negative class (i.e., with t = –1), the loss is 0 if s ≤ –1. This happens when the instance is off the street and on the negative side.
- The further away an instance is from the correct side of the margin, the higher the loss: it grows linearly for the hinge loss, and quadratically for the squared hinge loss.
- This makes the squared hinge loss more sensitive to outliers
- **LinearSVC** uses the **squared hinge loss**, while **SGDClassifier** uses the **hinge loss**
- Both classes let you choose the loss by setting the **loss** hyperparameter to "**hinge**"

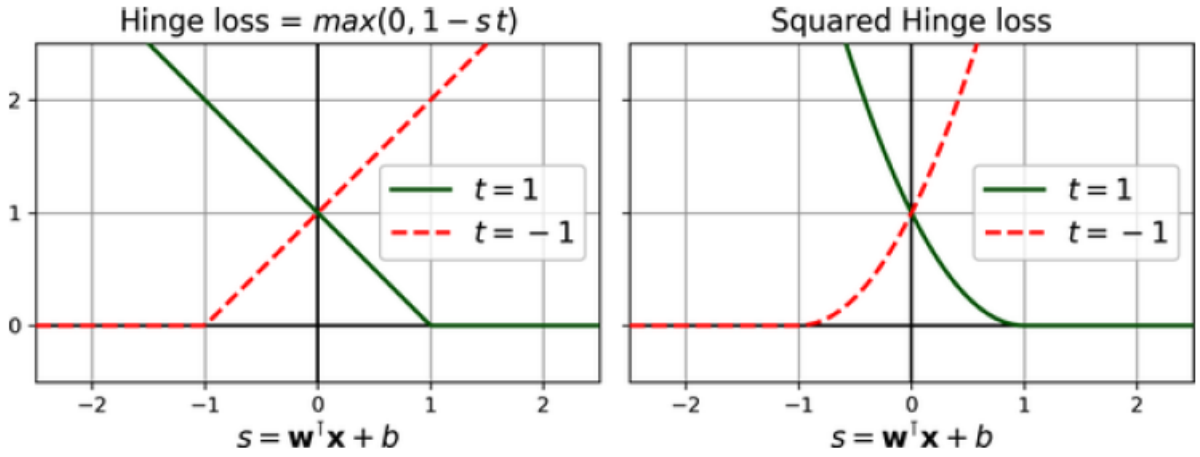


Figure 5-13. The hinge loss (left) and the squared hinge loss (right)

1.What is the fundamental idea behind Support vector machines?

Is to fit the widest possible “street” between the classes. In other words, the goal is to have the largest possible margin between the decision boundary that separates the two classes and the training instances.

2. What is a support vector?

A support vector is any instance located on the “street”, the decision boundary is determined by the support vectors, any “off the street” vector has no influence, you could remove them, add more instances or move them around as long as they stay off the street they won’t affect the decision boundary. Computing the predictions only involves the support vectors, not the whole training set

3. why is it important to scale the inputs when using SVMs?

If the training set is not scaled, the SVM will tend to neglect small features

4. Can an SVM classifier output a confidence score when it classifies an instance? What about a probability?

An SVM can output the distance between the test instance and the decision boundary, you can use as a confidence score, however this score cannot be converted into an estimation of the class probability.

If you set probability= True when creating SVM in scikit-learn, you can get the probability

5. should you use the primal or the dual form of the SVM problem to train a model on a training set with millions of instances and hundreds of features?

Kernelized SVMs can only use the dual form, so this question applies to linear SVMs, primal form is $O(m \times n)$ while dual form is $O(m^2 \times n)$ or $O(m^3 \times n)$.

In this case we should use the primal form, because the dual form will be too slow.