Supervised Learning – Part 5

ESM3081 Programming for Data Science

Seokho Kang

성균관대학교
SUNGKYUNKWAN UNIVERSITY(SKKU)

Learning algorithms covered in this course

- **Supervised Learning** (Classification/Regression)
  - K-Nearest Neighbors
  - Linear Models (Logistic/Linear Regression)
  - Decision Trees
  - Random Forests
  - **Support Vector Machines**
  - Neural Networks

** Many algorithms have a classification and a regression variant, and we will describe both.*

** We will review the most popular machine learning algorithms, explain how they learn from data and how they make predictions, and examine the strengths and weaknesses of each algorithm.*

Support Vector Machines

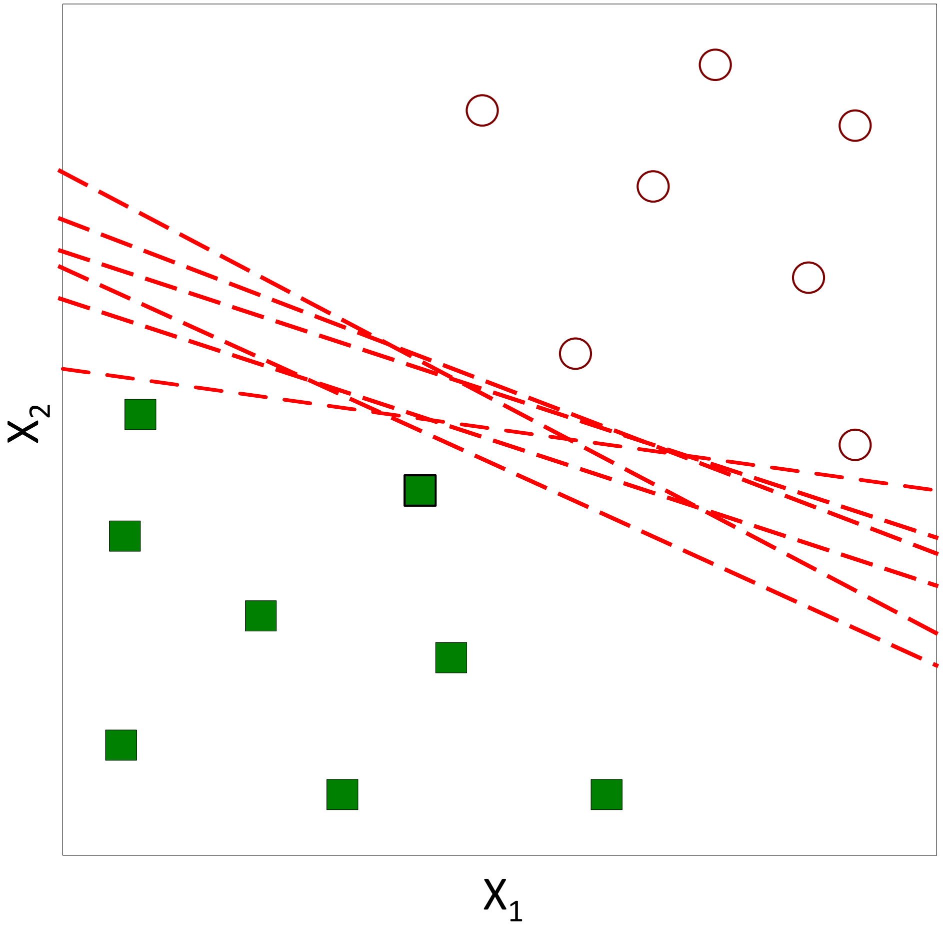
Support Vector Machines

- **Support Vector Machine for Classification**
  - Linear Support Vector Classification
  - Kernelized Support Vector Classification

- **Support Vector Machine for Regression**
  - Linear Support Vector Regression
  - Kernelized Support Vector Regression

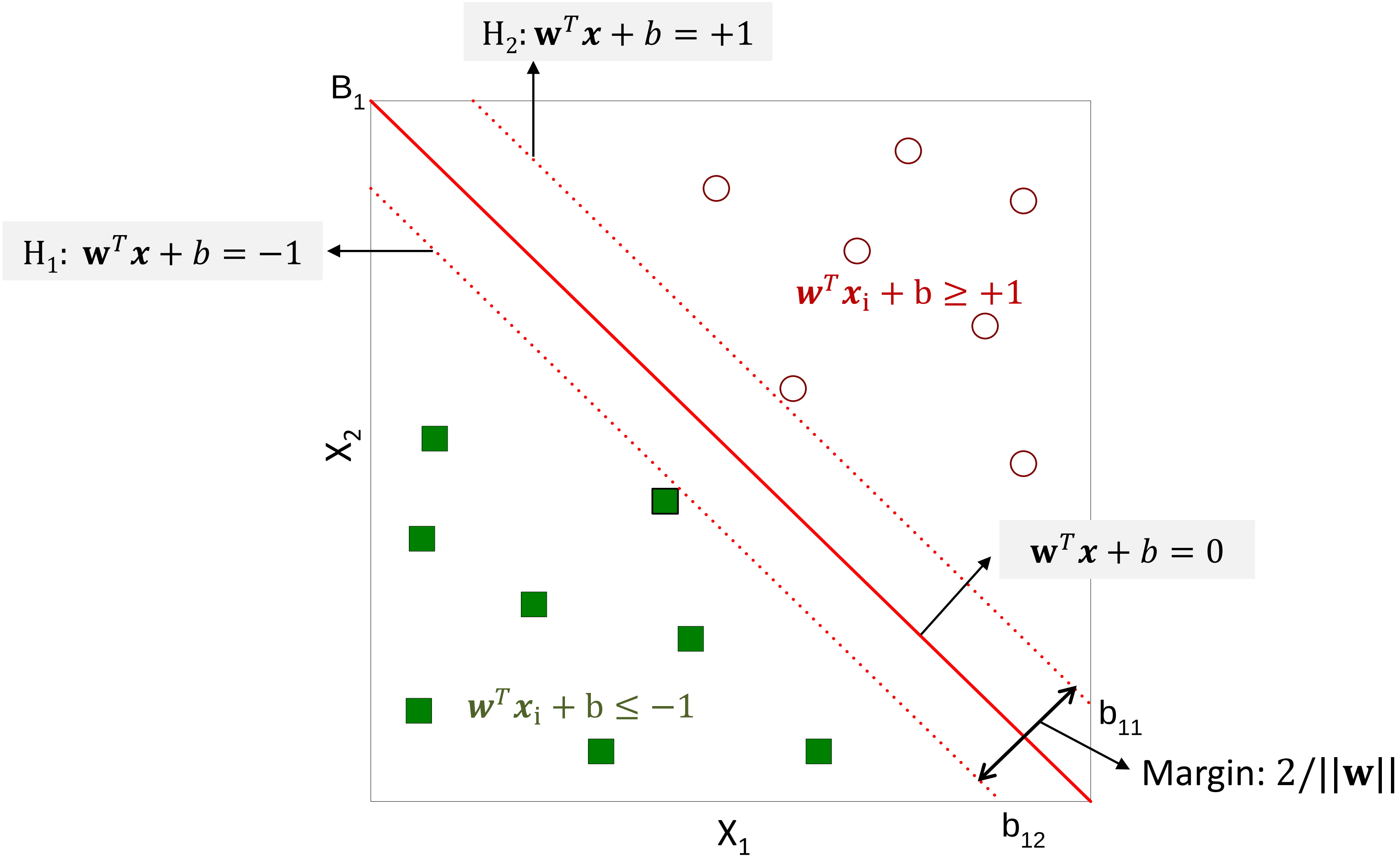
Support Vector Classification

- **Binary Classification – Find a hyperplane (linear decision boundary) that will separate the data**
  - Many possible hyperplanes ($\mathbf{w}^T\boldsymbol{x} + b = 0$) that separate the training data
  - Which one is better? How do you define better?



Support Vector Classification

- **Support Vector Classification – Find the hyperplane that maximizes the margin**



$H_2\text{: } \mathbf{w}^T\boldsymbol{x} + b = +1$

$H_1\text{: } \mathbf{w}^T\boldsymbol{x} + b = -1$

$\boldsymbol{w}^T\boldsymbol{x}_\mathrm{i} + \mathrm{b} \geq +1$

$\mathbf{w}^T\boldsymbol{x} + b = 0$

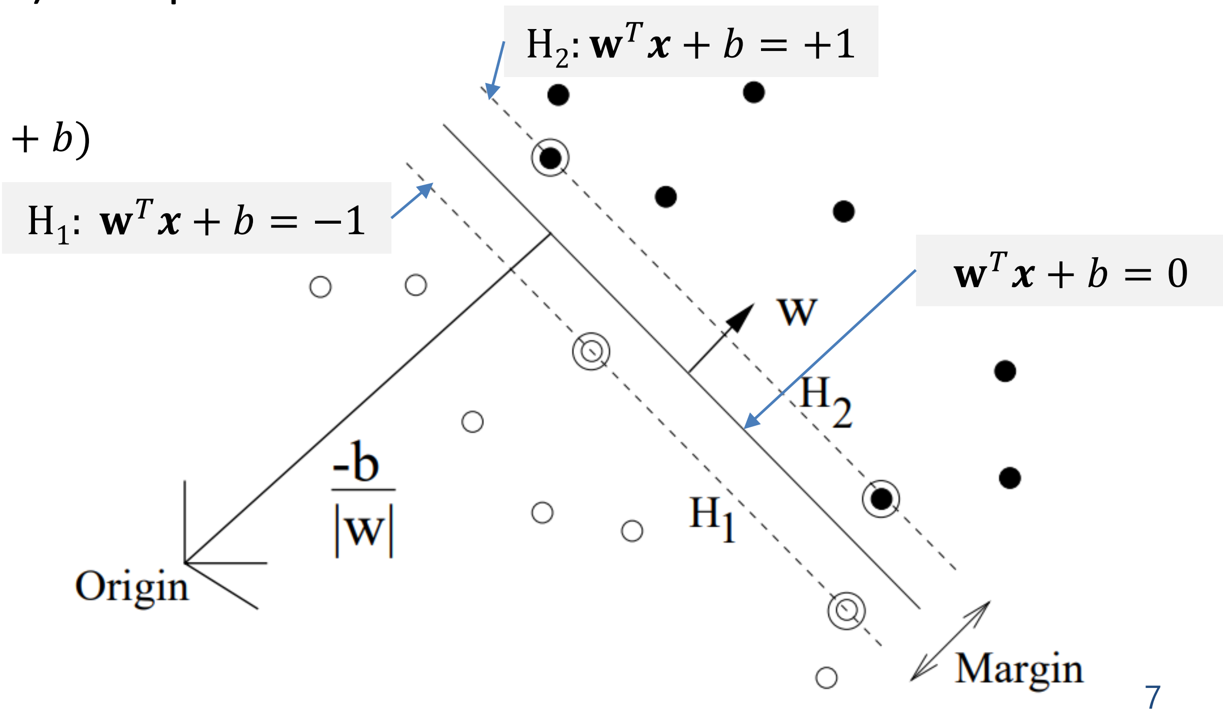
$\boldsymbol{w}^T\boldsymbol{x}_\mathrm{i} + \mathrm{b} \leq -1$

Margin: $2/||\mathbf{w}||$

Support Vector Classification

- Given a (training) dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of *d* features and $y_i \in \{-1, +1\}$ is the corresponding target label.

- SVM looks for the **maximum-margin hyperplane** $\mathbf{w}^T \boldsymbol{x} + b = 0$ between positive ($y_i = +1$) and negative ($y_i = -1$) data points
  - Margin: $2/||\mathbf{w}||$
  - Prediction $\hat{y} = f(\boldsymbol{x}) = \mathrm{sign}(\mathbf{w}^T \boldsymbol{x} + b)$



H$_2$: $\mathbf{w}^T \boldsymbol{x} + b = +1$
H$_1$: $\mathbf{w}^T \boldsymbol{x} + b = -1$
$\mathbf{w}^T \boldsymbol{x} + b = 0$
w
H$_2$
H$_1$
$\frac{-b}{|w|}$
Origin
Margin

Support Vector Classification

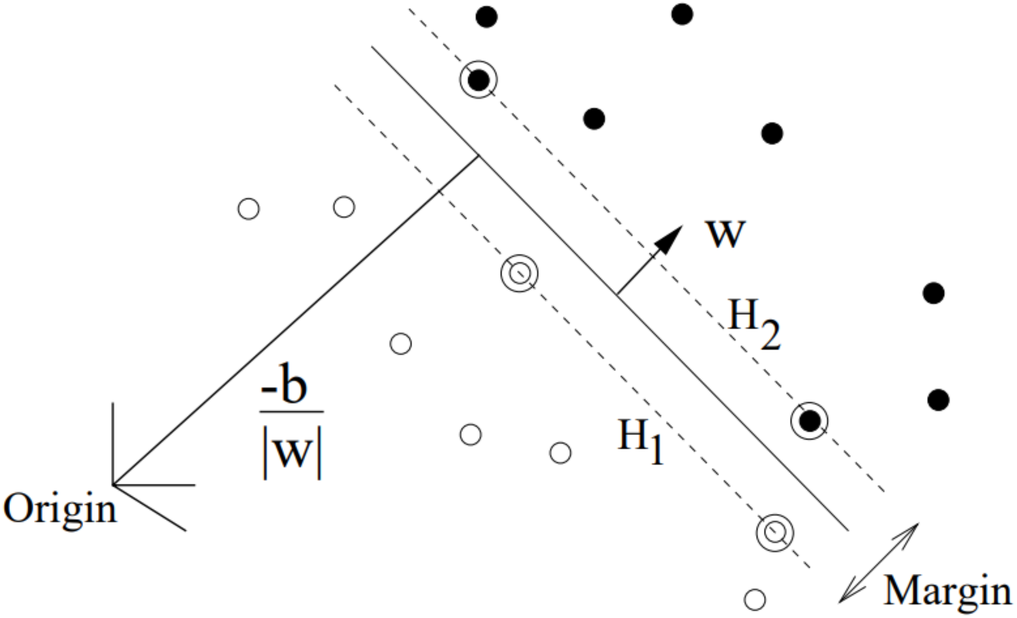
- **Hard-margin formulation**

  : Do not allow any errors, no training points fall between H_1 and H_2

$$\min J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$ ← *maximize the margin*

subject to $y_i(\mathbf{w}^T \boldsymbol{x}_i + b) \geq 1, \forall i$ ← *all training data points are outside the margin*

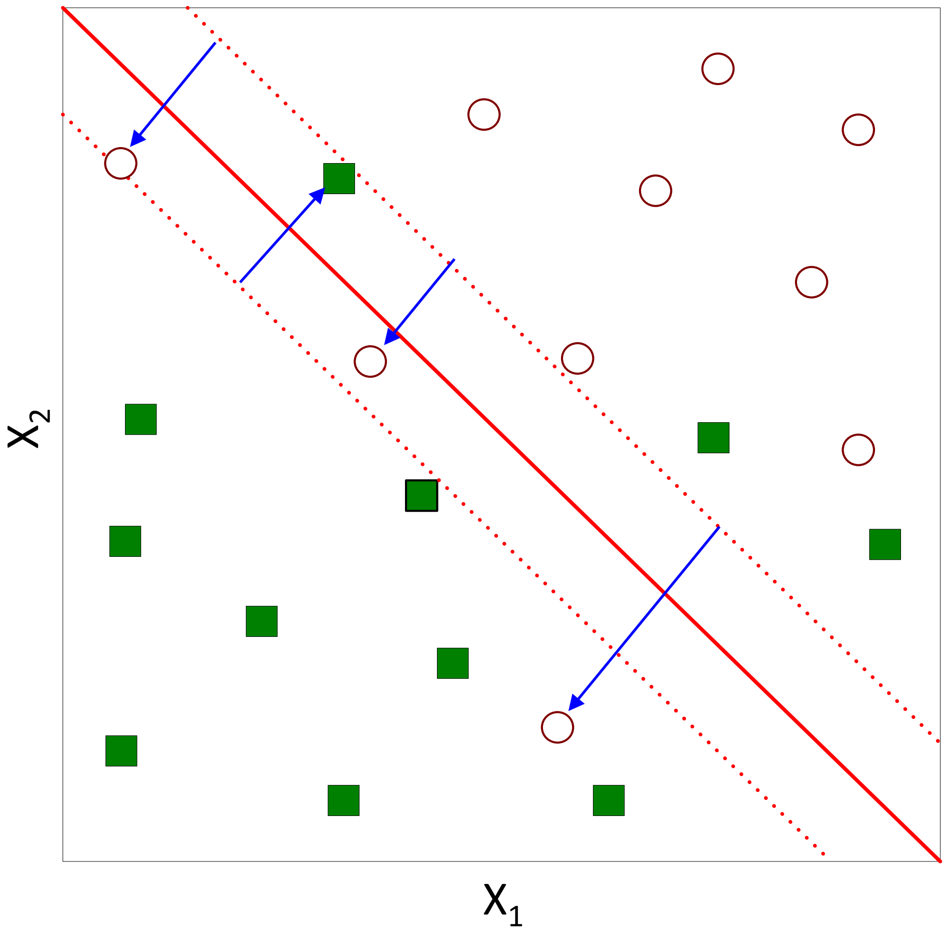
What are parameters?
What are hyperparameters?



Support Vector Classification

- **What if the data are linearly inseparable?**

  : Introduce slack variables ξ_i



Support Vector Classification

- **Soft-margin formulation**

  : Allow some errors by introducing slack variables $\xi_i \geq 0$

maximize the margin

$L(y, \mathbf{w}^T \boldsymbol{x} + b) = \max(0, 1 - y(\mathbf{w}^T \boldsymbol{x} + b))$

$$\min J(\mathbf{w}, b, \boldsymbol{\xi}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_i \xi_i$$

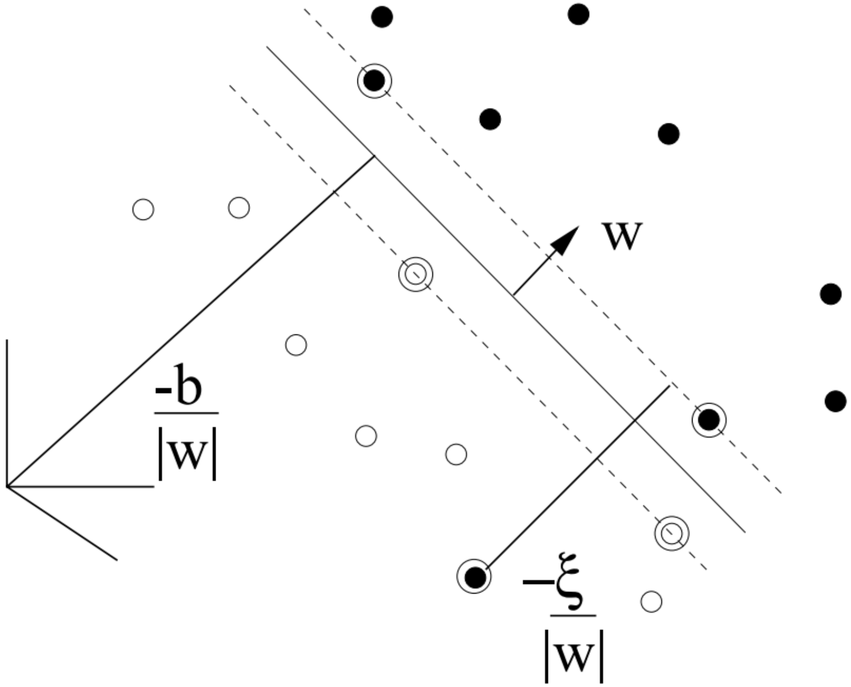
minimize empirical risk (hinge loss)
trade-off hyperparameter C

$$\text{subject to } \begin{array}{c} y_i(\mathbf{w}^\mathrm{T}\boldsymbol{x}_i + b) \geq 1 - \xi_i, \\ \xi_i \geq 0, \forall i \end{array}$$

most training data points are outside the margin,
but some are not

Constrained "convex" optimization problem
→ Solve it using Lagrange multiplier method

What are parameters?
What are hyperparameters?



Support Vector Classification

- **Soft-margin formulation**

  : Dual Problem (**Quadratic Programming**) → Use a QP Solver!

O(n^2) *space complexity*

usually O(n^3) *time complexity*

what if n is very large?

$$\max L(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \boxed{\boldsymbol{x}_i^T \boldsymbol{x}_j}$$

$$\text{subject to} \quad \sum_i \alpha_i y_i = 0$$

$$0 \le \alpha_i \le C, \forall i$$

n parameters $\alpha_1, \ldots, \alpha_n$

Convex optimization

→ Global optimum is guaranteed

Support Vector Classification

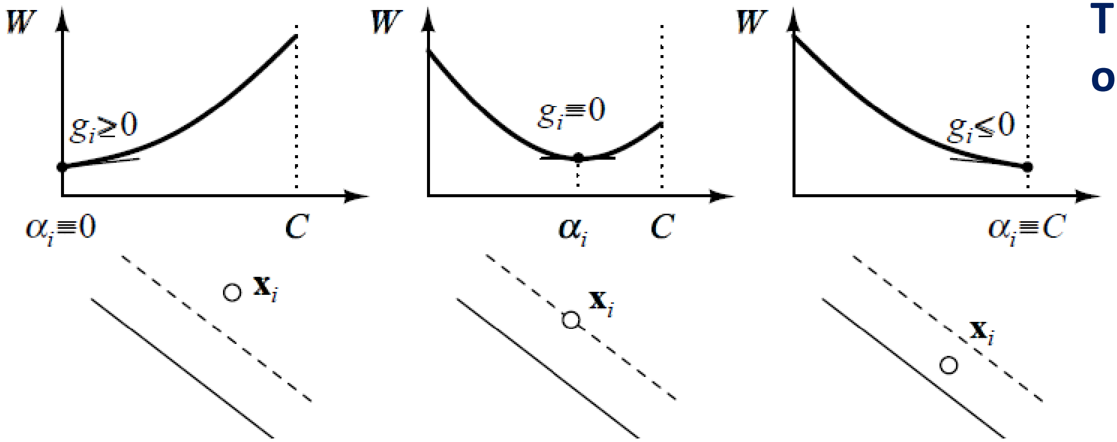
- **Soft-margin formulation**

  : After obtaining the maximum-margin hyperplane $\mathbf{w}^{*T}\boldsymbol{x} + b^*$, *how?*

$$\mathbf{w}^* = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \qquad b^* = \frac{1}{y_{sv}} - \mathbf{w}^{*T}\boldsymbol{x}_{sv} = \frac{1}{y_{sv}} - \sum_{i=1}^{n} \alpha_i y_i \boxed{\boldsymbol{x}_i^T \boldsymbol{x}_{sv}}$$

, where $(\boldsymbol{x}_{sv}, y_{sv}) \in \{(\boldsymbol{x}_i, y_i) | 0 < \alpha_i < C\}$

- **The trained model**
  - $f(\boldsymbol{x}) = \text{sign}\left(\mathbf{w}^{*T}\boldsymbol{x} + b^*\right) = \text{sign}\left(\sum_{(\boldsymbol{x}_i, y_i) \in D} \alpha_i y_i \boxed{\boldsymbol{x}_i^T \boldsymbol{x}} + b^*\right)$ *n parameters $\alpha_1, .., \alpha_n$*
  - Let $D_{SV} = \{(\boldsymbol{x}_i, y_i) \in D | \alpha_i > 0\}$, then $f(\boldsymbol{x}) = \text{sign}\left(\sum_{(\boldsymbol{x}_i, y_i) \in D_{SV}} \alpha_i y_i \boxed{\boldsymbol{x}_i^T \boldsymbol{x}} + b^*\right)$ (sparse solution)



non-support vectors $\{(\boldsymbol{x}_i, y_i) | \alpha_i = 0\}$

margin support vectors $\{(\boldsymbol{x}_i, y_i) | 0 < \alpha_i < C\}$

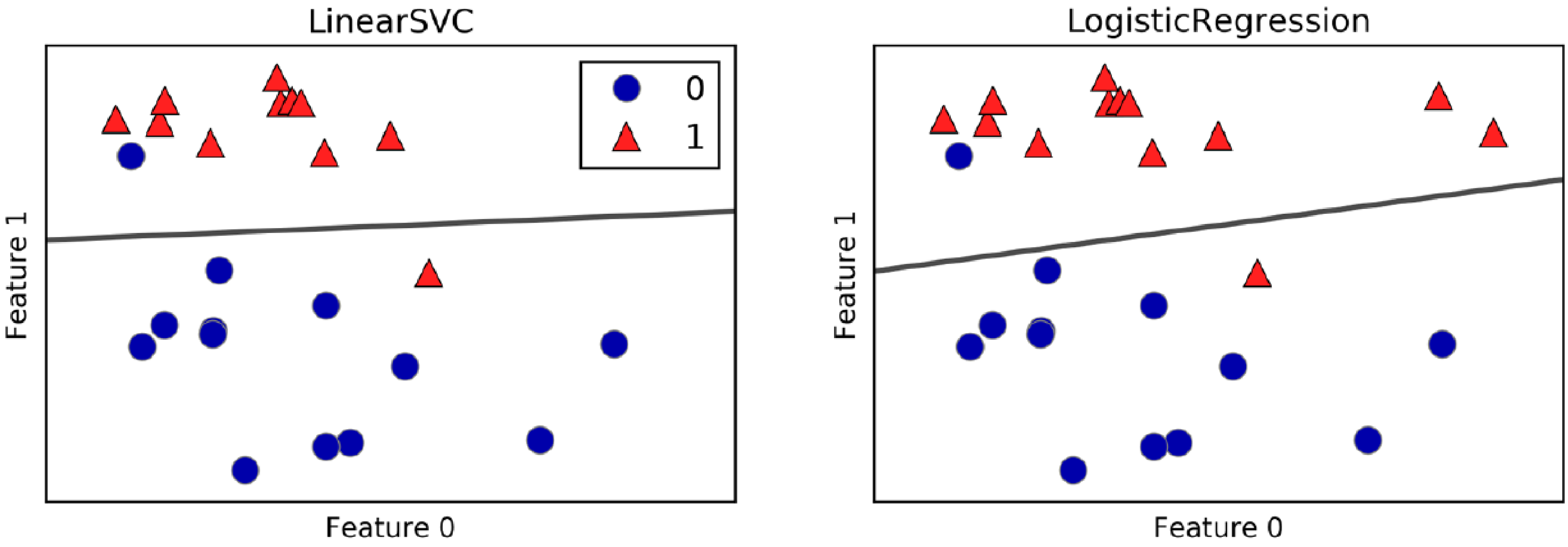
error support vectors $\{(\boldsymbol{x}_i, y_i) | \alpha_i = C\}$

The trained model depends only on support vectors

what are support vectors?

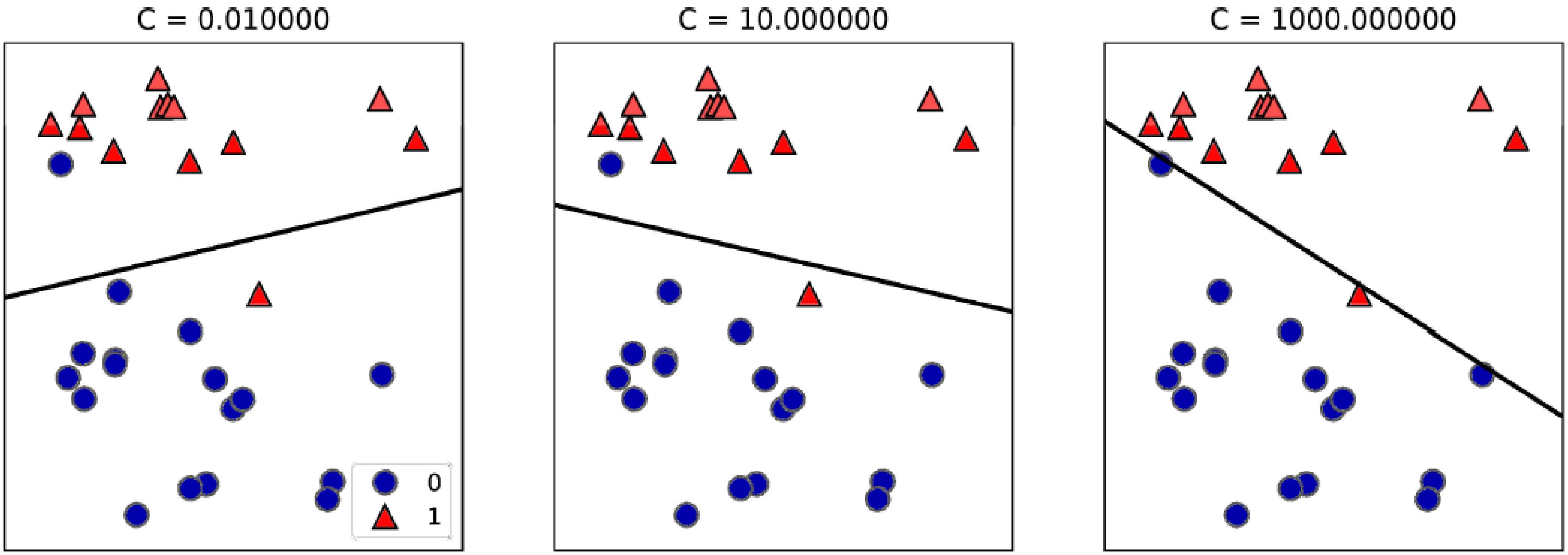
Support Vector Classification

- *Decision boundaries of a linear SVM and logistic regression on the forge dataset with the default hyperparameters*



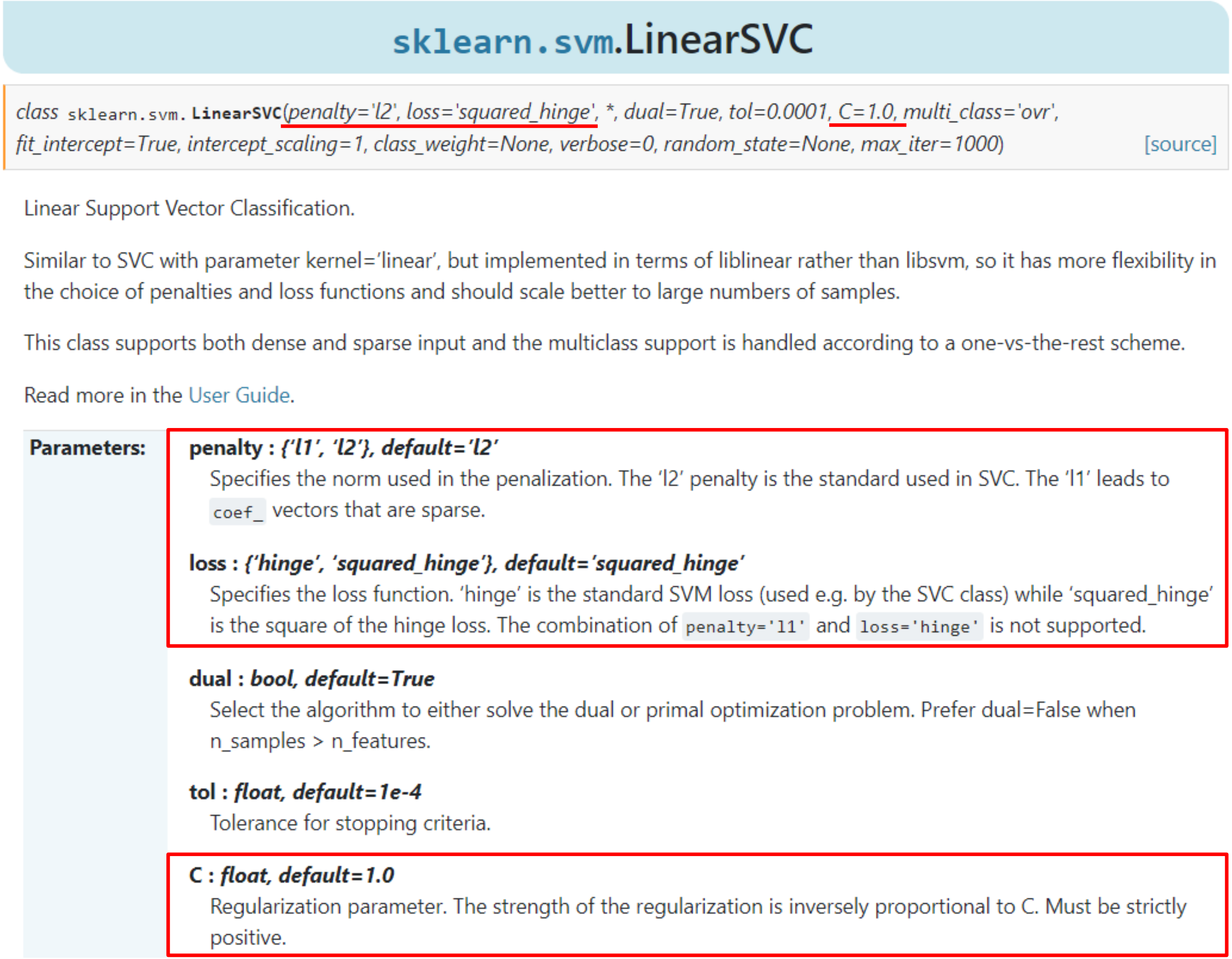
Support Vector Classification

- The trade-off hyperparameter (the strength of the regularization) C
  - lower values of C correspond to more regularization
    - The model puts more emphasis on finding a coefficient vector $\mathbf{w}$ that is close to zero
      → underfitting
  - Higher values of C correspond to less regularization
    - The model tries to fit the training set as best as possible
      → overfitting



scikit-learn Practice: *LinearSVC*

https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html



sklearn.svm.LinearSVC

class sklearn.svm.**LinearSVC**(*penalty='l2', loss='squared_hinge', *, dual=True, tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True, intercept_scaling=1, class_weight=None, verbose=0, random_state=None, max_iter=1000*) [source]

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

Read more in the User Guide.

Parameters:

penalty : *{'l1', 'l2'}, default='l2'*

Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

loss : *{'hinge', 'squared_hinge'}, default='squared_hinge'*

Specifies the loss function. 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss. The combination of `penalty='l1'` and `loss='hinge'` is not supported.

dual : *bool, default=True*

Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

tol : *float, default=1e-4*

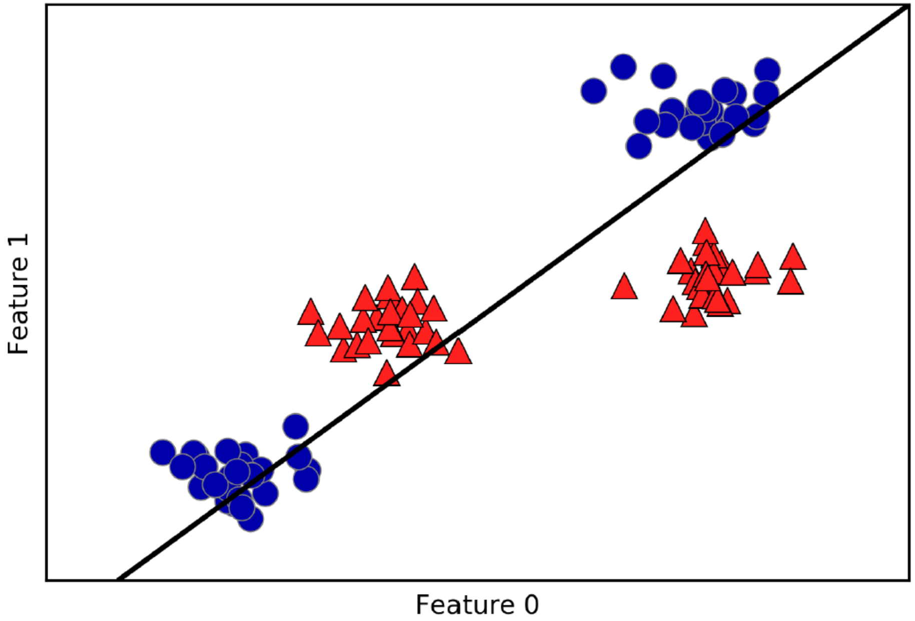
Tolerance for stopping criteria.

C : *float, default=1.0*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

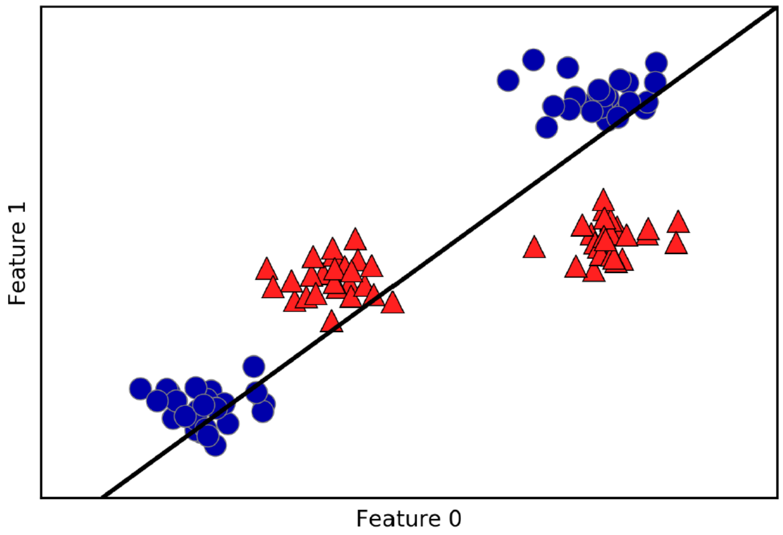
Kernelized Support Vector Classification

- Linear support vector classification can be quite limiting in low-dimensional spaces, as lines and hyperplanes have limited flexibility
- Kernelized support vector machines are an extension that allows for more complex models that are not defined simply by hyperplanes in the input space.
  - ***Example:** Given a two-class classification dataset in which classes are not linearly separable, the decision boundary found by a linear SVM*

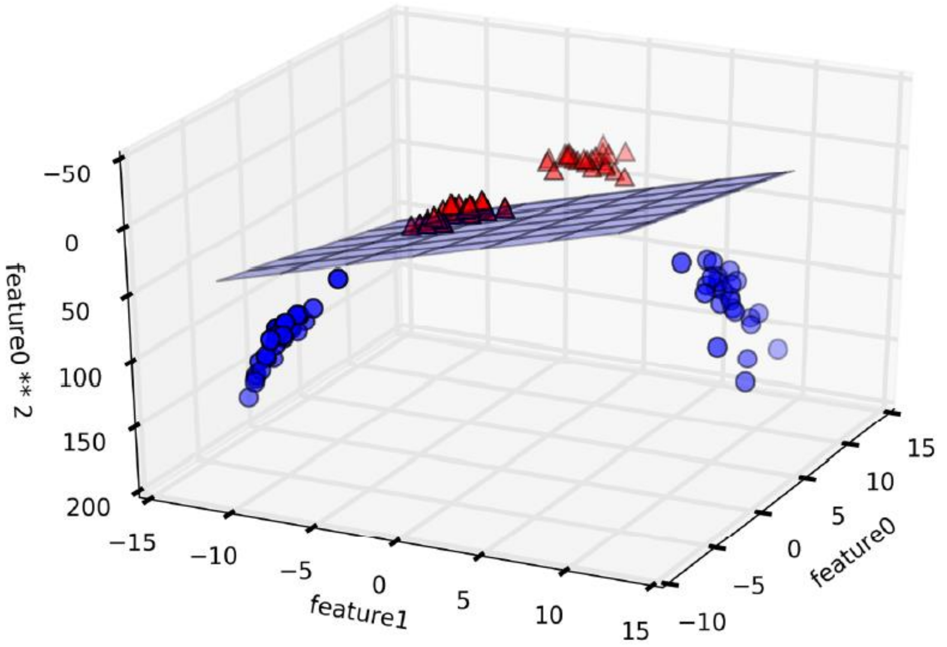


Kernelized Support Vector Classification

- One way to make a linear model more flexible is by adding more features—for example, by adding interactions or polynomials of the input features.
- **Example:** expanding the set of input features by adding *feature0**2*
  - It is now possible to separate the two classes using a linear model



2D: (*feature0, feature1*)



3D: (*feature0, feature1, feature0***2)

Kernelized Support Vector Classification

- Adding nonlinear features to the representation of our data can make linear models much more powerful.
- However, often we don’t know which features to add, and adding many features might make computation very expensive.
- Luckily, there is ***a mathematical trick*** that allows us to learn a classifier in a higher-dimensional space without actually computing the new, possibly very large representation.

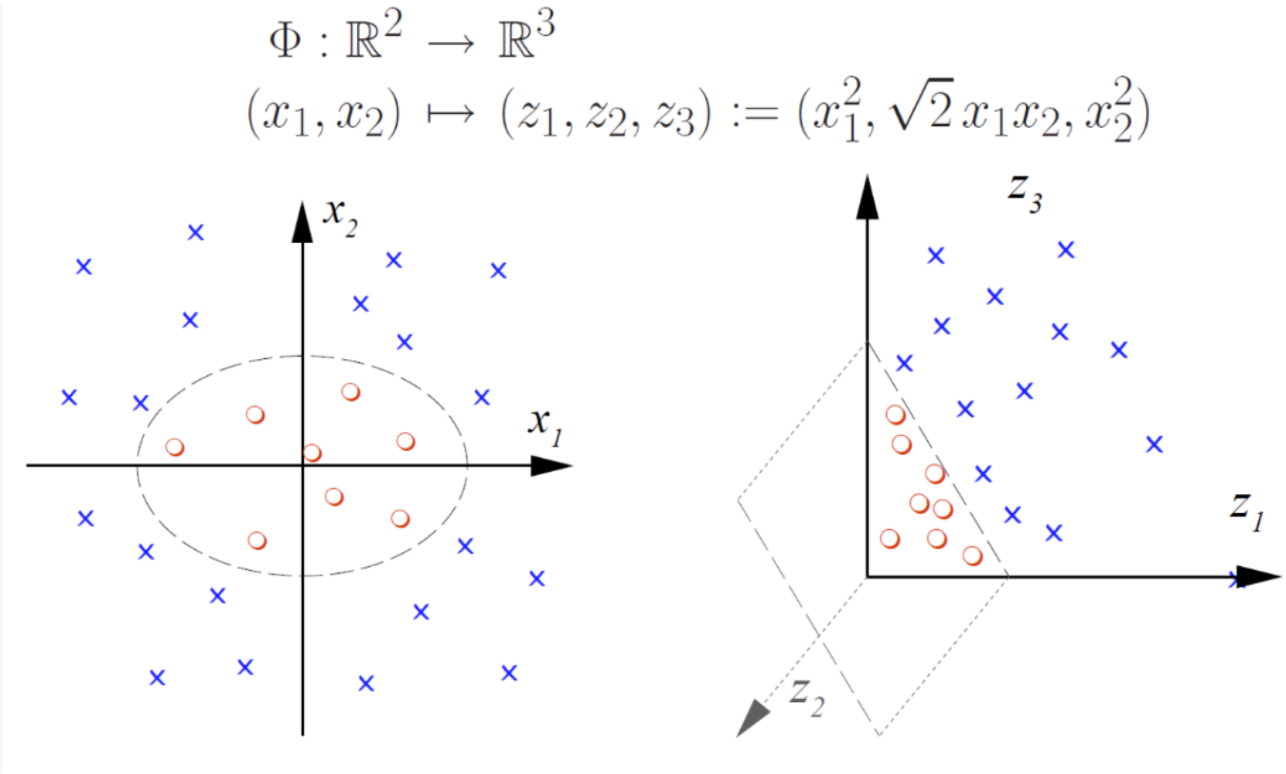
Kernelized Support Vector Classification

- **SVM for Non-linear Classification: Kernel Trick**

  : Use a function φ that maps the data into a higher dimensional space.

  - Replace $\boldsymbol{x}_i$ by $\varphi(\boldsymbol{x}_i)$

  - **Example**: $\varphi(x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$



$$\Phi : \mathbb{R}^2 \longrightarrow \mathbb{R}^3$$
$$(x_1, x_2) \longmapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}\,x_1x_2, x_2^2)$$

Kernelized Support Vector Classification

- **SVM for Non-linear Classification: Kernel Trick**

  : If there is a "kernel function" k that defines inner products in the transformed space, such that $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \varphi(\boldsymbol{x}_i)^T \varphi(\boldsymbol{x}_j)$, then **we don't have to know φ at all, but use k instead.**

  - Replace $\boldsymbol{x}_i^T \boldsymbol{x}_j$ by $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
  - Not all functions can be kernels (Mercer's theorem)

- **Examples of Kernel Functions**

  - **Linear Kernel** $k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$
  - **Polynomial Kernel** $k(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^T \boldsymbol{x}')^p$
  - **Tanh Kernel** $k(\boldsymbol{x}, \boldsymbol{x}') = \tanh(a + b\boldsymbol{x}^T \boldsymbol{x}')$
  - $\boxed{\textbf{RBF Kernel } k(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|^2)}$ ← most popular, default setting in scikit-learn

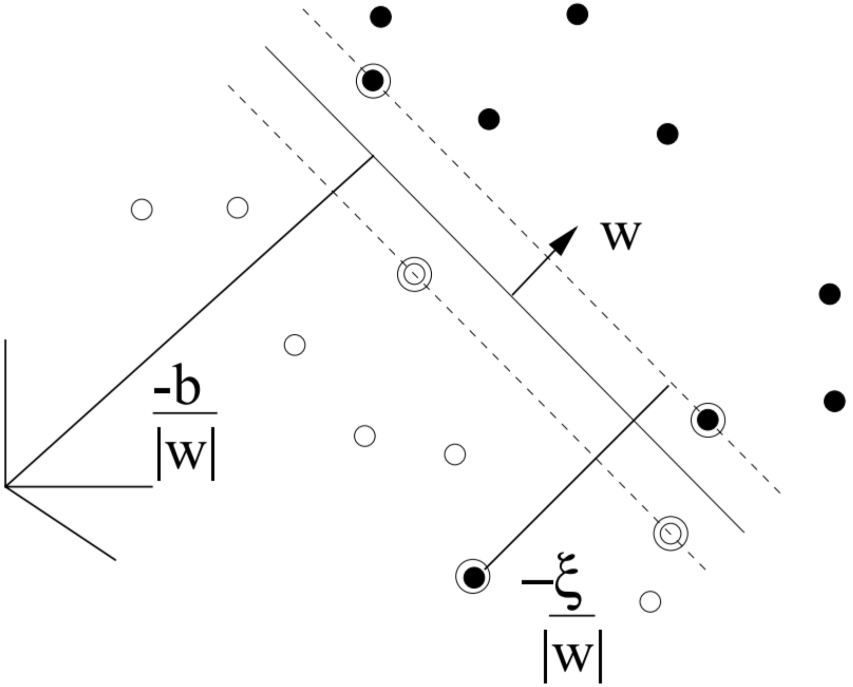
Kernelized Support Vector Classification

- **Soft-margin formulation**

  : Primal Problem with the function φ (feature map)

$$\min J(\mathbf{w}, b, \boldsymbol{\xi}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \mathrm{C}\sum_i \xi_i$$

$$\text{subject to} \quad \begin{array}{c} y_i(\mathbf{w}^T \boxed{\varphi(\boldsymbol{x}_i)} + b) \geq 1 - \xi_i, \\ \xi_i \geq 0, \forall i \end{array}$$



Kernelized Support Vector Classification

- **Soft-margin formulation**

  : Dual Problem (**Quadratic Programming**) → Use a QP Solver!

O(n^2) *space complexity*
usually O(n^3) *time complexity*
what if n is very large?

$$\max L(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \boxed{k(\boldsymbol{x}_i, \boldsymbol{x}_j)}$$

$$\text{subject to} \quad \sum_i \alpha_i y_i = 0$$

$$0 \le \alpha_i \le C, \forall i$$

n parameters $\alpha_1, \ldots, \alpha_n$

Convex optimization
→ Global optimum is guaranteed

Kernelized Support Vector Classification

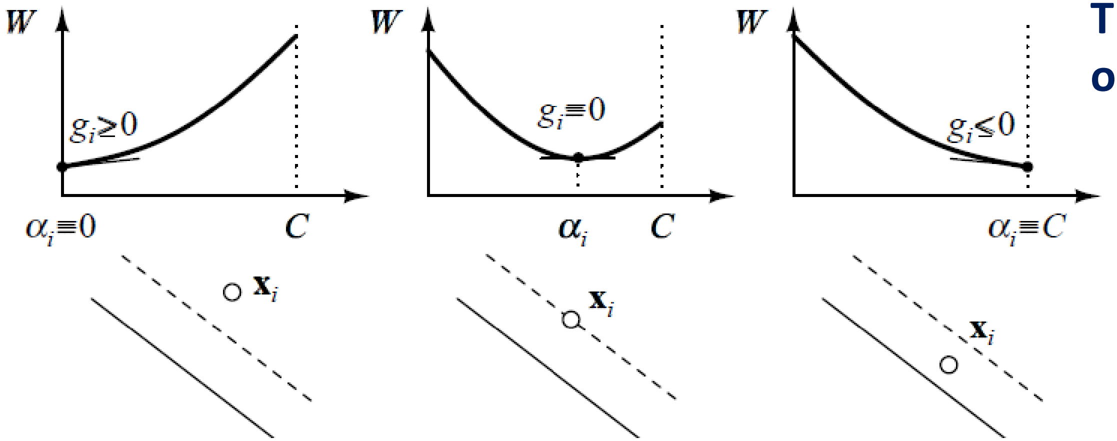
- **Soft-margin formulation**

  : After obtaining the maximum-margin hyperplane $\mathbf{w}^{*T}\varphi(\boldsymbol{x}) + b^*$,

$$\mathbf{w}^* = \sum_{i=1}^{n} \alpha_i y_i \varphi(\boldsymbol{x}_i) \qquad b^* = \frac{1}{y_{sv}} - \mathbf{w}^{*T}\boldsymbol{x}_{sv} = \frac{1}{y_{sv}} - \sum_{i=1}^{n} \alpha_i y_i \boxed{k(\boldsymbol{x}_i, \boldsymbol{x}_{sv})}$$

, where $(\boldsymbol{x}_{sv}, y_{sv}) \in \{(\boldsymbol{x}_i, y_i) | 0 < \alpha_i < C\}$

- **The trained model**
  - $f(\boldsymbol{x}) = \text{sign}\left(\mathbf{w}^{*T}\varphi(\boldsymbol{x}) + b^*\right) = \text{sign}\left(\sum_{(\boldsymbol{x}_i, y_i)\in D} \alpha_i y_i \boxed{k(\boldsymbol{x}_i, \boldsymbol{x})} + b^*\right)$
  - Let $D_{SV} = \{(\boldsymbol{x}_i, y_i) \in D | \alpha_i > 0\}$, then $f(\boldsymbol{x}) = \text{sign}\left(\sum_{(\boldsymbol{x}_i, y_i)\in D_{SV}} \alpha_i y_i \boxed{k(\boldsymbol{x}_i, \boldsymbol{x})} + b^*\right)$ (sparse solution)



non-support vectors $\{(\boldsymbol{x}_i, y_i) | \alpha_i = 0\}$

(unbounded) support vectors $\{(\boldsymbol{x}_i, y_i) | 0 < \alpha_i < C\}$

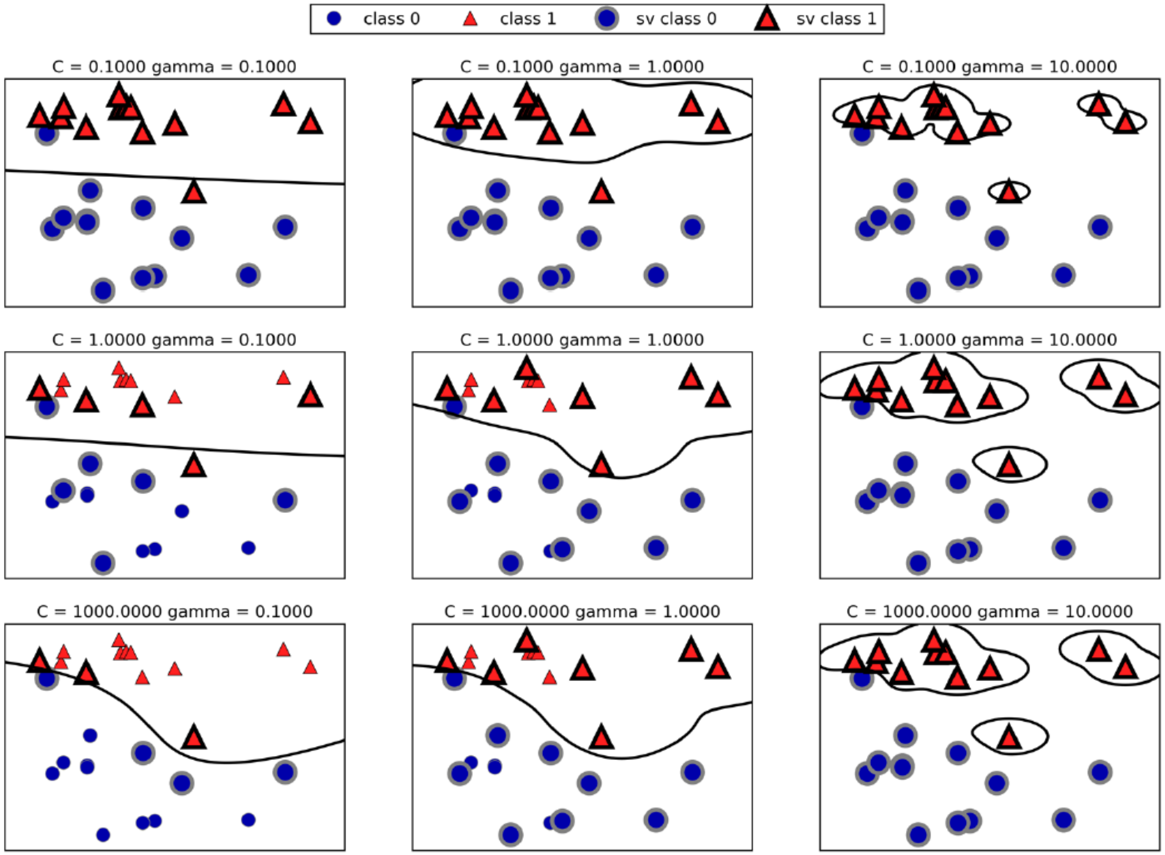
(bounded) support vectors $\{(\boldsymbol{x}_i, y_i) | \alpha_i = C\}$

The trained model depends only on support vectors

what are support vectors?

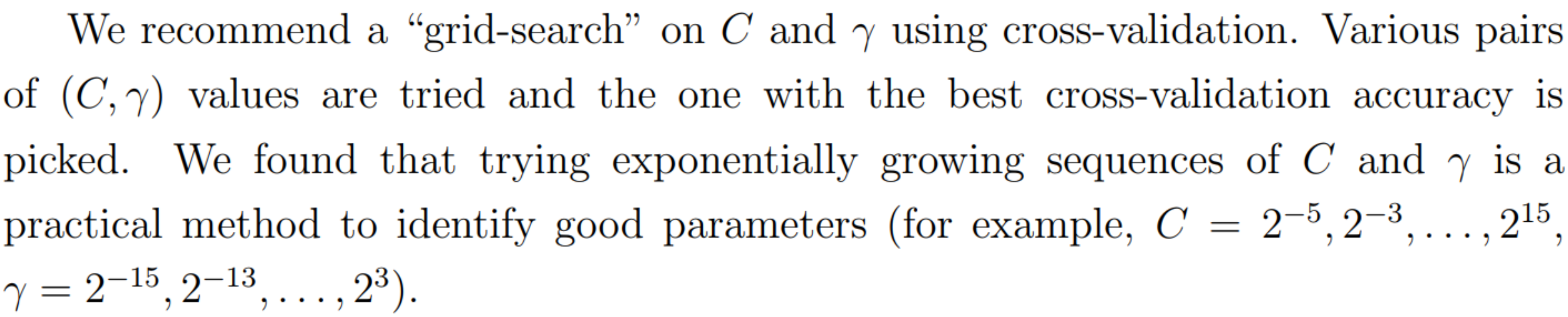
Kernelized Support Vector Classification

- **Hyperparameters for Kernelized Support Vector Classification**
  - *C, kernel (default='rbf'), gamma (if 'rbf' kernel):*
    - The *gamma* hyperparameter determines how far the influence of a single training data point reaches
    - lower value of gamma → lower model complexity (underfitting)
    - higher value of gamma → higher model complexity (overfitting)

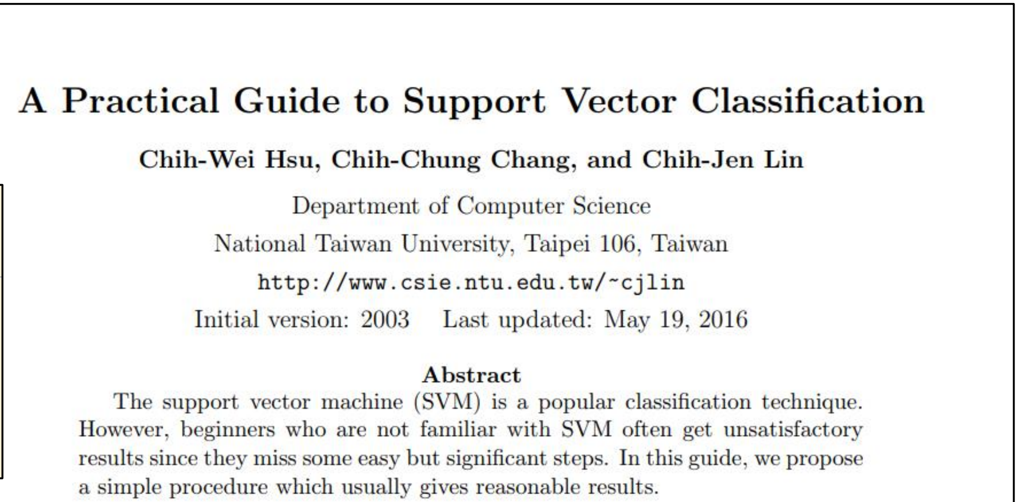


Kernelized Support Vector Classification

- **Practical Guideline when using SVC with RBF Kernel**



We recommend a "grid-search" on C and γ using cross-validation. Various pairs of (C, γ) values are tried and the one with the best cross-validation accuracy is picked. We found that trying exponentially growing sequences of C and γ is a practical method to identify good parameters (for example, $C = 2^{-5}, 2^{-3}, \ldots, 2^{15}$, $\gamma = 2^{-15}, 2^{-13}, \ldots, 2^{3}$).



A Practical Guide to Support Vector Classification

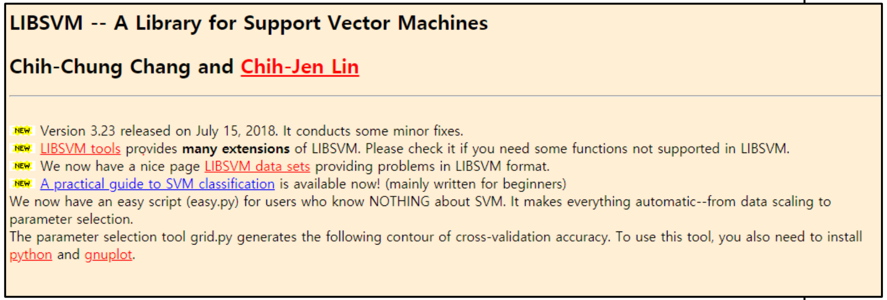
Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin

Department of Computer Science
National Taiwan University, Taipei 106, Taiwan
http://www.csie.ntu.edu.tw/~cjlin

Initial version: 2003 Last updated: May 19, 2016

Abstract

The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps. In this guide, we propose a simple procedure which usually gives reasonable results.



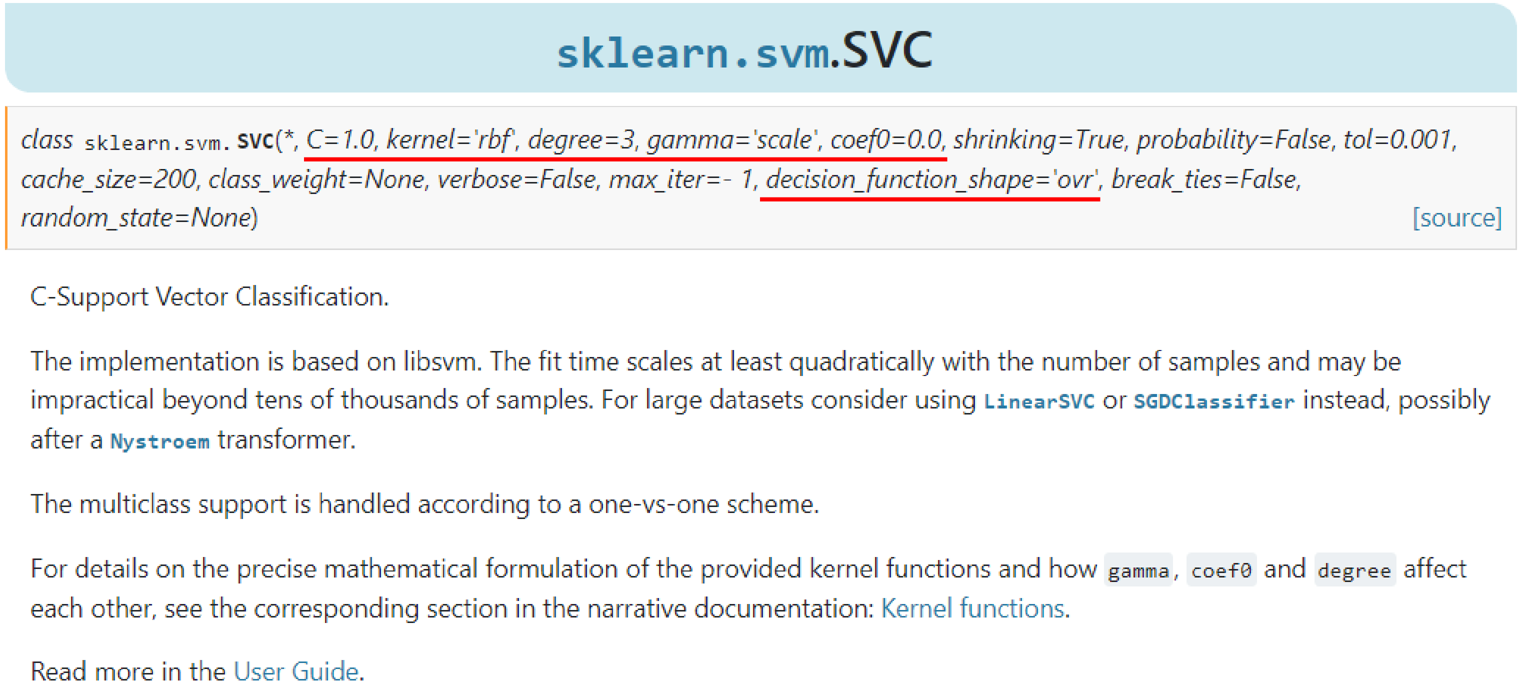
LIBSVM -- A Library for Support Vector Machines

Chih-Chung Chang and Chih-Jen Lin

NEW Version 3.23 released on July 15, 2018. It conducts some minor fixes.
NEW LIBSVM tools provides **many extensions** of LIBSVM. Please check it if you need some functions not supported in LIBSVM.
NEW We now have a nice page LIBSVM data sets providing problems in LIBSVM format.
NEW A practical guide to SVM classification is available now! (mainly written for beginners)
We now have an easy script (easy.py) for users who know NOTHING about SVM. It makes everything automatic--from data scaling to parameter selection.
The parameter selection tool grid.py generates the following contour of cross-validation accuracy. To use this tool, you also need to install python and gnuplot.

scikit-learn Practice: *SVC*

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html



`sklearn.svm`.SVC

class `sklearn.svm.`**`SVC`**(**, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=- 1, decision_function_shape='ovr', break_ties=False, random_state=None*) [source]

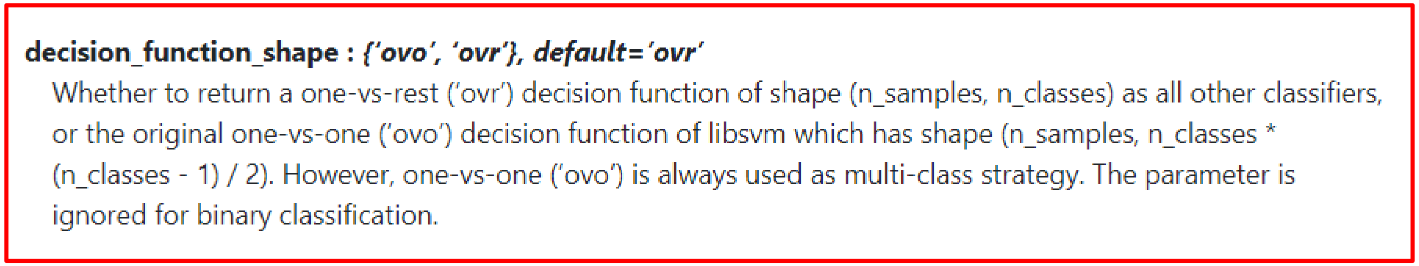
C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using `LinearSVC` or `SGDClassifier` instead, possibly after a `Nystroem` transformer.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how `gamma`, `coef0` and `degree` affect each other, see the corresponding section in the narrative documentation: Kernel functions.

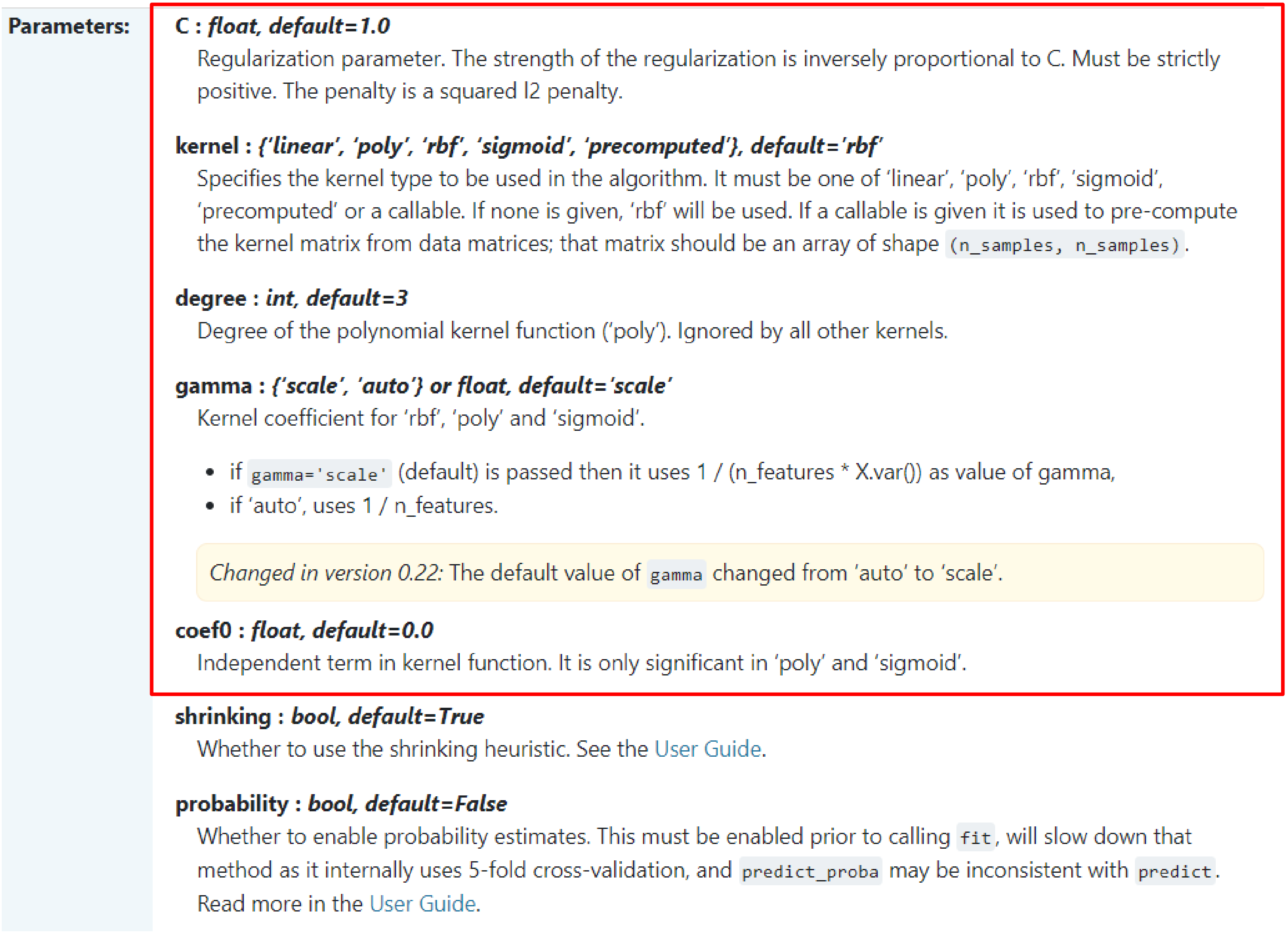
Read more in the User Guide.



decision_function_shape : *{'ovo', 'ovr'}, default='ovr'*
Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy. The parameter is ignored for binary classification.

scikit-learn Practice: *SVC*

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html



Parameters:

C : *float, default=1.0*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel : *{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'*

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree : *int, default=3*

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : *{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 : *float, default=0.0*

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking : *bool, default=True*

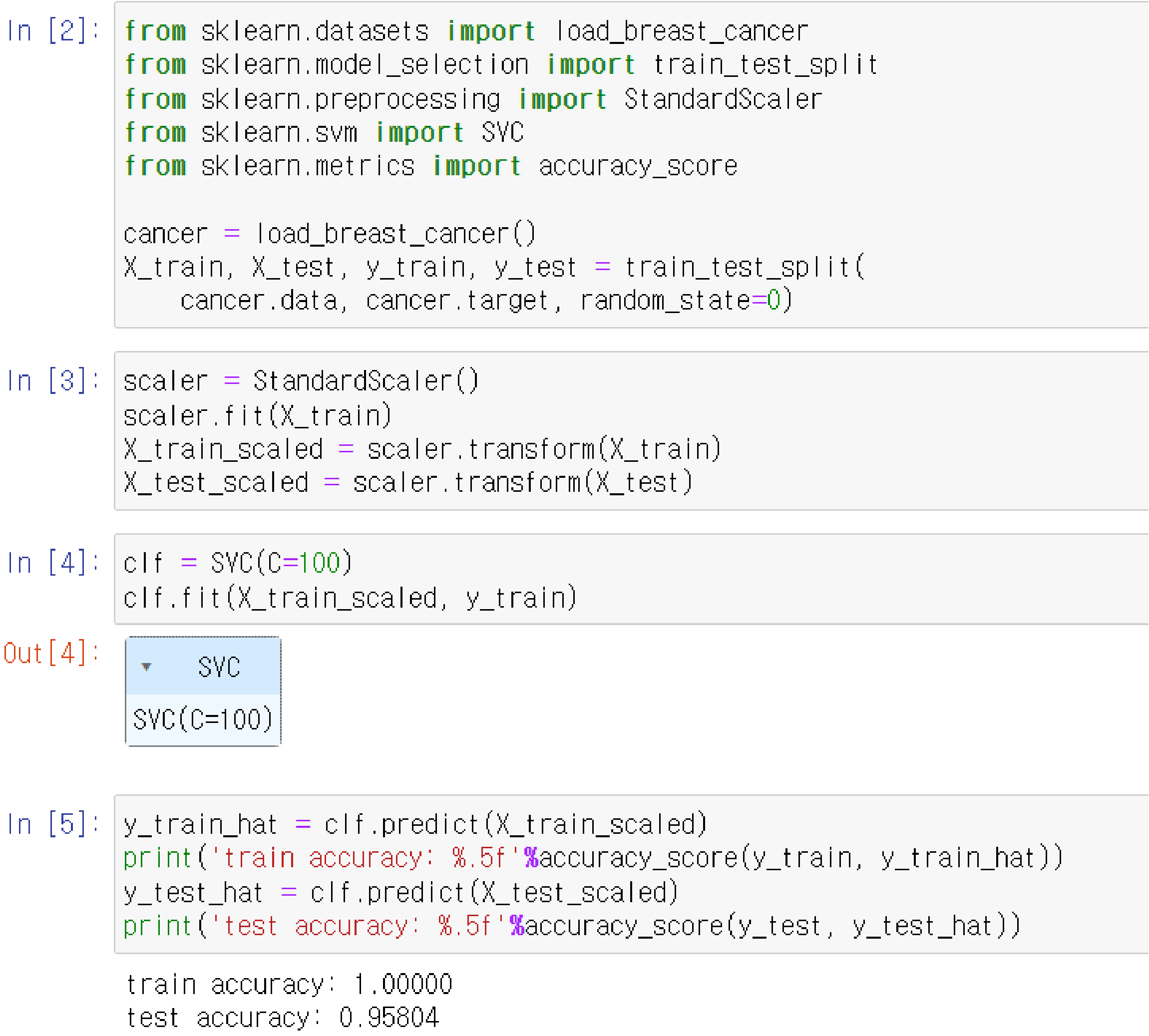
Whether to use the shrinking heuristic. See the User Guide.

probability : *bool, default=False*

Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the User Guide.

scikit-learn Practice: *SVC*

- **Example (*breast_cancer* dataset)**



```
In [2]: from sklearn.datasets import load_breast_cancer
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from sklearn.svm import SVC
        from sklearn.metrics import accuracy_score

        cancer = load_breast_cancer()
        X_train, X_test, y_train, y_test = train_test_split(
            cancer.data, cancer.target, random_state=0)
```

```
In [3]: scaler = StandardScaler()
        scaler.fit(X_train)
        X_train_scaled = scaler.transform(X_train)
        X_test_scaled = scaler.transform(X_test)
```

```
In [4]: clf = SVC(C=100)
        clf.fit(X_train_scaled, y_train)
```

```
Out[4]: ▾ SVC
        SVC(C=100)
```

```
In [5]: y_train_hat = clf.predict(X_train_scaled)
        print('train accuracy: %.5f'%accuracy_score(y_train, y_train_hat))
        y_test_hat = clf.predict(X_test_scaled)
        print('test accuracy: %.5f'%accuracy_score(y_test, y_test_hat))
```

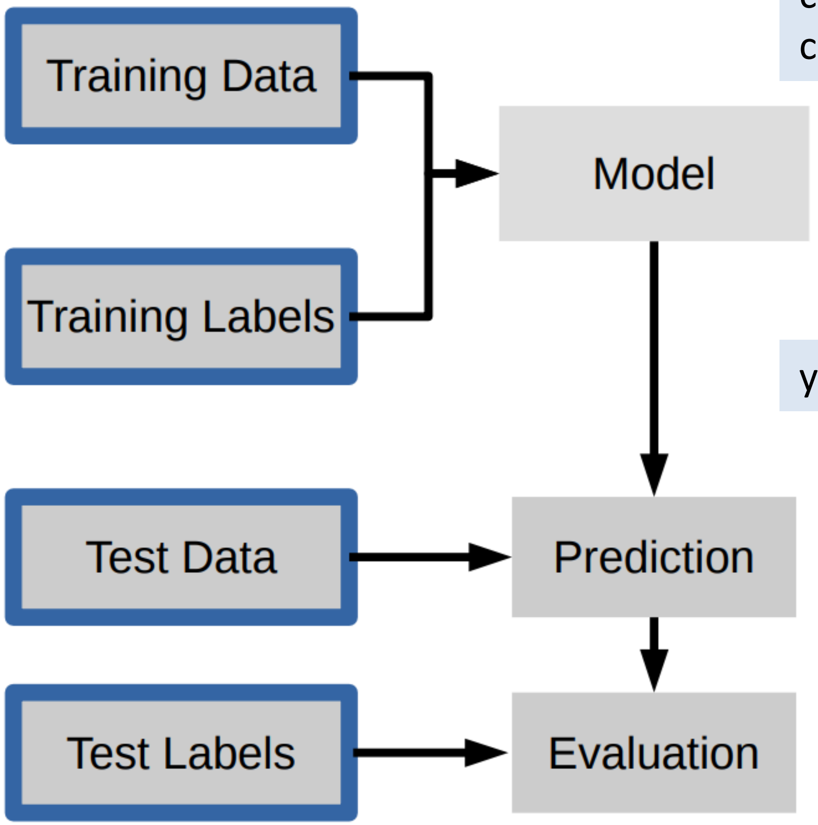
```
train accuracy: 1.00000
test accuracy: 0.95804
```

scikit-learn Practice: *SVC*

- **Example (*breast_cancer* dataset)**

```
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=42)
```

```
clf = SVC(C=100)
clf.fit(X_train, y_train)
```



X_train

y_train

X_test

y_test

clf

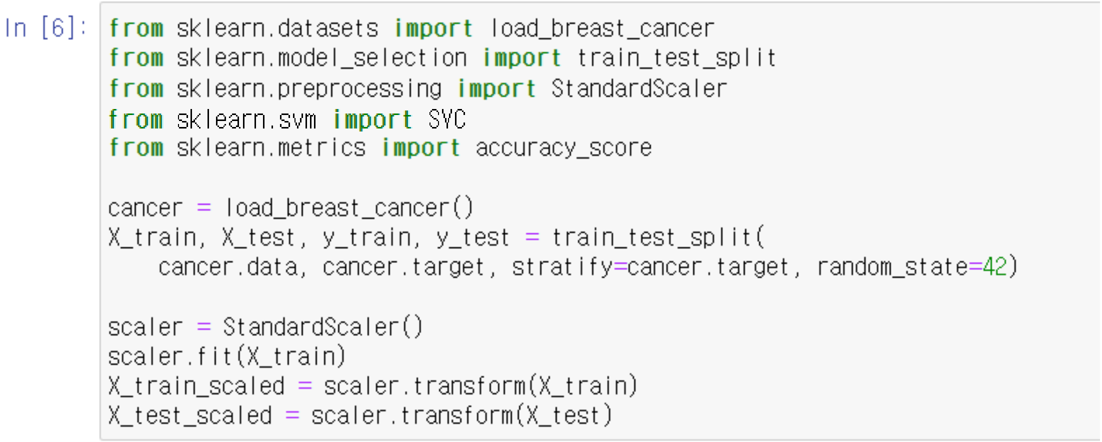
```
y_test_hat = clf.predict(X_test)
```

y_test_hat

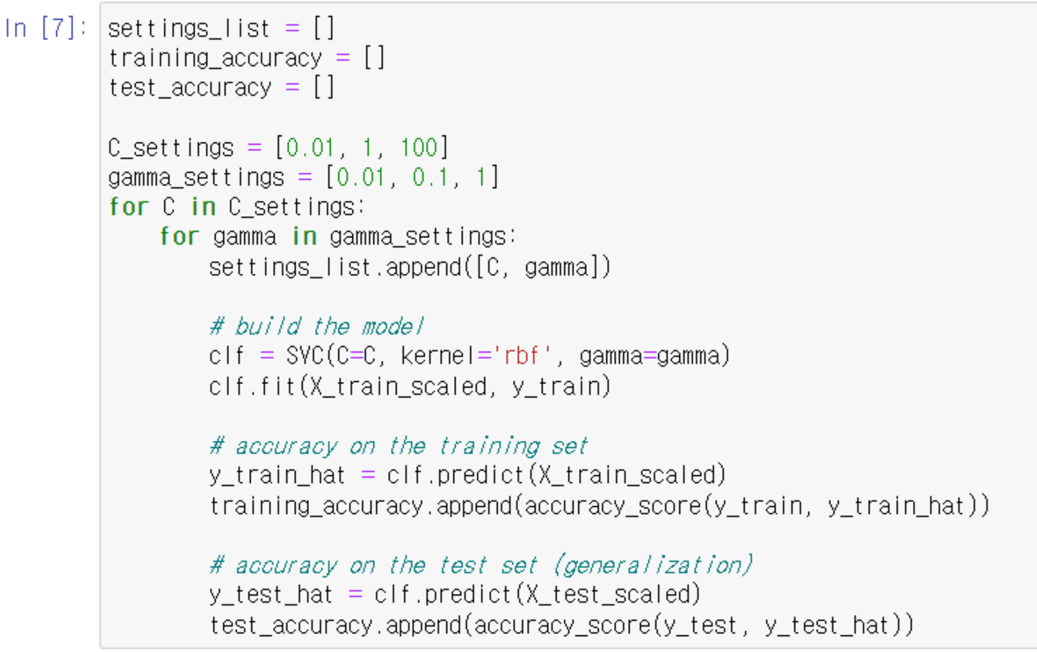
accuracy_score(y_test, y_test_hat)

scikit-learn Practice: *SVC*

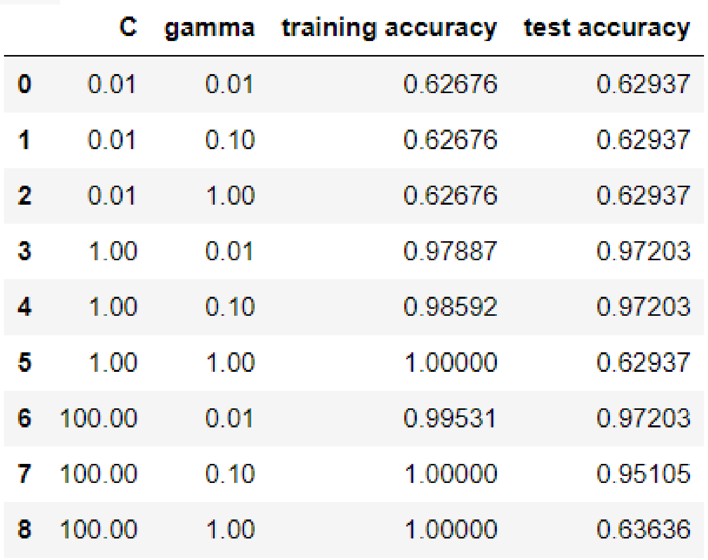
- **Example (*breast_cancer* dataset): varying the hyperparameters *C* and *gamma***



```
In [6]: from sklearn.datasets import load_breast_cancer
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from sklearn.svm import SVC
        from sklearn.metrics import accuracy_score

        cancer = load_breast_cancer()
        X_train, X_test, y_train, y_test = train_test_split(
            cancer.data, cancer.target, stratify=cancer.target, random_state=42)

        scaler = StandardScaler()
        scaler.fit(X_train)
        X_train_scaled = scaler.transform(X_train)
        X_test_scaled = scaler.transform(X_test)
```



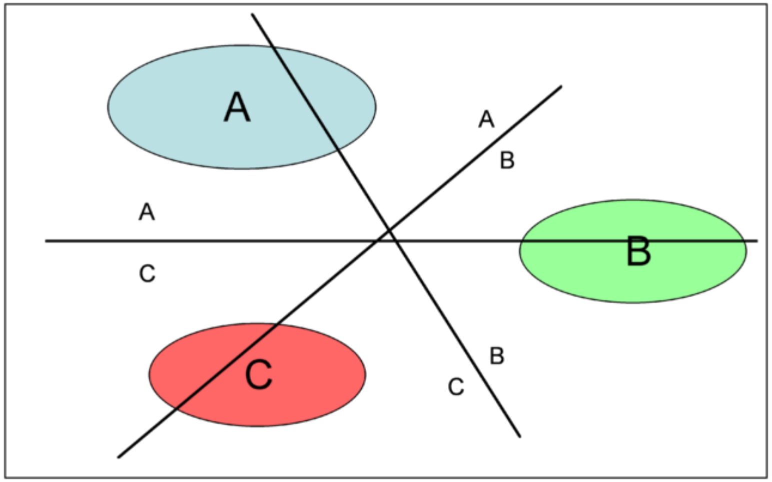
```
In [7]: settings_list = []
        training_accuracy = []
        test_accuracy = []

        C_settings = [0.01, 1, 100]
        gamma_settings = [0.01, 0.1, 1]
        for C in C_settings:
            for gamma in gamma_settings:
                settings_list.append([C, gamma])

                # build the model
                clf = SVC(C=C, kernel='rbf', gamma=gamma)
                clf.fit(X_train_scaled, y_train)

                # accuracy on the training set
                y_train_hat = clf.predict(X_train_scaled)
                training_accuracy.append(accuracy_score(y_train, y_train_hat))

                # accuracy on the test set (generalization)
                y_test_hat = clf.predict(X_test_scaled)
                test_accuracy.append(accuracy_score(y_test, y_test_hat))
```

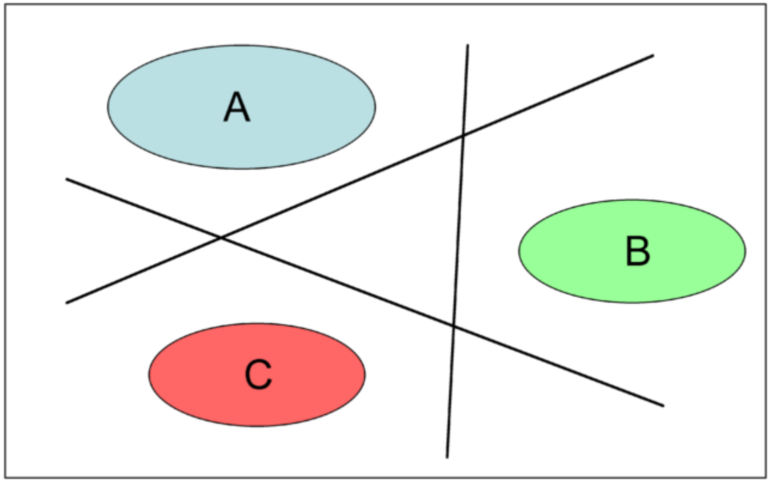


	C	gamma	training accuracy	test accuracy
0	0.01	0.01	0.62676	0.62937
1	0.01	0.10	0.62676	0.62937
2	0.01	1.00	0.62676	0.62937
3	1.00	0.01	0.97887	0.97203
4	1.00	0.10	0.98592	0.97203
5	1.00	1.00	1.00000	0.62937
6	100.00	0.01	0.99531	0.97203
7	100.00	0.10	1.00000	0.95105
8	100.00	1.00	1.00000	0.63636

SVC for Multi-Class Classification

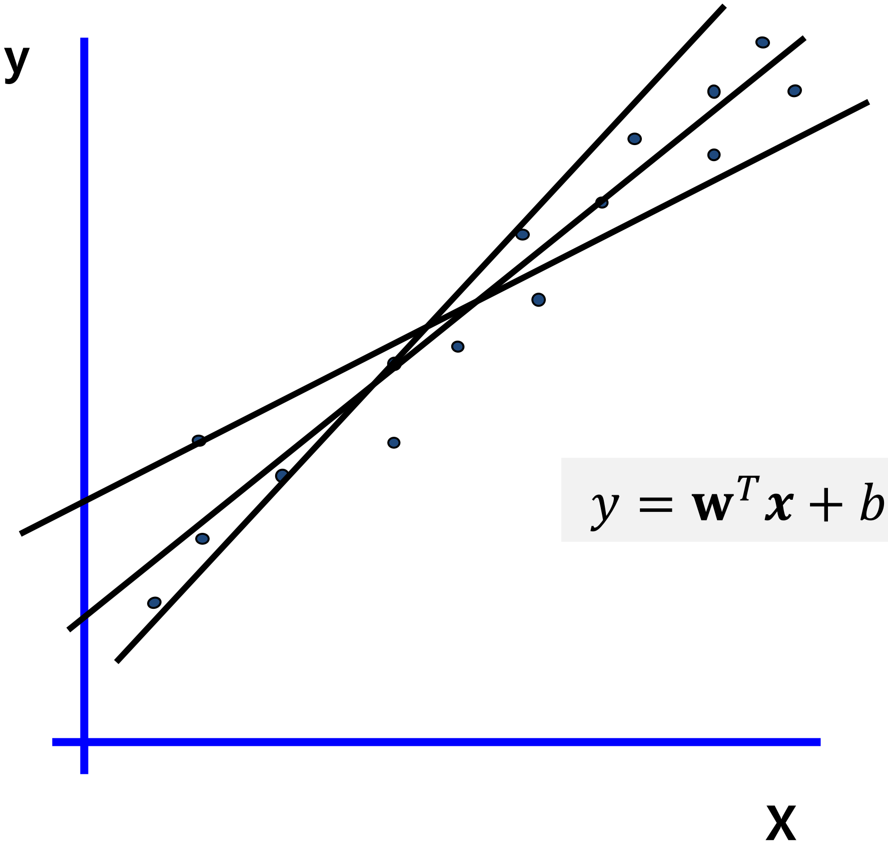
- **If multi-class classification,**
  - *decision_function_shape = 'ovr' (default) or 'ovo'*
  - ***One-vs.-Rest (OVR)* Approach**
    - A model is learned for each class that tries to separate that class from all of the other classes → *c* models
    - To make a prediction, all models are run on a test point. The model that has the highest score on its single class "wins," and this class label is returned as the prediction.
  - ***One-vs.-One (OVO) Approach***
    - A model is learned for each class pair → *c(c-1)/2* models
    - To make a prediction, the class label of a test data point is predicted based on majority voting by all models.





Support Vector Regression

- **Regression**
  - Many possible linear functions that approximately fit the training data



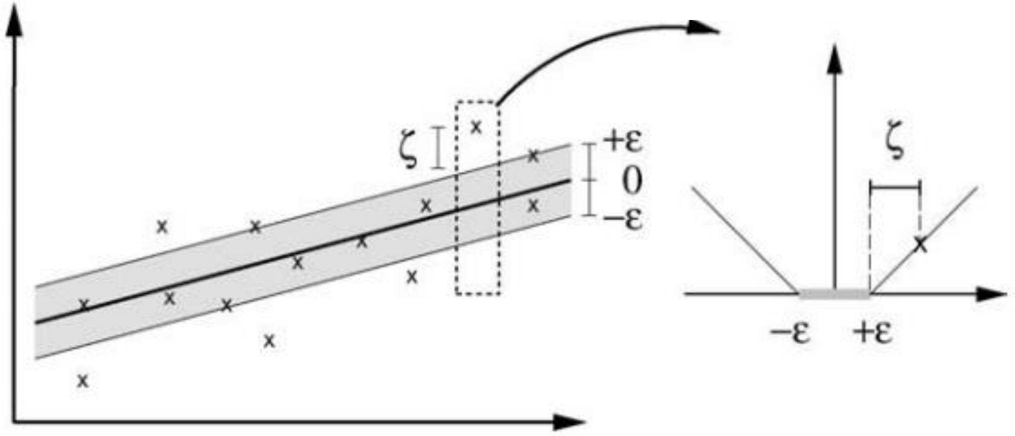
$$y = \mathbf{w}^T \boldsymbol{x} + b$$

Support Vector Regression

- Given a (training) dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of *d* features and $y_i \in \mathbb{R}$ is the corresponding target label.

- **Similar concepts apply to regression tasks → Support Vector Regression**

$$\begin{aligned} \underset{\mathbf{w},b,\xi_i,\xi_i^*}{\text{minimize}} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\left(\sum_i \xi_i + \sum_i \xi_i^*\right) \\ \text{subject to} \quad & y_i - (\mathbf{w}^T\varphi(\boldsymbol{x}_i) + b) \leqslant \epsilon + \xi_i, \\ & (\mathbf{w}^T\varphi(\boldsymbol{x}_i) + b) - y_i \leqslant \epsilon + \xi_i^*, \\ & \xi_i, \xi_i^* \geqslant 0, i = 1, \dots, N, \end{aligned}$$



ε–insensitive loss function $|\xi|_\varepsilon$ described by

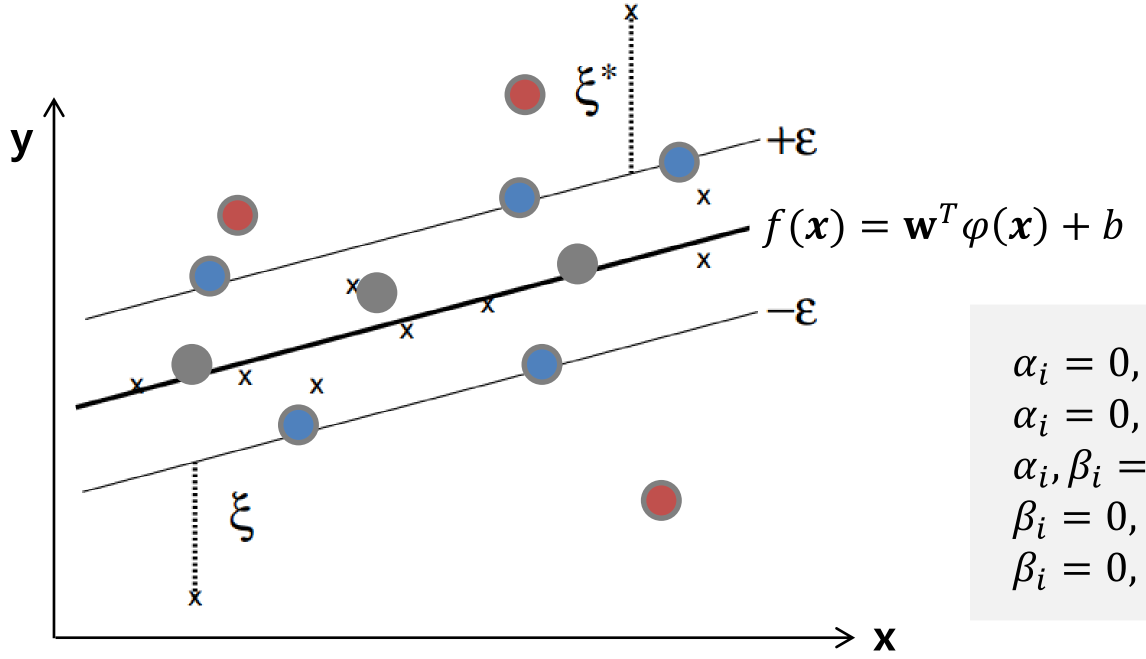
$$|\xi|_\varepsilon := \begin{cases} 0 & \text{if } |\xi| \le \varepsilon \\ |\xi| - \varepsilon & \text{otherwise.} \end{cases}$$

Support Vector Regression

- **The trained model**
  - $f(\boldsymbol{x}) = \mathbf{w}^T\phi(\boldsymbol{x}) + b = \sum_{i=1}^{N}(\alpha_i - \beta_i)k(\boldsymbol{x}_i, \boldsymbol{x}) + b$

    $2n$ parameters $\alpha_1, .., \alpha_n, \beta_1, .., \beta_n$
  - Let $D_{SV} = \{(\boldsymbol{x}_i, y_i) \in D | \alpha_i > 0 \text{ or } \beta_i > 0\}$,

    then $f(\boldsymbol{x}) = \sum_{(\boldsymbol{x}_i, y_i) \in D_{SV}}(\alpha_i - \beta_i)k(\boldsymbol{x}_i, \boldsymbol{x}) + b$ (sparse solution)

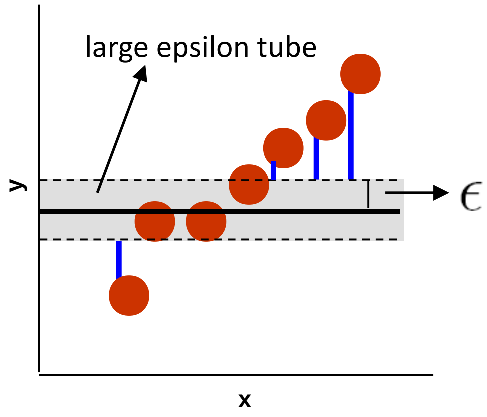

x
ξ*
+ε
y
x
$f(\boldsymbol{x}) = \mathbf{w}^T\varphi(\boldsymbol{x}) + b$
x
x
x
x
−ε
x
x
x
ξ
x
x

The trained model depends only on support vectors

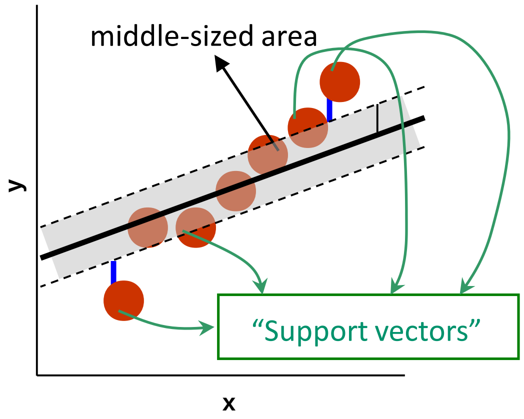
$\alpha_i = 0, \beta_i = C$	- (bounded) support vectors
$\alpha_i = 0, 0 < \beta_i < C$	- (unbounded) support vectors
$\alpha_i, \beta_i = 0$	- non-support vectors
$\beta_i = 0, 0 < \alpha_i < C$	- (unbounded) support vectors
$\beta_i = 0, \alpha_i = C$	- (bounded) support vectors

Support Vector Regression

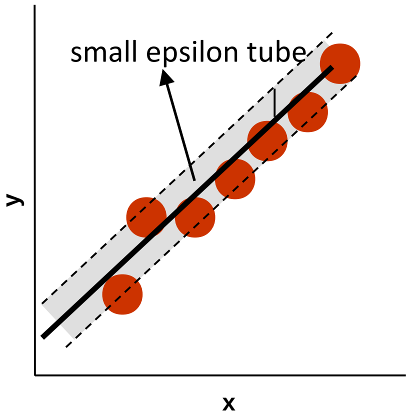
- **Hyperparameters for Linear Support Vector Regression**
  - *C, **epsilon***
- **Hyperparameters for Kernelized Support Vector Regression**
  - *C, kernel (default=‘rbf‘), gamma (if ‘rbf‘ kernel), **epsilon***
  - higher value of *epsilon* → lower model complexity (underfitting)
  - lower value of *epsilon* → higher model complexity (overfitting)



(underfitting)



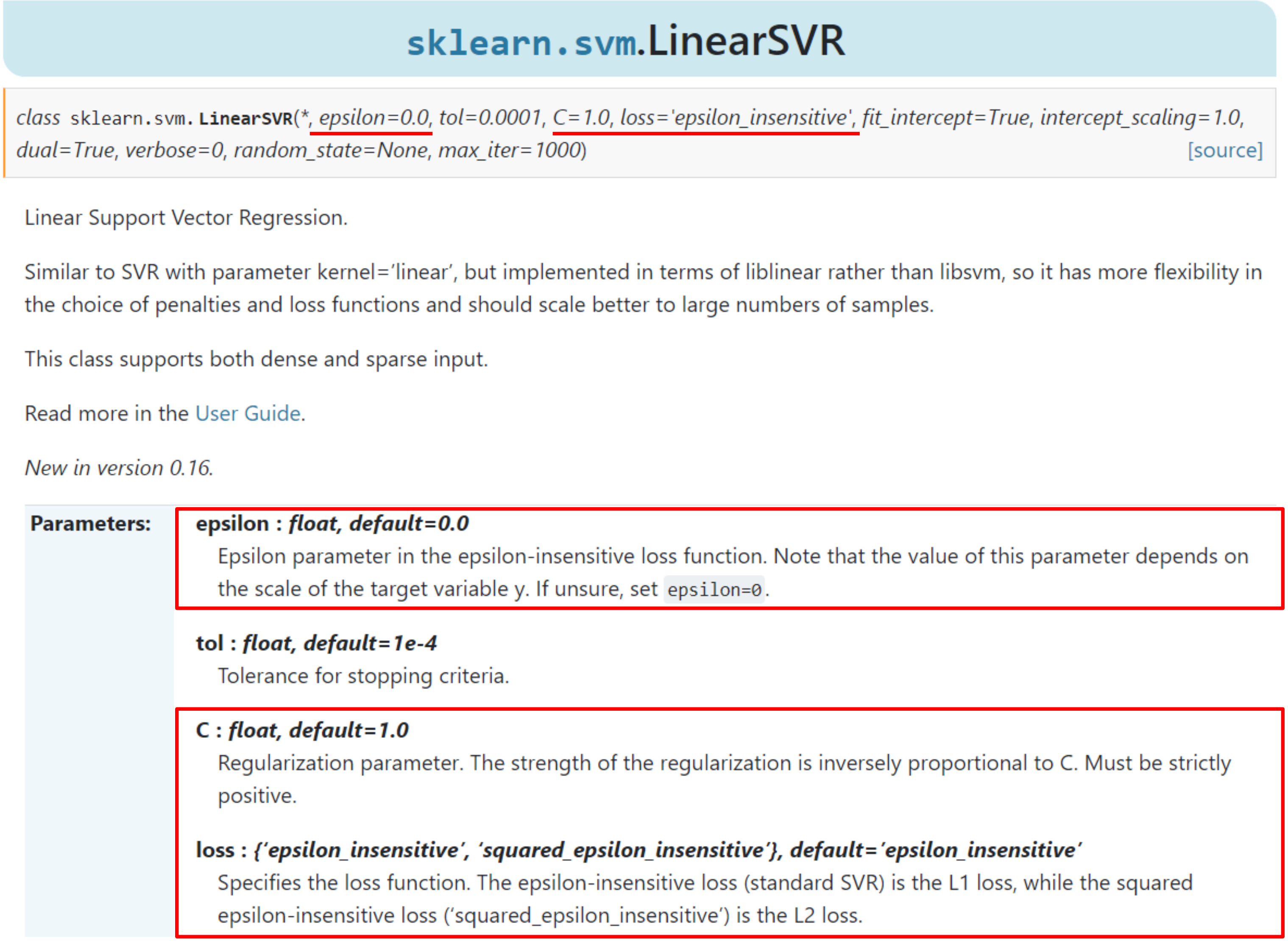
(good generalizability)



(overfitting)

scikit-learn Practice: *LinearSVR*

https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html



sklearn.svm.LinearSVR

class `sklearn.svm.`**`LinearSVR`**(**, epsilon=0.0, tol=0.0001, C=1.0, loss='epsilon_insensitive', fit_intercept=True, intercept_scaling=1.0, dual=True, verbose=0, random_state=None, max_iter=1000*) [source]

Linear Support Vector Regression.

Similar to SVR with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input.

Read more in the User Guide.

New in version 0.16.

Parameters:

epsilon : *float, default=0.0*
Epsilon parameter in the epsilon-insensitive loss function. Note that the value of this parameter depends on the scale of the target variable y. If unsure, set `epsilon=0`.

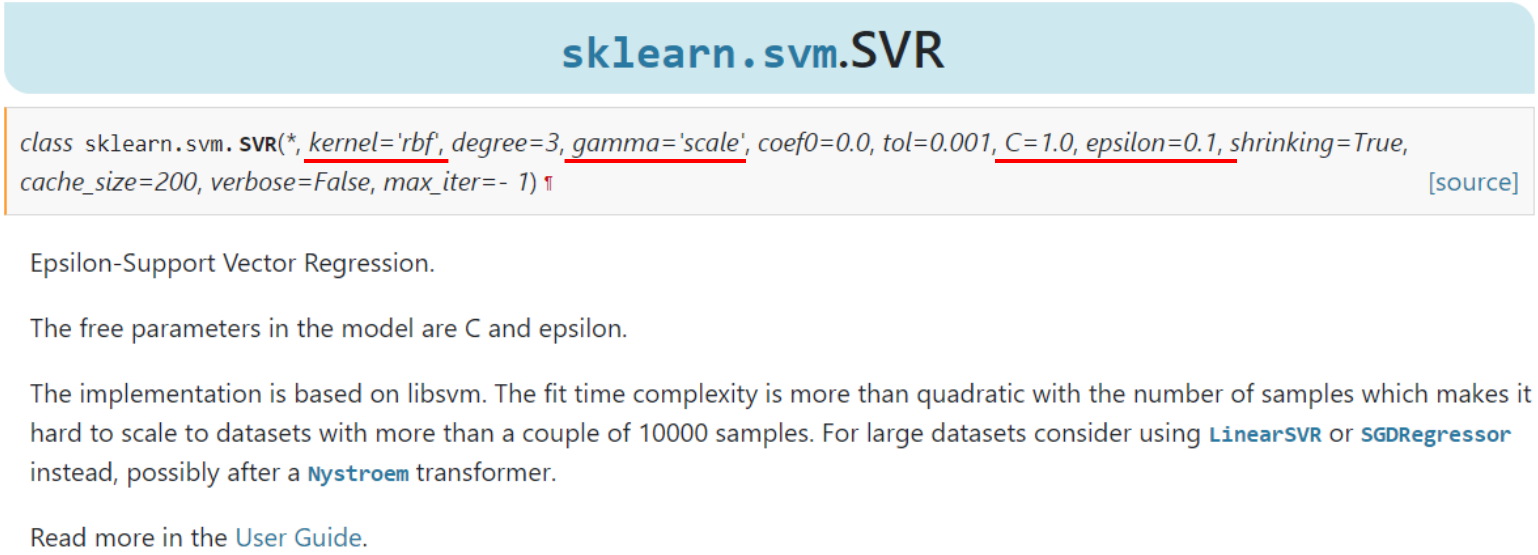
tol : *float, default=1e-4*
Tolerance for stopping criteria.

C : *float, default=1.0*
Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

loss : *{'epsilon_insensitive', 'squared_epsilon_insensitive'}, default='epsilon_insensitive'*
Specifies the loss function. The epsilon-insensitive loss (standard SVR) is the L1 loss, while the squared epsilon-insensitive loss ('squared_epsilon_insensitive') is the L2 loss.

scikit-learn Practice: *SVR*

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html



sklearn.svm.SVR

class `sklearn.svm.`**`SVR`**(**, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=- 1*) ¶ [source]

Epsilon-Support Vector Regression.

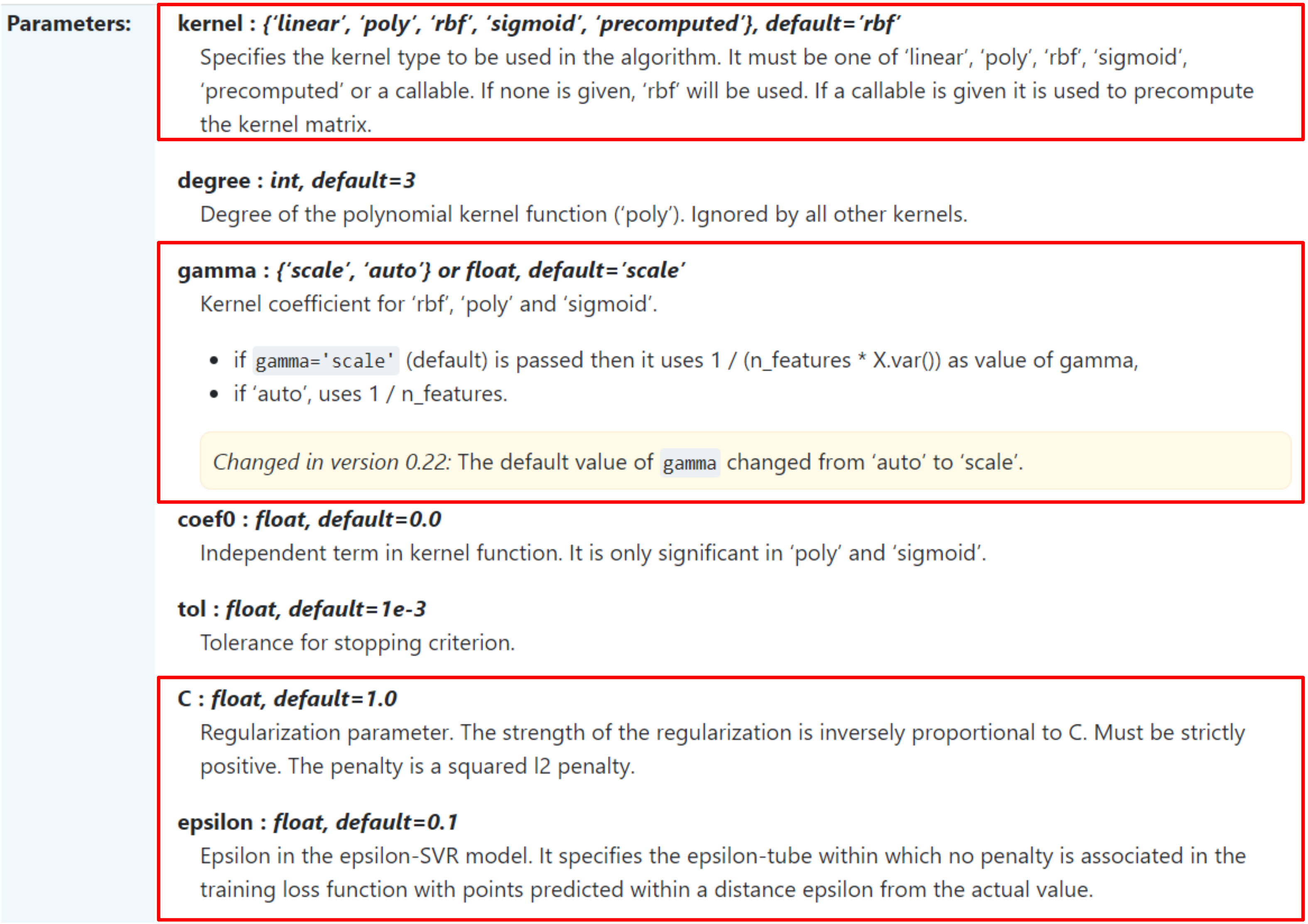
The free parameters in the model are C and epsilon.

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to datasets with more than a couple of 10000 samples. For large datasets consider using `LinearSVR` or `SGDRegressor` instead, possibly after a `Nystroem` transformer.

Read more in the User Guide.

scikit-learn Practice: *SVR*

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html



Parameters:

kernel : *{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'*

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree : *int, default=3*

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : *{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if `gamma='scale'` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

Changed in version 0.22: The default value of `gamma` changed from 'auto' to 'scale'.

coef0 : *float, default=0.0*

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol : *float, default=1e-3*

Tolerance for stopping criterion.

C : *float, default=1.0*

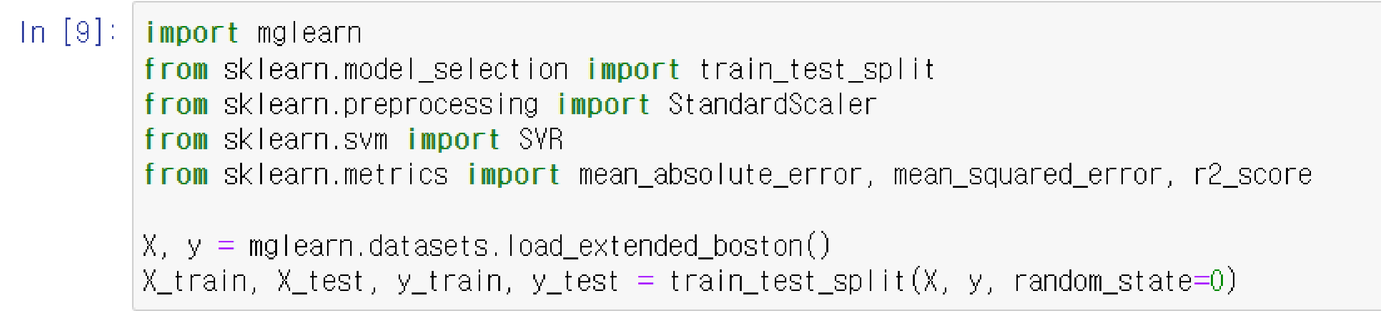
Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

epsilon : *float, default=0.1*

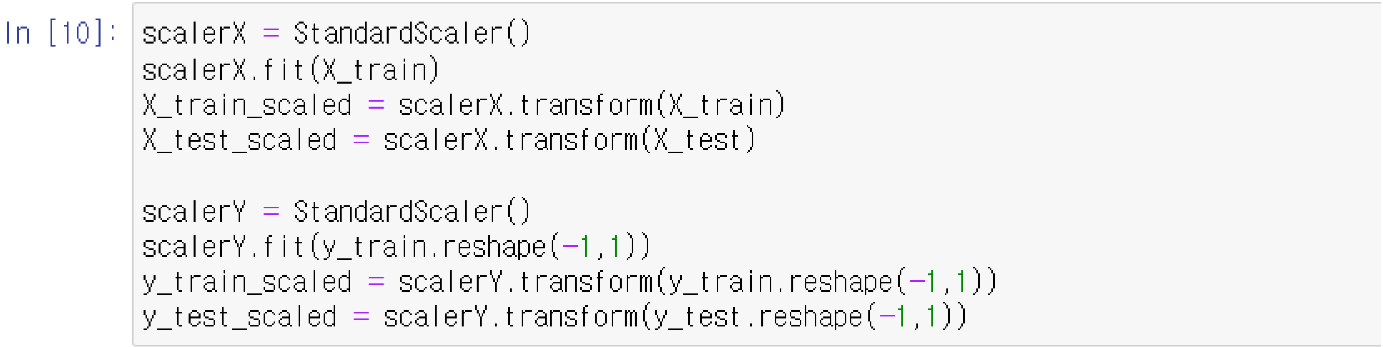
Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

scikit-learn Practice: *SVR*

- **Example (*extended_boston* dataset)**



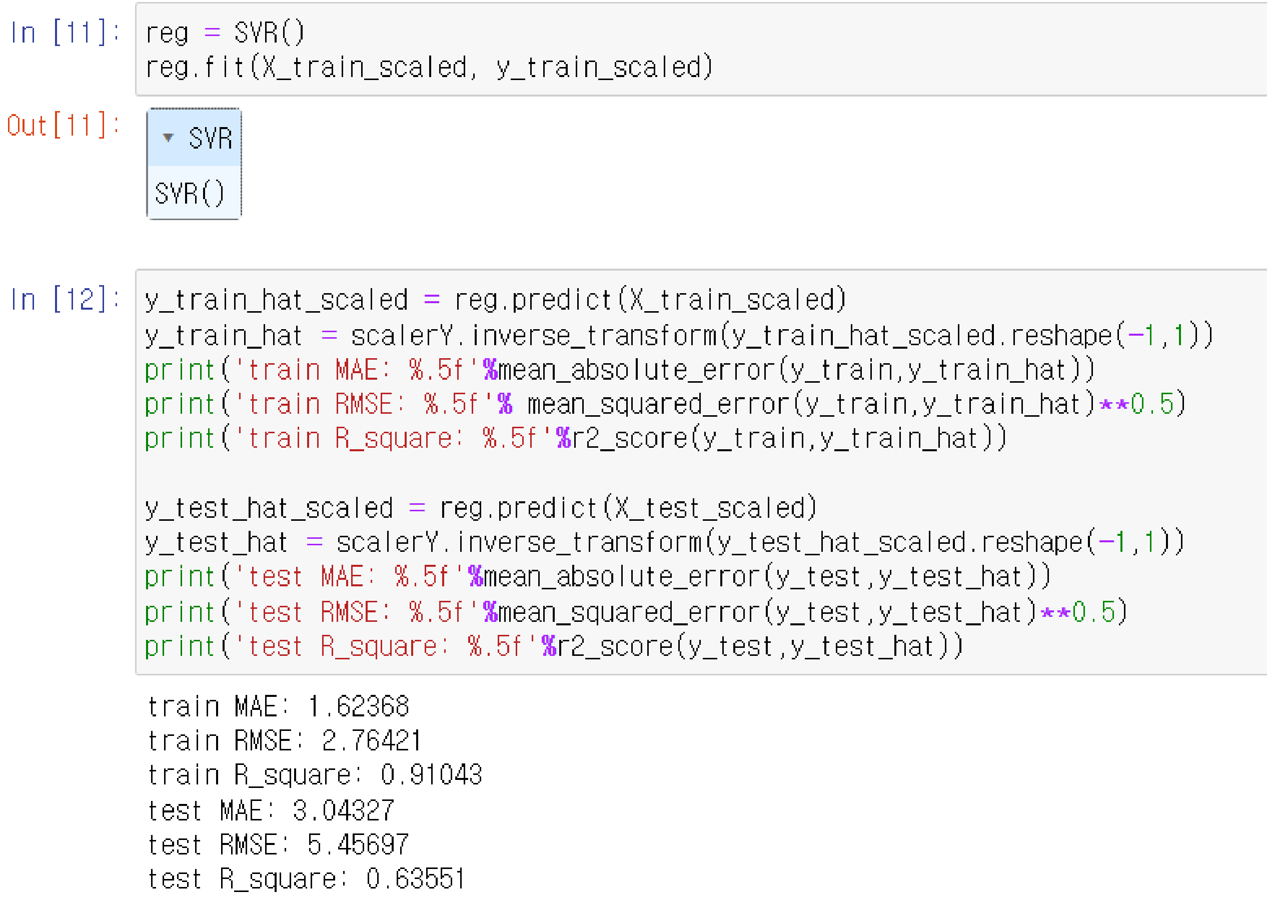
```
In [9]: import mglearn
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from sklearn.svm import SVR
        from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

        X, y = mglearn.datasets.load_extended_boston()
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```



```
In [10]: scalerX = StandardScaler()
         scalerX.fit(X_train)
         X_train_scaled = scalerX.transform(X_train)
         X_test_scaled = scalerX.transform(X_test)

         scalerY = StandardScaler()
         scalerY.fit(y_train.reshape(-1,1))
         y_train_scaled = scalerY.transform(y_train.reshape(-1,1))
         y_test_scaled = scalerY.transform(y_test.reshape(-1,1))
```

scikit-learn Practice: *SVR*

- **Example (*extended_boston* dataset)**



In [11]:

```
reg = SVR()
reg.fit(X_train_scaled, y_train_scaled)
```

Out[11]:

```
▾ SVR
SVR()
```

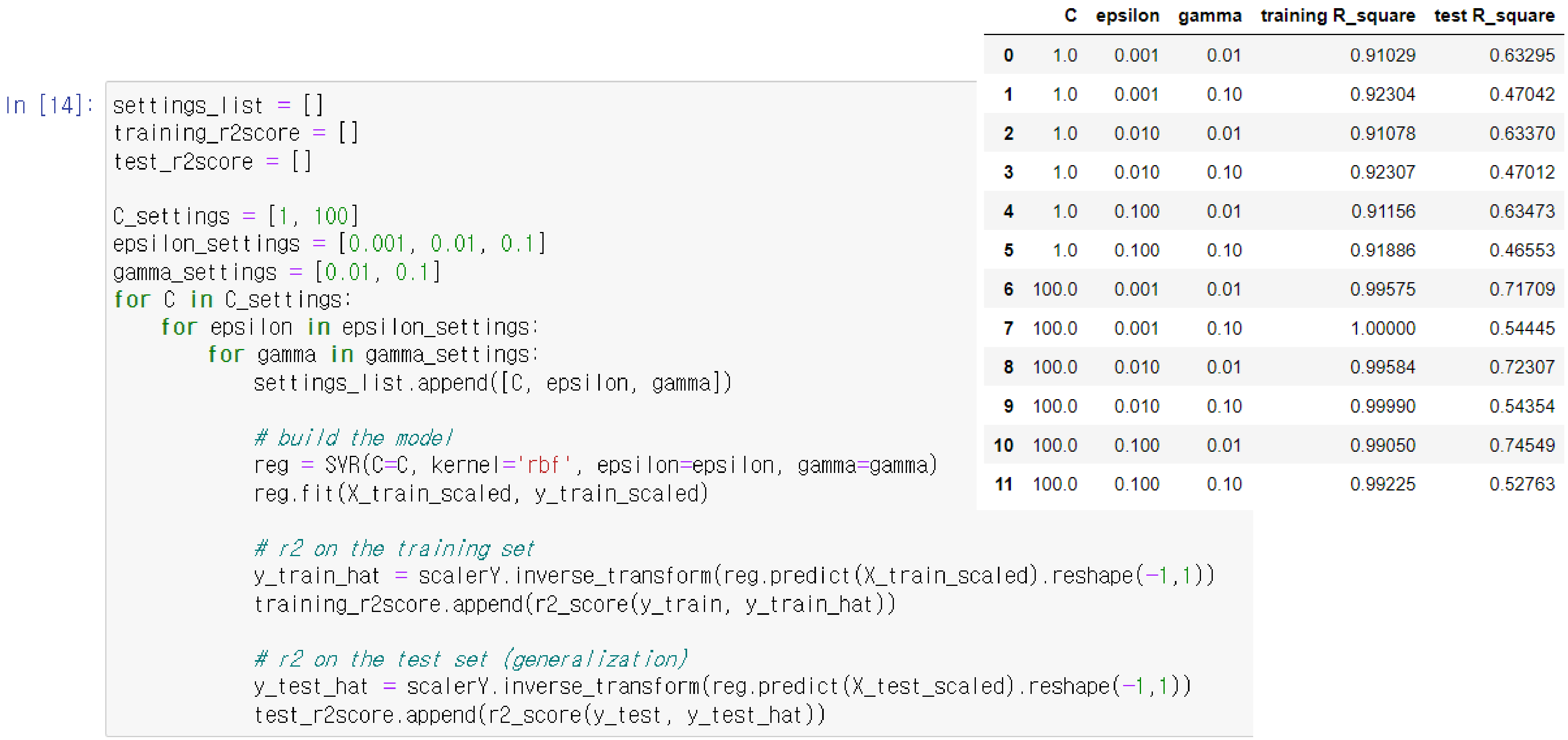
In [12]:

```
y_train_hat_scaled = reg.predict(X_train_scaled)
y_train_hat = scalerY.inverse_transform(y_train_hat_scaled.reshape(-1,1))
print('train MAE: %.5f'%mean_absolute_error(y_train,y_train_hat))
print('train RMSE: %.5f'% mean_squared_error(y_train,y_train_hat)**0.5)
print('train R_square: %.5f'%r2_score(y_train,y_train_hat))

y_test_hat_scaled = reg.predict(X_test_scaled)
y_test_hat = scalerY.inverse_transform(y_test_hat_scaled.reshape(-1,1))
print('test MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
print('test RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
print('test R_square: %.5f'%r2_score(y_test,y_test_hat))
```

```
train MAE: 1.62368
train RMSE: 2.76421
train R_square: 0.91043
test MAE: 3.04327
test RMSE: 5.45697
test R_square: 0.63551
```

scikit-learn Practice: *SVR*

- **Example (*extended_boston* dataset): varying the hyperparameters *C, epsilon,* and *gamma***



In [14]:

```
settings_list = []
training_r2score = []
test_r2score = []

C_settings = [1, 100]
epsilon_settings = [0.001, 0.01, 0.1]
gamma_settings = [0.01, 0.1]
for C in C_settings:
    for epsilon in epsilon_settings:
        for gamma in gamma_settings:
            settings_list.append([C, epsilon, gamma])

            # build the model
            reg = SVR(C=C, kernel='rbf', epsilon=epsilon, gamma=gamma)
            reg.fit(X_train_scaled, y_train_scaled)

            # r2 on the training set
            y_train_hat = scalerY.inverse_transform(reg.predict(X_train_scaled).reshape(-1,1))
            training_r2score.append(r2_score(y_train, y_train_hat))

            # r2 on the test set (generalization)
            y_test_hat = scalerY.inverse_transform(reg.predict(X_test_scaled).reshape(-1,1))
            test_r2score.append(r2_score(y_test, y_test_hat))
```

	C	epsilon	gamma	training R_square	test R_square
0	1.0	0.001	0.01	0.91029	0.63295
1	1.0	0.001	0.10	0.92304	0.47042
2	1.0	0.010	0.01	0.91078	0.63370
3	1.0	0.010	0.10	0.92307	0.47012
4	1.0	0.100	0.01	0.91156	0.63473
5	1.0	0.100	0.10	0.91886	0.46553
6	100.0	0.001	0.01	0.99575	0.71709
7	100.0	0.001	0.10	1.00000	0.54445
8	100.0	0.010	0.01	0.99584	0.72307
9	100.0	0.010	0.10	0.99990	0.54354
10	100.0	0.100	0.01	0.99050	0.74549
11	100.0	0.100	0.10	0.99225	0.52763

Discussion

- **The main hyperparameters of support vector machines**
  - *C, kernel, kernel-specific hyperparameters (for both SVC and SVR)*
  - *epsilon (for SVR)*

  * Typically chosen to have the highest performance in validation data

  * It's important to preprocess your data (including *data scaling* and *one-hot encoding*)
- **Strengths**
  - (Kernelized) SVMs perform well on a variety of datasets.
  - They allow for complex decision boundaries, even if the data has only a few features.
- **Weaknesses**
  - They don't scale very well with the number of data points. (Working with datasets of size 100,000 or more can become challenging in terms of runtime and memory usage.)
  - They require careful preprocessing of the data and tuning of the hyperparameters. (Good settings for the hyperparameters are usually strongly correlated.)
  - SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made.