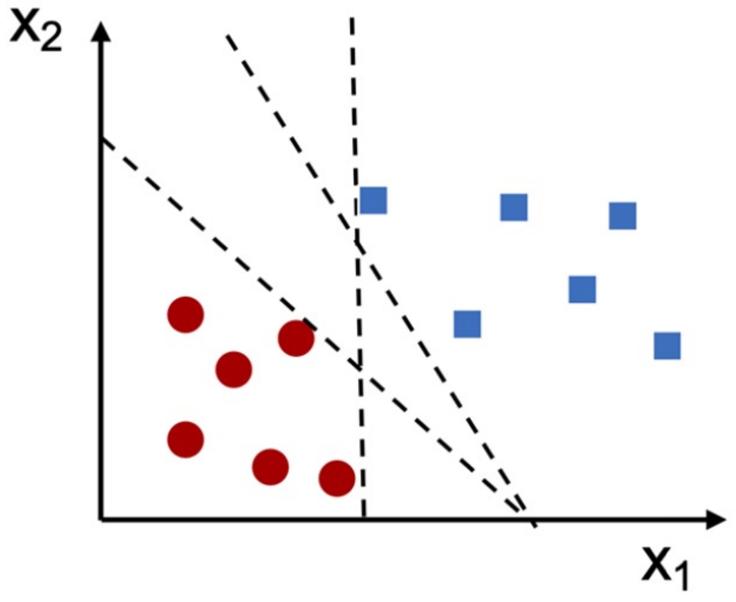
Scikit-learn and Tour of Classifiers

Support Vector Machines

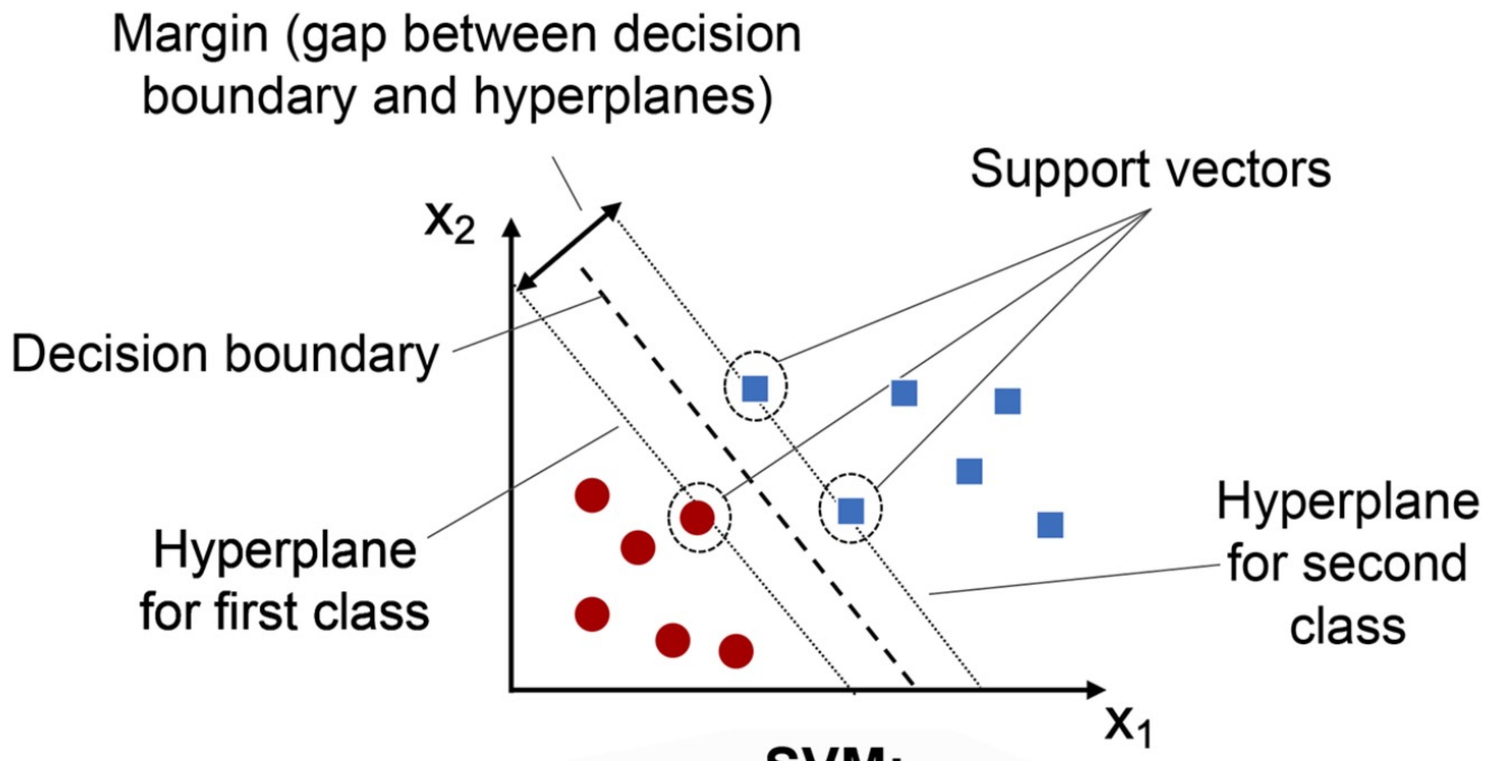
Support Vector Machines

- Widely-used, powerful classifiers
- Can be considered an extension of the Perceptron
- Support vector classification (SCV) objective: **maximize margin**
- Margin: the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane
- The training samples closest to the hyperplane are called **support vectors**
- Kernel-SVM for nonlinear decision boundaries, resulting predictor is sparse

Support Vector Machines



Which hyperplane?



Margin (gap between decision boundary and hyperplanes)
Support vectors
x_2
Decision boundary
Hyperplane for first class
Hyperplane for second class
x_1

SVM:
Maximize the margin

Figure from Python Machine Learning (Raschka & Mirjalili)

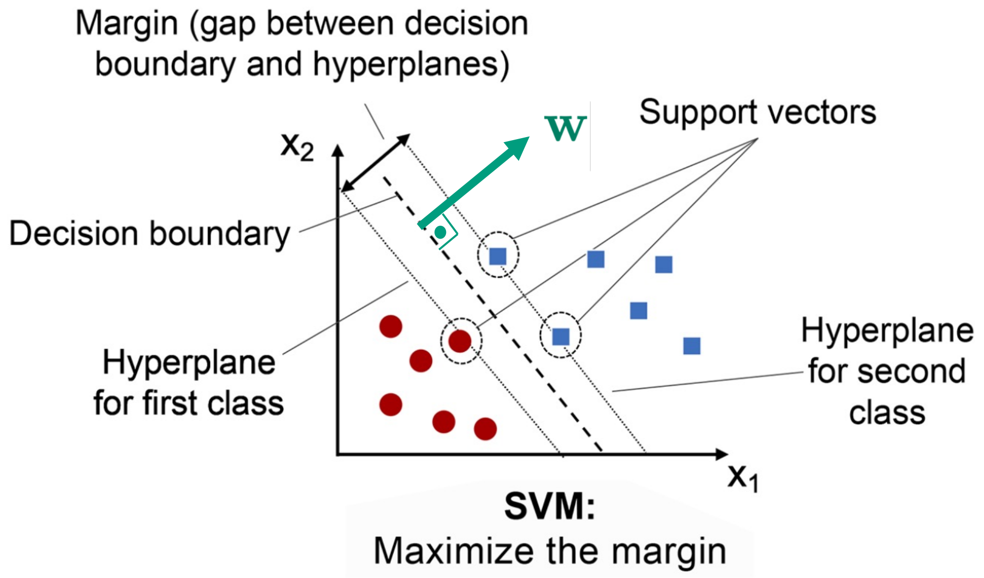
Maximum margin classifier

- Decision boundaries with **large margins** tend to have a **lower generalization error**
- Decision boundaries with **small margins** are more **prone to overfitting**
- **Linear decision boundary**:

$$\mathbf{x}^{(i)}\boldsymbol{\theta} = b + \mathbf{x}^{(i)}\underbrace{\mathbf{w}}_{\text{normal vector on hyperplane}} = 0$$

normal vector
on **hyperplane**

$$\mathbf{x}^{(i)} = \begin{bmatrix} 1 & x_1^{(i)} & x_2^{(i)} & \cdots & x_m^{(i)} \end{bmatrix} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_m \end{bmatrix} := \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_m \end{bmatrix}, \quad b = \theta_0$$



Margin (gap between decision boundary and hyperplanes)
w
Support vectors
x2
Decision boundary
Hyperplane for first class
Hyperplane for second class
x1
SVM:
Maximize the margin

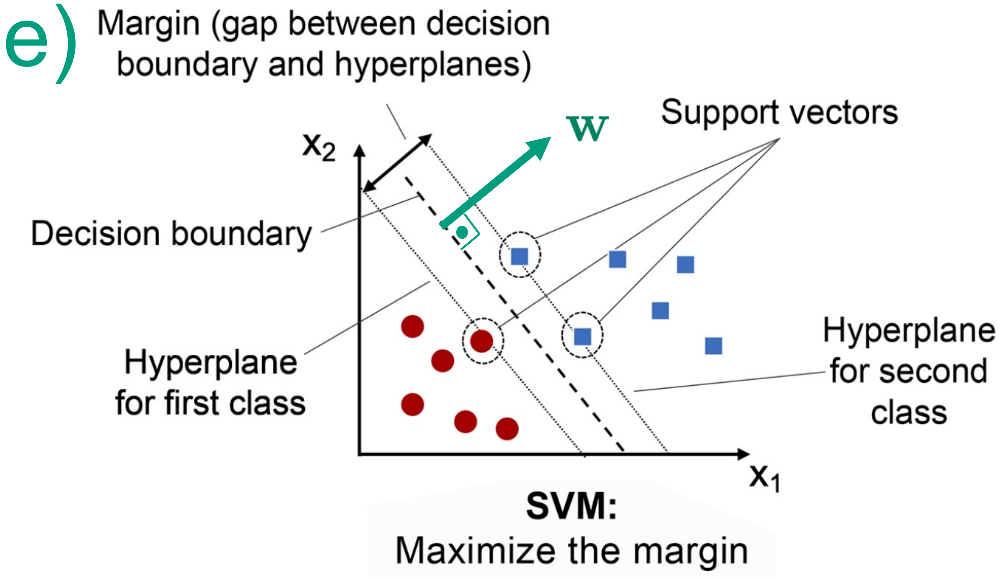
- **Distance** between positive and negative **hyperplanes**: 2 / ||**w**||
- **Maximize distance** while **classifying samples correctly**

Figure from Python Machine Learning (Raschka & Mirjalili)

Maximum margin (linearly separable)

- Derivation of maximum margin formula not syllabus; see e.g. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote09.html
- Finding the **maximum margin** hyperplanes can be cast into the **quadratic optimization problem**:

$$\boxed{\begin{aligned}&\min_{\mathbf{w},b} \mathbf{w}^T\mathbf{w}\\ &\text{such that for all } i \quad y^{(i)}\left(\mathbf{x}^{(i)}\mathbf{w}+b\right)\geq 1\end{aligned}}$$



Margin (gap between decision boundary and hyperplanes)
w
Support vectors
x2
Decision boundary
Hyperplane for first class
Hyperplane for second class
x1
SVM:
Maximize the margin

$$\boxed{\text{Classes: } \quad y^{(i)} \in \{-1, 1\}}$$

- For **support vectors** (**data on the hyperplanes**), it holds

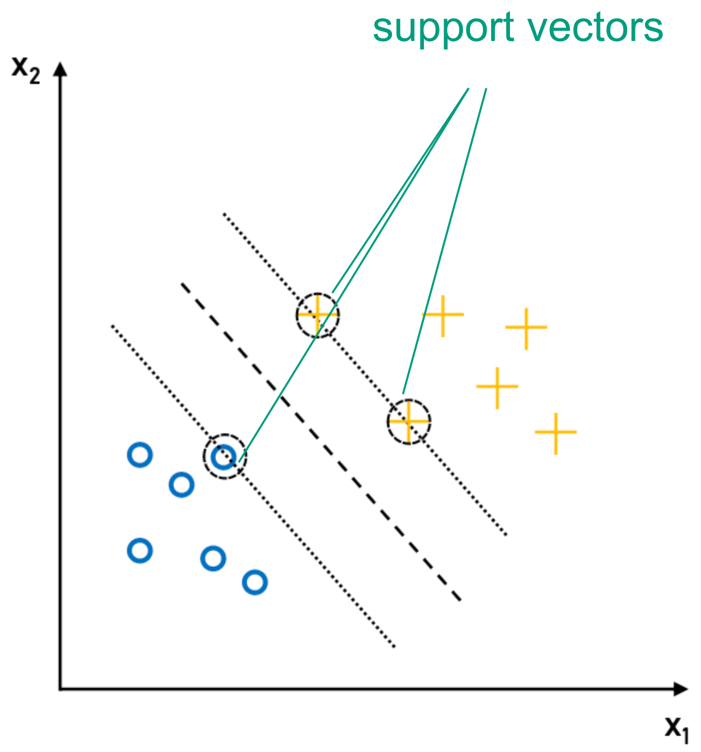
$$y^{(i)}\left(\mathbf{x}^{(i)}\mathbf{w}+b\right)=1$$

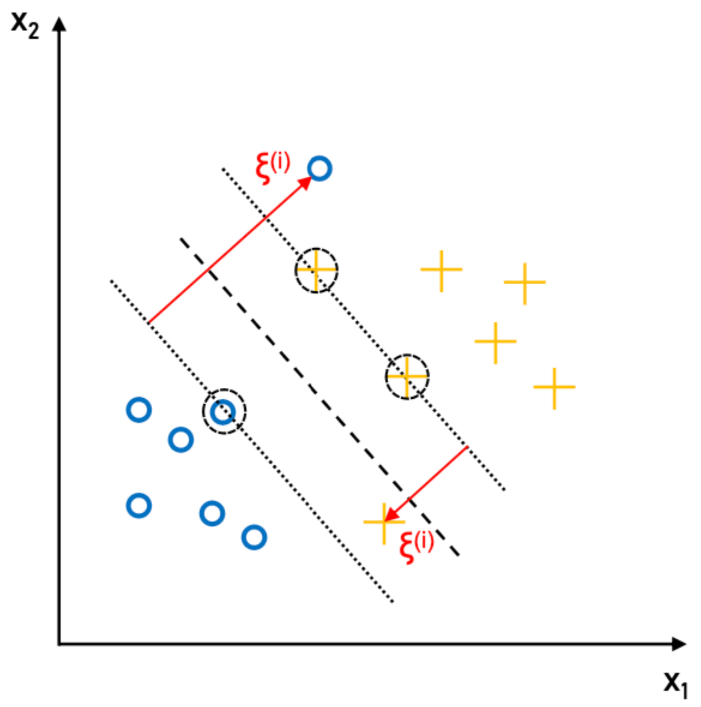
Recall that: $\mathbf{w}^T\mathbf{w} = ||\mathbf{w}||_2^2 = \sum_{i=1}^{m} w_i^2$

Figure from Python Machine Learning (Raschka & Mirjalili)

Maximum margin – soft margin

- Often, data is not linearly separable. Idea: soft margin by introducing “**slack variables**” $\xi^{(i)}$





Maximum margin – soft margin

- Often, data is not linearly separable. Idea: soft margin by introducing "**slack variables**" $\xi^{(i)}$
- Modified optimization problem:

$$\boxed{\min_{\mathbf{w},b} \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n}\xi^{(i)} \quad \text{such that for all } i \quad y^{(i)}\left(\mathbf{x}^{(i)}\mathbf{w}+b\right) \geq 1-\xi^{(i)} \quad \text{and} \quad \xi^{(i)} \geq 0}$$

- The slack variables allow for some samples to be within the margin (or even on the wrong side = misclassified)
- For high **C** we penalize sample in or on the wrong side of the margin strongly
- The optimization problem with slack variables can be written in terms of the loss function:

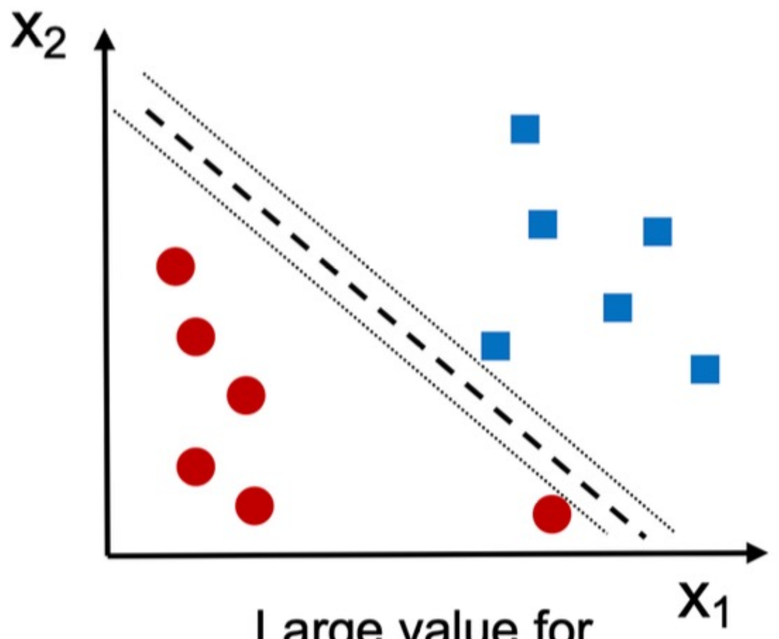
$$\boxed{L(\boldsymbol{\theta}) = L(\mathbf{w},b) = \underbrace{\mathbf{w}^T\mathbf{w}}_{\ell_2-\text{regul.}} + C\sum_{i=1}^{n}\underbrace{\max\left\{1-y^{(i)}\left(\mathbf{x}^{(i)}\mathbf{w}+b\right),0\right\}}_{\text{hinge-loss}}}$$

Recall that:

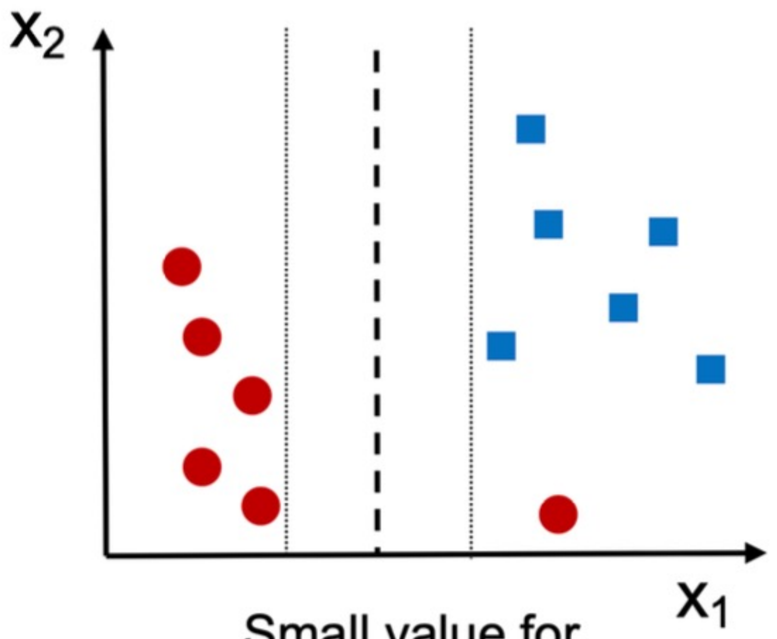
$$\mathbf{w}^T\mathbf{w} = ||\mathbf{w}||_2^2 = \sum_{i=1}^{m} w_i^2$$

Maximum margin – soft margin

- **Effect of C:** decreasing C increases the bias (underfitting) and lowers the variance (overfitting) of the model



Large value for parameter C

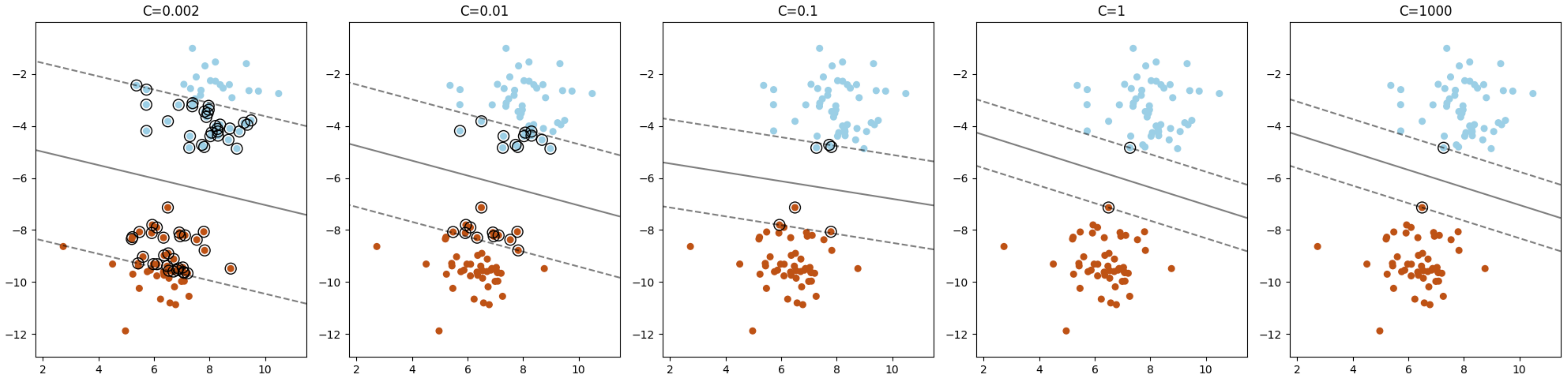


Small value for parameter C

Figure from Python Machine Learning (Raschka & Mirjalili)

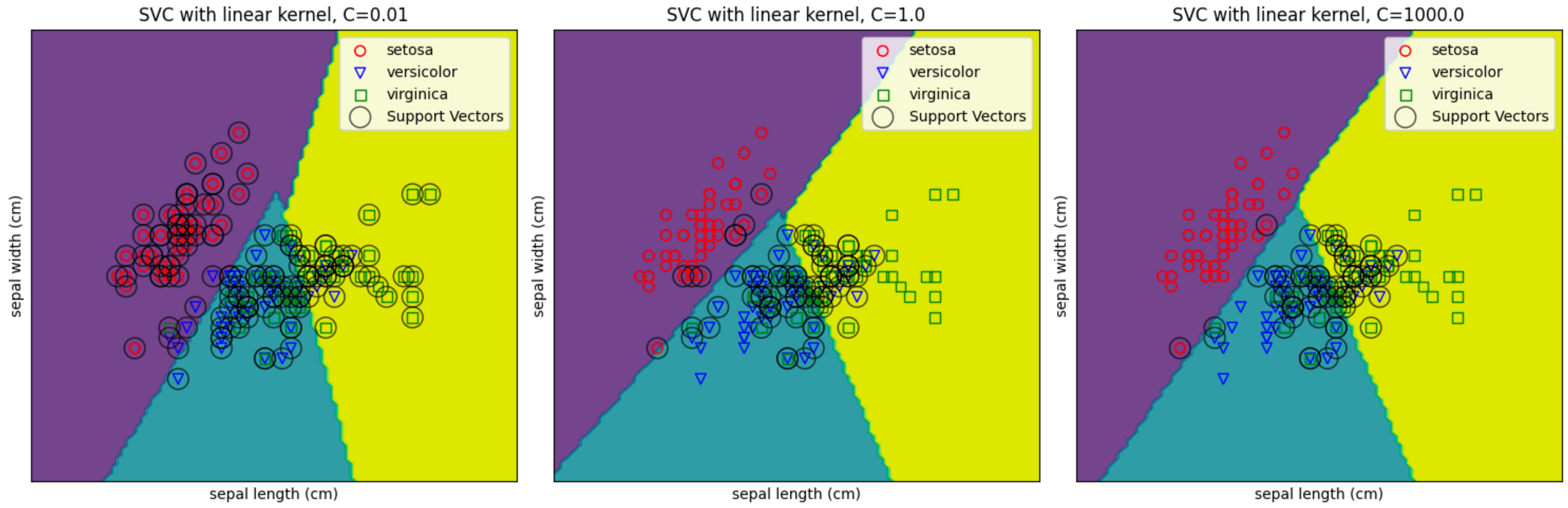
SVM code example

03_svm_linear_blobs.ipynb



SVM code example

03_svm_linear_iris.ipynb



Logistic regression versus Linear SVM

- LR and linear SVM often yield very similar results
- **Logistic regression** tries to **maximize** the **conditional likelihoods** for all of the training data, which makes it **more prone** to **outliers** than **SVMs**
- **SVMs** mostly care about the samples that are **closest** to the **decision boundary** (**support vectors**)
- **Logistic regression** has the advantage that it is a **simpler model** and can be implemented more easily

Linear model (linear decision boundaries)

- **Linear model classifiers** may **appear** to be very **restrictive** in **low-dimensional** spaces (**very few** features in $\boldsymbol{X}$) because of their **straight line or plane** boundaries
- For **high dimensional** data (**many** features in $\boldsymbol{X}$) linear model classifiers may act as a **guard against** overfitting

Scikit-learn implementation

- `Perceptron`, `LogisticRegression` and `LinearSVC` classes in scikit-learn

  make use of the **LIBLINEAR** library (highly optimized C/C++ library developed at the National Taiwan University)

- `SVC` and `SVM` classes in scikit-learn

  makes use of **LIBSVM** library (an equivalent C/C++ library specialized for SVMs developed at the National Taiwan University)

- LIBLINEAR and LIBSVM are **faster** than **native** Python implementations

Scikit-learn implementation

- `Perceptron`, `LogisticRegression` and `LinearSVC` classes in scikit-learn

  make use of the **LIBLINEAR** library (highly optimized C/C++ library developed at the National Taiwan University)

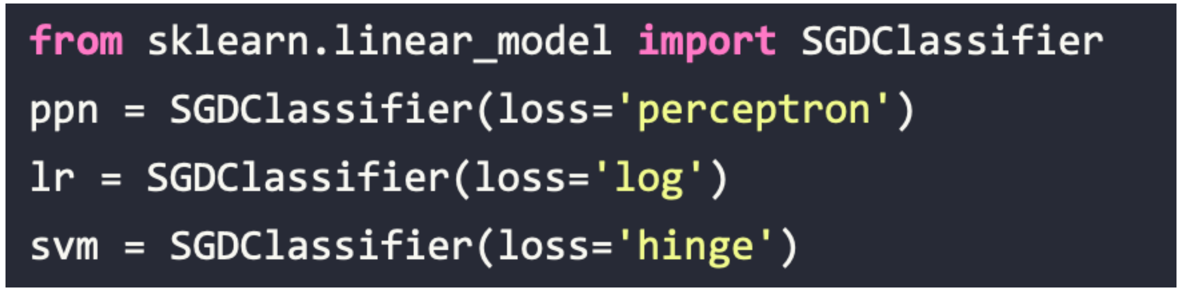
- `SVC` and `SVM` classes in scikit-learn

  makes use of **LIBSVM** library (an equivalent C/C++ library specialized for SVMs developed at the National Taiwan University)

- LIBLINEAR and LIBSVM are **faster** than **native** Python implementations

Scikit-learn alternative implementation

- When datasets are **too large** to fit into computer memory
  - Alternative implementations using the `SGDClassifier` class
  - `SGDClassifier` class supports online learning via the `partial_fit` method
  - Concept behind the `SGDClassifier` class is **the stochastic gradient descend algorithm**
- **Initialize** the stochastic gradient descent version of the **perceptron (`ppn`)**, **logistic regression (`lr`)**, and a **support vector machine (`svm`)** with default parameters as follows

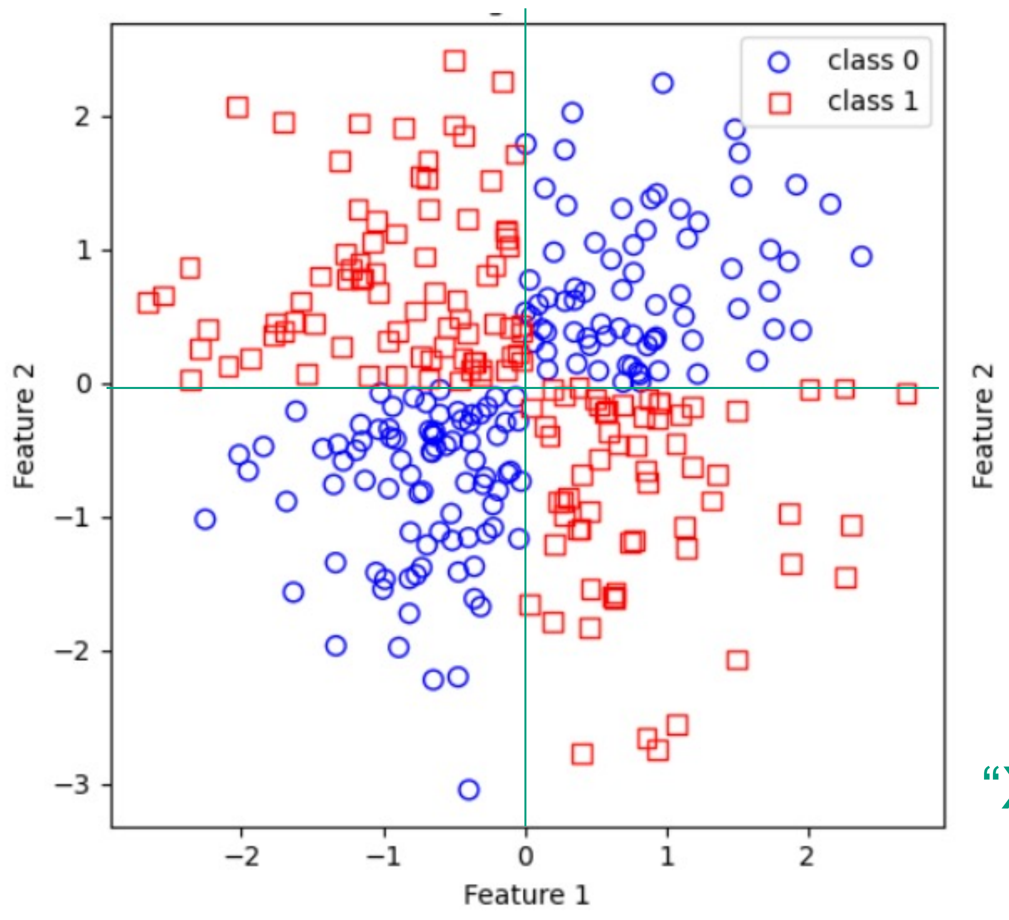


```python
from sklearn.linear_model import SGDClassifier
ppn = SGDClassifier(loss='perceptron')
lr = SGDClassifier(loss='log')
svm = SGDClassifier(loss='hinge')
```

Nonlinear decision boundaries

Kernel Support Vector Machines

Nonlinear decision boundaries – Motivation

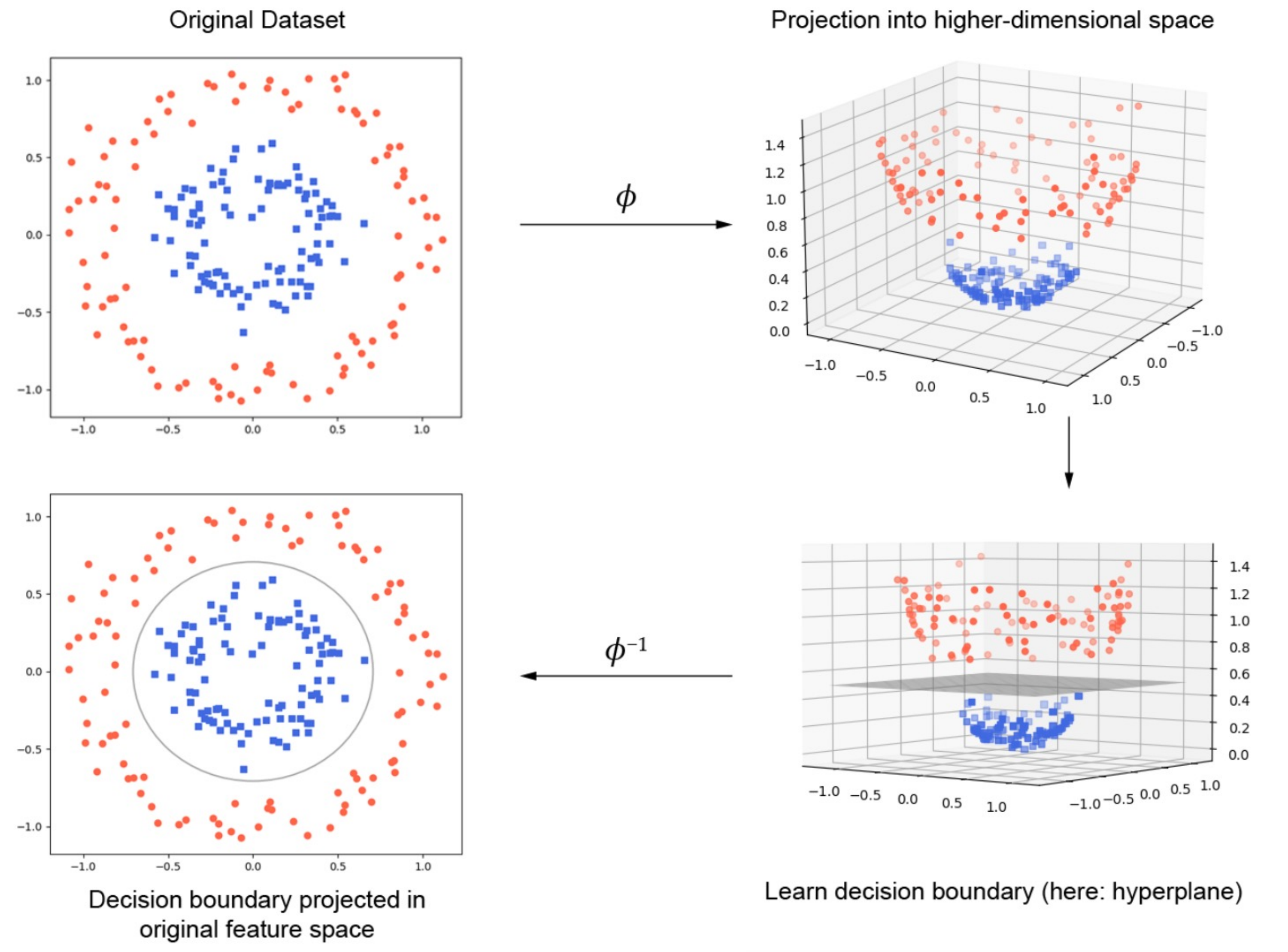


Data not linearly separable!

“XOR” dataset

Figure from Python Machine Learning (Raschka & Mirjalili)

Idea behind kernel methods

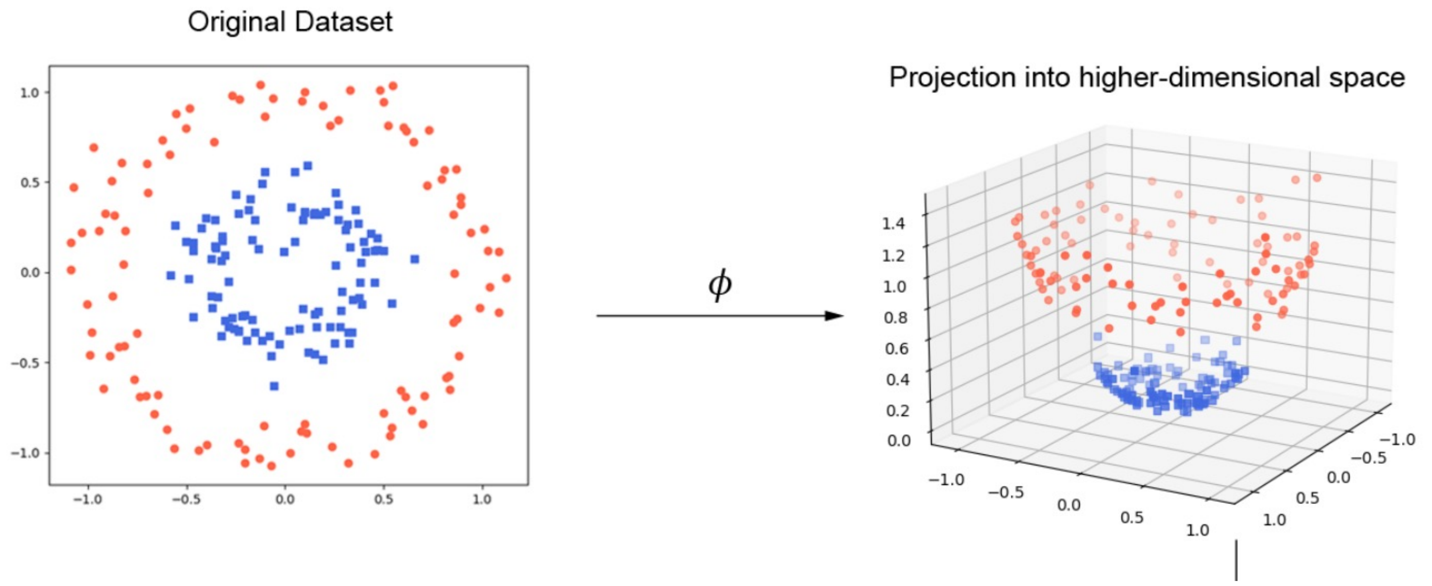


ϕ feature mapping

Data may be linearly separable in a higher dimensional feature space

Figure from Python Machine Learning (Raschka & Mirjalili)

Idea behind kernel methods



ϕ feature mapping, here:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

Figure from Python Machine Learning (Raschka & Mirjalili)

Kernel trick

sample: $\mathbf{x}^{(i)} = \begin{bmatrix} 1 & x_1^{(i)} & x_2^{(i)} & \cdots & x_m^{(i)} \end{bmatrix}$

- Many algorithms can be formulated in terms **of inner products** of the **samples**

$$\mathbf{x}^{(i)}\mathbf{x}^{(j)T} = \sum_{k=0}^{m} x_k^{(i)} x_k^{(j)}$$

- After **feature mapping**, we have an **inner product** in a **high-dimensional space**

$$\phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(j)})^T$$

- Often, there is an **easier way** to compute the resulting number in the original space

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(j)})^T$$

Kernel: $\kappa(\mathbf{x}, \mathbf{x}') : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$

without ever computing $\phi(\mathbf{x}^{(i)})$

- $\phi(\mathbf{x}^{(i)})$ is never needed. There is even kernels for which no explicit finite dimensional feature mapping exists

Kernel trick

- **Example: A quadratic polynomial kernel, m=3 (number of features)**

$$\begin{aligned}\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \left(\mathbf{x}^{(i)T}\mathbf{x}^{(j)}\right)^2 \\ &= \left(\sum_{k=1}^{m} x_k^{(i)} x_k^{(j)}\right)\left(\sum_{k=1}^{m} x_k^{(i)} x_k^{(j)}\right) \\ &= \sum_{k=1}^{m}\sum_{l=1}^{m} x_k^{(i)} x_l^{(i)} x_k^{(j)} x_l^{(j)} \\ &= \sum_{k=1}^{m}\sum_{l=1}^{m} (x_k^{(i)} x_l^{(i)})(x_k^{(j)} x_l^{(j)})\end{aligned}$$

Computation in original space via kernel: O(m)

Therefore, corresponding feature mapping is:

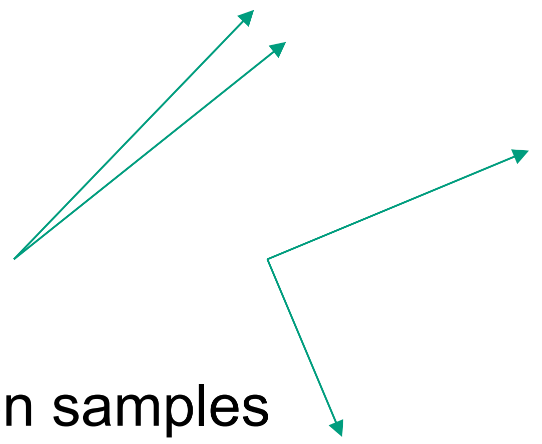
$$\phi(\mathbf{x}^{(i)}) = \left[x_1^{(i)}x_1^{(i)},\quad x_1^{(i)}x_2^{(i)},\quad x_1^{(i)}x_3^{(i)},\quad x_2^{(i)}x_1^{(i)},\quad x_2^{(i)}x_2^{(i)},\quad x_2^{(i)}x_3^{(i)},\quad x_3^{(i)}x_1^{(i)},\quad x_3^{(i)}x_2^{(i)},\quad x_3^{(i)}x_3^{(i)}\right]$$

$$\phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(j)})^T = \sum_{p=1}^{3} \phi(\mathbf{x}^{(i)})_p \phi(\mathbf{x}^{(j)})_p = \sum_{k=1}^{m=3}\sum_{l=1}^{m=3} (x_k^{(i)} x_l^{(i)})(x_k^{(j)} x_l^{(j)})$$

Computation in feature space: O(m^2)

Kernel trick

- If the data only occurs in form of **inner products** in an ML algorithm, we can replace these by evaluations of a **kernel function** to compute inner products in a high-dimensional feature space
- Evaluation with data in the original space is cheaper
- Evaluation in the feature space is expensive
- We might not even know what the feature mapping looks like!



Kernel functions

- Kernel can be seen to measure **similarity** between samples
- **“Valid”** kernel (**Mercer** kernel): if kernel matrix is **positive semi-definite**

$$K_{ij} = \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \begin{bmatrix} \kappa(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \cdots & \kappa(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}) \\ \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}) & \cdots & \kappa(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}) \end{bmatrix} \qquad \kappa(\mathbf{x}, \mathbf{x}') : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

$$\textit{a matrix } \mathbf{A} \textit{ is positive semi-definite if } \mathbf{v}^T \mathbf{A} \mathbf{v} \geq 0 \textit{ for all non-zero } \mathbf{v}$$

- For valid kernels, it is guaranteed that a corresponding feature mapping exists (although, it might be infinite dimensional)

Typical kernel functions

- Linear kernel

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})\phi(\mathbf{x}^{(j)})^T = \mathbf{x}^{(i)}\mathbf{x}^{(j)^T}$$

- Polynomial kernel (tuning parameter p)

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)}\mathbf{x}^{(j)^T} + 1)^p, \quad \textit{for polynomial degree } p$$

- Isotropic Gaussian kernel / Radial basis function (RBF) kernel (tuning parameter γ)

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2^2}{2\sigma^2}\right) := \exp\left(-\gamma||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2^2\right)$$

More kernel functions (not syllabus)

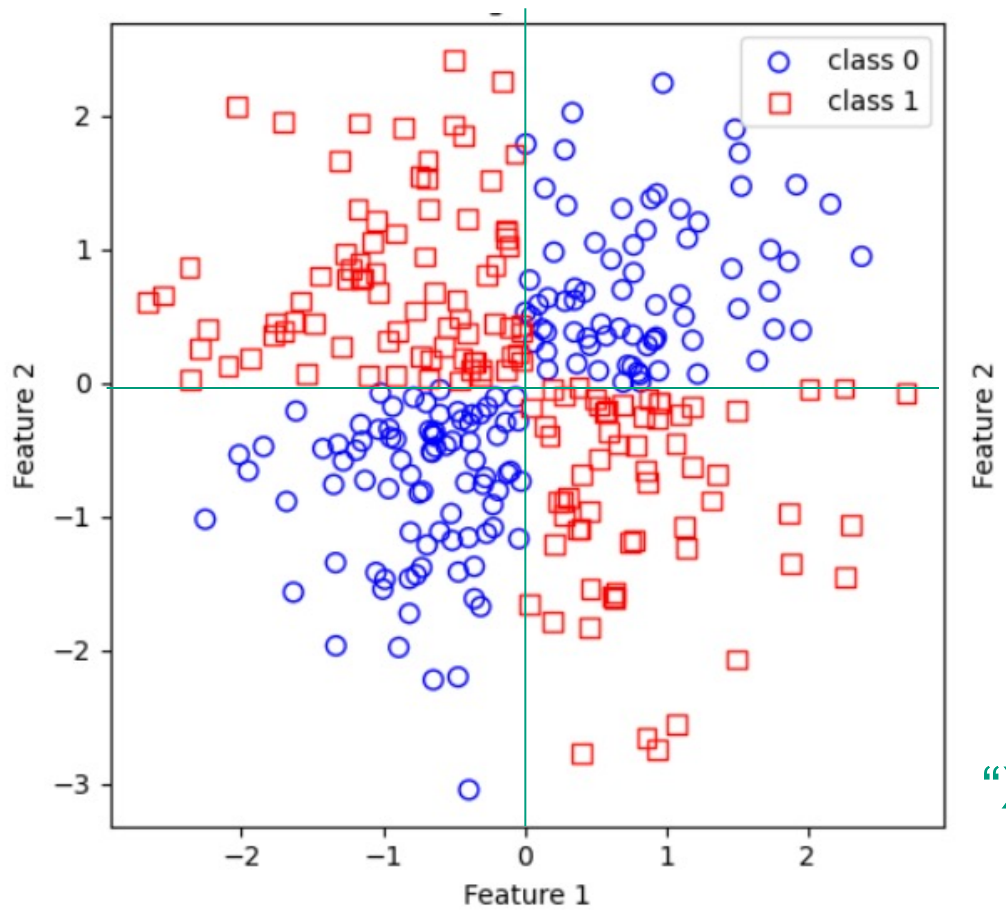
- There is also kernel functions for strings, text, etc.
- Kernelized algorithm can then operator directly on that data without explicit transformation
- Choice of **kernel function** should be **tailored to the data**

Kernel SVM

- In prediction and optimization algorithm (we didn’t look at it in detail) data only occurs as **inner products** → **replace inner products** by **kernel function** calls

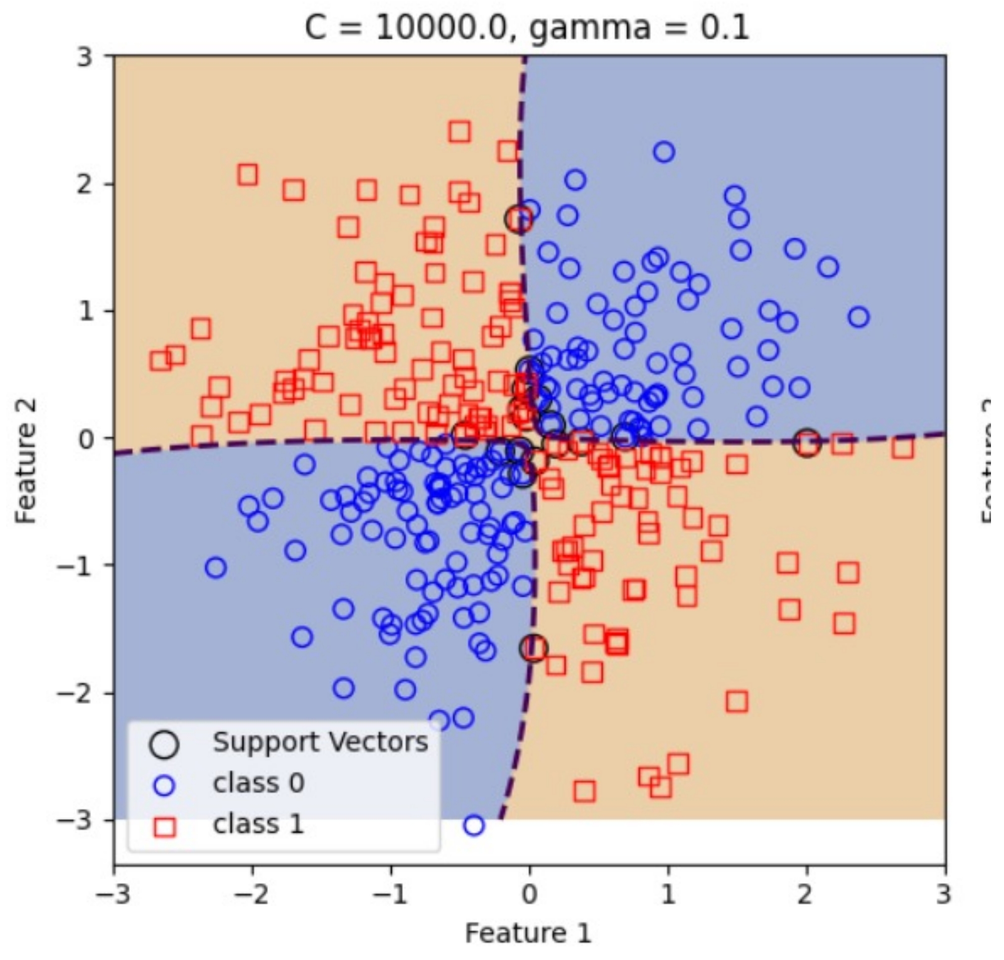
- In scikit-learn the type of kernel is a parameter of the SVC (Support Vector Classifier)
- The kernels have hyperparameters (e.g. `degree` for polynomial kernel)

Kernel SVM Code example



“XOR” dataset

Kernel SVM Code example (RBF kernel)



Chosen kernel function:

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) := \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|_2^2\right)$$

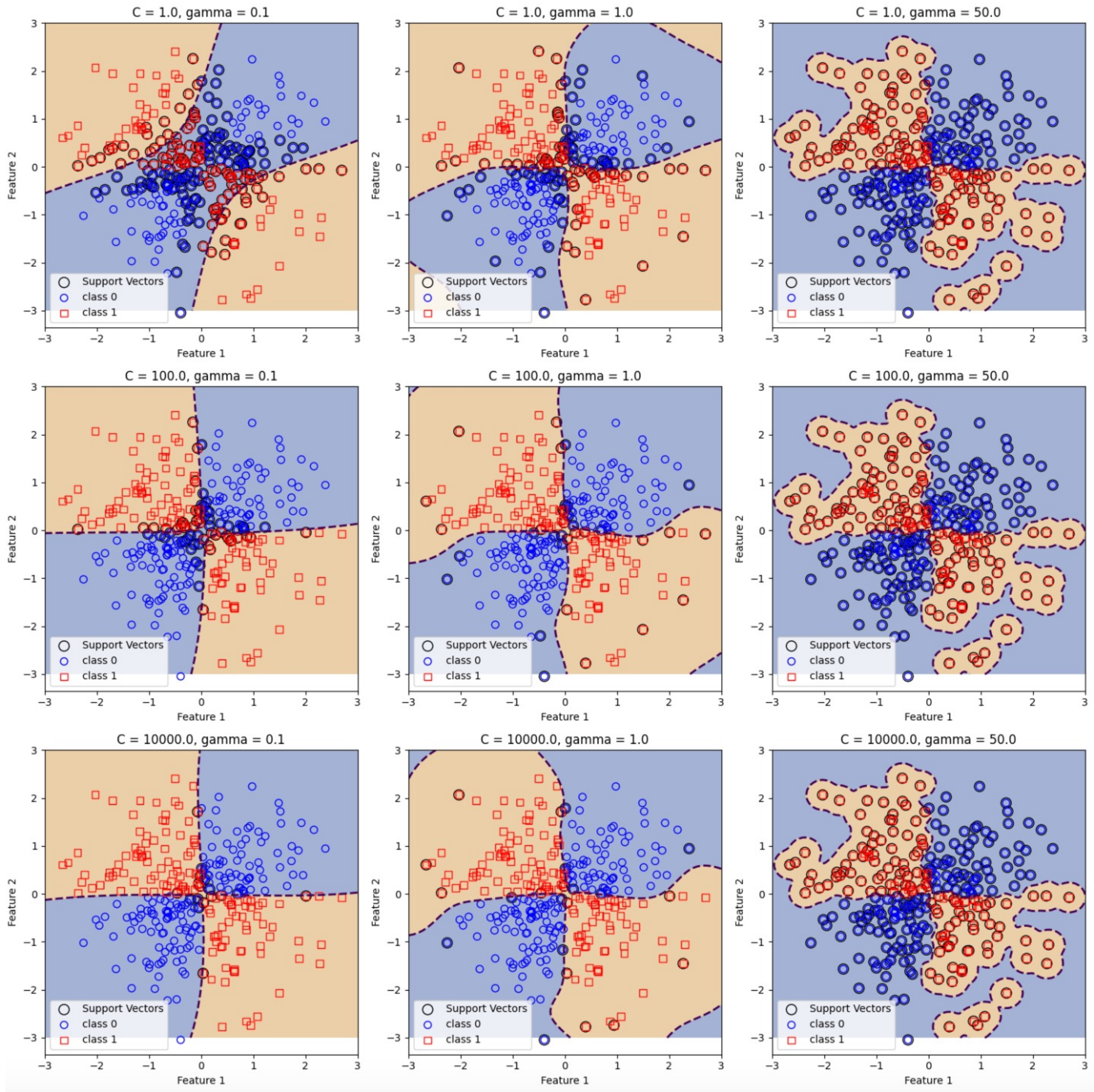
`03_svm_kernel_xor.ipynb`

Kernel SVM

- We need to tune both C and the kernel parameter(s)
- For example: grid search
  - C in [1.0, 100.0, 10000.0]
  - γ in [0.1, 1.0, 50.0]

`03_svm_kernel_xor.ipynb`

`03_svm_gridsearch_iris.ipynb`

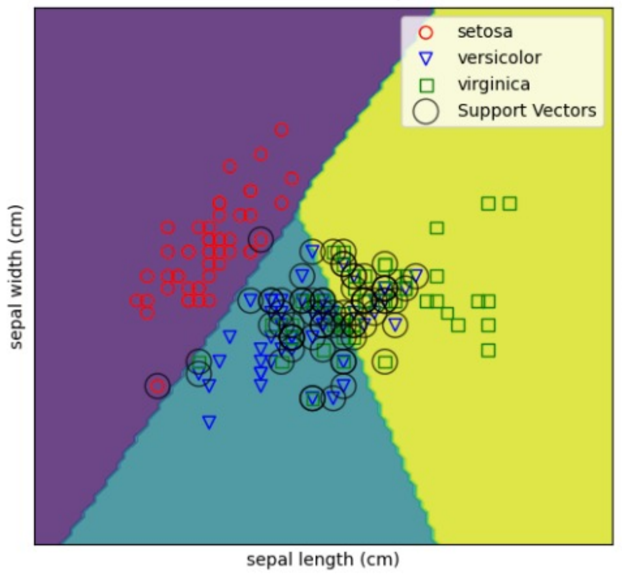


Kernel SVM

- We can also try different kernels
- For example:
  - Linear
  - Polynomial
  - RBF

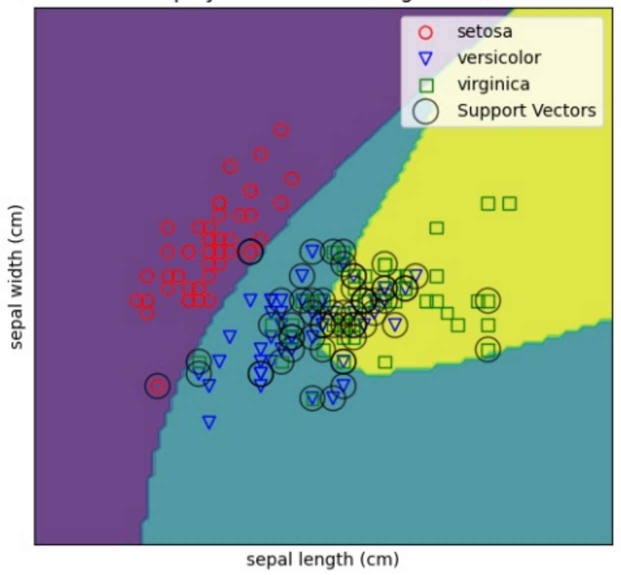
$\mathbf{x}^{(i)}\mathbf{x}^{(j)T}$

SVC with linear kernel, C=100



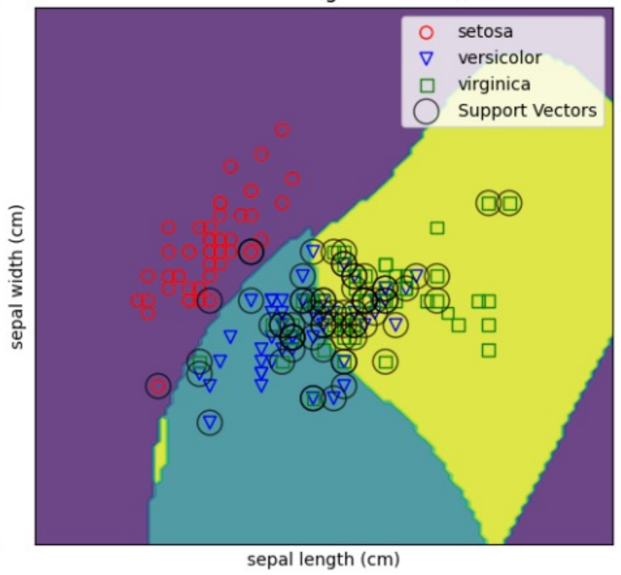
$(\mathbf{x}^{(i)}\mathbf{x}^{(j)T} + 1)^p$

SVC with polynomial kernel (degree=5), C=100



$\exp\left(-\gamma||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2^2\right)$

SVC with RBF kernel (gamma=0.3), C=100



`03_svm_different_kernels_iris.ipynb`

Other kernelized ML algorithms

Nonlinear decision boundaries

Kernelized ML algorithms

- Many of the presented algorithms can be **kernelized** by replacing inner products with kernel evaluations (sometimes after reformulation of the original algorithm)
- Examples:
  - **Kernel Perceptron**
  - **Kernel Logistic Regression**
  - **Kernel Ridge Regression** (linear regression + L2 regularizer)
  - **Kernel SVM**
  - **Kernel k-nearest neighbors** (next lecture)

Nonlinear decision boundaries

Kernel perceptron

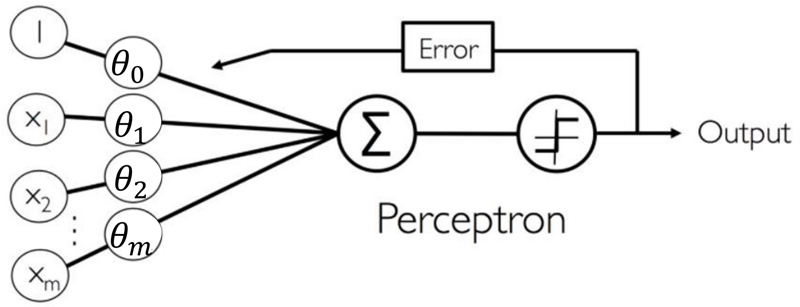
Kernel perceptron

- To gain intuition of how to kernelize linear models
- Recall that for the perceptron $y^{(i)} \in \{1, -1\}$
  - we **predict** as

$$\hat{y}^{(i)} = \text{sign}\{\mathbf{x}^{(i)}\boldsymbol{\theta}\}$$

  - we **train** as

```
$\boldsymbol{\theta} \leftarrow \mathbf{0}$
for $i = 1, \cdots, n$ do
    $\hat{y}^{(i)} \leftarrow \text{sign}\{\mathbf{x}^{(i)}\boldsymbol{\theta}\}$
    if $\hat{y} \neq y^{(i)}$ then
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta y^{(i)} \mathbf{x}^{(i)T}$
    end if
end for
```



this actually means: $\boldsymbol{\theta} = \sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} \mathbf{x}^{(j)T}$

$\alpha^{(j)}$: number of times sample j is misclassified

Figure from Python Machine Learning (Raschka & Mirjalili)

Kernel perceptron

- Parameters are linear combination of the samples: $\boldsymbol{\theta} = \sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} \mathbf{x}^{(j)T}$
- Recall that for the perceptron $y^{(i)} \in \{1, -1\}$
  - we **predict** as

$$\hat{y}^{(i)} = \text{sign}\{\mathbf{x}^{(i)}\boldsymbol{\theta}\} \longrightarrow \hat{y}^{(i)} = \text{sign}\left\{\sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} \mathbf{x}^{(i)} \mathbf{x}^{(j)T}\right\}$$

“dual” formulation

inner product

  - we **train** as

```
$\boldsymbol{\theta} \leftarrow \mathbf{0}$
for $i = 1, \cdots, n$ do
    $\hat{y}^{(i)} \leftarrow \text{sign}\{\mathbf{x}^{(i)}\boldsymbol{\theta}\}$
    if $\hat{y} \neq y^{(i)}$ then
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta y^{(i)} \mathbf{x}^{(i)T}$
    end if
end for
```

$\longrightarrow$

```
$\boldsymbol{\alpha} \leftarrow \mathbf{0}$
for $i = 1, \cdots, n$ do
    $\hat{y}^{(i)} \leftarrow \text{sign}\left\{\sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} \mathbf{x}^{(i)} \mathbf{x}^{(j)T}\right\}$
    if $\hat{y} \neq y^{(i)}$ then
        $\alpha^{(i)} \leftarrow \alpha^{(i)} + 1$
    end if
end for
```

Kernel perceptron

- Parameters are linear combination of the samples: $\boldsymbol{\theta} = \sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} {\mathbf{x}^{(j)}}^T$
- Dual formulation $y^{(i)} \in \{1, -1\}$
  - we **predict** as

$$\hat{y}^{(i)} = \text{sign}\left\{ \sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} \mathbf{x}^{(i)} {\mathbf{x}^{(j)}}^T \right\}$$

  - we **train** as

```
$\boldsymbol{\alpha} \leftarrow \mathbf{0}$
**for** $i = 1, \cdots, n$ **do**
    $\hat{y}^{(i)} \leftarrow \text{sign}\left\{ \sum_{j=1}^{n} \alpha^{(j)} \eta y^{(j)} \mathbf{x}^{(i)} {\mathbf{x}^{(j)}}^T \right\}$
    **if** $\hat{y} \neq y^{(i)}$ **then**
        $\alpha^{(i)} \leftarrow \alpha^{(i)} + 1$
    **end if**
**end for**
```

Implementation from scratch in Python:

`03_kernel_perceptron.ipynb`

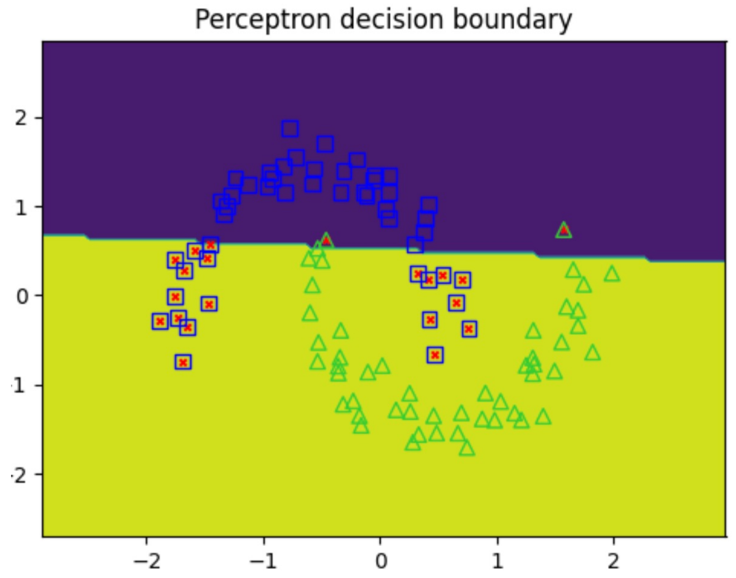
Kernel perceptron

- Kernel perceptron
  - we **predict** as

$$\hat{y}^{(i)} = \text{sign}\left\{\sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})\right\}$$

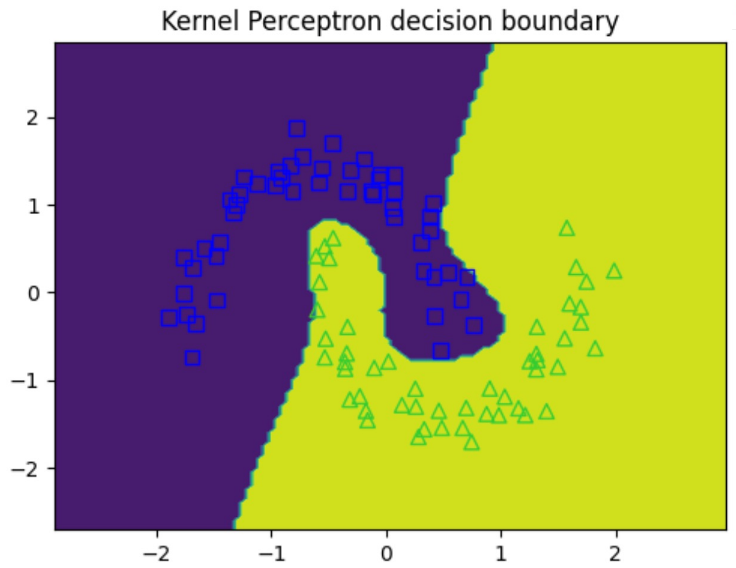
  - we **train** as

```
$\boldsymbol{\alpha} \leftarrow \mathbf{0}$
for $i = 1, \cdots, n$ do
    $\hat{y}^{(i)} \leftarrow \text{sign}\left\{\sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})\right\}$
    if $\hat{y} \neq y^{(i)}$ then
        $\alpha^{(i)} \leftarrow \alpha^{(i)} + 1$
    end if
end for
```



(*p*=1)

$$(\mathbf{x}^{(i)}\mathbf{x}^{(j)^T} + 1)^p$$



p=5

Nonlinear decision boundaries

Kernel logistic regression (extra, not syllabus)

Kernel logistic regression

- Logistic regression can be **kernelized** analogously to the perceptron by switching to a dual formulation

- Generalized linear models (like logistic regression) can be turned into **kernel machines** by considering a feature mapping of the form:

$$\boxed{\phi(\mathbf{x}^{(i)}) = [\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}), \ldots, \kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(n)})]}$$

Number of features is number of samples (or somewhat less for sparse kernel machines)

- This corresponds to the realization that the weights can be written as a linear combination of the samples

- When only a subset of the samples is used (**support vectors**), we call the resulting model a **sparse kernel machine**

Kernel logistic regression versus Kernel SVM

- **Kernel SVMs** yields **sparse** kernel machines which are faster in prediction and have lower memory consumption
- **Kernelized Logistic regression** uses all data points in kernel matrix, resulting in a dense kernel machine
- **Kernelized Logistic regression** with sparsity-promoting regularization like **L1-regularization** yields a **sparse** predictor (samples with zero weights can be dropped)