# 16. Support Vector Machine

M.A.Z. Chowdhury and M.A. Oehlschlaeger

Department of Mechanical, Aerospace and Nuclear Engineering
Rensselaer Polytechnic Institute
Troy, New York

*chowdm@rpi.edu and oehlsm@rpi.edu*

MANE 4962 and 6962

# Regular announcement

- HW 5 is due March 20, 2023
- Quiz 5 on March 20, 2023
- Office hour on Friday, March 17, 3-5 PM
- HW 5 discussion

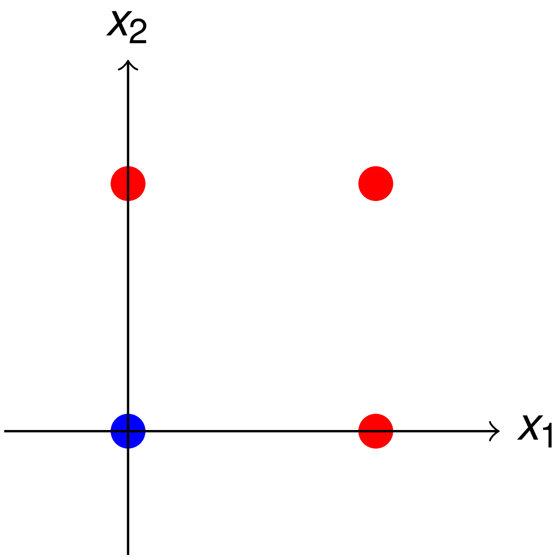
→ Quiz 5 is based on Lectures 14 & 15

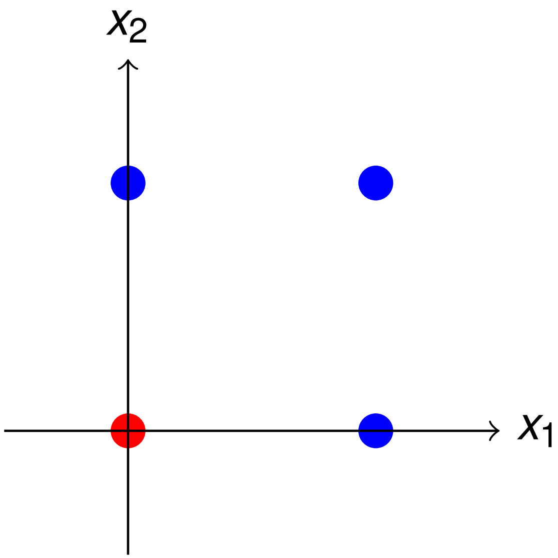
# OR Gate and NOR Gate

| input $\underline{x} = (x_1, x_2)$ | output y | prediction |
|---|---|---|
| (0,0) | 0 | $< 0.5$ |
| (0,1) | 1 | $\geq 0.5$ |
| (1,0) | 1 | $\geq 0.5$ |
| (1,1) | 1 | $\geq 0.5$ |

OR Gate: $w_0 = -1$ $w_1 = 2$ $w_2 = 2$

red-1, blue-0




| input $\underline{x} = (x_1, x_2)$ | output y | prediction |
|---|---|---|
| (0,0) | 1 | $\geq 0.5$ |
| (0,1) | 0 | $< 0.5$ |
| (1,0) | 0 | $< 0.5$ |
| (1,1) | 0 | $< 0.5$ |

NOR Gate: $w_0 = 1$ $w_1 = -2$ $w_2 = -2$

Note: $w$'s are not guaranteed to be unique

# AND Gate and NAND Gate

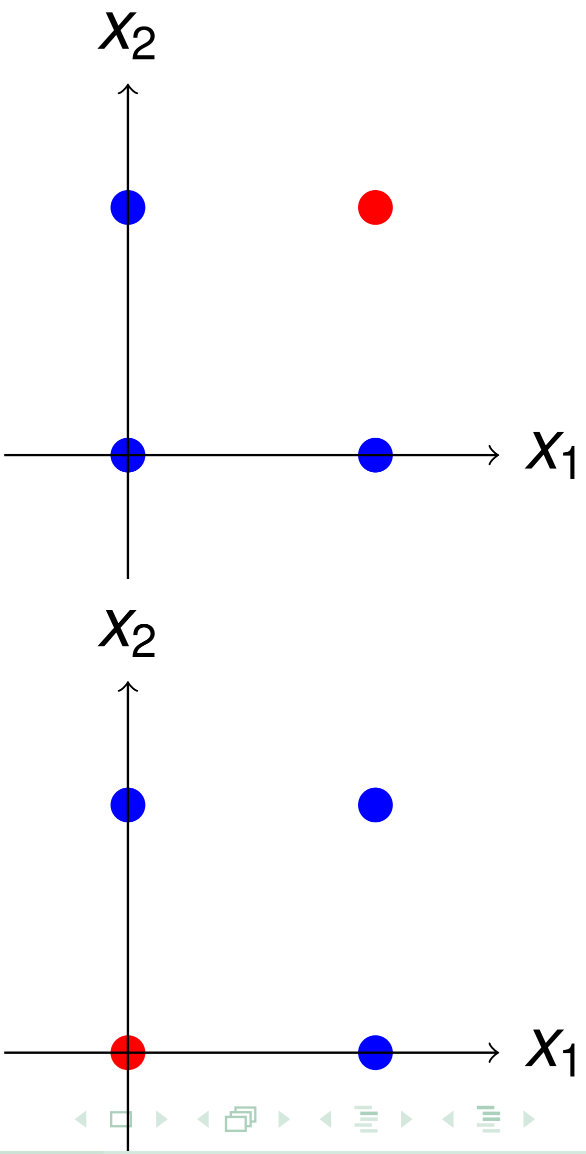
| input $\underline{x} = (x_1, x_2)$ | output y | prediction |
|---|---|---|
| (0,0) | 0 | $< 0.5$ |
| (0,1) | 0 | $< 0.5$ |
| (1,0) | 0 | $< 0.5$ |
| (1,1) | 1 | $\geq 0.5$ |

AND Gate: $w_0 = -3$ $w_1 = 2$ $w_2 = 2$

| input $\underline{x} = (x_1, x_2)$ | output y | prediction |
|---|---|---|
| (0,0) | 1 | $\geq 0.5$ |
| (0,1) | 0 | $< 0.5$ |
| (1,0) | 0 | $< 0.5$ |
| (1,1) | 0 | $< 0.5$ |

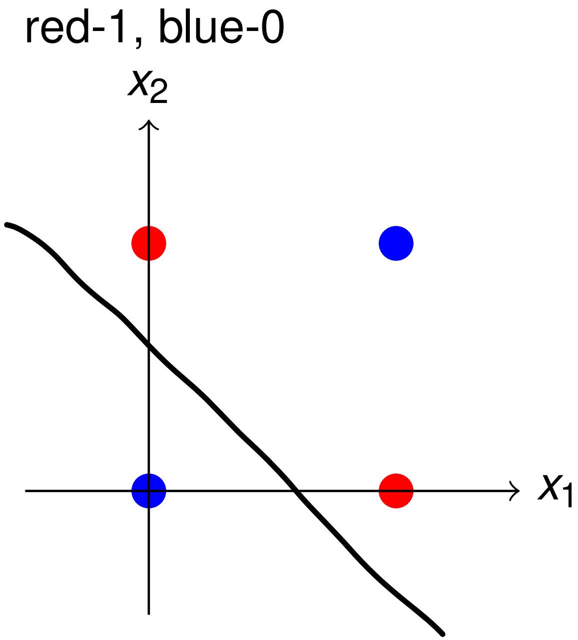
NAND Gate: $w_0 = 3$ $w_1 = -2$ $w_2 = -2$
Note: $w$'s are not guaranteed to be unique

red-1, blue-0



# XOR Gate

| input $\underline{x} = (x_1, x_2)$ | output y | prediction |
|---|---|---|
| (0,0) | 0 | × |
| (0,1) | 1 | × |
| (1,0) | 1 | × |
| (1,1) | 0 | × |




XOR Gate: No solution from logistic regression!

- Use non-linear feature transformation. (The kernel trick) SVMs.
- Add extra layers. (Deep neural nets)

# Support Vector Machines

A powerful and versatile machine learning model first proposed by Vladimir Vapnik. Check the very first notebook where we used SVM blindly to classify iris dataset which achieved very high classification accuracy even without data pre-processing.

- Linear or nonlinear classification
- Regression
- Outlier detection

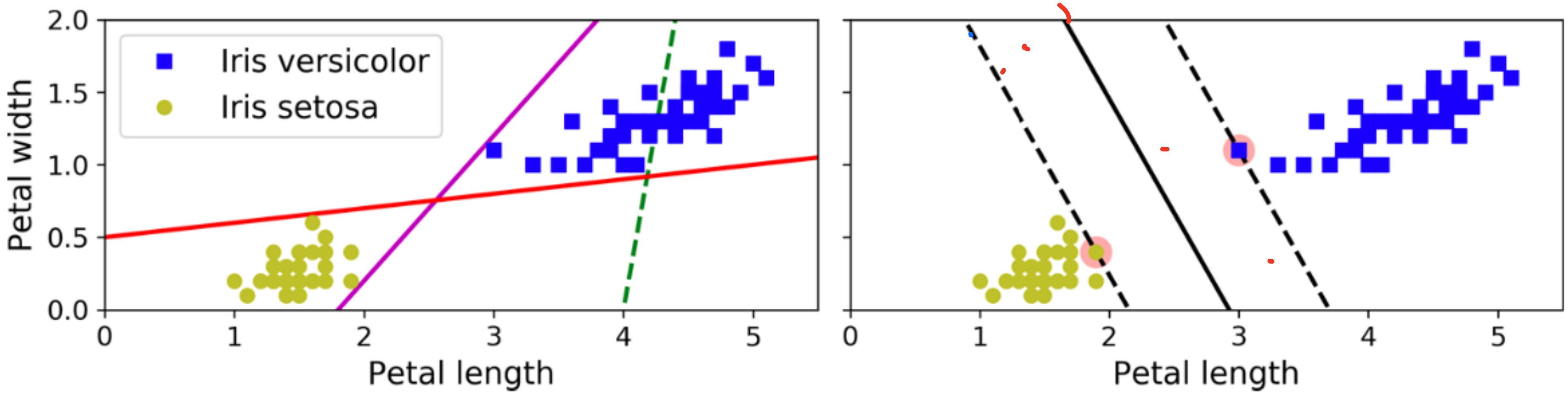
1958
Perceptron Rosenblatt
10 yr. Minsky & Papert

On solving XOR Gate

```python
import numpy as np
from sklearn import svm
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
y = np.array([0, 1, 1, 0])
model = svm.SVC(kernel='rbf', C=1, gamma='auto')
model.fit(X, y)
predictions = model.predict(X)
print("Predictions: ", predictions) # should classify all the data points correctly
```

1968-1980
MNIST digits classified SVM

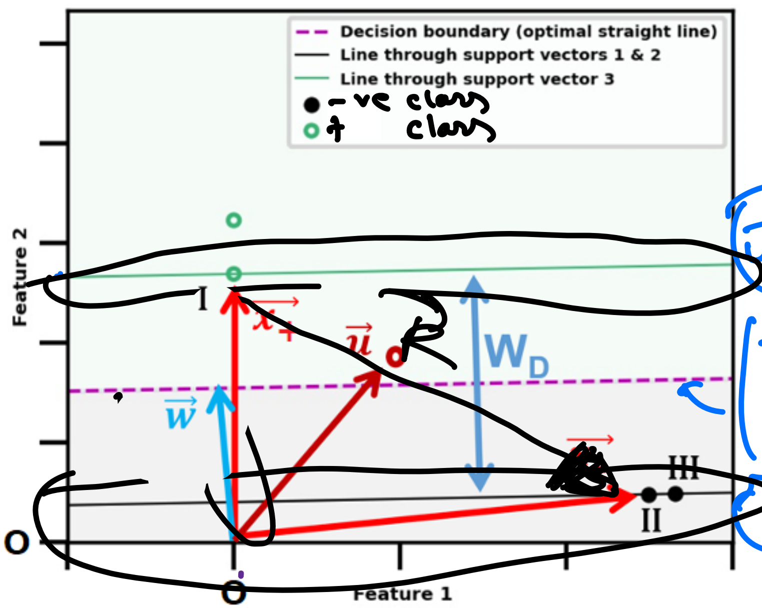
# Linear Support Vector Machine Classifier




Adding more training data points within the margins of the classes will not change the classifier. The marginal data points are known as support vectors.

# Support Vector Machine

1964




$\vec{w}\cdot\vec{x}+b=+1$

$\vec{w}\cdot\vec{x}+b = 0$

$\vec{w}\cdot\vec{x}+b=-1$

**Constrained optimization problem**

$L = \frac{1}{2}||\vec{w}||^2 - \sum_{i=1}^{m} \alpha_i[y_i(\vec{w}\cdot\vec{x_i}+b)-1]$
(Primal form)

$= 0$

$\frac{\partial L}{\partial \vec{w}} = 0$ , $\frac{\partial L}{\partial b} = 0$

$\vec{w} = \square$
$b = \square$

$L = \sum_{i=1}^{m}\alpha_i - \sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \Phi(x_i, x_j)$
(Dual form)

SVM Dual form

196

kernel function

solved under the constraints via the Lagrange multipliers

$\alpha_i \geq 0$ and $\alpha_i \leq C$

$C = 1000$

Larger values of $C$ reduce misclassifications.

**Decision rule**

$\vec{w}\cdot\vec{x_+} + b \geq 1, \quad y_i = +1$

$\vec{w}\cdot\vec{x_-} + b \leq -1, \quad y_i = -1$

**Width maximization**

$W_D = (\vec{x_+} - \vec{x_-})\cdot\frac{\vec{w}}{||\vec{w}||} = \frac{2}{||\vec{w}||}$

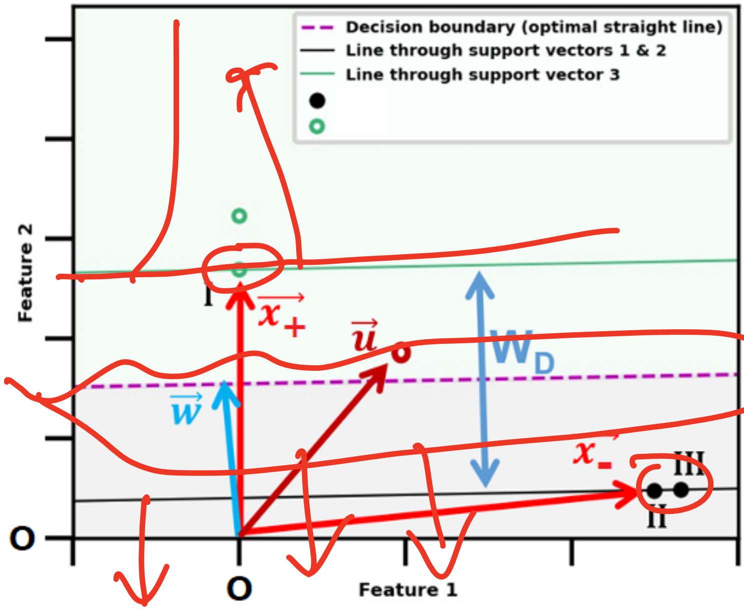
$max(\frac{2}{||\vec{w}||}) \rightarrow min(||\vec{w}||) \rightarrow min(\frac{1}{2}||\vec{w}||^2)$

Max<sup></sup>m margin classifier

$\vec{x}_+ + \vec{R} = \vec{x}_- \Rightarrow \vec{R} = \vec{x}_+ - \vec{x}_-$

$\vec{R} \cdot \frac{\vec{w}}{||\vec{w}||} = WD$

# Support Vector Machine




**Decision rule**

$\vec{w} \cdot \vec{x_+} + b \geq 1, \quad y_i = +1$

$\vec{w} \cdot \vec{x_-} + b \leq -1, \quad y_i = -1$

**Width maximization**

$W_D = (\vec{x_+} - \vec{x_-}) \cdot \frac{\vec{w}}{||\vec{w}||} = \frac{2}{||\vec{w}||}$

$max(\frac{2}{||\vec{w}||}) \rightarrow min(||\vec{w}||) \rightarrow min(\frac{1}{2}||\vec{w}||^2)$

**Contd.**

**Kernel functions ($\Phi(x_i, x_j)$)**

$\Phi(x_i, x_j) = x_i \cdot x_j$ (linear)

$\Phi(x_i, x_j) = e^{-\frac{||x_i - x_j||^2}{2\sigma^2}} = e^{-\gamma||x_i - x_j||^2}$ (radial basis function)

$\Phi(x_i, x_j) = (x_i \cdot x_j + k)^d$ (polynomial)

$\sigma$ relates sensitivity to variance in the feature vectors

$\gamma = \frac{1}{2\sigma^2} \geq 0$

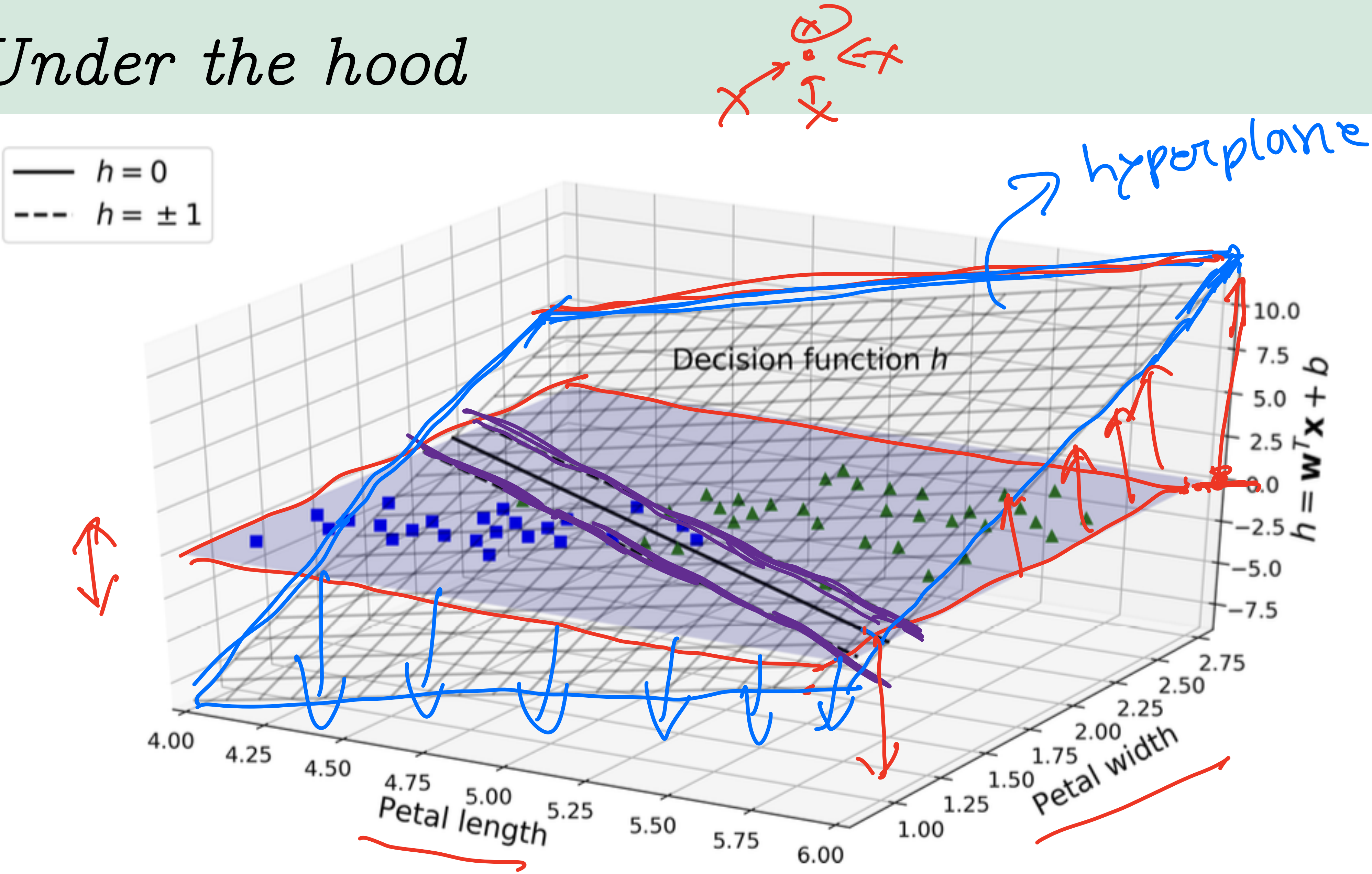
**Final solution**

$\vec{w} = \sum_{i=1}^{m} \alpha_i y_i \vec{u}$

$b = \frac{1}{N_S} \sum_{i \in y_i} y_i - \sum_{j \in y_i} \alpha_j y_i (x_i \cdot x_j)$

$N_S$ = Number of SV.

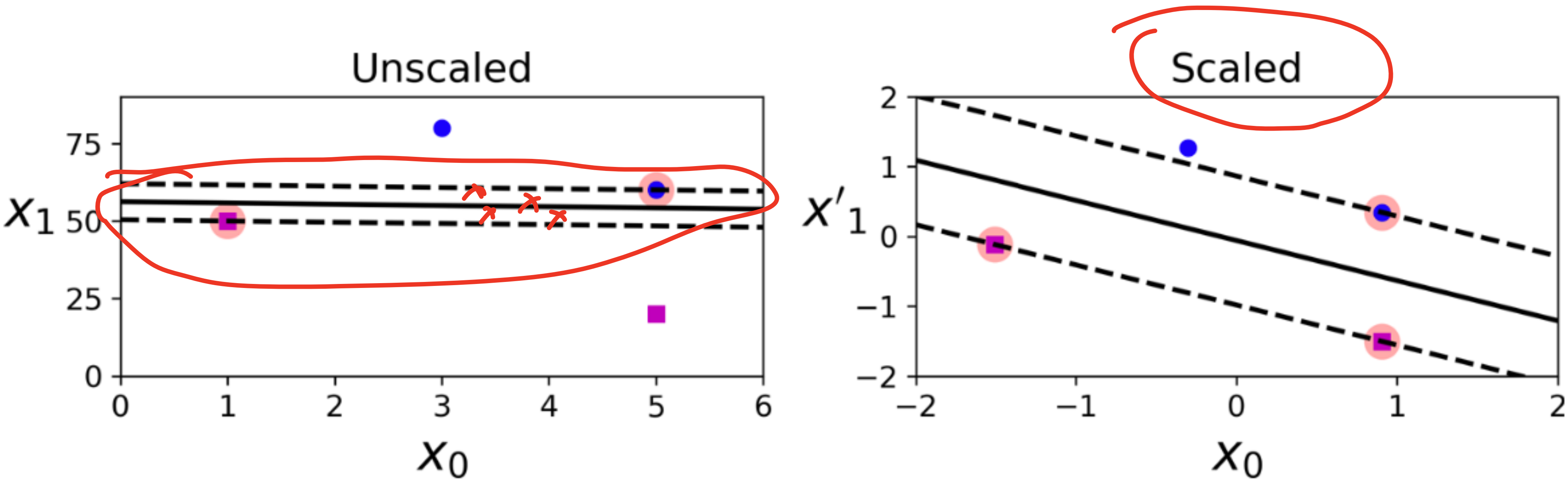
# Under the hood




SVM + PCA (Dimensionality)

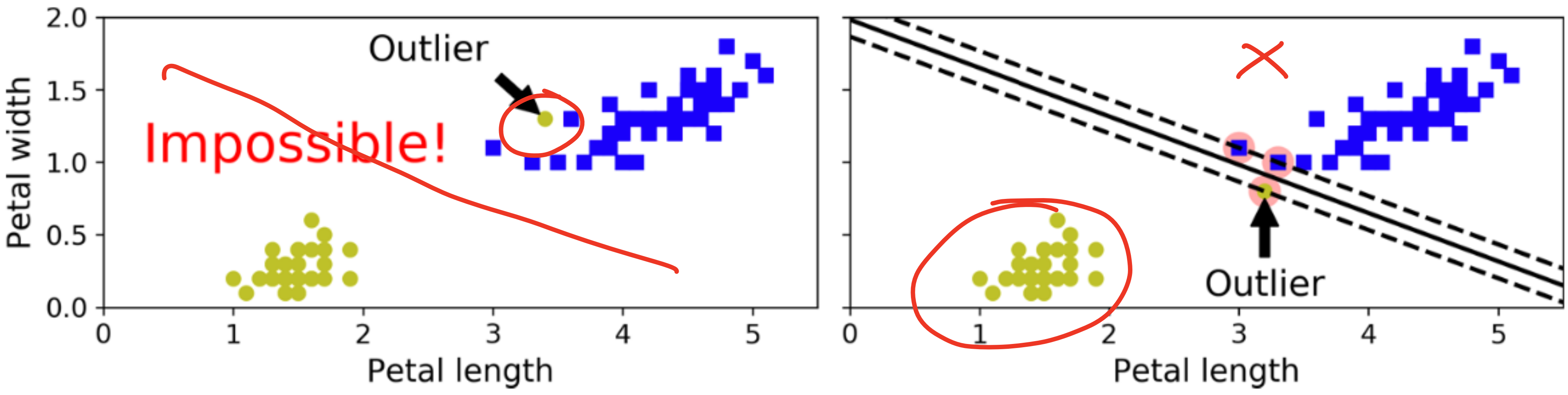
Aurélien Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

# Feature scaling

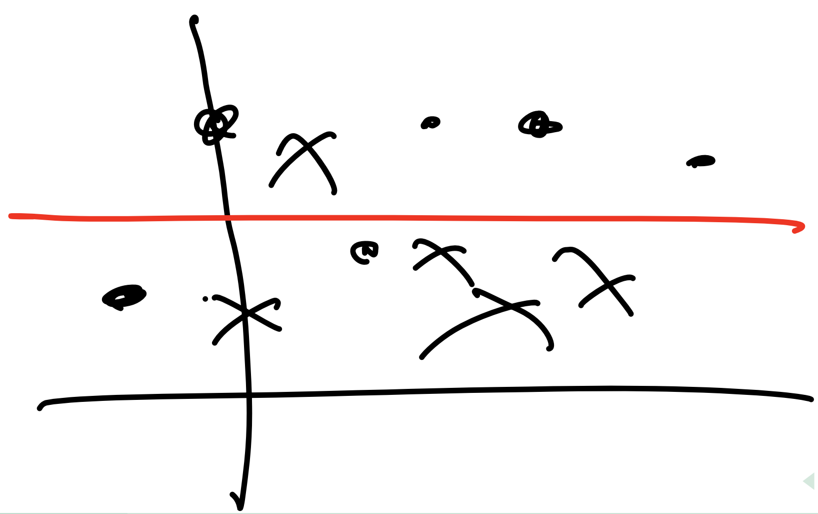


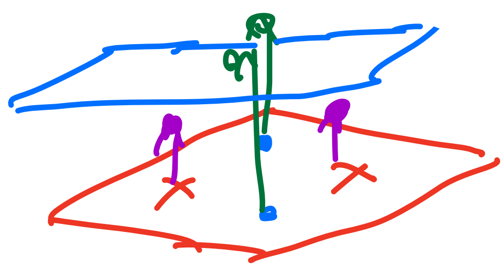

Feature scaling will help widen the gap between the margins.

# Hard margin vs Soft margin

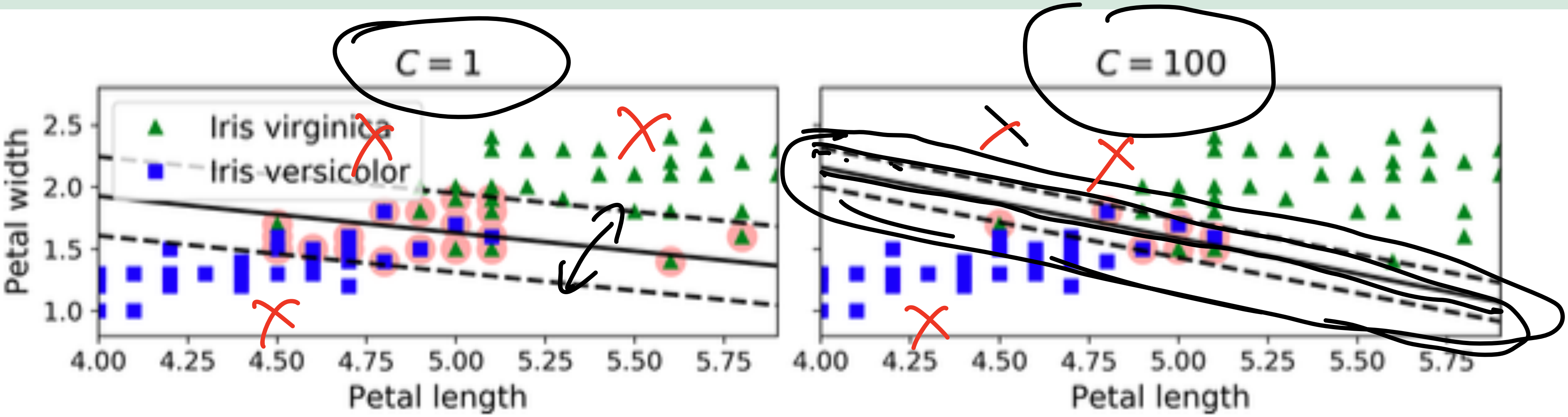



Hard margin works only for linearly separable data. It strictly separates the two classes.
Soft margin allows misclassification on either side of the margin.





# Hard margin vs Soft margin

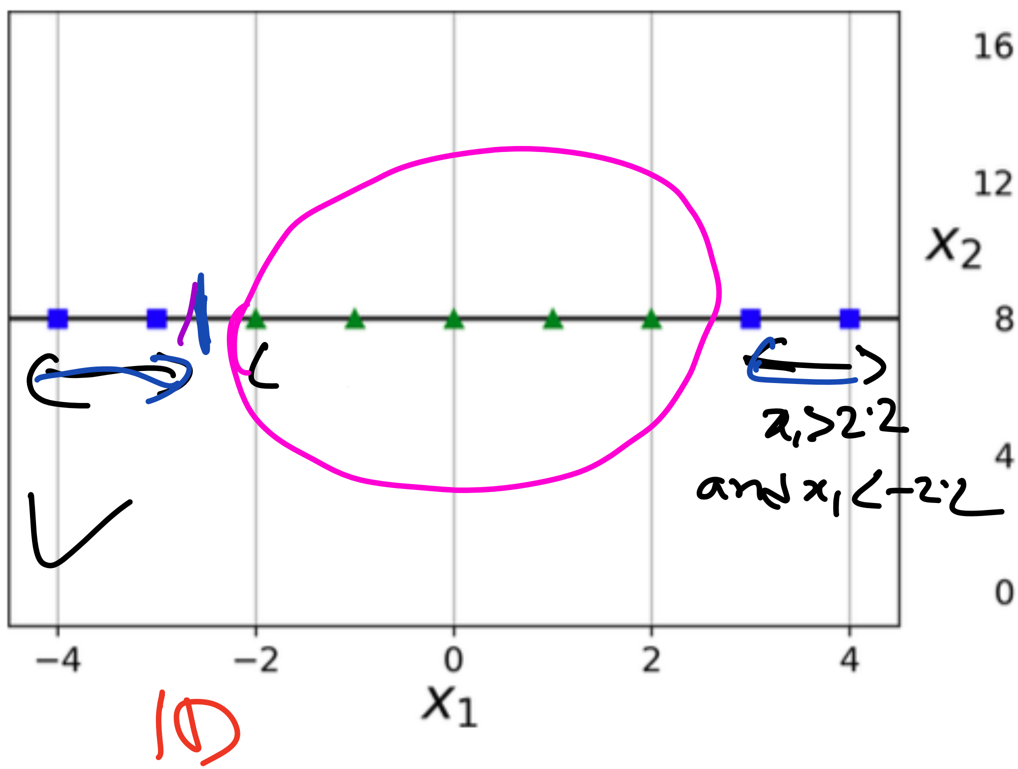


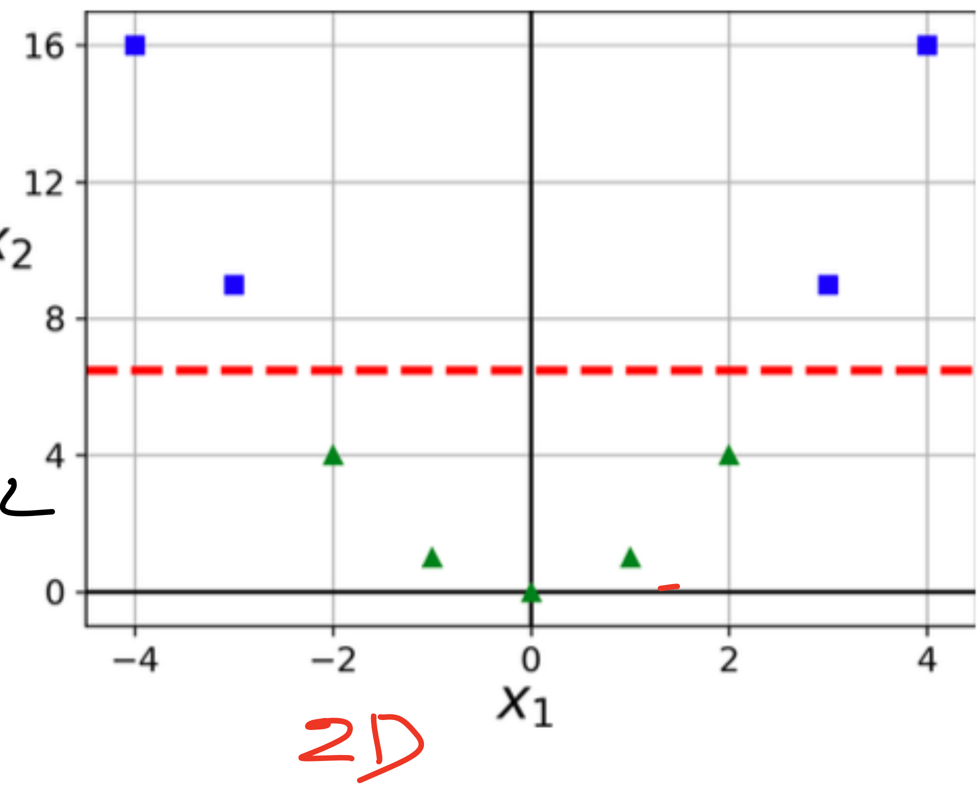

Margin violations are not desirable and thus minimized. Low C models generalize better. C is a regularization hyperparamter, called soft margin constant.

SVM C → Reduce

High C → Overfit

# Non-linear SVM: Adding polynomial features

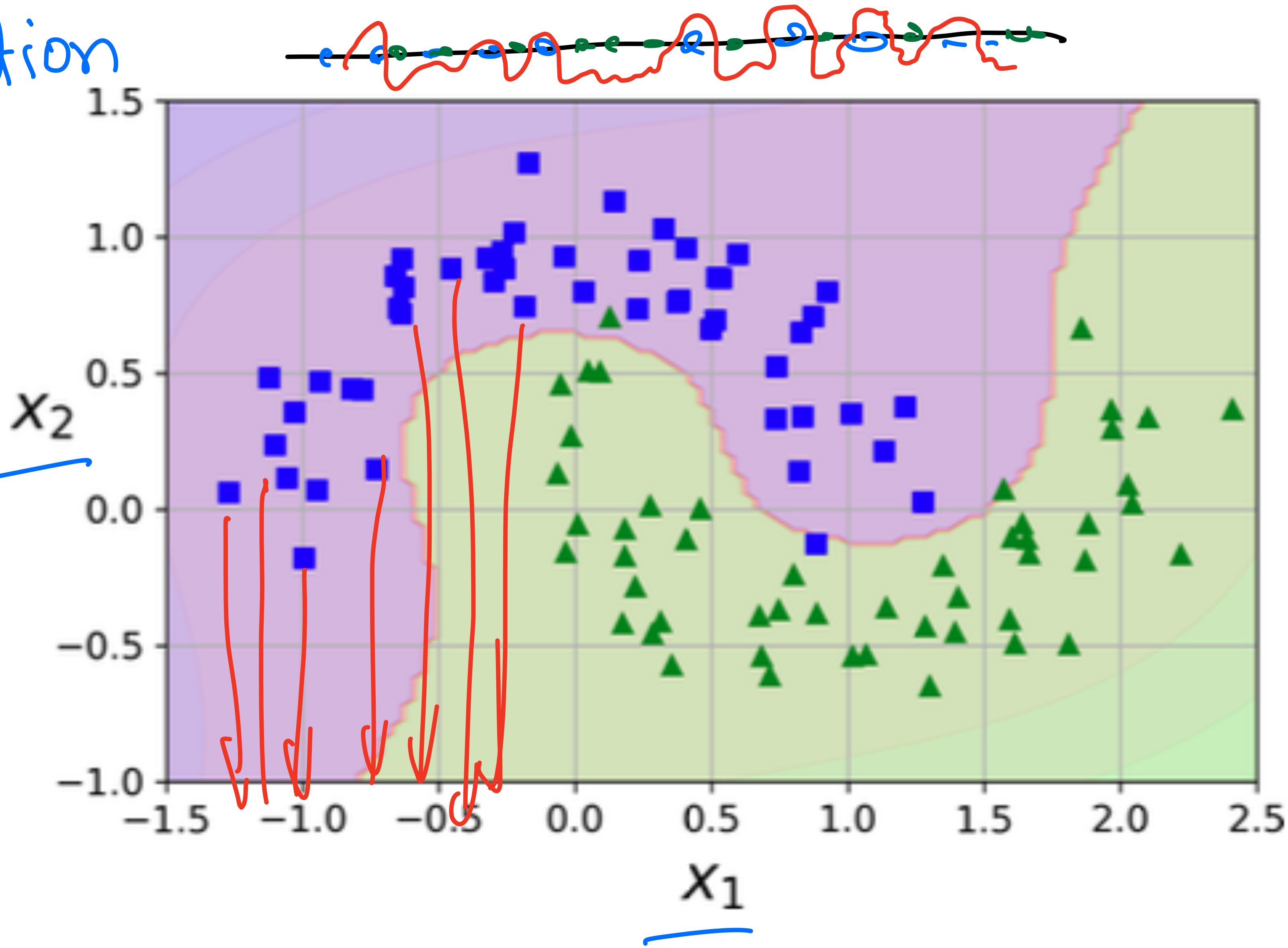



May make a problem linearly seperable $x_2 = x_1^2$

Kernel → transform input features to high-D space

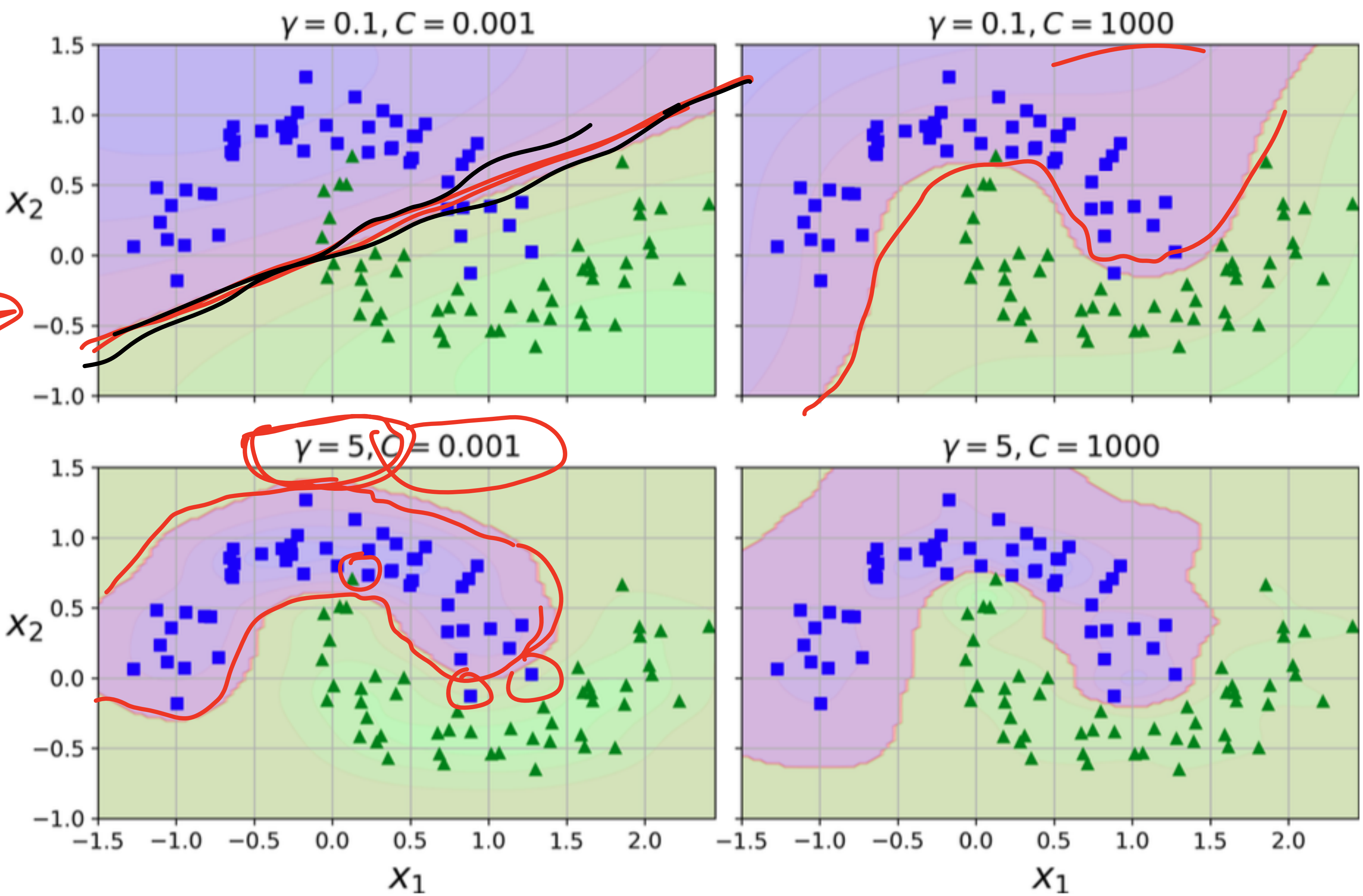
Ablation




Imagine classifying this dataset with just $x_1$ or $x_2$ alone.

# Radial Basis Function Kernel

$\rightarrow \phi(x_i, x_j)$




Aurélien Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

# Heuristics for choosing the right kernel

Most commonly used kernels are linear, polynomial and radial basis function (Gaussian).

- Always try the linear kernel first, especially if the training set is very large or if it has plenty of features linear kernel will be faster.
- If the training set is not too large, you should also try the Gaussian RBF kernel; it works well in most cases.
- Experiment with other kernels, using cross-validation and grid search if additional computing resources and project time is available.

RBF

# Scikit's implementation of SVM

Make sure to normalize the data. StandardScaler() is convenient for normalization. LinearSVC is faster but you must specify, the loss function to be hinge

```python
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
clf = make_pipeline(StandardScaler(), SVC(gamma='auto'))
clf.fit(X, y)
```

Linear, RBF, Polynomi

```python
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()

X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris virginica
  svm_clf = Pipeline([
          ("scaler", StandardScaler()),
          ("linear_svc", LinearSVC(C=1, loss="hinge")),
      ])
svm_clf.fit(X, y)
```

Linear

# Hinge loss

$y = -1$

$y = +1$

$+1 \quad \hat{y} = -1$

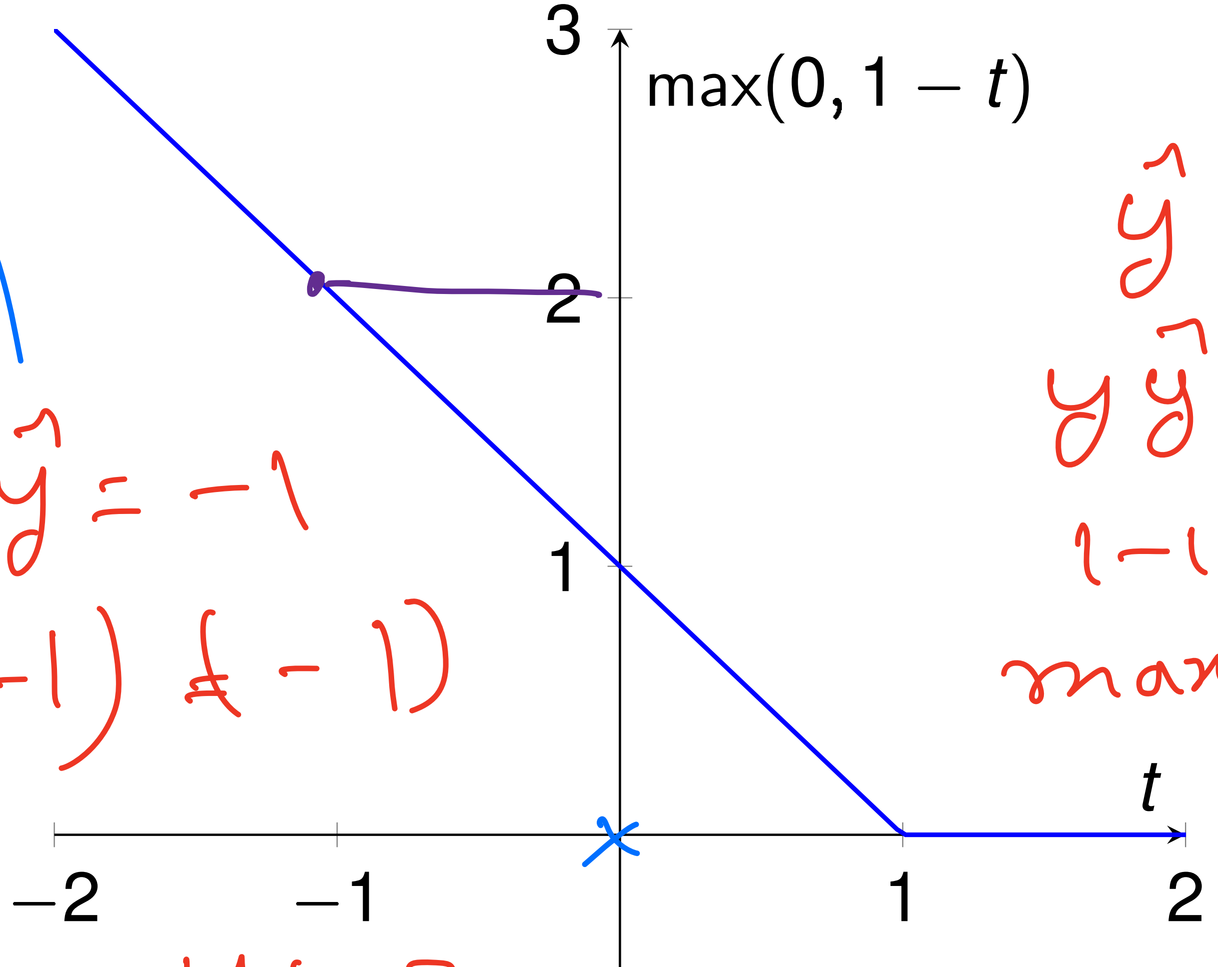
$(1)(-1) = -1$

$\hat{y} = +1$

$y\hat{y} = 1$

$1 - 1 = 0$

$\max(0, 0) = 0$




$1 - (-1) = 1 + 1 = 2$

$\max(0, 2) = 2$

$$L = max(0, 1 - y\hat{y}) \quad y = \pm 1$$

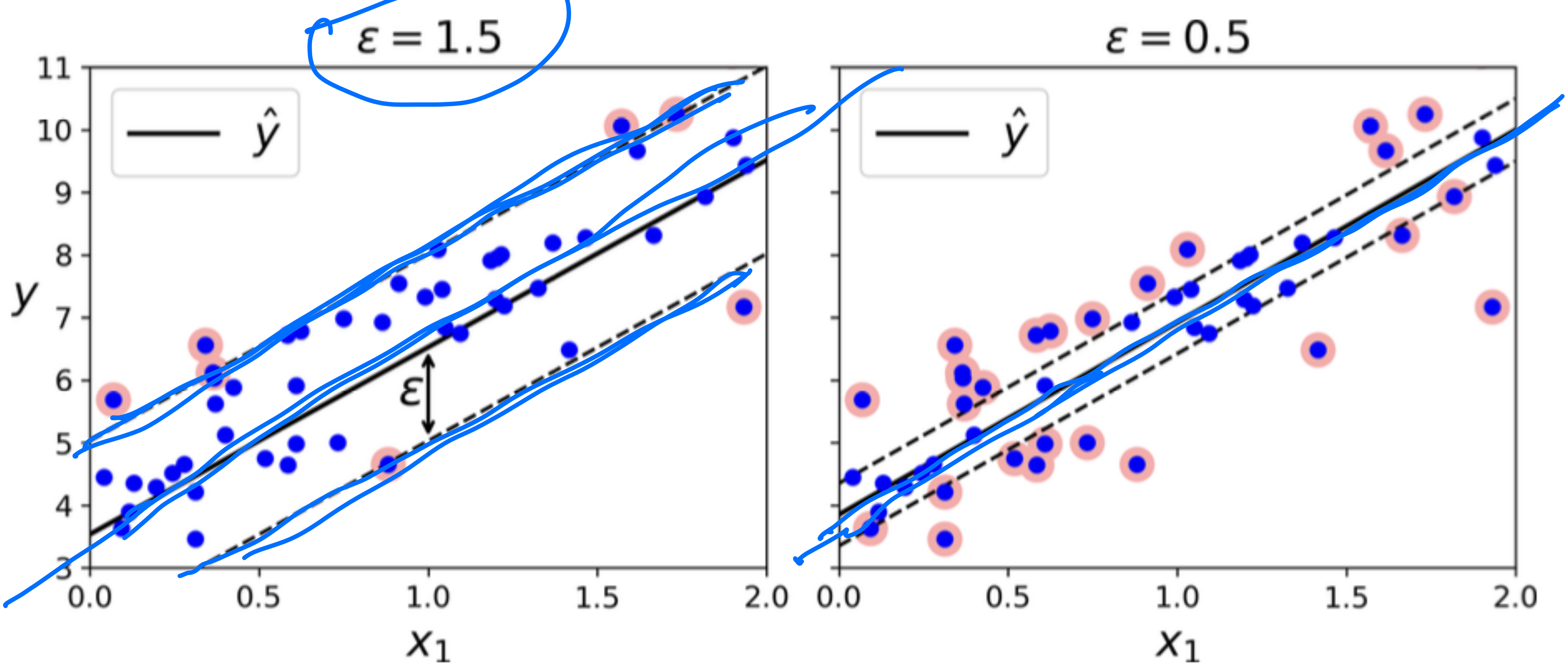
# Computational Time

- Scikit's LinearSVC class is based on the LIBLINEAR library, which implements an optimized algorithm for linear SVMs.
- It does not support the kernel trick.
- Computation time scales almost linearly with the number of training data points and the number of features.
- The LIBLINEAR based linear SVM algorithm can be computationally expensive if you need very high precision. This is controlled by the tolerance hyperparameter $\epsilon$ (called tol in Scikit-Learn).
- SVC class is based on the LIBSVM library which supports the kernel trick.
- Computation time will scale as a product of either the square or the cubic order of the number of training data points and the number of features. So it will be excruciatingly slow for large datasets.
- Works well with sparse features (meaning most of the features have zero value). This makes the algorithm compute faster.

$x_1, x_2, x_3, \ldots x_n$

$x_1, x_2 \simeq 0.0081$

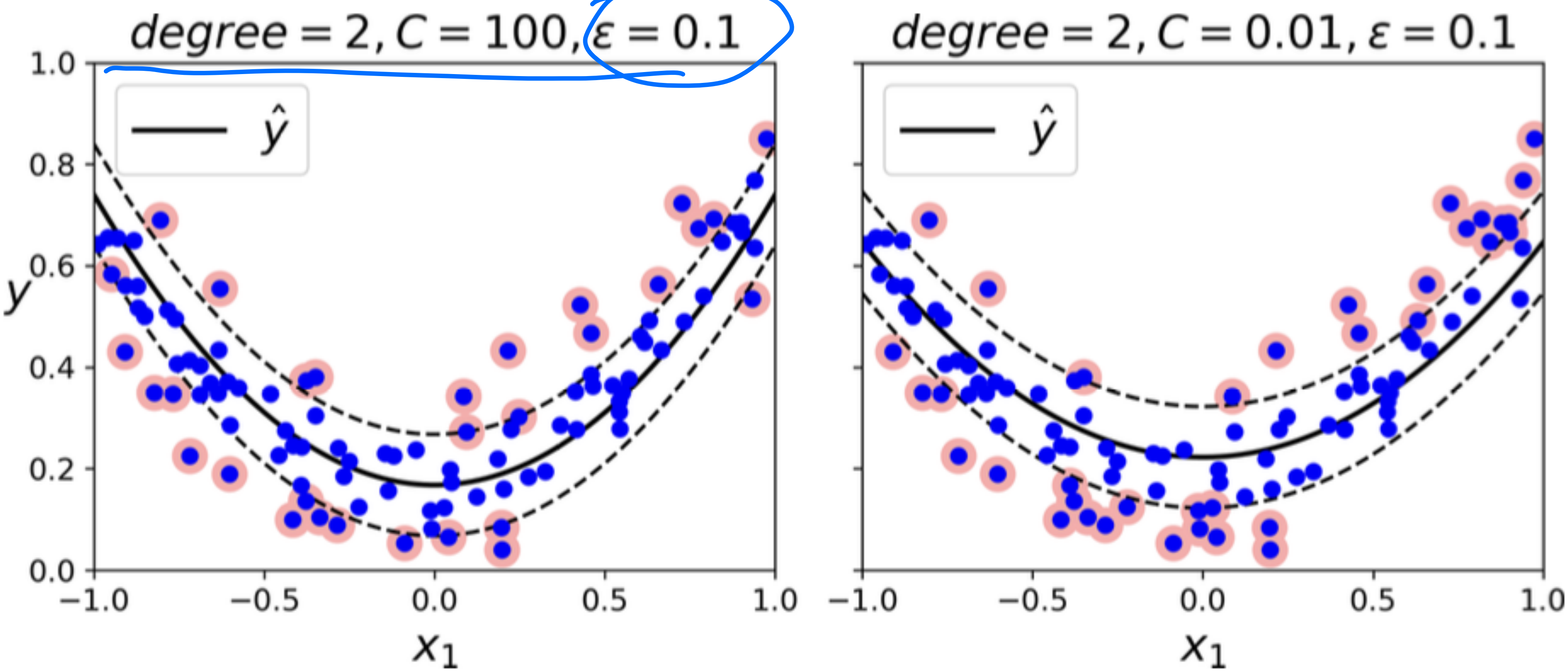
# SVM Regression (Do not Use it)




SVM Regression attempts to fit as many data points as possible on the street while limiting margin violations. The width of the street is controlled by the epsilon hyperparameter. Results shown for linear kernel. $\varepsilon$ - insensitive

Aurélien Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

# SVM Regression with polynomial kernel




```python
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
```

Aurélien Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

# Advantages and disadvantages

- **Advantages:**
  - Works well for small to moderate datasets. It will work on larger dataset but computational time may be very high.
  - Applicable to classification, regression, outlier detection.
  - Needs only the support vectors to make decision.
- **Disadvantages:**
  - Can be slow and scaling issues may arise.
  - Binary classifier by default, need to extend by training multiple classifiers
  - For high-dimensional data (>3), interpretation is difficult.
  - Unlike logistic regression, does not output estimated probabilities
  - **Optimizing C is crucial for SVM performance.**

Last Words: Dimensionality Reduction

Iris

| $x_1$ | $x_2$ | $x_3$ |
|---|---|---|
| ~~5~~ | 2 | 5 cm |
| 1.5 | 2.2 | 5.3 |
| 1.7 | | |
| 1.3 | 1.9 | 4.5 |
| 0.02 | 0.03 | 0.06 |

100,000

10,000

sparse