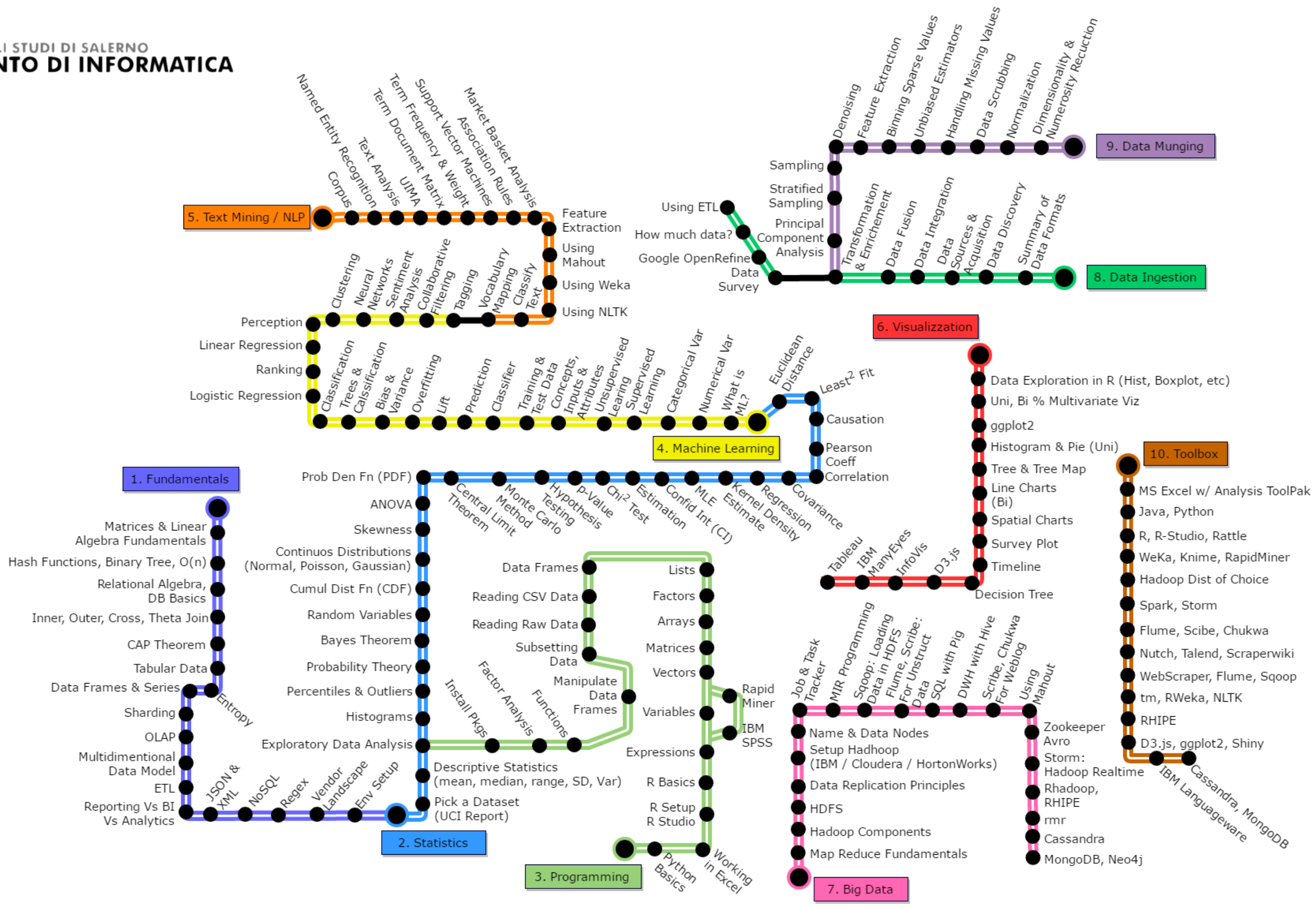
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science e Machine Learning

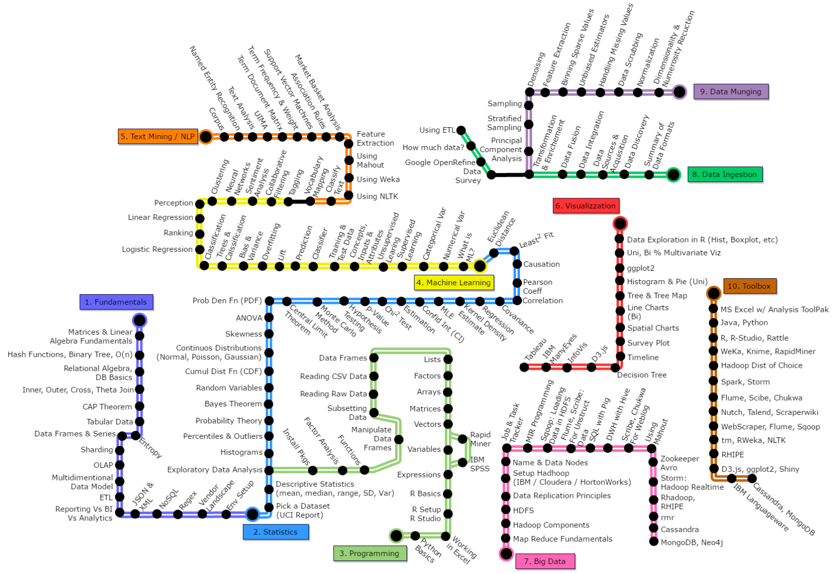
Support Vector Machine (Chapter 5 Geron's Book)

Aurelien Geron: «Hands on Machine Learning with Scikit Learn and TensorFlow, O'Reilly ed.

Prof. Giuseppe Polese, aa 2024-25

Outline

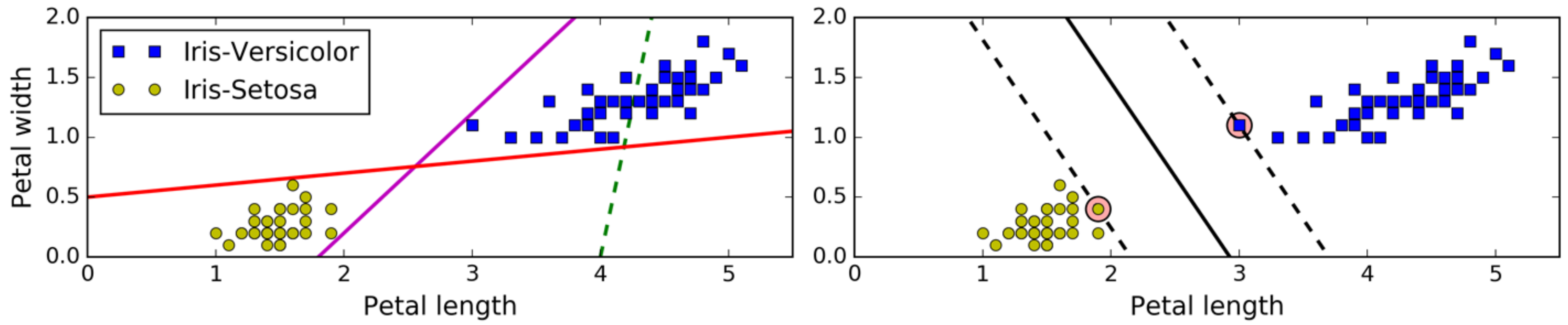
- **Support Vector Machine Classification**
  - Linear SVM Classification
    - Soft Margin Classification
  - Nonlinear SVM Classification
    - Polynomial Kernel
    - Adding Similarity Features
    - Gaussian RBF Kernel
    - Computational complexity
- Support Vector Machine Regression
- Decision Function and Predictions
  - Training Objective



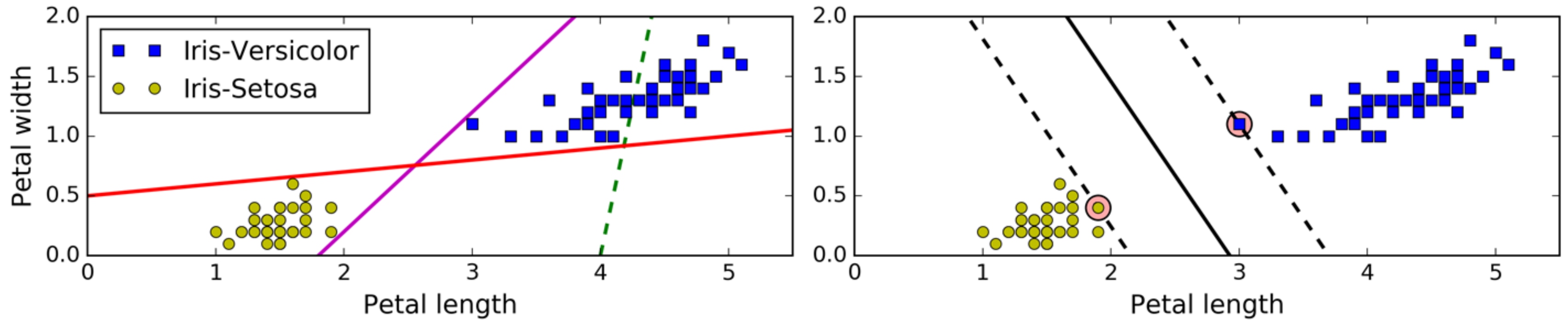
SVM

- A ***Support Vector Machine*** (SVM) is a powerful ML model, capable of performing linear/ nonlinear
  - Classification
  - Regression
  - Outlier detection

- SVMs are particularly well suited for classification of complex but small- or medium-sized datasets

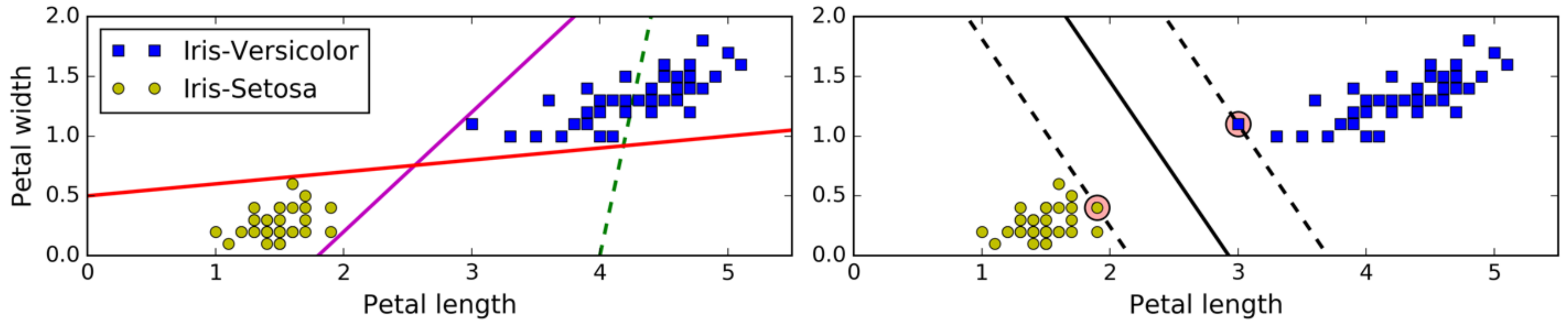


Linear classification: An example



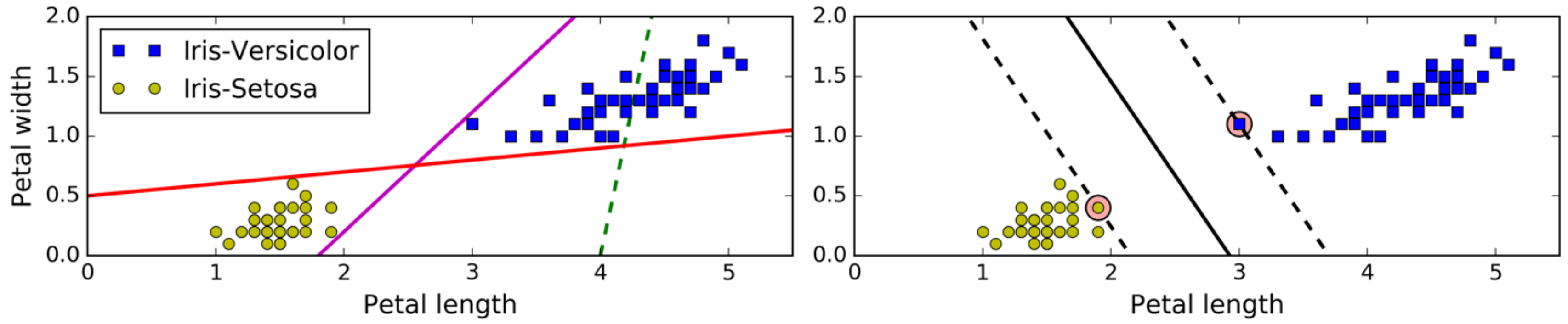
- The left plot shows the decision boundaries of three possible linear classifiers (*linearly separable*)
  - The model whose decision boundary is represented by the **dashed line** is so bad that it does not even separate the classes properly
  - The other two models work perfectly on this training set
    - Their decision boundaries come so close to the instances that these models could not perform as well on new instances

Linear SVM classification: An example



- The right plot shows the decision boundary of an SVM classifier
  - The **solid line** on the right not only separates the two classes but also stays far away from the closest training instances
- A SVM classifier can be seen as fitting the widest possible street (parallel dashed lines) between the classes
  - Large margin classification

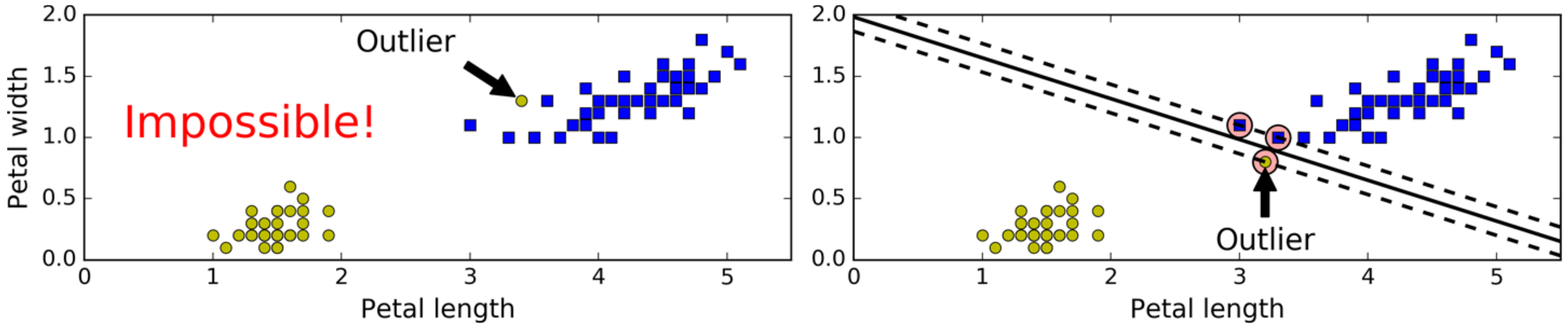
Linear SVM classification



- Notice that adding more training “off the street” will not effect the decision boundary at all
  - It is fully determined or “supported” by the instances located on the edge of the street
  - Such instances are called the support vectors (circled in Figure)
- SVMs are sensitive to the feature scales

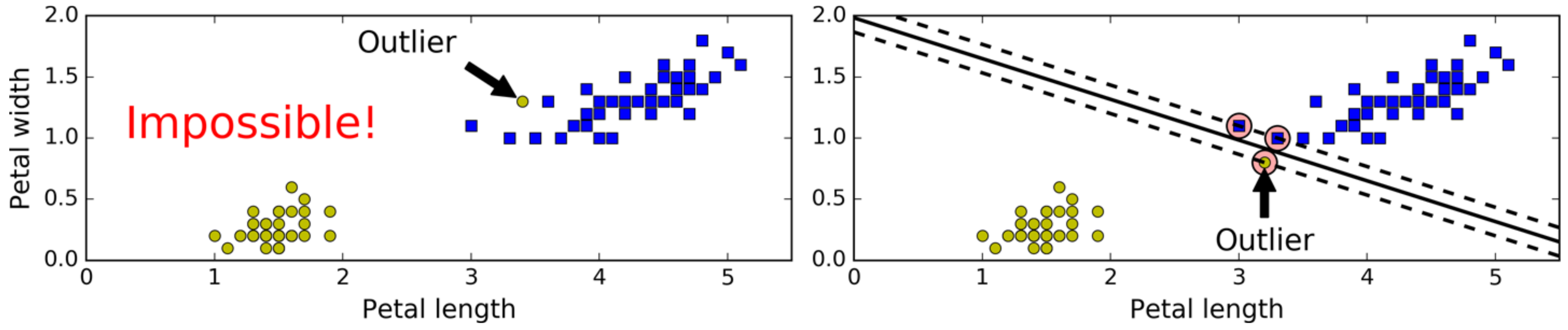
Hard Margin Classification (1)

- It is possible imposing to a SVM classifier that all instances “be off the street” and on the right side
  - Hard margin classification
- There are two main issues with hard margin classification
  - It only works if the data is linearly separable
  - It is quite sensitive to outliers



Hard Margin Classification (2)

- Consider the *iris* dataset with just one additional outlier



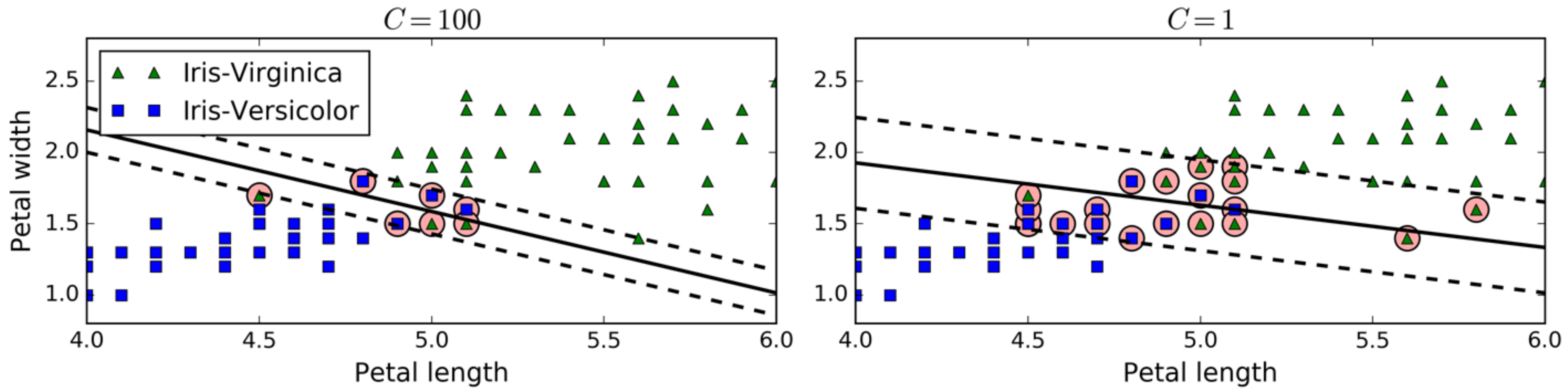
- On the left plot it is impossible to find a hard margin, and
- On the right plot the decision boundary ends up very different from the one we saw in the example without the outlier
    - It will probably not generalize as well

Soft Margin Classification

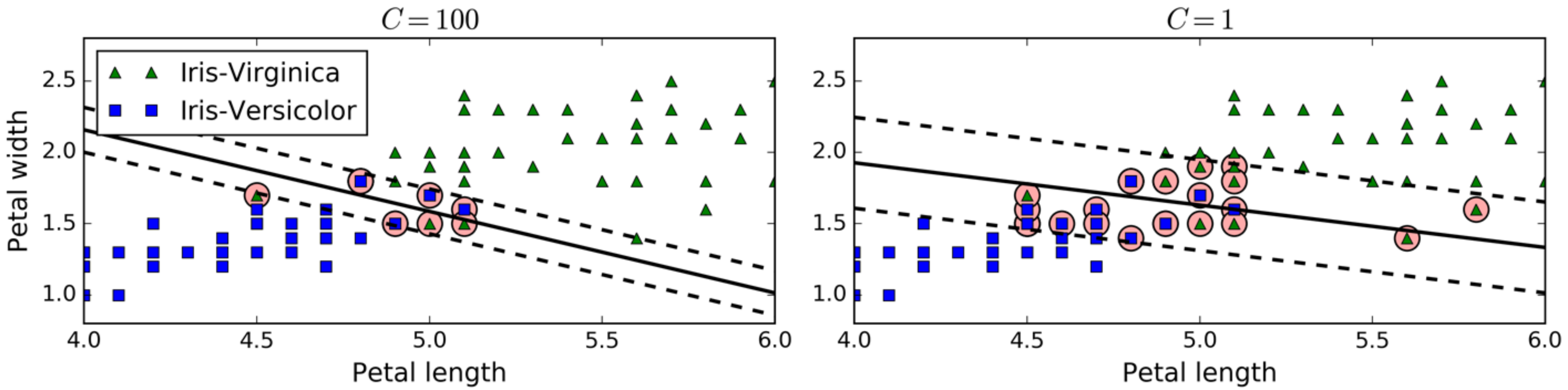
- To avoid these issues it is preferable to use a more flexible model
- The goal is to find a good balance between keeping the street as large as possible and limiting the margin violations
    - Instances that end up in the middle of the street or even on the wrong side
- This is called soft margin classification

SVM classes in Scikit-Learn (1)

- In Scikit-Learn's SVM classes, it is possible to control the balance using the C hyperparameter
  - A smaller C value leads to a wider street but more margin violations
- The Figure shows the decision boundaries and margins of two soft margin SVM classifiers on a nonlinearly separable dataset



SVM classes in Scikit-Learn (2)



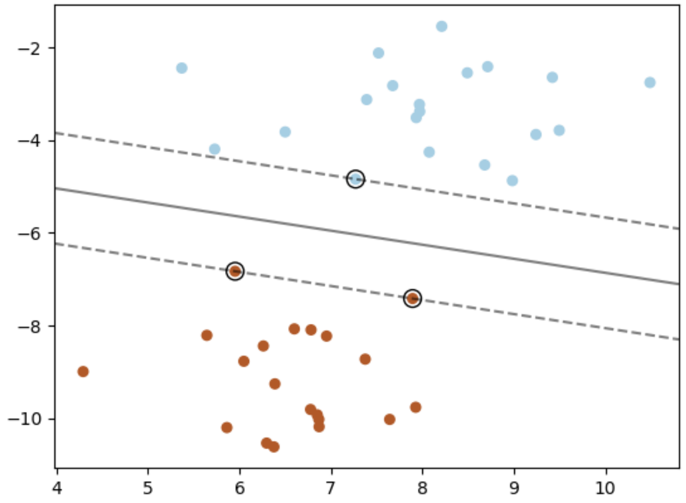
- On the left plot by using a high C value the classifier makes fewer margin violations but ends up with a smaller margin
- On the right plot by using a low C value the margin is much larger, but many instances end up on the street

- It seems that the second classifier will generalize better
  - It makes fewer prediction errors on this training set
    - margin violations are on the correct side of the decision boundary

Linear SVM classifier: Example code

- A linear SVM classifier on make_bobs dataset of the library sklearn.datasets

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs
# we create 40 separable points
X, y = make_blobs(n_samples=40, centers=2, random_state=6)
# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)
# plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()
# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)
# plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
# plot support vectors
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
           linewidth=1, facecolors='none', edgecolors='k')
plt.show()
```



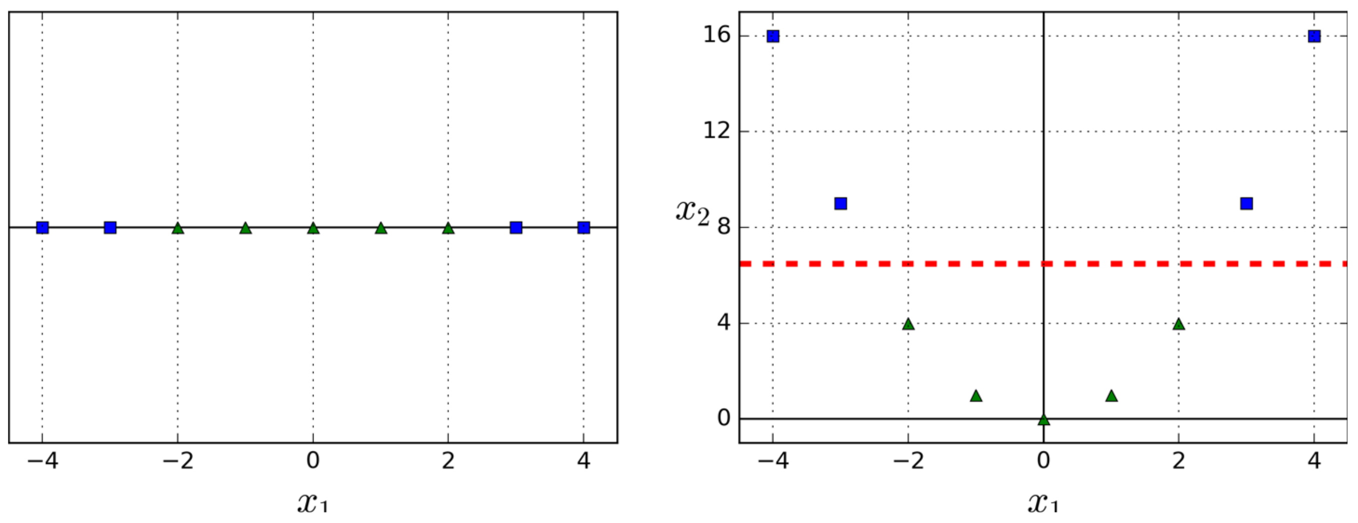
Nonlinear SVM classification

- Linear SVM classifiers are efficient and work surprisingly well in many cases
- However, many datasets are not even close to being linearly separable

- To handle nonlinear datasets
  - Add more features, such as polynomial features
- In some cases this can result in a linearly separable dataset

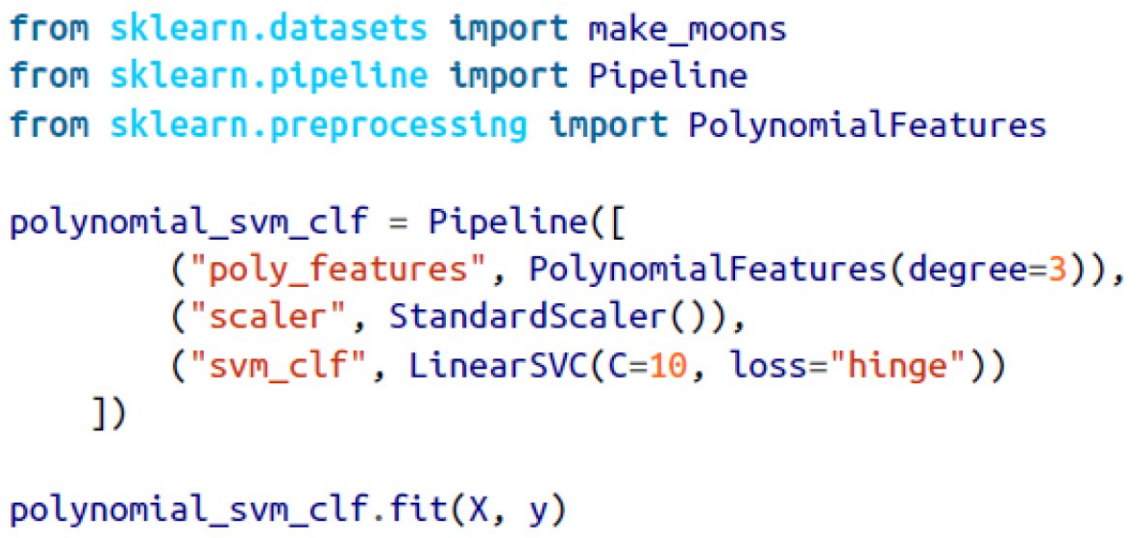
Nonlinear SVM classification: An example

- Consider the left plot of the Figure
  - A simple dataset with just one feature x_1
  - This dataset is not linearly separable
- If you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset is perfectly linearly separable



Adding Features in Scikit-Learn

- To add features in Scikit-Learn we create a Pipeline with a PolynomialFeatures transformer (see “Polynomial Regression”), followed by a StandardScaler and a LinearSVC
- The Python code for the make_moons dataset, from the library sklearn.datasets will be:

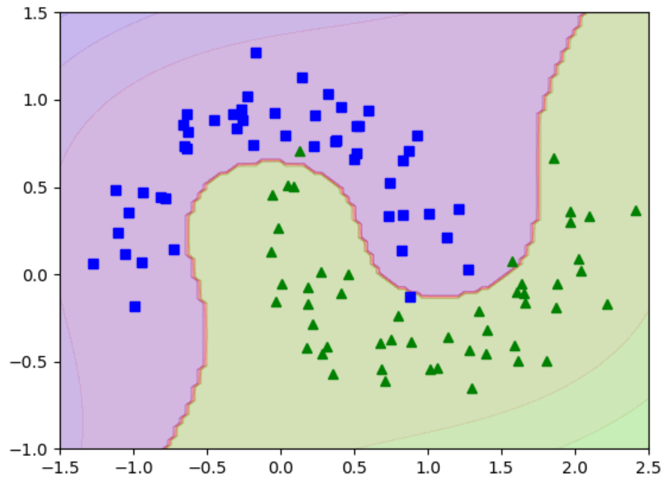


```python
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```

Nonlinear SVM classifier: Example code

- The complete Python code and the plot will be:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)
#Plot two different classes
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
#Define Pipeline
polynomial_svm_clf = Pipeline([
  ("poly_features", PolynomialFeatures(degree=3)),
  ("scaler", StandardScaler()), ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))])
#Fit dataset
polynomial_svm_clf.fit(X, y)
#Call defined fuction
plot_predictions(polynomial_svm_clf,

    [-1.5, 2.5, -1, 1.5])
plt.show()
```



```python
#Define procedure for creating graphic
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision,
        cmap=plt.cm.brg, alpha=0.1)
```

Polynomial Kernel (1)

- Adding polynomial features is simple to implement and can work great with all ML algorithms, but:
  - a low polynomial degree cannot deal with very complex datasets;
  - a high polynomial degree creates a huge number of features, making the model too slow
- Fortunately, when using SVMs it is possible to apply an almost miraculous mathematical technique called the kernel trick

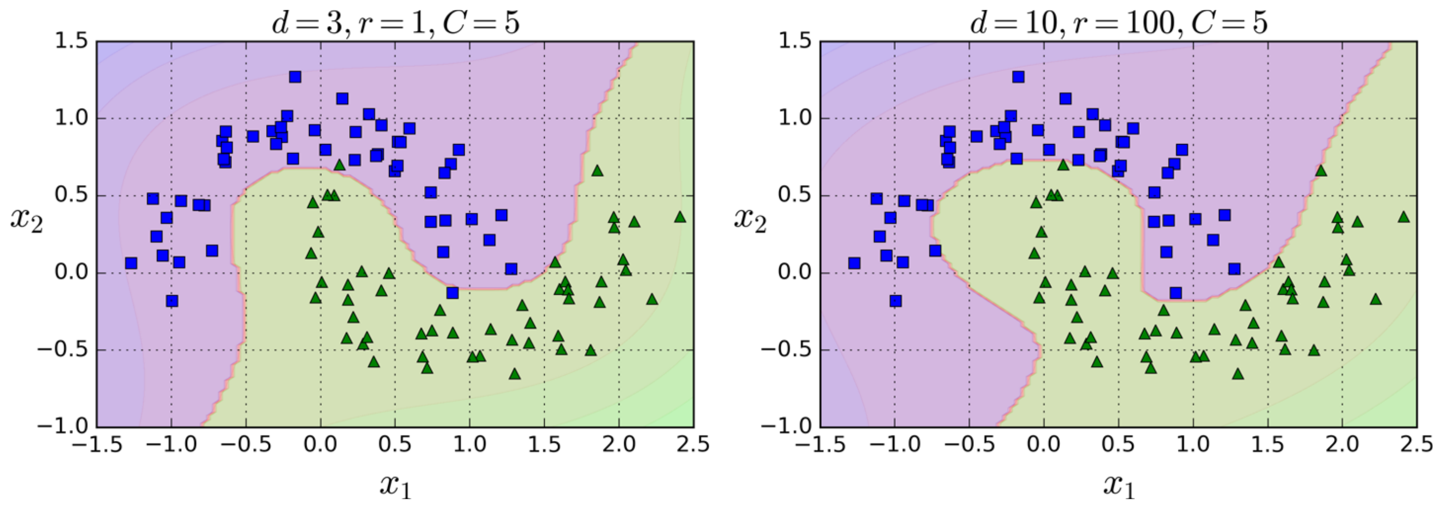
Polynomial Kernel (2)

- Kernel trick makes it possible to get the same result as if you added many polynomial features
  - even with very high-degree polynomials
  - without actually having to add them
- There is no combinatorial explosion of the number of features since you don’t actually add any features

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
("scaler", StandardScaler()),
("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
poly_kernel_svm_clf.fit(X, y)
```

Polynomial Kernel: An example

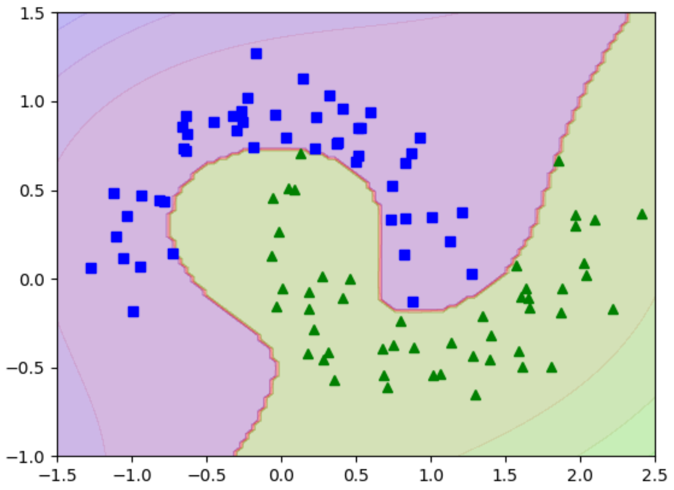
- Let’s try the code on the *Moons* dataset:
  - On the left plot is used a 3rd-degree polynomial kernel
  - On the right plot is used a 10th-degree polynomial kernel



- If your model is overfitting, you might want to reduce the polynomial degree
- If your model is underfitting, you can try increasing it

Polynomial Kernel: Example code

- A SVM classifier with polynomial kernel on make_moons dataset:



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

#Plot two different classes
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")

#Define Pipeline
poly_kernel_svm_clf = Pipeline([
("scaler", StandardScaler()),("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))])
#Fit dataset
poly_kernel_svm_clf.fit(X, y)
#Call defined fuction
plot_predictions(poly_kernel_svm_clf,

   [-1.5, 2.5, -1, 1.5])
plt.show()
```

```
#Define procedure for creating graphic
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision,
        cmap=plt.cm.brg, alpha=0.1)
```

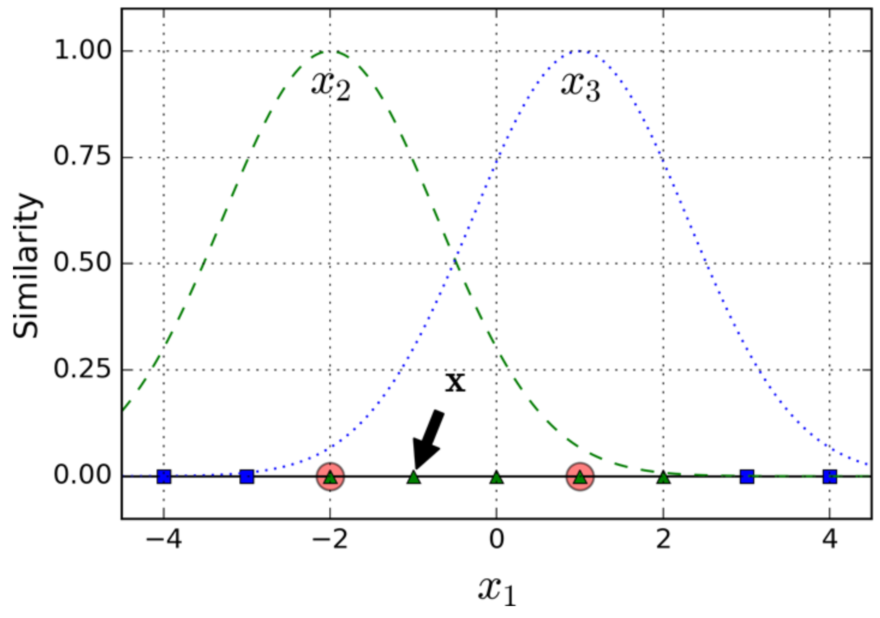
Adding Similarity Features

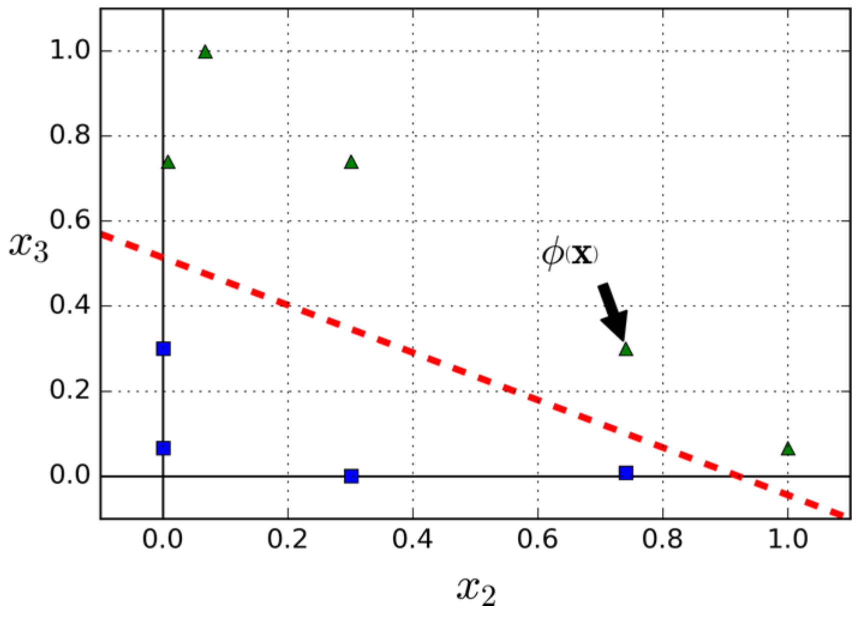
- Another technique to tackle nonlinear problems is to add features computed using a similarity function
  - It measures how much each instance resembles a particular landmark
- Let's take the one-dimensional dataset discussed earlier and *add two landmarks at* $x_1 = -2$ *and* $x_1 = 1$
- Next, let's add new features based on the Gaussian Radial Basis Function (RBF) as a similarity function, where ℓ is a landmark:

$$\phi_\gamma(x, \ell) = \exp(-\gamma ||x - \ell||^2)$$

Similarity Features: An example (1)

- The plot on the left shows the RBF with $\gamma = 0.3$
- It is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark)





Similarity Features: An example (2)

- Let’s look at the instance $x_1 = -1$
  - It is located at a distance of 1 from the first landmark, and 2 from the second landmark
- Therefore its new features are
  - $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$
  - $x_3 = \exp(-0.3 \times 22) \approx 0.30$
- The plot on the right shows the transformed dataset by dropping the original features
  - It is now linearly separable

Selecting Landmarks

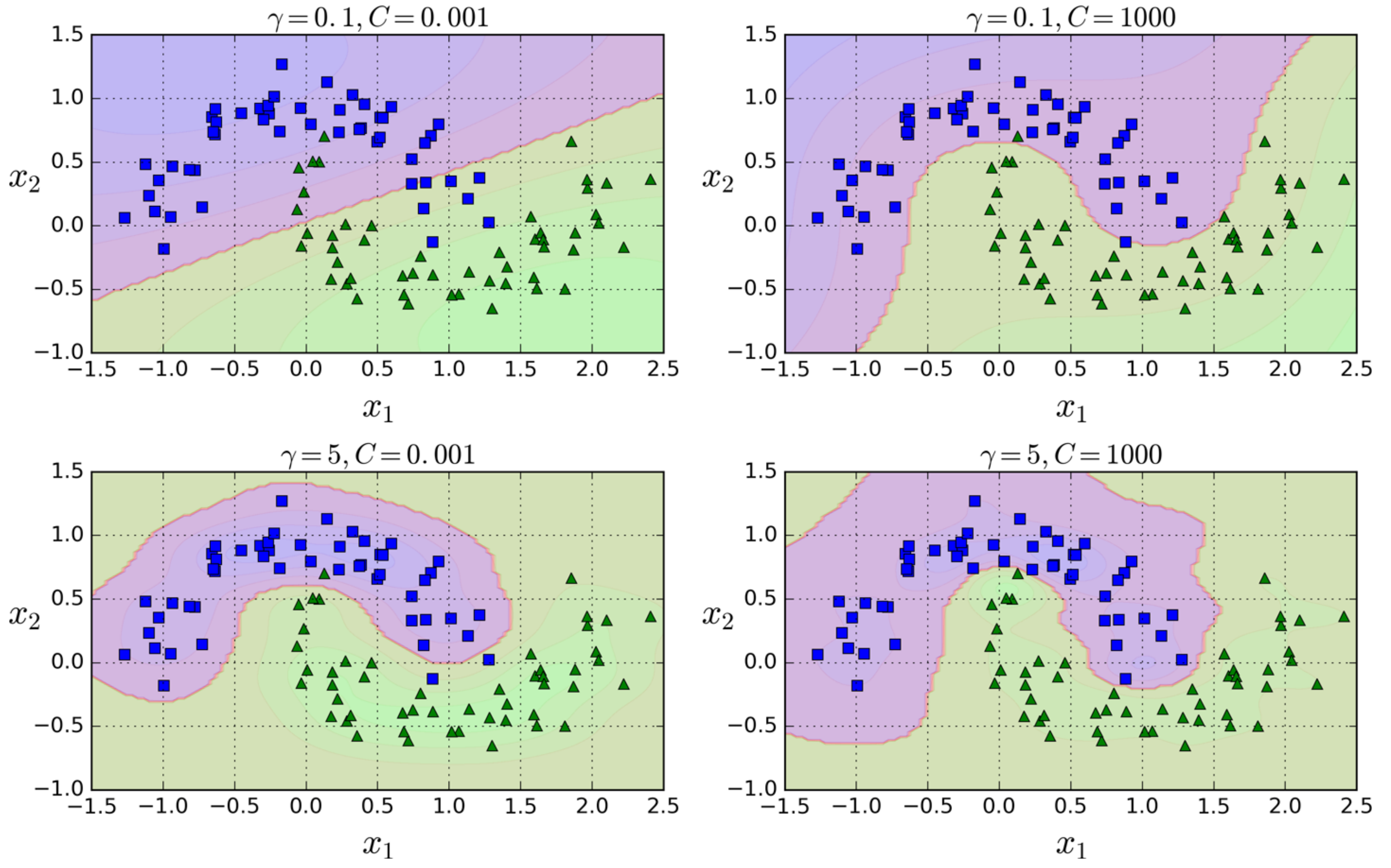
- The simplest approach for selecting landmarks is to create a landmark at the location of each and every instance in the dataset
  - This creates many dimensions and thus increases the chances that the transformed training set will be linearly separable
  - The downside is that a training set with *m* instances and *n* features gets transformed into a training set with *m* instances and *m* features (one for each landmark)
    - If your training set is very large, we end up with an equally large number of features

Gaussian RBF Kernel

- The similarity features method can be useful with any ML algorithm, but:
  - It may be computationally expensive to compute all the additional features, especially on large training sets
- However, the kernel trick makes SVM magic:
  - It makes it possible to obtain a similar result as if you had added many similarity features, without actually having to add them

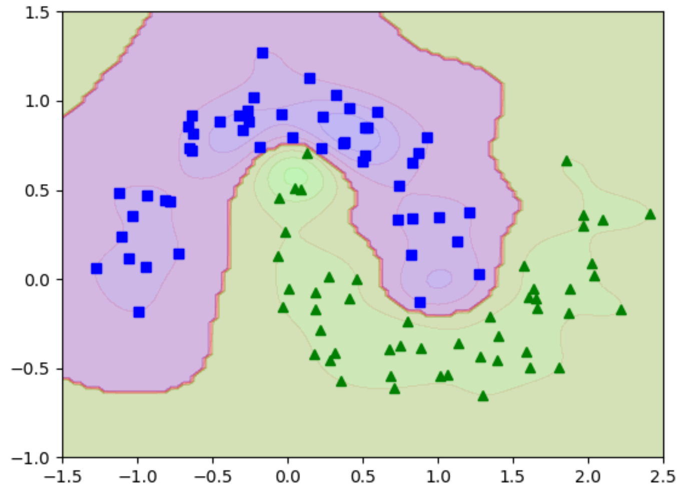
Gaussian RBF Kernel (2)

- Figure shows SVM with RBF kernel by varying γ and C parameters



Gaussian RBF Kernel: Example code

- A SVM classifier with RBF kernel on make_moons dataset of the library sklearn.datasets



```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

#Plot two different classes
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")

#Define Pipeline
rbf_kernel_svm_clf = Pipeline([
("scaler", StandardScaler()), ("svm_clf", SVC(kernel="rbf", gamma=5, C=1000))])

#Fit dataset
rbf_kernel_svm_clf.fit(X, y)

#Call defined fuction
plot_predictions(rbf_kernel_svm_clf,

     [-1.5, 2.5, -1, 1.5])
plt.show()
```

```python
#Define procedure for creating graphic
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision,
        cmap=plt.cm.brg, alpha=0.1)
```

Computational complexity

- The `LinearSVC` class is based on the `liblinear` library implementing an optimized algorithm for linear SVMs
  - It does not support the kernel trick, but it scales almost linearly with the training instances and the features
    - Its training time complexity is roughly $O(m \times n)$
- The `SVC` class is based on the `libsvm` library, which implements an algorithm that supports the kernel trick
  - It gets dreadfully slow when the number of training instances gets large
    - Its training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$

SVM Regression (1)

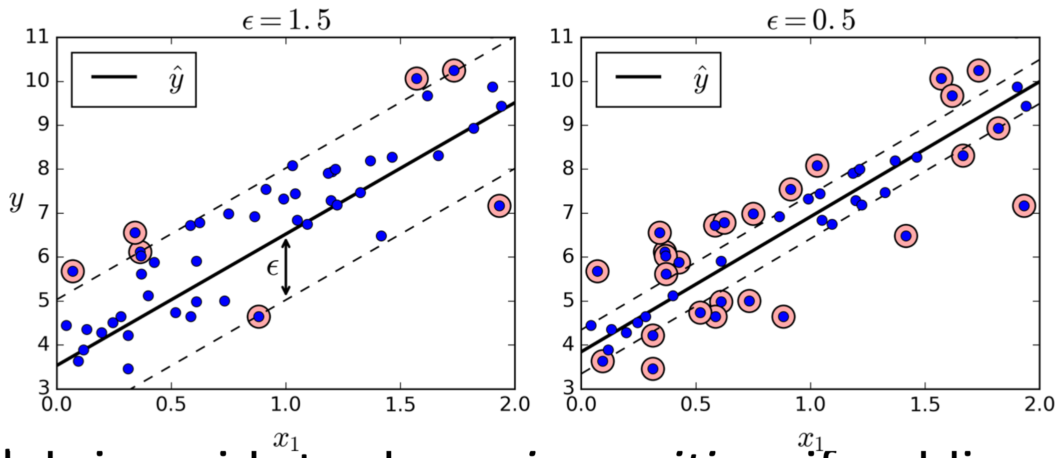
- SVM algorithm is quite versatile
  - not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression
- The goal is try to fit the largest possible street between two classes while limiting margin violations
- SVM Regression tries to fit as many instances as possible *on* the street while limiting margin violations
  - i.e., instances *off* the street
- The width of the street is controlled by a hyperparameter ϵ.

SVM Regression (2)

- The following code produces the left diagram, after training a linear SVM Regression model on some random linear data, after scaling and centering them.

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

- The right one is produced with $\epsilon = 0.5$ (smaller margin)

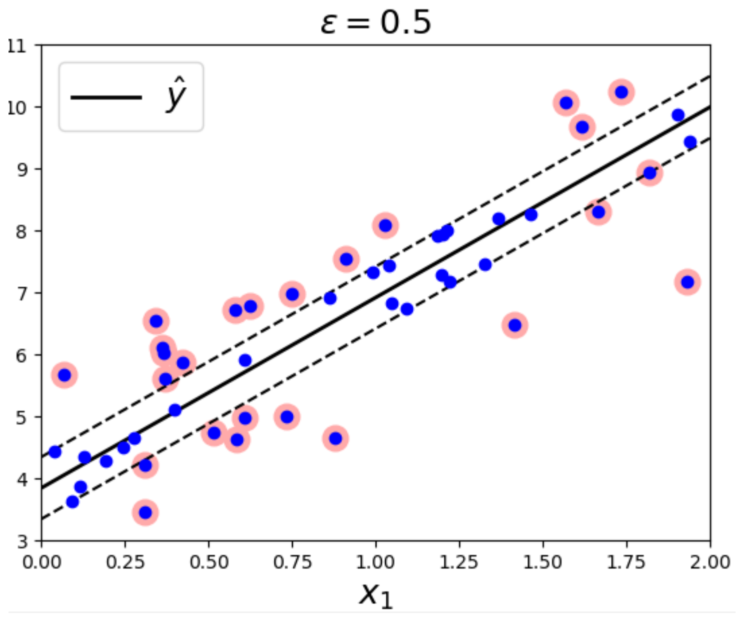


- The model is said to be *ϵ-insensitive* if adding more training instances within the margin does not affect the model's predictions

SVM Regression: Example code

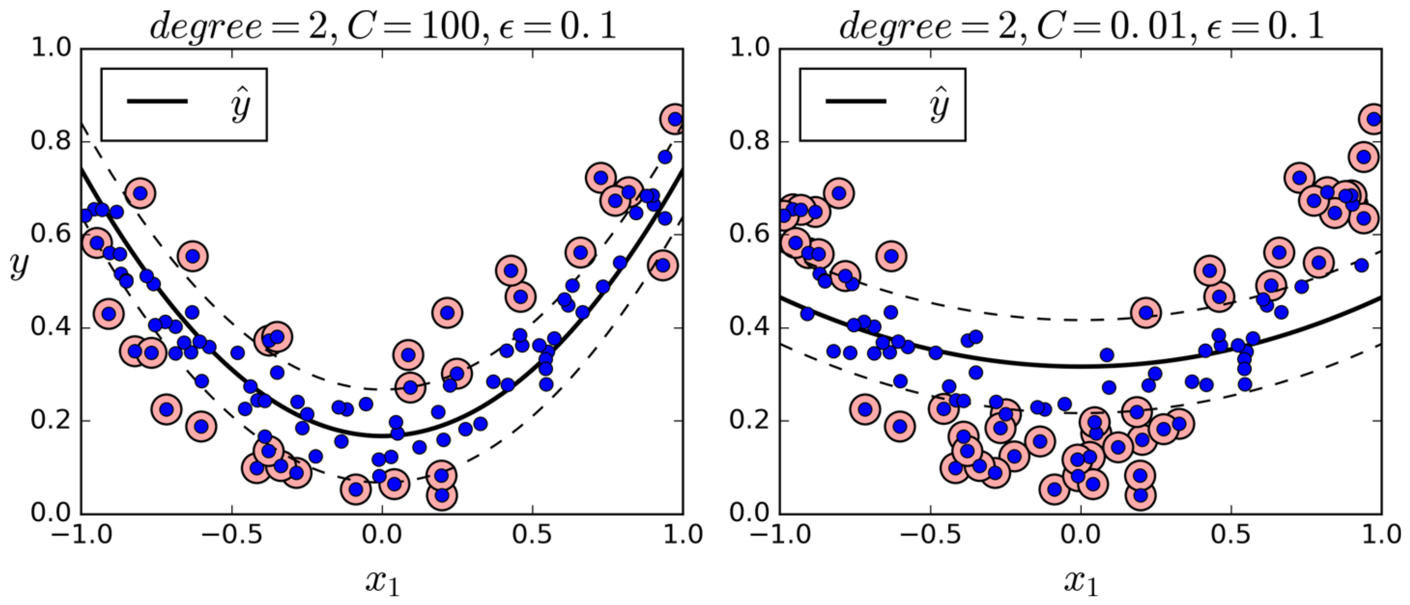
- A SVM classifier with linear regression on ad hoc dataset

```
import numpy as np
from sklearn.svm import LinearSVR
import matplotlib.pyplot as plt
#Definine random seed
np.random.seed(42)
m = 50
#Define axis (x) values
X = 2 * np.random.rand(m, 1)
#Define axis (y) values
y = (4 + 3 * X + np.random.randn(m, 1)).ravel()
#Define support vectors
def find_support_vectors(svm_reg, X, y):
    y_pred = svm_reg.predict(X)
    off_margin = (np.abs(y - y_pred) >= svm_reg.epsilon)
    return np.argwhere(off_margin)
#SVM linear regression
svm_reg2 = LinearSVR(epsilon=0.5, random_state=42)
svm_reg2.fit(X, y)
svm_reg2.support_ = find_support_vectors(svm_reg2, X, y)
#Define plot figure
def plot_svm_regression(svm_reg, X, y, axes):
    x1s = np.linspace(axes[0], axes[1], 100).reshape(100, 1)
    y_pred = svm_reg.predict(x1s)
    plt.plot(x1s, y_pred, "k-", linewidth=2, label=r"$\hat{y}$")
    plt.plot(x1s, y_pred + svm_reg.epsilon, "k--")
    plt.plot(x1s, y_pred - svm_reg.epsilon, "k--")
    plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=180, facecolors='#FFAAAA')
    plt.plot(X, y, "bo")
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.legend(loc="upper left", fontsize=18)
    plt.axis(axes)
plot_svm_regression(svm_reg2, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg2.epsilon), fontsize=18)
plt.show()
```



Kernelized SVM models

- To tackle nonlinear regression tasks we can use a kernelized SVM model
- The Figure shows SVM Regression on a random quadratic training set, using a 2nd-degree polynomial kernel
  - A little regularization on the left plot (i.e., a large C value)
  - A much more regularization on the right plot (i.e., a small C value)



Regression classes in Python

- The following code produces the model of left Figure, using Scikit-Learn's SVR class (supports kernel trick).

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2,
C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

- SVR is the regression equivalent of SVC class, and LinearSVR is the regression equivalent of LinearSVC.
- LinearSVR scales linearly with the size of the training set (like LinearSVC), while SVR gets too slow when the training set grows large (just like the SVC class).

Decision Function and Predictions

- The linear SVM classifier model predicts the class of a new instance $\mathbf{x}$ by computing the decision function

$$\hat{\boldsymbol{y}} = \mathbf{w}^T \cdot \mathbf{x} + b = w_1\, x_1 + \cdots + wn\, xn + b$$

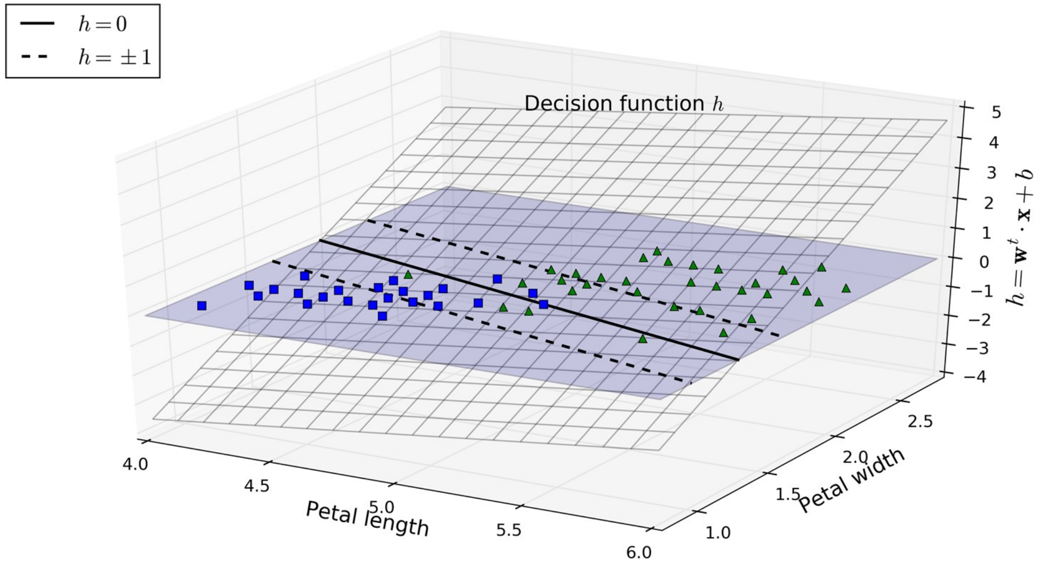
  - *$\hat{\boldsymbol{y}}$ is the predicted value;* $\hat{\boldsymbol{y}} \in \{0,1\}$
  - *n is the number of features (predictive attributes);*
  - *x_i is the ith feature value;* $x_i \in \mathbb{R}$
  - *b is the bias value;* $b \in \mathbb{R}$
  - $\mathbf{w}_i$ is the ith weight of the vector $\mathbf{w}$; $\mathbf{w}_i \in \mathbb{R}$

- If the result is positive, the predicted class $\hat{\boldsymbol{y}}$ is the positive class (1), or else it is the negative class (0):

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

Decision Function and Predictions: An example

- Figure shows the decision function for the iris dataset
  - It is a two-dimensional plane since this dataset has two features (petal width and petal length)
- The decision boundary is the set of points where the decision function is equal to 0
  - It is the intersection of two planes, which is a straight line

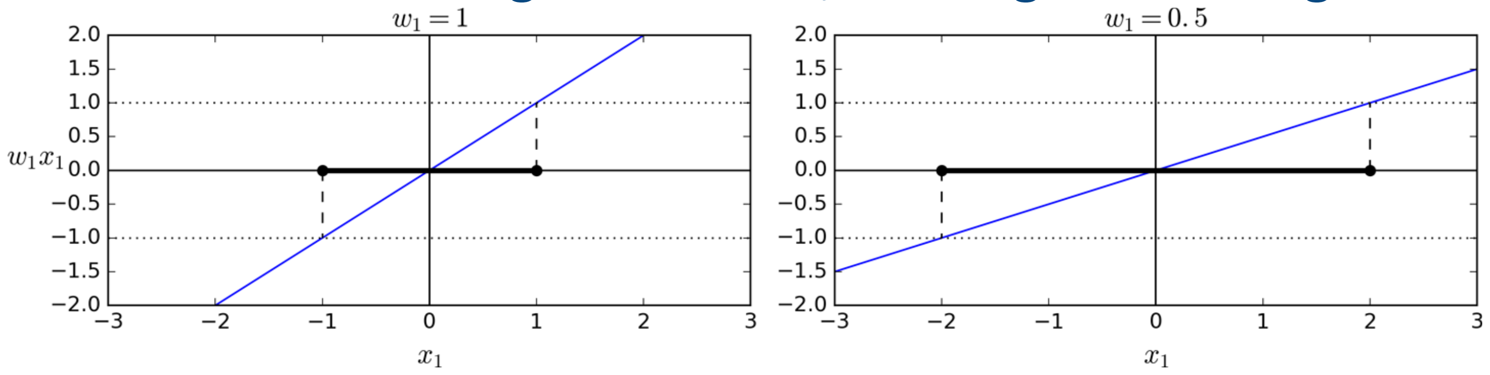


Decision Function and Predictions

- Dashed lines in Figure represent the points where the decision function is equal to 1 or –1
  - they are parallel and at equal distance to the decision boundary, forming a margin around it
- Training a linear SVM classifier means finding the value of **w** and *b* that make this margin as wide as possible while
  - avoiding margin violations (hard margin), or
  - limiting them (soft margin)

Training Objective (1)

- Considering the slope of previous decision function, it is equal to the norm of the weight vector, ‖ **w** ‖
  - Dividing the slope by 2, the points where the function is equal to ± 1 are going to be twice as far away from the decision boundary
- Dividing the slope by 2 will multiply the margin by 2
  - Perhaps this is easier to visualize in 2D
  - The smaller the weight vector **w**, the larger the margin



Training Objective (2)

- The goal is minimize || **w** || to get a large margin
  - If we also want to avoid margin violations (hard margin), we need the decision function > 1 for all positive training instances, and < –1 for negative instances

- If we define $\boldsymbol{t}^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $\boldsymbol{t}^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $\boldsymbol{t}^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ for all instances

Training Objective (3)

- We can therefore express the hard margin linear SVM classifier objective as the constrained optimization problem

minimize: $\quad \frac{1}{2} \cdot \boldsymbol{w}^T \cdot \boldsymbol{w}$
w,b

subject to: $\quad \boldsymbol{t}^{(i)}(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b) \geq 1 \; for \; i = 1,2, \ldots, m$

Training Objective (4)

- Notice that in the above formula we minimize $\frac{1}{2} \cdot \boldsymbol{w}^T \cdot \boldsymbol{w}$, which is equal to $\frac{1}{2} \cdot ||\boldsymbol{w}||^2$, rather than minimizing ‖ **w** ‖
  - this is because it will give the same result (since the values of **w** and *b* that minimize a value also minimize half of its square),
  - but $\frac{1}{2} \cdot ||\boldsymbol{w}||^2$ has a nice and simple derivative (it is just **w**) while ‖ **w** ‖ is not differentiable at **w** = **0**.
- Optimization algorithms work much better on differentiable functions

Training Objective (5)

- In order to get the soft margin objective, we need to introduce a slack variable $\zeta^{(i)} \geq 0$ for each instance
  - $\zeta^{(i)}$ measures how much the ith instance is allowed to violate the margin
- We now have two conflicting objectives:
  - making the slack variables as small as possible to reduce the margin violations
  - making $\frac{1}{2} \cdot \boldsymbol{w}^T \cdot \boldsymbol{w}$ as small as possible to increase the margin

Training Objective (6)

- To have the right tradoff between the two objectives mentioned above we use the C hyperparameter
  - This gives us the constrained optimization problem

minimize: $_{w,b,\zeta}$ $$\frac{1}{2} \cdot \boldsymbol{w}^T \cdot \boldsymbol{w} + \boldsymbol{C} \sum_{\boldsymbol{i=1}}^{\boldsymbol{m}} \boldsymbol{\zeta}^{(i)}$$

subject to: $$\boldsymbol{t}^{(i)}(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \, and \; \zeta^{(i)} \geq 0 \; for \; i = 1,2, \dots, m$$

Quadratic Programming

- Hard and soft margin are both convex quadratic optimization problems with linear constraints
  - They are known as Quadratic Programming (QP) problems
    - The general problem formulation is

$$\underset{\boldsymbol{p}}{\textit{minimize:}} \qquad \frac{1}{2} \cdot \boldsymbol{p}^T \cdot \boldsymbol{H} \cdot \boldsymbol{p} + \boldsymbol{f}^T \cdot \boldsymbol{p}$$

$$\textit{subject to:} \qquad \boldsymbol{A} \cdot \boldsymbol{p} \leq \boldsymbol{b}$$

  - where
    - $\boldsymbol{p}$ is an $\mathrm{np} -$ dimensional vector ($\mathrm{n_p} =$ number of parameters)
    - $\boldsymbol{H}$ is an $\mathrm{n_p} \times \mathrm{n_p}$ matrix
    - $\boldsymbol{f}$ is an $\mathrm{n_p} -$ dimensional vector
    - $\boldsymbol{A}$ is $\mathrm{n_c} \times \mathrm{n_p}$ matrix ($\mathrm{n_c} =$ number of constraints)
    - $\boldsymbol{b}$ is an $\mathrm{n_c} -$ dimensional vector

Quadratic Programming

- The expression $\boldsymbol{A} \cdot \boldsymbol{p} \leq \boldsymbol{b}$ actually defines $\boldsymbol{n_c}$ constraints: $\boldsymbol{p}^T \cdot \boldsymbol{a}^{(i)} \leq \boldsymbol{b}^{(i)}$ for $\boldsymbol{i}$ = 1, 2, ⋯, $\boldsymbol{n_c}$, where $\boldsymbol{a}^{(i)}$ is the vector containing the elements of the **i**th row of **A** and $\boldsymbol{b}^{(i)}$ is the **i**th element of $\boldsymbol{b}$

- We can easily verify that setting the QP parameters in the following way, we get the hard margin linear SVM classifier objective:
  - $n_p = n + 1$ *where n is the number of features* (*the* + 1 *is for the bias term*)
  - $n_c = m$ *where m is the number of training instances*
  - *H is the* $n_p \times n_p$ *identity matrix, except with a zero in the top − left cell* (*to ignore the bias term*)
  - $f = 0$, *an* n_p − *dimensional vector full of 0s*
  - $b = 1$, *an* n_c − *dimensional vector full of 1s*
  - $a^{(i)} = -t^{(i)} x^{(i)}$, *with extra bias feature* $x_0 = 1$

- So one way to train a hard margin linear SVM classifier is to use an off-the-shelf QP solver passing it previous parameters.

The Dual Problem

- Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*
- The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can even have the same solutions as the primal problem.
- Luckily, the SVM problem happens to meet these conditions, so you can choose to solve the primal problem or the dual problem; both will have the same solution. The follow equation shows the dual form of the linear SVM objective

Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)T}\cdot\mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The Dual Problem

- Once you find the vector $\boldsymbol{\alpha}_0$ that minimizes this equation (using a QP solver), you can compute $\mathbf{w}_0$ and $\boldsymbol{b}_0$ that minimize the primal problem by using the follow equation

From the dual solution to the primal solution

$$w_0 = \sum_{i=1}^{m} \alpha_0{}^{(i)} t^{(i)} x^{(i)}$$

$$b_0 = \frac{1}{n_s} \cdot \sum_{\substack{i=1 \\ \alpha_0{}^{(i)} > 0}}^{m} \left(t^{(i)} - w_0{}^T \cdot x^{(i)}\right)$$

- The dual problem is faster to solve than the primal when the number of training instances is smaller than the number of features. More importantly, it makes the kernel trick possible, while the primal does not

Kernelized SVM

- Suppose you want to apply a 2nd-degree polynomial transformation to a two dimensional training set, then train a linear SVM classifier on the transformed training set. The following shows the 2nd-degree polynomial mapping function ϕ that you want to apply

Second-degree polynomial mapping

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

Kernelized SVM

- Notice that the transformed vector is three-dimensional instead of two-dimensional. Now let’s look at what happens to a couple of two-dimensional vectors, **a** and **b**, if we apply this 2nd-degree polynomial mapping and then compute the dot product of the transformed vectors:

Kernel trick for 2nd –degree polynomial mapping

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = \begin{pmatrix} a_1{}^2 \\ \sqrt{2}\, a_1 a_2 \\ a_2{}^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1{}^2 \\ \sqrt{2}\, b_1 b_2 \\ b_2{}^2 \end{pmatrix} = a_1{}^2 b_1{}^2 + 2a_1 b_1 a_2 b_2 + a_2{}^2 b_2{}^2$$

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

Kernelized SVM

- The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ is called a 2nd-degree *polynomial kernel*. In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$ based only on the original vectors **a** and **b**, without having to compute the transformation ϕ. The follow equation lists some of the most commonly used kernels

Common Kernels

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

Online SVMs

- Finally, let’s take a quick look at online SVM classifiers (recall that online learning means learning incrementally, typically as new instances arrive)

- For linear SVM classifiers, one method is to use Gradient Descent (e.g., using *SGDClassifier*) to minimize the cost function in the following equation, which is derived from the primal problem. Unfortunately it converges much more slowly than the methods based on QP

Linear SVM classifier cost function

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

Online SVMs

- The first sum in the cost function $J(\mathbf{w}, \mathbf{b})$ will push the model to have a small weight vector **w**, leading to a larger margin
- The second sum computes the total of all margin violations. An instance's margin violation is equal to **0** if it is located off the 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 as few as possible
- It is also possible to implement online kernelized SVMs—for example, using **Incremental and Decremental SVM Learning** or **Fast Kernel Classifiers with Online and Active Learning**