Hands-on Machine Learning

5. Support Vector Machines

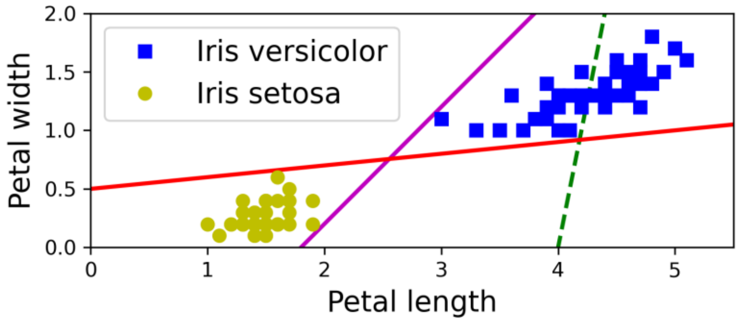
Support Vector Machine

- A *support vector machine* (SVM) is a powerful machine learning model, capable of performing linear or nonlinear classification and regression tasks.
- SVMs shine with small to medium-sized nonlinear datasets (i.e., hundreds to thousands of instances), especially for classification tasks.
- SVMs don’t scale very well to very large datasets.

1.
Linear SVM Classification

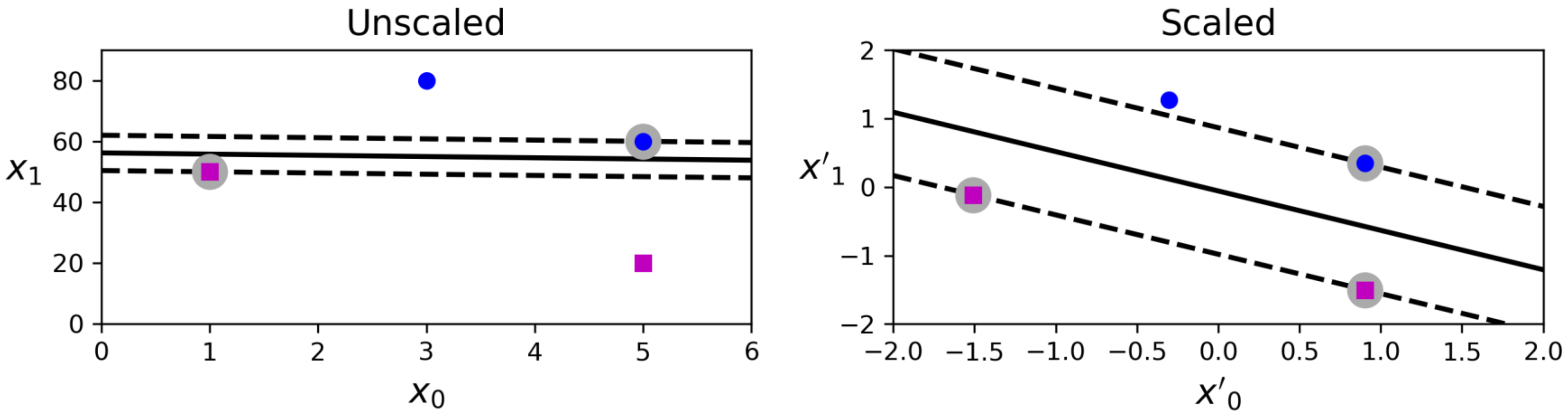
The Idea Behind SVM

- If classes are linearly separable, find a line that separates the two classes and stays far away from the closest training instances.
- You can think of an SVM classifier as fitting the widest possible street. This is called *large margin classification*.
- *Support vectors*: the instances located on the edge of the street.



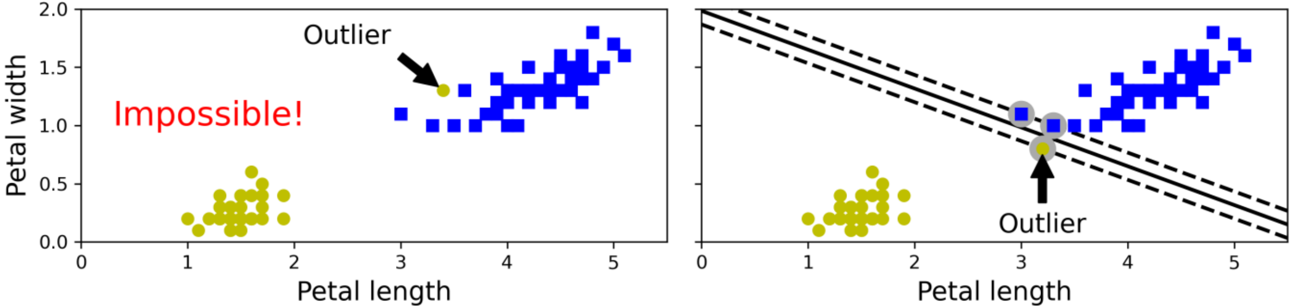
Sensitivity to Feature Scales

- SVMs are sensitive to the feature scales.
- After feature scaling using Scikit-Learn’s `StandardScaler`:



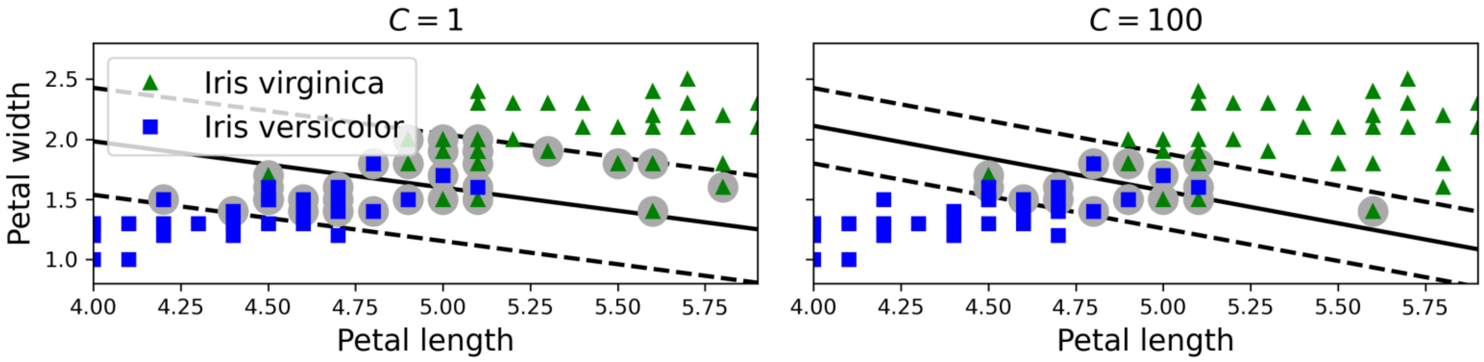
Soft Margin Classification

- *Hard margin classification:* we strictly impose that all instances must be off the street and on the right side.
  - It only works if the data is linearly separable
  - It is sensitive to outliers.
- *Soft margin classification*: find a good balance between keeping the street as large as possible and limiting the *margin violations*.



Soft Margin Classification

- We can use the regularization parameter C in SVM model to balance the tradeoff between margin violations and a better generalization (i.e., wider street).
- If your SVM model is overfitting, you can try regularizing it by reducing C.



Implementation in Scikit-Learn

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

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

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

```python
X_new = [[5.5, 1.7], [5.0, 1.5]]
svm_clf.predict(X_new)
```

```
array([ True, False])
```

`LinearSVC` vs. `SVC`

- Instead of using the `LinearSVC` class, we could use the `SVC` class with a linear kernel:

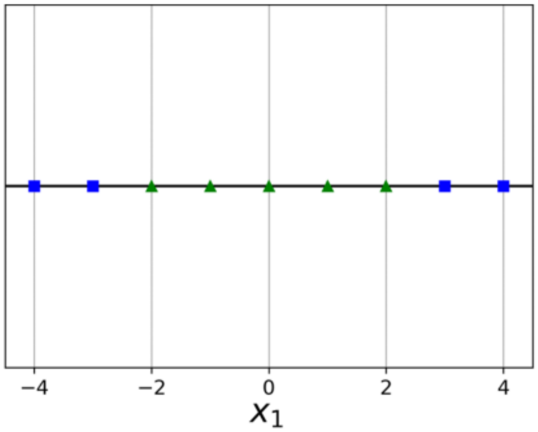
```
SVC(kernel="linear", C=1)
```

- Unlike `LogisticRegression`, `LinearSVC` doesn't have a `predict_proba`() method to estimate the class probabilities.
  - Use the SVC class and set its `probability` hyperparameter to `True`.
  - The model will fit an extra model at the end of training to map the SVM decision function scores to estimated probabilities.
  - After that, the `predict_proba`() and `predict_log_proba`() methods will be available.

2.
Nonlinear SVM Classification

Nonlinear Datasets

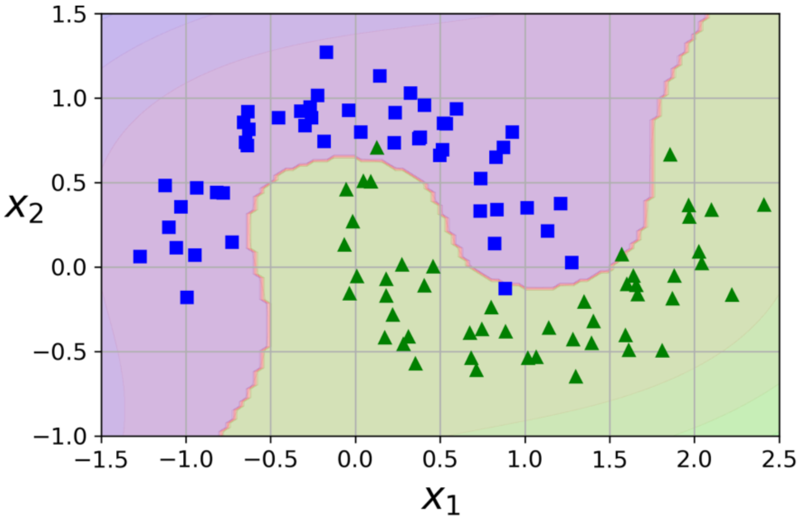
- Many datasets are not even close to being linearly separable.
- One approach to handling nonlinear datasets is to add more features, such as polynomial features.
  - This may result in a linearly separable dataset.



Linear SVC for Nonlinear Dataset

➢ Create a pipeline containing a `PolynomialFeatures` transformer followed by a `StandardScaler` and a `LinearSVC`.

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

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

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



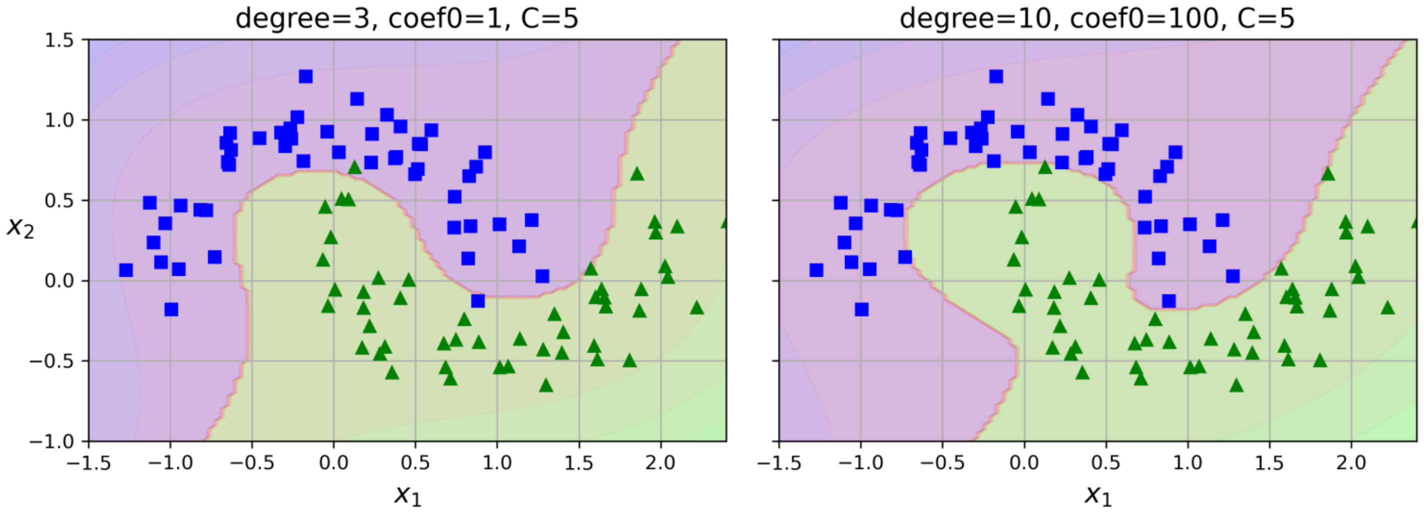
Polynomial Kernel

- Adding polynomial features is simple to implement and work with all sorts of ML algorithms.
  - Low polynomial degree: cannot deal with very complex datasets.
  - High polynomial degree: huge number of features make a slow model.

- The *kernel trick* makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them.
  - No combinatorial explosion of the number of features because you don’t actually add any features.

Polynomial Kernel

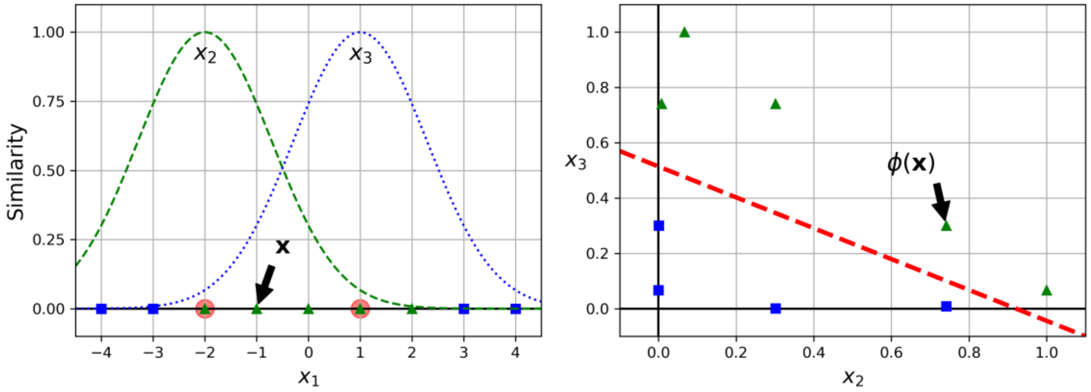
```
from sklearn.svm import SVC

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



Similarity Features

- Another way to tackle nonlinear problems is to add features computed using a *similarity function*, which measures how much each instance resembles a particular *landmark*.
- Example. Gaussian *Radial Basis Function* (RBF):

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

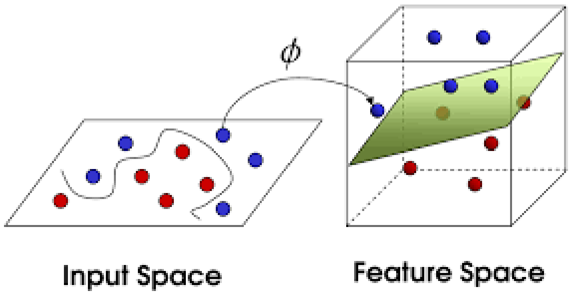


Landmarks

- How to select the landmarks?
  - The simplest approach is to create a landmark at the location of each and every instance in the dataset.
    - Doing that creates many dimensions and 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 (assuming you drop the original features).
  - If your training set is very large, you end up with an equally large number of features.

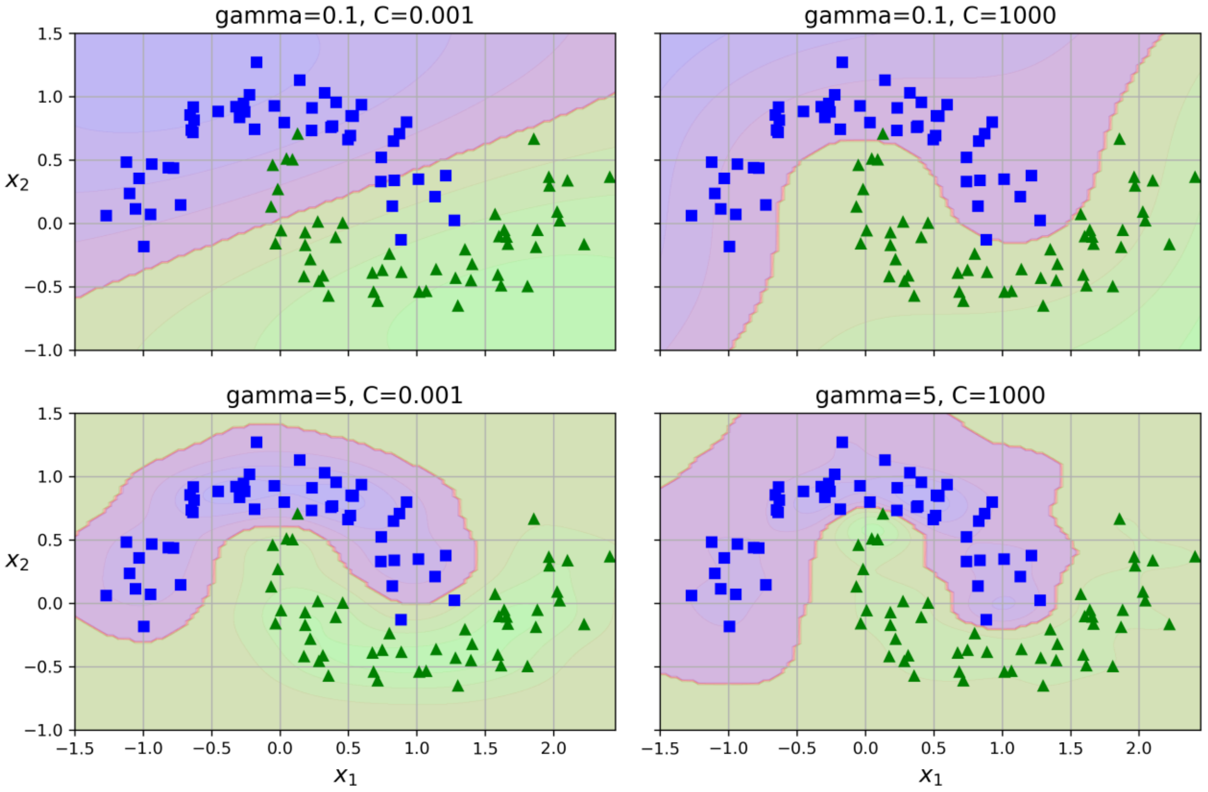
Gaussian RBF Kernel

- The similarity features method can be useful with any machine learning algorithm, but it is computationally expensive.
  - For each landmark, we have a new feature.
- The kernel trick makes it possible to obtain a similar result as if you had added many similarity features.




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

SVM Classifiers with RBF Kernel



Computational Complexity

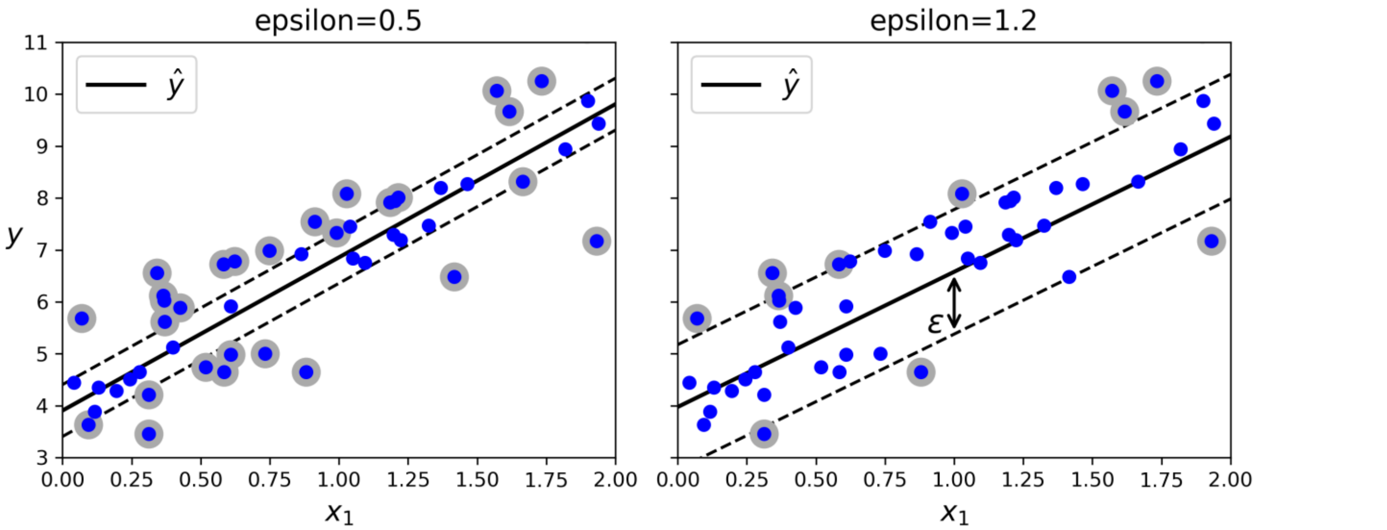
- The `LinearSVC` class is based on the `liblinear` library.
  - Its training time complexity is almost $O(m \times n)$.
- The `SVC` class is based on the `libsvm` library.
  - Training time complexity is between $O(m^2 \times n)$ and $O(m^3 \times n)$.
  - It scales well with the number of features, especially with *sparse features* (i.e., when each instance has few nonzero features).

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

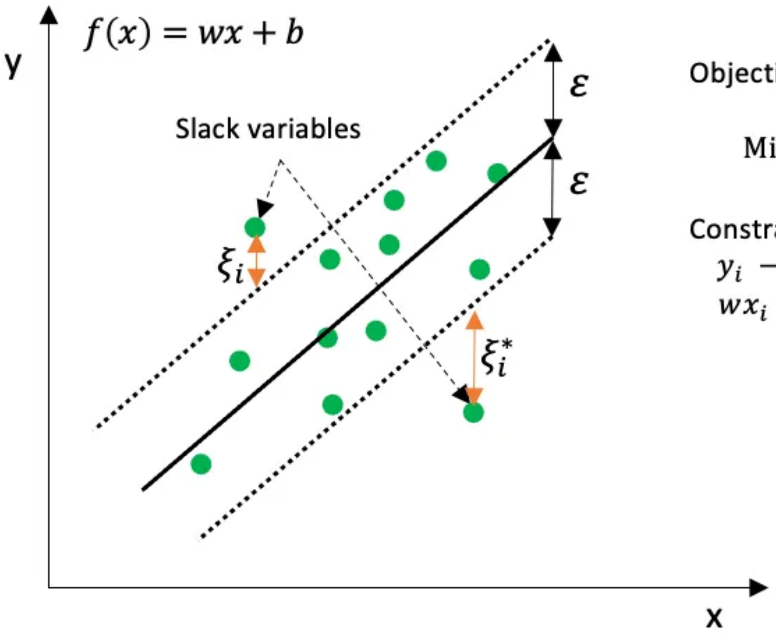
3.
SVM Regression

SVM Regression

- Using SVM for regression: 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



Objective:

$$\text{Minimize: } \frac{1}{2}||w||^2 + \mathrm{C}\sum_{i=1}^{l}(\xi_i + \xi_i^*)$$

Constraints:

$$y_i - wx_i - b \leq \varepsilon + \xi_i$$
$$wx_i + b - y_i \leq \varepsilon + \xi_i^*$$
$$\xi_i^*, \xi_i^* \geq 0$$

Univariate linear SVR (allowing for errors)

Implementing SVM Regression

- Use Scikit-Learn's `LinearSVR` class to perform linear SVM Regression:

```
from sklearn.svm import LinearSVR

svm_reg = make_pipeline(StandardScaler(),
                        LinearSVR(epsilon=0.5, random_state=42))
svm_reg.fit(X, y)
```

- To tackle nonlinear regression tasks, use a kernelized SVM model:

```
from sklearn.svm import SVR

svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1))
svm_poly_reg.fit(X, y)
```

SVM Regression