Machine Learning

| Radial Basis Function (RBF) Kernel

Prof. Dr. Mostafa Elhosseini

https://youtube.com/ElhosseiniAcademy

Agenda

- ~~Polynomial kernel~~
- ~~Dot Product~~
- ~~Polynomial kernel – derivation~~
- ~~Coding~~

Model

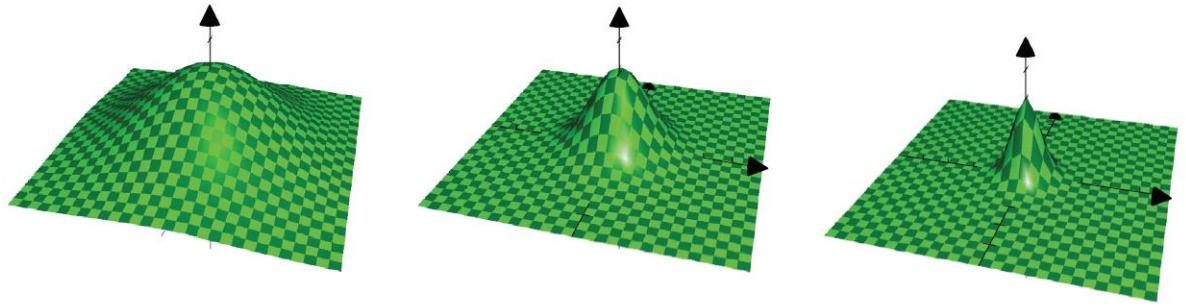
- $e^{-\gamma(a-b)^2}$
- Just like with the Polynomial Kernel, a and b refer to two different Dosage measurements.
- The difference between the measurements is then squared, giving us the squared distance between the two observations.
- Thus, the amount of influence one observation has on another is a function of the squared distance.
- γ (gamma), which is determined by Cross Validation, scales the squared distance, and thus, it scales the influence.

The γ parameter - Overfitting : Underfitting

- The γ hyperparameter in the Radial Basis Function (RBF) kernel plays a crucial role in determining the influence of a single training example on the decision boundary in Support Vector Machines (SVMs) or other kernel methods.

The γ parameter - Overfitting : Underfitting

- Many different radial basis functions exist, namely one per point in the plane. There are actually many more. Some of them lift the plane at a point and form a narrow surface, and others form a wide surface.



Small γ Medium γ Large γ

The γ parameter - Overfitting : Underfitting

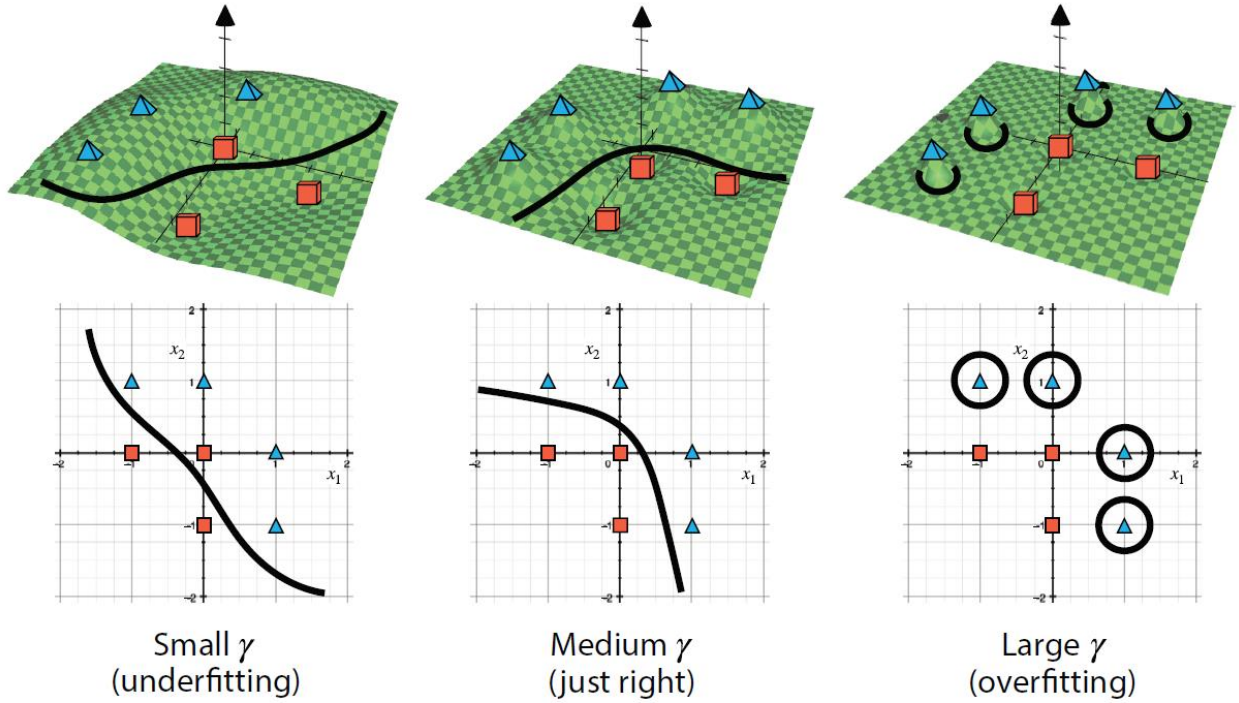
- In practice, the wideness of our radial basis functions is something we want to tune. For this, we use a parameter called the gamma parameter. When gamma is small, the surface formed is very wide, and when it is large, the surface is very narrow.
- The RBF kernel is mathematically defined as:
- $K(x, c) = e^{-\gamma \|x - c\|^2}$
  - x is the input vector
  - c is the center or another input vector.
  - $\|x - c\|$is the Euclidean distance between x and c
  - γ (gamma) is a hyperparameter that controls the width of the RBF.

The γ parameter - Overfitting : Underfitting

- The RBF kernel is mathematically defined as:
- $K(x, c) = e^{-\gamma \|\boldsymbol{x}-\boldsymbol{c}\|^2}$
- When γ is large, the function $e^{-\gamma \|\boldsymbol{x}-\boldsymbol{c}\|^2}$ decreases very rapidly as the distance $\|\boldsymbol{x} - \boldsymbol{c}\|$ increases.
- This means that only points very close to the center c have a significant influence on the kernel output.
- The decision boundary will be highly influenced by individual data points, leading to a more complex and possibly overfitted model that can capture fine details in the data but may not generalize well to new data.

The γ parameter - Overfitting : Underfitting

- The RBF kernel is mathematically defined as:
- $K(x, c) = e^{-\gamma \|\boldsymbol{x}-\boldsymbol{c}\|^2}$



The γ parameter - Overfitting : Underfitting

- The RBF kernel is mathematically defined as:
- $K(x, c) = e^{-\gamma \|x - c\|^2}$
- When γ is increased, the bell-shaped curve becomes narrower, meaning that each training instance has a smaller range of influence. This results in a more complex, irregular decision boundary that closely follows individual data points. While this can lead to a more accurate fit on the training data, it also increases the risk of overfitting, where the model captures noise rather than the underlying pattern.

The γ parameter - Overfitting : Underfitting

- The RBF kernel is mathematically defined as:
- $K(x, c) = e^{-\gamma \|x - c\|^2}$

- Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. String kernels are sometimes used when classifying text documents or DNA sequences

Numerical Example

- Let's create a small dataset of 5 points that are not linearly separable. Assume we have two classes, labeled as +1 and -1.

Point	x_1	x_2	Class
A	1.0	2.0	+1
B	2.0	1.0	+1
C	2.0	2.0	-1
D	3.0	3.0	-1
E	3.0	1.0	+1

- The RBF kernel is defined as:
- $K(\boldsymbol{x}, \acute{\boldsymbol{x}}) = exp(-\gamma \|\boldsymbol{x} - \acute{\boldsymbol{x}}\|^2)$
- Let's choose $\gamma = 0.5$ for simplicity.

Numerical Example

- We need to compute the RBF kernel between each pair of points. The kernel matrix K will be a 5x5 matrix where each element $K_{i,j}$ represents the kernel function applied to points x_i and x_j.
- Point A vs. All Points:
  - $K(\boldsymbol{A}, \acute{\boldsymbol{A}}) = exp(-0.5\|[1,2] - [1,2]\|^2 = \exp(0) = 1$
  - $K(\boldsymbol{A}, \boldsymbol{B}) = exp(-0.5\|[1,2] - [2,1]\|^2 = \exp(-1) = 0.3679$
  - $K(\boldsymbol{A}, \boldsymbol{C}) = exp(-0.5\|[1,2] - [2,2]\|^2 = \exp(-0.5) = 0.6065$
  - $K(\boldsymbol{A}, \boldsymbol{D}) = exp(-0.5\|[1,2] - [3,3]\|^2 = \exp(-2.5) = 0.0821$
  - $K(\boldsymbol{A}, \boldsymbol{E}) = exp(-0.5\|[1,2] - [3,1]\|^2 = \exp(-2.5) = 0.0821$

Numerical Example

- Point B vs. All Points:
  - $K(B, A) = 0.3679$ (Already calculated)
  - $K(B, B) = \exp(0) = 1$
  - $K(B, C) = 0.6065$
  - $K(B, D) = 0.0821$
  - $K(B, E) = 0.6065$
- Do the same for other points

$$K = \begin{pmatrix} 1 & 0.3679 & 0.6065 & 0.0821 & 0.0821 \\ 0.3679 & 1 & 0.6065 & 0.0821 & 0.6065 \\ 0.6065 & 0.6065 & 1 & 0.3679 & 0.1353 \\ 0.0821 & 0.0821 & 0.3679 & 1 & 0.1353 \\ 0.0821 & 0.6065 & 0.1353 & 0.1353 & 1 \end{pmatrix}$$

Numerical Example

- Once the kernel matrix is computed, the SVC algorithm uses this matrix instead of the original feature vectors to find the optimal hyperplane in the transformed space.
- The RBF kernel allows the SVC to map the input data into a higher-dimensional space where the data points are more easily separable. Even though the data points are not linearly separable in the original 2D space, the RBF kernel transforms the problem into a space where a linear boundary (hyperplane) can effectively separate the classes — see Appendix

Use cases

- With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first.
- The LinearSVC class is much faster than SVC(kernel="linear"), especially if the training set is very large. If it is not too large, you should also try kernelized SVMs, starting with the Gaussian RBF kernel; it often works really well.
- Then, if you have spare time and computing power, you can experiment with a few other kernels using hyperparameter search.
- If there are kernels specialized for your training set's data structure, make sure to give them a try too.

Computational Complexity

- The LinearSVC class is based on the liblinear library, which implements an optimized algorithm for linear SVMs. It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of 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. The training time complexity is usually between $O(m^2 \times n)$and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances), so this algorithm is best for small or medium-sized nonlinear training sets

Computational Complexity

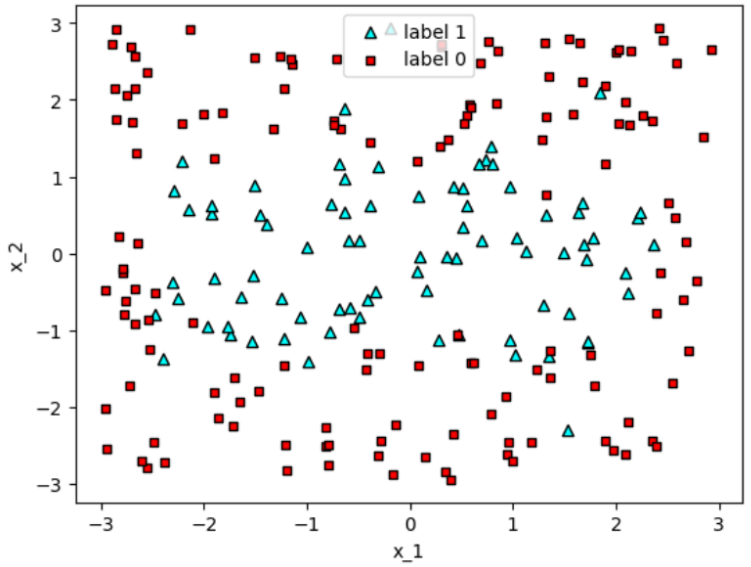
- The SGDClassifier is designed for linear models, meaning it is suitable for algorithms like linear Support Vector Machines (SVM), logistic regression, and linear regressions.
- The SGDClassifier class also performs large margin classification by default, and its hyperparameters–especially the regularization hyperparameters (alpha and penalty) and the learning_rate–can be adjusted to produce similar results as the linear SVMs

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

Coding the RBF kernel

```
# Loading the two_circles dataset

two_circles_data = pd.read_csv('two_circles.csv')
features = np.array(two_circles_data[['x_1', 'x_2']])
labels = np.array(two_circles_data['y'])
utils.plot_points(features, labels)
```

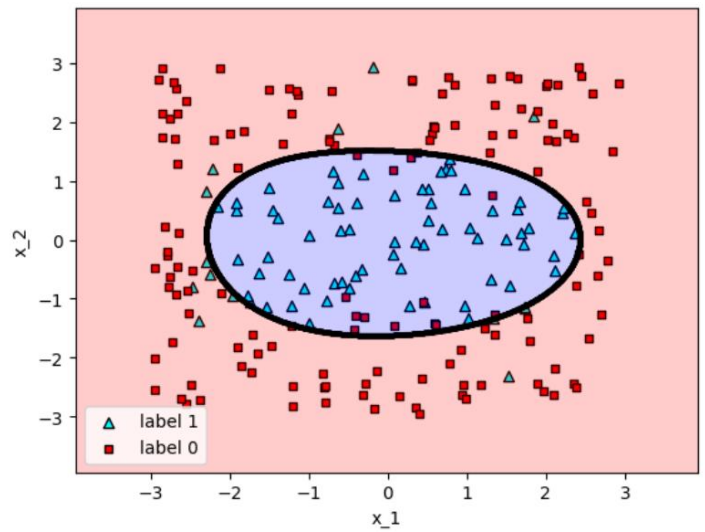


Coding the RBF kernel

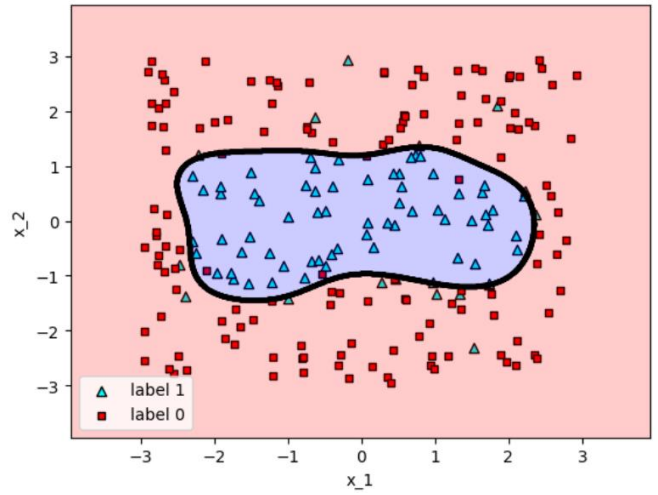
```
# gamma = 0.1
svm_gamma_01 = SVC(kernel='rbf', gamma=0.1)
svm_gamma_01.fit(features, labels)
print("Gamma = 0.1")
print("Accuracy:", svm_gamma_01.score(features, labels))
utils.plot_model(features, labels, svm_gamma_01)
```

```
# gamma = 1
svm_gamma_1 = SVC(kernel='rbf', gamma=1)
svm_gamma_1.fit(features, labels)
print("Gamma = 1")
print("Accuracy:", svm_gamma_1.score(features, labels))
utils.plot_model(features, labels, svm_gamma_1)
```

```
Gamma = 0.1
Accuracy: 0.8772727272727273
```



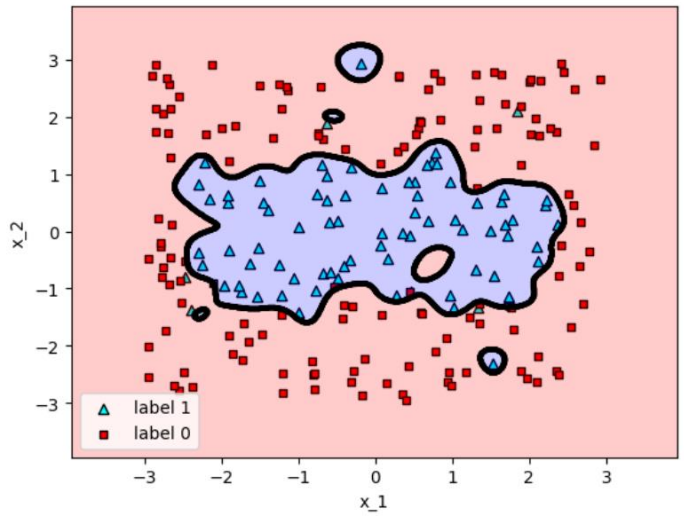

```
Gamma = 1
Accuracy: 0.9045454545454545
```



Coding the RBF kernel

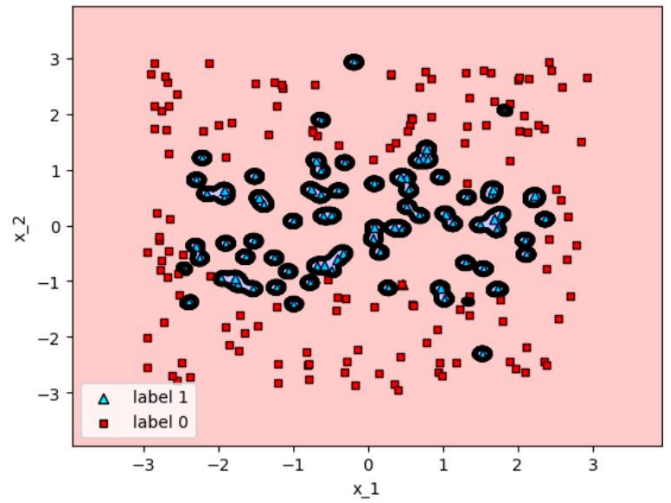
```
# gamma = 10
svm_gamma_10 = SVC(kernel='rbf', gamma=10)
svm_gamma_10.fit(features, labels)
print("Gamma = 10")
print("Accuracy:", svm_gamma_10.score(features, labels))
utils.plot_model(features, labels, svm_gamma_10)
```

```
Gamma = 10
Accuracy: 0.9636363636363636
```




```
# gamma = 100
svm_gamma_100 = SVC(kernel='rbf', gamma=100)
svm_gamma_100.fit(features, labels)
print("Gamma = 100")
print("Accuracy:", svm_gamma_100.score(features, labels))
utils.plot_model(features, labels, svm_gamma_100)
```

```
Gamma = 100
Accuracy: 0.990909090909091
```



SVM Regression

- The objective of SVR is to find a function (a hyperplane in a high-dimensional space) that approximates the relationship between input features and the continuous output values. This function should predict the output with minimal error while being as flat (simple) as possible.
- One of the central ideas in SVR is the concept of the ε-tube. Instead of requiring the model to pass exactly through every data point (as in standard linear regression), SVR allows a margin of tolerance, ε, within which predictions can deviate from the actual values without incurring any penalty. This margin defines a "tube" around the predicted function.

SVM Regression

- The goal is to find a function that fits within this ε-tube as closely as possible, ensuring that most data points fall within this margin.
- To use SVMs for regression instead of classification, the trick is to tweak the objective: instead of trying 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

- Reducing ϵ increases the number of support vectors, which regularizes the model.
- When ε is reduced, the width of the ε-tube decreases. This means that the model now allows less deviation from the predicted function without incurring a penalty. In other words, the model becomes more sensitive to small errors, and more data points will lie outside the ε −tube.
- With a narrower ε −tube, more data points will lie outside this margin. In SVR, only the data points outside the ε-tube (or exactly on its boundary) contribute to the model as support vectors. These points influence the position and shape of the regression line.

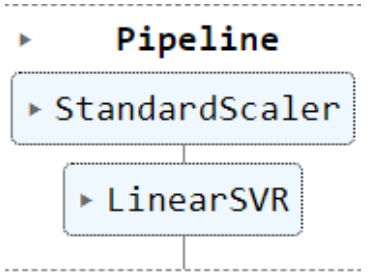
SVM Regression

- As more points fall outside the ε-tube when ε is reduced, these points become support vectors. Support vectors are the critical data points that define the regression function, so increasing the number of support vectors means the model is being influenced by a larger portion of the data.

```
from sklearn.svm import LinearSVR

# extra code – these 3 lines generate a simple linear dataset
np.random.seed(42)
X = 2 * np.random.rand(50, 1)
y = 4 + 3 * X[:, 0] + np.random.randn(50)

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

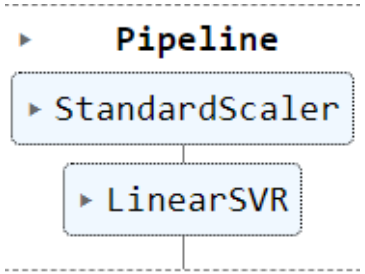


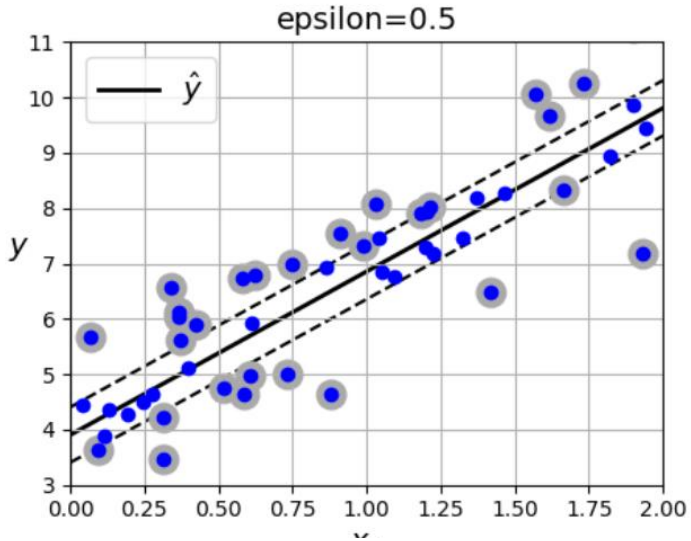
SVM Regression

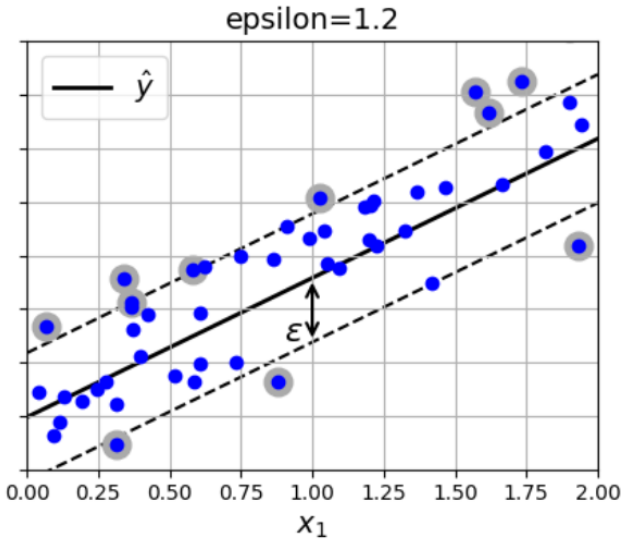
```
from sklearn.svm import LinearSVR

# extra code – these 3 lines generate a simple linear dataset
np.random.seed(42)
X = 2 * np.random.rand(50, 1)
y = 4 + 3 * X[:, 0] + np.random.randn(50)

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







SVM Regression

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

```
from sklearn.svm import SVR

X, y = [...]  # a quadratic dataset
svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1))
svm_poly_reg.fit(X, y)
```