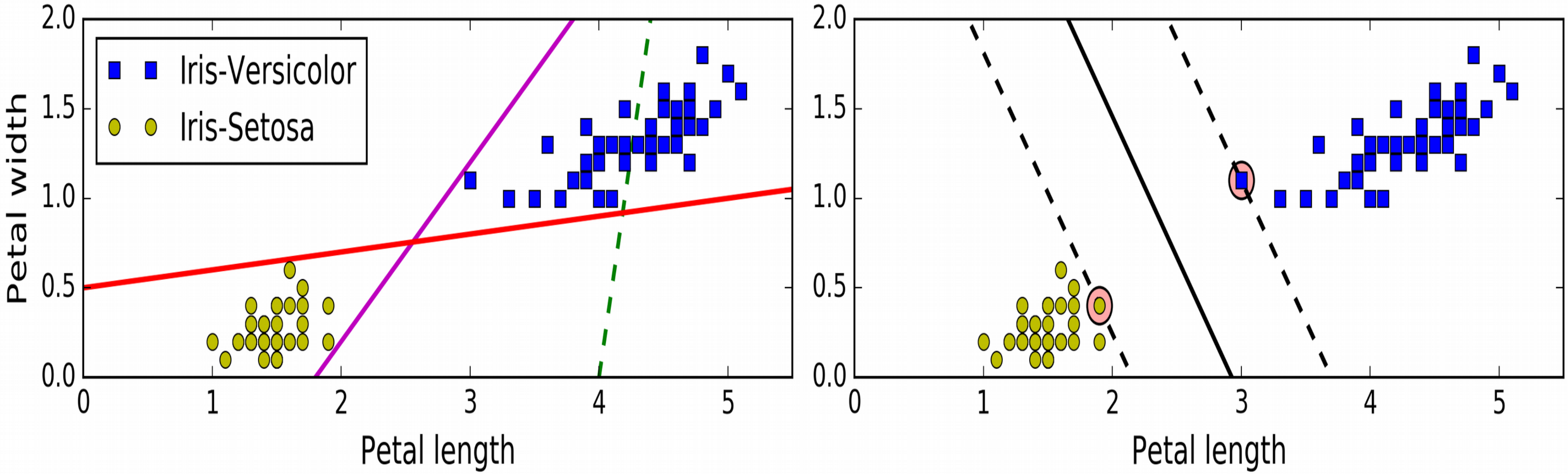
Support Vector Machine

Support Vector Machine

```
fl=fileloader("input")
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]

# SVM Classifier model
svm_clf = SVC(kernel="linear", C=float("inf"))
svm_clf.fit(X, y)
print("svm_clf:",svm_clf)
```

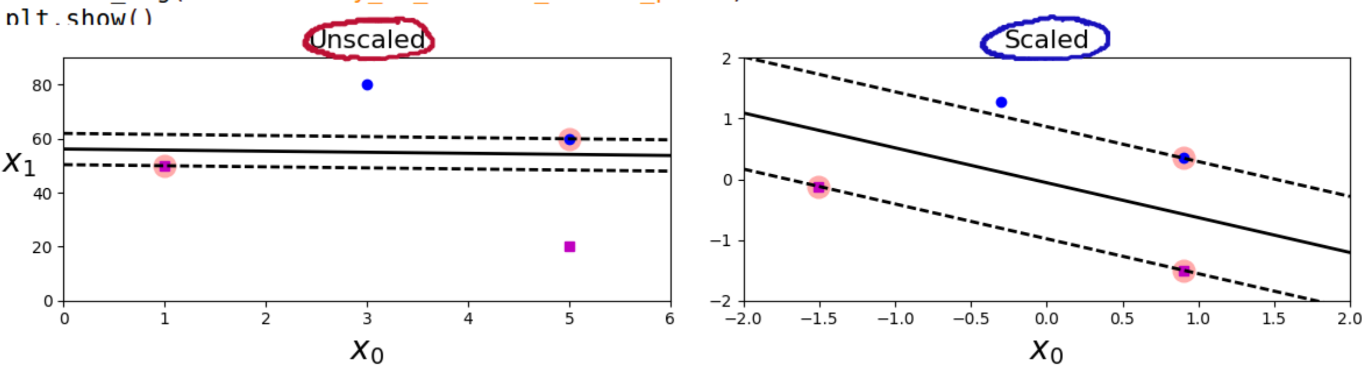
Support Vector Machine



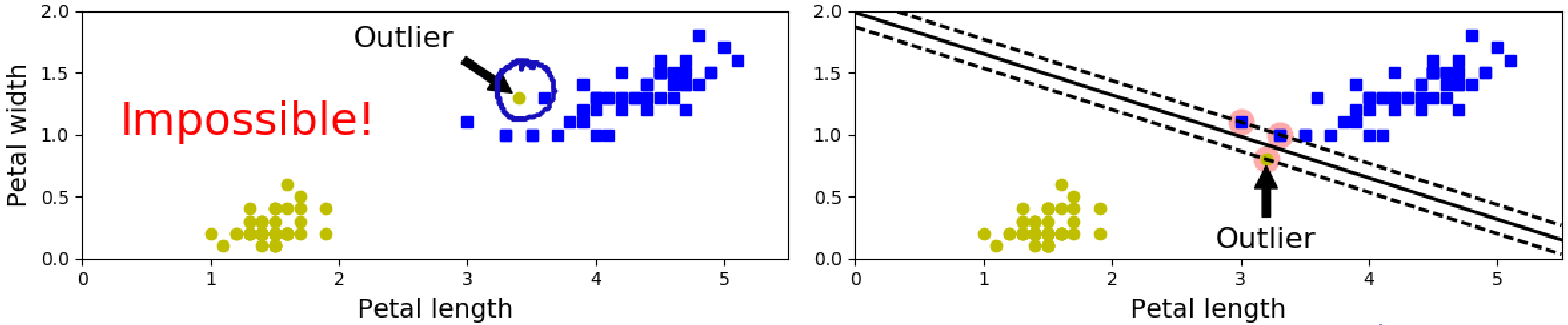
Support Vector Machine

```
fl=fileloader("input")
Xs = np.array([[1, 50], [5, 20], [3, 80], [5, 60]]).astype(np.float64)
ys = np.array([0, 0, 1, 1])
svm_clf = SVC(kernel="linear", C=100)
svm_clf.fit(Xs, ys)

plt.figure(figsize=(12,3.2))
plt.subplot(121)
plt.plot(Xs[:, 0][ys==1], Xs[:, 1][ys==1], "bo")
plt.plot(Xs[:, 0][ys==0], Xs[:, 1][ys==0], "ms")
plot_svc_decision_boundary(svm_clf, 0, 6)
plt.xlabel("$x_0$", fontsize=20)
plt.ylabel("$x_1$  ", fontsize=20, rotation=0)
plt.title("Unscaled", fontsize=16)
plt.axis([0, 6, 0, 90])


scaler = StandardScaler()
X_scaled = scaler.fit_transform(Xs)
svm_clf.fit(X_scaled, ys)

plt.subplot(122)
plt.plot(X_scaled[:, 0][ys==1], X_scaled[:, 1][ys==1], "bo")
plt.plot(X_scaled[:, 0][ys==0], X_scaled[:, 1][ys==0], "ms")
plot_svc_decision_boundary(svm_clf, -2, 2)
plt.xlabel("$x_0$", fontsize=20)
plt.title("Scaled", fontsize=16)
plt.axis([-2, 2, -2, 2])

fl.save_fig("sensitivity_to_feature_scales_plot")
plt.show()
```

- features in different scales
- SVM is quite sensitive to feature scales.
- The decision boundary looks much better with scaling.



Support Vector Machine:Hard Margin

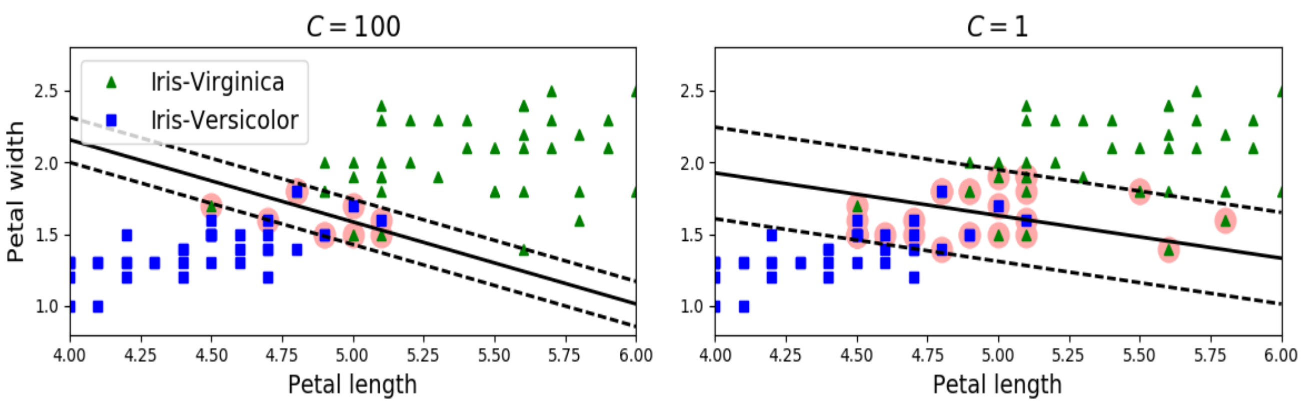


- Hard margin classification requires data to be linearly separable.
- It is also quite sensitive to outliers.

Support Vector Machine:Soft Margin

```python
scaler = StandardScaler()
svm_clf1 = LinearSVC(C=100, loss="hinge")
svm_clf2 = LinearSVC(C=1, loss="hinge")

scaled_svm_clf1 = Pipeline((
        ("scaler", scaler),
        ("linear_svc", svm_clf1),
    ))
scaled_svm_clf2 = Pipeline((
        ("scaler", scaler),
        ("linear_svc", svm_clf2),
    ))

scaled_svm_clf1.fit(X, y)
scaled_svm_clf2.fit(X, y)
```

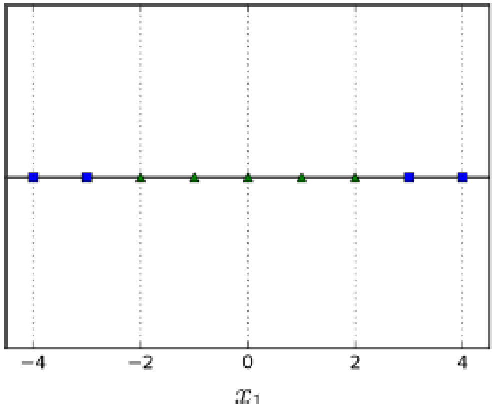
- The objective is to find a good balance between keeping the street as wide as possible and limiting margin violations.
- Recall that C is like $1/\lambda$. So lower values have the effect of regularization

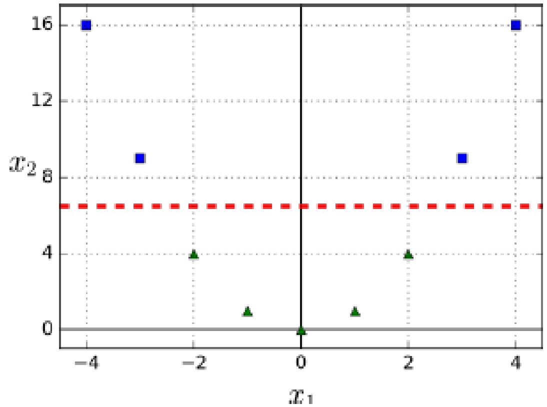


Support Vector Machine:Soft Margin

- Alternatively, you could use the SVC class, using SVC(kernel="linear", C=1), but it is much slower, especially with large training sets, so it is not recommended.
- Another option is to use the SGDClassifier class, with SGDClassifier(loss="hinge", alpha=1/(m*C)). This applies regular Stochastic Gradient Descent to train a linear SVM classifier.
  - It does not converge as fast as the LinearSVC class, but it can be useful to handle huge datasets that do not fit in memory (out-of-core training), or to handle online classification tasks.

Support Vector Machine:Non Linear

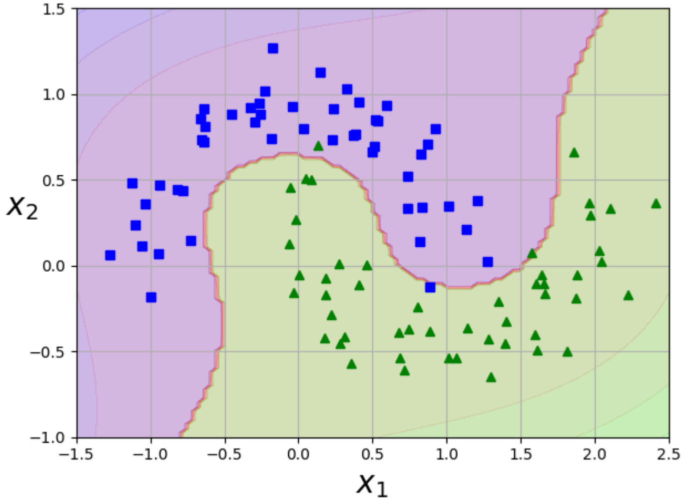




- linear SVM classifiers are good but many dataset are not even close to being linearly seperable.
- Often adding more features make it so.

Support Vector Machine:Non Linear

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

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

polynomial_svm_clf.fit(X, y)

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)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
```

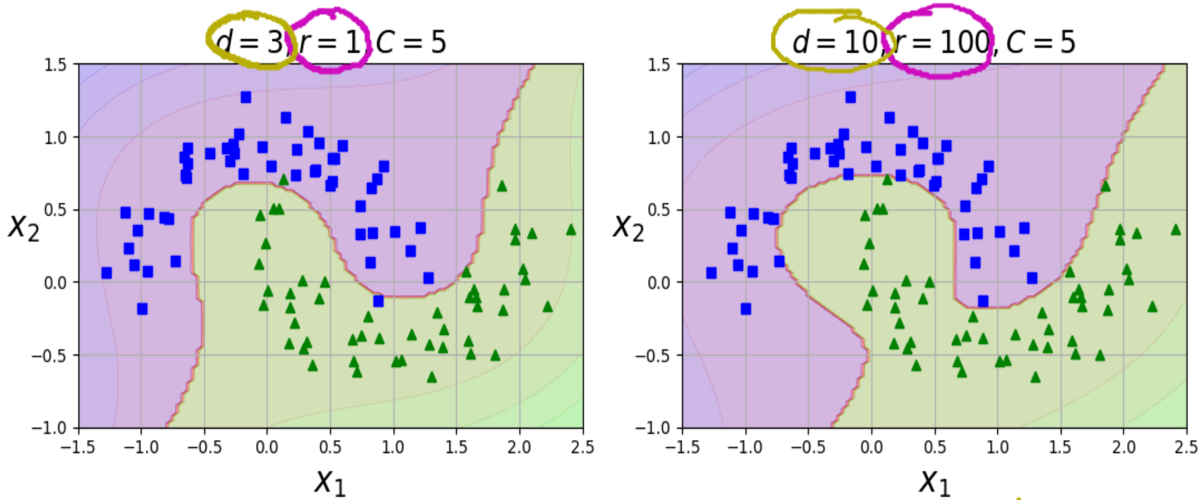


- Adding polynomial works great with most ML algorithms.
- Low degrees cannot deal with complex data set, higher degree creates huge number of features making the model too slow.

Support Vector Machine:Non Linear:Kernel

```
poly_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ))
poly100_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))
    ))

poly_kernel_svm_clf.fit(X, y)
poly100_kernel_svm_clf.fit(X, y)
```

- SVMs however can use a great mathematical trick called Kernel.
- It has the effect of high degree polynomial without the combinatorial explosion.
- r controls how much the model is effected by low and high degree polynomials.



- Increase or decrease depending on whether the model is underfitting or overfitting.
- Grid search to find the right balance.

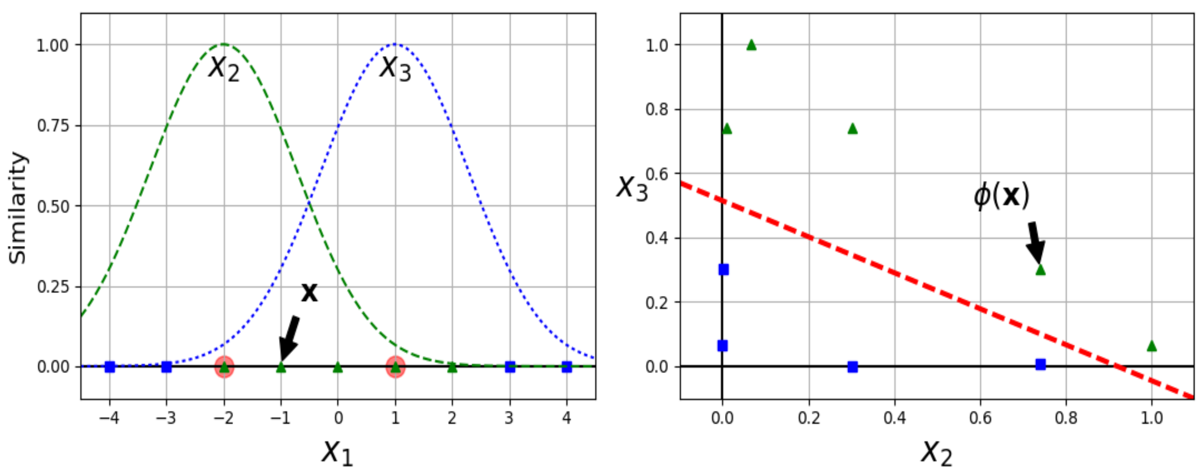
Support Vector Machine:Non Linear:Kernel

```
fl=fileloader("input")
X1D = np.linspace(-4, 4, 9).reshape(-1, 1)
X2D = np.c_[X1D, X1D**2]
y = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])
def gaussian_rbf(x, landmark, gamma):
    return np.exp(-gamma * np.linalg.norm(x - landmark, axis=1)**2)

gamma = 0.3

x1s = np.linspace(-4.5, 4.5, 200).reshape(-1, 1)
x2s = gaussian_rbf(x1s, -2, gamma)
x3s = gaussian_rbf(x1s, 1, gamma)

XK = np.c_[gaussian_rbf(X1D, -2, gamma), gaussian_rbf(X1D, 1, gamma)]
yk = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])
```

- $sim(x, l) = exp(-\lambda ||x - l||^2)$
- if $\lambda = 0.3$

$$x2 = e^{-(-0.3 \times 1^2)} = 0.74$$

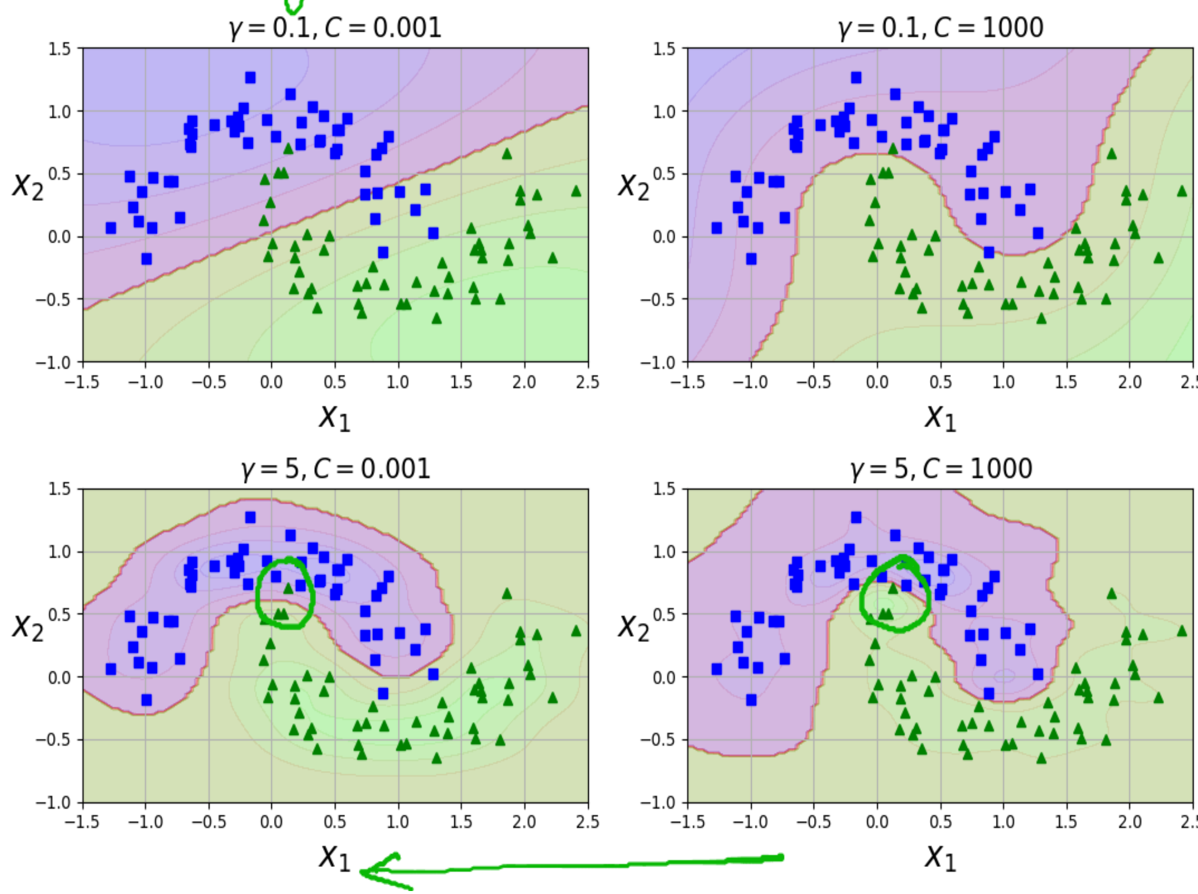
$$x_3 = e^{-(-0.3 \times 2^2)} = 0.3$$



Support Vector Machine:Non Linear:Kernel

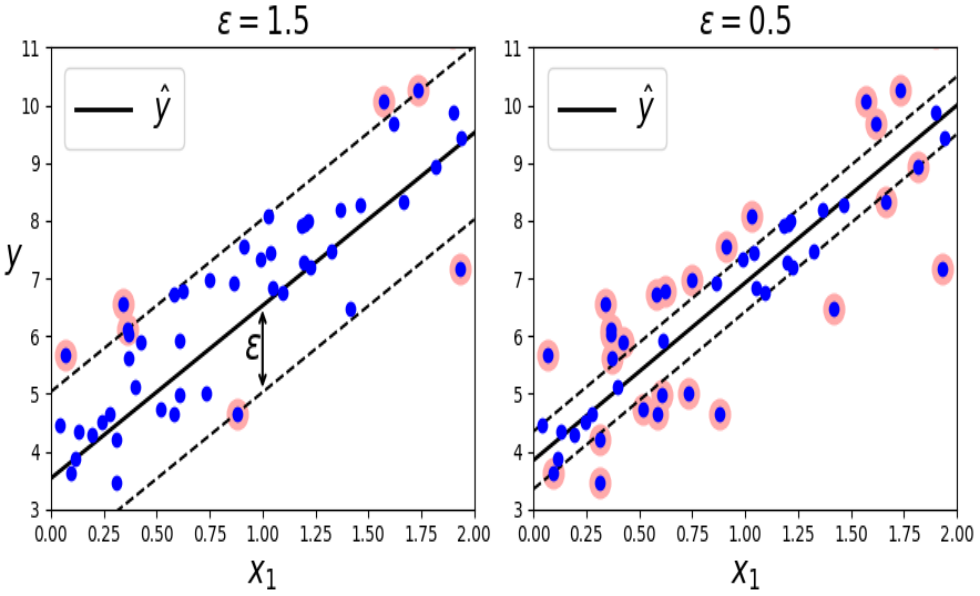
```python
gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline((
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ))
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)

plt.figure(figsize=(11, 7))

for i, svm_clf in enumerate(svm_clfs):
    plt.subplot(221 + i)
    plot_predictions(svm_clf, [-1.5, 2.5, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fonts
```

- higher λ would make the bell curve steeper.
- lower C values would have wider street at the cost of margin violations



Support Vector Machine:Regression

```
rnd.seed(42)
m = 50
X = 2 * rnd.rand(m, 1)
y = (4 + 3 * X + rnd.randn(m, 1)).ravel()

svm_reg1 = LinearSVR(epsilon=1.5)
svm_reg2 = LinearSVR(epsilon=0.5)
svm_reg1.fit(X, y)
svm_reg2.fit(X, y)
```

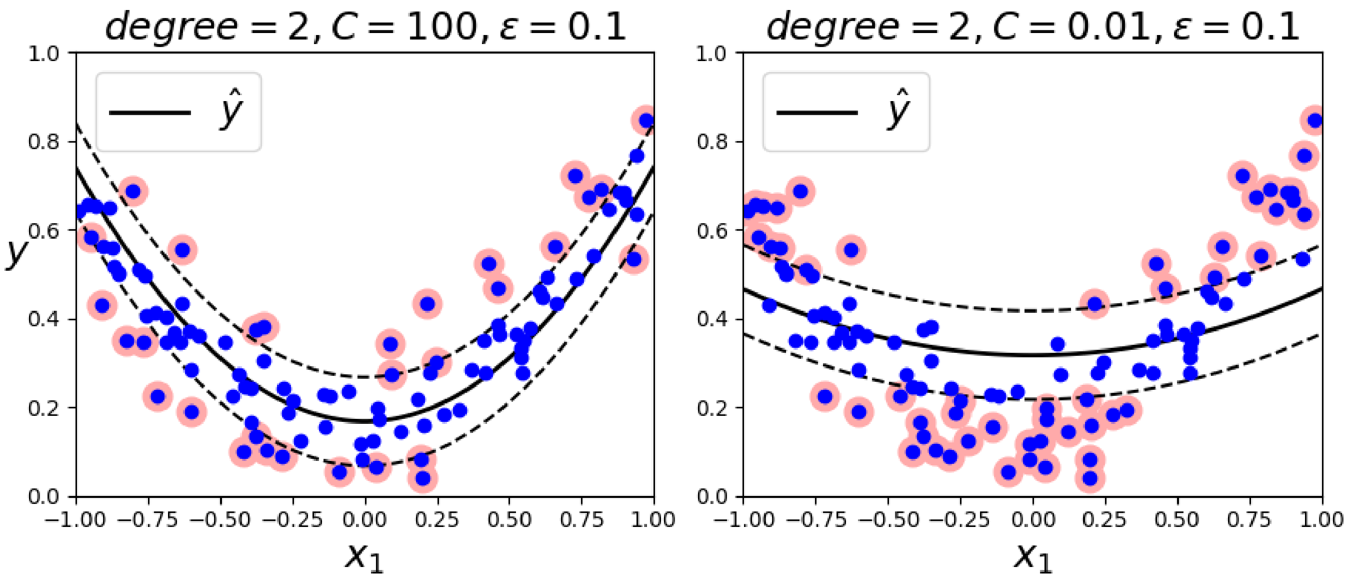


- Objective is reversed. Instead of fitting the widest possible street, while limiting margin violations, SVM Regression tries to fit as many instances on the street with as few margin violations.
- ϵ controls the width of the street.

Support Vector Machine:Regression

```
rnd.seed(42)
m = 100
X = 2 * rnd.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + rnd.randn(m, 1)/10).ravel()

svm_poly_reg1 = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg2 = SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1)
svm_poly_reg1.fit(X, y)
svm_poly_reg2.fit(X, y)
```

- SVR Kernel.



Support Vector Machine:Regression

```
X, y = make_moons(n_samples=1000, noise=0.4)
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
plt.show()


tol = 0.1
tols = []
times = []
for i in range(10):
    svm_clf = SVC(kernel="poly", gamma=3, C=10, tol=tol, verbose=1)
    t1 = time.time()
    svm_clf.fit(X, y)
    t2 = time.time()
    times.append(t2-t1)
    tols.append(tol)
    #print(i, tol, t2-t1)
    tol /= 10
plt.semilogx(tols, times)
plt.show()
```