Module 5: Support Vector Machines

Video: 670_mod5_vid1

Support Vector Machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression - they can even perform outlier detection. In this module, we will develop the intuition behind support vector machines and their use in classification problems, as well as regression problems. We begin with the standard imports:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# use Seaborn plotting defaults
import seaborn as sns; sns.set()
```

Motivation for Support Vector Machines

Here we will consider a process called ***discriminative classification***: rather than modeling each class directly, we simply find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

As an example of this, consider the simple case of a classification task, in which the two classes of points are well separated:

```
from sklearn.datasets.samples_generator import make_blobs
X, y = make_blobs(n_samples=50, centers=2, random_state=0,
     cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification. For two-dimensional data like that shown here, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes!

We can draw these lines using this code:

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
```

```
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2,
     markersize=10)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
     plt.plot(xfit, m * xfit + b, '-k')

plt.xlim(-1, 3.5);
```

These are three very different separators that all perfectly discriminate between these sample data. Depending on which you choose, a new data point (for example, the one marked by the "X") will be assigned a different label! Evidently our simple intuition of "drawing a line between classes" is not enough, and we need to think a bit deeper.

Video: 670_mod5_vid2

Support Vector Machines: Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a margin of some width, up to the nearest point. This code shows an example of how this might look:

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
     yfit = m * xfit + b
     plt.plot(xfit, yfit, '-k')
     plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
          color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);
```

In support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a ***maximum margin estimator****.*

Fitting a Support Vector Machine

Let's see the result of an actual fit to this data: we will use Scikit-Learn's support vector classifier to train an SVM model on this data. For the time being, we will use a ***linear kernel*** and set the C parameter to a very large number (we'll discuss the meaning of these in more depth momentarily):

```
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

To better visualize what's happening here, let's create a quick function for convenience that will plot SVM decision boundaries for us:

```
def plot_svc_decision_function(model, ax=None, plot_support=True):
      """Plot the decision function for a two-dimensional SVC"""
      if ax is None:
      ax = plt.gca()
      xlim = ax.get_xlim()
      ylim = ax.get_ylim()

      # create grid to evaluate model
      x = np.linspace(xlim[0], xlim[1], 30)
      y = np.linspace(ylim[0], ylim[1], 30)
      Y, X = np.meshgrid(y, x)
      xy = np.vstack([X.ravel(), Y.ravel()]).T
      P = model.decision_function(xy).reshape(X.shape)

      # plot decision boundary and margins
      ax.contour(X, Y, P, colors='k',
            levels=[-1, 0, 1], alpha=0.5,
            linestyles=['--', '-', '--'])

      # plot support vectors
      if plot_support:
            ax.scatter(model.support_vectors_[:, 0],
            model.support_vectors_[:, 1],
            s=300, linewidth=1, facecolors='none');
      ax.set_xlim(xlim)
      ax.set_ylim(ylim)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(model);
```

This is the dividing line that maximizes the margin between the two sets of points. Notice that a few of the training points just touch the margin; they are indicated by the black circles. These points are the pivotal elements of this fit, and are known as the ***support vectors***, and give the algorithm its name. In Scikit-Learn, the location of these points is stored in the support_vectors_ attribute of the classifier:

```
model.support_vectors_
```

A key to this classifier's success is that for the fit, only the position of the support vectors matters; any points further from the margin that are on the correct side do not modify the fit! Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

We can see this, for example, if we plot the model learned from the first 60 points and first 120 points of this dataset:

```
def plot_svm(N=10, ax=None):
X, y = make_blobs(n_samples=200, centers=2,
      random_state=0, cluster_std=0.60)

X = X[:N]
y = y[:N]
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)

ax = ax or plt.gca()
ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
ax.set_xlim(-1, 4)
ax.set_ylim(-1, 6)
plot_svc_decision_function(model, ax)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, N in zip(ax, [60, 120]):
      plot_svm(N, axi)
      axi.set_title('N = {0}'.format(N))
```

In the left panel, we see the model and the support vectors for 60 training points. In the right panel, we have doubled the number of training points, but the model has not changed: the three support vectors from the left panel are still the support vectors from the right panel. This insensitivity to the exact behavior of distant points is one of the strengths of the SVM model.

We can use IPython's interactive widgets to view this feature of the SVM model interactively:

```
from ipywidgets import interact, fixed
interact(plot_svm, N=[10, 200], ax=fixed(None));
```

SVMs are sensitive to the feature scales, as you can see in the following figure: in the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn's StandardScaler), the decision boundary in the right plot looks much better.

```
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)

def plot_svc_decision_boundary(svm_clf, xmin, xmax):
    w = svm_clf.coef_[0]
    b = svm_clf.intercept_[0]

    # At the decision boundary, w0*x0 + w1*x1 + b = 0
    # => x1 = -w0/w1 * x0 - b/w1
    x0 = np.linspace(xmin, xmax, 200)
    decision_boundary = -w[0]/w[1] * x0 - b/w[1]

    margin = 1/w[1]
    gutter_up = decision_boundary + margin
    gutter_down = decision_boundary - margin

    svs = svm_clf.support_vectors_
    plt.scatter(svs[:, 0], svs[:, 1], s=180, facecolors='#FFAAAA')
    plt.plot(x0, decision_boundary, "k-", linewidth=2)
    plt.plot(x0, gutter_up, "k--", linewidth=2)
    plt.plot(x0, gutter_down, "k--", linewidth=2)

plt.figure(figsize=(9,2.7))
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])

from sklearn.preprocessing import StandardScaler
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.ylabel("$x'_1$  ", fontsize=20, rotation=0)
plt.title("Scaled", fontsize=16)
plt.axis([-2, 2, -2, 2])

save_fig("sensitivity_to_feature_scales_plot")
```

Video: 670_mod5_vid3

Beyond Linear Boundaries: Kernel SVM

Where SVM becomes extremely powerful is when they are combined with kernels. We have seen a version of kernels before when dealing with polynomial regression, although we did not

use the "kernel" jargon then. With polynomial regression, we projected our data into a higher-dimensional space defined by polynomials, and thereby were able to fit for nonlinear relationships with a linear classifier.

In SVM models, we can use a version of the same idea. To motivate the need for kernels, let's look at some data that is not linearly separable:

```
from sklearn.datasets.samples_generator import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

clf = SVC(kernel='linear').fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf, plot_support=False);
```

It is clear that no linear discrimination will ever be able to separate this data. But we can draw a lesson from our discussion of polynomial regression, and think about how we might project the data into a higher dimension such that a linear separator would be sufficient. For example, one simple projection we could use would be to compute a ***radial basis function*** centered on the middle clump:

```
r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot:

```
from mpl_toolkits import mplot3d
def plot_3D(elev=30, azim=30, X=X, y=y):
     ax = plt.subplot(projection='3d')
     ax.scatter3D(X[:,   0],   X[:,   1],   r,   c=y,   s=50,
     cmap='autumn')
     ax.view_init(elev=elev, azim=azim)
     ax.set_xlabel('x')
     ax.set_ylabel('y')
     ax.set_zlabel('r')

     interact(plot_3D, elev=[-90, 90], azip=(-180, 180),
          X=fixed(X), y=fixed(y));
```

We can see that with this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, r=0.7.

Here we had to choose and carefully tune our projection; if we had not centered our radial basis function in the right location, we would not have seen such clean, linearly separable results. In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use. One strategy to this end is to compute a basis function centered at every point in the dataset, and let the SVM algorithm sift through the results. This type of basis function transformation is known as a ***kernel transformation***, as it is based on a ***similarity relationship*** (or ***kernel***) between each pair of points.

A potential problem with this strategy—projecting N points into N dimensions—is that it might become very computationally intensive as N grows large. However, because of a neat little procedure known as the ***kernel trick***, a fit on kernel-transformed data can be done implicitly—that is, without ever building the full N-dimensional representation of the kernel projection! This kernel trick is built into the SVM, and is one of the reasons the method is so powerful.

In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to a ***RBF (radial basis function) kernel***, using the kernel model hyperparameter:

```
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
      s=300, lw=1, facecolors='none');
```

Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used.

Video: 670_mod5_vid4

Tuning SVMs: Softening Margins

Our discussion so far has centered on very clean datasets, in which a perfect decision boundary exists. If two classes can clearly be separated easily with a straight line, then we say they are ***linearly separable***. But what if your data has some amount of overlap? For example, you may have data like this:

```
X, y = make_blobs(n_samples=100, centers=2,
      random_state=0, cluster_std=1.2)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

To handle this case, the SVM implementation has a bit of a fudge-factor that "softens" the margin; that is, it allows some of the points to creep into the margin if that allows a better fit. The hardness of the margin is controlled by a tuning parameter, most often known as ***the C hyperparameter***. For very large C, the margin is hard, and points cannot lie in it. For smaller C, the margin is softer, and can grow to encompass some points. The plot shown below gives a visual picture of how a changing C parameter affects the final fit, via the softening of the margin:

```
X, y = make_blobs(n_samples=100, centers=2,
      random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, C in zip(ax, [10.0, 0.1]):
      model = SVC(kernel='linear', C=C).fit(X, y)
      axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
      plot_svc_decision_function(model, axi)
      axi.scatter(model.support_vectors_[:, 0],
            model.support_vectors_[:, 1],
            s=300, lw=1, facecolors='none');
      axi.set_title('C = {0:.1f}'.format(C), size=14)
```

The optimal value of the C parameter will depend on your dataset, and should be tuned via cross-validation or a similar procedure.

You should understand the separating hyperplane generated by the SVM algorithm as separating the two classes while also staying as far away as possible from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street between the classes. This is called ***large margin classification***. If we strictly impose that all instances must be off the street and on the right side, this is called ***hard margin classification***. There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers.

Let's take a look at some more examples of nonlinear SVM classification to supplement the discussion we have already had.

Video: 670_mod5_vid5

Nonlinear SVM Classification

Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable, as we have already seen. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as you did in module 4); in some cases this can result in a linearly separable dataset. Consider the left plot in the following figure: it represents a simple dataset with just one feature, x1. This dataset is not linearly separable, as you can see. But if you add a second feature x2 = x1^2, the resulting 2D dataset is perfectly linearly separable.

```
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])

plt.figure(figsize=(10, 3))

plt.subplot(121)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.plot(X1D[:, 0][y==0], np.zeros(4), "bs")
plt.plot(X1D[:, 0][y==1], np.zeros(5), "g^")
plt.gca().get_yaxis().set_ticks([])
plt.xlabel(r"$x_1$", fontsize=20)
plt.axis([-4.5, 4.5, -0.2, 0.2])

plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(X2D[:, 0][y==0], X2D[:, 1][y==0], "bs")
plt.plot(X2D[:, 0][y==1], X2D[:, 1][y==1], "g^")
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"$x_2$  ", fontsize=20, rotation=0)
plt.gca().get_yaxis().set_ticks([0, 4, 8, 12, 16])
plt.plot([-4.5, 4.5], [6.5, 6.5], "r--", linewidth=3)
plt.axis([-4.5, 4.5, -1, 17])

plt.subplots_adjust(right=1)

save_fig("higher_dimensions_plot", tight_layout=False)
plt.show()
```

Linear Kernels with Polynomial Features

To implement this idea using Scikit-Learn, let's create a Pipeline containing a PolynomialFeatures transformer, followed by a StandardScaler, and a LinearSVC. Let's test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles. You can generate this dataset using the make_moons() function:

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

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

Next, we can train a linear SVC on our dataset that we extended with PolynomialFeatures:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
("poly_features", PolynomialFeatures(degree=3)),
("scaler", StandardScaler()),
("svm_clf", LinearSVC(C=10, loss="hinge"))])
polynomial_svm_clf.fit(X, y)
```

Let's plot the decision surface:

```
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])

save_fig("moons_polynomial_svc_plot")
plt.show()
```

Video: 670_mod5_vid6

Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow. Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the ***kernel trick*** (explained in a moment). 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. So there is no combinatorial explosion of the number of features because you don't actually add any features. This trick is implemented by the SVC class. Let's test it on the moons dataset using two different SCV models - each having a different polynomial degree:

```
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)

-----------------------------------------------------------------
-

poly100_kernel_svm_clf = Pipeline([
         ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="poly", degree=10, coef0=100,
C=5))
    ])
poly100_kernel_svm_clf.fit(X, y)

-----------------------------------------------------------------
-
```

```
fig, axes = plt.subplots(ncols=2, figsize=(10.5, 4),
sharey=True)

plt.sca(axes[0])
plot_predictions(poly_kernel_svm_clf, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.4, -1, 1.5])
plt.title(r"$d=3, r=1, C=5$", fontsize=18)

plt.sca(axes[1])
plot_predictions(poly100_kernel_svm_clf, [-1.5, 2.45, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.4, -1, 1.5])
plt.title(r"$d=10, r=100, C=5$", fontsize=18)
plt.ylabel("")

save_fig("moons_kernelized_polynomial_svc_plot")
plt.show()
```

This code trains an SVM classifier using a third-degree polynomial kernel. It is represented on the left in the figure. On the right is another SVM classifier using a 10thdegree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter coef0 controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

A common approach to finding the right hyperparameter values is to use grid search (which we discussed in module 2). It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good sense of what each hyperparameter actually does can also help you search in the right part of the hyperparameter space.

In short, the C hyperparameter controls the size of the margins, or the width of the street. This means, in some sense, that it also dictates how much error you want - meaning how many margin errors are allowed. Small C means wider, softer margins. Large C means tighter, harder margins. The gamma hyperparameter is used only for the RBF kernel (not the linear or polynomial kernels), and dictates how much curvature you want in the decision boundary. High gamma means more curvature. Low gamma means less curvature. Actually, the gamma hyperparameter is used for linear kernels and polynomial kernels, but it is merely a normalizing term.

Gaussian RBF Kernel

Below is the equation for the ***Gaussian Radial Basis Function (RBF)***:

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

Let's try the SVC class with the Gaussian RBF kernel:

```
rbf_kernel_svm_clf = Pipeline([
      ("scaler", StandardScaler()),
      ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))])
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented at the bottom left in the following figure.

```
from sklearn.svm import SVC

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)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10.5, 7), sharex=True, sharey=True)

for i, svm_clf in enumerate(svm_clfs):
    plt.sca(axes[i // 2, i % 2])
    plot_predictions(svm_clf, [-1.5, 2.45, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.45, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fontsize=16)
    if i in (0, 1):
        plt.xlabel("")
    if i in (1, 3):
        plt.ylabel("")

save_fig("moons_rbf_svc_plot")
plt.show()
```

The other plots show models trained with different values of hyperparameters gamma (γ) and C. Increasing gamma makes the bell-shaped curve narrower (see the left-hand plots in this figure). As a result, each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small gamma value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization hyperparameter: if your model is overfitting, you should reduce it; if it is underfitting, you should increase it (similar to the C hyperparameter).

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 (remember that LinearSVC is much faster than SVC(kernel="linear")), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should also try the Gaussian RBF kernel; it works well in most cases. Then if you have spare time and computing power, you can experiment with a few other kernels, using cross-validation and grid search. You'd want to experiment like that especially if there are kernels specialized for your training set's data structure.

Below is a summary of the different Scikit-Learn classes that can be used to perform SVM classification:

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

Video: 670_mod5_vid7

SVM Regression

As mentioned earlier, the SVM algorithm is versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. To use SVMs for regression instead of classification, the trick is to reverse 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 (in the case of regression, margin violations are instances being off the street). The width of the street is controlled by a hyperparameter, ϵ. The following figure shows two linear SVM Regression models trained on some random linear data, one with a large margin ($\epsilon = 1.5$) and the other with a small margin ($\epsilon = 0.5$).

Adding more training instances within the margin does not affect the model's predictions; thus, the model is said to be ϵ-insensitive.

Let's begin by generating some data to train an SVM on:

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

You can use Scikit-Learn's LinearSVR class to perform linear SVM Regression. The following code produces the model represented in the below figure on the left (the training data should be scaled and centered first):

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

Next, we can plot this model and compare it to another LinearSVR model with an epsilon value of 0.5:

```
svm_reg1 = LinearSVR(epsilon=1.5, random_state=42)
svm_reg2 = LinearSVR(epsilon=0.5, random_state=42)
svm_reg1.fit(X, y)
svm_reg2.fit(X, y)

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_reg1.support_ = find_support_vectors(svm_reg1, X, y)
svm_reg2.support_ = find_support_vectors(svm_reg2, X, y)

eps_x1 = 1
eps_y_pred = svm_reg1.predict([[eps_x1]])

------------------------------------------------------------------
--

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)

fig, axes = plt.subplots(ncols=2, figsize=(9, 4), sharey=True)
plt.sca(axes[0])
plot_svm_regression(svm_reg1, X, y, [0, 2, 3, 11])
```

```
plt.title(r"$\epsilon = {}$".format(svm_reg1.epsilon), fontsize=18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
#plt.plot([eps_x1, eps_x1], [eps_y_pred, eps_y_pred - svm_reg1.epsilon], "k-",
linewidth=2)
plt.annotate(
        '', xy=(eps_x1, eps_y_pred), xycoords='data',
        xytext=(eps_x1, eps_y_pred - svm_reg1.epsilon),
        textcoords='data', arrowprops={'arrowstyle': '<->', 'linewidth': 1.5}
    )
plt.text(0.91, 5.6, r"$\epsilon$", fontsize=20)
plt.sca(axes[1])
plot_svm_regression(svm_reg2, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg2.epsilon), fontsize=18)
save_fig("svm_regression_plot")
plt.show()
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. The following figure shows SVM Regression on a random quadratic training set, using a second-degree polynomial kernel. There is little regularization in the left plot (i.e., a large C value), and much more regularization in the right plot (i.e., a small C value).

The following code uses Scikit-Learn's SVR class (which supports the kernel trick) to produce the model represented in the figure on the left and right:

```
from sklearn.svm import SVR

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

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

Lastly, SVMs can also be used for outlier detection, but we will not discuss that here. You should see Scikit-Learn's documentation for more details.