Chapter 5: Support Vector Machines

Warren Alphonso

March 2019

1 Linear SVM Classification

The decision boundary of a Support Vector Machine classifier not only separates the two classes but also stays as far away from the closest training instances as possible. Think of an SVM classifier as fitting the widest possible street between the classes. This is called *large margin classification*.
Note that adding more training instances "off the street" will not affect the decision boundary: it is **fully determined by instances located on the edge** of the street. These instances are called the *support vectors*.
SVMs are sensitive to feature scales. After feature scaling (again using Scikit-Learn's `StandardScaler`), the decision boundary looks much better.

1.1 Soft Margin Classification

If we strictly impose that all instances be off the street and on the correct side, this is called *hard margin classification*. There are two main issues with hard margin classification:

1. It only works if the data is linearly separable.
2. It is sensitive to outliers.

To avoid these issues, it is preferable to use a more flexible model. We must find a balance between keeping the street as large as possible and limiting the *margin violations*. This is called *soft margin classification*.
In Scikit-Learn's SVM classes, we can control this balance using the `C` hyperparameter: a smaller `C` value leads to a wider street but more margin violations. Using a high `C` value leads to a smaller margin and fewer margin violations. If an SVM model is overfitting, reducing `C` might help.

We can train a linear SVM model using the *hinge* loss function to detect Iris=Virginica flowers:

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris-Virginica

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge", random_state=42)),
    ])

svm_clf.fit(X, y)
```

Alternatively, we can use the `SVC` class, using `SVC(kernel="linear", C=1)`, but it is much slower with large training sets. 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 quickly as the `LinearSVC` class, but it can be useful to handle huge datasets.

2 Nonlinear SVM classification

One approach to handling nonlinear datasets is to add more features, such as polynomial features. Adding a second feature $x_2 = (x_1)^2$ may cause the resulting quadratic graph to be linearly separable.
To implement this idea in Scikit-Learn, we can create a `Pipeline` containing a `PolynomialFeatures` transformer, followed by a `StandardScaler` and a `LinearSVC`.

2.1 Polynomial Kernel

Adding polynomial features is simple to implement and can work great with many ML algorithms, but a low polynomial degree cannot deal with very complex datasets and a high polynomial degree makes the model too slow.
When using SVMs, we can apply a mathematical technique called the *kernel trick* (explained later). It makes it possible to get the same result as if you added many polynomial features, without actually having to add them. This means we avoid the combinatorial explosion of number of features. Implementation in Sci-kit Learn:

```
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)
```

The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

2.2 Adding Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a *similarity function* that measures how each instance resembles a particular *landmark*. For example, using a one-dimensional dataset, we add two landmarks at $x_1 = -2$ and $x_2 = 1$. Let's define the similarity function to be the Gaussian *Radial Basis Function* with $\gamma = 0.3$.

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

Figure 1: Gaussian RBF

It is a bell-shaped function varying from 0 (very far from landmark) to 1 (at the landmark). To compute the new features, we look at the instance $x_1 = -1$: it is located a distance of 1 from first landmark and distance of 2 from second landmark. Therefore, its new features are $x_2 = exp(-0.3 * 1^2) \approx 0.74$ and $x_3 = exp(-0.3 * 2^2) \approx 0.30$. These become the x and y values, respectively, of the transformed dataset, which is now linearly separable.

When deciding how to select landmarks, the simplest approach is to create a landmark at 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. If your training set is very large, you end up with equally large number of features.

3 SVM Regression

The SVM algorithm is versatile: it support linear and nonlinear classification, but it also supports linear and nonlinear regression. 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 (instances *off* the street).

We can use Scikit-Learn's `LinearSVR` class to perform linear SVM Regression:

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5, random_state=42)
svm_reg.fit(X, y)
```

4 Under the Hood

This section deals with the math behind a lot of the tricks used in earlier parts of this chapter. Here we will use a different convention: the bias term will be called b and the feature weights vector will be called w. No bias feature will be added to the input feature vectors.

4.1 Decision Functions and Predictions

The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function $w^Tx + b$. If the result is positive, the predicted class $\hat{y}$ si the positive class (1), else it is the negative class (0).

$$\hat{y} = \begin{cases} 0 & \text{if } w^Tx + b < 0, \\ 1 & \text{if } w^Tx + b \geq 0 \end{cases}$$

Figure 2: Linear SVM classifier prediction

The decision function is $h = w^Tx + b$. When graphing the points being classified, the *decision boundary* is the set of points where the decision function is equal to 0.

4.2 Training Objective

Note that the slope of the decision function is equal to the norm of the weight vector, $|| \ w \ ||$. If we divide this slope by 2, the points where the decision function is equal to ±1 are going to be twice as far away from the decision boundary. Dividing slope by 2 will multiply the margin by 2.

Thus, we want to minimize $|| \ w \ ||$ to get a large margin, but also want to avoid many margin violations.

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 i$^{\text{th}}$ instance is allowed to violate the margin. Note that we now have two conflicting objectives: making the slack variables as small as possible to reduce margin violations and making $\frac{1}{2}w^Tw$ as small as possible to increase the margin. This is where the `C` hyperparameter comes in: it allows us to define the trade-off between these two.

$$\begin{cases} \underset{w,b,\zeta}{\text{minimize}} \ \frac{1}{2}w^Tw + C\sum_{i=1}^{m}\zeta^{(i)} \\ \text{subject to } t^{(i)}(w^Tx^{(i)}+b) \geq 1-\zeta^{(i)} \text{ and } \zeta^{(i)} \geq 0 \text{ for } i = 1,2,...,m \end{cases}$$

Figure 3: Soft margin linear SVM classifier objective

We are minimizing $\frac{1}{2}w^Tw$, which is equal to $\frac{1}{2} \mid\mid w \mid\mid^2$, rather than minimizing $\mid\mid w \mid\mid$. This is because the former has a simple derivative (it is just w), while the latter is not differentiable at $w = 0$. Optimization algorithms generally work much better on differentiable functions.

4.3 Kernelized SVM

Suppose we want to apply a 2$^{\text{nd}}$-degree polynomial transformation to a two-dimensional training set, then train a linear SVM classifier on the transformed training set.

$$\phi(x) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Note the transformed vector is three-dimensional. Now we apply this 2$^{\text{nd}}$-degree polynomial mapping to two vectors a and b, and then we compute the dot product.

$$\begin{aligned} \phi(a)^T\phi(b) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} \\ &= a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 = (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 \\ &= (a^Tb)^2 \end{aligned}$$

Figure 4: Kernel trick for a 2$^{\text{nd}}$-degree polynomial mapping

This means the dot product of the transformed vectors is **equal to the square of the dot product of the original vectors**: $\phi(a)^T\phi(b) = (a^Tb)^2$.

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

Figure 5: Dual form of the linear SVM objective

- $\alpha^i \geq 0$ for all i
- t is 1 for all positive training instances and -1 for all negative instances.

This means if we apply the transformation ϕ to all training instances, then the dual problem will contain the dot product $\phi(x^{(i)})^T\phi(x^{(j)}$. But if ϕ is the 2nd-degree polynomial transformation defined above, then we can replace this dot product of transformed vectors simply by $((x^{(i)})^T(x^{(j)}))^2$. Thus, we do not actually need to apply the transformation since we get the same result by replacing the dot product by its square in the dual problem. This makes the whole process much more computationally efficient.

Generally, the function $K(a, b) = (a^T b)^2$ is called the 2nd-degree *polynomial kernel*. In ML, a *kernel* is a function capable of computing the dot product $\phi(a)^T\phi(b)$ based only on the original vectors a and b, without having to compute the transformation ϕ.

Common kernels:

- *Linear*: $K(a, b) = a^T b$
- *Polynomial*: $K(a, b) = (\gamma a^T b + r)^d$
- *Gaussian RBF*: $K(a, b) = exp(-\gamma \mid\mid a - b \mid\mid^2)$
- *Sigmoid*: $K(a, b) = tanh(\gamma a^T b + r)$