Lecture 18

In this lecture we return to the task of classification. As seen earlier, examples include spam filters, letter recognition, or text classification. In this lecture we introduce a popular method for classification, **Support Vector Machines (SVMs)**, from the point of view of convex optimization.

18.1 Linear Support Vector Machines

In the simplest case there is a set of labels $\mathcal{Y} = \{-1, 1\}$ and the set of training points $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ is *linearly separable*: this means that there exists an affine hyperplane $h(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ such that $h(\boldsymbol{x}_i) > 0$ if $y_i = 1$ and $h(\boldsymbol{x}_j) < 0$ if $y_j = -1$. We call the points for which $y_i = 1$ *positive*, and the ones for which $y_j = -1$ *negative*. The problem of finding such a hyperplane can be posed as a linear programming feasibility problem as follows: we look for a vector of *weights* $\boldsymbol{w}$ and a *bias term* b (together a $(p+1)$-dimensional vector) such that

$$\boldsymbol{w}^\top \boldsymbol{x}_i + b \geq 1, \text{ for } y_i = 1, \quad \boldsymbol{w}^\top \boldsymbol{x}_j + b \leq -1, \text{ for } y_j = -1.$$

Note that we can replace the $+1$ and -1 with any other positve or negative quantity by rescaling the $\boldsymbol{w}$ and $\boldsymbol{b}$, so this is just convention. We can also describe the two inequalities concisely as

$$y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) - 1 \geq 0. \tag{18.1}$$

A hyperplane separating the two point sets will in general not be unique. As we want to use the linear classifier on new, yet unknown data, we want to find a separating hyperplane with best possible **margin**. Let d_+ and d_- denote the distance of a separating hyperplane to the closest positive and closest negative point, respectively. The quantity $d = d_+ + d_-$ is then called the margin or the classifier, and we want to find a hyperplane with largest possible margin.

We next show that the margin for a separating hyperplane that satisfies (18.1) is $d = 2/\|\boldsymbol{w}\|_2$. Given a hyperplane H described in (18.1) and a point $\boldsymbol{x}$ such that we have the equality $\boldsymbol{w}^\top \boldsymbol{x} + b = 1$ (the point is as close as possible to the hyperplane, also called a **support vector**), the distance of that point to the hyperplane can be computed by first taking the difference of $\boldsymbol{x}$ with a point $\boldsymbol{p}$ on H (an *anchor*), and

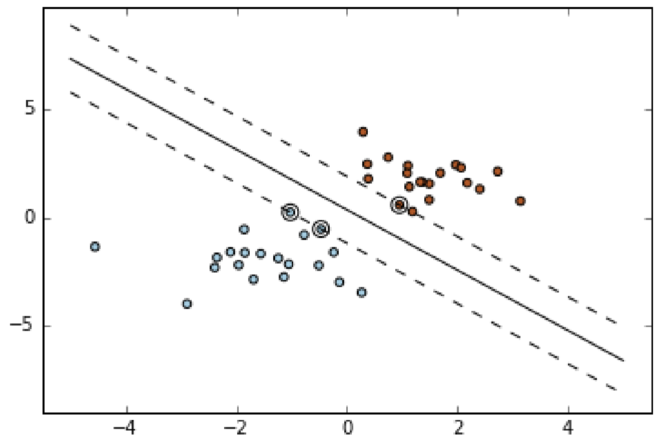


Figure 18.1: A hyperplane separating two sets of points with margin and support vectors.

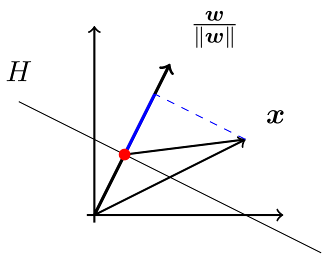


Figure 18.2: Computing the distance to the hyperplane

then computing the dot product of $\boldsymbol{x}-\boldsymbol{p}$ with the unit vector $\boldsymbol{w}/\|\boldsymbol{w}\|$ orthogonal to H (see Calculus and Vectors A, math10121).

As anchor point $\boldsymbol{p}$ we can just choose a multiple $c\boldsymbol{w}$ that is on the plane, i.e., that satisfies $\langle \boldsymbol{w}, c\boldsymbol{w}\rangle + b = 0$. This implies that $c = -b/\|\boldsymbol{w}\|^2$, and consequently $\boldsymbol{p} = -(b/\|\boldsymbol{w}\|^2)\boldsymbol{w}$. The distance is then

$$\begin{aligned} d_+ &= \langle \boldsymbol{x} + \frac{b}{\|\boldsymbol{w}\|^2}\boldsymbol{w}, \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\rangle = \frac{\langle \boldsymbol{x}, \boldsymbol{w}\rangle}{\|\boldsymbol{w}\|} + \frac{b}{\|\boldsymbol{w}\|^2}\langle \boldsymbol{w}, \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\rangle \\ &= \frac{1-b}{\|\boldsymbol{w}\|} + \frac{b}{\|\boldsymbol{w}\|} = \frac{1}{\|\boldsymbol{w}\|}. \end{aligned}$$

Similarly, we get $d_- = 1/\|\boldsymbol{w}\|$. The margin of this particular separating hyperplane is thus $d = 2/\|\boldsymbol{w}\|$. If we want to find a hyperplane with *largest* margin, we thus have to solve the quadratic optimization problem

$$\begin{aligned}
&\text{minimize} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 \\
&\text{subject to} \quad y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) - 1 \geq 0, \quad 1 \leq i \leq n.
\end{aligned}$$

Note that b is also an unknown variable in this problem! The factor $1/2$ in the objective function is just to make the gradient look nicer. The Lagrangian of this problem is

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda}) &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^m \lambda_i y_i \boldsymbol{w}^\top \boldsymbol{x}_i - \lambda_i y_i b + \lambda_i \\
&= \frac{1}{2}\boldsymbol{w}^\top \boldsymbol{w} - \boldsymbol{\lambda}^\top \boldsymbol{X}\boldsymbol{w} - b\boldsymbol{\lambda}^\top \boldsymbol{y} + \sum_{i=1}^m \lambda_i,
\end{aligned}$$

where we denote by $\boldsymbol{X}$ the matrix with the $y_i \boldsymbol{x}_i^\top$ as rows. We can then write the conditions on the gradient with respect to $\boldsymbol{w}$ and b of the Lagrangian as

$$\begin{aligned}
\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda}) &= \boldsymbol{w} - \boldsymbol{X}^\top \boldsymbol{\lambda} = \boldsymbol{0} \\
\frac{\partial \mathcal{L}}{\partial b}(\boldsymbol{w}, b, \boldsymbol{\lambda}) &= \boldsymbol{y}^\top \boldsymbol{\lambda} = 0.
\end{aligned} \tag{18.2}$$

Replacing $\boldsymbol{w}$ by $\boldsymbol{X}^\top \boldsymbol{\lambda}$ and $\boldsymbol{\lambda}^\top \boldsymbol{y}$ by 0 in the Lagrangian function then gives the expression for the Lagrange dual $g(\boldsymbol{\lambda})$,

$$g(\boldsymbol{\lambda}) = -\frac{1}{2}\boldsymbol{\lambda}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{\lambda} + \sum_{i=1}^m \lambda_i.$$

Finally, changing the sign and the maximum with a minimum, we can formulate the Lagrange dual optimization problem as

$$\text{minimize} \quad \frac{1}{2}\boldsymbol{\lambda}^\top \boldsymbol{X}\boldsymbol{X}^\top \boldsymbol{\lambda} - \boldsymbol{\lambda}^\top \boldsymbol{e} \quad \text{subject to} \quad \boldsymbol{\lambda} \geq \boldsymbol{0}, \tag{18.3}$$

where $\boldsymbol{e}$ is the vector of all ones.

Note that there is one dual variable λ_i per data point $\boldsymbol{x}_i$. We can find the optimal value by solving the dual problem (18.3), but that does not give us automatically the weights $\boldsymbol{w}$ and the bias b. We can find the weights by $\boldsymbol{w} = \boldsymbol{X}^\top \boldsymbol{\lambda}$. As for b, this is best determined from the KKT conditions of the problem. These can be written by combining the constraints of the primal problem with the conditions on the gradient of the Lagrangian (18.2), the condition $\boldsymbol{\lambda} \geq \boldsymbol{0}$, and complementary slackness as

$$\begin{aligned}
\boldsymbol{X}\boldsymbol{w} + b\boldsymbol{y} - \boldsymbol{e} &= \boldsymbol{0} \\
\boldsymbol{\lambda} &\geq \boldsymbol{0} \\
\lambda_i(1 - y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)) &= 0 \text{ for } 1 \leq i \leq n \\
\boldsymbol{w} - \boldsymbol{X}^\top \boldsymbol{\lambda} &= \boldsymbol{0} \\
\boldsymbol{y}^\top \boldsymbol{\lambda} &= 0.
\end{aligned}$$

To get b, we can choose one of the equations in which $\lambda_i \neq 0$, and then find b by setting $b = y_i(1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i)$. With the KKT conditions written down, we can go about solving the problem of finding a maximum margin linear classifier using methods such as the barrier method.

18.2 Extensions

So far we looked at the particularly simple case where (a) the data falls into two classes, (b) the points can actually be well separated, and (c) they can be separated by an affine hyperplane. In reality, these three assumptions may not hold. We briefly discuss extensions of the basic model to account for the three situations just mentioned.

Non-exact separation

What happens when the data can not be separated by a hyperplane? In this case the constraints can not be satisfied: there is no feasible solution to the problem. We can still modify the problem to allow for *misclassification*: we want to find a hyperplane that separates the two point sets as good as possible, but we allow for some mistakes.

One approach is to add an additional set of n *slack variables* $s_1, \dots, s_n$, and modify the constraints to

$$\boldsymbol{w}^\top \boldsymbol{x}_i + b \geq 1 - s_i, \text{ for } y_i = 1, \quad \boldsymbol{w}^\top \boldsymbol{x}_j + b \leq -1 + s_j, \text{ for } y_j = -1, \quad s_i \geq 0.$$

The i-th data point can land on the wrong side of the hyperplane if $s_i > 1$, and consequently the sum $\sum_{i=1}^n s_i$ is an upper bound on the number of errors possible. If we want to minimize the number of misclassified points, we may want to minimize this upper bound, so a sensible choice for objective function would be to add a multiple of this sum. The new problem thus becomes

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + \mu \sum_{j=1}^n s_j \\ \text{subject to} \quad & y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) - 1 + s_i \geq 0, \quad 1 \leq i \leq n \\ & s_i \geq 0, \quad 1 \leq i \leq n, \end{aligned}$$

for some parameter μ. The Lagrangian of this problem and the KKT conditions can be derived in a similar way as in the separable case and are left as an exercise.

Non-linear separation and kernels

The key to extending SVMs from linear to non-linear separation is the observation that the dual form of the optimization problem (18.3) depends only on the dot products $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$ of the data points. In fact, the (i, j)-th entry of the matrix $\boldsymbol{X}\boldsymbol{X}^\top$ is precisely $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$!

If we map our data into a higher (possibly infinite) dimensional space $\mathcal{H}$,

$$\varphi \colon \mathbb{R}^p \to \mathcal{H},$$

and consider the data points $\varphi(\boldsymbol{x}_i)$, $1 \leq i \leq n$, then applying the support vector machine to these higher dimensional vectors will only depend on the dot products

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \varphi(\boldsymbol{x}_i), \varphi(\boldsymbol{x}_j) \rangle.$$

The function K is called a **kernel function**. A typical example, often used in practice, is the Gaussian radial basis function (RBF),

$$K(\boldsymbol{x}, \boldsymbol{y}) = e^{-\|\boldsymbol{x}-\boldsymbol{y}\|^2/2\sigma^2}.$$

Note that we *don't need to know how the function φ looks like*! In the equation for the hyperplane we simply replace $\boldsymbol{w}^\top \boldsymbol{x}$ with $K(\boldsymbol{w}, \boldsymbol{x})$. The only difference now is that the function ceases to be linear in $\boldsymbol{x}$: we get a non-linear decision boundary.

Multiple classes

One is often interested in classifying data into more than two classes. There are two simple ways in which support vector machines can be extended for such problems: one-vs-one and one-vs-rest. In the one-vs-one case, given k classes, we train one classifier for each pair of classes in the training data, obtaining a total of $k(k-1)/2$ classifiers. When it comes to prediction, we apply each of the classifiers to our test data and choose the class that was chosen the most among all the classifiers. In the one-vs-rest approach, each train k binary classifiers: in each one, one class corresponds to a chosen class, and the second class corresponds to the rest. By associating confidence scores to each classifier, we choose the one with the highest confidence score.

Example 18.1. An example that uses all three extensions mentioned is handwritten digit recognition. Suppose we have a series of pixels, each representing a number, and associated labels $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. We would like to train a support vector machine to recognize new digits. Given the knowledge we have, we can implement this task using standard optimization software such as CVXPY. Luckily, there are packages that have this functionality already implemented, such as the SCIKIT-LEARN package for Python. We illustrate its functioning below. The code also illustrates some standard procedures when tackling a machine learning problem:

- **Separate** the data set randomly into *training data* and *test data*;
- **Create** a support vector classifier with optional parameters;
- **Train** (using FIT) the classifier with the training data;
- **Predict** the response using the test data and compare with the true response;
- **Report** the results.

An important aspect to keep in mind is that when testing the performance using the test data, we should compare the classification accuracy to a naive baseline: if, for example, 80% of the test data is classified as $+1$, then making a prediction of $+1$ for all the data will give us an accuracy of 80%; in this case, we would want our classifier to perform considerably better than getting the right answer 80% of the time!

In [1]:

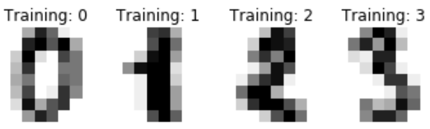
```
import numpy as np
import matplotlib.pyplot as plt
% matplotlib inline
from sklearn import svm, datasets, metrics
from sklearn.model_selection import train_test_split
```

In [2]:

```
digits = datasets.load_digits()

# Display images and labels
images_and_labels = list(zip(digits.images, digits.target))
for index, (image, label) in enumerate(images_and_labels[:4]):
    plt.subplot(2, 4, index + 1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title('Training: %i' % label)

# Turn images into 1-D arrays
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create classifier
svc = svm.SVC(gamma=0.001)

# Randomly split data into train and test set
X_train, X_test, y_train, y_test = train_test_split(data,
 digits.target, test_size = 0.4, random_state=0)
svc.fit(X_train, y_train)
```

Out [2]:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma=0.001,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```



Now apply prediction to test set and report performance.

In [3]:

```
predicted = svc.predict(X_test)
print("Classification report for classifier %s:\n%s\n"
      % (svc, metrics.classification_report(y_test, predicted)))
```

Out [3]:
```
Classification report for classifier SVC(C=1.0, cache_size=200,
 class_weight=None, coef0=0.0,vdecision_function_shape=None,
 degree=3, gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
 random_state=None, shrinking=True, tol=0.001, verbose=False):
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        60
          1       0.97      1.00      0.99        73
          2       1.00      0.97      0.99        71
          3       1.00      1.00      1.00        70
          4       1.00      1.00      1.00        63
          5       1.00      0.98      0.99        89
          6       0.99      1.00      0.99        76
          7       0.98      1.00      0.99        65
          8       1.00      0.99      0.99        78
          9       0.99      1.00      0.99        74

avg / total       0.99      0.99      0.99       719
```

In [4]:
```
import skimage
from skimage import data
from skimage.transform import resize
from skimage import io
import os
```

Now try this out on some original data!

In [5]:
```
mydigit1 = io.imread('images/digit9.png')
mydigit2 = io.imread('images/digit4.png')
plt.figure(figsize=(8, 4))
plt.subplot(1,2,1)
plt.imshow(mydigit1, cmap=plt.cm.gray_r, interpolation='nearest')
plt.axis('off')
plt.subplot(1,2,2)
plt.imshow(mydigit2, cmap=plt.cm.gray_r, interpolation='nearest')
plt.axis('off')
plt.show()
```

In [6]:
```
smalldigit1 = resize(mydigit1, (8,8))
smalldigit2 = resize(mydigit2, (8,8))
mydigits = np.concatenate((np.round(15*(np.ones((8,8))-
                        smalldigit1[:,:,0])).reshape((64,1)).T,
            np.round(15*(np.ones((8,8))-
            smalldigit2[:,:,0])).reshape((64,1)).T),axis=0)
# After some preprocessing, make prediction
guess = svc.predict(mydigits)
print guess
```

```
[9 4]
```