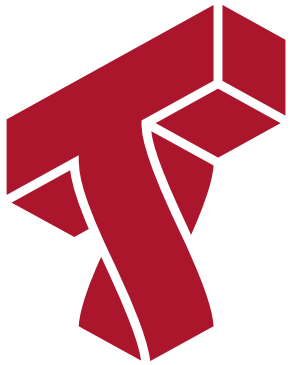


Lecture 11: Kernels

Applied Machine Learning

Volodymyr Kuleshov
Cornell Tech

Part 1: The Kernel Trick: Motivation

So far, the majority of the machine learning models we have seen have been *linear*.

In this lecture, we will see a general way to make many of these models *non-linear*. We willl use a new idea called *kernels*.

Review: Linear Regression

Recall that a linear model has the form

$$f(x) = \sum_{j=0}^{d} \theta_j \cdot x_j = \theta^\top x.$$

where x is a vector of features and we used the notation $x_0 = 1$.

We pick θ to minimize the (L2-regularized) mean squared error (MSE):

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (y^{(i)} - \theta^\top x^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{d} \theta_j^2$$

Towards General Non-Linear Features

Any non-linear feature map $\phi(x) : \mathbb{R}^d \to \mathbb{R}^p$ can be used to obtain general models of the form

$$f_\theta(x) := \theta^\top \phi(x)$$

that are highly non-linear in x but linear in θ.

The Featurized Design Matrix

It is useful to represent the featurized dataset as a matrix $\Phi \in \mathbb{R}^{n \times p}$:

$$\Phi = \begin{bmatrix} \phi(x^{(1)})_1 & \phi(x^{(1)})_2 & \dots & \phi(x^{(1)})_p \\ \phi(x^{(2)})_1 & \phi(x^{(2)})_2 & \dots & \phi(x^{(2)})_p \\ \vdots & & & \\ \phi(x^{(n)})_1 & \phi(x^{(n)})_2 & \dots & \phi(x^{(n)})_p \end{bmatrix} = \begin{bmatrix} - & \phi(x^{(1)})^\top & - \\ - & \phi(x^{(2)})^\top & - \\ & \vdots & \\ - & \phi(x^{(n)})^\top & - \end{bmatrix}.$$

Featurized Normal Equations

The normal equations provide a closed-form solution for θ:

$$\theta = (X^\top X + \lambda I)^{-1} X^\top y.$$

When the vectors of attributes $x^{(i)}$ are featurized, we can write this as

$$\theta = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top y.$$

Push-Through Matrix Identity

We can modify this expression by using a version of the push-through matrix identity (https://en.wikipedia.org/wiki/Woodbury_matrix_identity#Discussion):

$$(\lambda I + UV)^{-1}U = U(\lambda I + VU)^{-1}$$

where $U \in \mathbb{R}^{n \times m}$ and $V \in \mathbb{R}^{m \times n}$ and $\lambda \neq 0$

Proof sketch: Start with $U(\lambda I + VU) = (\lambda I + UV)U$ and multiply both sides by $(\lambda I + VU)^{-1}$ on the right and $(\lambda I + UV)^{-1}$ on the left.

Normal Equations: Dual Form

We can apply the identity $(\lambda I + UV)^{-1}U = U(\lambda I + VU)^{-1}$ to the normal equations with $U = \Phi^\top$ and $V = \Phi$.

$$\theta = (\Phi^\top \Phi + \lambda I)^{-1}\Phi^\top y$$

to obtain the *dual* form:

$$\theta = \Phi^\top (\Phi\Phi^\top + \lambda I)^{-1} y.$$

The first approach takes $O(p^3)$ time; the second is $O(n^3)$ and is faster when $p > n$.

Feature Representations for Parameters

An interesting corollary of the dual form

$$\theta = \Phi^\top \underbrace{(\Phi\Phi^\top + \lambda I)^{-1} y}_{\alpha}$$

is that the optimal θ is a linear combination of the n training set features:

$$\theta = \sum_{i=1}^{n} \alpha_i \phi(x^{(i)}).$$

Here, the weights α_i are derived from $(\Phi\Phi^\top + \lambda I)^{-1} y$ and equal

$$\alpha_i = \sum_{j=1}^{n} L_{ij} y_j$$

where $L = (\Phi\Phi^\top + \lambda I)^{-1}$.

Predictions From Features

Consider now a prediction $\phi(x')^\top \theta$ at a new input x':

$$\phi(x')^\top \theta = \sum_{i=1}^{n} \alpha_i \phi(x')^\top \phi(x^{(i)}).$$

The crucial observation is that the features $\phi(x)$ are never used directly in this equation. Only their dot product is used!

This observation will be at the heart of a powerful new idea called *the kernel trick*.

Learning From Feature Products

We also don't need features ϕ for learning θ, just their dot product! First, recall that each row i of Φ is the i-th featurized input $\phi(x^{(i)})^\top$.

Thus $K = \Phi\Phi^\top$ is a matrix of all dot products between all the $\phi(x^{(i)})$

$$K_{ij} = \phi(x^{(i)})^\top \phi(x^{(j)}).$$

We can compute $\alpha = (K + \lambda I)^{-1} y$ and use it for predictions

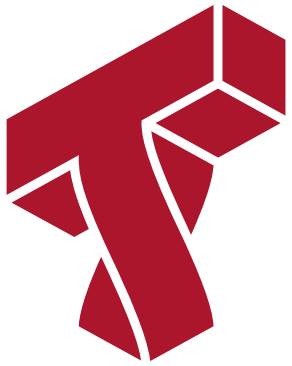
$$\phi(x')^\top \theta = \sum_{i=1}^{n} \alpha_i \phi(x')^\top \phi(x^{(i)}).$$

and all this only requires dot products, not features ϕ!

The Kernel Trick

The above observations hint at a powerful new idea -- if we can compute dot products of features $\phi(x)$ efficiently, then we will be able to use high-dimensional features easily.

It turns our that we can do this for many ML algorithms -- we call this the Kernel Trick.



Part 2: The Kernel Trick: An Example

Many ML algorithms can be written down as optimization problems in which the features $\phi(x)$ only appear as dot products $\phi(x)^\top \phi(z)$ that can be computed efficiently.

Let's look at an example.

Review: Linear Regression

Recall that a linear model has the form

$$f(x) = \sum_{j=0}^{d} \theta_j \cdot x_j = \theta^\top x.$$

where x is a vector of features and we used the notation $x_0 = 1$.

Review: Non-Linear Features

Any non-linear feature map $\phi(x) : \mathbb{R}^d \to \mathbb{R}^p$ can be used in this way to obtain general models of the form

$$f_\theta(x) := \theta^\top \phi(x)$$

that are highly non-linear in x but linear in θ.

Review: Featurized Design Matrix

It is useful to represent the featurized dataset as a matrix $\Phi \in \mathbb{R}^{n \times p}$:

$$\Phi = \begin{bmatrix} \phi(x^{(1)})_1 & \phi(x^{(1)})_2 & \dots & \phi(x^{(1)})_p \\ \phi(x^{(2)})_1 & \phi(x^{(2)})_2 & \dots & \phi(x^{(2)})_p \\ \vdots & & & \\ \phi(x^{(n)})_1 & \phi(x^{(n)})_2 & \dots & \phi(x^{(n)})_p \end{bmatrix} = \begin{bmatrix} - & \phi(x^{(1)})^\top & - \\ - & \phi(x^{(2)})^\top & - \\ & \vdots & \\ - & \phi(x^{(n)})^\top & - \end{bmatrix}.$$

Review: Normal Equations

The normal equations provide a closed-form solution for θ:

$$\theta = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top y.$$

They also can be written in this form:

$$\theta = \Phi^\top (\Phi \Phi^\top + \lambda I)^{-1} y.$$

The first approach takes $O(d^3)$ time; the second is $O(n^3)$ and is faster when $d > n$.

Learning From Feature Products

An interesting corollary is that the optimal θ is a linear combination of the n training set features:

$$\theta = \sum_{i=1}^{n} \alpha_i \phi(x^{(i)}).$$

We can compute a prediction $\phi(x')^\top \theta$ for x' without ever using the features (only their dot products):

$$\phi(x')^\top \theta = \sum_{i=1}^{n} \alpha_i \phi(x')^\top \phi(x^{(i)}).$$

Equally importantly, we can learn θ from only dot products.

Review: Polynomial Regression

Note that a p-th degree polynomial

$$a_p x^p + a_{p-1} x^{p-1} + \ldots + a_1 x + a_0.$$

forms a linear model with parameters a_p, a_{p-1}, $\ldots$, a_0. This means we can use our algorithms for linear models to learn non-linear features!

Specifically, given a one-dimensional continuous variable x, we can defining a feature function $\phi : \mathbb{R} \to \mathbb{R}^p$ as

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^p \end{bmatrix}.$$

Then the class of models of the form

$$f_\theta(x) := \sum_{j=0}^{p} \theta_p x^p = \theta^\top \phi(x)$$

with parameters θ encompasses the set of p-degree polynomials. Specifically,

- It is non-linear in the input variable x, meaning that we can model complex data relationships.
- It is a linear model as a function of the parameters θ, meaning that we can use our familiar ordinary least squares algorithm to learn these features.

The Kernel Trick: A First Example

Can we compute the dot product $\phi(x)^\top \phi(x')$ of polynomial features $\phi(x)$ more efficiently than using the standard definition of a dot product? Let's look at an example.

To start, consider polynomial features $\phi : \mathbb{R}^d \to \mathbb{R}^{d^2}$ of the form

$$\phi(x)_{ij} = x_i x_j \quad \text{for } i, j \in \{1, 2, \ldots, d\}.$$

For $d = 3$ this looks like

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_1 \\ x_2 x_2 \\ x_3 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}.$$

The product of x and z in feature space equals:

$$\phi(x)^\top \phi(z) = \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j z_i z_j$$

Computing this dot product invovles the sum over d^2 terms and takes $O(d^2)$ time.

An altenative way of computing the dot product $\phi(x)^\top \phi(z)$ is to instead compute $(x^\top z)^2$. One can check that this has the same result:

$$\begin{aligned}
(x^\top z)^2 &= (\sum_{i=1}^{d} x_i z_i)^2 \\
&= (\sum_{i=1}^{d} x_i z_i) \cdot (\sum_{j=1}^{d} x_j z_j) \\
&= \sum_{i=1}^{d} \sum_{j=1}^{d} x_i z_i x_j z_j \\
&= \phi(x)^\top \phi(z)
\end{aligned}$$

However, computing $(x^\top z)^2$ can be done in only $O(d)$ time!

This is a very powerful idea:

- We can compute the dot product between $O(d^2)$ features in only $O(d)$ time.
- We can use high-dimensional features within ML algorithms that only rely on dot products (like kernelized ridge regression) without incurring extra costs.

The Kernel Trick: Polynomial Features

The number of polynomial features ϕ_p of degree p when $x \in \mathbb{R}^d$

$$\phi_p(x)_{i_1,i_2,\ldots,i_p} = x_{i_1} x_{i_2} \cdots x_{i_p} \text{ for } i_1, i_2, \ldots, i_p \in \{1, 2, \ldots, d\}$$

scales as $O(d^p)$.

However, we can compute the dot product $\phi_p(x)^\top \phi_p(z)$ in this feature space in only $O(d)$ time for any p as:

$$\phi_p(x)^\top \phi_p(z) = (x^\top z)^p.$$

Algorithm: Kernelized Polynomial Ridge Regression

- **Type**: Supervised learning (Regression)
- **Model family**: Polynomials.
- **Objective function**: $L2$-regularized ridge regression.
- **Optimizer**: Normal equations (dual form).
- **Probabilistic interpretation**: No simple interpretation!

The Kernel Trick: General Idea

Many types of features $\phi(x)$ have the property that their dot product $\phi(x)^\top \phi(z)$ can be computed more efficiently than if we had to form these features explicitly.

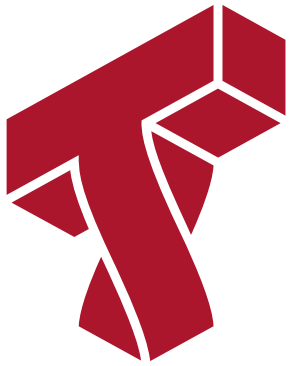
Also, we will see that many algorithms in machine learning can be written down as optimization problems in which the features $\phi(x)$ only appear as dot products $\phi(x)^\top \phi(z)$.

The *Kernel Trick* means that we can use complex non-linear features within these algorithms with little additional computational cost.

Examples of algorithms in which we can use the Kernel trick:

- Supervised learning algorithms: linear regression, logistic regression, support vector machines, etc.
- Unsupervised learning algorithms: PCA, density estimation.

We will look at more examples shortly.



Part 3: The Kernel Trick in SVMs

Many ML algorithms can be written down as optimization problems in which the features $\phi(x)$ only appear as dot products $\phi(x)^\top \phi(z)$ that can be computed efficiently.

We will now see how SVMs can benefit from the Kernel Trick as well.

Review: Binary Classification

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

We distinguish between two types of supervised learning problems depnding on the targets $y^{(i)}$.

1. **Regression**: The target variable $y \in \mathcal{Y}$ is continuous: $\mathcal{Y} \subseteq \mathbb{R}$.
2. **Binary Classification**: The target variable y is discrete and takes on one of $K = 2$ possible values.

In this lecture, we assume $\mathcal{Y} = \{-1, +1\}$.

Review: SVM Model Family

We will consider models of the form

$$f_\theta(x) = \theta^\top \phi(x) + \theta_0$$

where x is the input and $y \in \{-1, 1\}$ is the target.

Review: Primal and Dual Formulations

Recall that the the max-margin hyperplane can be formualted as the solution to the following *primal* optimization problem.

$$\begin{aligned} \min_{\theta, \theta_0, \xi} \ & \frac{1}{2}||\theta||^2 + C \sum_{i=1}^{n} \xi_i \\ \text{subject to } \ & y^{(i)}((x^{(i)})^\top \theta + \theta_0) \geq 1 - \xi_i \text{ for all } i \\ & \xi_i \geq 0 \end{aligned}$$

The solution to this problem also happens to be given by the following *dual* problem:

$$\max_{\lambda} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \lambda_i \lambda_k y^{(i)} y^{(k)} \left(x^{(i)}\right)^\top x^{(k)}$$

$$\text{subject to } \sum_{i=1}^{n} \lambda_i y^{(i)} = 0$$

$$C \geq \lambda_i \geq 0 \text{ for all } i$$

Review: Primal Solution

We can obtain a primal solution from the dual via the following equation:

$$\theta^* = \sum_{i=1}^{n} \lambda_i^* y^{(i)} \phi(x^{(i)}).$$

Ignoring the θ_0 term for now, the score at a new point x' will equal

$$\theta^\top \phi(x') = \sum_{i=1}^{n} \lambda_i^* y^{(i)} \phi(x^{(i)})^\top \phi(x').$$

The Kernel Trick in SVMs

Notice that in both equations, the features x are never used directly. Only their *dot product* is used.

$$\sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \lambda_i \lambda_k y^{(i)} y^{(k)} \phi(x^{(i)})^\top \phi(x^{(k)})$$

$$\theta^\top \phi(x') = \sum_{i=1}^{n} \lambda_i^* y^{(i)} \phi(x^{(i)})^\top \phi(x').$$

If we can compute the dot product efficiently, we can potentially use very complex features.

The Kernel Trick in SVMs

More generally, given features $\phi(x)$, suppose that we have a function $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ that outputs dot products between vectors in $\mathcal{X}$

$$K(x, z) = \phi(x)^\top \phi(z).$$

We will call K the *kernel* function.

Recall that an example of a useful kernel function is

$$K(x, z) = (x \cdot z)^p$$

because it computes the dot product of polynomial features of degree p.

Then notice that we can rewrite the dual of the SVM as

$$\max_{\lambda} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \lambda_i \lambda_k y^{(i)} y^{(k)} K(x^{(i)}, x^{(k)})$$

$$\text{subject to } \sum_{i=1}^{n} \lambda_i y^{(i)} = 0$$

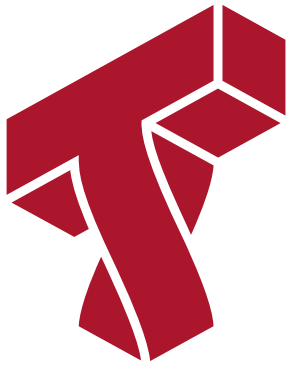
$$C \geq \lambda_i \geq 0 \text{ for all } i$$

and predictions at a new point x' are given by $\sum_{i=1}^{n} \lambda_i^* y^{(i)} K(x^{(i)}, x')$.

Using our earlier trick, we can use polynomial features of any degree p in SVMs without forming these features and at no extra cost!

Algorithm: Kernelized Support Vector Machine Classification (Dual Form)

- **Type**: Supervised learning (binary classification)
- **Model family**: Non-linear decision boundaries.
- **Objective function**: Dual of SVM optimization problem.
- **Optimizer**: Sequential minimial optimization.
- **Probabilistic interpretation**: No simple interpretation!



Part 4: Types of Kernels

Now that we saw the kernel trick, let's look at several examples of kernels.

Review: Linear Model Family

We will consider models of the form

$$f_\theta(x) = \theta^\top \phi(x) + \theta_0$$

where x is the input and y is the target.

Kernel Trick for Ridge Regression

The normal equations provide a closed-form solution for θ:

$$\theta = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top y.$$

They also can be written in this form:

$$\theta = \Phi^\top (\Phi \Phi^\top + \lambda I)^{-1} y.$$

The first approach takes $O(d^3)$ time; the second is $O(n^3)$ and is faster when $d > n$.

An interesting corollary is that the optimal θ is a linear combination of the n training set features:

$$\theta = \sum_{i=1}^{n} \alpha_i \phi(x^{(i)}).$$

We can compute a prediction $\phi(x')^\top \theta$ for x' without ever using the features (only their dot products):

$$\phi(x')^\top \theta = \sum_{i=1}^{n} \alpha_i \phi(x')^\top \phi(x^{(i)}).$$

Equally importantly, we can learn θ from only dot products.

Review: Kernel Trick in SVMs

Notice that in both equations, the features x are never used directly. Only their *dot product* is used.

$$\sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \lambda_i \lambda_k y^{(i)} y^{(k)} \phi(x^{(i)})^\top \phi(x^{(k)})$$

$$\theta^\top \phi(x') = \sum_{i=1}^{n} \lambda_i^* y^{(i)} \phi(x^{(i)})^\top \phi(x').$$

If we can compute the dot product efficiently, we can potentially use very complex features.

Definition: Kernels

The *kernel* corresponding to features $\phi(x)$ is a function $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ that outputs dot products between vectors in $\mathcal{X}$

$$K(x, z) = \phi(x)^\top \phi(z).$$

We will also consider general functions $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ and call these *kernel functions*.

Kernels have various intepreations:

- The dot product or geometrical angle between x and z
- A notion of similarity between x and z

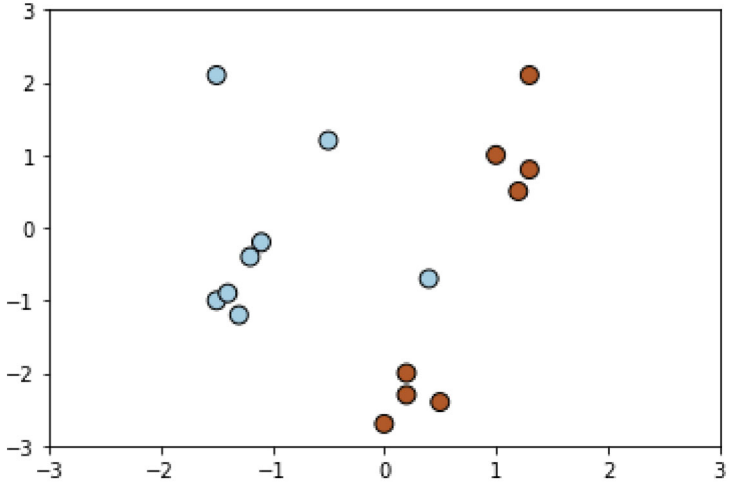
In order to illustrate kernels, we will use this dataset.

In [17]:
```
# https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

# Our dataset and targets
X = np.c_[(.4, -.7), (-1.5, -1), (-1.4, -.9), (-1.3, -1.2), (-1.1, -.2), (-1.2, -.
4), (-.5, 1.2), (-1.5, 2.1), (1, 1),
          (1.3, .8), (1.2, .5), (.2, -2), (.5, -2.4), (.2, -2.3), (0, -2.7), (1.3,
2.1)].T
Y = [0] * 8 + [1] * 8

x_min, x_max = -3, 3
y_min, y_max = -3, 3
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k',
s=80)
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

Out[17]:
```
(-3.0, 3.0)
```



Example: Linear Kernel

The simplest kind of kernel that exists is called the linear kernel. This simply corresponds to dot product multiplication of the features:

$$K(x, z) = x^\top z$$

Applied to an SVM, this corresponds to a linear decision boundary.

Below is an example of how we can use the SVM implementation in `sklearn` with a linear kernel.

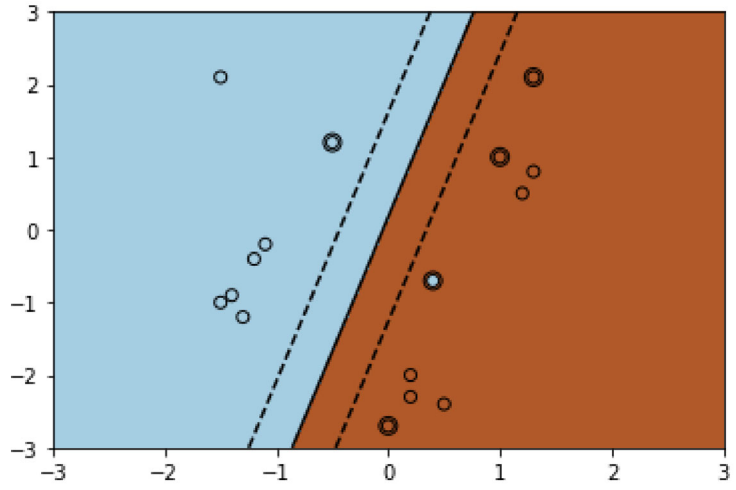
Internally, this solves the dual SVM optimization problem.

In [20]:
```
# https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html
clf = svm.SVC(kernel='linear', gamma=2)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80, facecolo
rs='none', zorder=10, edgecolors='k')
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k')
XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

# Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],levels
=[-.5, 0, .5])
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

Out[20]:
```
(-3.0, 3.0)
```



Example: Polynomial Kernel

A more interesting example is the polynomial kernel of degree p, of which we have already seen a simple example:

$$K(x, z) = (x^\top z + c)^p.$$

This corresponds to a mapping to a feature space of dimension $\binom{d+p}{p}$ that has all monomials $x_{i_1} x_{i_2} \cdots x_{i_p}$ of degree at most p.

For $d = 3$ this feature map looks like

The polynomial kernel allows us to compute dot products in a $O(d^p)$-dimensional space in time $O(d)$.

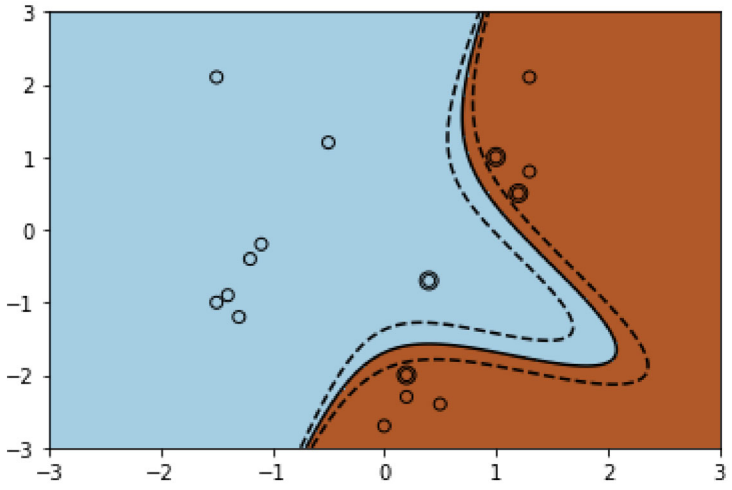
Let's see how it would be implemented in `sklearn` .

In [19]:
```
# https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html
clf = svm.SVC(kernel='poly', degree=3, gamma=2)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80, facecolo
rs='none', zorder=10, edgecolors='k')
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k')
XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

# Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],levels
=[-.5, 0, .5])
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

Out[19]:
```
(-3.0, 3.0)
```



Example: Radial Basis Function Kernel

Another example is the Radial Basis Function (RBF; sometimes called Gaussian) kernel

$$K(x, z) = \exp\left(-\frac{||x - z||^2}{2\sigma^2}\right),$$

where σ is a hyper-parameter. It's easiest to understand this kernel by viewing it as a similarity measure.

We can show that this kernel corresponds to an *infinite-dimensional* feature map and the limit of the polynomial kernel as $p \to \infty$.

To see why that's intuitively the case, consider the Taylor expansion

$$\exp\left(-\frac{||x-z||^2}{2\sigma^2}\right) \approx 1 - \frac{||x-z||^2}{2\sigma^2} + \frac{||x-z||^4}{2! \cdot 4\sigma^4} - \frac{||x-z||^6}{3! \cdot 8\sigma^6} + \dots$$

Each term on the right hand side can be expanded into a polynomial.

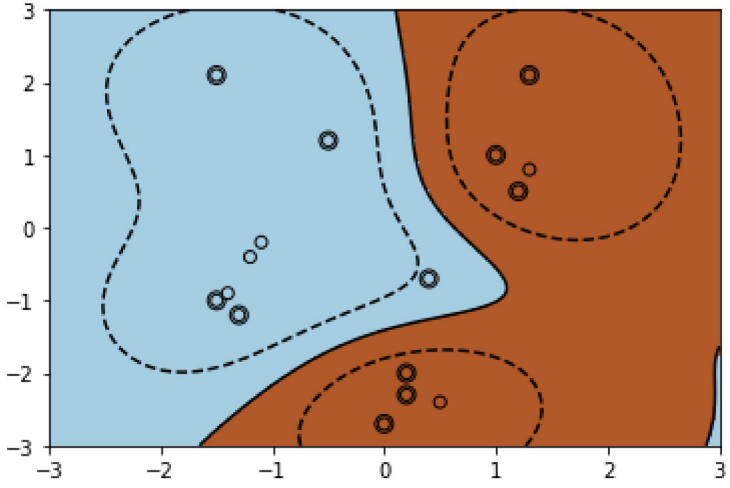
We can look at the `sklearn` implementation again.

In [29]:
```
# https://scikit-learn.org/stable/auto_examples/svm/plot_svm_kernels.html
clf = svm.SVC(kernel='rbf', gamma=.5)
clf.fit(X, Y)

# plot the line, the points, and the nearest vectors to the plane
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80, facecolo
rs='none', zorder=10, edgecolors='k')
plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k')
XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

# Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],levels
=[-.5, 0, .5])
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

Out[29]: (-3.0, 3.0)



When is K A Kernel?

We've seen that for many features ϕ we can define a kernel function $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ that efficiently computes $\phi(x)^\top \phi(x)$.

Suppose now that we use some kernel function $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ in an ML algorithm. Is there an implicit feature mapping ϕ that corresponds to using K?

Let's start by defining a necessary condition for $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ to be associated with a feature map.

Suppose that K is a kernel for some feature map ϕ, and consider an arbitrary set of n points $\{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$.

Consider the matrix $L \in \mathbb{R}^{n \times n}$ defined as $L_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^\top \phi(x^{(j)})$. We claim that L must be symmetric and positive semidefinite.

Indeed, it L is symmetric because the dot product $\phi(x^{(i)})^\top \phi(x^{(j)})$ is symmetric. Moreover, for any z,

$$\begin{aligned} z^\top L z &= \sum_{i=1}^n \sum_{j=1}^n z_i L_{ij} z_j = \sum_{i=1}^n \sum_{j=1}^n z_i \phi(x^{(i)})^\top \phi(x^{(j)}) z_j \\ &= \sum_{i=1}^n \sum_{j=1}^n z_i (\sum_{k=1}^n \phi(x^{(i)})_k \phi(x^{(j)})_k) z_j \\ &= \sum_{k=1}^n \sum_{i=1}^n \sum_{j=1}^n z_i \phi(x^{(i)})_k \phi(x^{(j)})_k z_j \\ &= \sum_{k=1}^n \sum_{i=1}^n \left(z_i \phi(x^{(i)})_k \right)^2 \geq 0 \end{aligned}$$

Thus if K is a kernel, L must be positive semidefinite for any n points $x^{(i)}$.

Mercer's Theorem

if K is a kernel, L must be positive semidefinite for any set of n points $x^{(i)}$. It turns out that it is is also a sufficent condition.

Theorem. (Mercer) Let $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ be a kernel function. There exists a mapping ϕ associated with K if for any n and any dataset $\{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ of size $n \geq 1$, if and only if the matrix L defined as $L_{ij} = K(x^{(i)}, x^{(j)})$ is symmetric and positive semidefinite.

This characterizes precisely which kernel functions correspond to some ϕ.

Pros and Cons of Kernels

Are kernels a free lunch? Not quite.

- Kernels allow us to use features ϕ of very large dimension d.
- However computation is at least $O(n^2)$, where n is the dataset size. We need to compute distances $K(x^{(i)}, x^{(j)})$, for all i, j.
- Approximate solutions can be found more quickly, but in practice kernel methods are not used with today's massive datasets.
- However, on small and medium-sized data, kernel methods will be at least as good as neural nets and probably much easier to train.

Summary: Kernels

- A kernel is a function $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ that defines a notion of similarity over pairs of vectors in $\mathcal{X}$.

- Kernels are often associated with high-dimensional features ϕ and implicitly map inputs to this feature space.

- Kernels can be incorporated into many machine learning algorithms, which enables them to learn highly nonlinear models.

Examples of algorithms in which we can use kernels include:

- Supervised learning algorithms: linear regression, logistic regression, support vector machines, etc.
- Unsupervised learning algorithms: PCA, density estimation.

Kernels are very powerful because they can be used throughout machine learning.