CS 453X: Class 14

Jacob Whitehill

Kernel trick

Kernelization

- Both during training and testing, we only use each training point $\mathbf{x}^{(i)}$ as part of an inner product — *we don't need the raw values themselves*.

- Therefore, even if we want to transform each input **using** ϕ, we only really need to know the inner products between each $\phi(\mathbf{x})$ and $\phi(\mathbf{x}^{(i)})$ (for training):

$$L(\alpha) = \sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha^{(i)} \alpha^{(i')} y^{(i)} y^{(i')} \phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(i')})$$

Kernelization

- Both during training and testing, we only use each training point $\mathbf{x}^{(i)}$ as part of an inner product — *we don't need the raw values themselves*.

- Therefore, even if we want to transform each input **using** ϕ, we only really need to know the inner products between each $\phi(\mathbf{x})$ and $\phi(\mathbf{x}^{(i)})$ (for testing):

$$\mathbf{x}^\top \mathbf{w} + b \quad = \quad \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \phi(\mathbf{x})^\top \phi(\mathbf{x}^{(i)}) + b$$

Kernelization

- For training, rather than compute $\boldsymbol{\phi}(\mathbf{x}^{(i)})$ for every training example $\mathbf{x}^{(i)}$…:

$$\tilde{\mathbf{X}} = \begin{bmatrix} | & & | \\ \phi\left(\mathbf{x}^{(1)}\right) & \dots & \phi\left(\mathbf{x}^{(n)}\right) \\ | & & | \end{bmatrix}$$

m* x *n

Kernelization

- …instead compute the **kernel matrix** containing all pairs of inner products:

$$\mathbf{K} = \begin{bmatrix} \phi(\mathbf{x}^{(1)})^\top \phi(\mathbf{x}^{(1)}) & \dots & \phi(\mathbf{x}^{(1)})^\top \phi(\mathbf{x}^{(n)}) \\ & \ddots & \\ \phi(\mathbf{x}^{(n)})^\top \phi(\mathbf{x}^{(1)}) & \dots & \phi(\mathbf{x}^{(n)})^\top \phi(\mathbf{x}^{(n)}) \end{bmatrix}$$

n* x *n

Kernelization

- Then we just need to pass **K** to the SVM solver:

```
svm = sklearn.svm.SVC(kernel='precomputed')
K = Xtilde.T.dot(Xtilde)    # K = X̃ᵀX̃
svm.fit(K, y)
```

$\mathbf{K} = \tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}$

Kernelization

- Let's define a function *k* — called a **kernel** — that computes the inner product between any two training examples:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right)$$

- Now can we can express **K** as:

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \dots & k(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}) \\ & \ddots & \\ k(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}) & \dots & k(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}) \end{bmatrix}$$

Kernelization

- Using kernel functions, we can sometimes express the inner product of two transformed training examples **more compactly** and **more computationally efficiently**.

Kernel example

- Example — suppose you want ϕ to compute polynomial features of $\mathbf{x}$ of degree 2, i.e.,

$$\phi\left(\left[\begin{array}{c} x \\ y \end{array}\right]\right) = \left[\begin{array}{c} 1 \\ \sqrt{2}x \\ \sqrt{2}y \\ \sqrt{2}xy \\ x^2 \\ y^2 \end{array}\right]$$

- The transformed feature space has 6 dimensions.

- Computing $\phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right)$ directly therefore requires 6 multiplications, plus the cost of transforming each vector.

Kernel example

- On the other hand, we can derive that:

$$
\begin{aligned}
k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right) \\
&= \phi\left(\begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix}\right)^\top \phi\left(\begin{bmatrix} x^{(j)} \\ y^{(j)} \end{bmatrix}\right)
\end{aligned}
$$

Kernel example

- On the other hand, we can derive that:

$$
\begin{aligned}
k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right) \\
&= \phi\left(\left[\begin{array}{c} x^{(i)} \\ y^{(i)} \end{array}\right]\right)^\top \phi\left(\left[\begin{array}{c} x^{(j)} \\ y^{(j)} \end{array}\right]\right) \\
&= \left[\begin{array}{c} 1 \\ \sqrt{2}x^{(i)} \\ \sqrt{2}y^{(i)} \\ \sqrt{2}x^{(i)}y^{(i)} \\ {x^{(i)}}^2 \\ {y^{(i)}}^2 \end{array}\right]^\top \left[\begin{array}{c} 1 \\ \sqrt{2}x^{(j)} \\ \sqrt{2}y^{(j)} \\ \sqrt{2}x^{(j)}y^{(j)} \\ {x^{(j)}}^2 \\ {y^{(j)}}^2 \end{array}\right]
\end{aligned}
$$

Kernel example

- On the other hand, we can derive that:

$$
\begin{aligned}
k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right) \\
&= \phi\left(\left[\begin{array}{c} x^{(i)} \\ y^{(i)} \end{array}\right]\right)^\top \phi\left(\left[\begin{array}{c} x^{(j)} \\ y^{(j)} \end{array}\right]\right) \\
&= \left[\begin{array}{c} 1 \\ \sqrt{2}x^{(i)} \\ \sqrt{2}y^{(i)} \\ \sqrt{2}x^{(i)}y^{(i)} \\ {x^{(i)}}^2 \\ {y^{(i)}}^2 \end{array}\right]^\top \left[\begin{array}{c} 1 \\ \sqrt{2}x^{(j)} \\ \sqrt{2}y^{(j)} \\ \sqrt{2}x^{(j)}y^{(j)} \\ {x^{(j)}}^2 \\ {y^{(j)}}^2 \end{array}\right] \\
&= 1 + 2x^{(i)}x^{(j)} + 2y^{(i)}y^{(j)} + 2x^{(i)}y^{(i)}x^{(j)}y^{(j)} + (x^{(i)}x^{(j)})^2 + (y^{(i)}y^{(j)})^2
\end{aligned}
$$

Kernel example

- On the other hand, we can derive that:

$$
\begin{aligned}
k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right) \\
&= \phi\left(\left[\begin{array}{c} x^{(i)} \\ y^{(i)} \end{array}\right]\right)^\top \phi\left(\left[\begin{array}{c} x^{(j)} \\ y^{(j)} \end{array}\right]\right) \\
&= \left[\begin{array}{c} 1 \\ \sqrt{2}x^{(i)} \\ \sqrt{2}y^{(i)} \\ \sqrt{2}x^{(i)}y^{(i)} \\ {x^{(i)}}^2 \\ {y^{(i)}}^2 \end{array}\right]^\top \left[\begin{array}{c} 1 \\ \sqrt{2}x^{(j)} \\ \sqrt{2}y^{(j)} \\ \sqrt{2}x^{(j)}y^{(j)} \\ {x^{(j)}}^2 \\ {y^{(j)}}^2 \end{array}\right] \\
&= 1 + 2x^{(i)}x^{(j)} + 2y^{(i)}y^{(j)} + 2x^{(i)}y^{(i)}x^{(j)}y^{(j)} + (x^{(i)}x^{(j)})^2 + (y^{(i)}y^{(j)})^2 \\
&= (1 + x^{(i)}x^{(j)} + y^{(i)}y^{(j)})^2
\end{aligned}
$$

Kernel example

- On the other hand, we can derive that:

$$
\begin{aligned}
k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right) \\
&= \phi\left(\begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix}\right)^\top \phi\left(\begin{bmatrix} x^{(j)} \\ y^{(j)} \end{bmatrix}\right) \\
&= \begin{bmatrix} 1 \\ \sqrt{2}x^{(i)} \\ \sqrt{2}y^{(i)} \\ \sqrt{2}x^{(i)}y^{(i)} \\ {x^{(i)}}^2 \\ {y^{(i)}}^2 \end{bmatrix}^\top \begin{bmatrix} 1 \\ \sqrt{2}x^{(j)} \\ \sqrt{2}y^{(j)} \\ \sqrt{2}x^{(j)}y^{(j)} \\ {x^{(j)}}^2 \\ {y^{(j)}}^2 \end{bmatrix} \\
&= 1 + 2x^{(i)}x^{(j)} + 2y^{(i)}y^{(j)} + 2x^{(i)}y^{(i)}x^{(j)}y^{(j)} + (x^{(i)}x^{(j)})^2 + (y^{(i)}y^{(j)})^2 \\
&= (1 + x^{(i)}x^{(j)} + y^{(i)}y^{(j)})^2 \\
&= \left(1 + \begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix}^\top \begin{bmatrix} x^{(j)} \\ y^{(j)} \end{bmatrix}\right)^2
\end{aligned}
$$

Kernel example

- On the other hand, we can derive that:

$$
\begin{aligned}
k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) &= \phi\left(\mathbf{x}^{(i)}\right)^\top \phi\left(\mathbf{x}^{(j)}\right) \\
&= \phi\left(\begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix}\right)^\top \phi\left(\begin{bmatrix} x^{(j)} \\ y^{(j)} \end{bmatrix}\right) \\
&= \begin{bmatrix} 1 \\ \sqrt{2}x^{(i)} \\ \sqrt{2}y^{(i)} \\ \sqrt{2}x^{(i)}y^{(i)} \\ {x^{(i)}}^2 \\ {y^{(i)}}^2 \end{bmatrix}^\top \begin{bmatrix} 1 \\ \sqrt{2}x^{(j)} \\ \sqrt{2}y^{(j)} \\ \sqrt{2}x^{(j)}y^{(j)} \\ {x^{(j)}}^2 \\ {y^{(j)}}^2 \end{bmatrix} \\
&= 1 + 2x^{(i)}x^{(j)} + 2y^{(i)}y^{(j)} + 2x^{(i)}y^{(i)}x^{(j)}y^{(j)} + (x^{(i)}x^{(j)})^2 + (y^{(i)}y^{(j)})^2 \\
&= (1 + x^{(i)}x^{(j)} + y^{(i)}y^{(j)})^2 \\
&= \left(1 + \begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix}^\top \begin{bmatrix} x^{(j)} \\ y^{(j)} \end{bmatrix}\right)^2 \\
&= \left(1 + {\mathbf{x}^{(i)}}^\top \mathbf{x}^{(j)}\right)^2
\end{aligned}
$$

We can compute the inner product of the transformed vectors more efficiently (just 2 multiplies and a power).

Kernel functions

- This was a polynomial kernel of degree 2.

- In general, we can devise many kernels of the form:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \left(\lambda + \gamma {\mathbf{x}^{(i)}}^\top \mathbf{x}^{(j)}\right)^d$$

  where γ, λ, d can be tuned for the particular application.

Kernel functions

- **`sklearn`** supports polynomial (and several other) kernels off-the-shelf:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \left(\lambda + \gamma \mathbf{x}^{(i)\top} \mathbf{x}^{(j)}\right)^d$$

```
svm = sklearn.svm.SVC(kernel='poly', degree=2,
                      gamma=1, coef0=1)
```

- When using a "pre-built" kernel function, we don't need to manually compute K — just pass the *raw* (untransformed) **X** to **`fit`**:

```
svm.fit(X, y)
```

Kernel functions

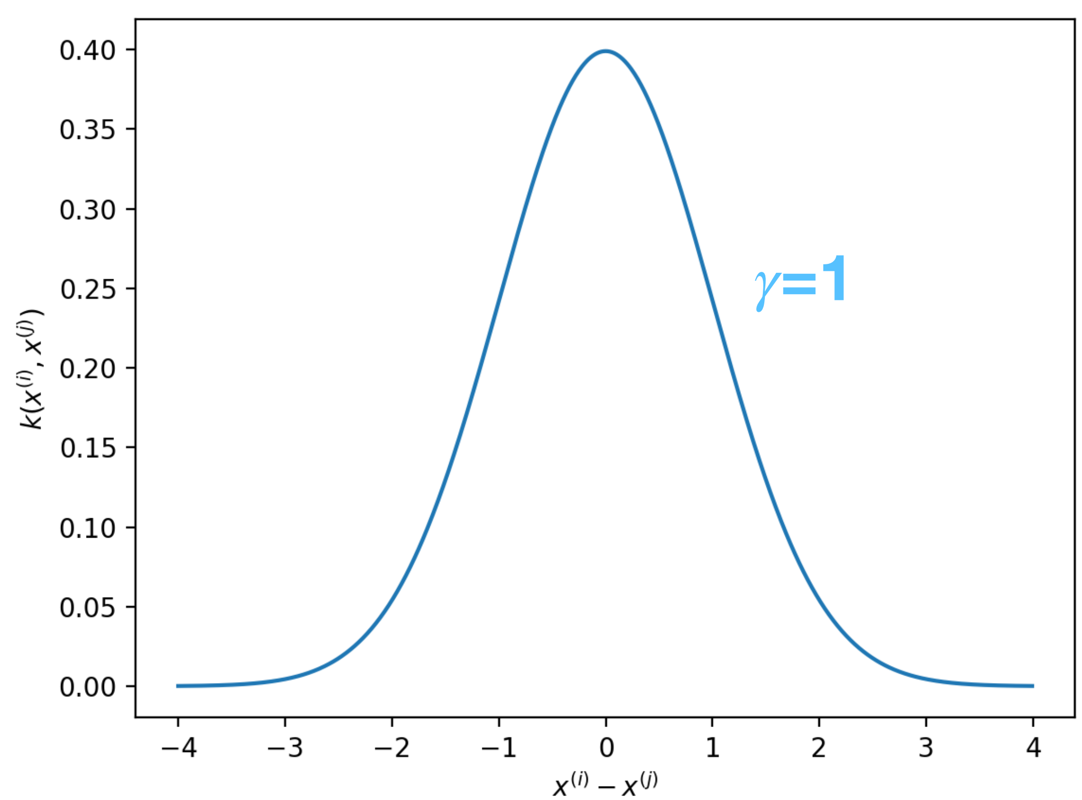
- Not only can kernel functions be more efficient than transforming each input — they can also offer more representational power.
- For the kernel k, we can use any function that computes the inner product between $\mathbf{x}^{(i)}$, $\mathbf{x}^{(j)}$ after applying some transformation to each vector.
- But the transformation can be *anything — we may not even care what it is.*

Kernel functions

- One of the most popular SVM kernels is the Gaussian radial basis function (RBF) kernel:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

- The RBF kernel expresses that two vectors close together should have a higher kernel value than two vectors far apart:

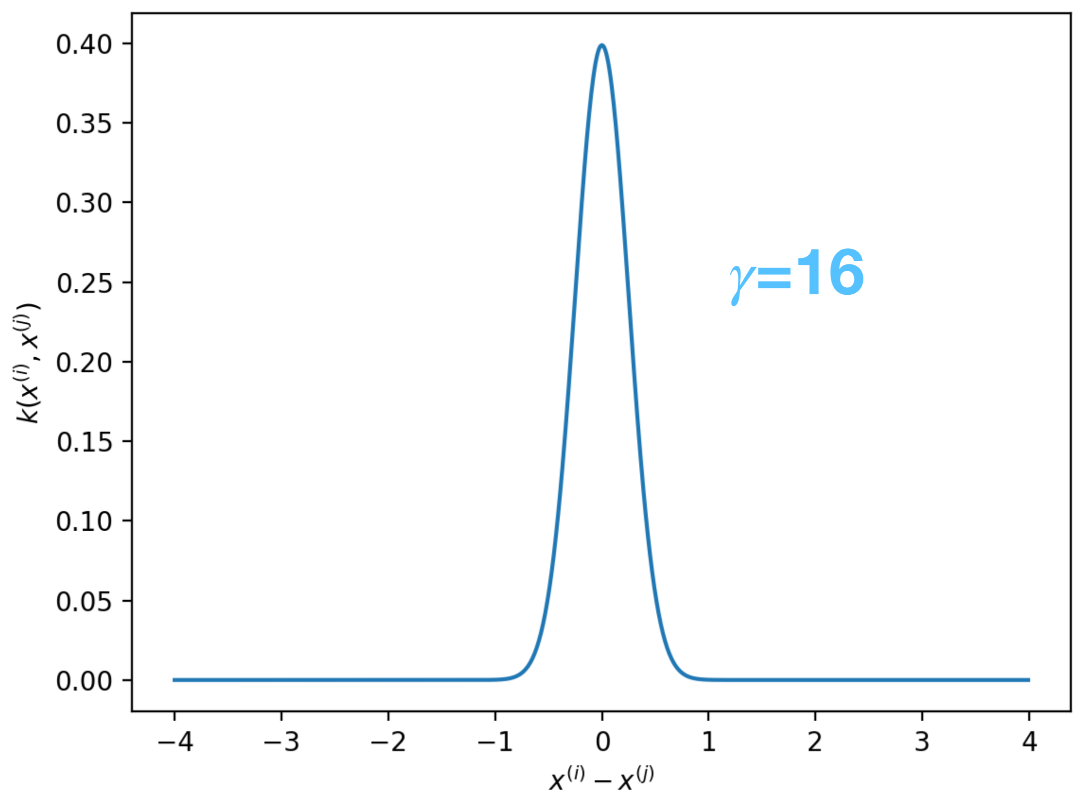


Kernel functions

- One of the most popular SVM kernels is the Gaussian radial basis function (RBF) kernel:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

- The **bandwidth** γ controls how quickly the kernel value decreases as a function of the distance between the two input vectors:



Kernel functions

- One of the most popular SVM kernels is the Gaussian radial basis function (RBF) kernel:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

- The “transformation” ϕ is completely hidden — mathematically it can be proven to exist, but we don’t have to care what it is.

    - In fact, for RBF, the implicit transformation has *infinitely* many dimensions.

Kernel functions

- One of the most popular SVM kernels is the Gaussian radial basis function (RBF) kernel:

$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

- We can use RBF in **`sklearn`** with:

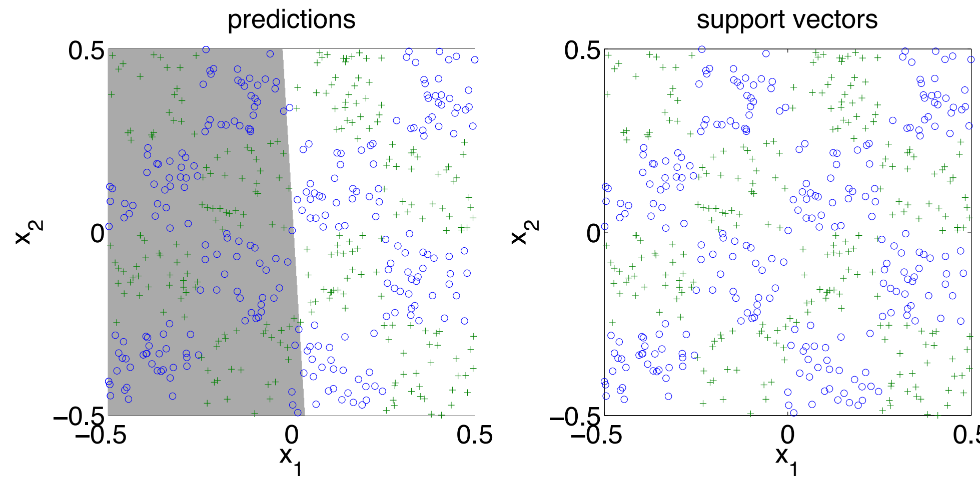
```
svm = sklearn.svm.SVC(kernel='rbf', gamma=1)
```

Kernelization

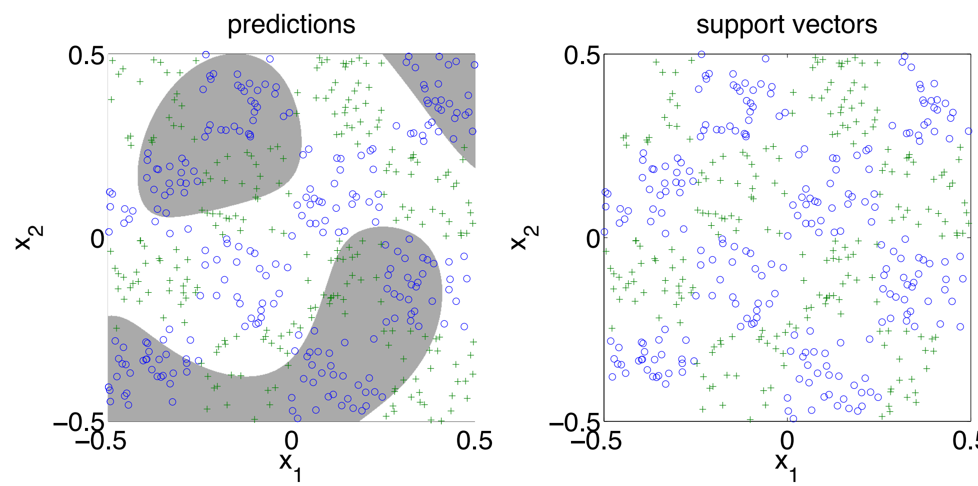
- SVMs always try to separate the positive from the negative examples using a hyperplane — a linear decision boundary.
- But the hyperplane might exist in a very different (transformed) space than the raw input data.
- In the original input space, the decision boundary can be non-linear.

Non-linear decision boundaries

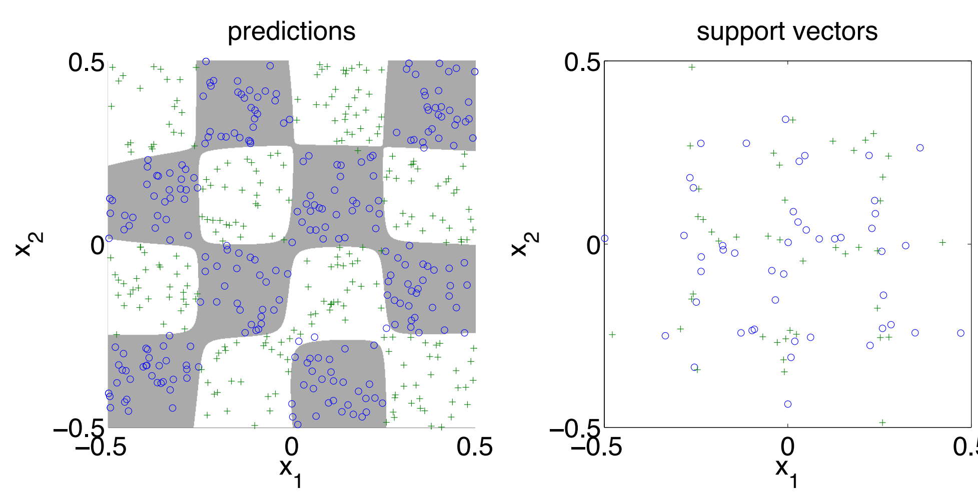
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^5$.



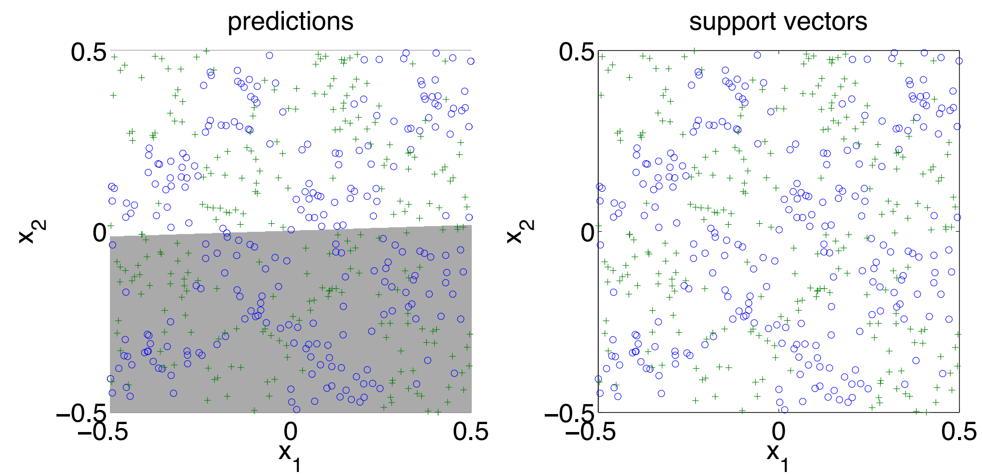
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^{10}$.



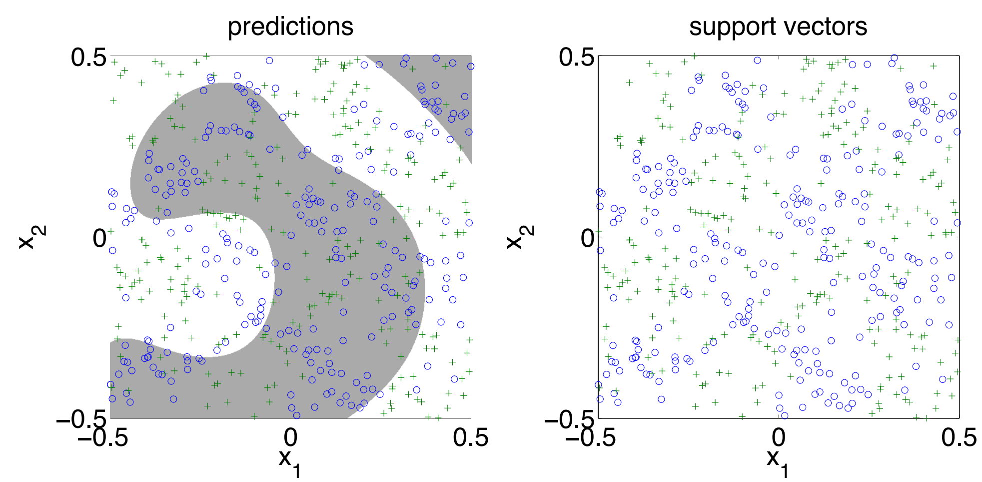
https://people.cs.umass.edu/~domke/courses/sml2010/06kernels.pdf

Non-linear decision boundaries

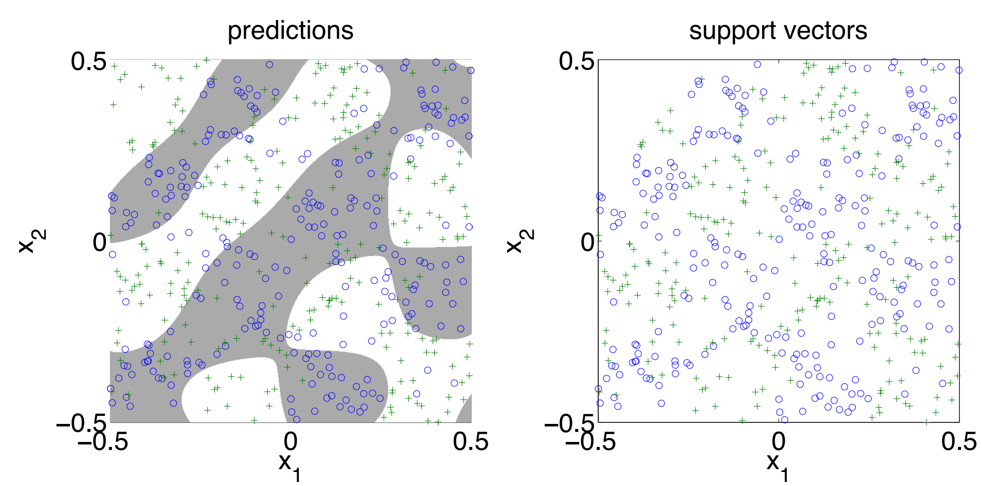
Dataset C (dataset B with noise), $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^5$.



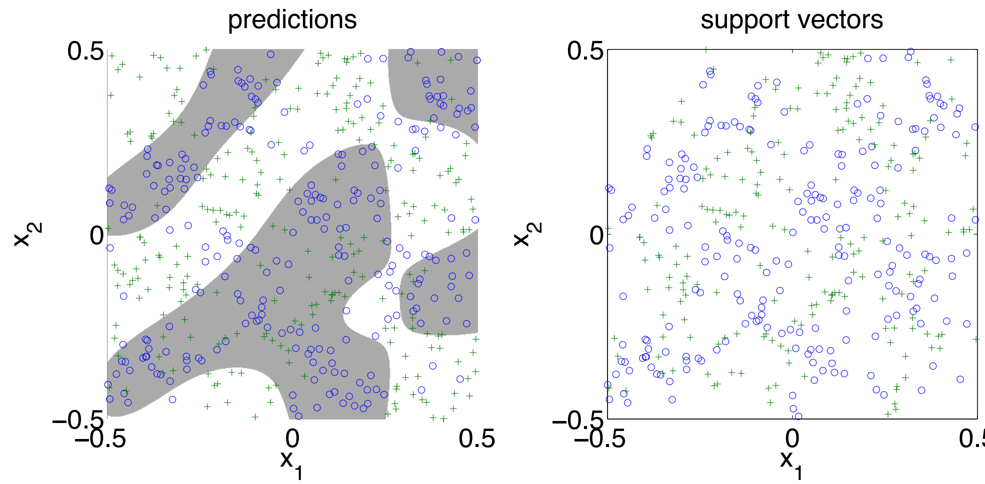
Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^{10}$.



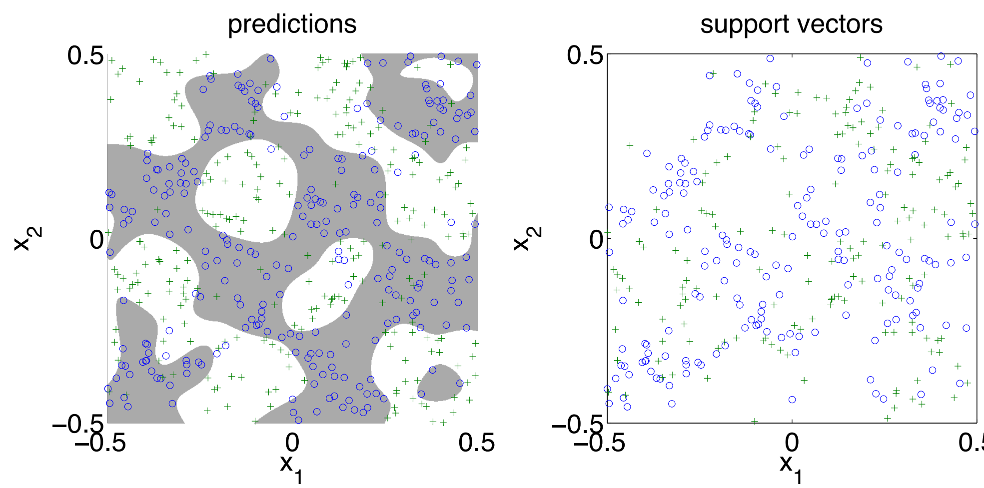
https://people.cs.umass.edu/~domke/courses/sml2010/06kernels.pdf

Non-linear decision boundaries

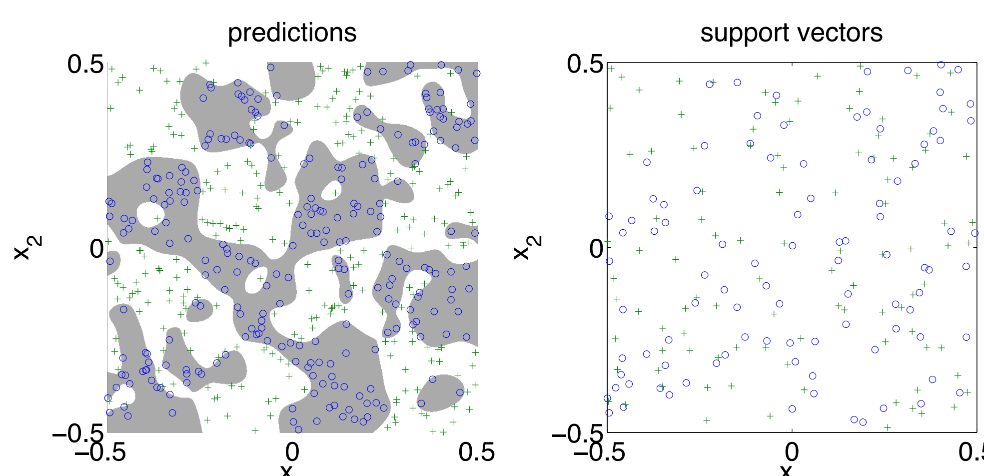
Dataset C (dataset B with noise), $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\left(-2||\mathbf{x} - \mathbf{v}||^2\right)$.



Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\left(-20||\mathbf{x} - \mathbf{v}||^2\right)$.



Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\left(-200||\mathbf{x} - \mathbf{v}||^2\right)$.



https://people.cs.umass.edu/~domke/courses/sml2010/06kernels.pdf

Kernelization

- Note that, when using non-linear kernel functions, we typically never compute **w** explicitly because we don't need it.

  - When predicting class of a new point **x**, we just need to know *k*(**x**, **x**$^{(i)}$) for each support vector i in our training set.

$$
\begin{aligned}
g(\mathbf{x}) &= \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \phi(\mathbf{x})^\top \phi(\mathbf{x}^{(i)}) + b \\
&= \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b
\end{aligned}
$$

Kernelization

- In fact, for some kernel functions, it may not even be possible to compute **w** explicitly.
- Hence, when training an SVM with non-linear kernel, we must solve in dual form, where we just need to find the optimal α.

$$L(\alpha) \quad = \quad \sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha^{(i)} \alpha^{(i')} y^{(i)} y^{(i')} k(\mathbf{x}^{(i)}, \mathbf{x}^{(i')})$$

Hyperparameter tuning

Hyperparameters

- How do we pick the right kernel for our ML problem?
- For a particular kernel, how do we decide the associated hyperparameters (e.g., γ)?
    - **Hyperparameters**: parameters that are not directly optimized during training but that can still impact training & testing performance.

Hyperparameter tuning

- Two main strategies:

  1. **Domain knowledge**: based on your knowledge of the application domain, you can decide which kernel is more sensible.

Hyperparameter tuning

- Two main strategies:

1. **Domain knowledge**: based on your knowledge of the application domain, you can decide which kernel is more sensible.

2. **Automatic tuning**: systematically search for the best kernel to maximize performance.

Automatic hyperparameter tuning

- The choice of the kernel and its associated hyper parameters can make a big impact on performance.
- However: directly optimizing on the test set is dangerous:
  - Any “increase” you observe might be spurious — you just got “lucky” in a way that would not generalize to unseen data.

Automatic hyperparameter tuning

- To avoid overly optimistic accuracy estimates, you should select a *subset of the training data* on which to optimize.
    - This is sometimes called a **validation set**.

Exercise

Exercise

- For a soft-margin SVM with cost $C > 0$, where the SVM is trained on linearly separable data with a linear kernel:
  - Minimize: $$\frac{1}{2}\mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^{n} \xi^{(i)}$$
  - Subject to: $$y^{(i)} \left(\mathbf{x}^{(i)\top} \mathbf{w} + b \right) \geq 1 - \xi^{(i)}$$
  - Will the optimal hyperplane ever *not* perfectly separate the data? Describe why not, or show an example of when it would.