CS 4342: Class 16

Jacob Whitehill

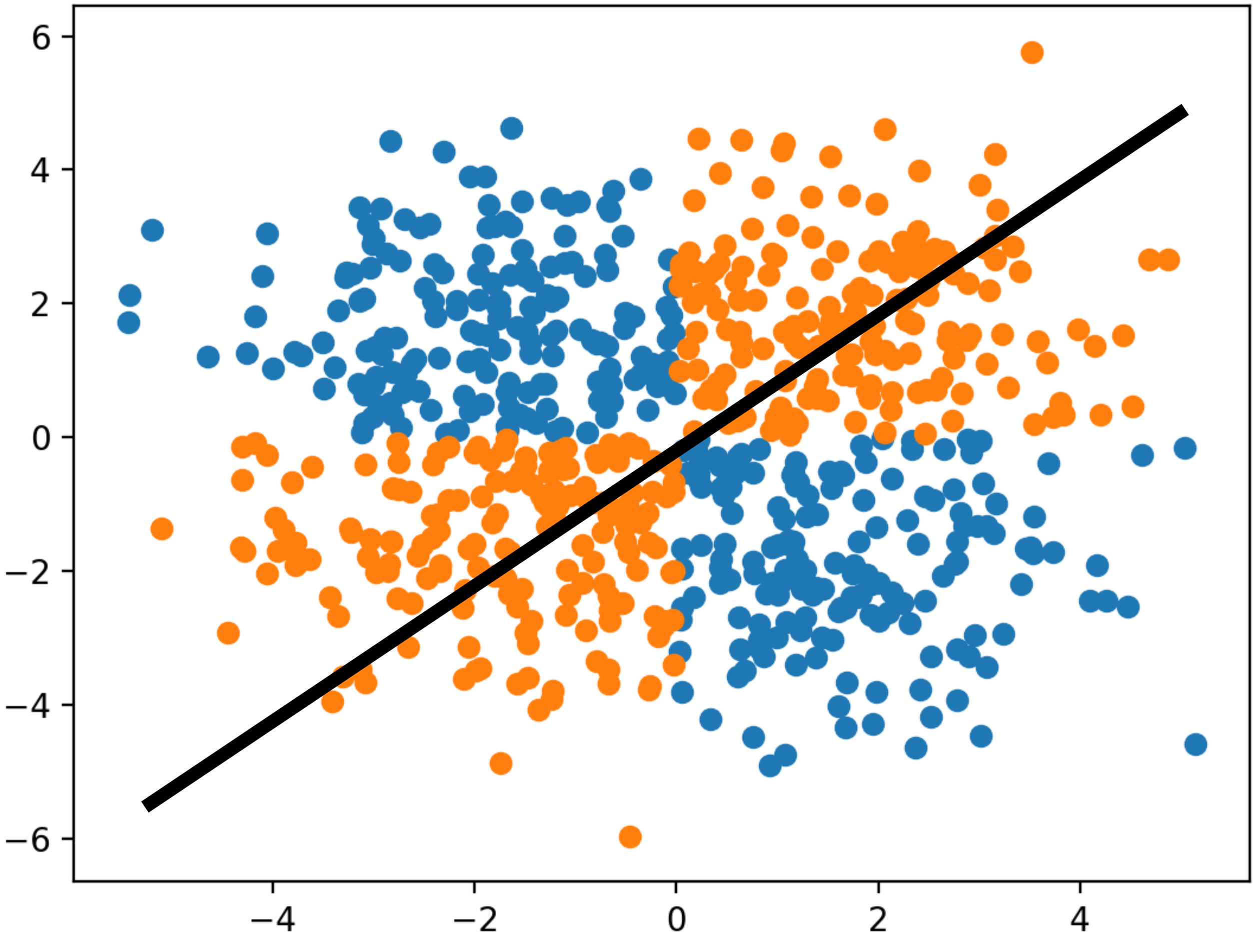
Kernel trick

Linearly inseparable data

- SVMs use a hyperplane to separate data in two classes.

- But what if the data are **linearly inseparable**, e.g.:

- No matter what **w**, *b* we choose, the SVM will never do a good job of classifying the data.

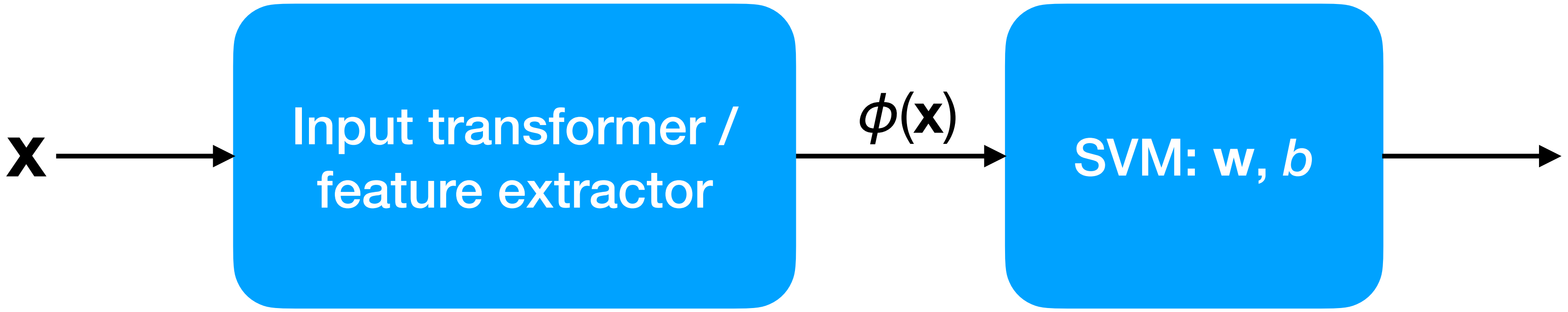


“XOR” problem

Feature transformations

- But what if we somehow transformed the raw input **x** into some (possibly higher-dimensional) representation ϕ(**x**)?

- Might the classes become linearly separable then?



x

Input transformer / feature extractor

ϕ(**x**)

SVM: **w**, *b*

$\hat{y} = \mathrm{sgn}(\phi(\mathbf{x})^\top \mathbf{w} + b)$

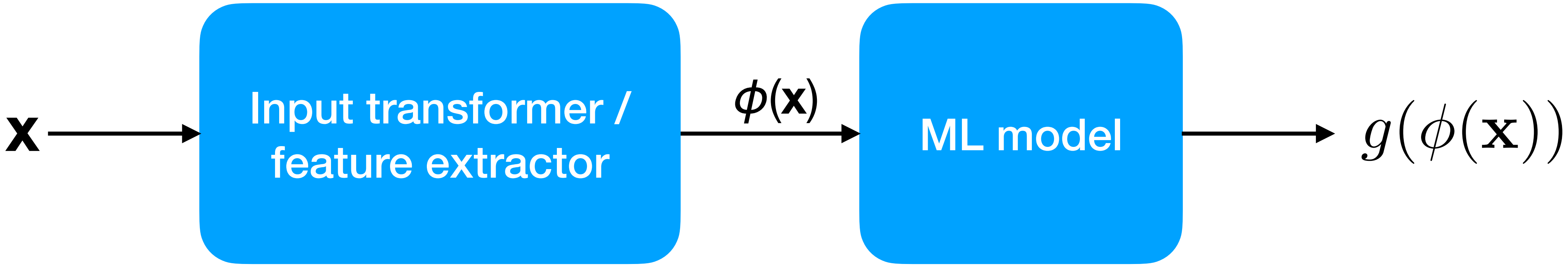
Can decide class arbitrarily if sgn = 0.

Feature transformations

- The conceptually simplest approach to training a classifier using transformed features is:
  - Transform each example $\mathbf{x}$ into $\phi(\mathbf{x})$.
  - Train on the transformed data $\phi(\mathbf{x}^{(1)})$, …, $\phi(\mathbf{x}^{(n)})$

- At test time:
  - Transform the test point $\mathbf{x}$ to $\phi(\mathbf{x})$; then classify $\phi(\mathbf{x})$.

- This can be done for *any* ML model.



Feature transformations

- To train a model in this way, we could easily construct the design matrix of transformed examples:

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

- We can then pass $\tilde{\mathbf{X}}$ to the SVM solver:

```
svm = sklearn.svm.SVC(kernel='linear')
svm.fit(Xtilde, y)
```

* Note that sklearn actually expects the design matrix to be examples x features, which is the transpose of how I define it in this course.

Feature transformations

- While this works fine in principle, for certain kinds of models — those that can be **kernelized —** the process can be made:
  - More efficient.
  - More powerful.
- SVMs are probably the most prominent kernelizable ML model…

Kernelization

- Recall that, in an SVM, the optimal **w** will always be a **linear combination** of the data points $\mathbf{x}^{(i)}$, weighted by the $\alpha^{(i)}$.

- Only the support vectors — those examples $\mathbf{x}^{(i)}$ such that $\alpha^{(i)} > 0$ — will contribute to **w**:

$$\begin{array}{rcl}
L(\mathbf{w}, b, \alpha) & = & \frac{1}{2}\mathbf{w}^\top \mathbf{w} - \sum_{i=1}^n \alpha^{(i)} \left(y^{(i)} \left(\mathbf{x}^{(i)\top} \mathbf{w} + b - 1 \right) \right) \\
\frac{\partial L}{\partial \mathbf{w}} & = & \mathbf{w} - \sum_{i=1}^n \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \\
\implies \mathbf{w} & = & \sum_{i=1}^n \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)}
\end{array}$$

Kernelization

- By differentiating w.r.t. *b* and setting to 0, we can make a further deduction:

$$
\begin{aligned}
L(\mathbf{w}, b, \alpha) &= \frac{1}{2}\mathbf{w}^\top \mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} \left(y^{(i)} \left({\mathbf{x}^{(i)}}^\top \mathbf{w} + b - 1 \right) \right) \\
\frac{\partial L}{\partial b} &= -\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \\
\Longrightarrow \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} &= 0
\end{aligned}
$$

Kernelization

- After substituting for **w** and *b*, the Lagrangian can be simplified to yield:

$$
\begin{aligned}
L(\mathbf{w}, b, \alpha) &= \frac{1}{2}\mathbf{w}^\top \mathbf{w} - \sum_{i=1}^{n} \alpha^{(i)} \left(y^{(i)} \left(\mathbf{x}^{(i)^\top} \mathbf{w} + b - 1 \right) \right) \\
&= \frac{1}{2} \left| \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \right|^2 - \sum_{i=1}^{n} \alpha^{(i)} \left(y^{(i)} \left(\mathbf{x}^{(i)^\top} \left(\sum_{i'=1}^{n} \alpha^{(i')} y^{(i')} \mathbf{x}^{(i')} \right) + b - 1 \right) \right) \\
\implies L(\alpha) &= \sum_{i=1}^{n} \alpha^{(i)} - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha^{(i)} \alpha^{(i')} y^{(i)} y^{(i')} \mathbf{x}^{(i)^\top} \mathbf{x}^{(i')}
\end{aligned}
$$

Only a function of *α* now.

The training data occur only as inner products in the function *L* that we optimize.

Kernelization

- At test time, we compute the inner product between **x** and **w**:

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

Kernelization

- At test time, we compute the inner product between **x** and **w**:

$$\begin{aligned}\mathbf{x}^\top \mathbf{w} + b &= \mathbf{x}^\top \left(\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \right) + b \\ &= \sum_{i=1}^{n} \alpha^{(i)} y^{(i)} \mathbf{x}^\top \mathbf{x}^{(i)} + b\end{aligned}$$

- The result depends only on the inner products between the test point **x** and each of the support vectors $\mathbf{x}^{(i)}$.

Kernelization

- Both during training and testing, we only use each training point $\mathbf{x}^{(i)}$ as part of an inner product — *we never need the raw values themselves*.

- Similarly, 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):

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

Kernelization

- Both during training and testing, we only use each training point $\mathbf{x}^{(i)}$ as part of an inner product — *we never need the raw values themselves*.

- Similarly, 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 store a matrix containing $\phi(\mathbf{x}^{(i)})$ for every training example $\mathbf{x}^{(i)}$…:

$$\tilde{\mathbf{X}} = \left[\begin{array}{ccc} | & & | \\ \phi\left(\mathbf{x}^{(1)}\right) & \ldots & \phi\left(\mathbf{x}^{(n)}\right) \\ | & & | \end{array}\right]$$

m* x *n

Kernelization

- …instead store the **kernel matrix** containing all pairs of inner products of the training data:

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

n* x *n

Kernelization

- …instead store the **kernel matrix** containing all pairs of inner products of the training data:

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

- The kernel matrix **K** can be much smaller than $\tilde{\mathbf{X}}$ if $n < m$.

Kernelization

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

```
svm = sklearn.svm.SVC(kernel='precomputed')
K = ...                                 # $\mathbf{K} = \tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}$
svm.fit(K, y)
```

Kernelization

- **K** is an *n* x *n* matrix, where *n* is # training examples.
- Suppose *n*=1000, *m*=10000 (e.g., 100x100 pixels).
- Storing each $\phi(\mathbf{x}^{(i)})$ explicitly would take *O*(10,000,000) bytes.
- Storing just **K** will take *O*(1,000,000) bytes — 10x less!
- Training the SVM in dual form can also be much *faster* (for $n \ll m$).

Kernelization

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

$$\mathbf{K} = \left[\begin{array}{ccc} k(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \dots & k(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}) \\ & \ddots & \\ k(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}) & \dots & k(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}) \end{array} \right]$$

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 each example has 2 dims, and you want ϕ to compute poly. features of **x** of degree 2, i.e.,

$$\phi\left(\left[\begin{array}{c} u \\ v \end{array}\right]\right) = \left[\begin{array}{c} 1 \\ \sqrt{2}u \\ \sqrt{2}v \\ \sqrt{2}uv \\ u^2 \\ v^2 \end{array}\right]$$

- 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:

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

Kernel example

- On the other hand, we can derive that:

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

Kernel example

- On the other hand, we can derive that:

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

Kernel example

- On the other hand, we can derive that:

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

Kernel example

- On the other hand, we can derive that:

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

Kernel example

- On the other hand, we can derive that:

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

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.

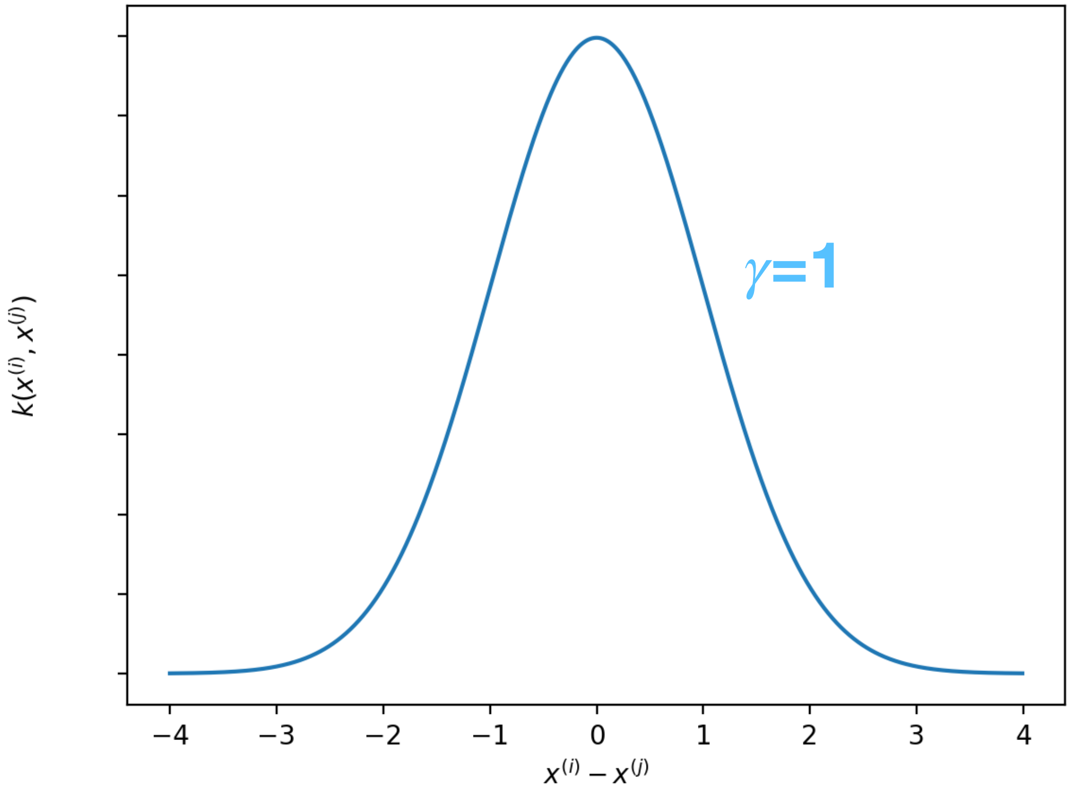
- But the transformation can be *anything — we may not even care what it is, as long as it theoretically exists.*

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 larger inner-product than two vectors far apart:

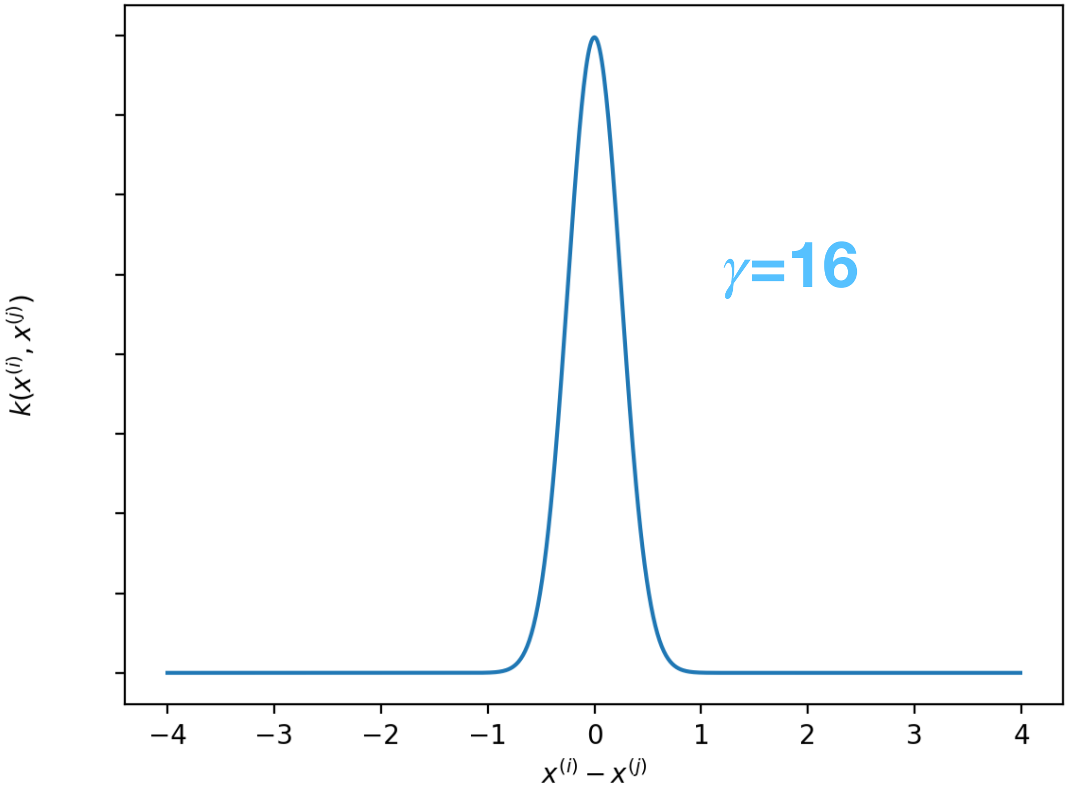


Kernel functions

- 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 inner-product decreases as a function of the distance between the two input vectors:



Jacob Whitehill, WPI

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

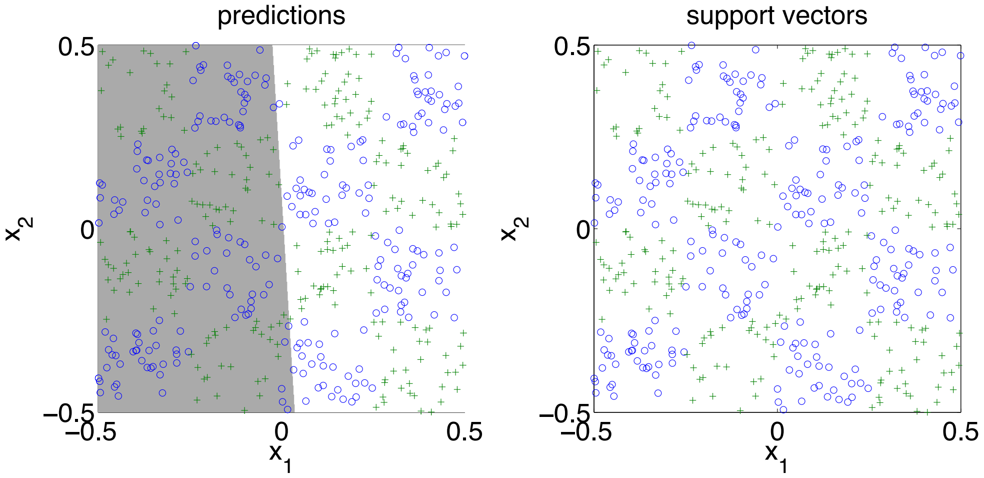
- SVMs **always** try to separate the positive from the negative examples using a **hyperplane** — a linear decision boundary.

- But the hyperplane might exist in a very different (transformed) space than the raw input data.

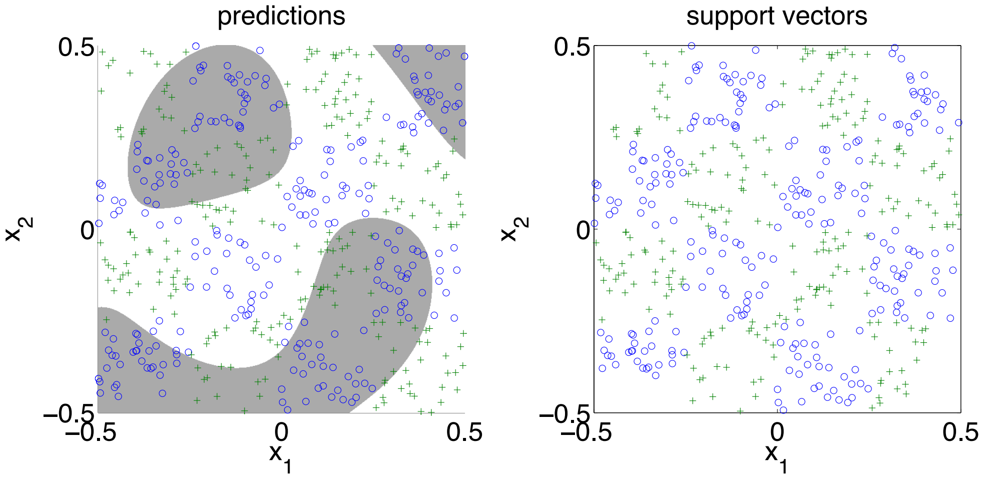
- In the original input space, the decision boundary can be non-linear.

Non-linear decision boundaries

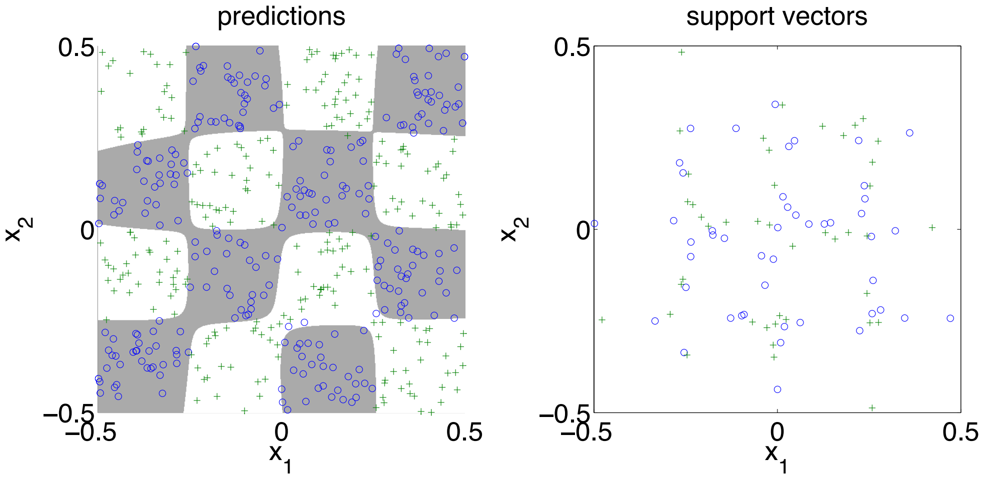
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



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



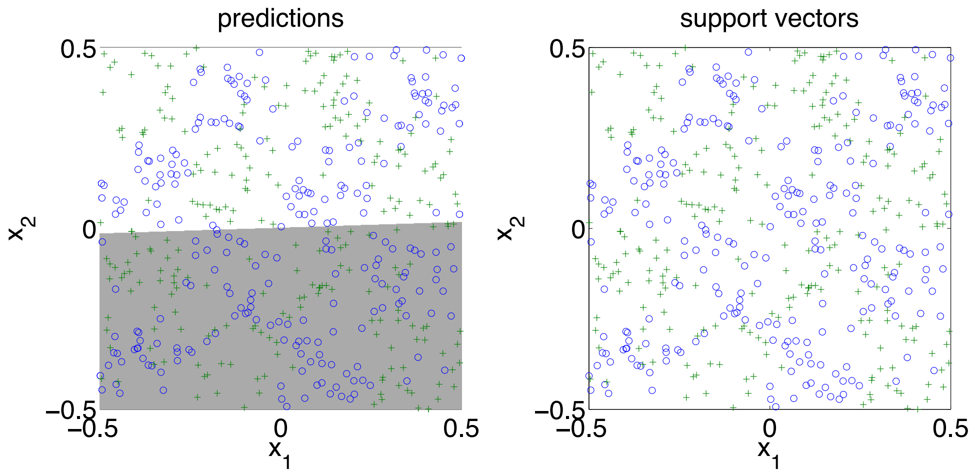
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^{10}$.



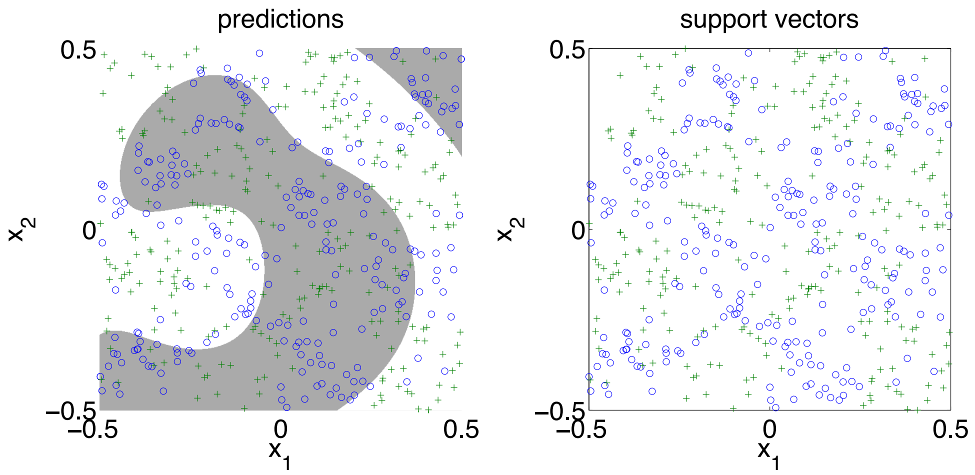
https://people.cs.umass.edu/~domke/courses/sml2010/06kernels.pdf

Non-linear decision boundaries

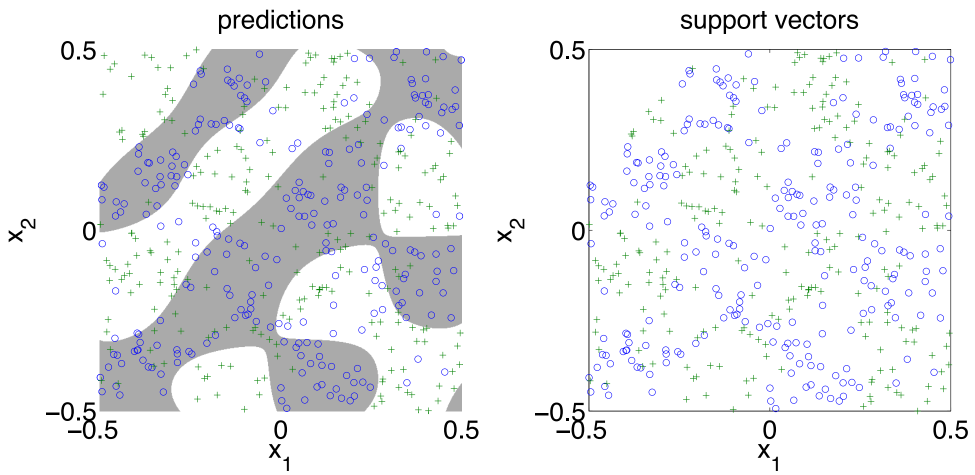
Dataset C (dataset B with noise), $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



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



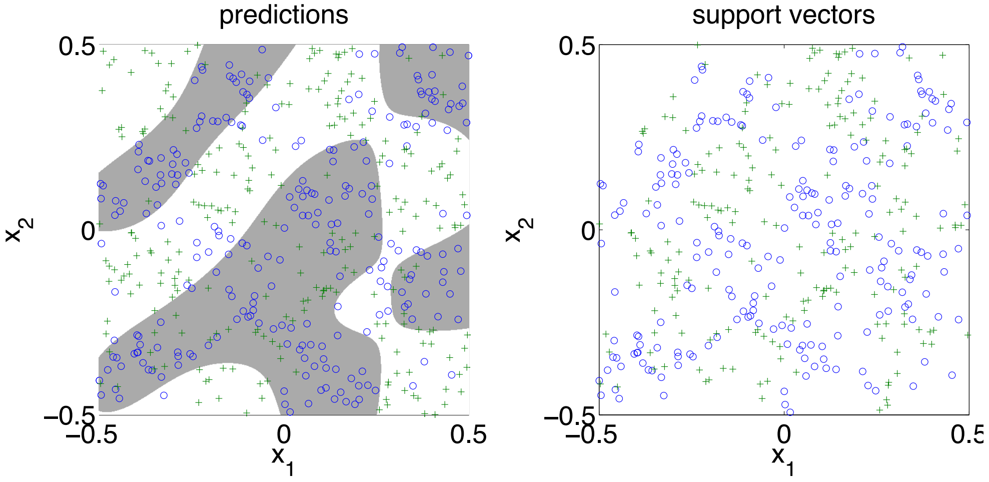
Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^{10}$.



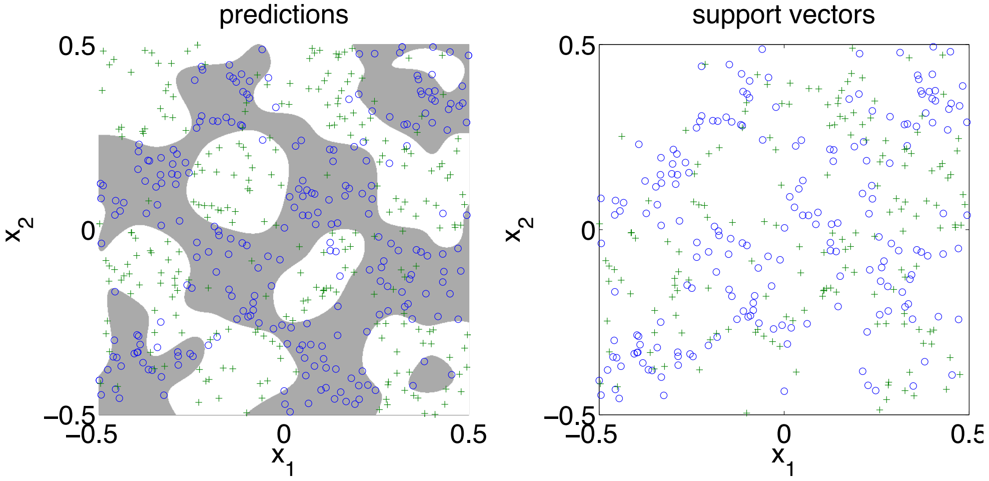
https://people.cs.umass.edu/~domke/courses/sml2010/06kernels.pdf

Non-linear decision boundaries

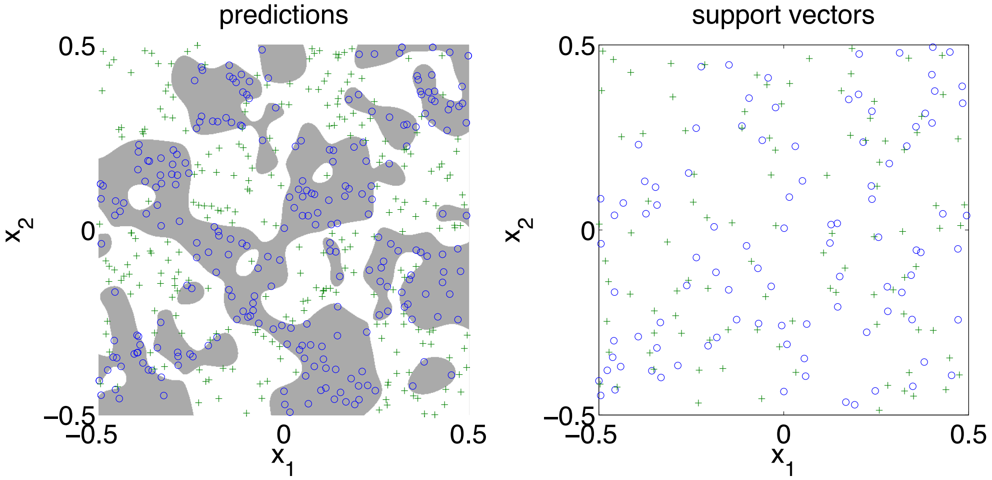
Dataset C (dataset B with noise), $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\big(-2||\mathbf{x} - \mathbf{v}||^2\big)$.



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



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



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

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., ɣ)?

  - **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.

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