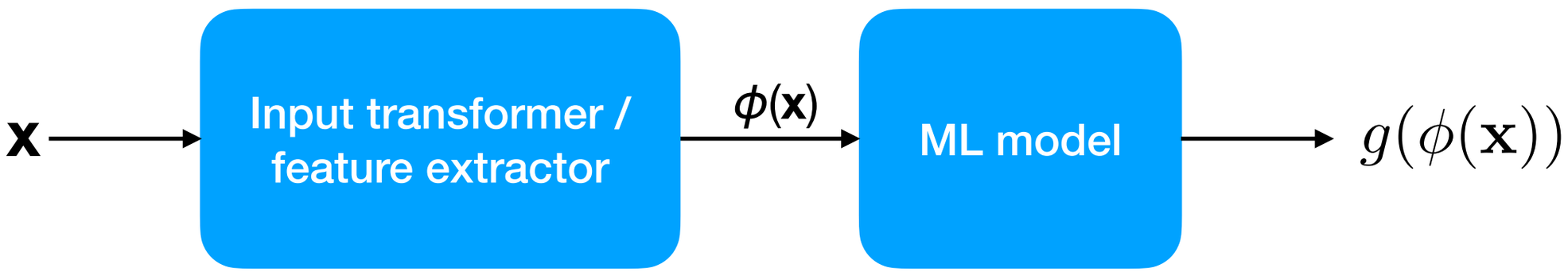
CS 4342: Class 17

Jacob Whitehill

Kernel trick

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 $\phi(\mathbf{x}^{(1)}), \ldots, \phi(\mathbf{x}^{(n)})$
- 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:

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

- 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**:

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

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 $\phi(\mathbf{x}^{(j)})$ (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 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):

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

Kernelization

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

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

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

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

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:

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

Kernel functions

- Using the kernel function, the Lagrangian used for training the SVM becomes:

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

- At test time, we compute:

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

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.,

$$\phi\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) = \begin{bmatrix} 1 \\ \sqrt{2}u \\ \sqrt{2}v \\ \sqrt{2}uv \\ u^2 \\ v^2 \end{bmatrix}$$

- 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:

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

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.

Exercise

- Suppose that we train a *linear* SVM and obtain the following two support vectors:
  - $\mathbf{x}^{(1)}=[1, 3]^T$, $y^{(1)}=+1$, $\alpha^{(1)}=.11$
  - $\mathbf{x}^{(2)}=[-2, 0]^T$, $y^{(2)}=-1$, $\alpha^{(2)}=.11$
- Suppose $b=-.33$
- What is the SVM's output $\mathbf{x}^T\mathbf{w}+b$ for $\mathbf{x}=[-1, -2]^T$?

Exercise

- Suppose that we train a *poly-2* SVM ($k(\mathbf{x}, \mathbf{x}')=(1+\mathbf{x}^T\mathbf{x}')^2$) and obtain the following two support vectors:
  - $\mathbf{x}^{(1)}=[1, 3]^T$, $y^{(1)}=+1$, $\alpha^{(1)}=.01$
  - $\mathbf{x}^{(2)}=[-2, 0]^T$, $y^{(2)}=-1$, $\alpha^{(2)}=.01$
- Suppose $b=-.67$
- What is the SVM's output $\mathbf{x}^T\mathbf{w}+b$ for $\mathbf{x}=[-1, -2]^T$?

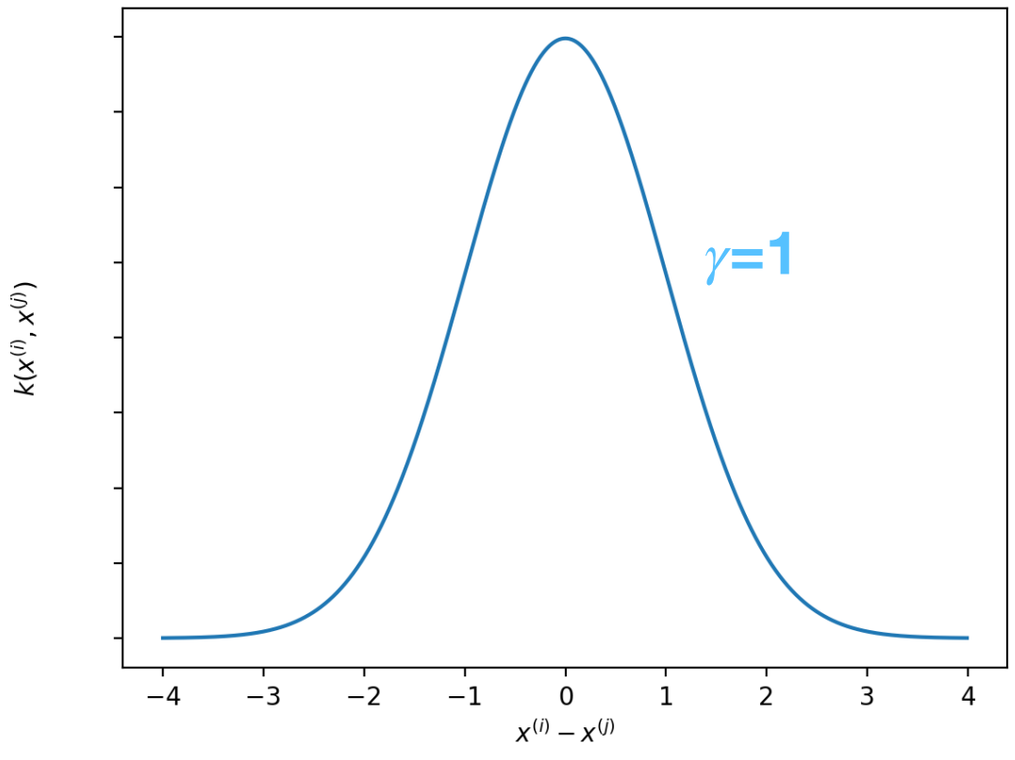
Gaussian RBF SVM

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:

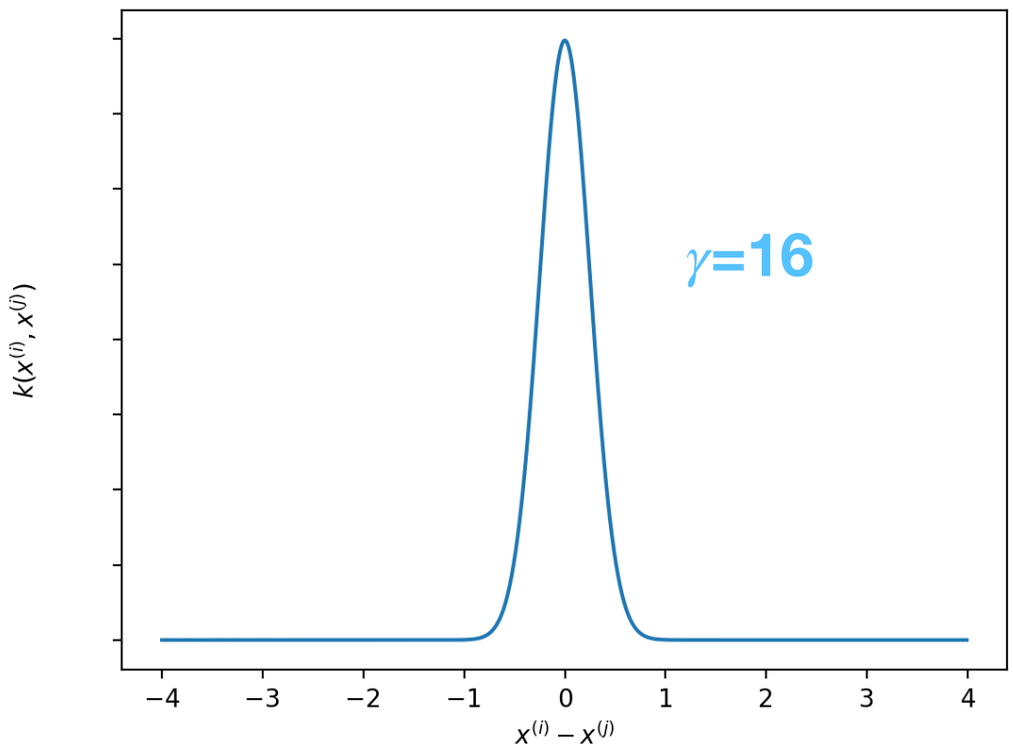


Kernel functions

- 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:



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.

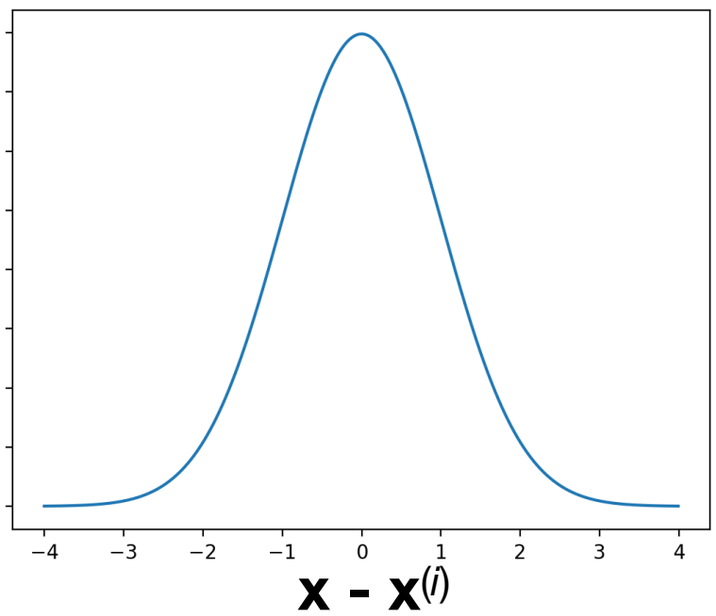
RBF SVM at test time

- An RBF-SVM's output on some example **x** will be:

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

- where:

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



RBF SVM at test time

- An RBF-SVM's output on some example **x** will be:

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

- Procedure:
  - Compute the kernel response of the example **x** with each of our support vectors $\mathbf{x}^{(i)}$.

RBF SVM at test time

- An RBF-SVM's output on some example **x** will be:

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

- Procedure:
  - Compute the kernel response of the example **x** with each of our support vectors $\mathbf{x}^{(i)}$.
  - Multiply the kernel response by *i*'s label $y^{(i)}$ and the dual variable $\alpha^{(i)}$.

RBF SVM at test time

- An RBF-SVM's output on some example **x** will be:

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

- Procedure:
  - Compute the kernel response of the example **x** with each of our support vectors $\mathbf{x}^{(i)}$.
  - Multiply the kernel response by *i*'s label $y^{(i)}$ and the dual variable $\alpha^{(i)}$.
  - Sum across all support vectors.

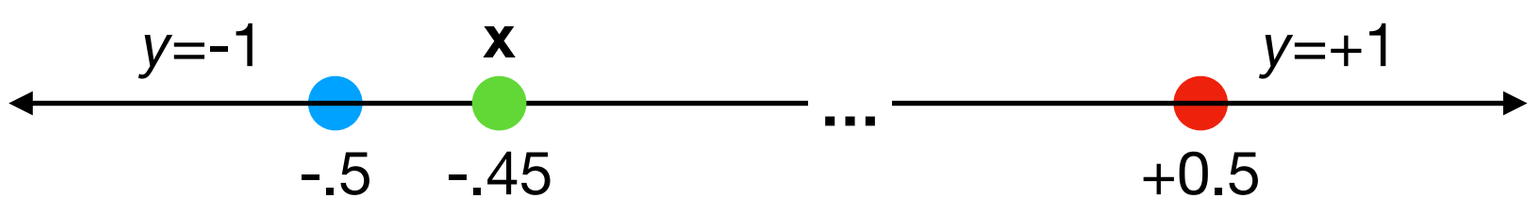
Exercise

- Suppose there are just two n=2 examples in our training set, both of which are support vectors (SV).
- Suppose α=1 for each SV; let b=0.
- Let **x** be a test point.

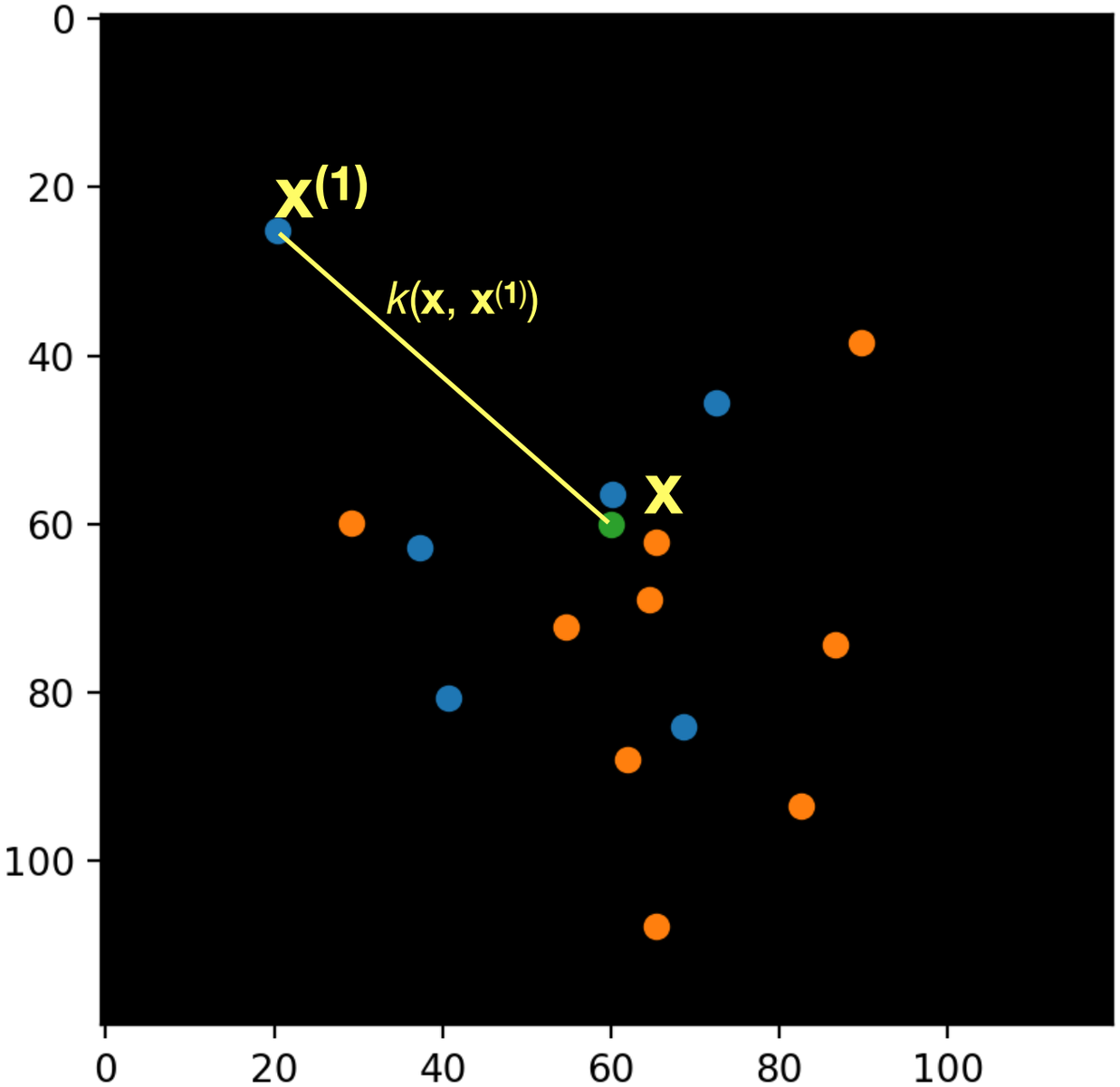
$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

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

- What will be the prediction $g(\mathbf{x})$ when γ=1?
  What will be the prediction $g(\mathbf{x})$ when γ=100?
  How does the impact of the red point on $g(\mathbf{x})$ change as γ grows?



Example

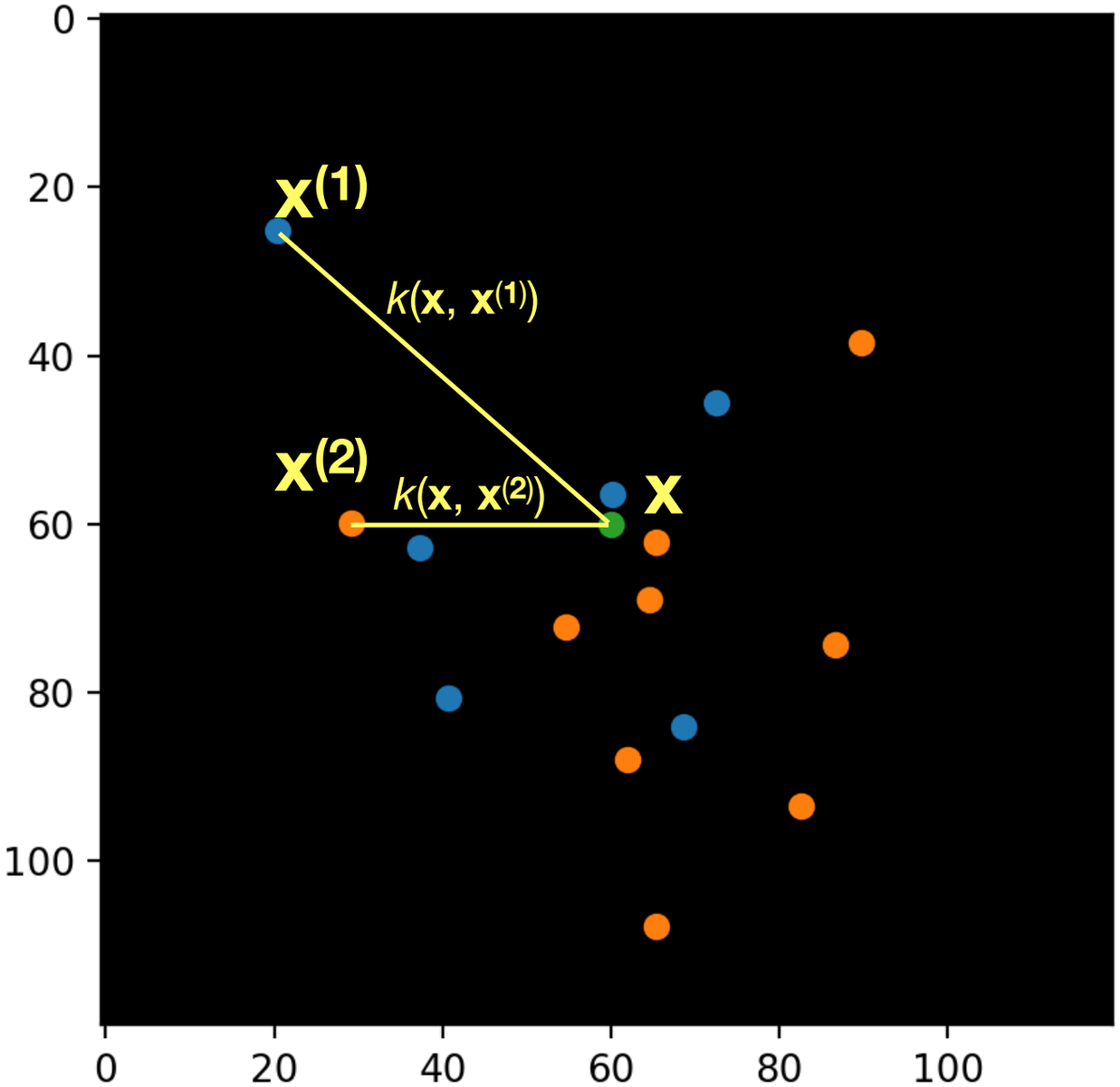


$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

$$\alpha^{(1)} y^{(1)} k(\mathbf{x}, \mathbf{x}^{(1)})$$

The value of *k* can be thought of as an inverse distance.

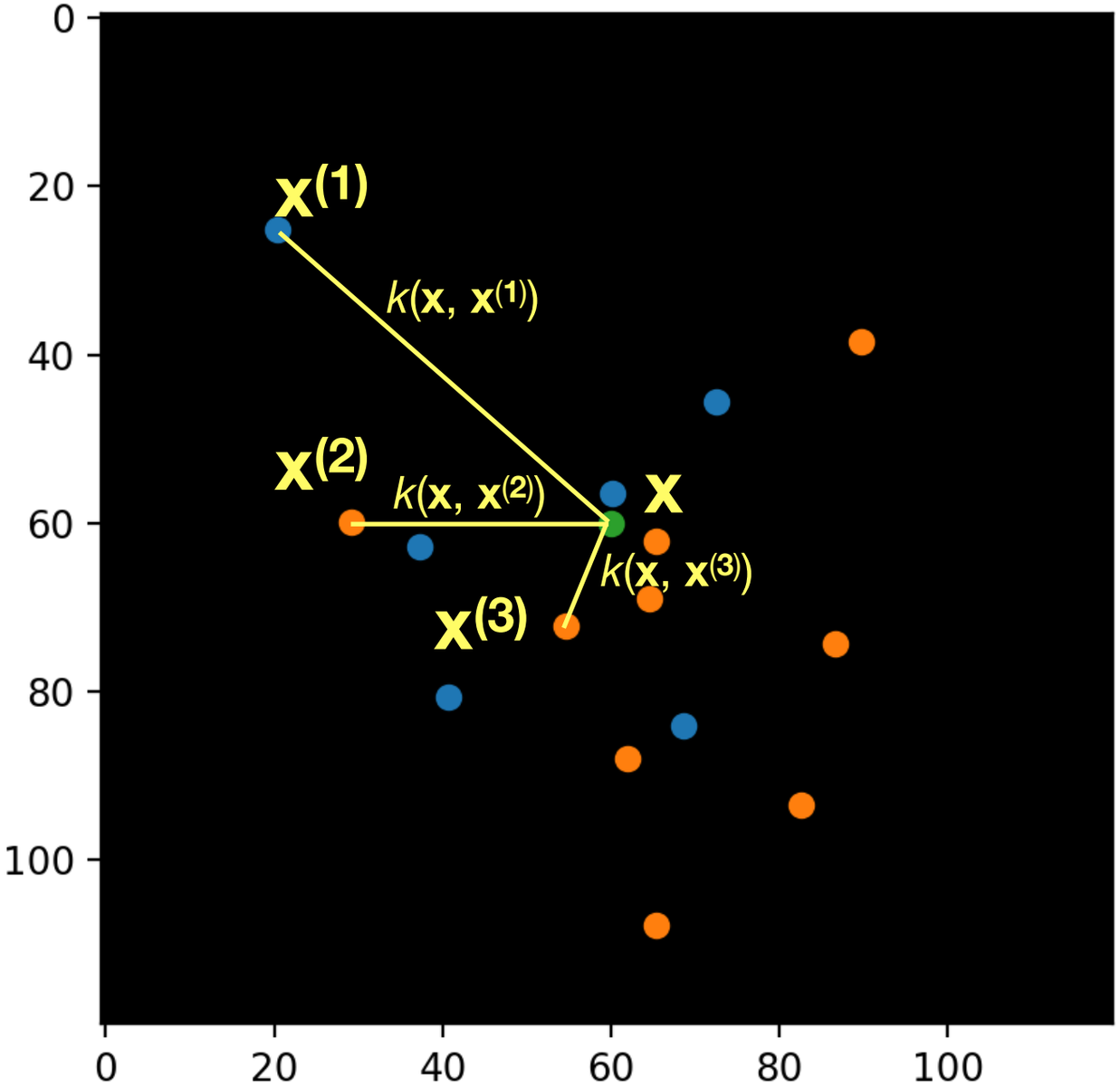
Example



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

$$\alpha^{(2)} y^{(2)} k(\mathbf{x}, \mathbf{x}^{(2)})$$

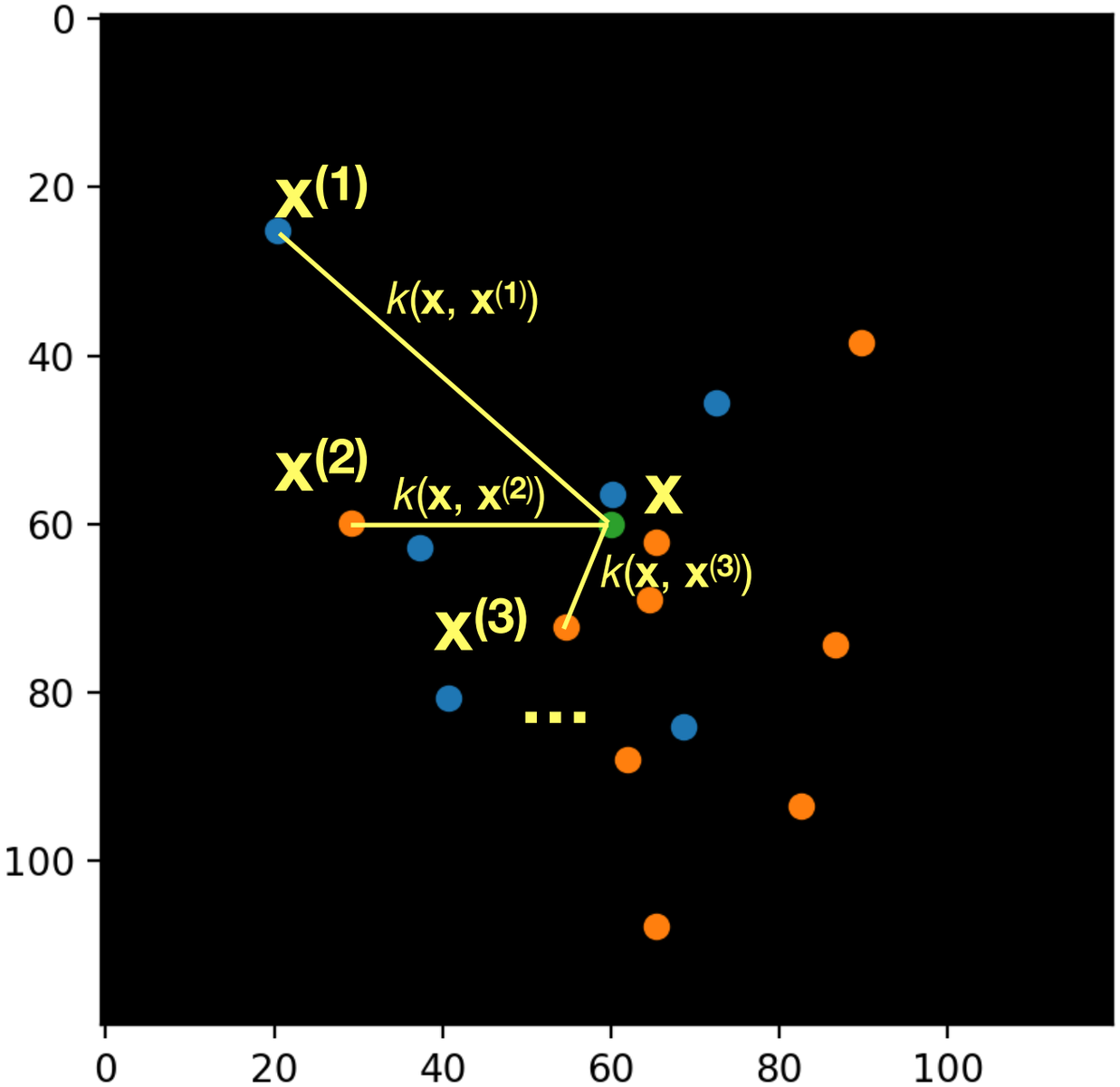
Example 2



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

$$\alpha^{(3)} y^{(3)} k(\mathbf{x}, \mathbf{x}^{(3)})$$

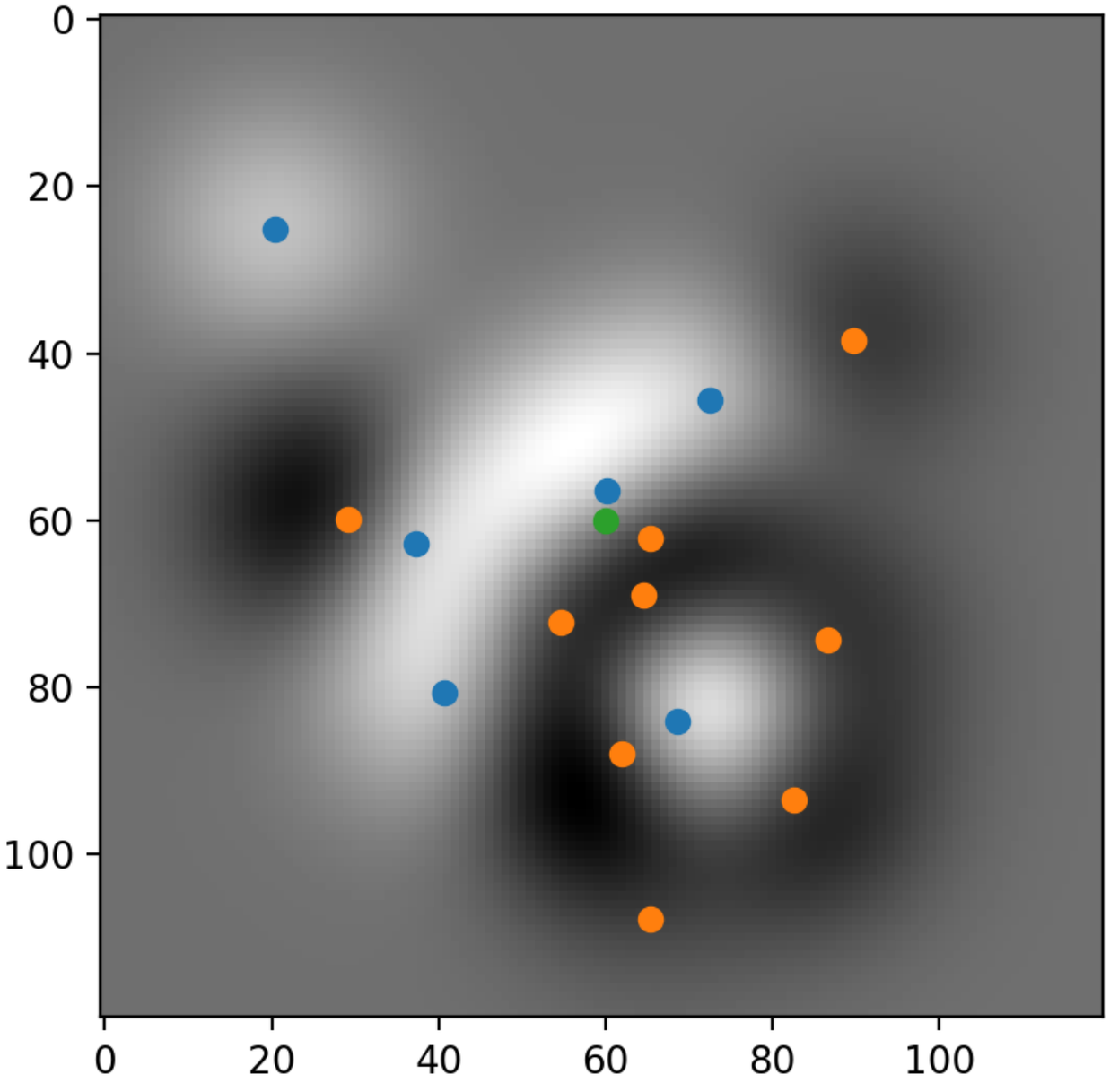
Example



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

$$\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b$$

Example



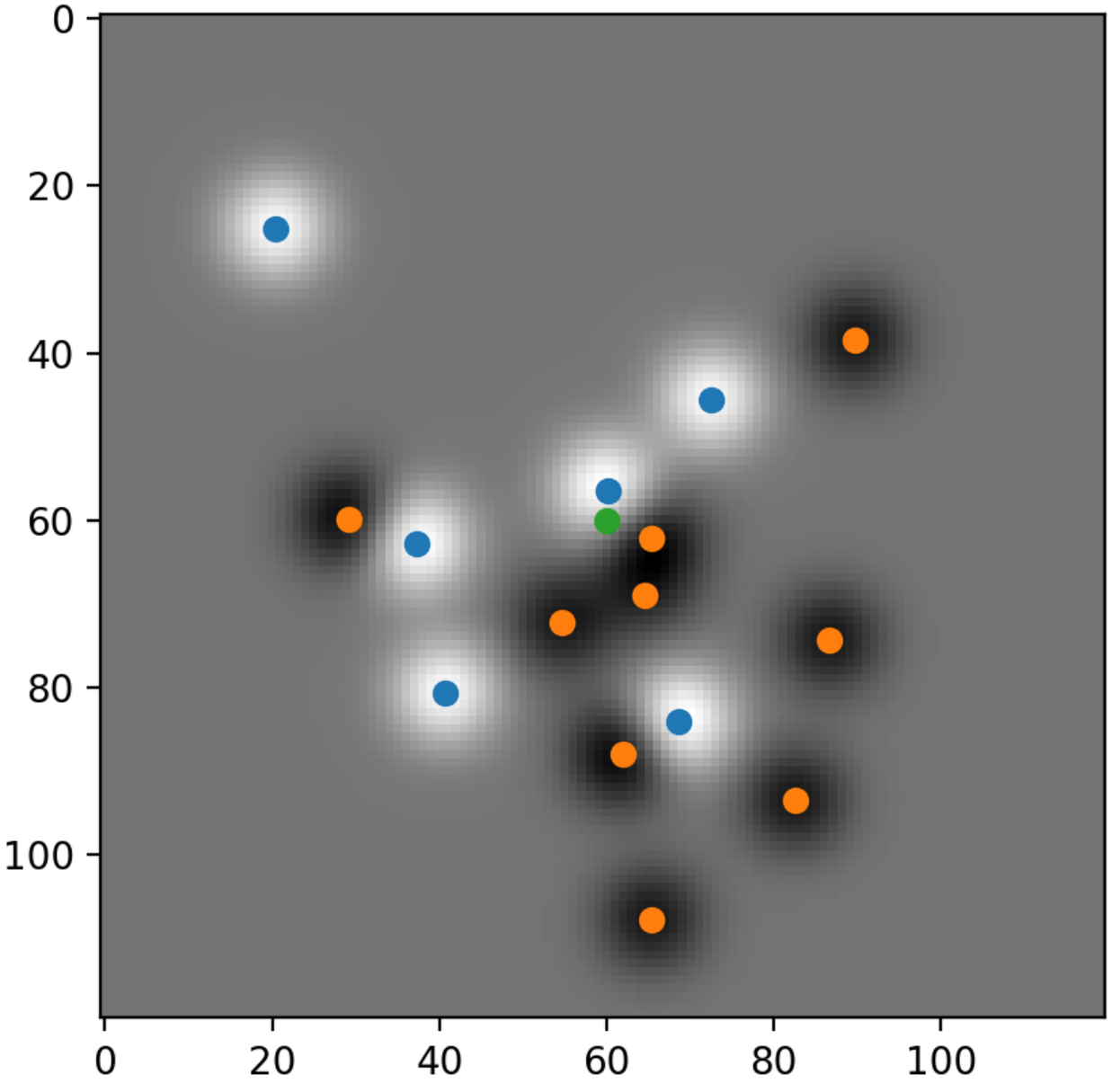
$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

This graph shows, for each possible x, the value of:

$$\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b$$

for γ = $10^{0.25}$

Example



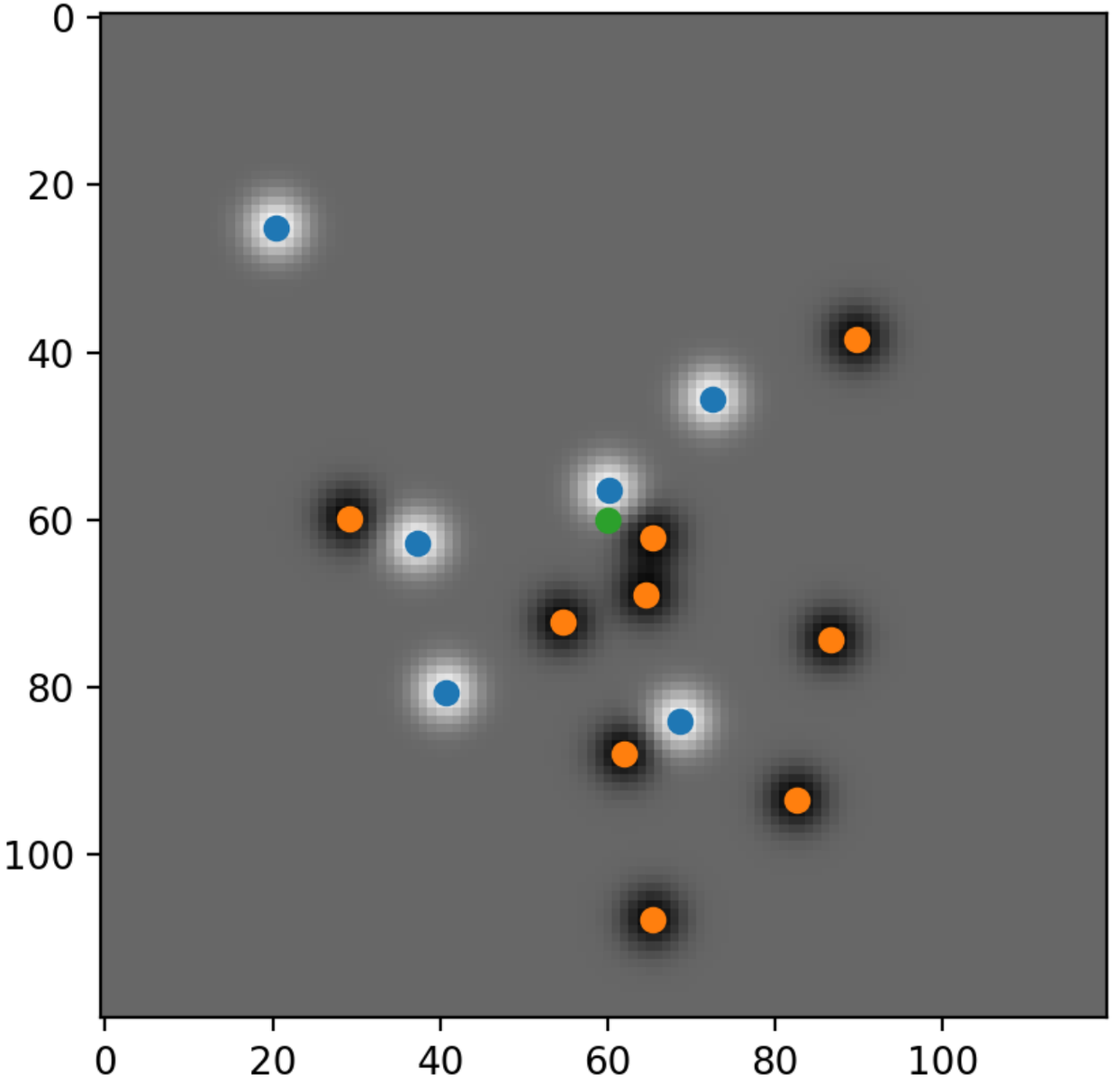
$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

This graph shows, for each possible x, the value of:

$$\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b$$

for γ = 10^1

Example



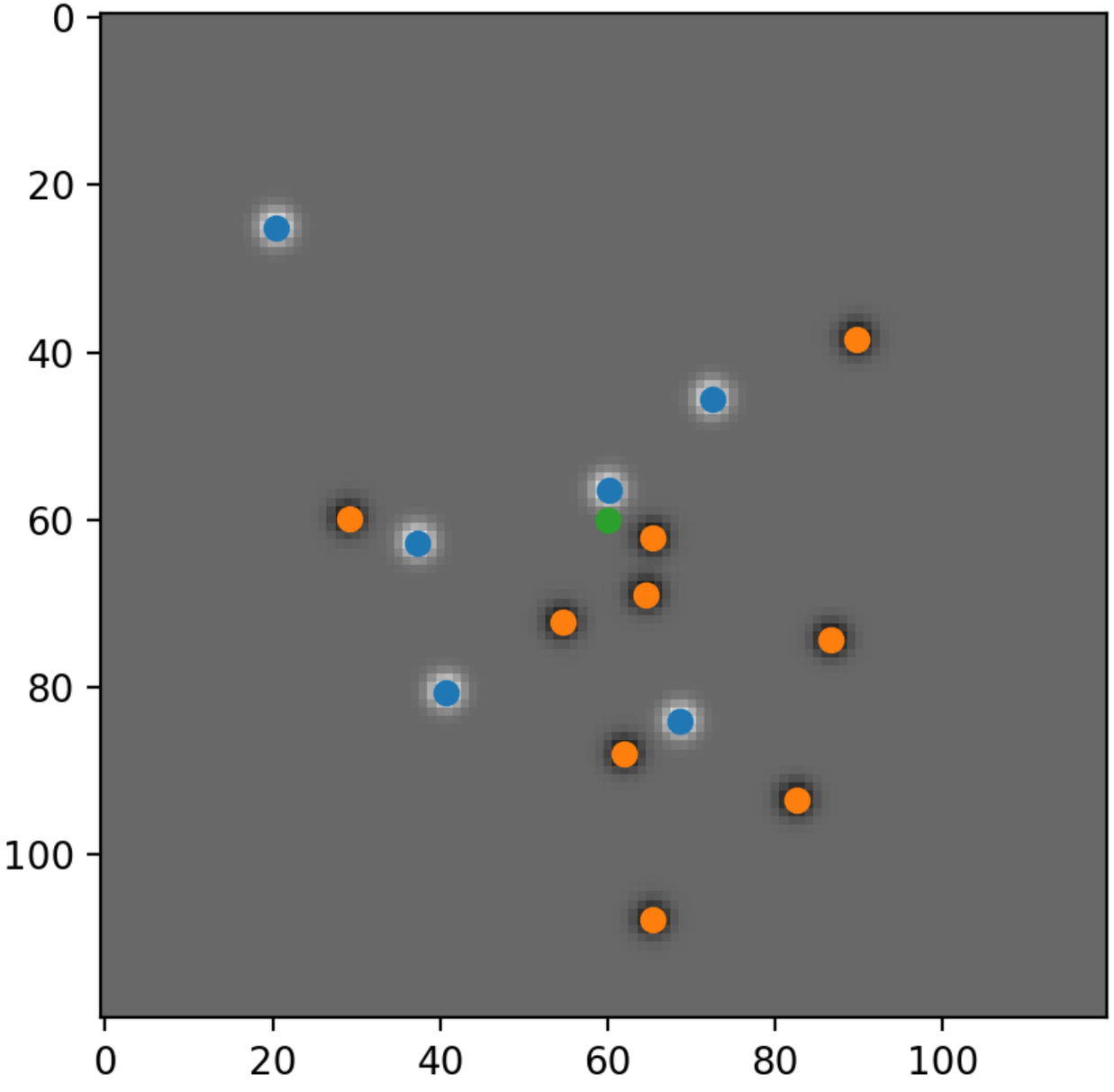
$$k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma\left(\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\right)^2\right)$$

This graph shows, for each possible x, the value of:

$$\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b$$

for $\gamma = 10^{1.5}$

Example



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

This graph shows, for each possible x, the value of:

$$\sum_{i=1}^{n} \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b$$

for γ = 10^2

For very large γ, only the nearest neighbor to x will impact g(x).

Nearest neighbors

Nearest neighbor

- **Nearest neighbor** is both a classification and a regression method.

- It is one of the simplest ML models.

- Algorithm:

  - To predict the label of a new data point **x**, find the data point $\mathbf{x}^{(i)}$ in the training set closest to **x**:

$$\arg\min_i |\mathbf{x}^{(i)} - \mathbf{x}|$$

  - Return the closest example *i*'s associated label $y^{(i)}$.

k nearest neighbors

- Instead of examining just the single data point closest to **x**, we can look at the *k* neighbors closest to **x**.

- To predict the label of **x**, we can either vote (for classification) or compute the average (for regression) of the *k* neighbors' labels.

k nearest neighbors

- In **`sklearn`**, use either:
  - **`sklearn.neighbors.KNeighborsClassifier(n_neighbors)`**
  - **`sklearn.neighbors.KNeighborsRegressor(n_neighbors)`**

k nearest neighbors

- While very simple, *k* nearest neighbors (kNN) has three significant drawbacks:

    1. The machine must always store the entire training set to make decisions (high storage costs).

$$\arg\min_i |\mathbf{x}^{(i)} - \mathbf{x}|$$

k nearest neighbors

- While very simple, *k* nearest neighbors (kNN) has three significant drawbacks:

    1. The machine must always store the entire training set to make decisions (high storage costs).

    2. The machine can be slow since the distance to *every* training example must be computed.

$$\arg\min_i |\mathbf{x}^{(i)} - \mathbf{x}|$$

k nearest neighbors

- While very simple, *k* nearest neighbors (kNN) has three significant drawbacks:

    1. The machine must always store the entire training set to make decisions (high storage costs).

    2. The machine can be slow since the distance to *every* training example must be computed.

        - There is a lot of research on *approximate* nearest neighbors — with *high probability*, find a neighbor very close to **x**.

k nearest neighbors

- While very simple, *k* nearest neighbors (kNN) has three significant drawbacks:

    1. The machine must always store the entire training set to make decisions (high storage costs).

    2. The machine can be slow since the distance to *every* training example must be computed.

        - Note that RBF-SVMs are generally faster since *only the support vectors* need to be stored & compared.

k nearest neighbors

- While very simple, *k* nearest neighbors (kNN) has three significant drawbacks:

1. The machine must always store the entire training set to make decisions (high storage costs).

2. The machine can be slow since the distance to *every* training example must be computed.

3. For high-dimensional inputs, many training examples are needed to “fill” the space.

Curse of dimensionality

- Suppose we want to have at least 10 training examples along each dimension of our input space.
  - 1 dimension ==> need 10 examples
  - 2 dimensions ==> need 10^2 examples

Curse of dimensionality

- Suppose we want to have at least 10 training examples along each dimension of our input space.
  - 1 dimension ==> need 10 examples
  - 2 dimensions ==> need 10^2 examples
  - …
  - *d* dimensions ==> need 10^d examples

Curse of dimensionality

- Suppose we want to have at least 10 training examples along each dimension of our input space.
  - 1 dimension ==> need 10 examples
  - 2 dimensions ==> need 10^2 examples
  - …
  - *d* dimensions ==> need 10^d examples
- Without good “coverage” of the input space, the kNN machine’s predictions may be very inaccurate.