COM SCI M148, Lecture 6:
K-Nearest Neighbors (Regression)

Prof. Nathan Szymanski
nszymanski@seas.ucla.edu
Department of Materials Science

UCLA

Samueli
School of Engineering

Recap: Models we've covered so far

$$y = f(x)$$

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

Multiple variables (x_i) – each linear

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n$$

Multiple terms (x^n) of one variable

We are assuming $f(x)$ has a very *specific form*

And our goal is to find the best β_i given that form

Recap: Models we've covered so far

$$y = f(x)$$

$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$ **Multiple variables (x_i)** – each linear

$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n$ **Multiple terms (x^n)** of one variable

$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2$ **(MSE loss)** → $\beta = (X^T X)^{-1} X^T y$

$\mathcal{L} = \text{MSE} + \lambda \sum_{i=1}^{n} \beta_i^2$ **(Regularization)** → $\beta = (X^T X + \lambda I)^{-1} X^T y$

Recap: Models we've covered so far

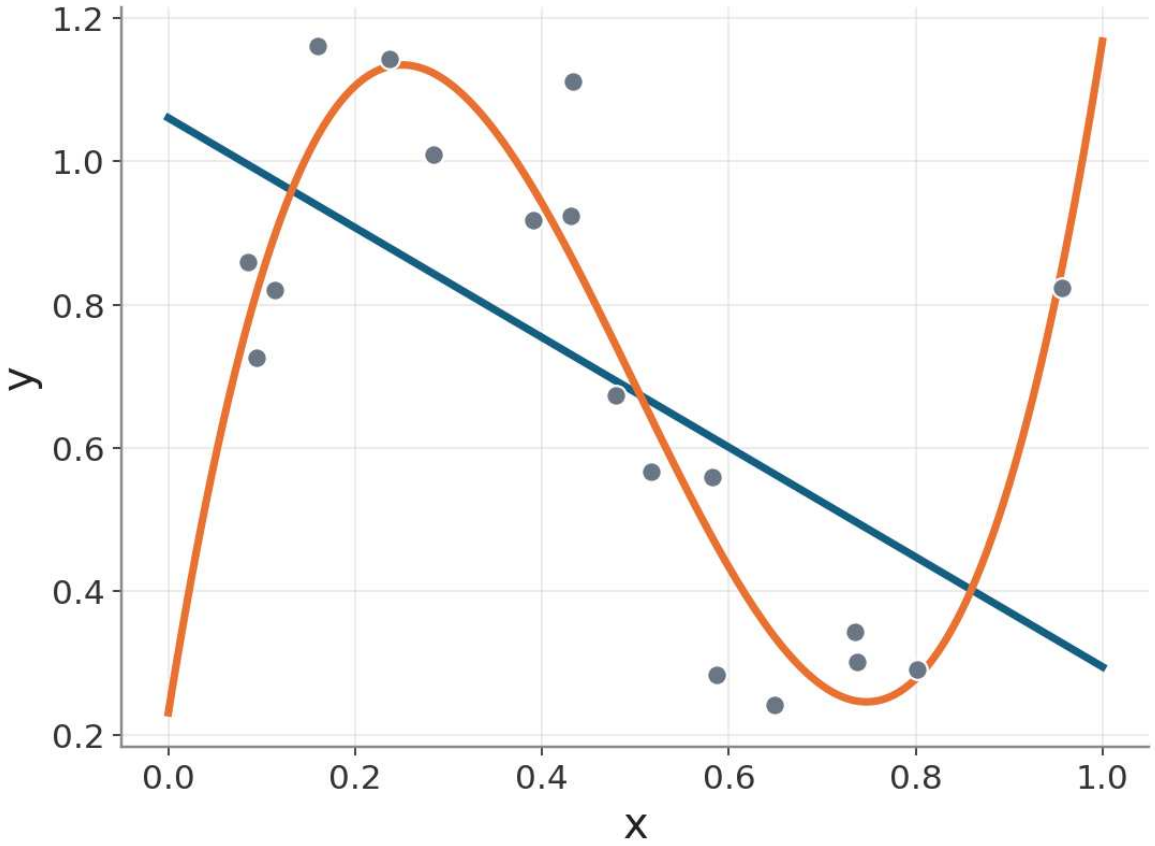
$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n$$

Once we solve for the coefficients (β), *they are fixed*.
We can "throw away" the training data thereafter and just use our known model to make predictions on new data

$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|^2$ **(MSE loss)** ⟶ $\beta = (X^T X)^{-1} X^T y$

$\mathcal{L} = \text{MSE} + \lambda \sum_{i=1}^{n} \beta_i^2$ **(Regularization)** ⟶ $\beta = (X^T X + \lambda I)^{-1} X^T y$

Assuming $f(x)$ has a specific form





"I think $f(x)$ is a **cubic** function!"

Both assume a **specific form**



"I think $f(x)$ is a **linear** function!"

Regularization can help us choose terms in an automated way:

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n$$

But there is still an assumption here: that $f(x)$ is a polynomial

Parametric *vs.* Non-parametric models

Parametric models

- We assume $f(x)$ has a specific form.
- The fitted model is completely described by a finite set of parameters (β); the training data is not needed for prediction.

Today: *Non-Parametric* models

- We make *no assumption* about the form $f(x)$ takes.
- Predictions *depend directly* on the stored training data, which is actively referenced when making predictions.

K-Nearest Neighbors (KNN)

Here we'll make predictions based on "neighboring" samples

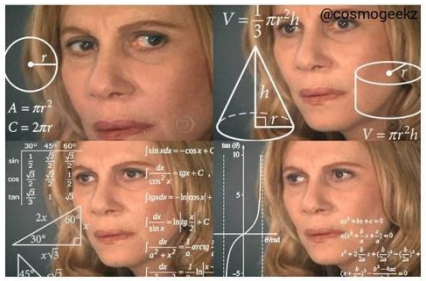
An example: Housing prices

Suppose I asked you are living in LA and would like to sell your house…
But for how much?





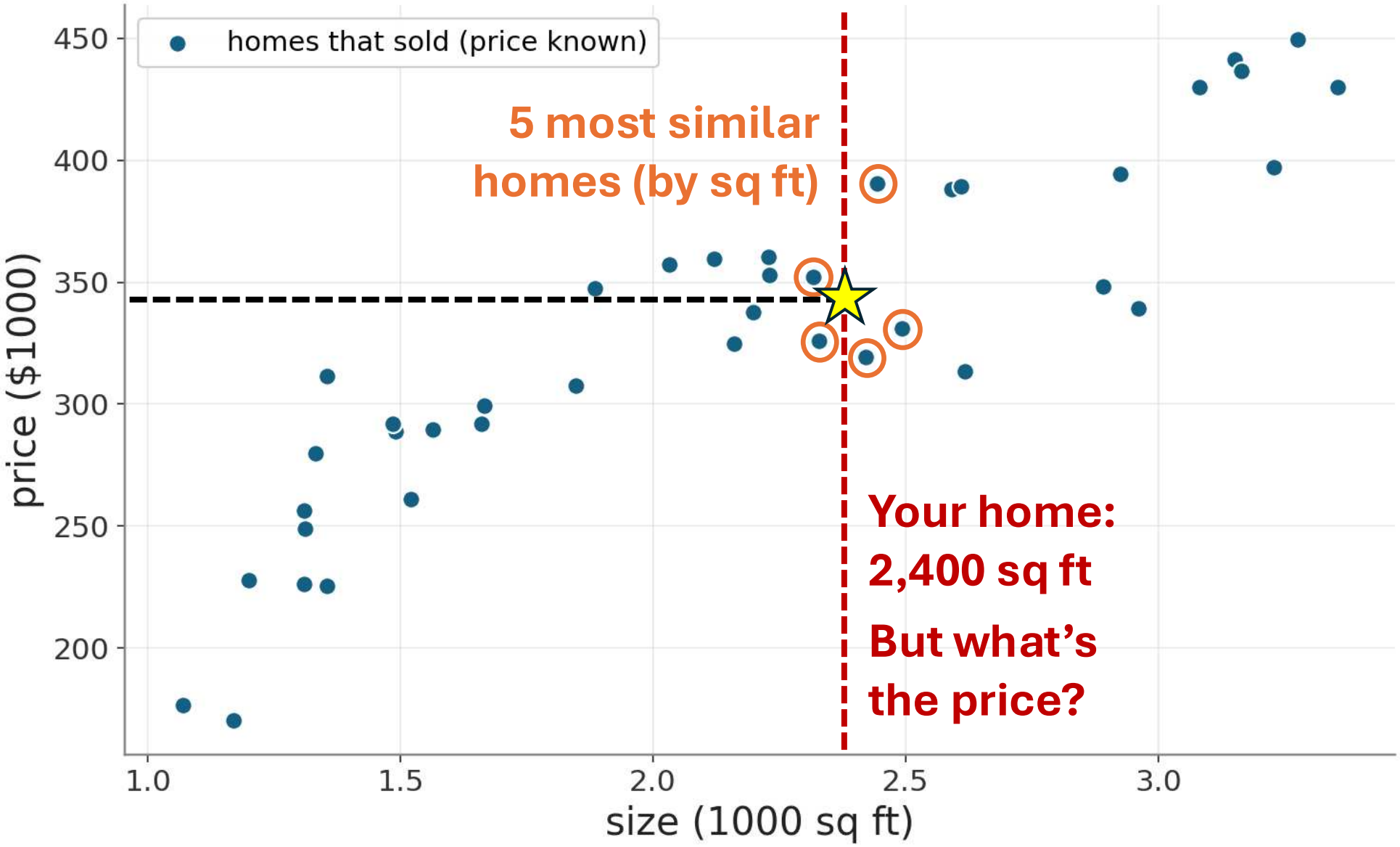
To estimate the sale price:



- You probably wouldn't try to fit an equation in your head
- Rather, you might look up the prices of recently sold homes that are ***similar***
  - **Based on location, square footage, number of bedrooms, age, …**
- Then make an estimate by ***averaging their sale prices***

Zillow

An example: Housing prices



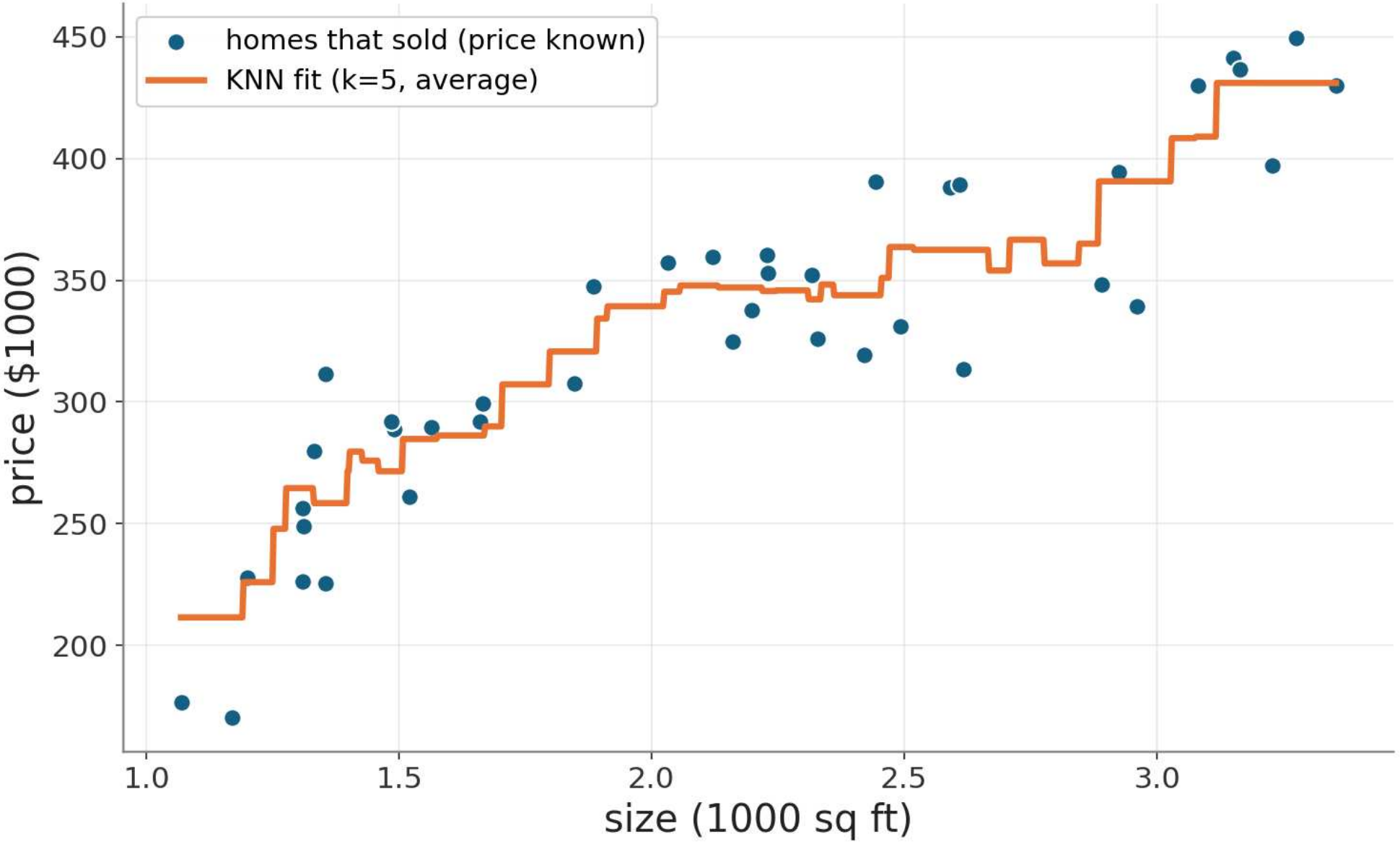
Size (1000 sq ft)	Sale price ($1000)
2.35	355
2.38	330
2.44	320
2.49	335
2.47	390

Average

$346k

This is *k*-Nearest Neighbors (*k* = 5)!

An example: Housing prices



We could repeat this procedure over a uniform grid of *x*-values (different sq ft) and obtain this function

Note this was done ***without any fitting!***

Now let's walk through it step by step…

The KNN algorithm for regression

Given **training data** (x_i, y_i) and a **query** x': ⟶ Where we'd like to make a prediction

1. Compute the **distance** $D(x', x_i)$ between the query and every datapoint

The KNN algorithm for regression

Given **training data** (x_i, y_i) and a **query** x'**:**

1. Compute the **distance** $D(x', x_i)$ between the query and every datapoint

2. Sort by the datapoints by their distance to the query: from **lowest** to **highest**

most similar **least similar**

The KNN algorithm for regression

Given **training data** $(\boldsymbol{x_i}, \boldsymbol{y_i})$ and a **query** $\boldsymbol{x'}$**:**

1. Compute the **distance** $\boldsymbol{D}(x', x_i)$ between the query and every datapoint
2. Sort by the datapoints by their distance to the query: from lowest to highest
3. Choose **how many neighbors (**$\boldsymbol{k}$**)** to consider
4. Take the first k nearest datapoints from the sorted list → **Neighbors** $\boldsymbol{N_k}(\boldsymbol{x'})$

The KNN algorithm for regression

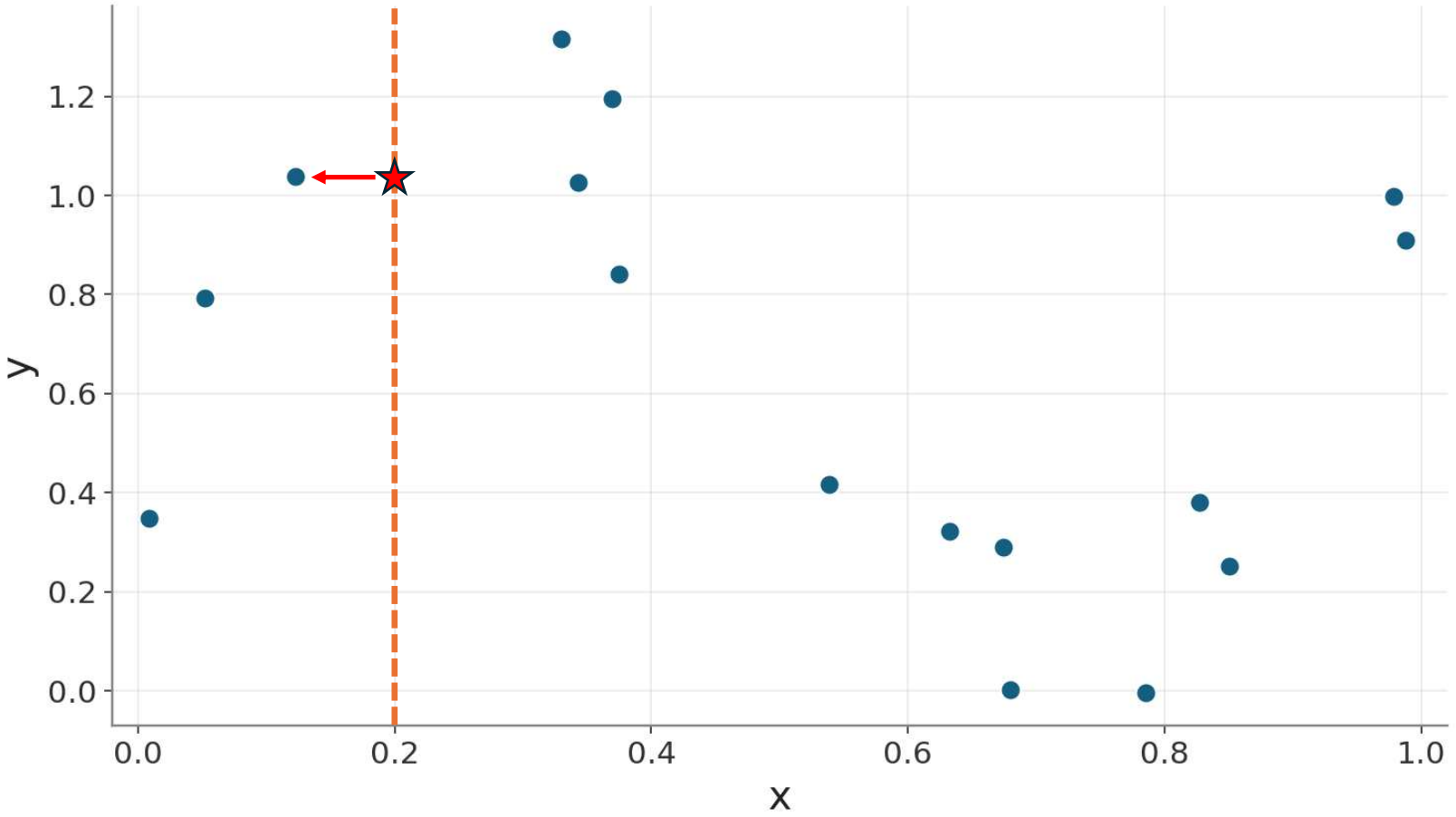
Given **training data** (x_i, y_i) and a **query** x'**:**

1. Compute the **distance** $D(x', x_i)$ between the query and every datapoint
2. Sort by the datapoints by their distance to the query: from lowest to highest
3. Choose **how many neighbors (**k**)** to consider
4. Take the first k nearest datapoints from the sorted list → **Neighbors** $N_k(x')$
5. Predict $\hat{y}$ by **averaging** over those nearest datapoints (y_i)

$$\hat{y} = \frac{1}{k}\sum_i y_i \text{ for } i \in N_k(x')$$

That's it! Repeat this process for each query. No fitting needed.

A simple walkthrough: $k = 1$



These points represent our known training data

Suppose our query is $x' = 0.2$ and we'd like to predict $\hat{y}(x')$ using $k = 1$ (one neighbor)

x_i	y_i
0.13	1.03
0.37	1.28
0.07	0.79
…	…

$k = 1$

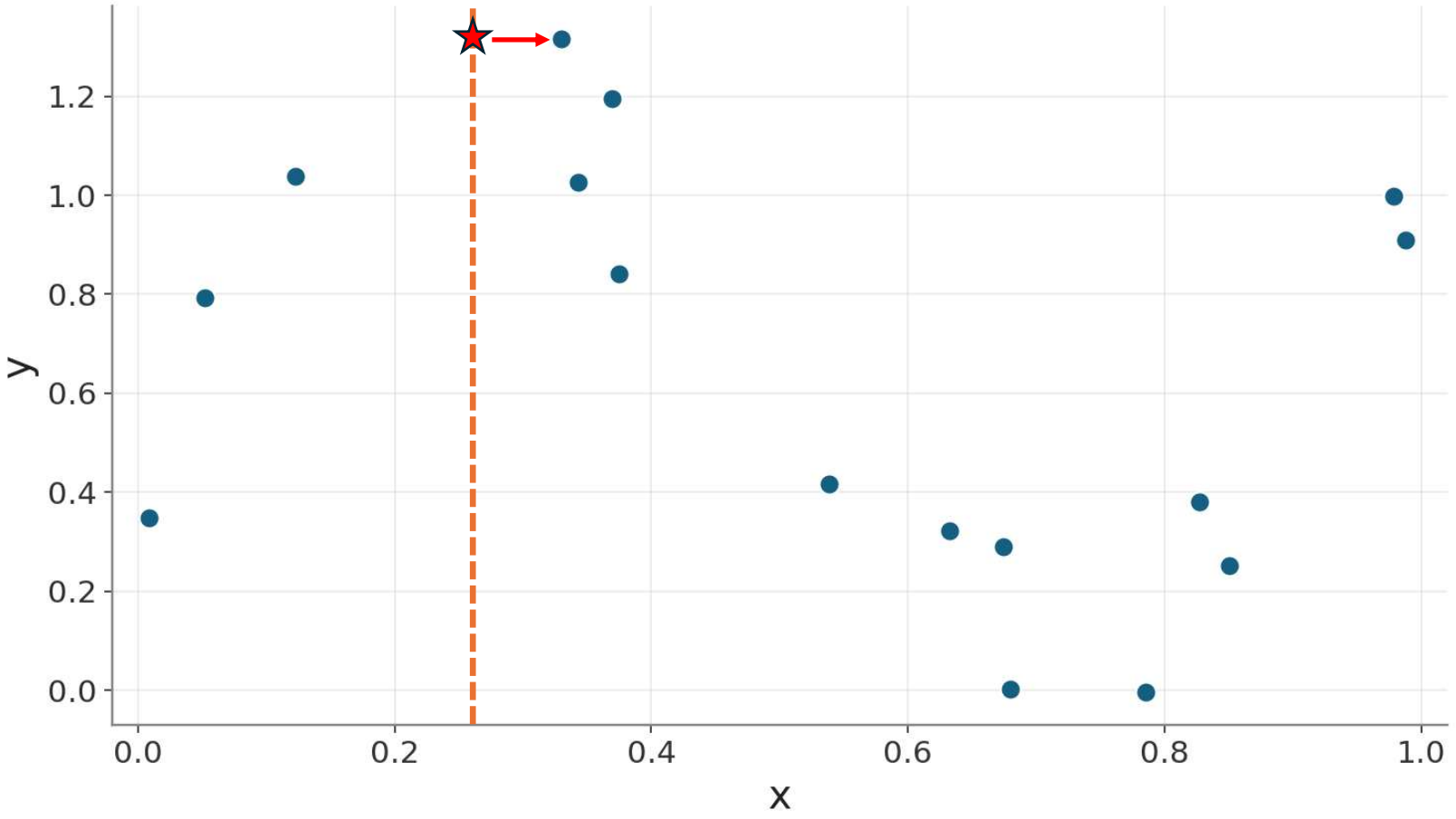
Sorted

$$D = |x_i - x'|$$

In the case of $k = 1$, we just look for the closest point and use its value without averaging:

$$\hat{y} = y_{\text{nearest}}$$

A simple walkthrough: $k = 1$



These points represent our known training data

Suppose our **new query** is $x' = 0.25$ and we'd like to predict $\hat{y}(x')$

x_i	y_i
0.13	1.03
0.37	1.28
0.07	0.79
…	…

$k = 1$

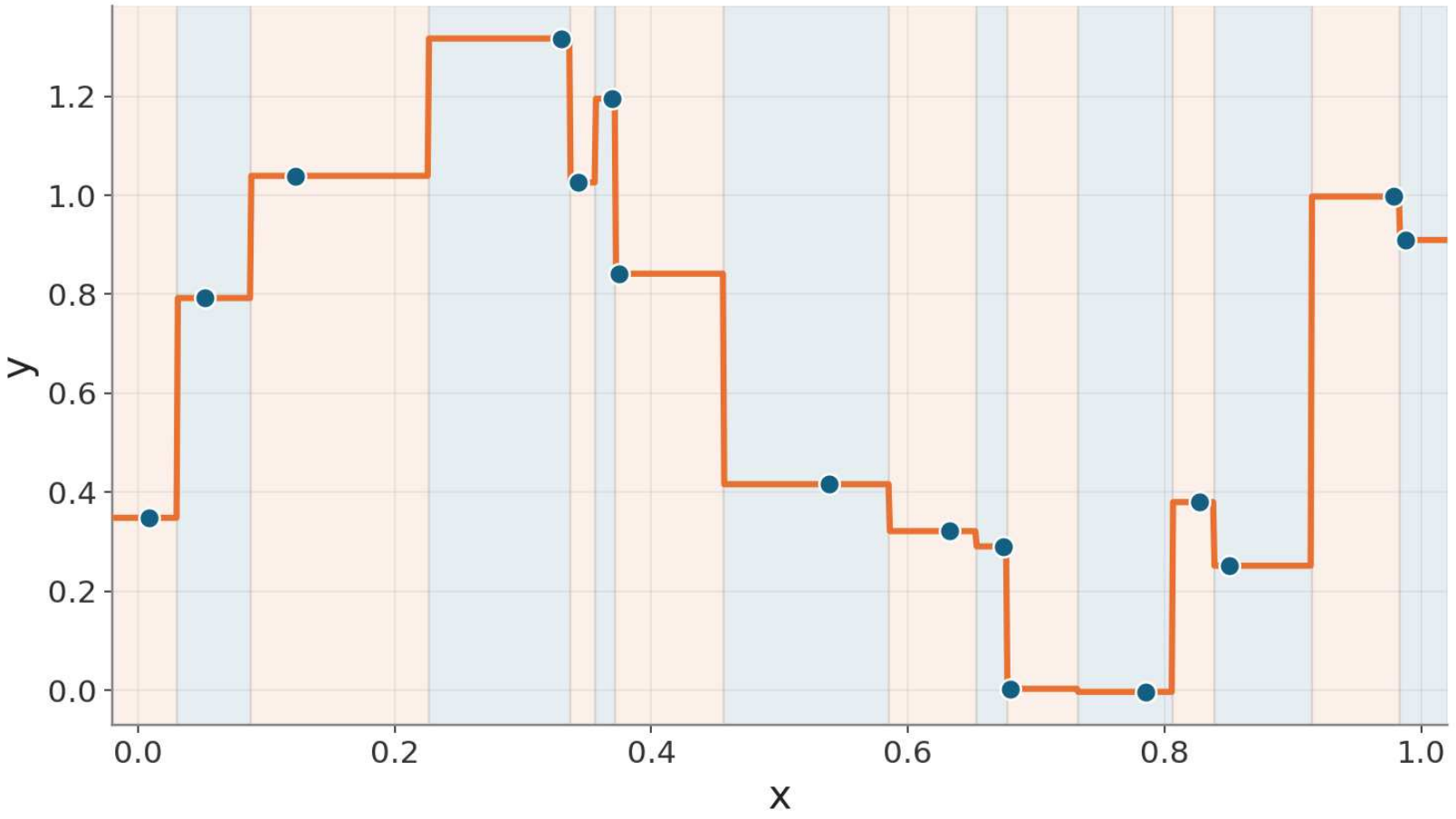
Sorted

$$D = |x_i - x'|$$

In the case of $k = 1$, we just look for the closest point and use its value without averaging:

$$\hat{y} = y_{\text{nearest}}$$

A simple walkthrough: $k = 1$



These points represent our known training data

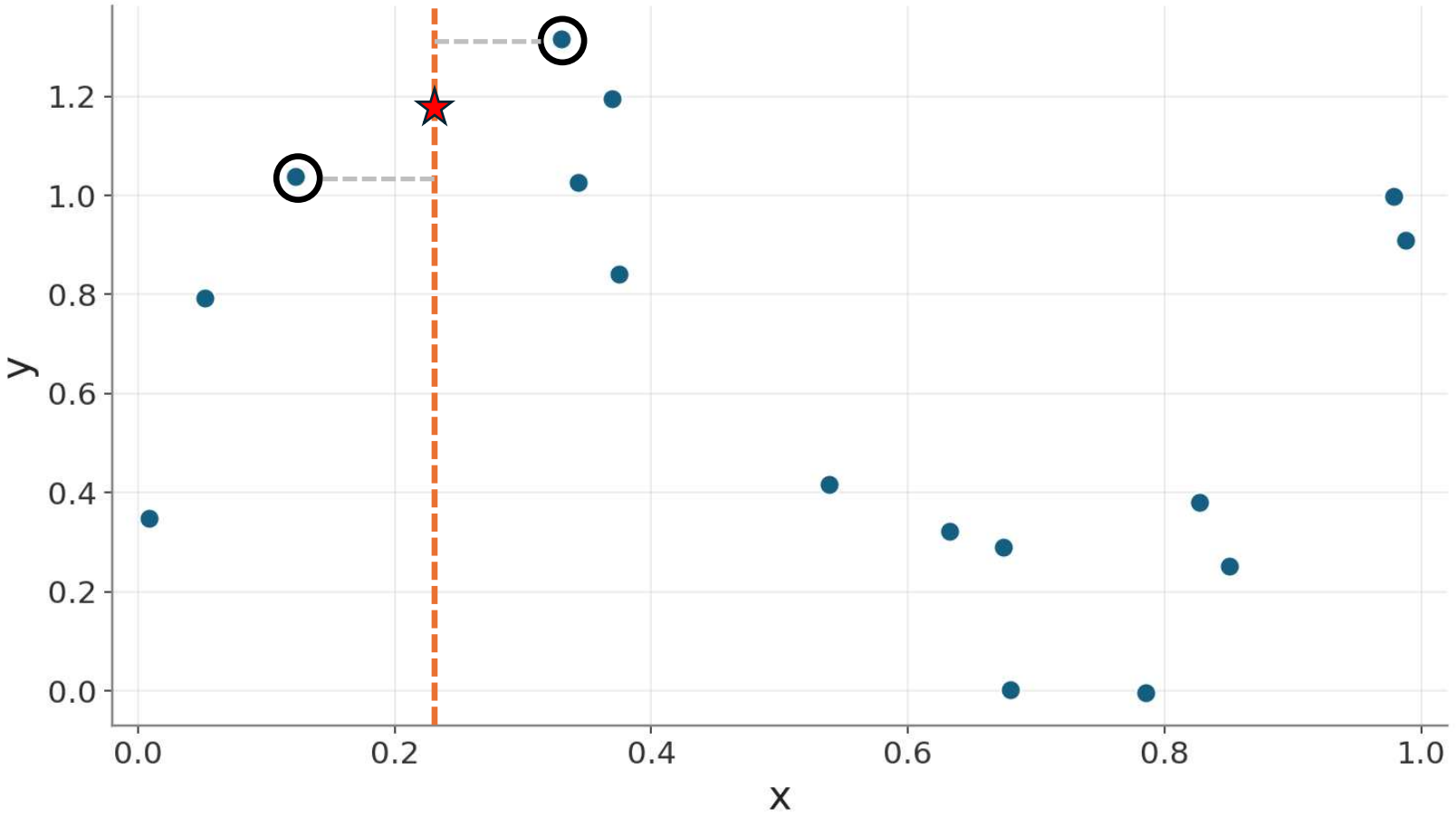
Repeat this for many queries at different x' values to generate a complete curve

Notice this is just many "flat" regions (constant $\hat{y}$) with abrupt jumps between them caused by a change in the nearest neighbor

When $k = 1$, training error will always be zero!

We are forcing the curve to run through every point. → ***Likely to overfit***

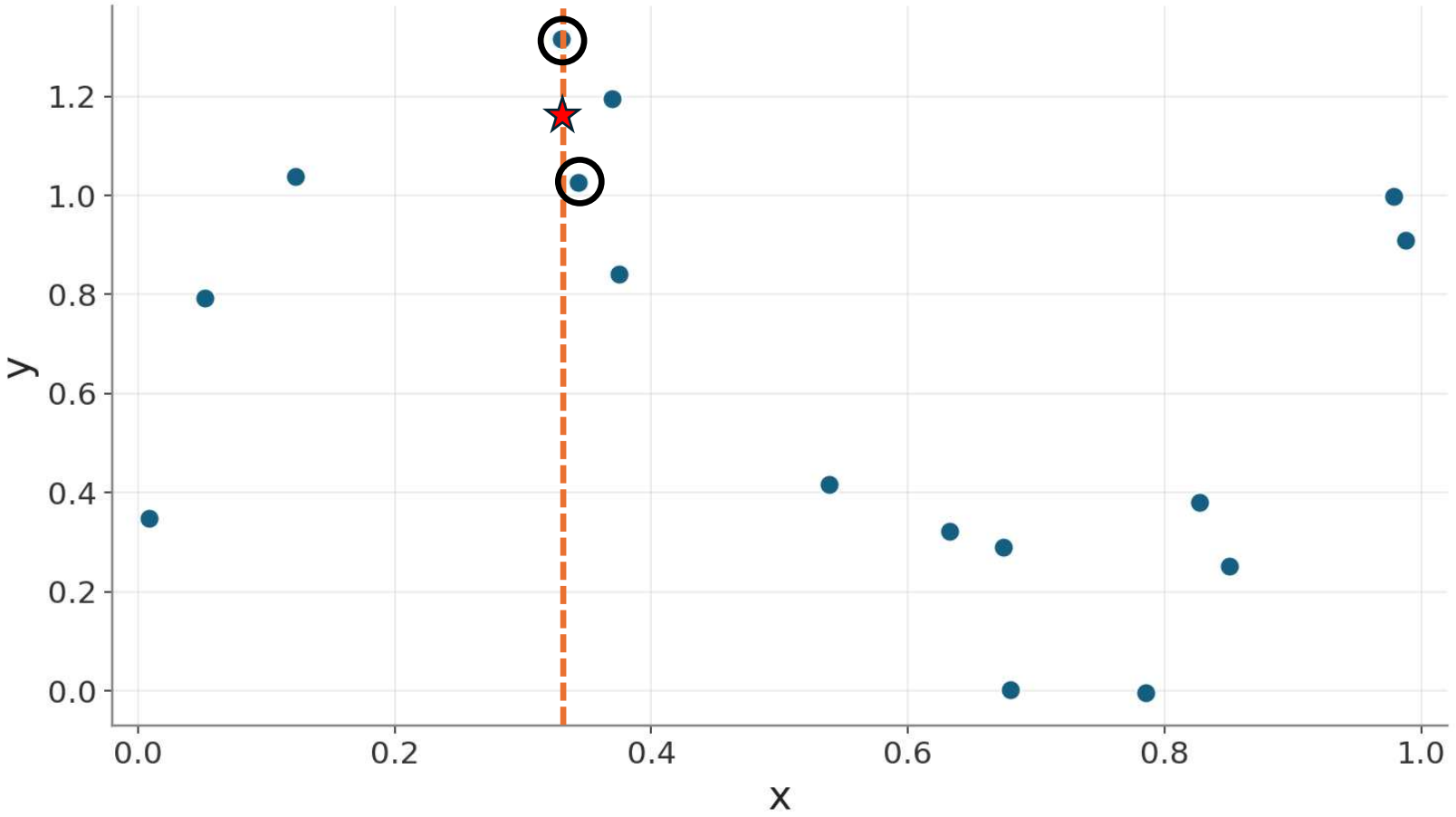
Let's increase to $k = 2$



When $k = 2$, the ouput ($\hat{y}$) is decided by the average between the two closest datapoints

$$\hat{y} = \frac{1}{2}(y_{\text{nearest}} + y_{\text{next-nearest}})$$

Let's increase to $k = 2$



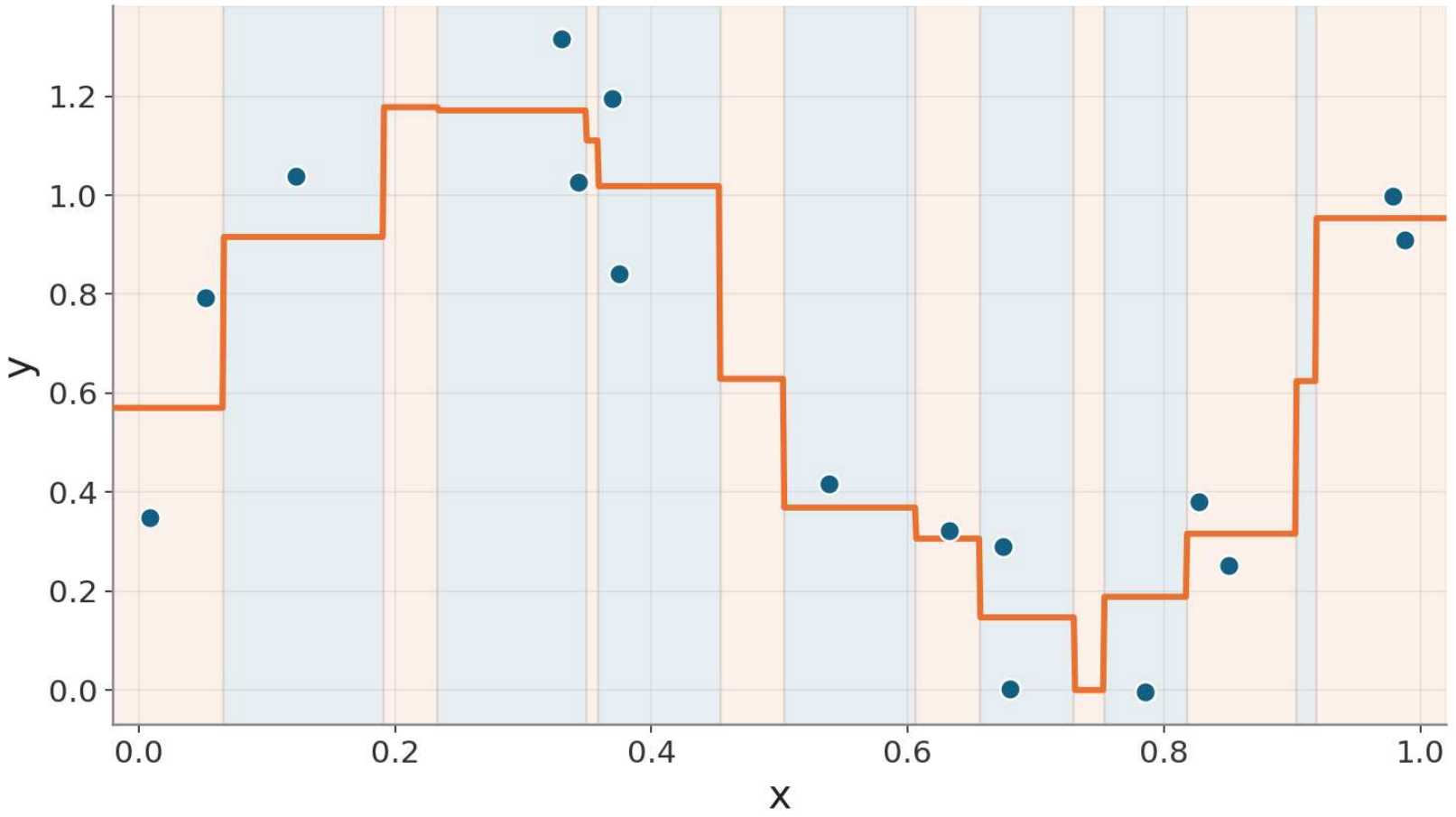
When $k = 2$, the ouput ($\hat{y}$) is decided by the average between the two closest datapoints

$$\hat{y} = \frac{1}{2}(y_{\text{nearest}} + y_{\text{next-nearest}})$$

It need not pass through the known datapoints!

→ **training error $\neq 0$**

Let's increase to $k = 2$



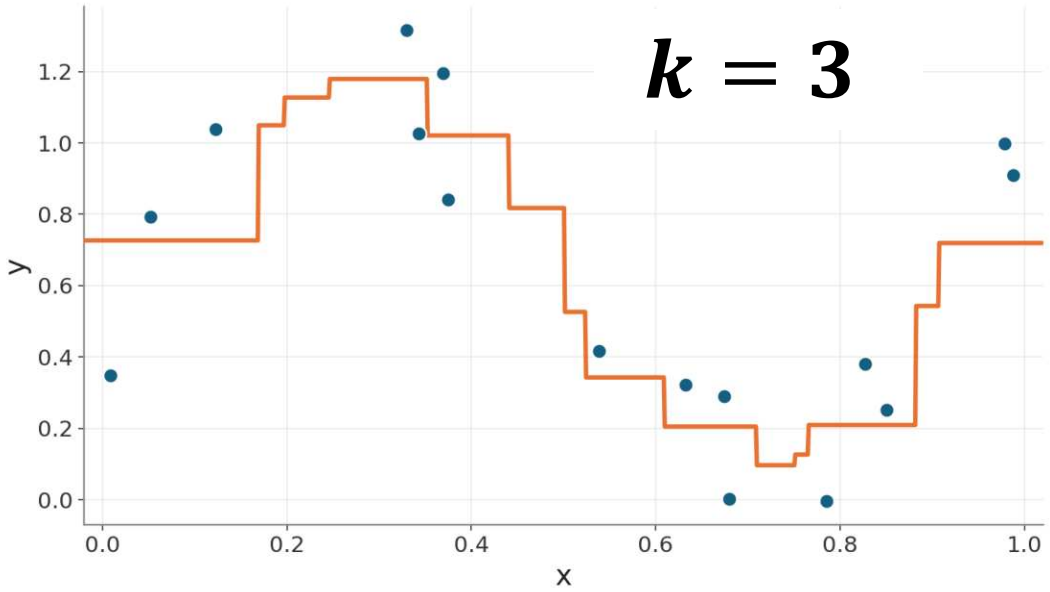
When $k = 2$, the ouput ($\hat{y}$) is decided by the average between the two closest datapoints

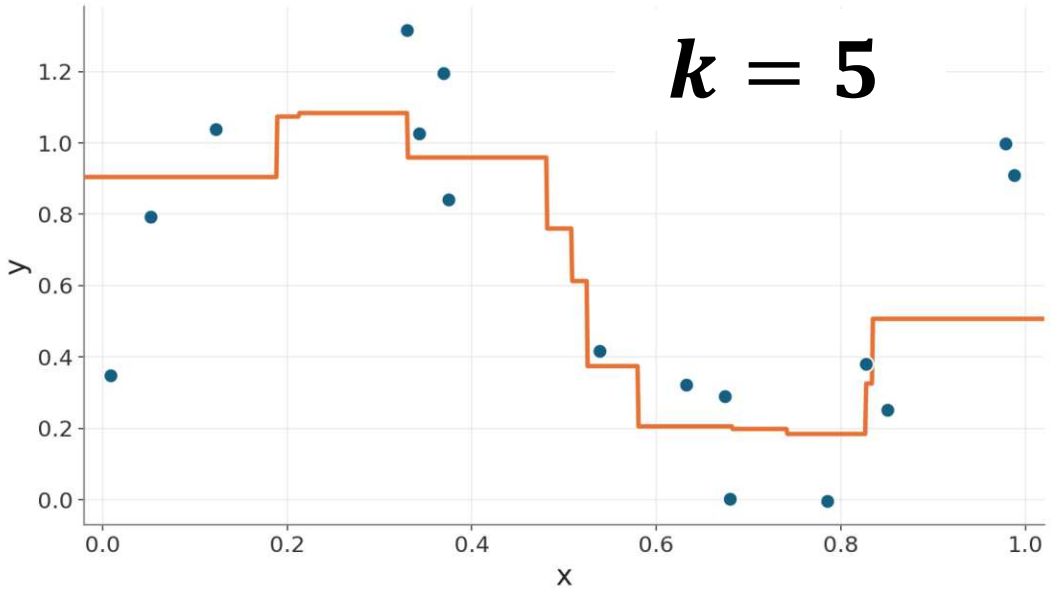
$$\hat{y} = \frac{1}{2}(y_{\text{nearest}} + y_{\text{next-nearest}})$$

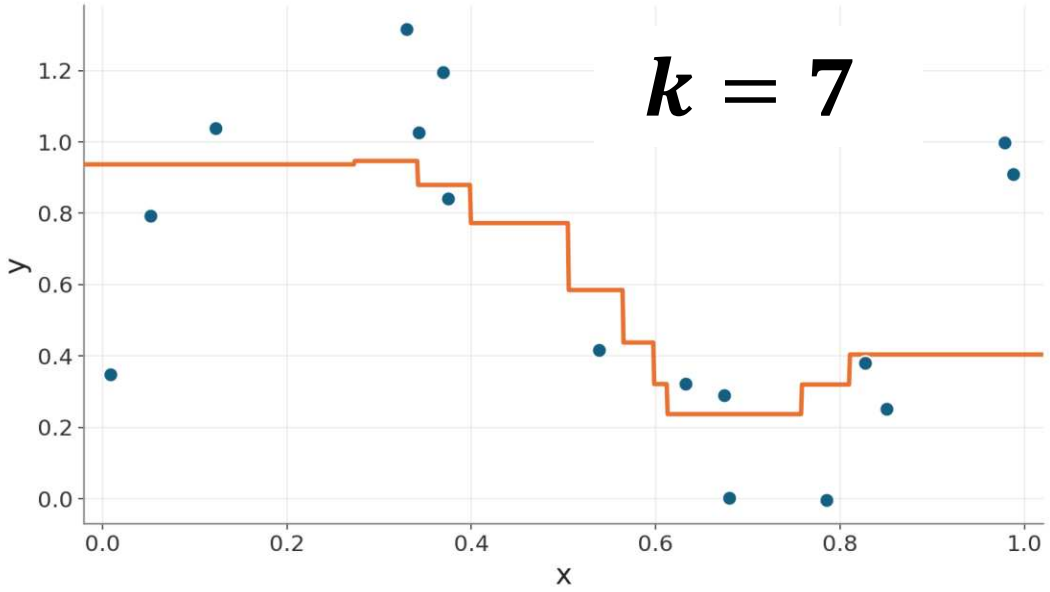
It need not pass through the known datapoints!
→ **training error ≠ 0**

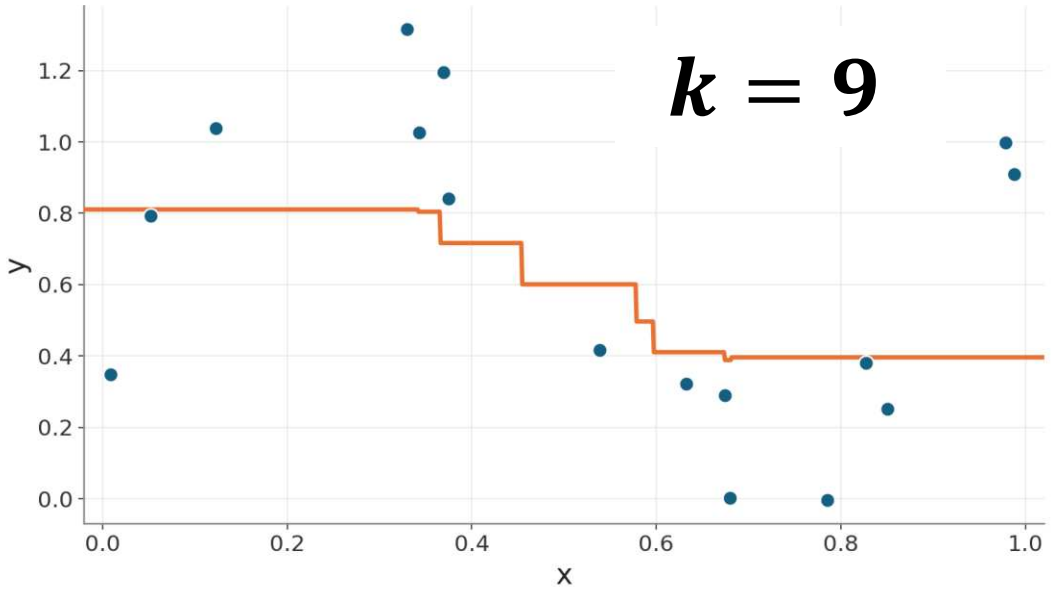
We see that with **larger $k = 2$**, there is **less overfitting.**
But the shape of the curve is still flat plateaus with jumps in between.

The influence of larger k on the fit

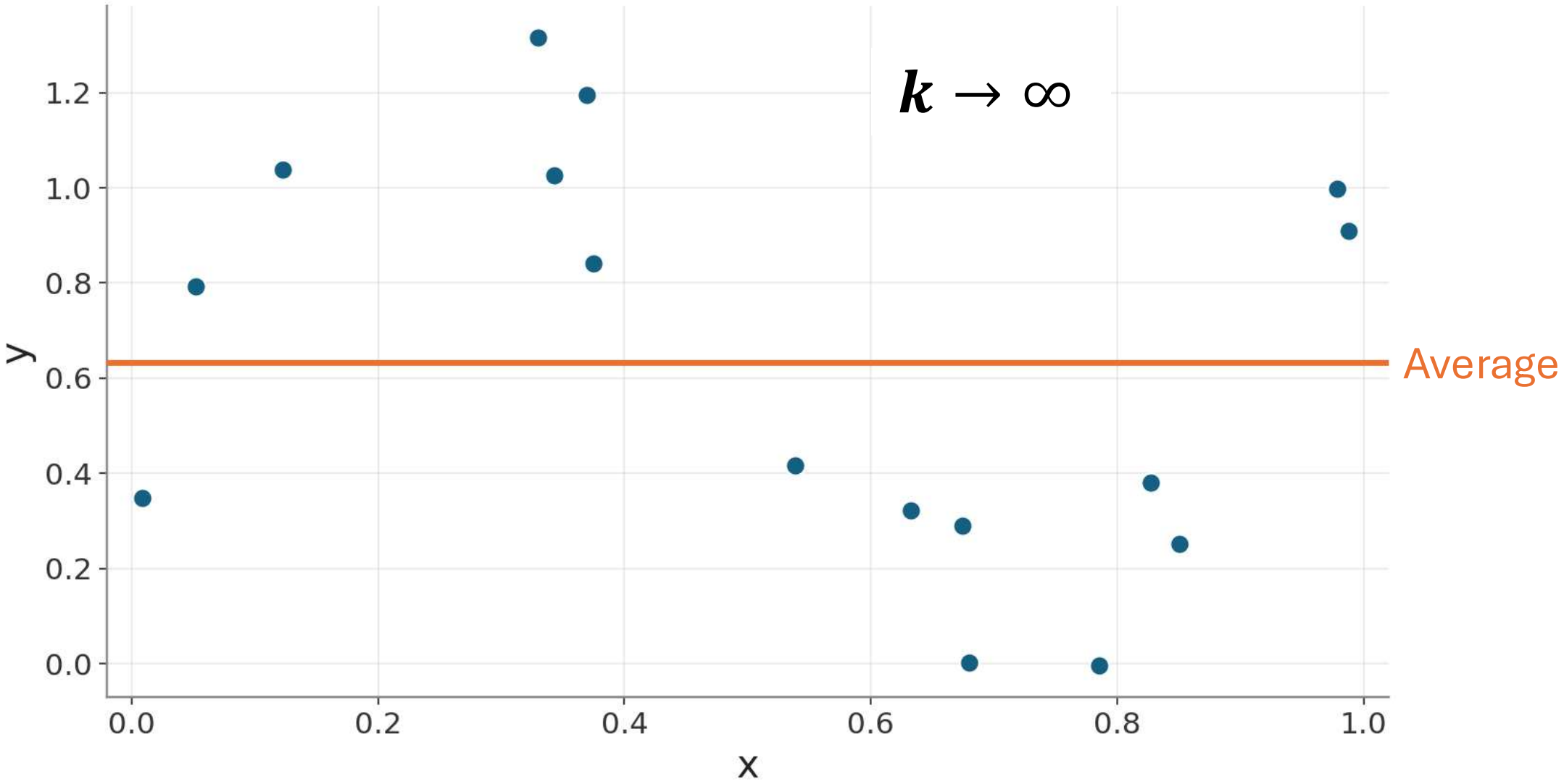








Eventually…$k \rightarrow$ entire dataset



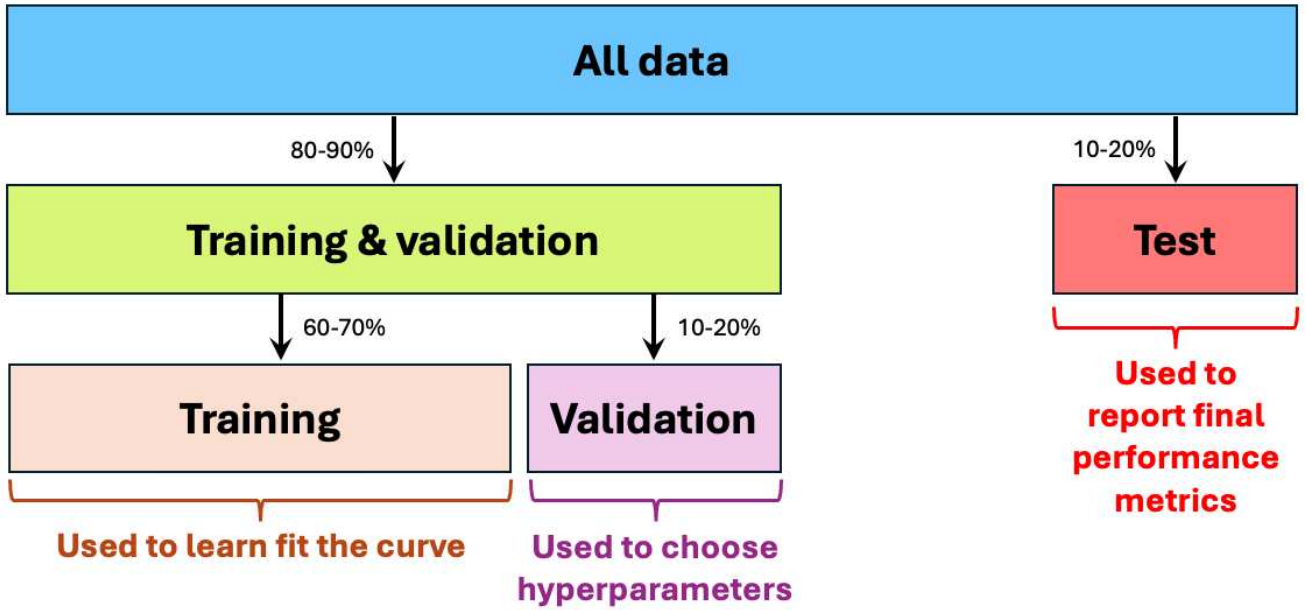
Choosing "the right" k for your dataset

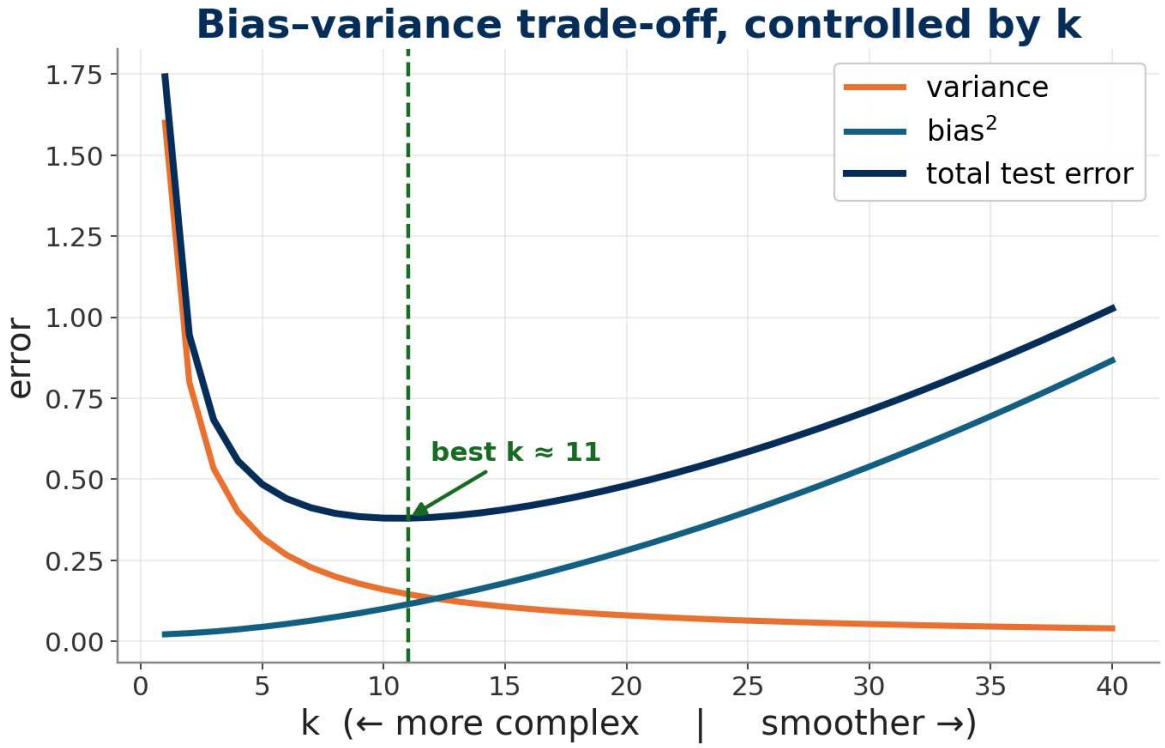
Small k
Very flexible model, fits the noise
Low bias, high variance

k is a **hyperparameter** to be tuned!

Large k
More rigid model, "flatter" curves
High bias, low variance

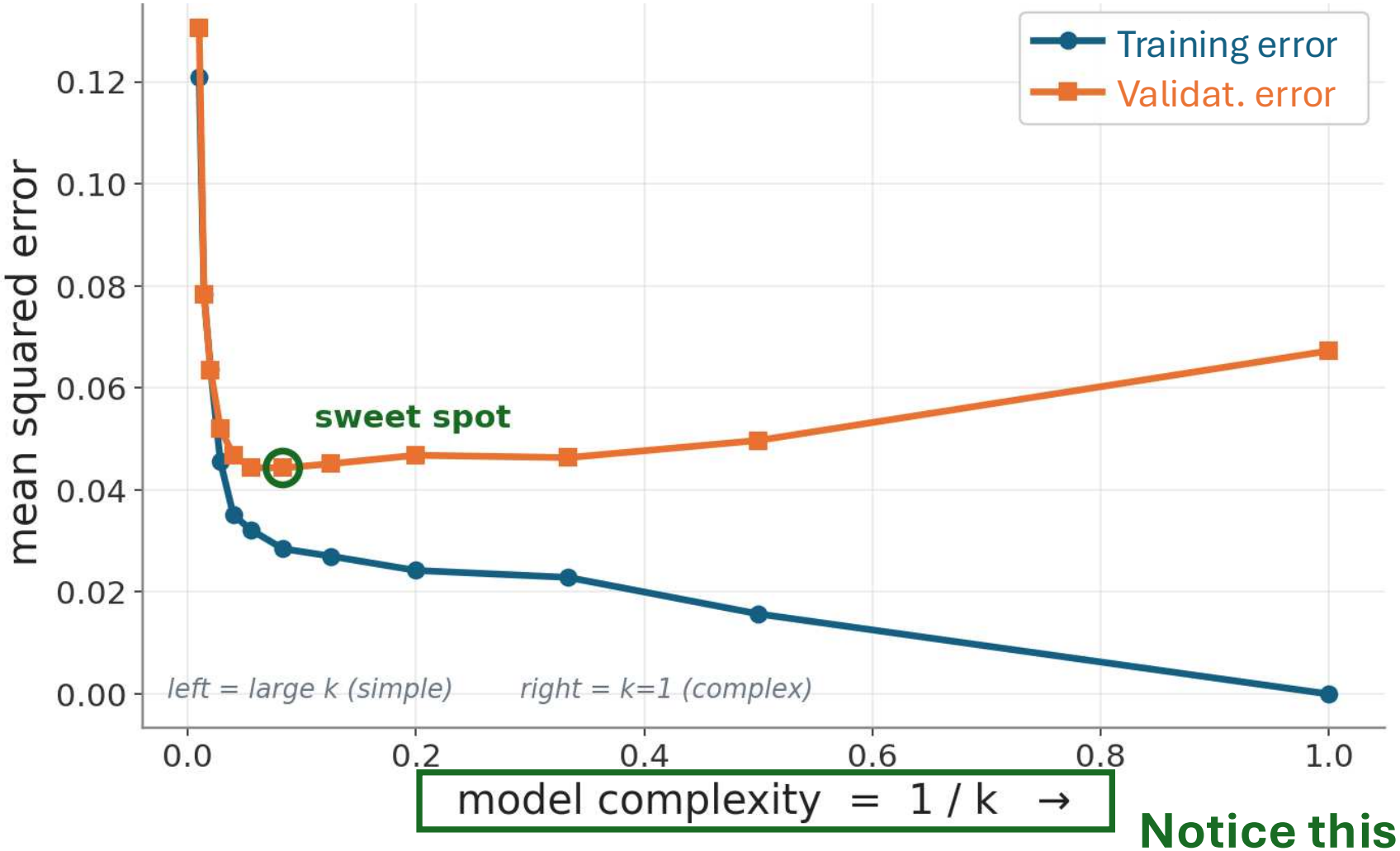
Remember we choose hyperparameters by **minimizing error on the validation set**





Model complexity in KNN regression

Choosing *k* through training and validation



In KNN, we can think of **$1/k$ as the model complexity**

Lower k always reduces training error – but that's *not* the goal! It overfits, increasing variance and hurting performance on new (test) data.

Notice this is the inverse of *k*

Some general rules on choosing k

Situation	Typical choice of k	Reasoning
Lots of training data	Higher k	You can average over more neighbors while still keeping them local (nearby).
Noisier data	Higher k	Averaging over more neighbors smooths out noise and reduces overfitting.
Complex functions	Lower k	Small neighborhoods can adapt to rapid local changes.
High-dimensional input	(Usually) higher k	Neighborhoods become sparse, so slightly larger k can stabilize predictions.

Some general rules on choosing k



Notice that large k works well when we have lots of data, but over-smooths when we don't have much data

A decent rule of thumb – start with:

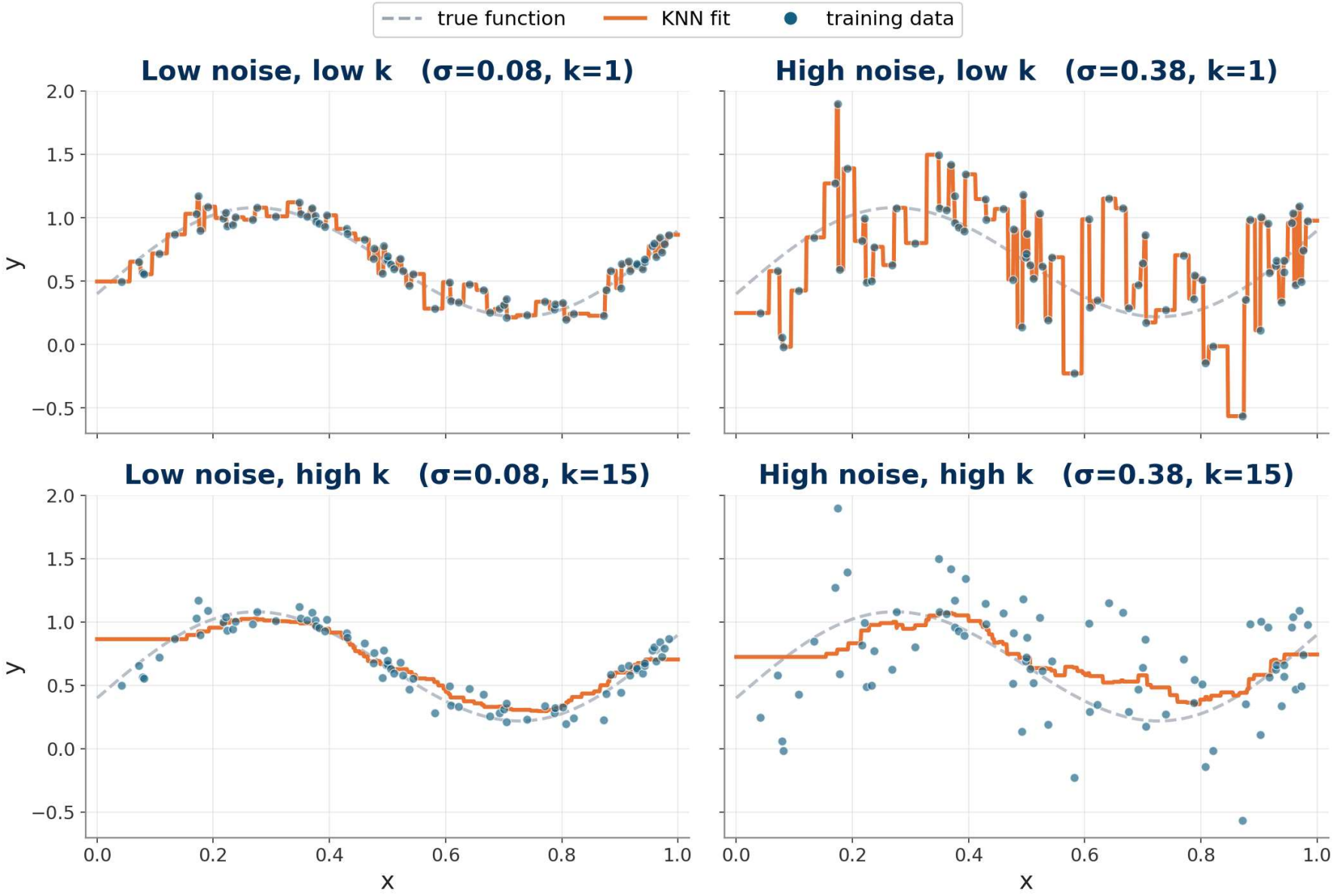
$$k \approx \sqrt{N}$$

Number of samples

Some general rules on choosing k

Situation	Typical choice of k	Reasoning
Lots of training data	Higher k	You can average over more neighbors while still keeping them local.
Noisier data	Higher k	Averaging over more neighbors smooths out noise and reduces overfitting.
Complex functions	Lower k	Small neighborhoods can adapt to rapid local changes.
High-dimensional input	(Usually) higher k	Neighborhoods become sparse, so slightly larger k can stabilize predictions.

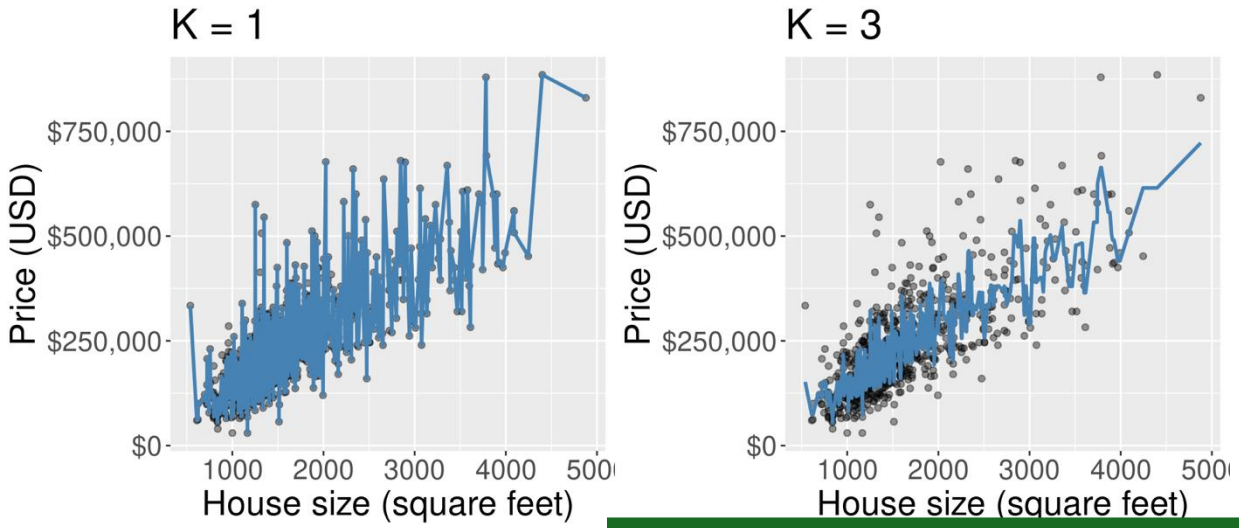
Some general rules on choosing k



Small k is prone to overfitting on the noise…

But if there is not much noise, small k works quite well

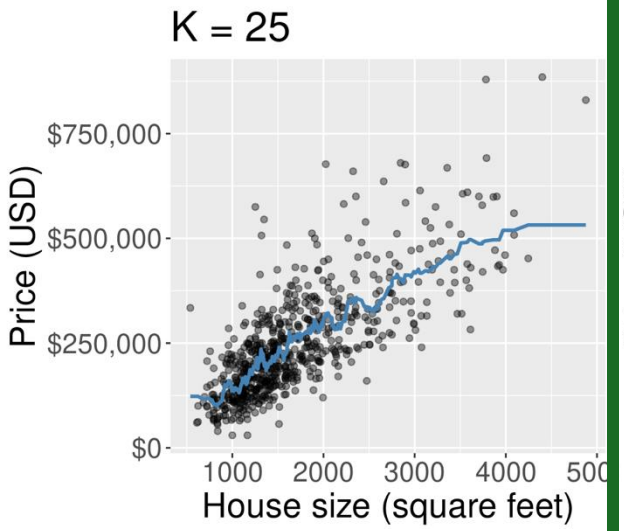
Some general rules on choosing k

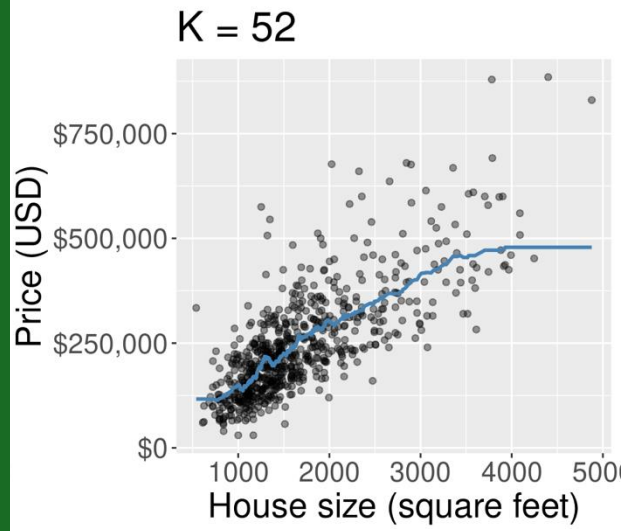


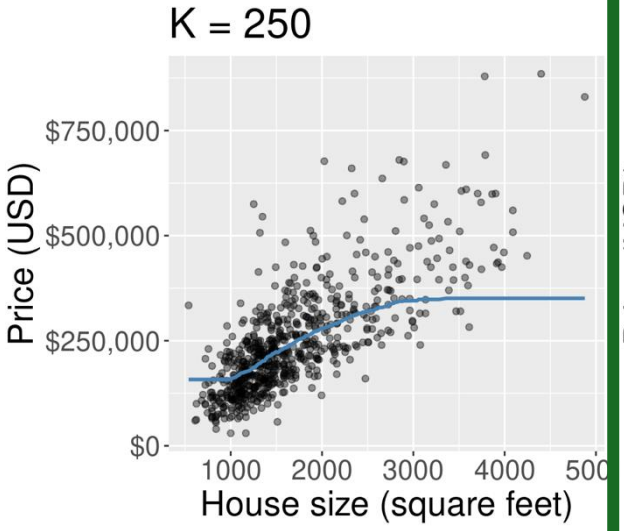
Here's a real example with lots of noisy data: **Housing prices**

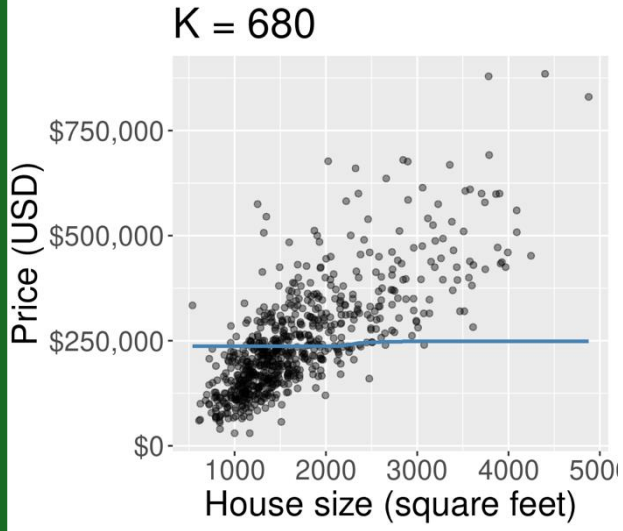


Large k (on the order of 50-100) is needed to get a reasonable trend







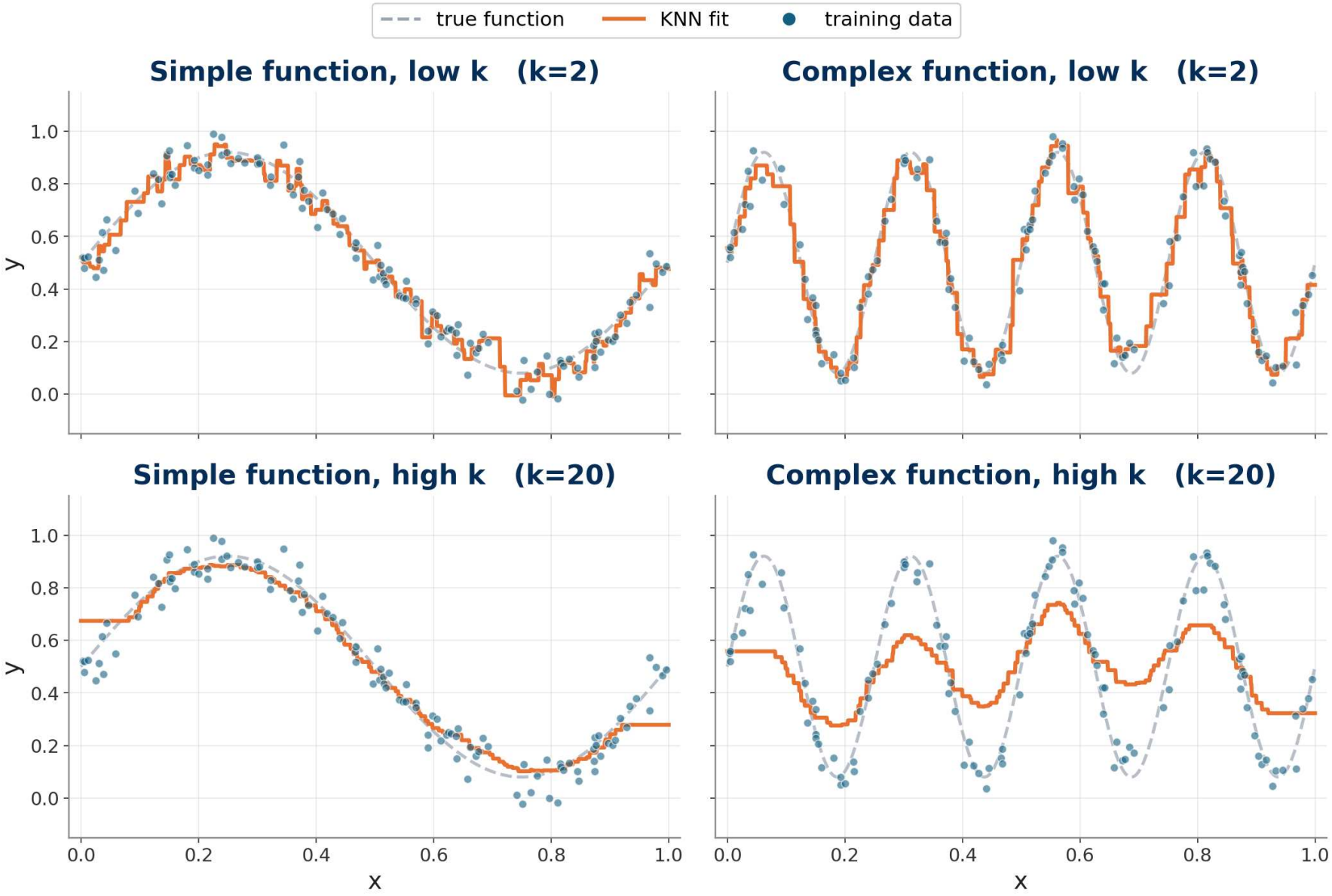


Source: *Timbers, Campbell & Lee, "Data Science: A First Introduction"*

Some general rules on choosing k

Situation	Typical choice of k	Reasoning
Lots of training data	Higher k	You can average over more neighbors while still keeping them local.
Noisier data	Higher k	Averaging over more neighbors smooths out noise and reduces overfitting.
Complex functions	Lower k	Small neighborhoods can adapt to rapid local changes.
High-dimensional input	(Usually) higher k	Neighborhoods become sparse, so slightly larger k can stabilize predictions.

Some general rules on choosing k



Notice that when the true function fluctuates wildly, high k tends to over-smooth and underfit…

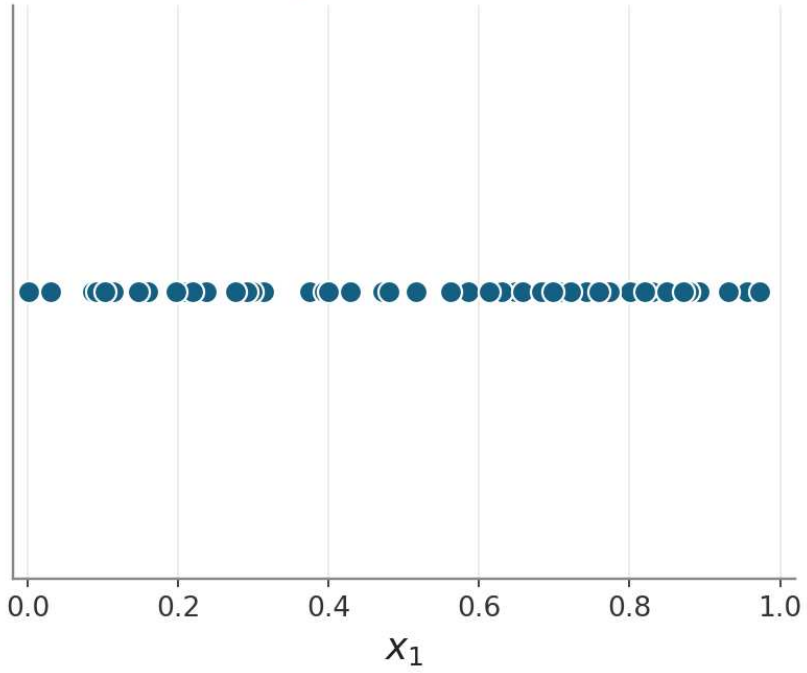
But when the function is simple, high k works just fine and is less noisy than low k

Some general rules on choosing k

Situation	Typical choice of k	Reasoning
Lots of training data	Higher k	You can average over more neighbors while still keeping them local.
Noisier data	Higher k	Averaging over more neighbors smooths out noise and reduces overfitting.
Complex functions	Lower k	Small neighborhoods can adapt to rapid local changes.
High-dimensional input	(Usually) higher k	Neighborhoods become sparse (curse of dim), larger k can stabilize predictions.

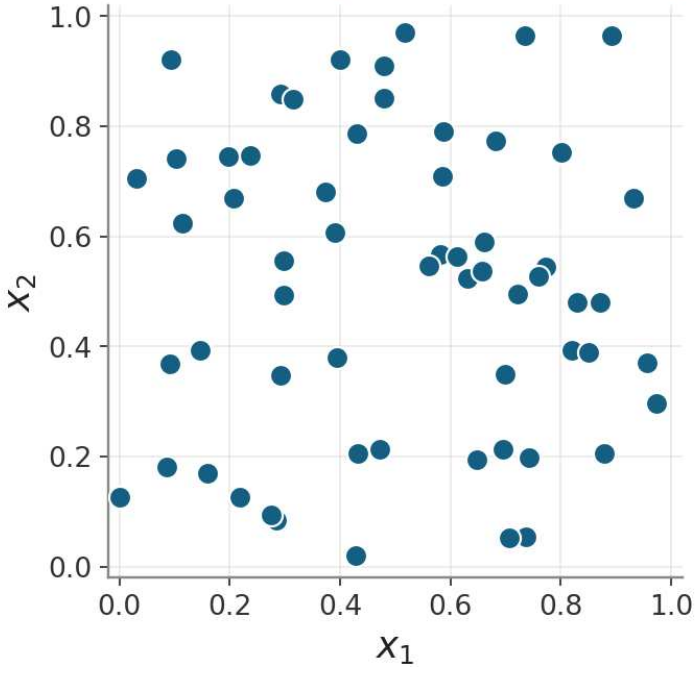
Some general rules on choosing k

1D: 60 points on a line



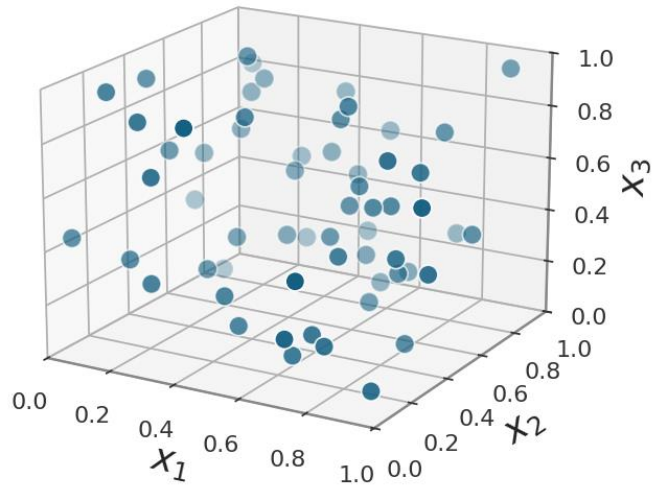
Avg. neighbor distance ~ 0.01

2D: same 60 points in a square



Avg. neighbor distance ~ 0.07

3D: same 60 points in a cube



Avg. neighbor distance ~ 0.16

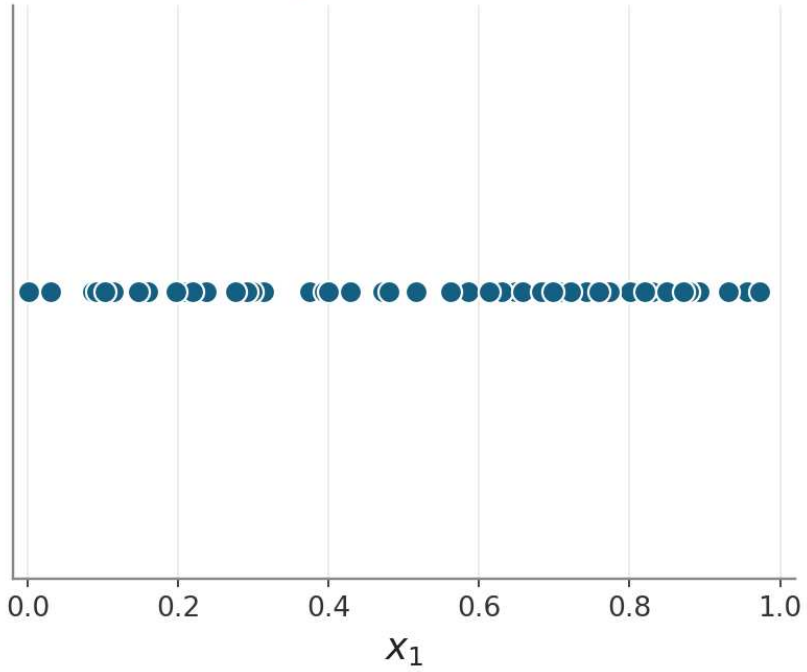
Recall **the curse of dimensionality –** data gets **sparse** as the dimensions increase. This means "nearest" neighbors are not actually that close, and averaging across many approximately equidistant neighbors can help prevent overfitting

Questions?

Next: how do we measure **distance**? Not as simple as it seems…

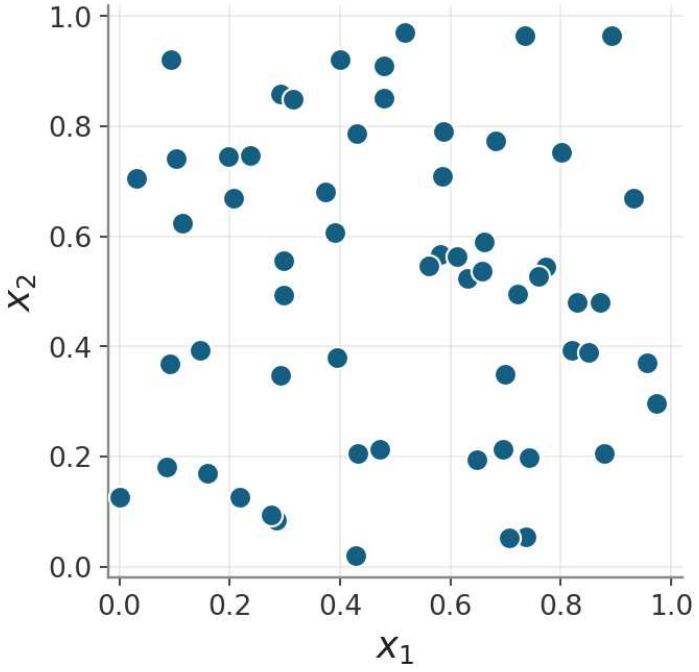
How do we define distance?

1D: 60 points on a line



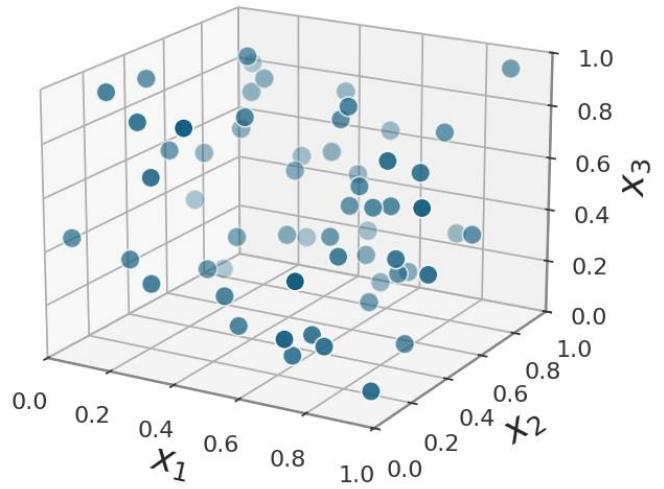
Avg. neighbor distance ~ 0.01

2D: same 60 points in a square



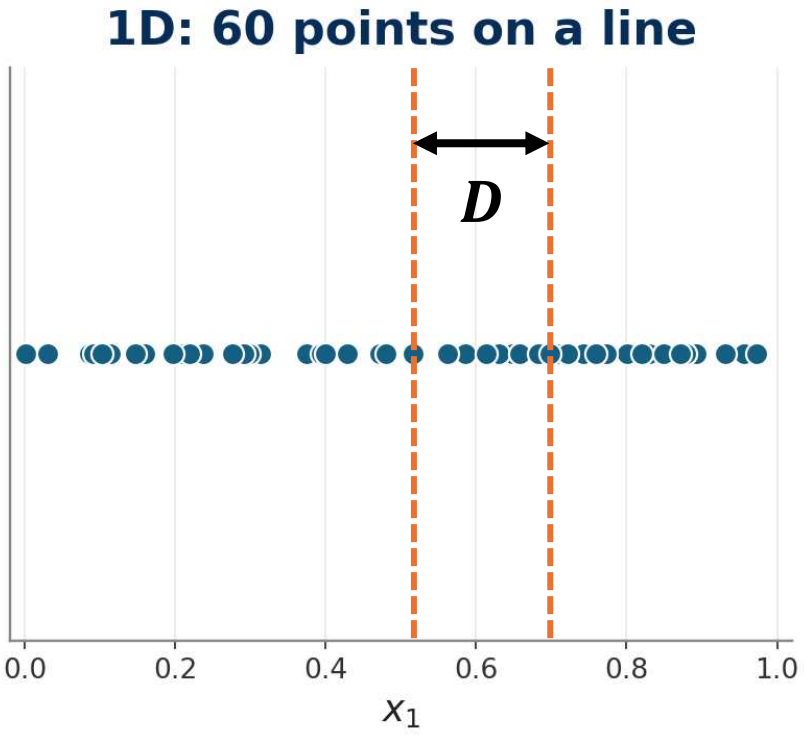
Avg. neighbor distance ~ 0.07

3D: same 60 points in a cube



Avg. neighbor distance ~ 0.16

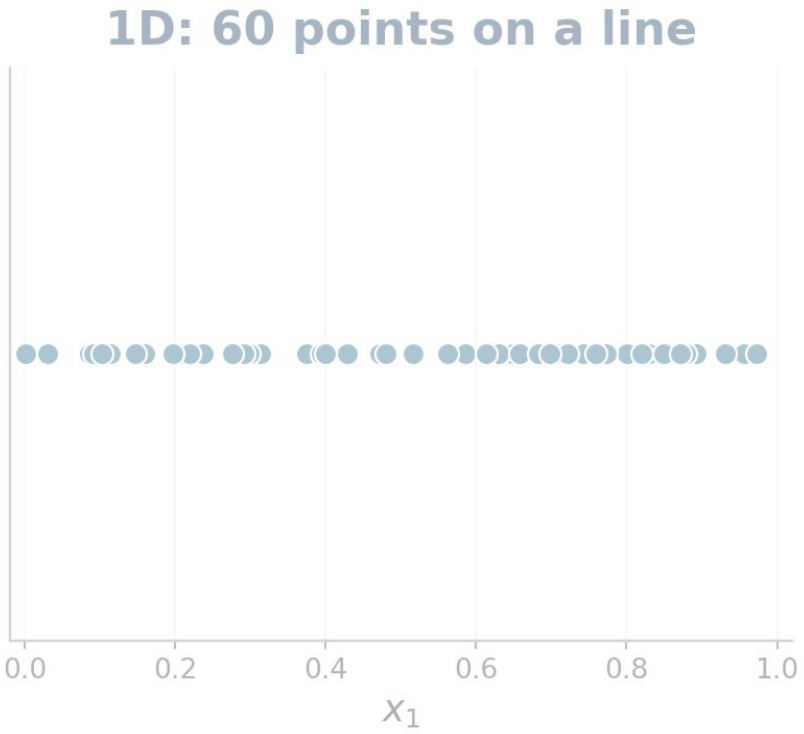
How do we define distance?

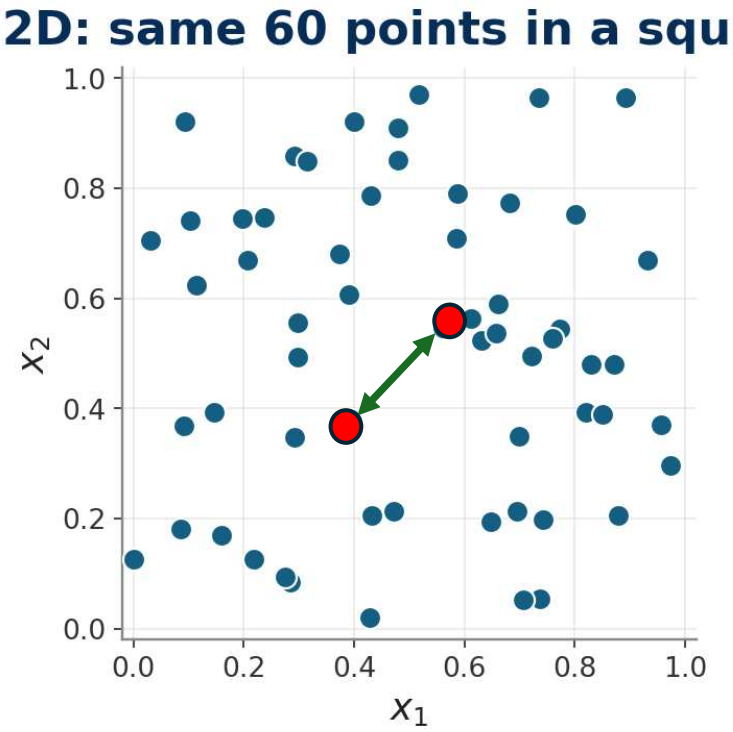


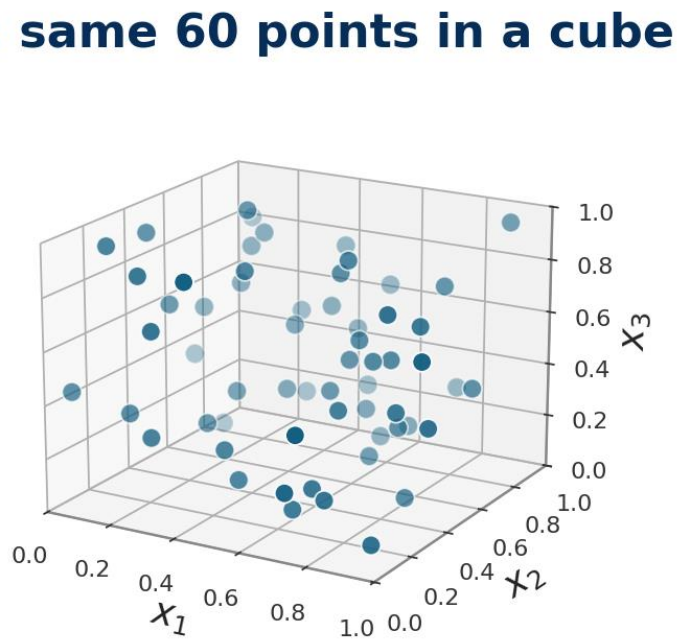
In one dimension, distance is simple:

$$D(x', x) = |x' - x|$$

How do we define distance?







How about in 2 and 3 dimensions?

You first instinct is probably to use **Euclidean distance:** $D(x', x) = \sqrt{\sum_d |x'_d - x_d|^2}$

...But this is just one type of distance – **there are others!**

A generalized definition of *distance*

Minkowski distance:

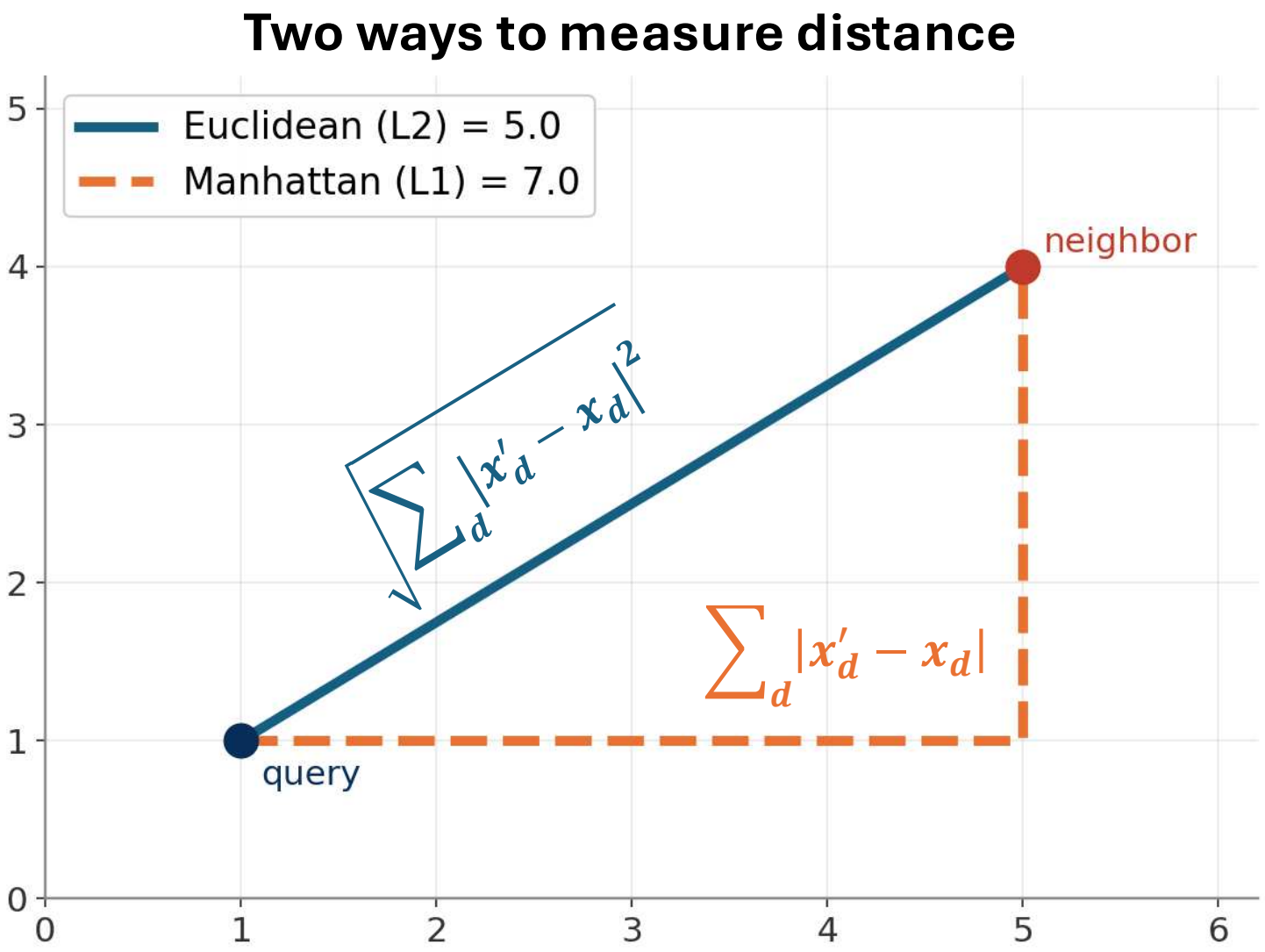
$$D(x', x) = \left(\sum_d |x'_d - x_d|^p \right)^{1/p}$$

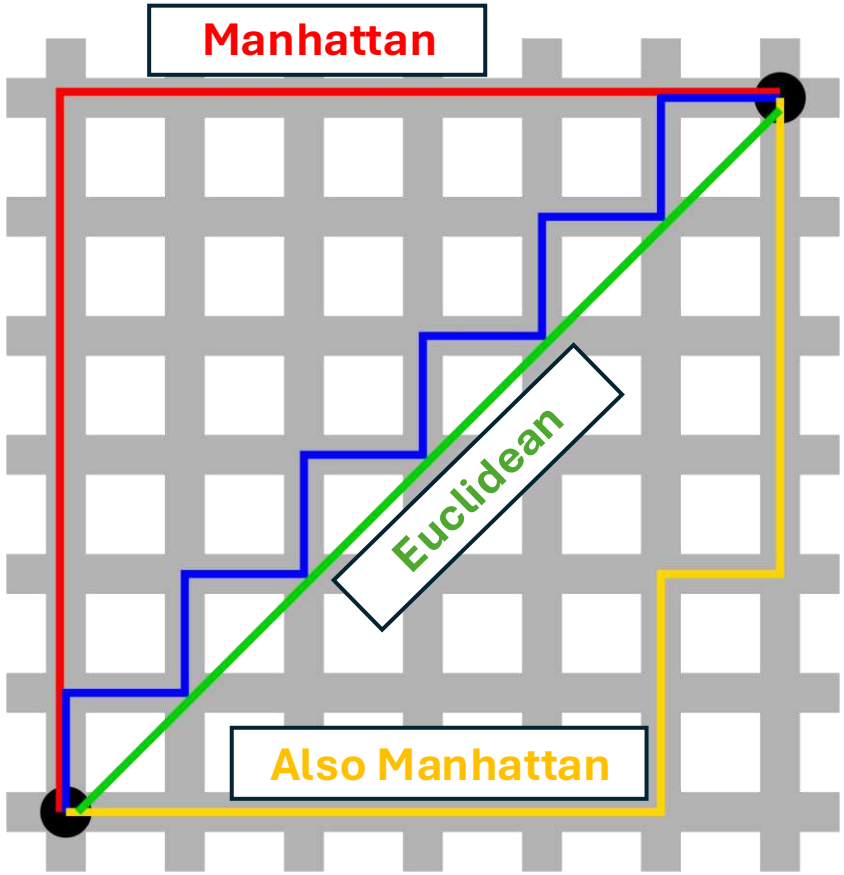




- $p = 1 \rightarrow$ **Manhattan distance (L_1 norm)** $\rightarrow D(x', x) = \sum_d |x'_d - x_d|$
- $p = 2 \rightarrow$ **Euclidean distance (L_2 norm)** $\rightarrow D(x', x) = \sqrt{\sum_d |x'_d - x_d|^2}$
- ...others are rarely used (at least not for ML)...
- $p = \infty \rightarrow$ **Chebyshev distance (L_∞ norm)** $\rightarrow D(x', x) = \max_d |x'_d - x_d|$

Manhattan *vs.* Euclidean distance

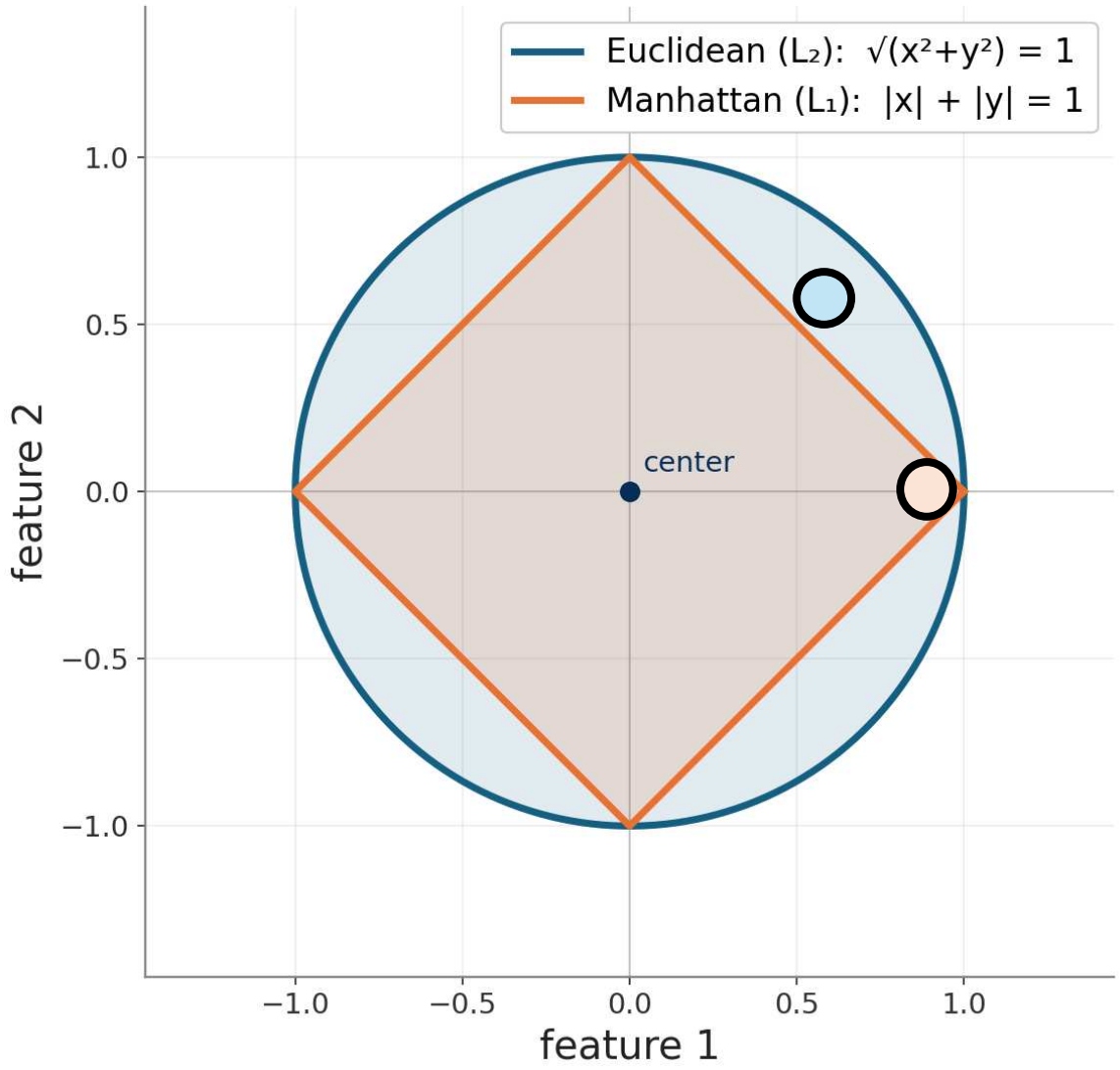




It's called Manhattan distance for a reason!

Your choice of distance affects the model!

“Distance = 1” from the center

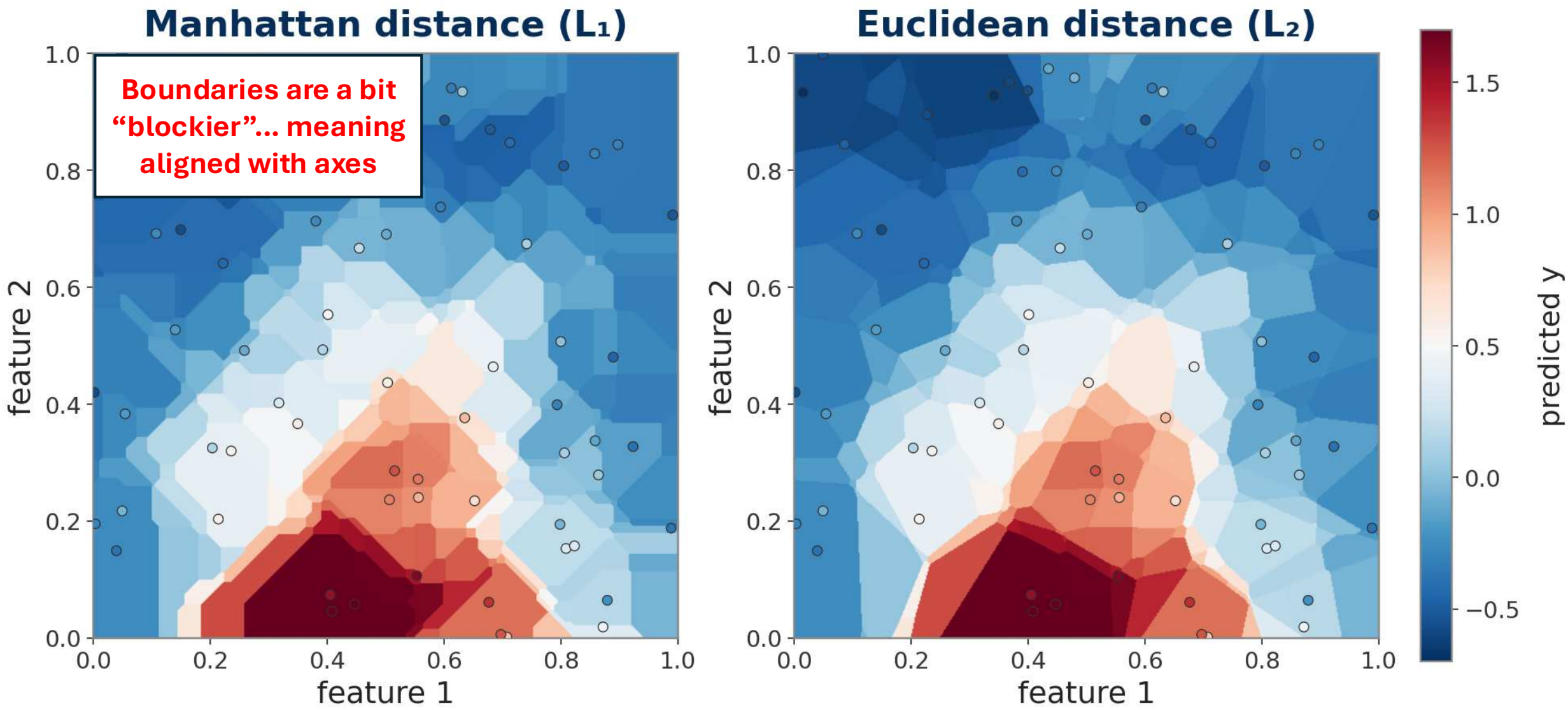


Notice that depending on which distance metric you choose, **the closest datapoint will differ…**
Leading to different model predictions in KNN regression

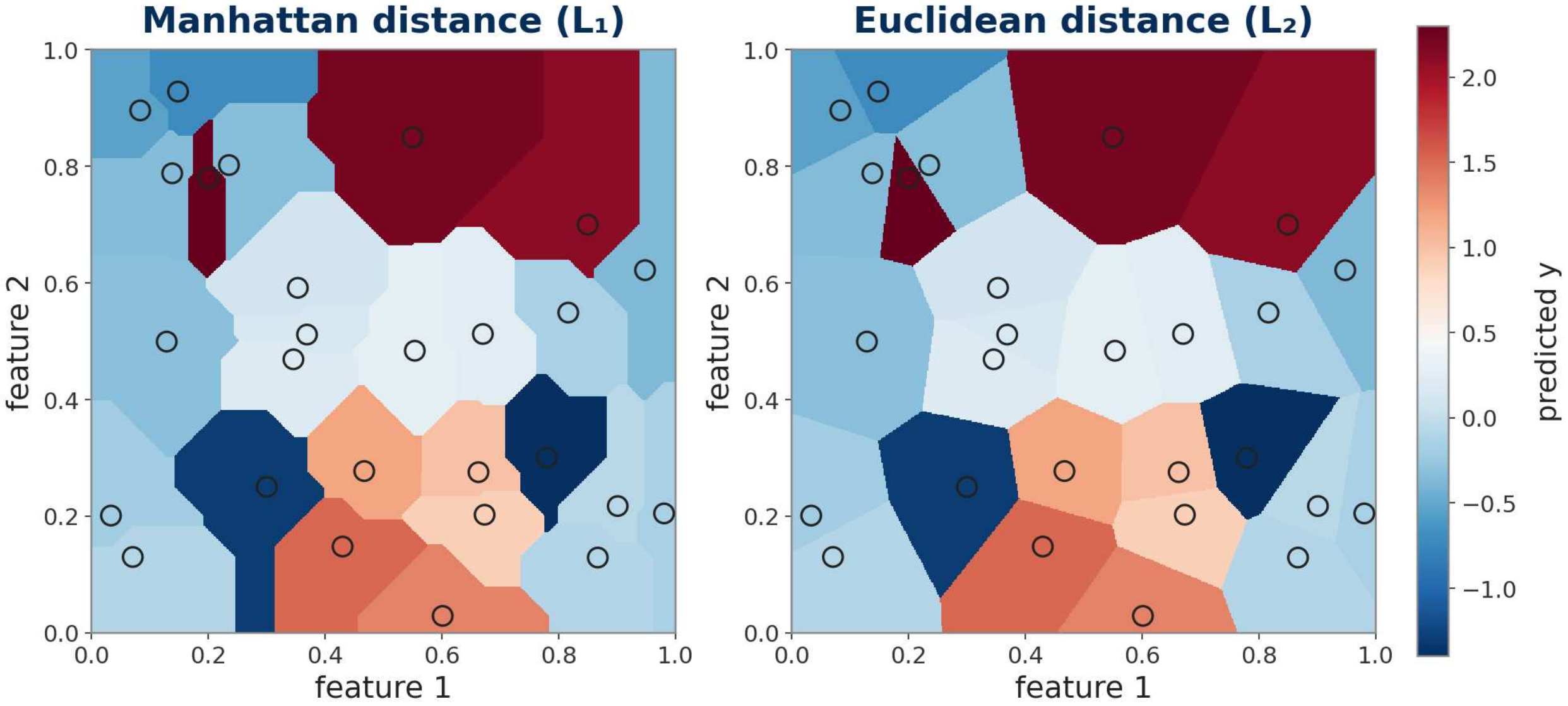
Manhattan nearest neighbor

Euclidean nearest neighbor

Same data and same k, but different norm



A clearer example, using with lower *k*



When to use which distance?

Use Manhattan distance when…

- **Outliers are present and you want to disregard them.** Large differences are not squared, so one feature won't dominate the distance as easily
- **Features contribute independently.** The total distance is simply the sum of differences across dimensions
- **Working in high dimensions**. L_1 often degrades less quickly than L_2

When to use which distance?

Use Manhattan distance when…

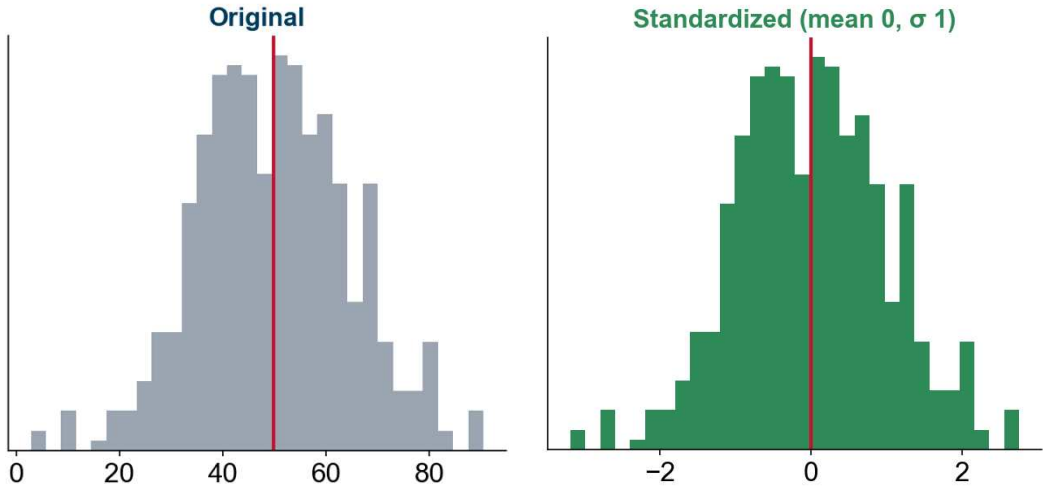
- **Outliers are present.** Large differences are not squared, so one feature won't dominate the distance as easily
- **Features contribute independently.** The total distance is simply the sum of differences across dimensions
- **Working in high dimensions**. L_1 often degrades less quickly than L_2

Use Euclidean distance when…

- **The underlying notion of similarity is geometric.**
- **Large differences should matter more.** Squaring emphasizes these.
- **The data are smooth and approximately isotropic.**

→ This one is usually the default – much more common in ML. Good starting point.

When to use which distance?



$$Z = \frac{x - \mu}{\sigma}$$

1) Subtract the mean to **zero-center** data
2) Divide by Z to **standardize** it

Remember that we should ***standardize*** our data to ensure features are reasonably **isotropic.**

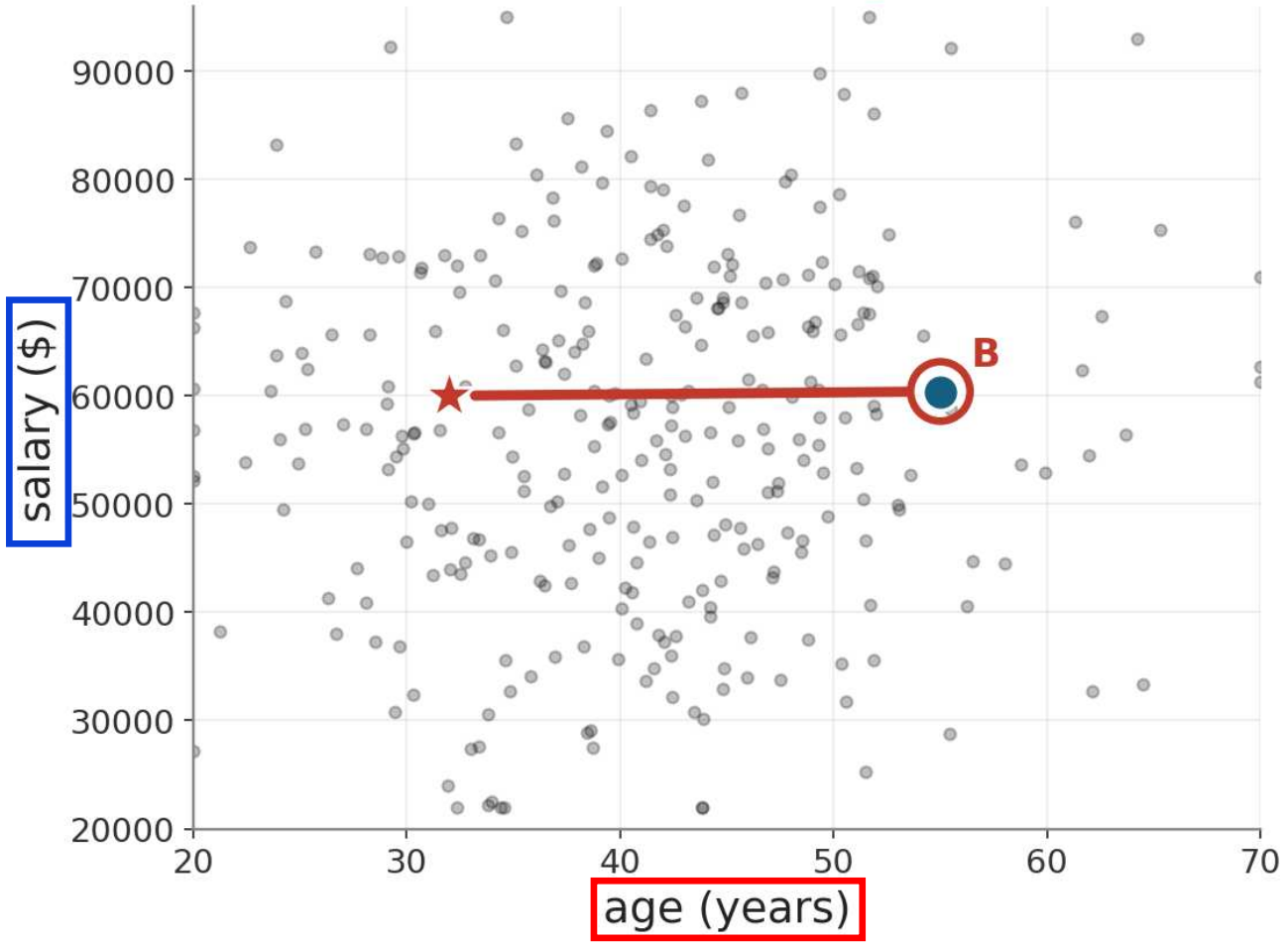
(comparable along each dimension)

Use Euclidean distance when…

- **The underlying notion of similarity is geometric.**
- **Large differences should matter more.** Squaring emphasizes these.
- **The data are smooth and approximately isotropic.**

Standardization matters a lot for KNNs!

Without standardization :-(



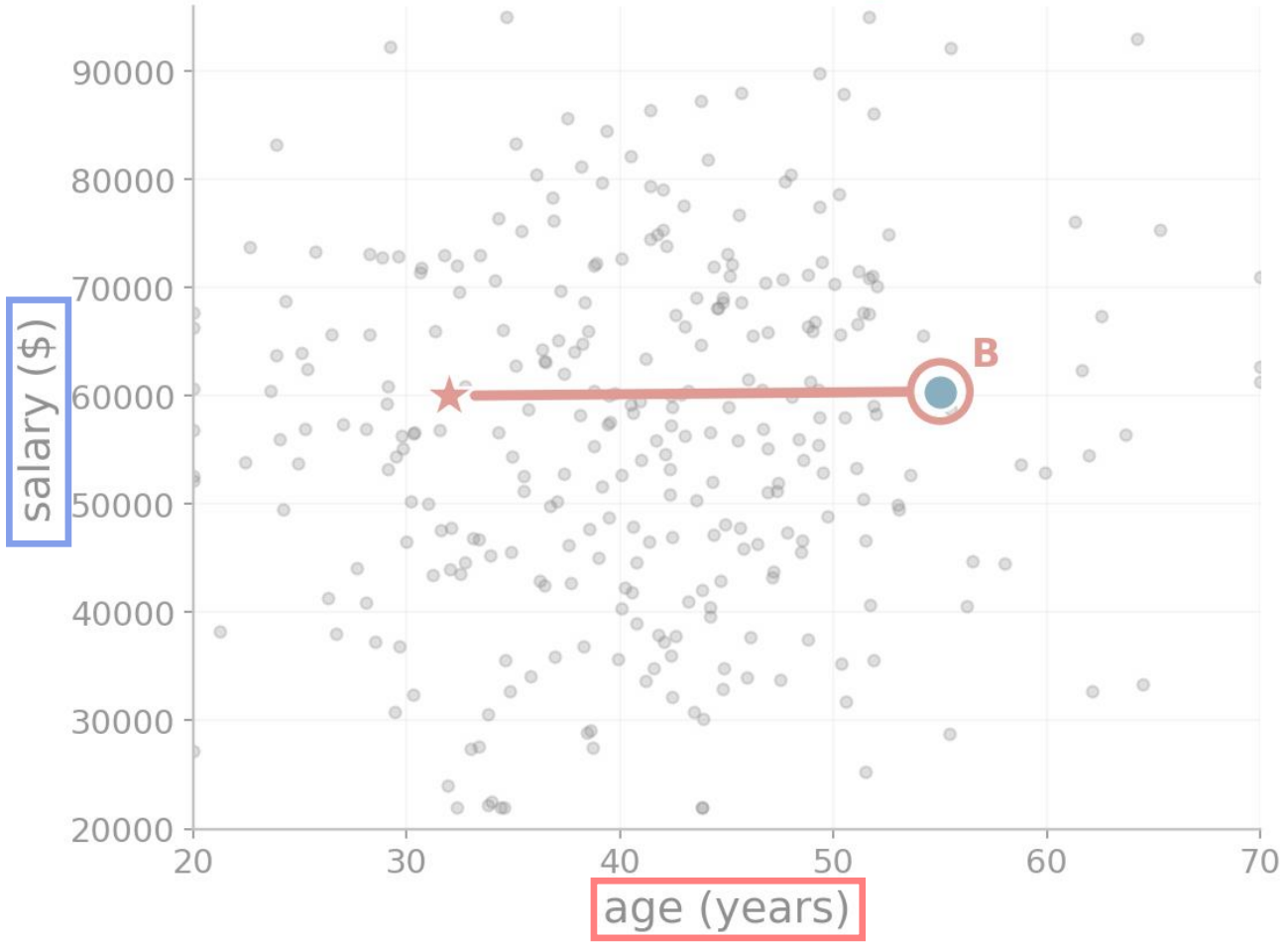
The “nearest neighbor” doesn’t mind being far away on the age scale, since that scale is tiny compared to salary

But is a 32-year-old really that similar to a 56-year-old?

Because **salary** is much larger than **age**, it dominates the distance metric!

Standardization matters a lot for KNNs!

Without standardization :-(

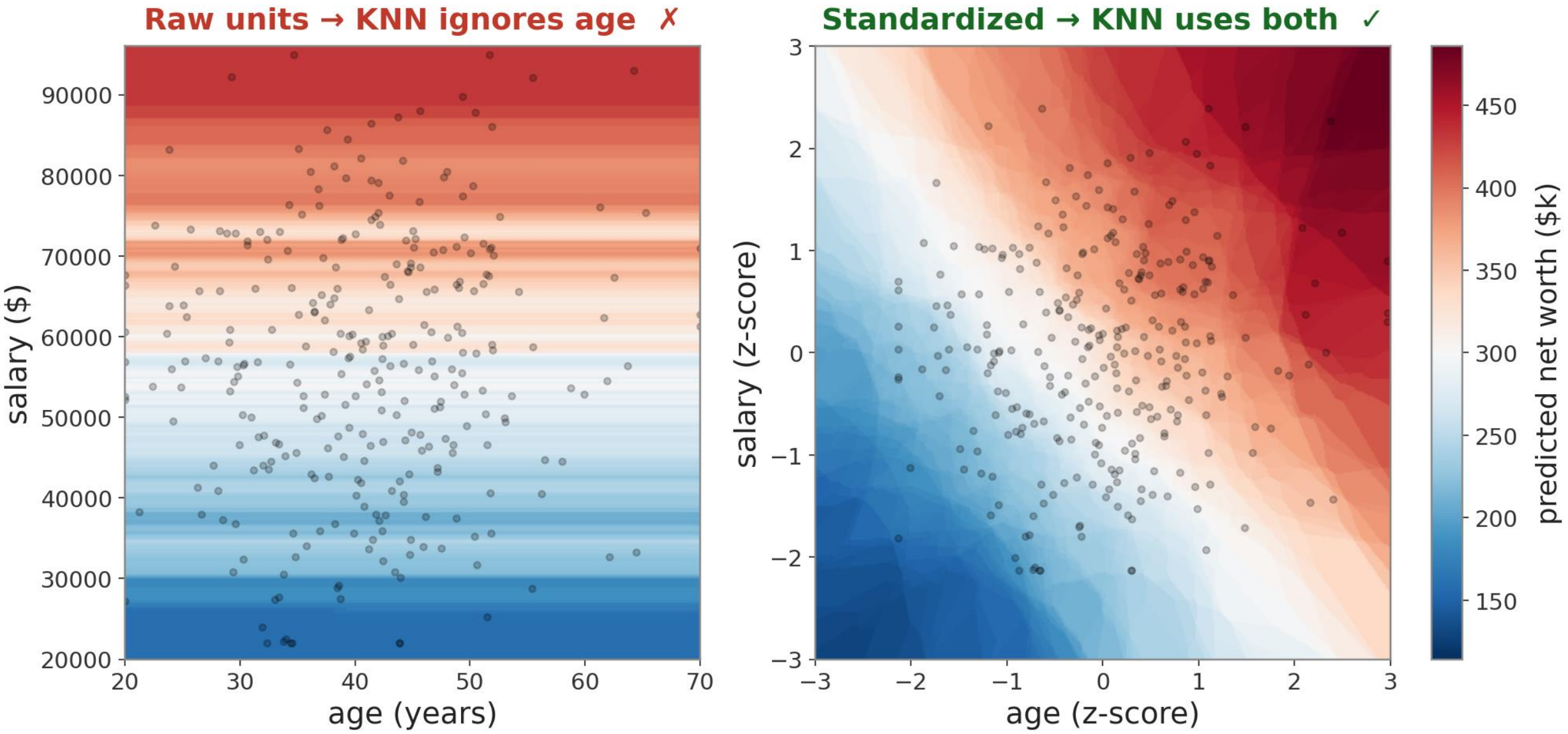


With standardization :-)



Because **salary** is much larger than **age**, it dominates the distance metric!

Standardization matters a lot for KNNs!

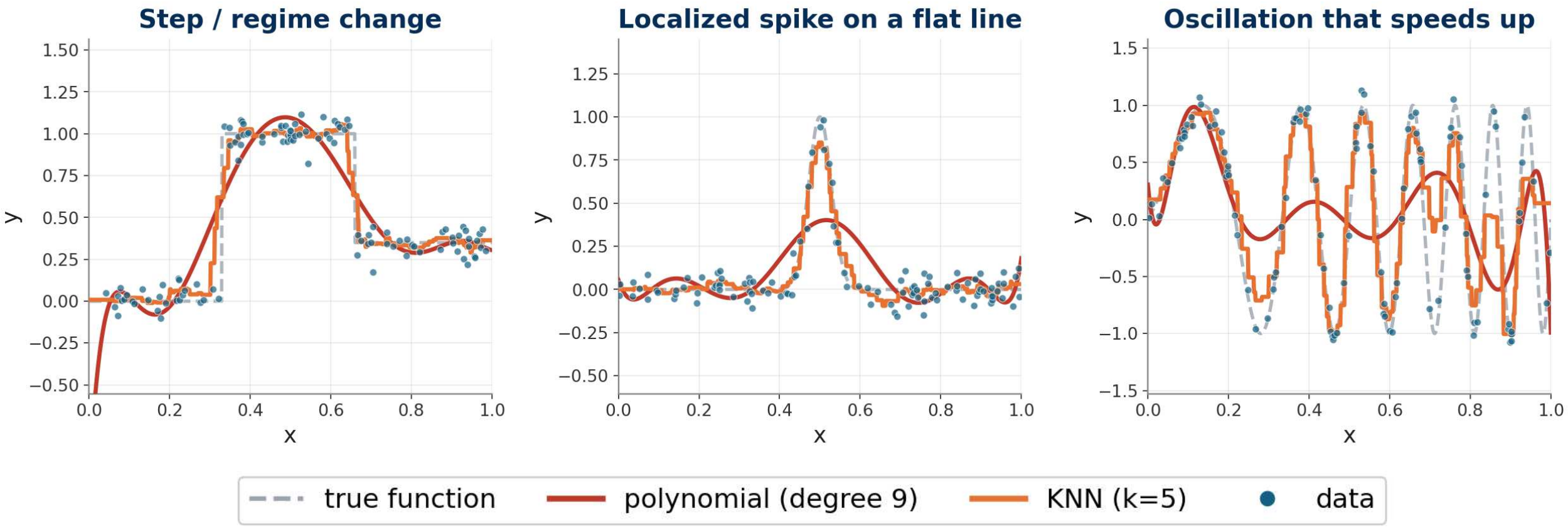


Questions?

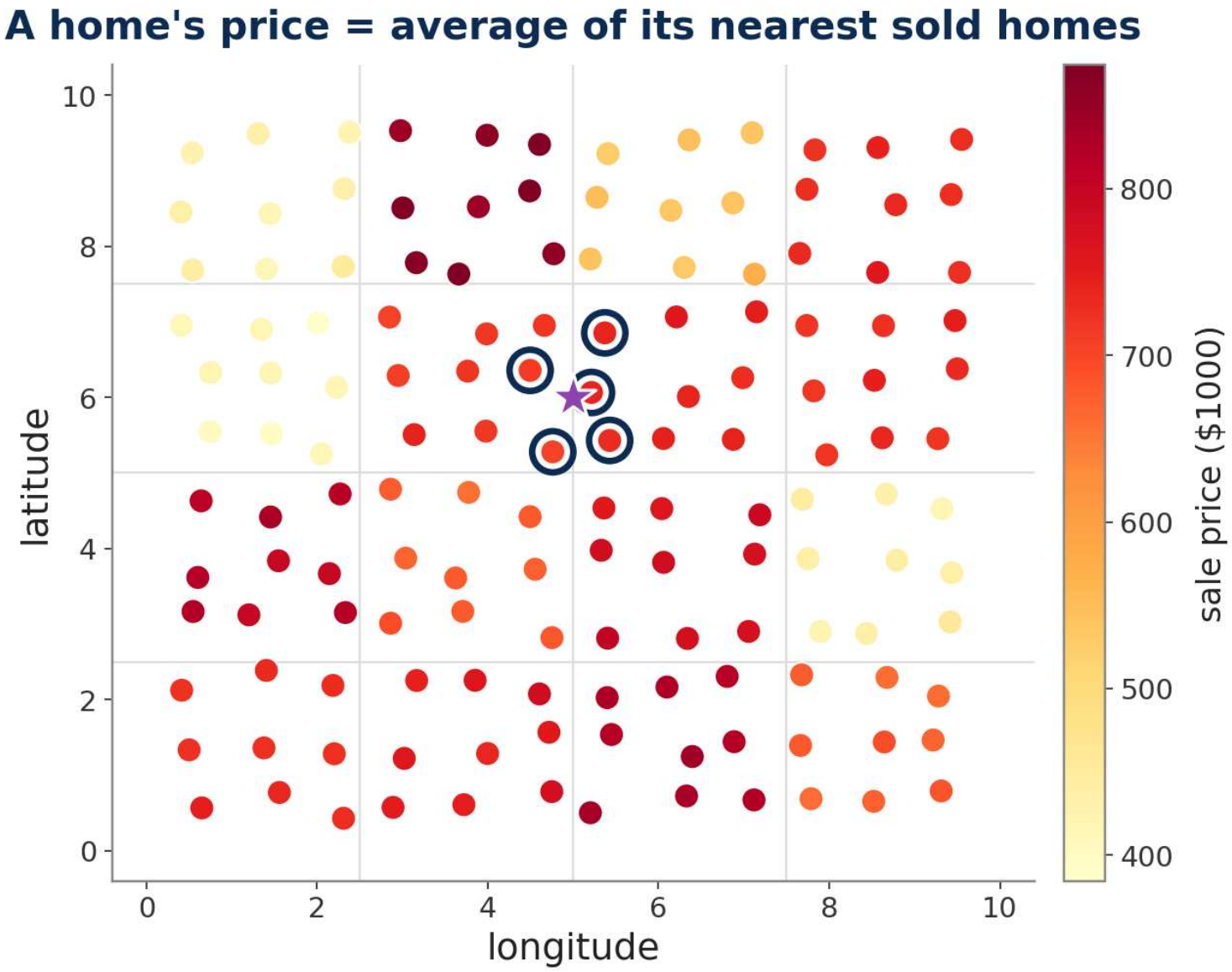
KNN *vs.* Linear Regression

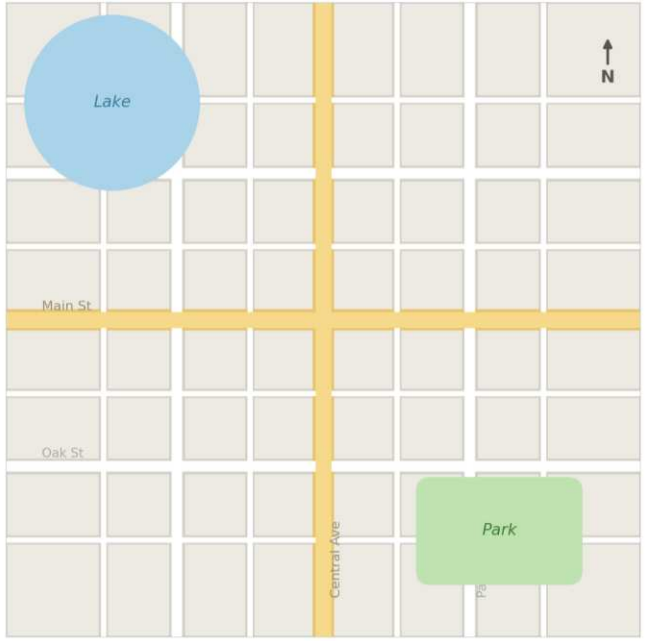
When to use KNNs

1. The function you're trying to learn has **complicated behavior / fluctuations**, and you **don't know what its form might be** (*e.g.*, when a polynomial fails…)



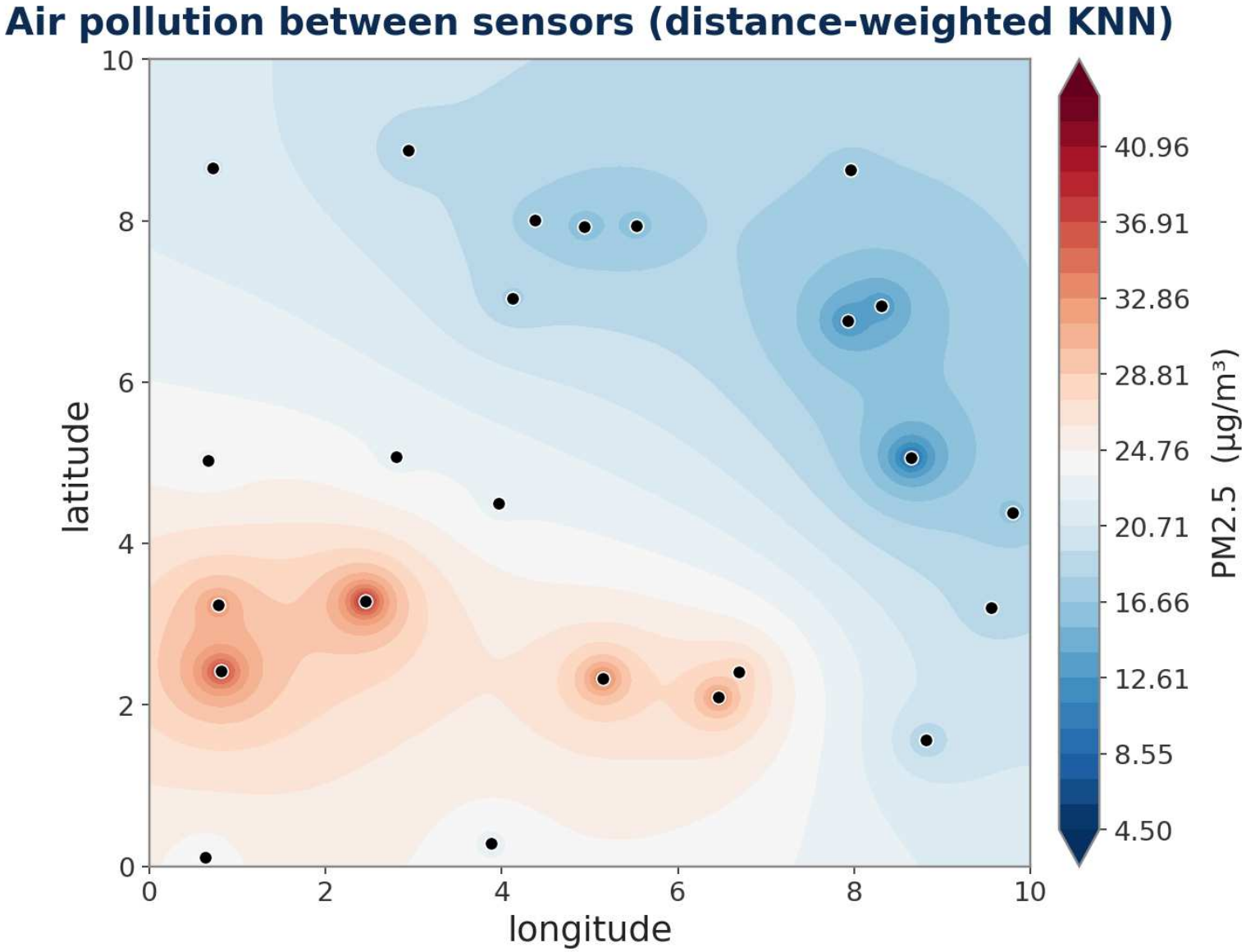
Real-world examples of KNN usage

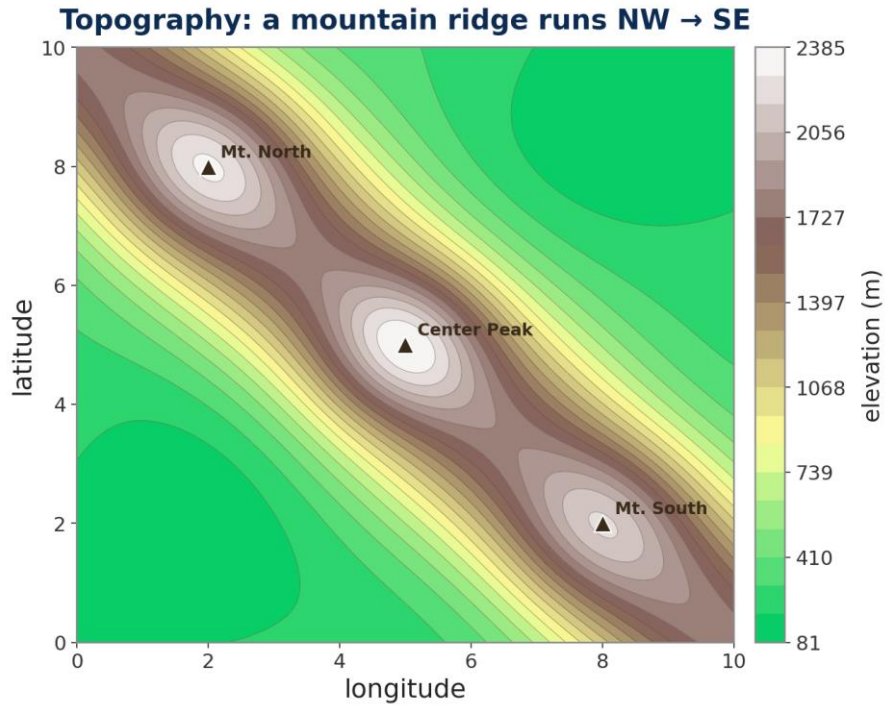




Home prices change quite abruptly across different neighborhoods – forming "clusters" that are well suited for KNN

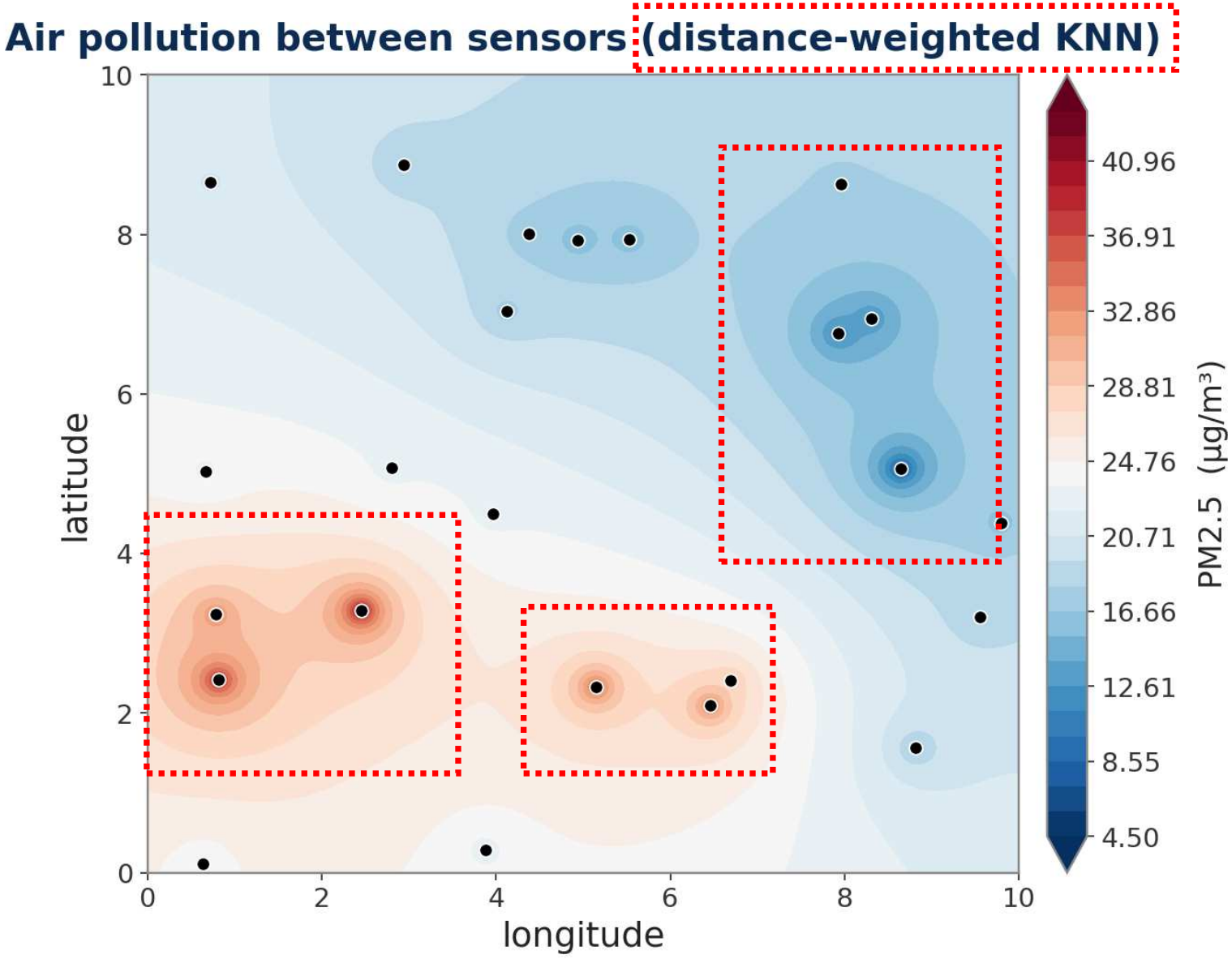
Real-world examples of KNN usage





Pollution levels are heavily affected by geography, causing abrupt changes which KNNs can handle

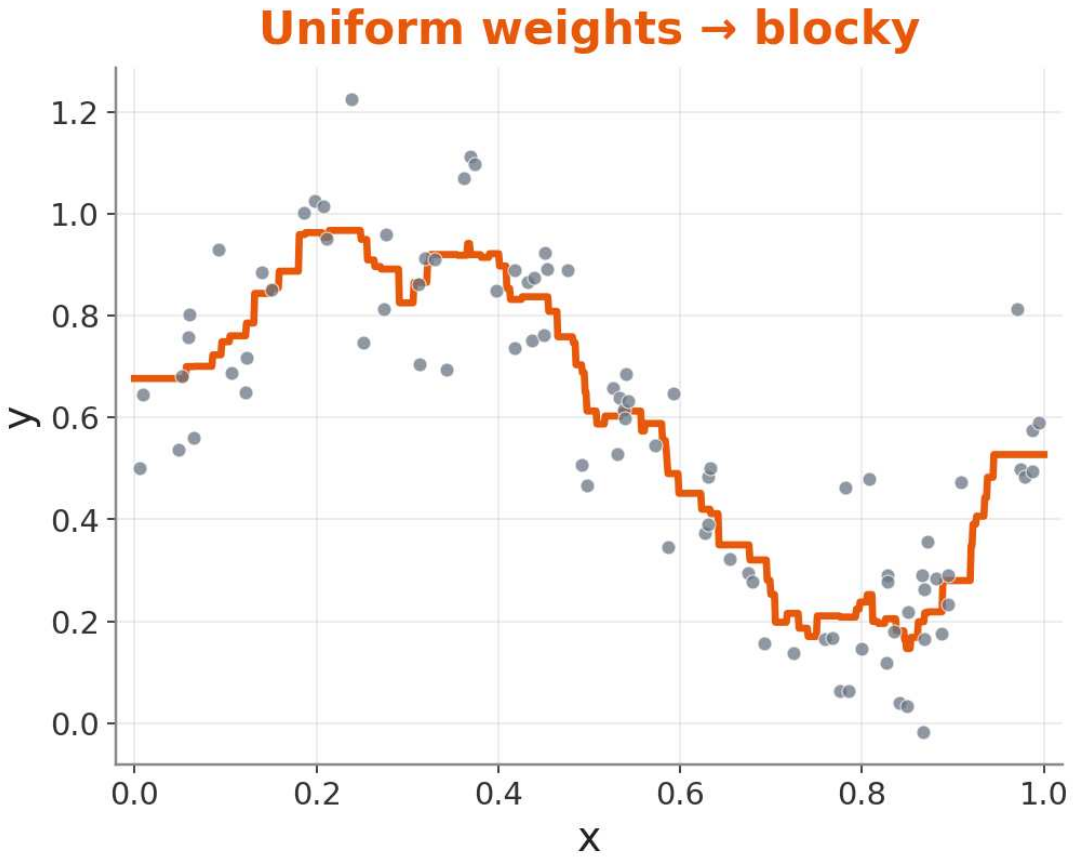
Real-world examples of KNN usage

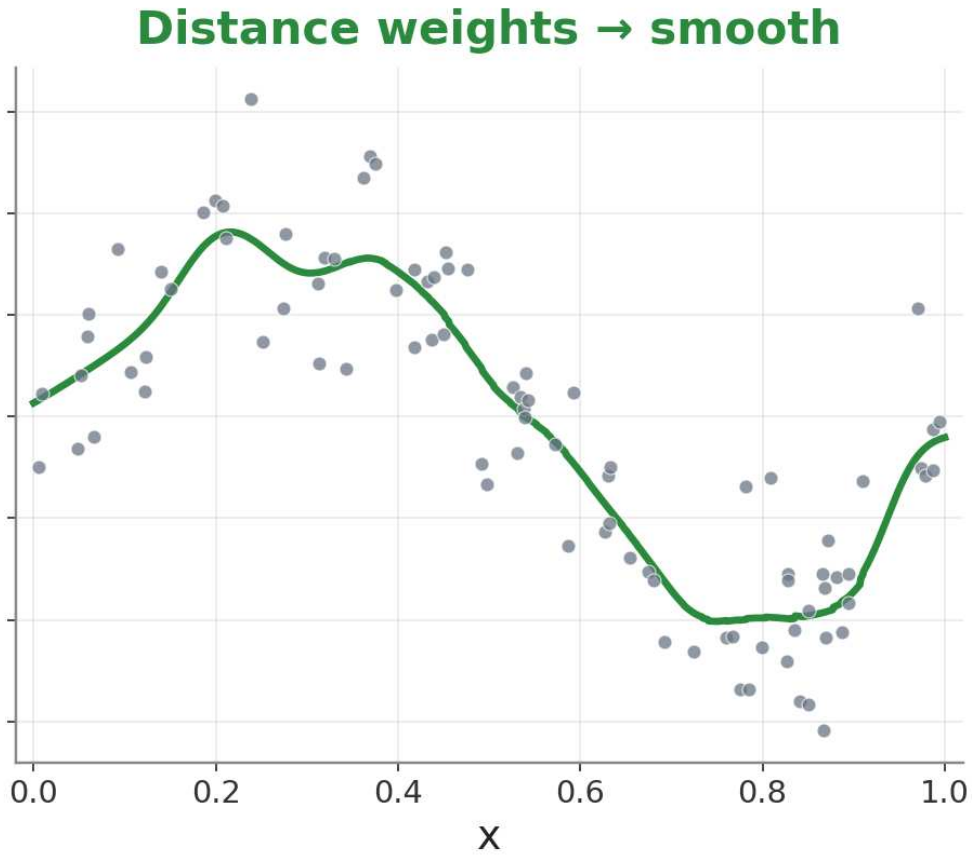


Notice the output map is much smoother and more “circular” than what we’ve seen previously…

This is because we’re using **distance-weighted** KNN as opposed to simple averaging

Weighted averaging creates smoother curves





$$\hat{y} = \frac{1}{k}\sum_i y_i \text{ for } i \in N_k(x')$$

Simple average

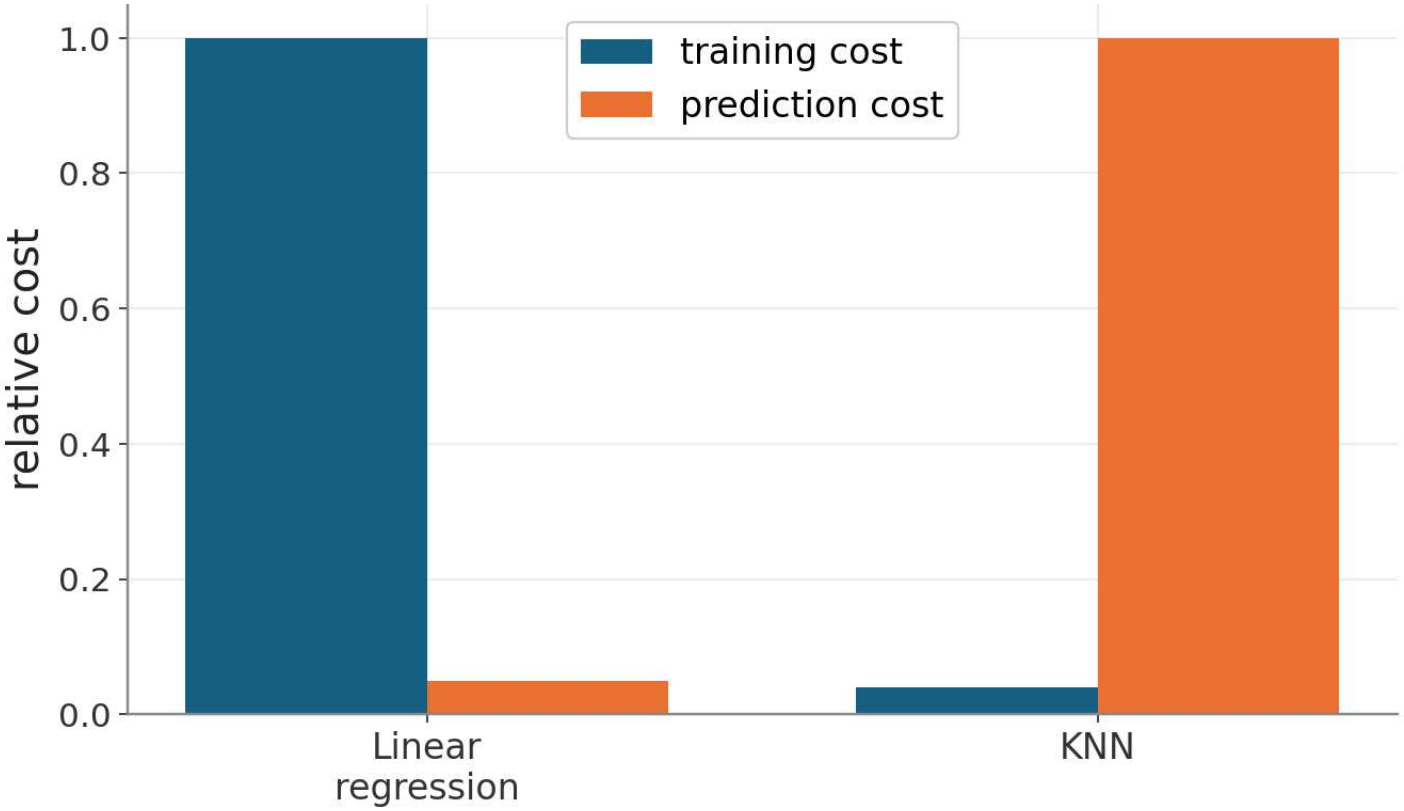
$$\hat{y} = \sum_i w_i y_i \,/\, \sum_i w_i \text{ for } i \in N_k(x')$$

The weights are: $w_i \propto e^{-D(x,x_i)}$

When to use KNNs

2. You **don't want to spend time re-training** your model every time you collect new data, and you are okay with spending more time on predictions

KNN is a "lazy" learner (no training needed)



Training is basically free – just store the data – but prediction searches the whole dataset.

As a result, compute time scales with $O(nd)$

Number of samples

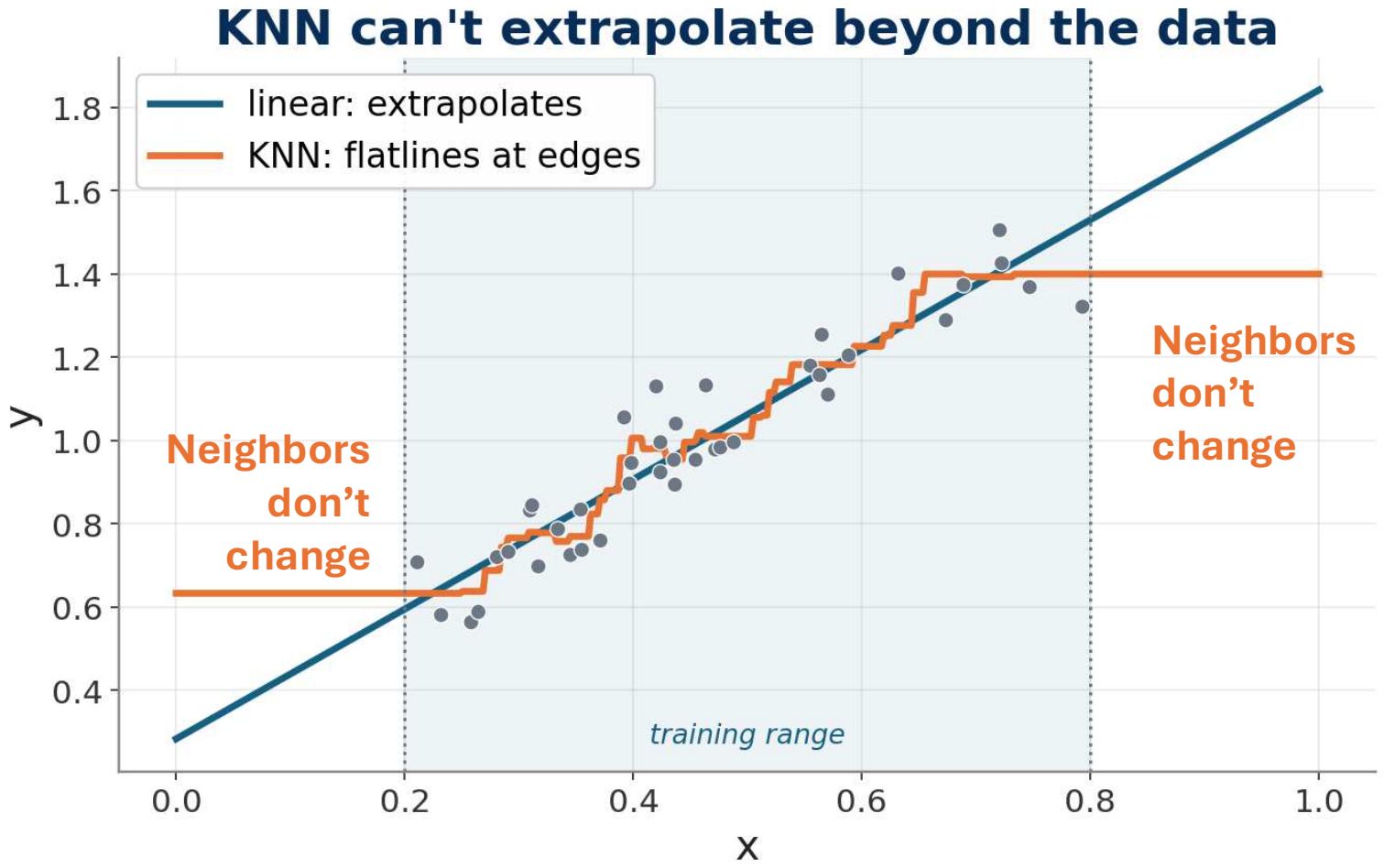
Number of dimensions

Where KNNs fail, and linear regression is preferred

1. You want to **extrapolate** beyond the training data

By design, KNN cannot extrapolate! Its predictions will remain flat outside the range of the training data

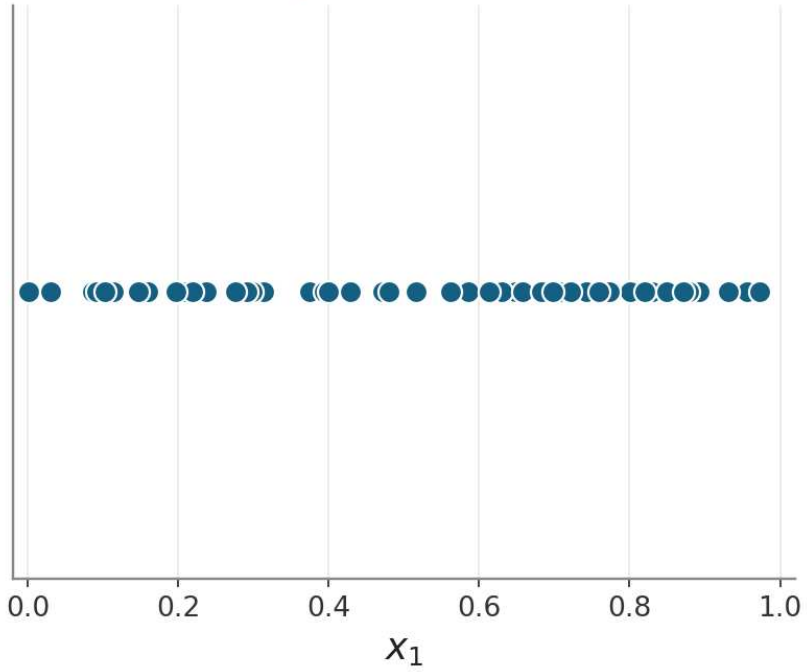
Linear regression *can* extrapolate; though whether these are good predictions depends on the problem…



Where KNNs fail, and linear regression is preferred

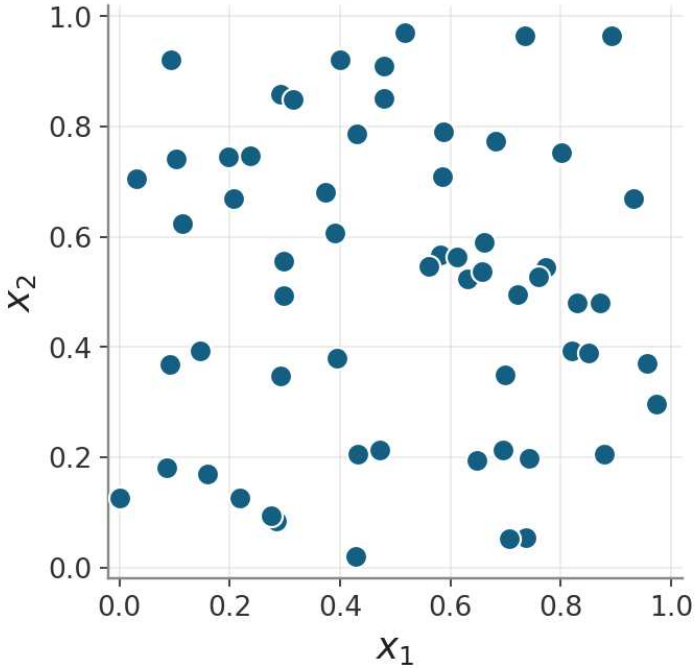
2. You have **many features in high dimensions** → *the curse of dimensionality*
   (you *can* use KNN here, but regularized linear regression is usually preferred)

1D: 60 points on a line



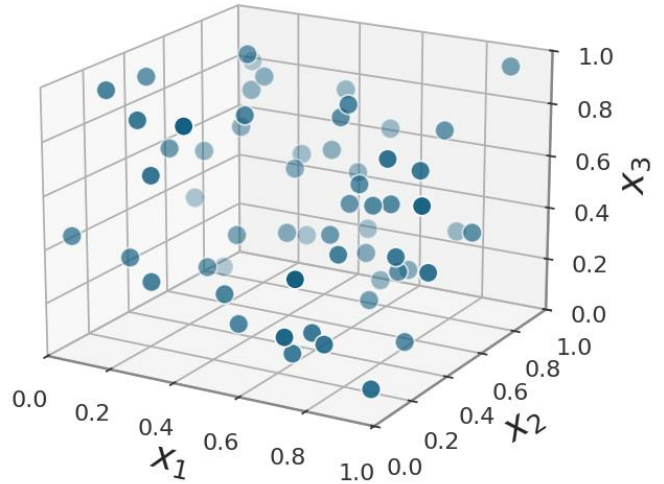
Avg. neighbor distance ~ 0.01

2D: same 60 points in a square



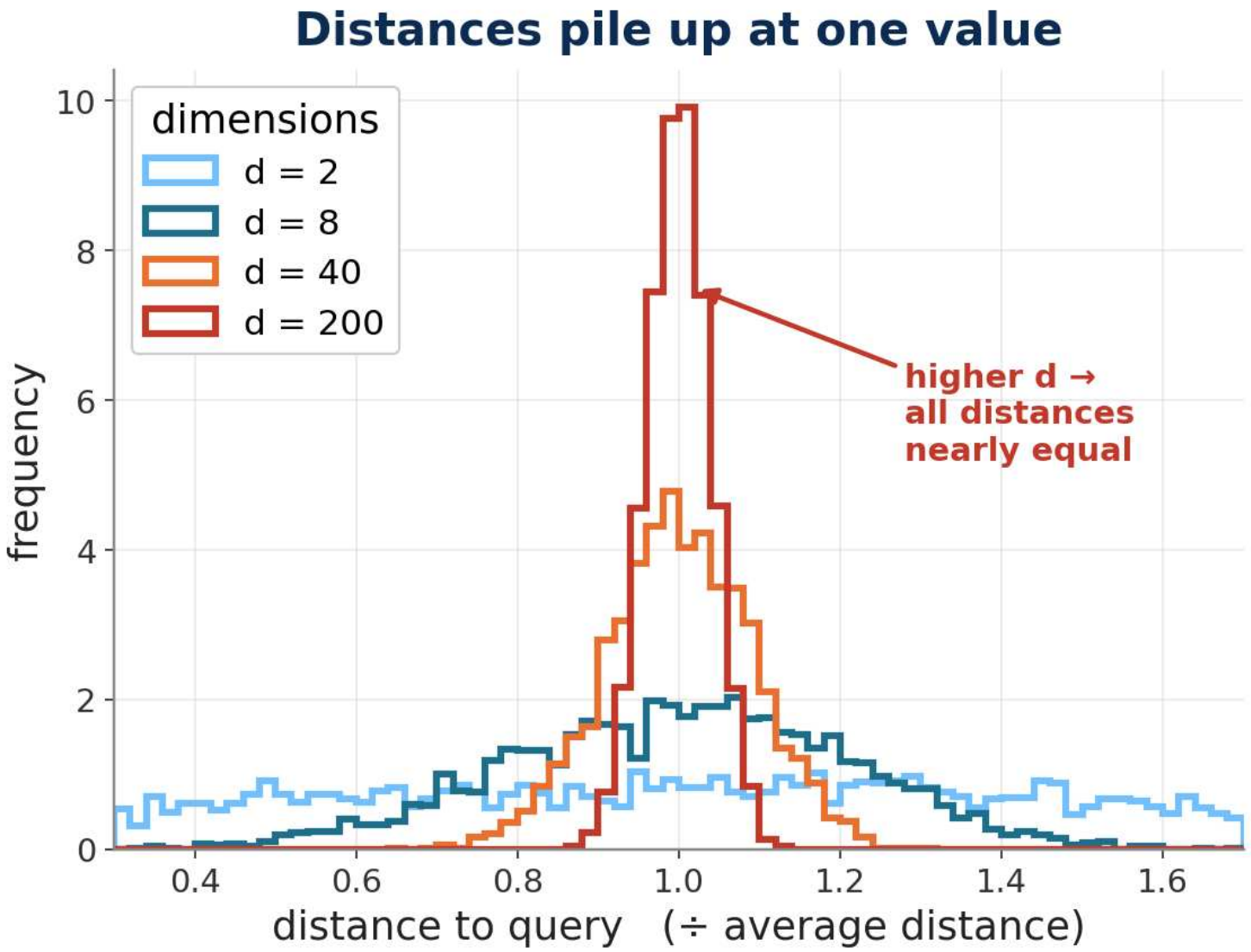
Avg. neighbor distance ~ 0.07

3D: same 60 points in a cube



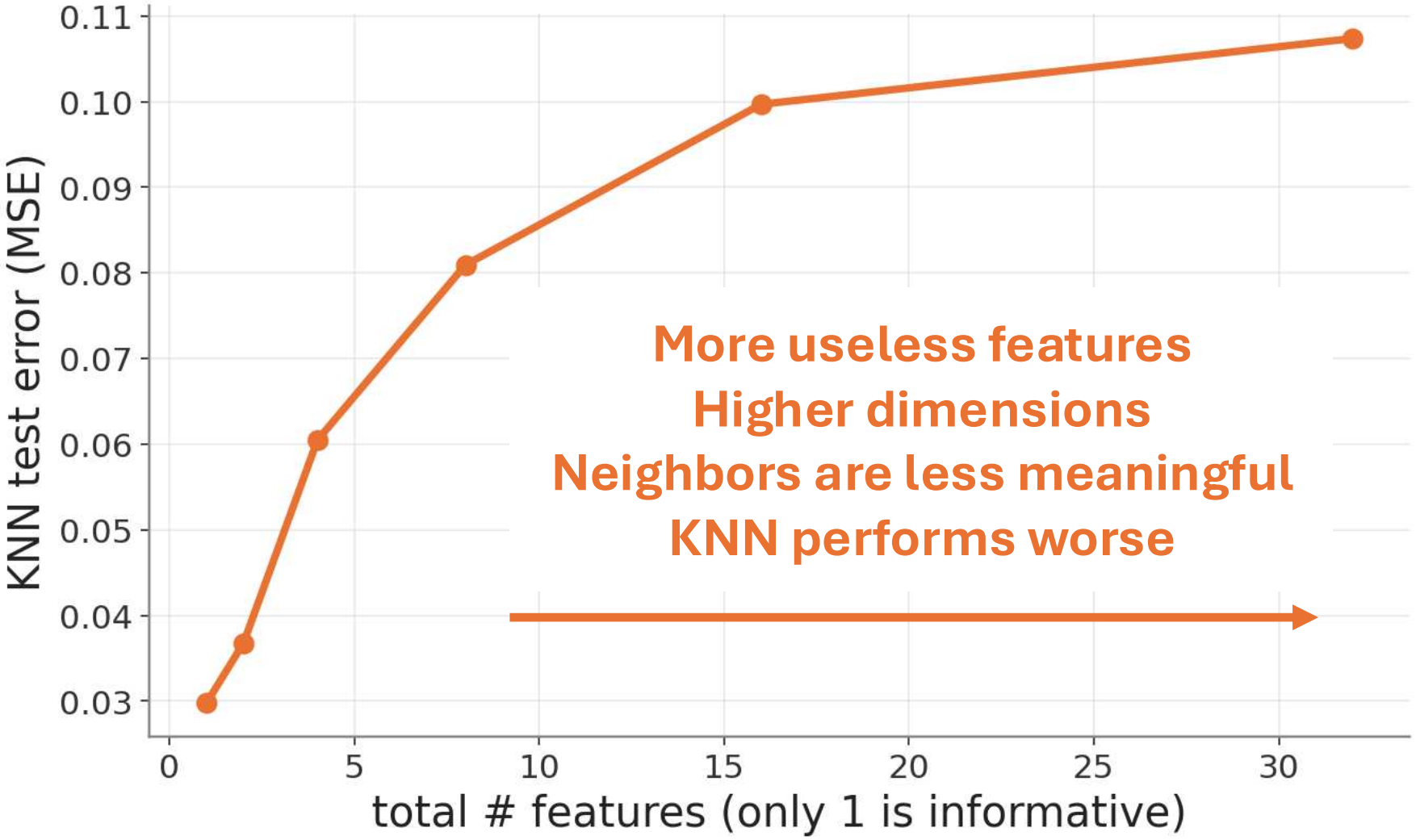
Avg. neighbor distance ~ 0.16

Why KNNs fail in high dimensions



- When we have many features (high dimension) – all points look equally similar
- As a result, the definition of a “neighbor” starts to lose any real meaning

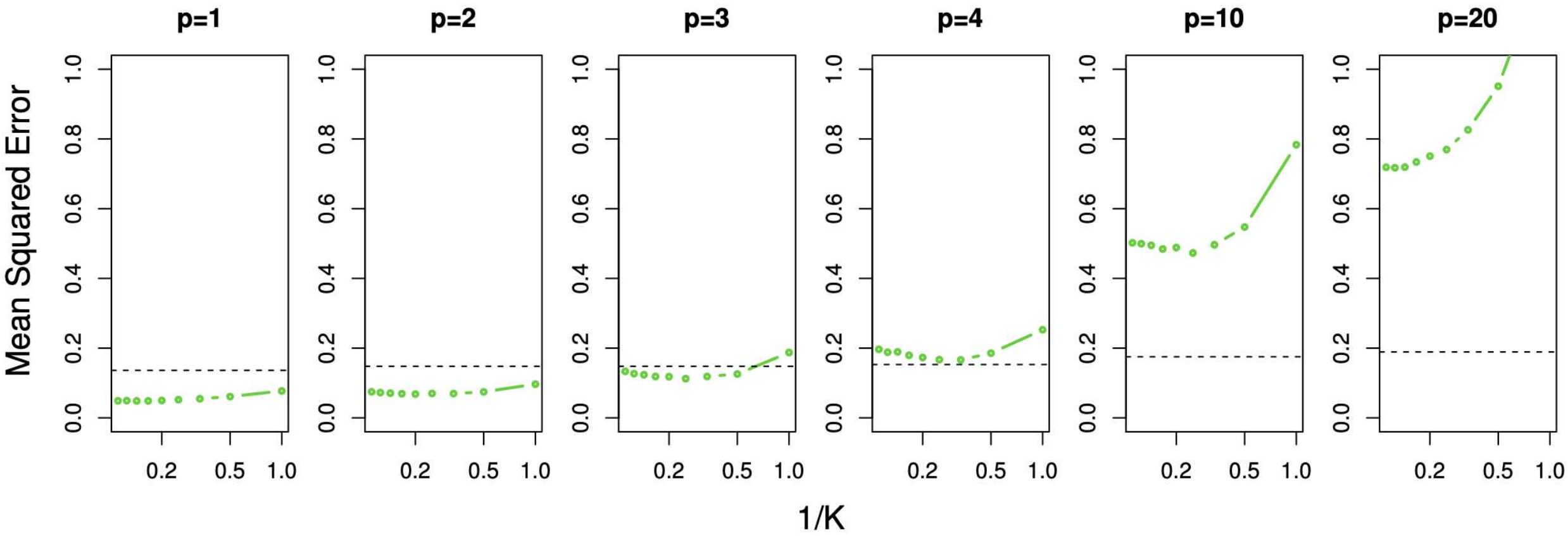
Adding “junk” features hurts KNNs



In contrast:
Ridge and lasso regression would just learn to ignore these features!

So KNN is at a disadvantage here…
Can we solve this by choosing a better k?

This can’t be solved by choosing a good k



As predictors (*p*) grow from 1 to 20, KNN's test error climbs no matter how you set *k*

Source: James, Witten, Hastie & Tibshirani, “An Introduction to Statistical Learning”, Fig. 3.20

Where KNNs fail, and linear regression is preferred

2. You have **many features in high dimensions** → *the curse of dimensionality*



Either choose your features very carefully, only keeping a few that you suspect matter the most

...or...

Use a parametric model such as Ridge or Lasso regression, which can learn to throw out features

Where KNNs fail, and linear regression is preferred

Final benefit of linear regression *vs.* KNN:

3. You want a model that is **interpretable** and/or **reusable**

Sales of a product depends on advertising through different venues



TV budget	Radio budget	Paper budget	Unit sales
$100k	$8k	$59k	11k
$80k	$25k	$18k	9k
$195k	$42k	$92k	18k
$232k	$21k	$5k	23k

KNNs do *not* have any coefficients to be interpreted

$$\hat{y} = \beta_0 + \beta_1 x_{\text{TV}} + \beta_2 x_{\text{Radio}} + \beta_3 x_{\text{News}}$$

These coefficients tell you something about the important of each feature

Questions?

What to know for HW + Exams

- How to **calculate a prediction use *k*-nearest neighbors**
- The influence of distance (Euclidean/Manhattan)
- More importantly: the **influence of *k***, and how you choose it
- When to use KNN *vs.* linear regression