

Reminders

- Homework due Thu 11:59 PM

Today

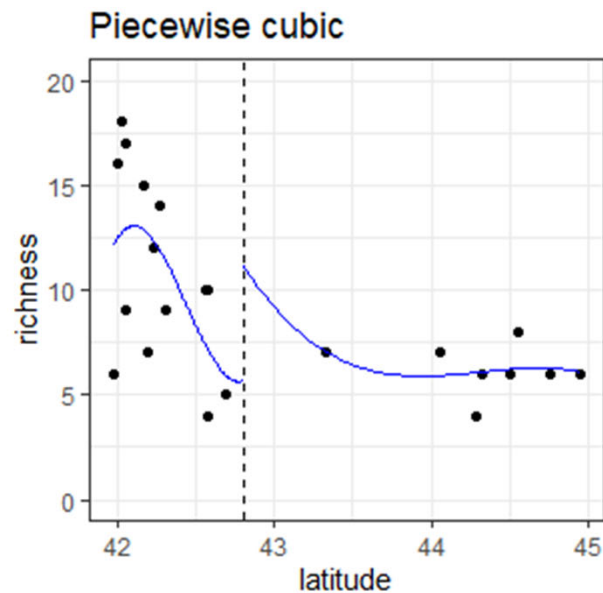
- Model algorithms
 - Smoothing spline algorithm
 - K nearest neighbors
- Training algorithm
 - optimize penalized SSQ
 - regularization
- Theory of bias-variance tradeoff

Smoothing spline model algo

- James et al. Ch 7.1 - 7.5
- Cubic smoothing spline
 - piecewise cubic **polynomial**
- **Knots** (joins) at data
- **Constraints**: continuous first and second derivatives at knots

Building a smoothing spline

Ants data

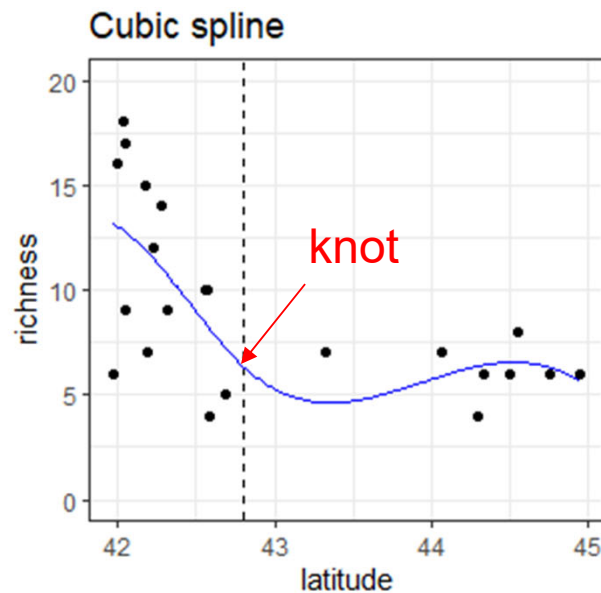
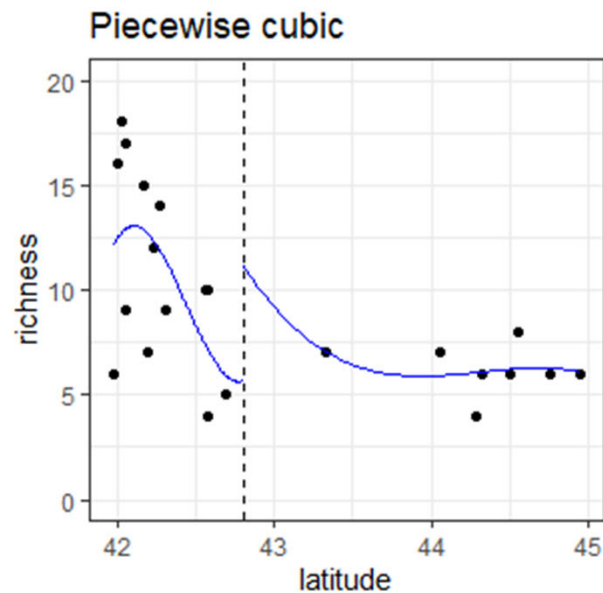


Cubic polynomials:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

Building a smoothing spline

Ants data

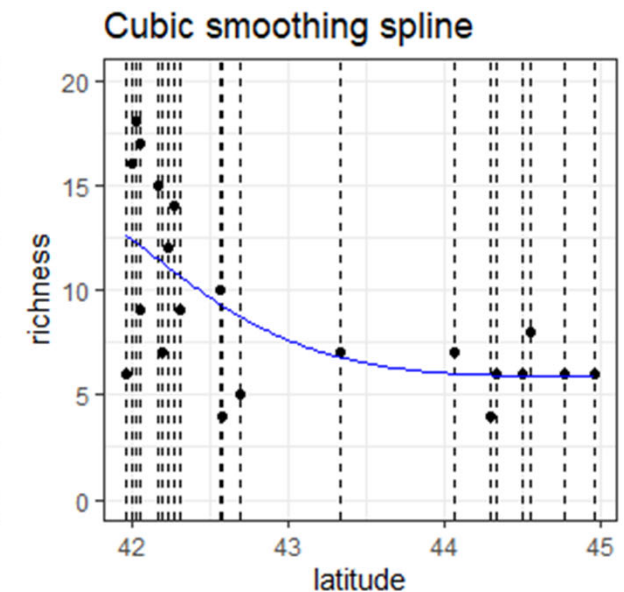
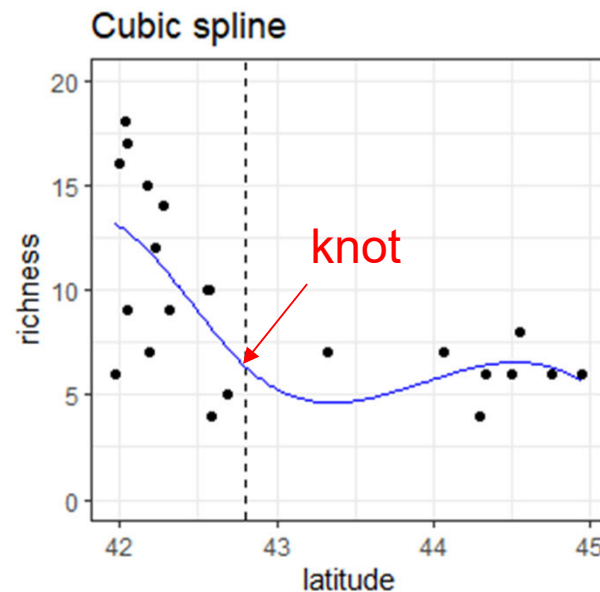
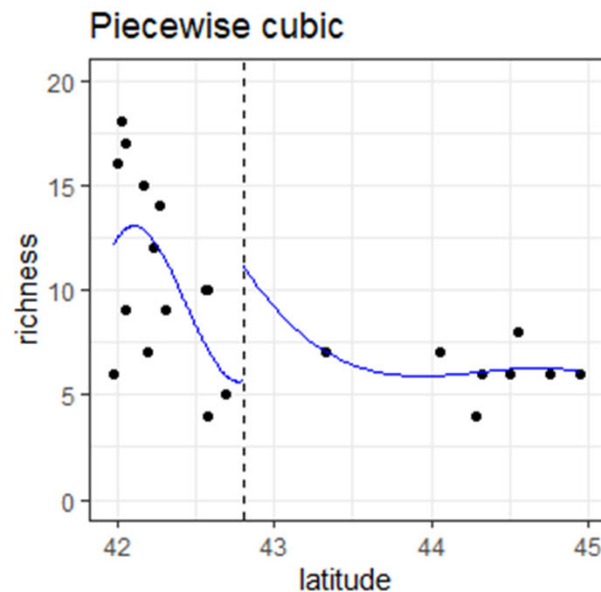


Cubic polynomials:
 $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$

+ continuous 1st & 2nd
derivatives at knot

Building a smoothing spline

Ants data



Cubic polynomials:
 $y = \beta_0 + \beta_1x + \beta_2x^2 + \beta_3x^3$

+ continuous 1st & 2nd
derivatives at knot

+ knots at each datum

Smoothing spline training algo

Penalized least squares

Optimize this objective function:

$$\underbrace{\sum_{i=1}^n (y_i - f(x_i))^2}_{\text{SSQ}} + \underbrace{\lambda \int f''(x)^2 dx}_{\text{Penalty}}$$

Regularization

Shrinks β s of
polynomials
toward zero

Penalty term = “wiggleness”

λ is held constant to optimize

Tuning parameter d.f. is a function of λ . We vary d.f. in an inference algorithm to find the d.f. (hence λ) with the best predictive performance.

Smoothing spline training algo

Optimizing algorithm

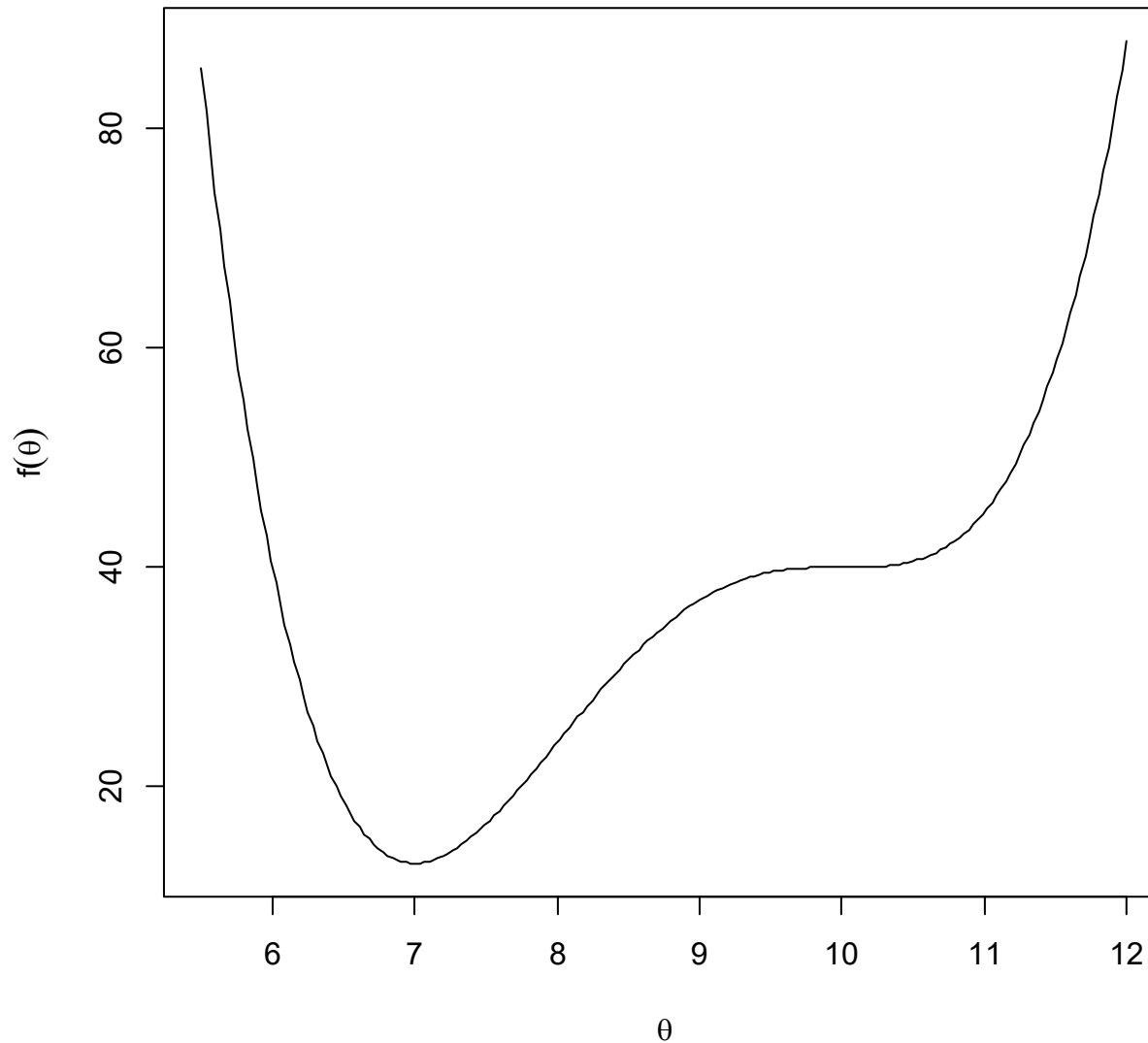
Strategy 2: descent algorithm

Diving deep into the source code for `smooth.spline()` we come to the file `sbart.c` where we find the exact algorithm is due to Forsythe, Malcom & Moler, presumably circa 1977, in turn a slightly modified version of the Algol 60 (!!) procedure `localmin` from Brent (1973) *Algorithms for minimization without derivatives*, Prentice-Hall. Such legacy procedures and code are the case for much numerical work!

The algorithm uses a combination of [golden section search](#) and successive [parabolic interpolation](#). We don't need to worry about these details but the following slides give a sense for descent algorithms in general using the simple bisection algorithm as an example.

A feel for descent algorithms

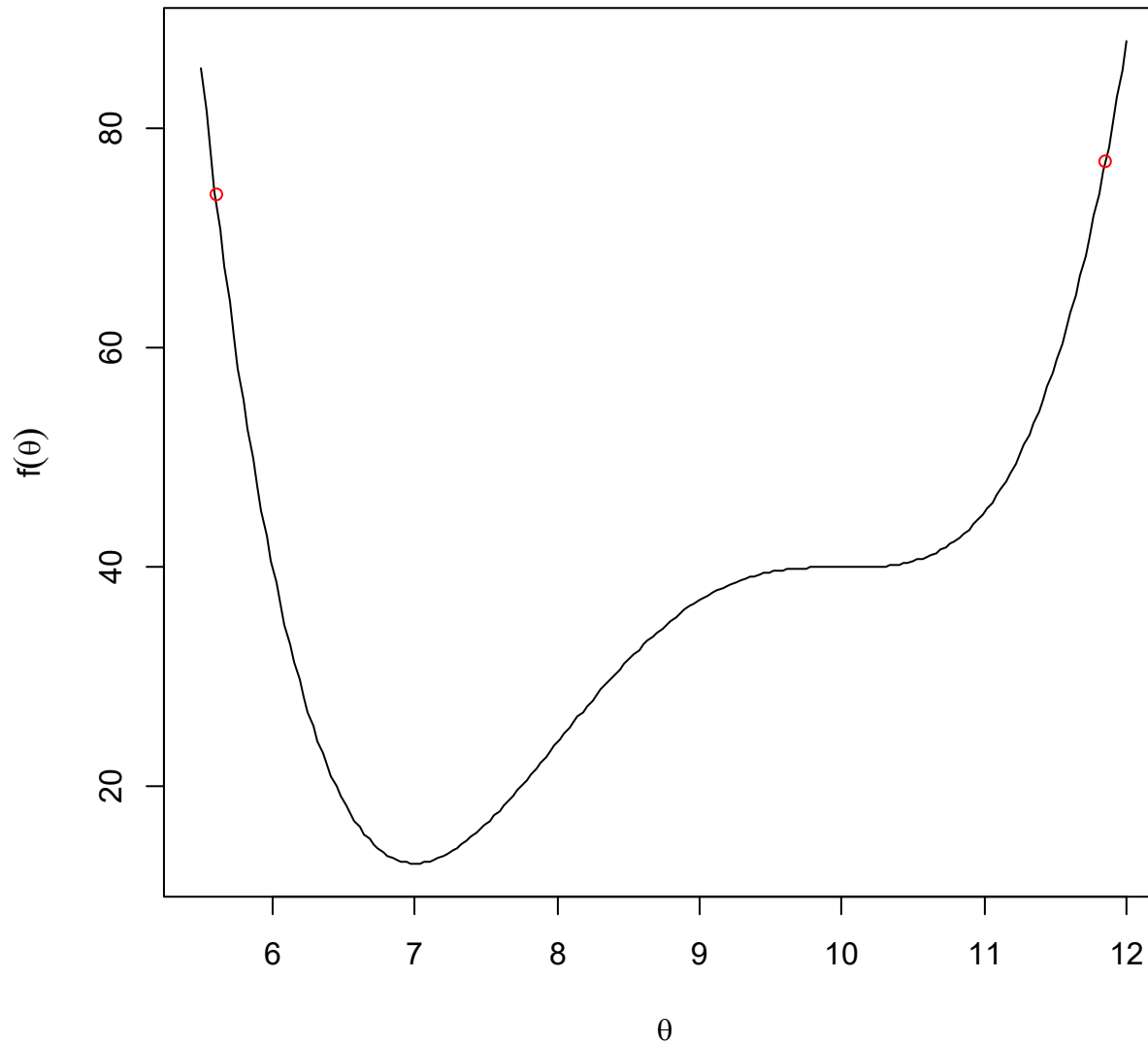
Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum

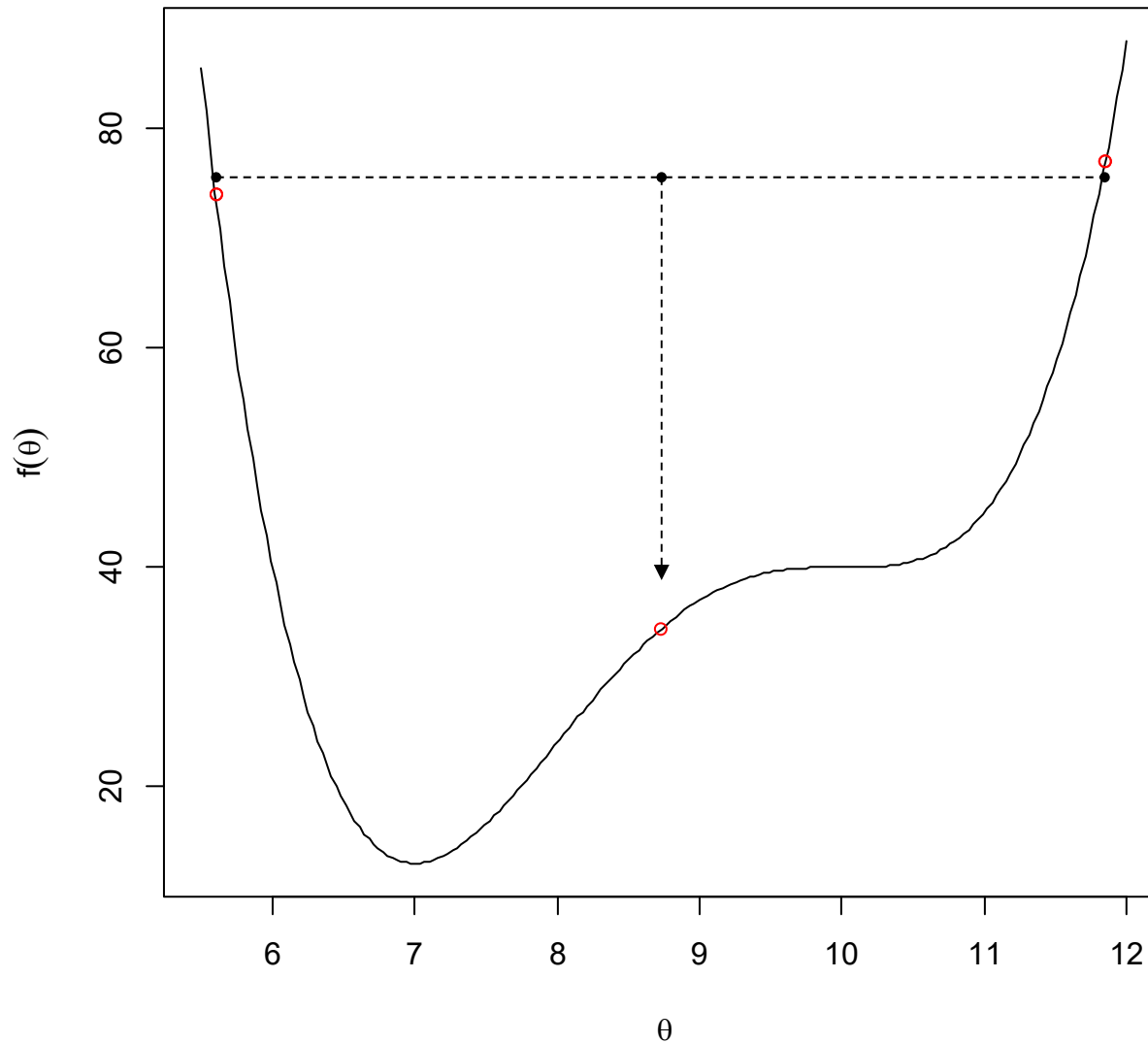


**Bisection
algorithm**

Start with 2 points

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



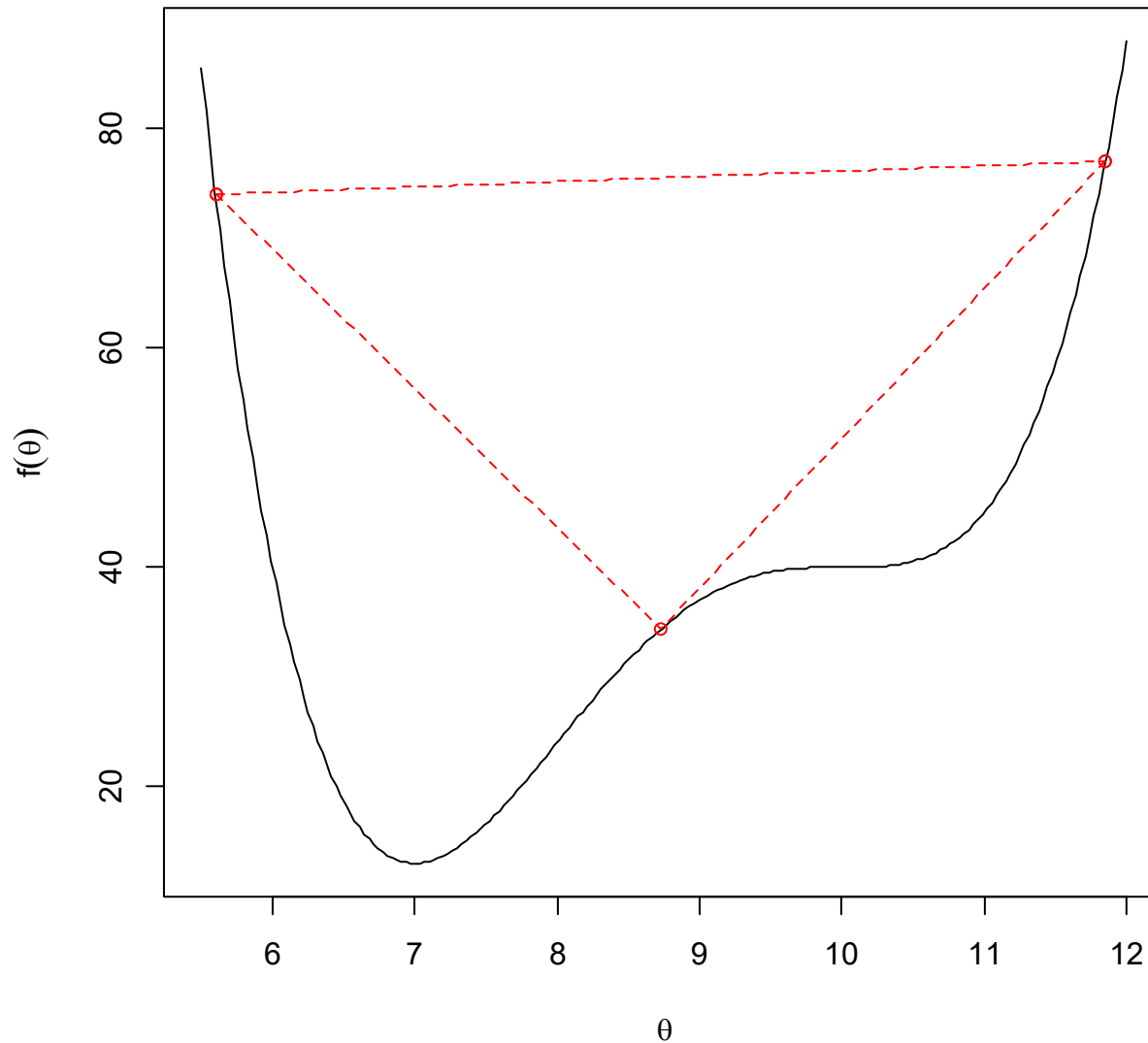
**Bisection
algorithm**

Start with 2 points

Bisect

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

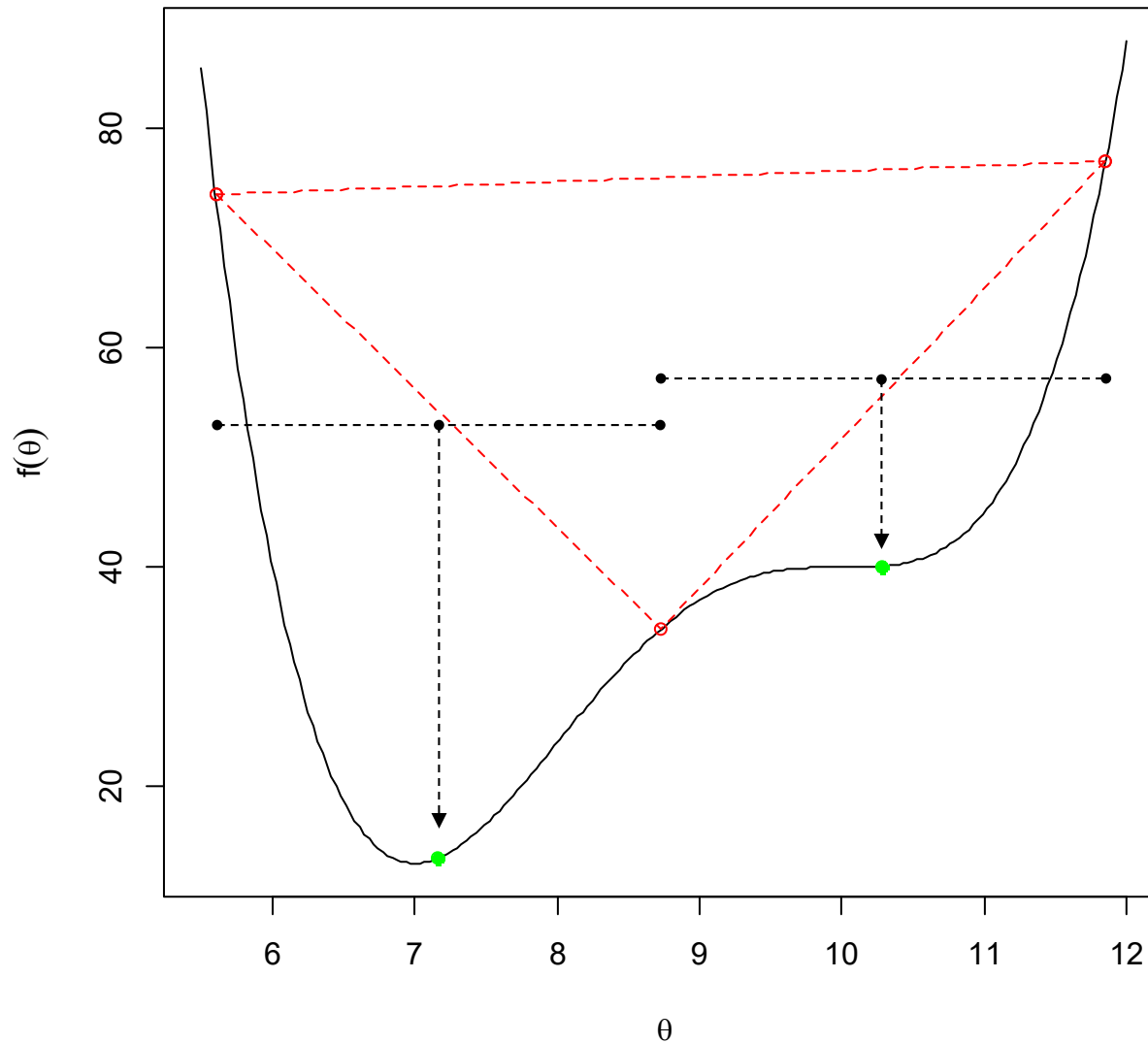
Start with 2 points

Bisect

Make triangle

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

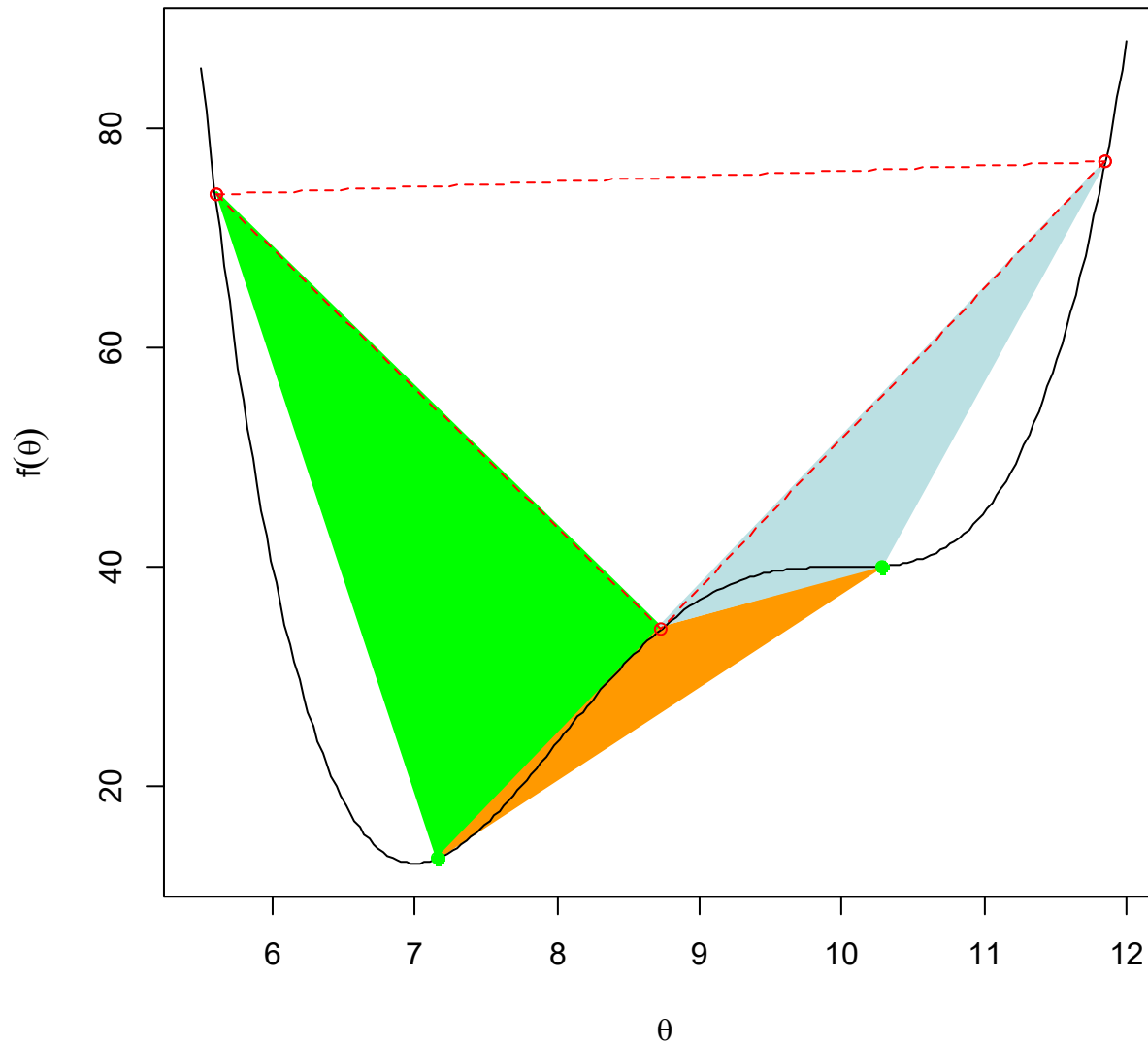
Bisect

Make triangle

Bisect lower sides

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

Bisect

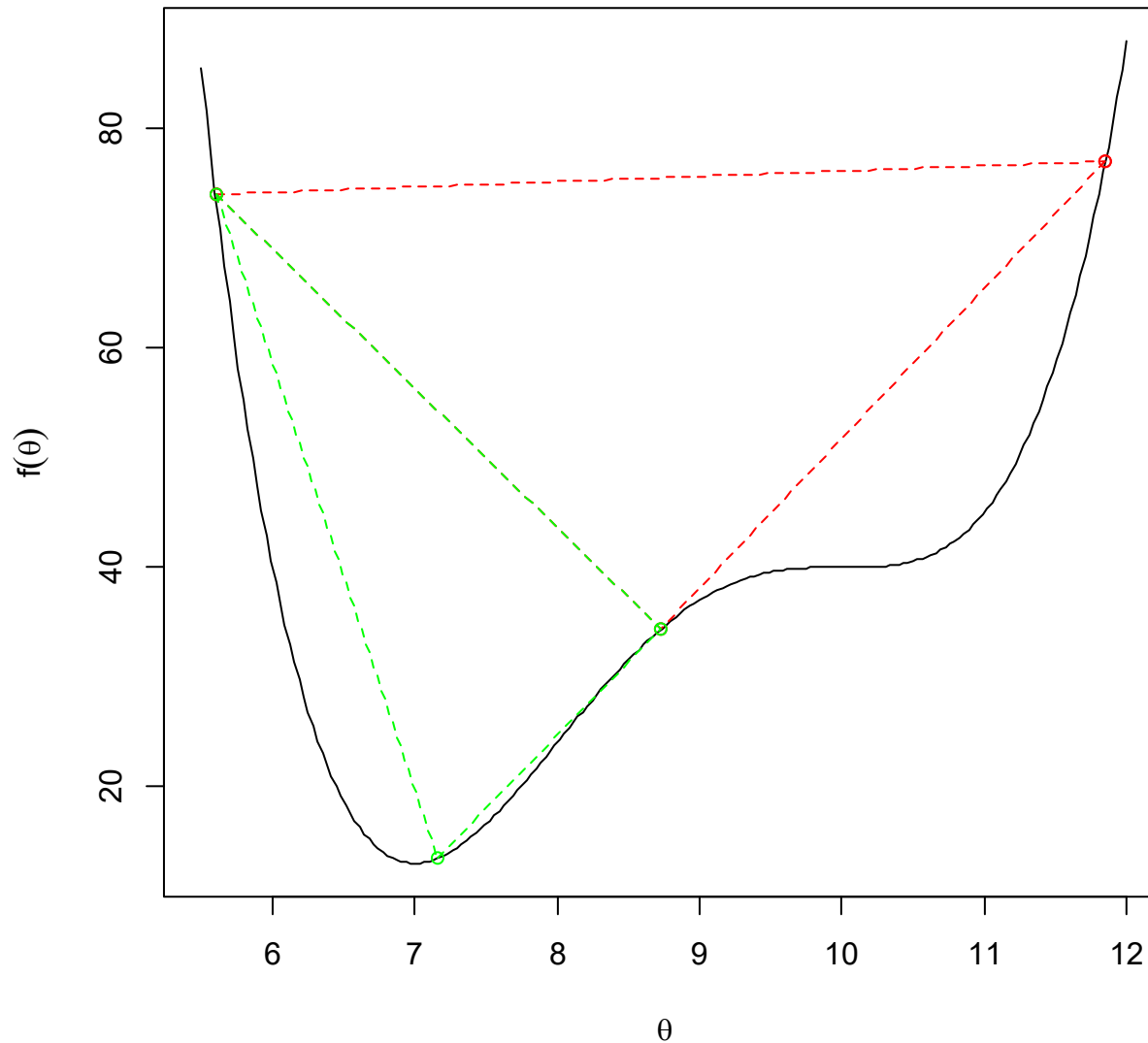
Make triangle

Bisect lower sides

Make lowest triangle

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

Bisect

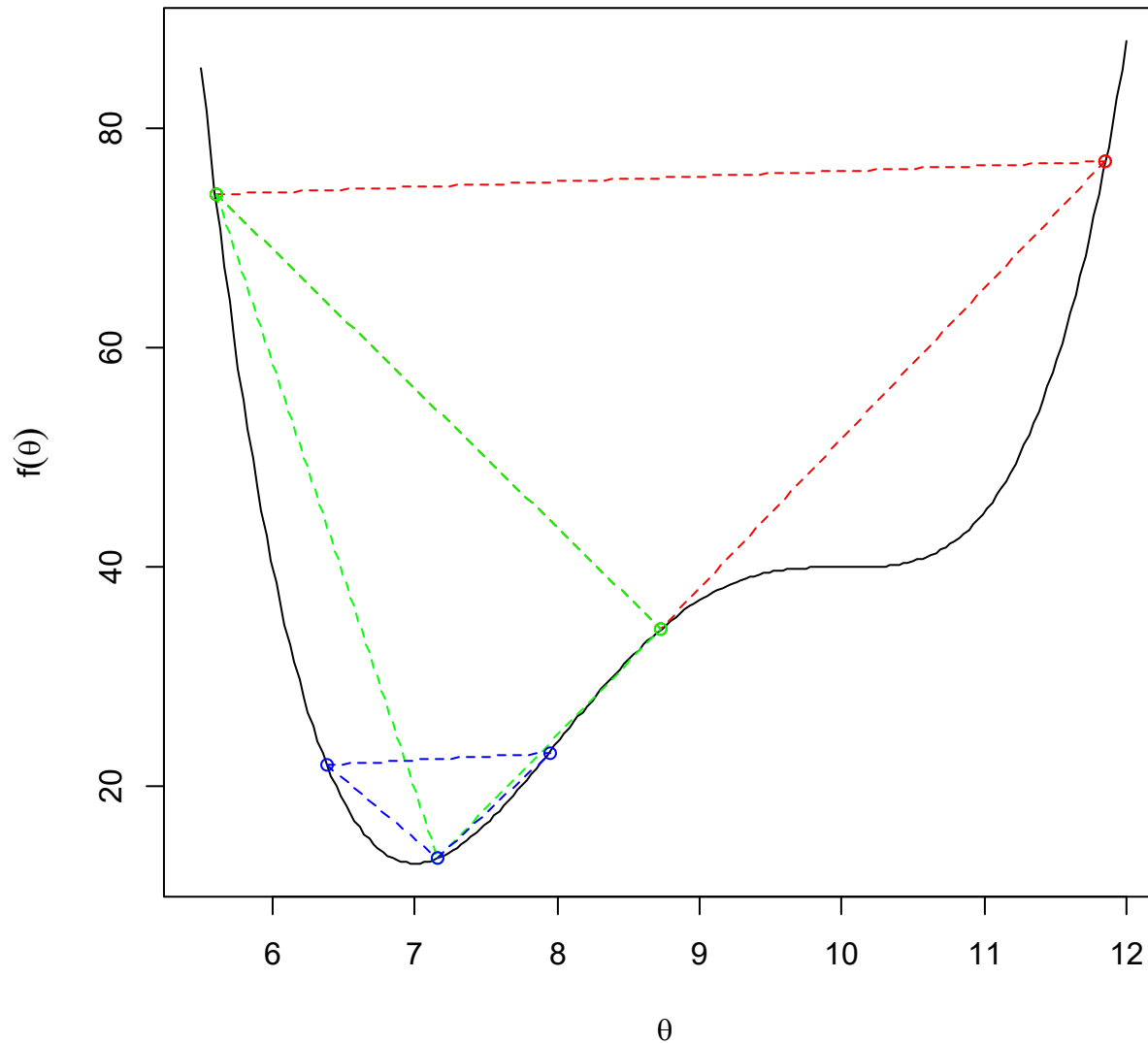
Make triangle

Bisect lower sides

Make lowest triangle

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum

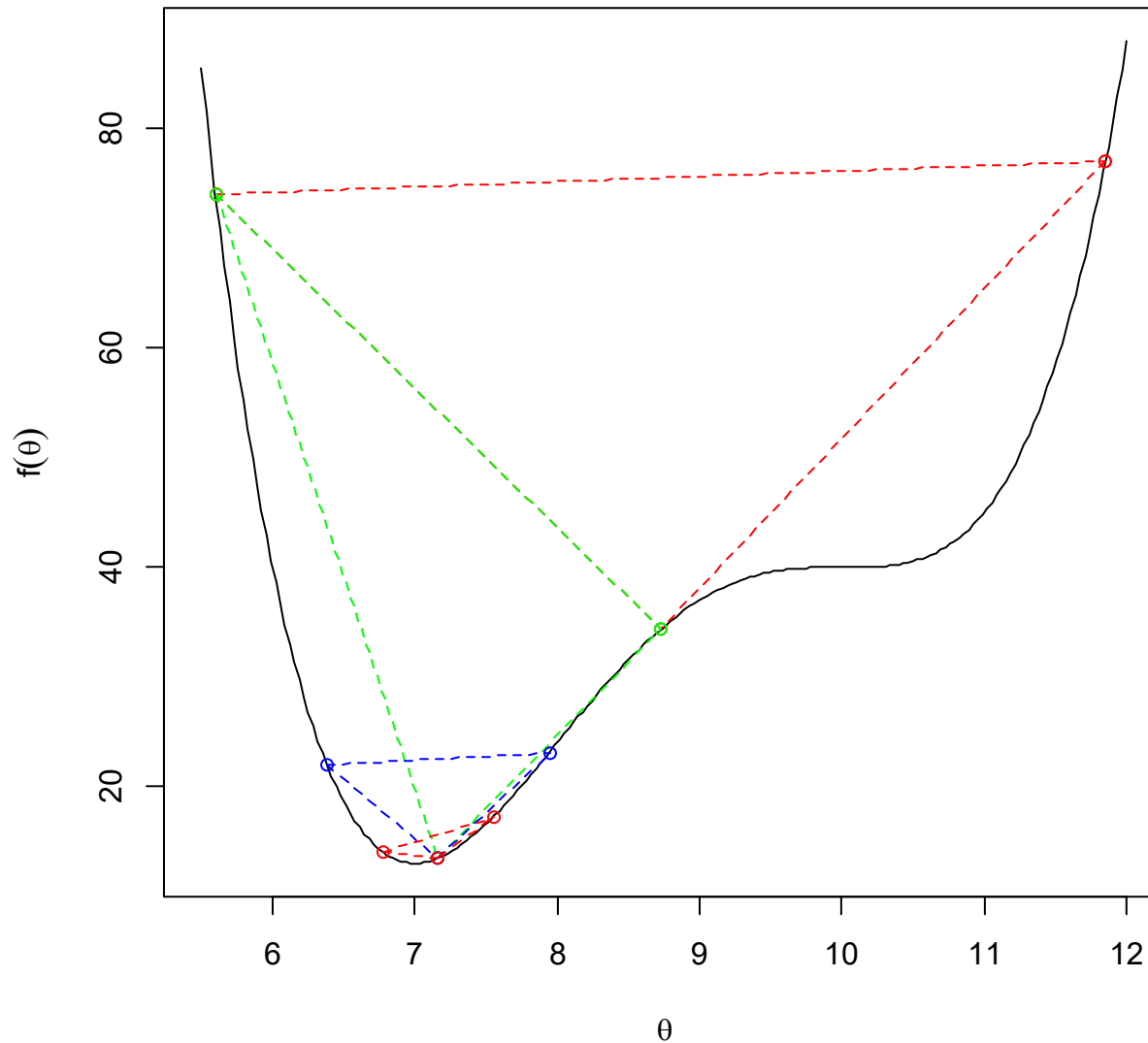


**Bisection
algorithm**

Start with 2 points
Bisect
Make triangle
Bisect lower sides
Make lowest triangle
Keep repeating

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

Bisect

Make triangle

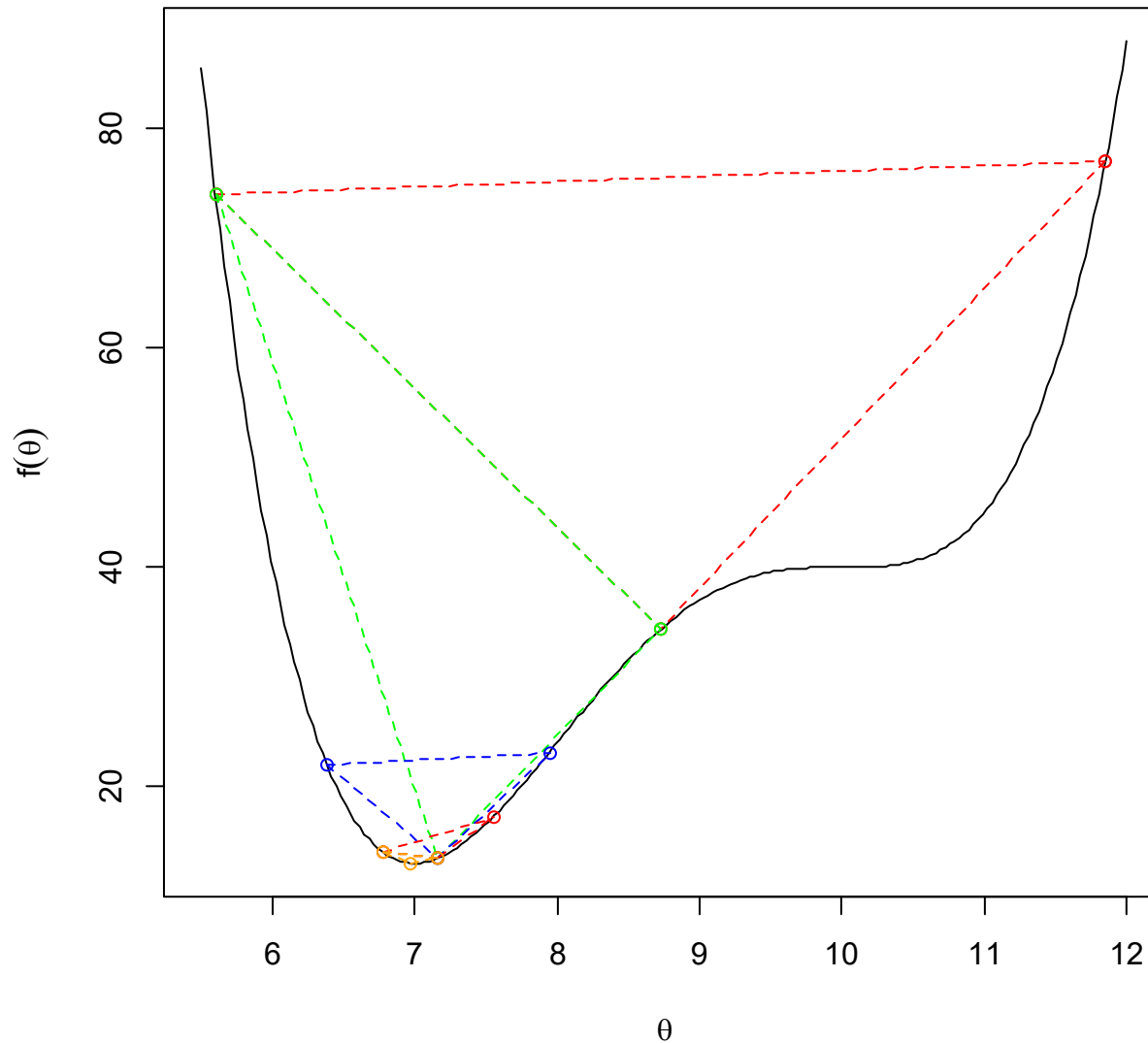
Bisect lower sides

Make lowest triangle

Keep repeating

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

Bisect

Make triangle

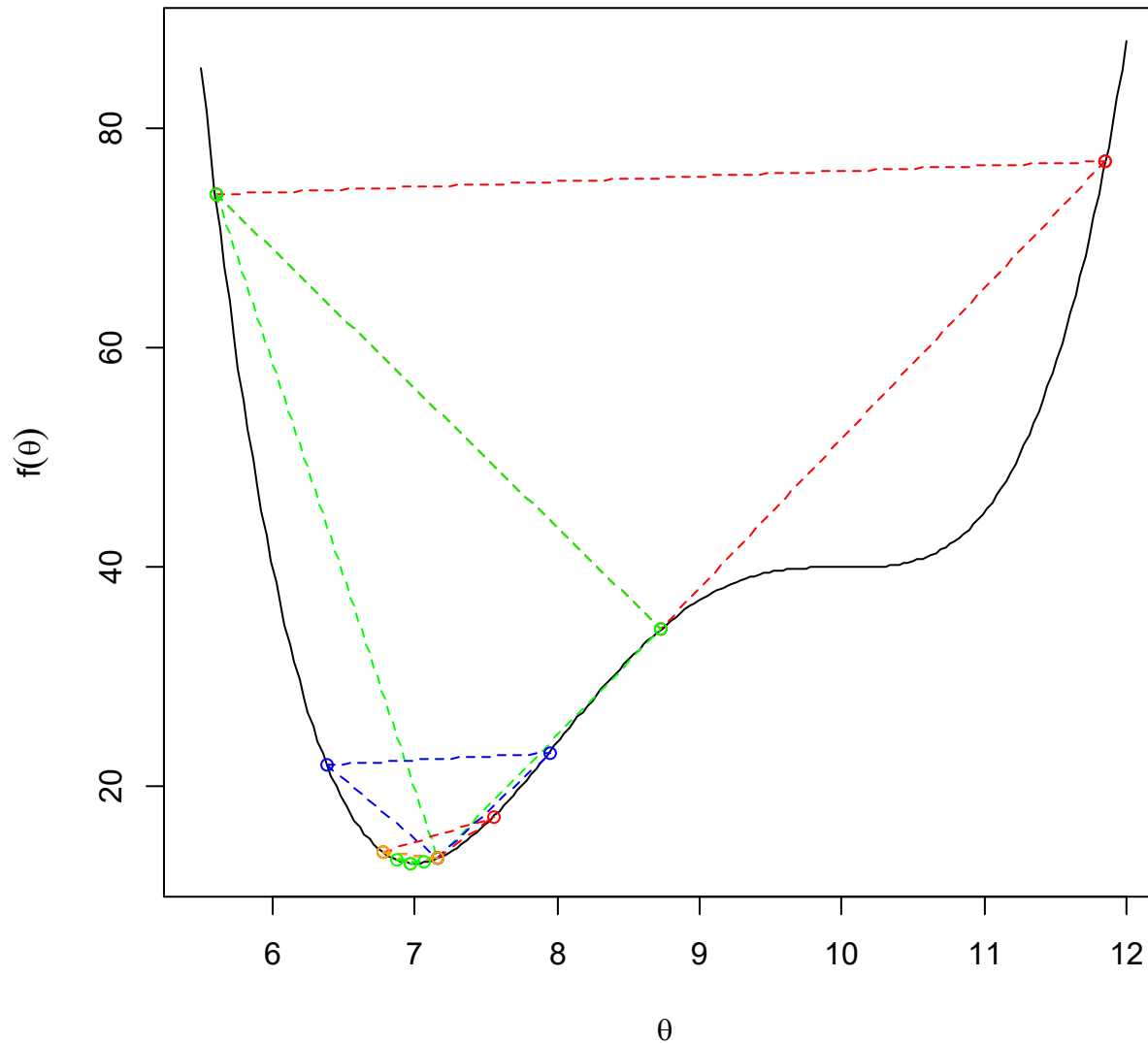
Bisect lower sides

Make lowest triangle

Keep repeating

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

Bisect

Make triangle

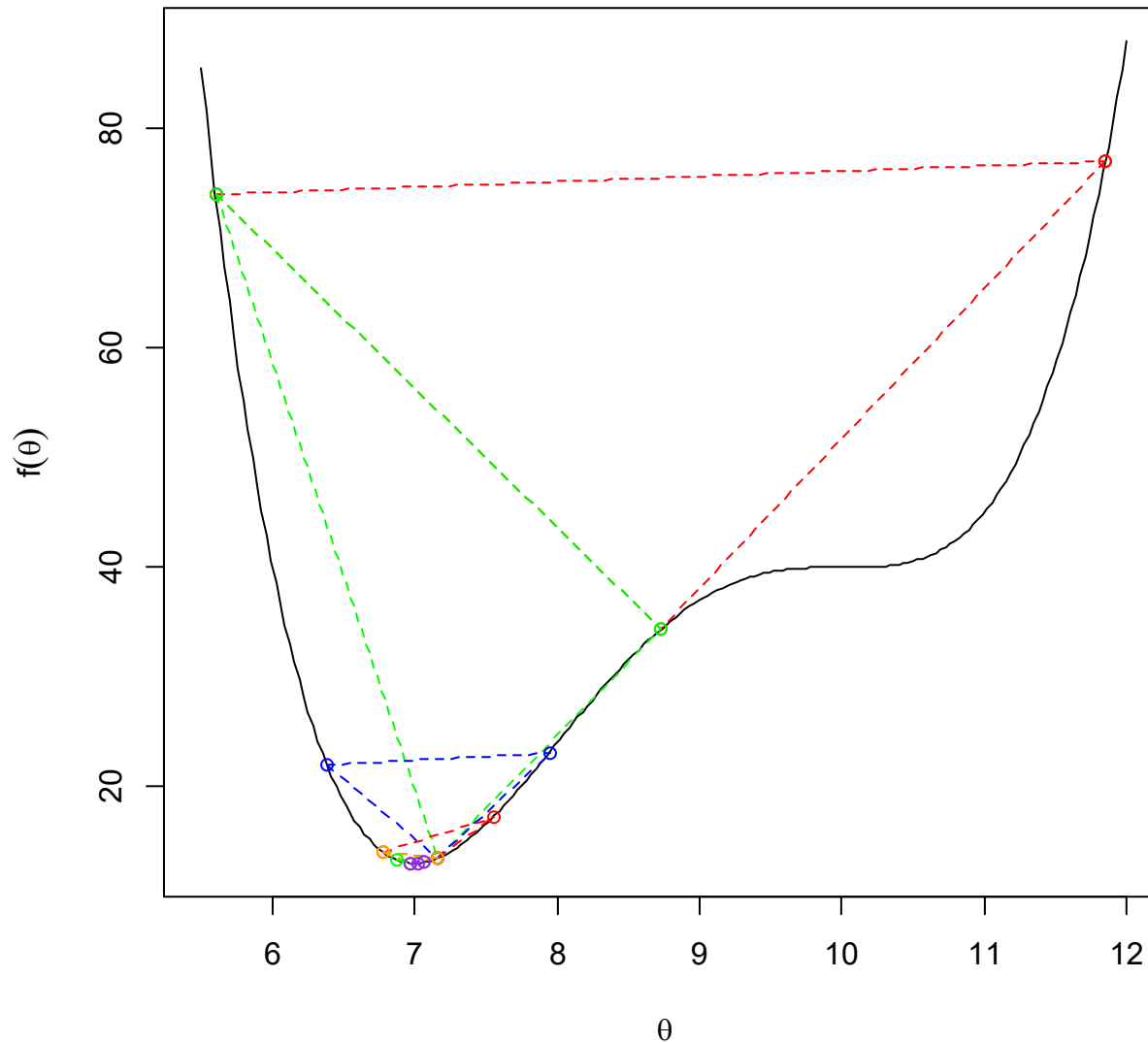
Bisect lower sides

Make lowest triangle

Keep repeating

A feel for descent algorithms

Optimize θ : find θ such that $f(\theta)$ is minimum



**Bisection
algorithm**

Start with 2 points

Bisect

Make triangle

Bisect lower sides

Make lowest triangle

Keep repeating

Roll your own

Almost always we will be using whatever optimization algorithm is implemented within the R function that trains the model, e.g. within `lm()` or `smooth.spline()`. Usually the function authors will have made an excellent choice for that particular model!

If you need to roll your own, an excellent starting place is the versatile [Nelder-Mead simplex algorithm](#), a descent algorithm, and the default in the R function `optim()`.

Tuning parameter

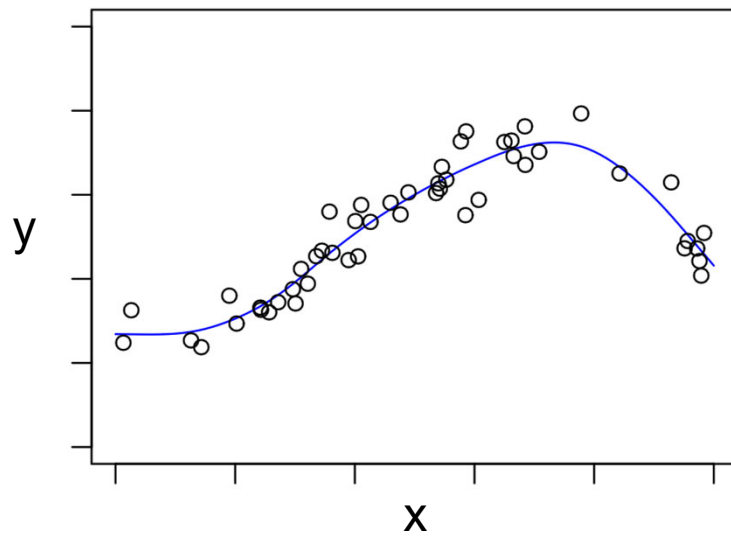
- Smoothing spline
- df = degrees of freedom
- df is a function of λ

Code

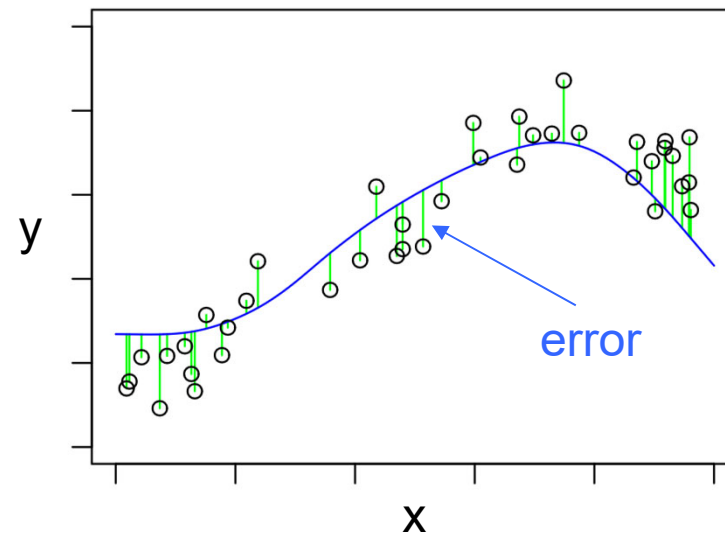
- `02_2_ants_cv_smooth.R`

Bias-variance tradeoff

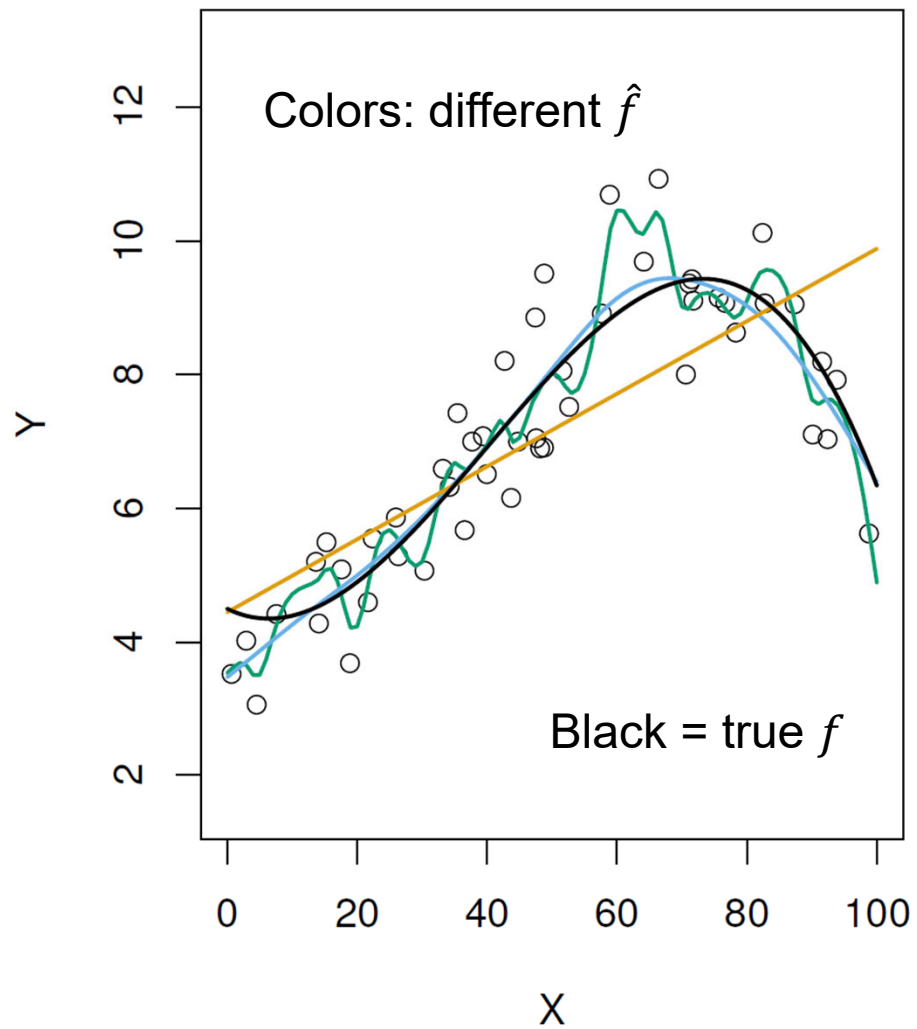
\hat{f} fitted on training data



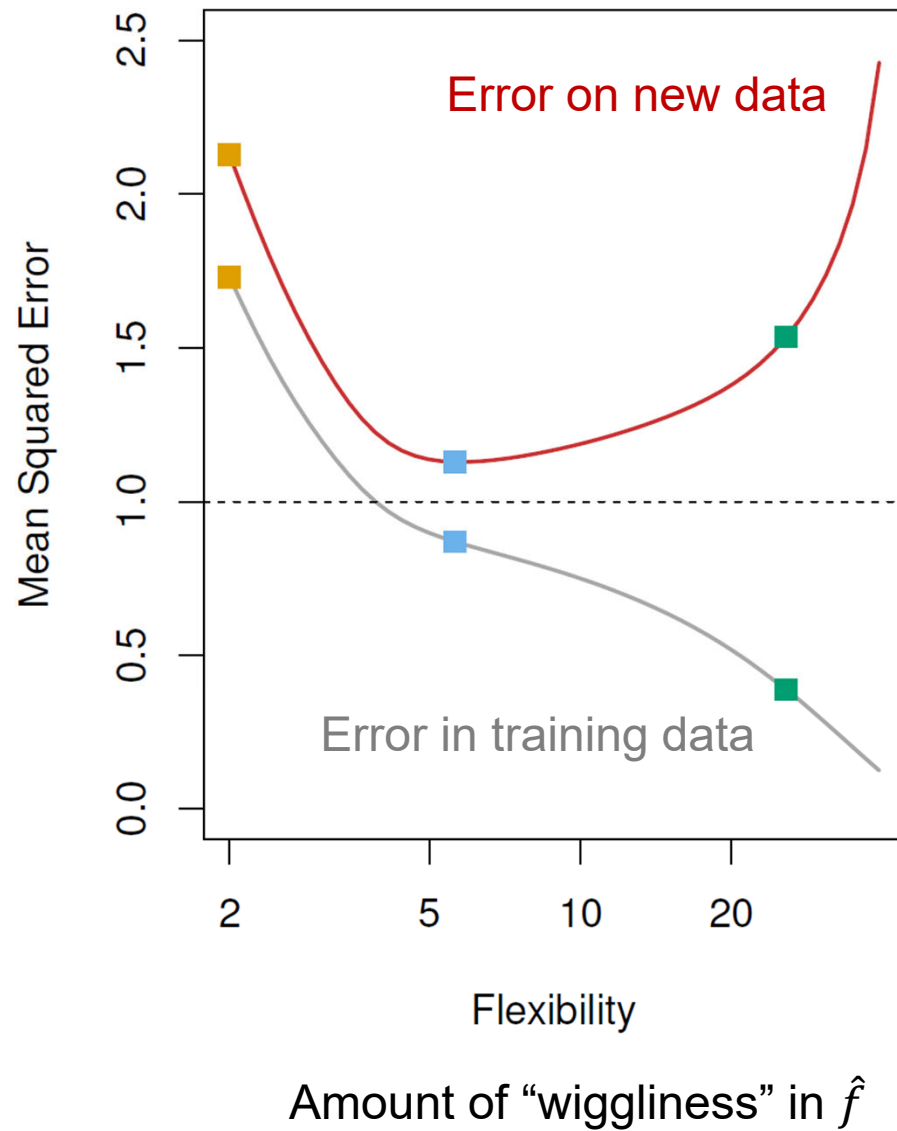
\hat{f} predicting new data

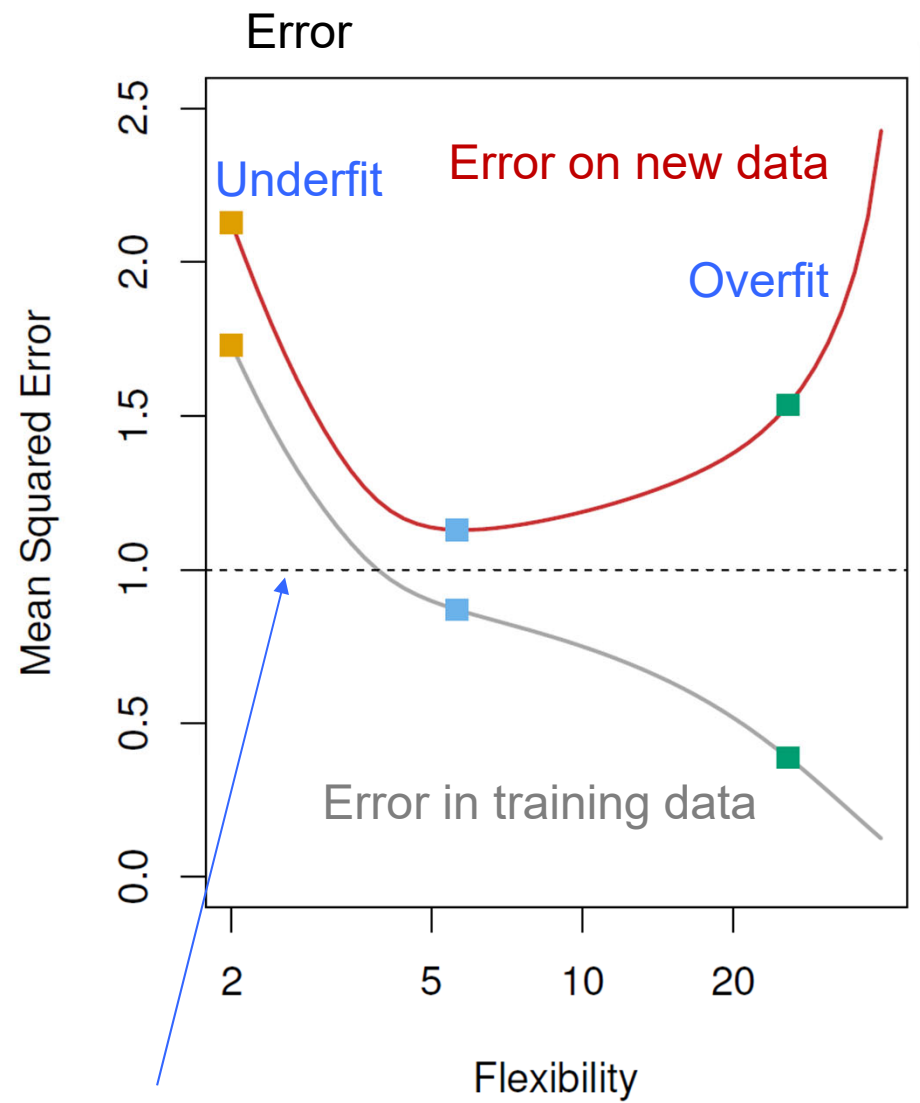
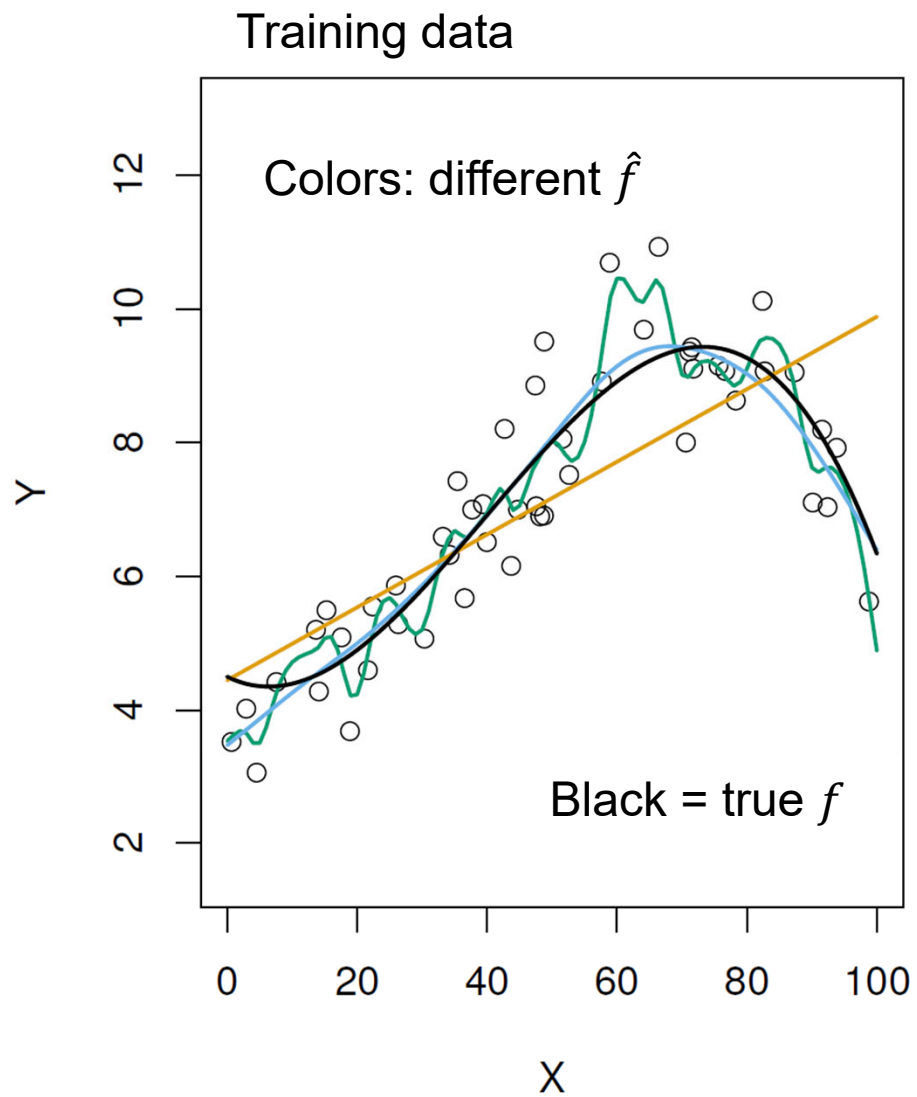


Training data



Error



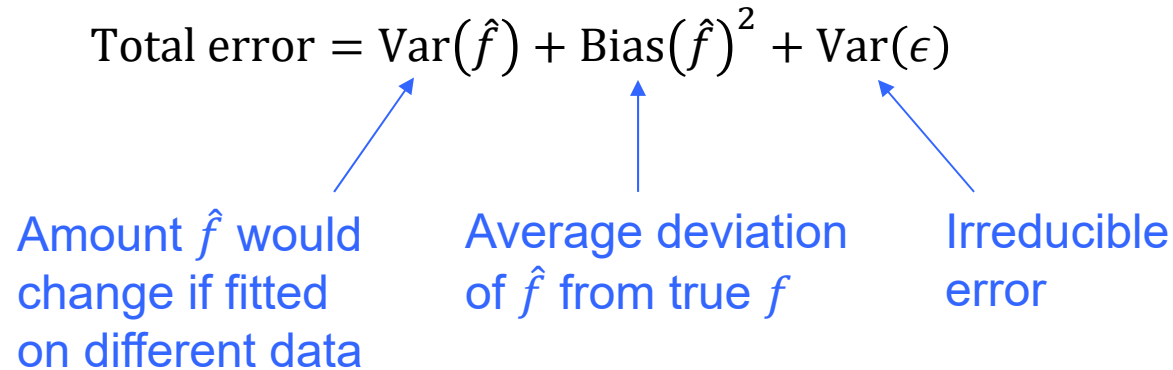


Goal: balance underfit and overfit

Bias-variance tradeoff

$$\text{Total error} = \text{Var}(\hat{f}) + \text{Bias}(\hat{f})^2 + \text{Var}(\epsilon)$$

Amount \hat{f} would
change if fitted
on different data



The diagram illustrates the bias-variance tradeoff equation. At the top, the equation is written: Total error = Var(f-hat) + Bias(f-hat)^2 + Var(epsilon). Below the equation, three blue arrows point upwards to the terms. The first arrow points from the text 'Amount f-hat would change if fitted on different data' to the Var(f-hat) term. The second arrow points from the text 'Average deviation of f-hat from true f' to the Bias(f-hat)^2 term. The third arrow points from the text 'Irreducible error' to the Var(epsilon) term.

Average deviation
of \hat{f} from true f

Irreducible
error

Bias-variance tradeoff

$$\text{Total error} = \text{Var}(\hat{f}) + \text{Bias}(\hat{f})^2 + \text{Var}(\epsilon)$$

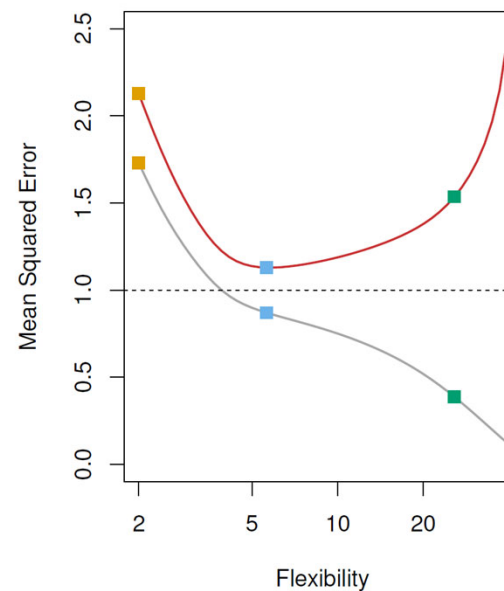
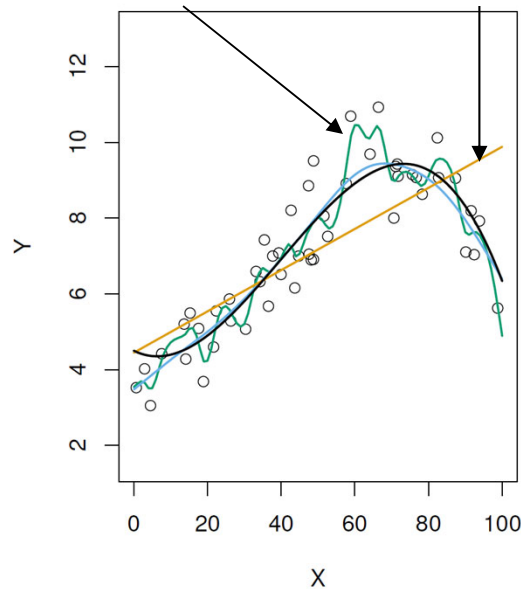
Amount \hat{f} would
change if fitted
on different data

Average deviation
of \hat{f} from true f

Irreducible
error

High variance

High bias



Bias-variance tradeoff

$$\text{Total error} = \text{Var}(\hat{f}) + \text{Bias}(\hat{f})^2 + \text{Var}(\epsilon)$$

Amount \hat{f} would
change if fitted
on different data

Average deviation
of \hat{f} from true f

Irreducible
error

High variance

High bias

High bias

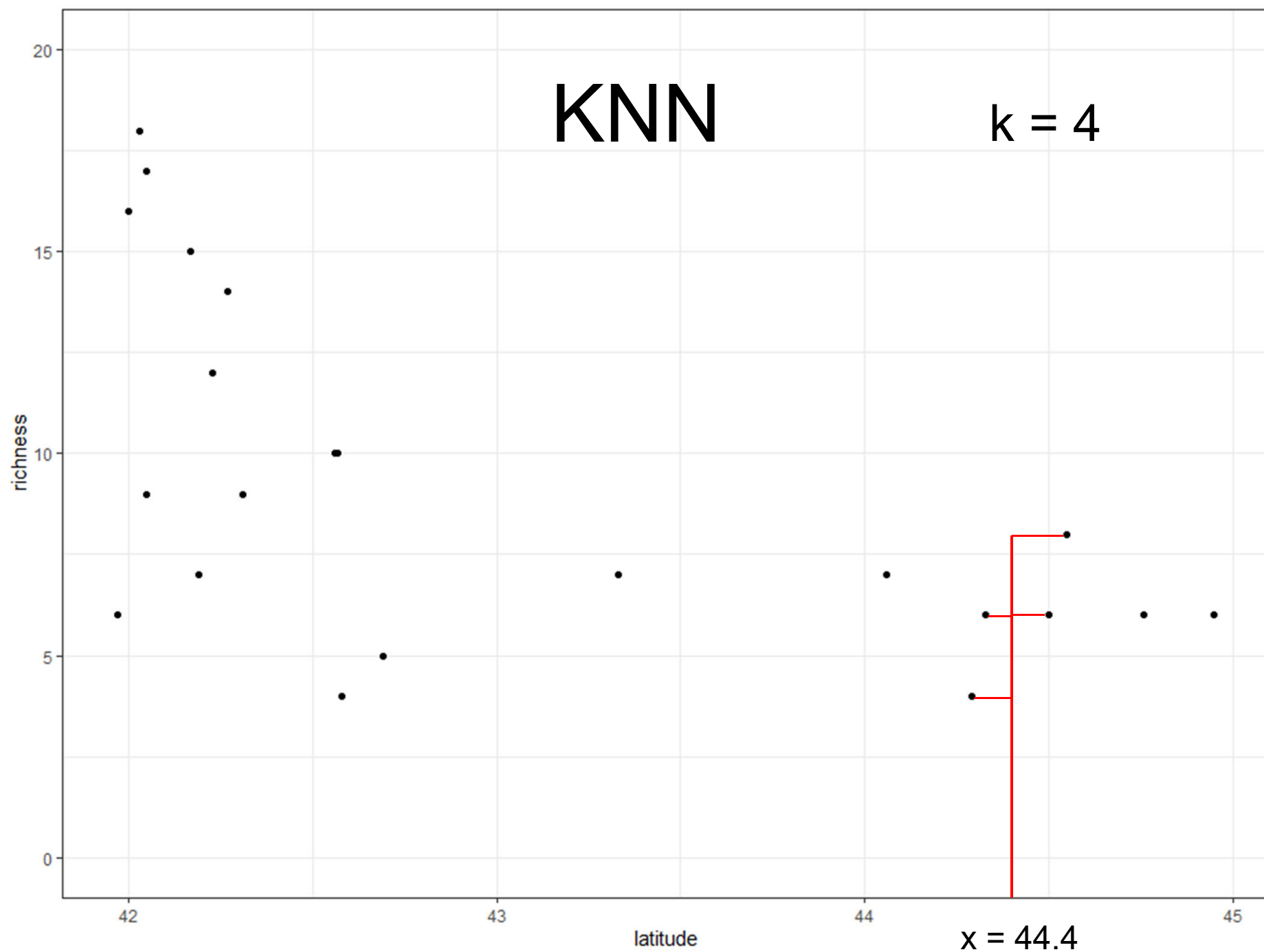
← High variance

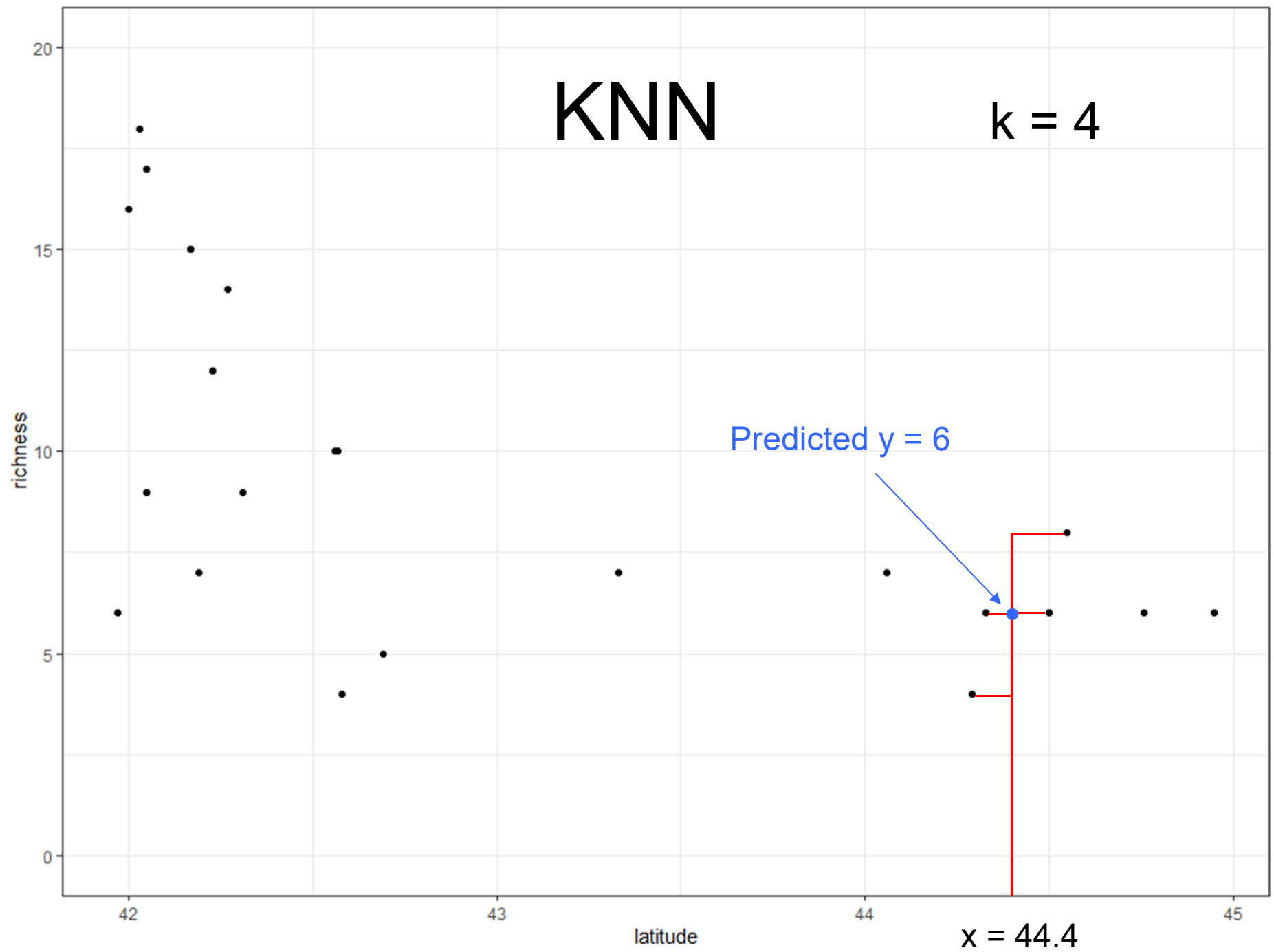
Amount \hat{f} would
change if fitted
on different data

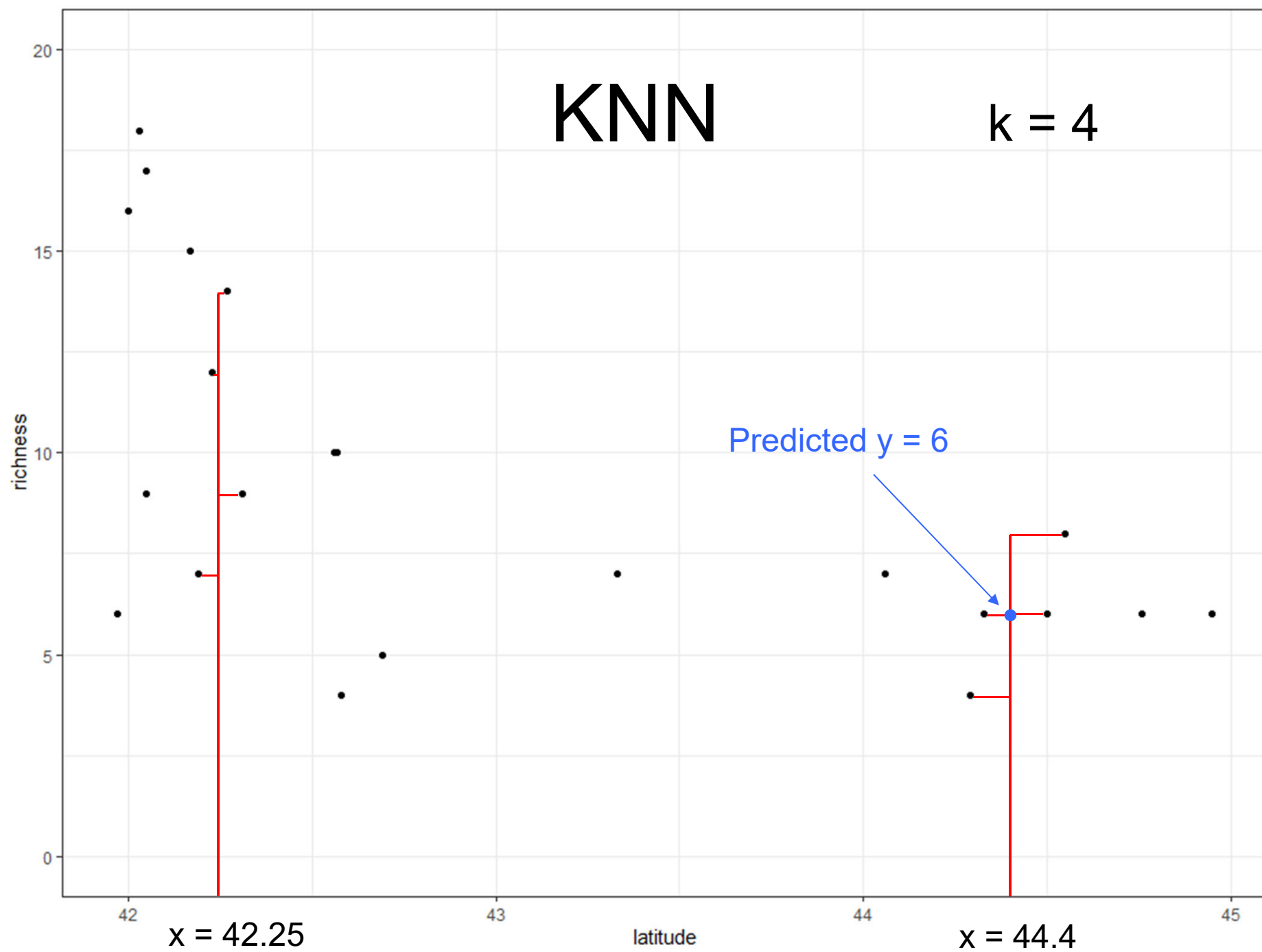
Sweet spot:
low variance
low bias

KNN model algorithm

- k nearest neighbors
- e.g of a more “algorithmic” model algorithm
- no parameters to be trained
- one tuning parameter

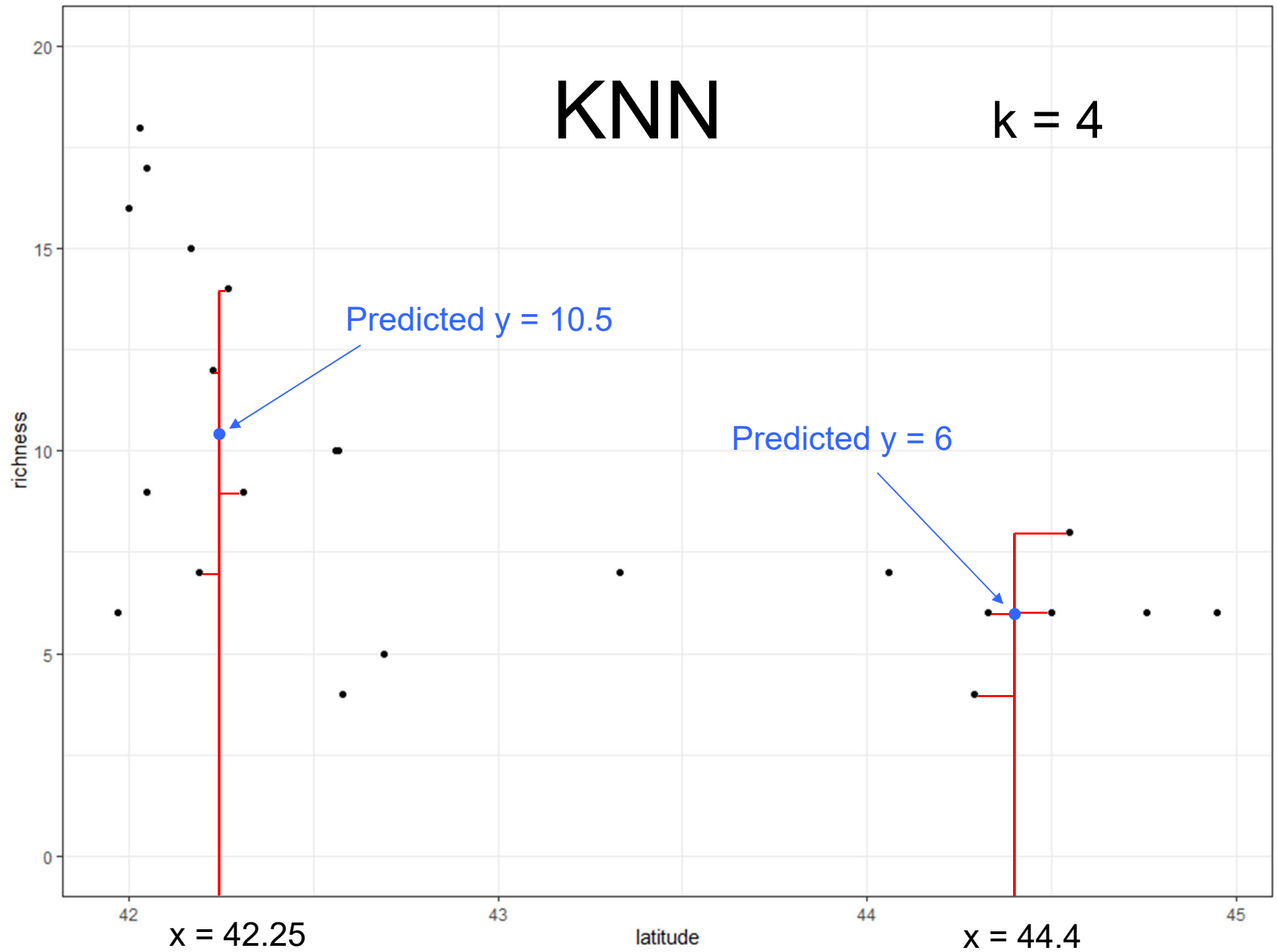


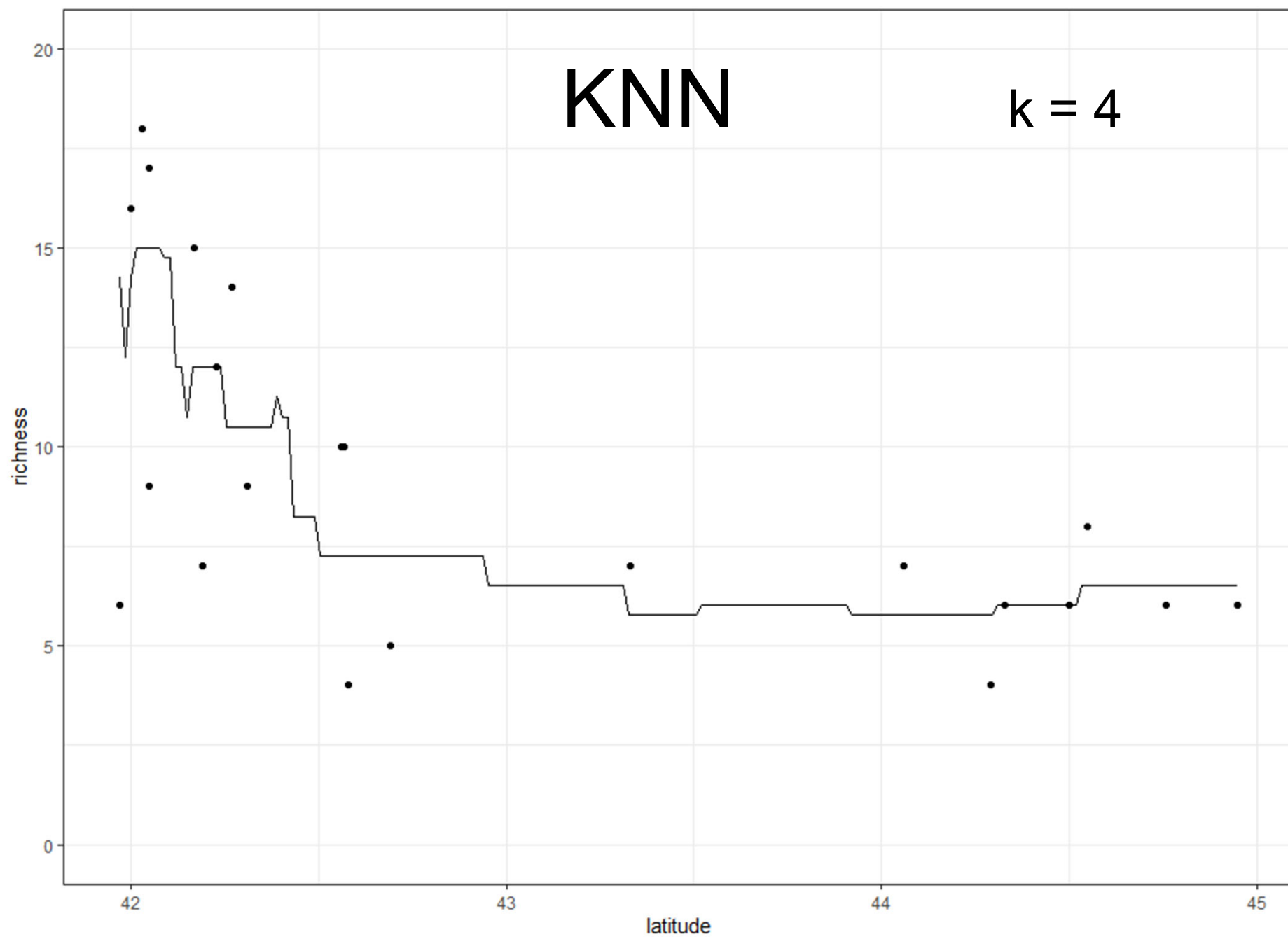


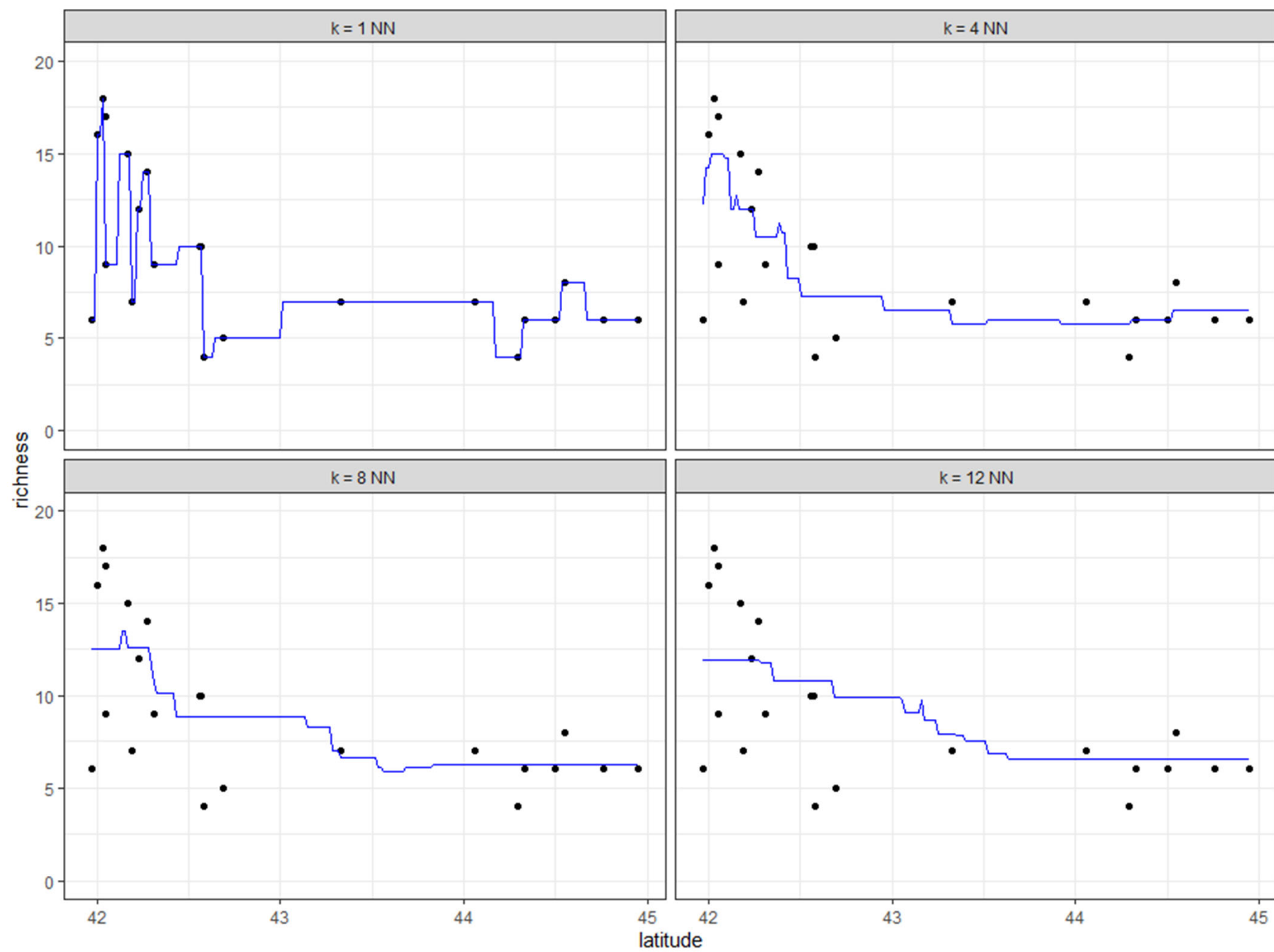


KNN

$k = 4$







KNN training algorithm

- There is no training algorithm!
- Automatic
- No parameters

KNN

Algorithm

Set k = number of nearest neighbors

Input (x, y) = x, y data pairs

Input x_{new} = x value at which to predict y_{new}

Calculate d = distance of x_{new} to other x

Sort y data ascending by d ; break ties randomly

Predict new y = mean of k nearest neighbors;
i.e. mean of first k values in y_{sort}

Code

- `ants_cv_knn.R`
- k-fold CV for KNN models with different numbers of nearest neighbors