Semi- and non-parametric time series models

FISH 507 – Applied Time Series Analysis

Eric Ward

26 Feb 2019

Overview of today's material

- Gaussian process models
- Neural network models
- Empirical dynamic modeling

Gaussian processes for time series

Last week, we discussed exponential smoothing and in lab touched on GAMs

- Both approaches are similar in that they borrow information from neighbors
- Exponential smoothing usually borrows information from past data for forecasting
- Generalized additive models (GAMs) usually borrow information from both future and past data

Gaussian processes for time series

GAMs estimate the *trend* using a smooth function,

$$E[Y] = B_0 + f(x)$$

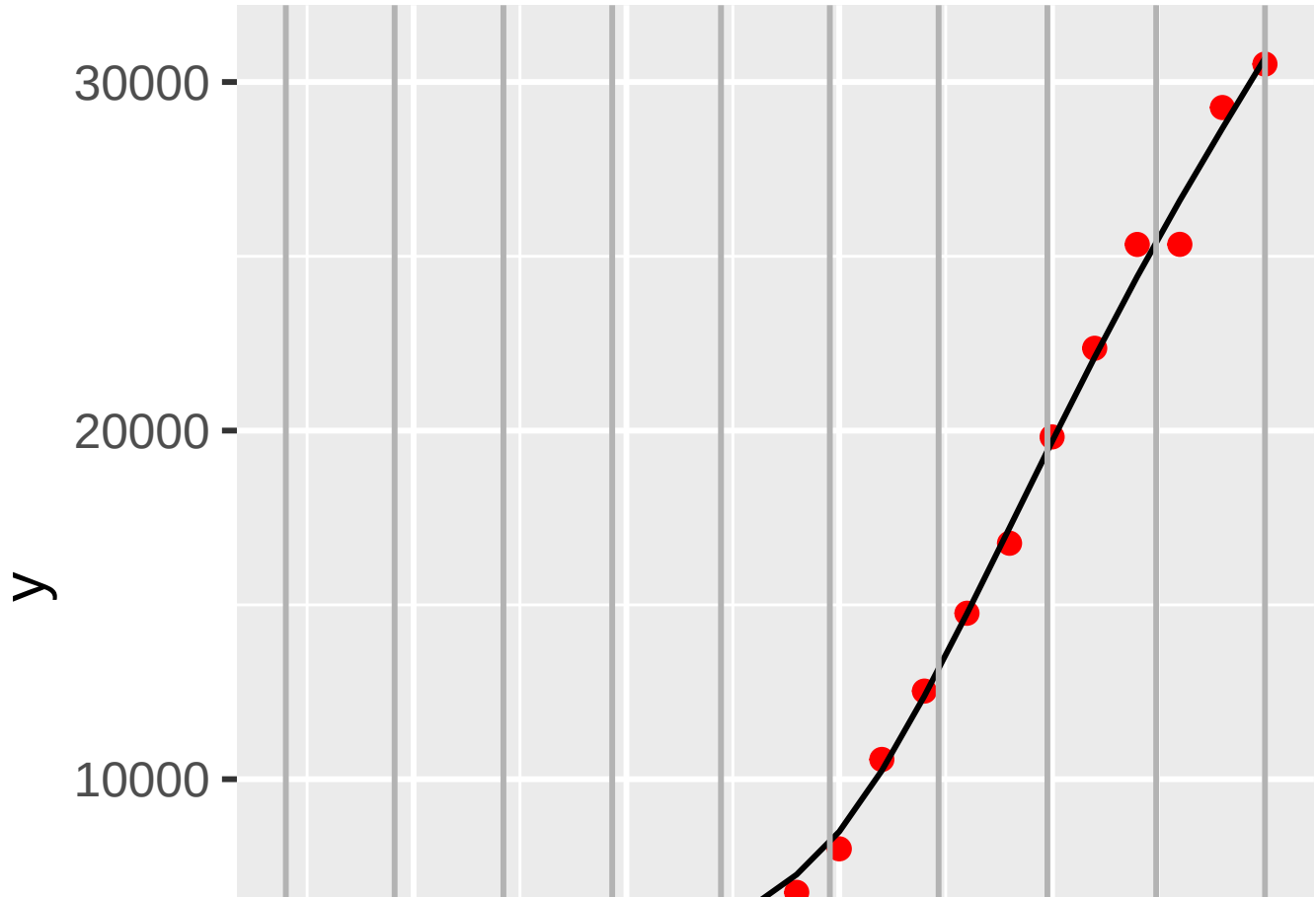
where like regression, we assume $Y \sim Normal(E[Y], \sigma)$

- The smooth function approximates the trend at a smaller subset of locations (aka *knots*)
- The density and location of the knots can affect how 'wiggly' the function is

Gaussian processes for time series

For data that are regularly spaced in time, this probably isn't a big deal

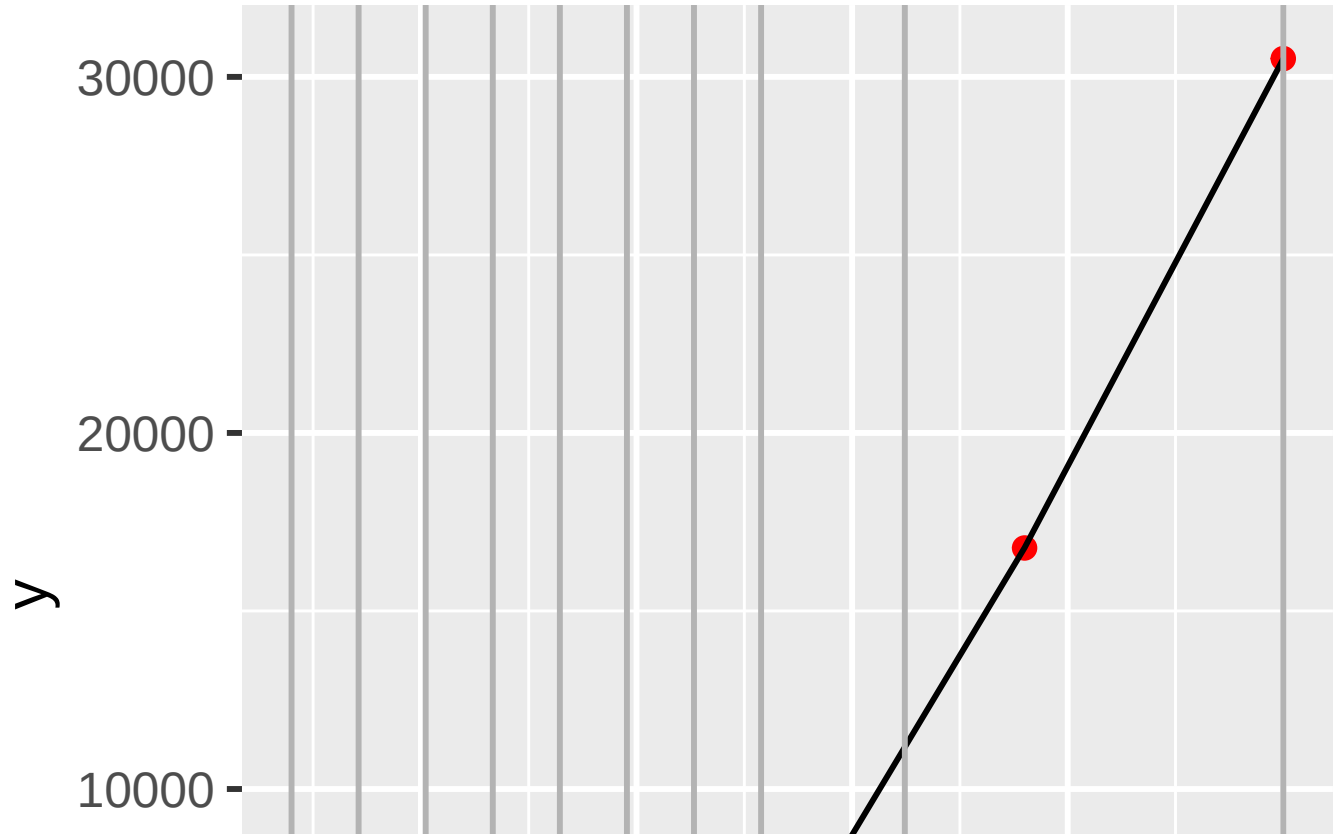
- For instance if we use a cubic spline (default) on the 'airmiles' dataset (n = 23), a function approximating the trend is estimated at 10 equally spaced locations (grey vertical lines).



Gaussian processes for time series

Let's try again, this time knocking a few holes in the data. Removing years 1950:1953 and 1955:1959, the knot locations are no longer equally spaced, and weighted more toward the locations of data points.

- Greater spacing between knots = less flexibility, more uncertainty (you can look at the 'se.fit' part of predict output)



Gaussian processes for time series

Recapping, GAMs are estimating the underlying *trend* using a smooth function,

$$E[Y] = B_0 + f(x)$$

- It's important to note that this underlying trend function $f(x)$ is modeling the **mean**
- Smooths are very flexible (with respect to # knots, locations, smooth type). See 'mgcv' and 'gamm'

We're going to leave GAMs alone for now, but there's lots of great references out there. Examples:

- Gavin Simpson's work with GAMs and time series here
- Simon Wood's book

Gaussian processes for time series

Similarities between GAMs and GP models:

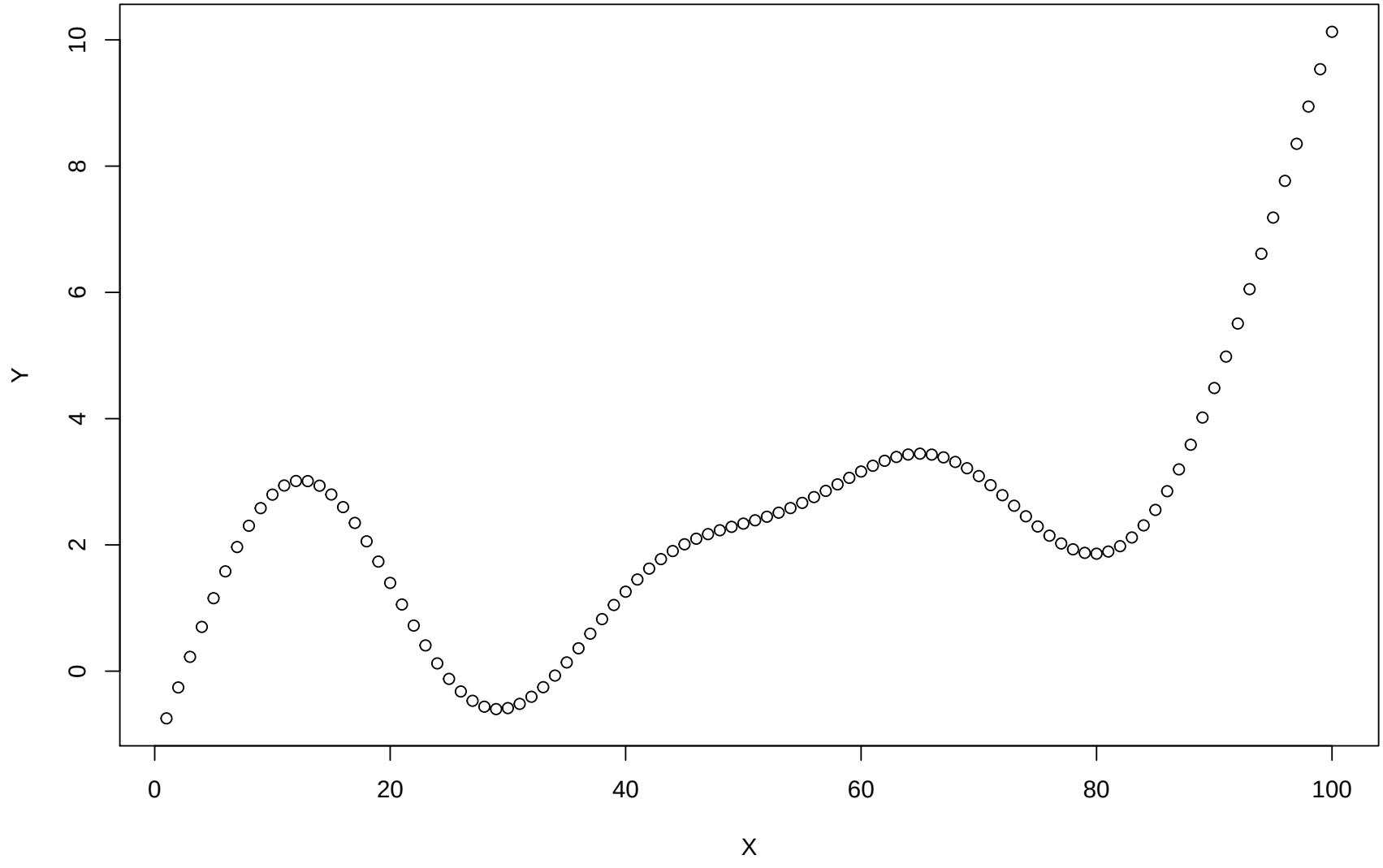
- GAMs and GP predictive models use reduced dimensionality (knots) to constrain flexibility

Differences:

- GAMs use smooth functions & knot locations to constrain how neighbors affect mean
- GP models use covariance function to control how much neighbors can influence eachother based on how far apart they are

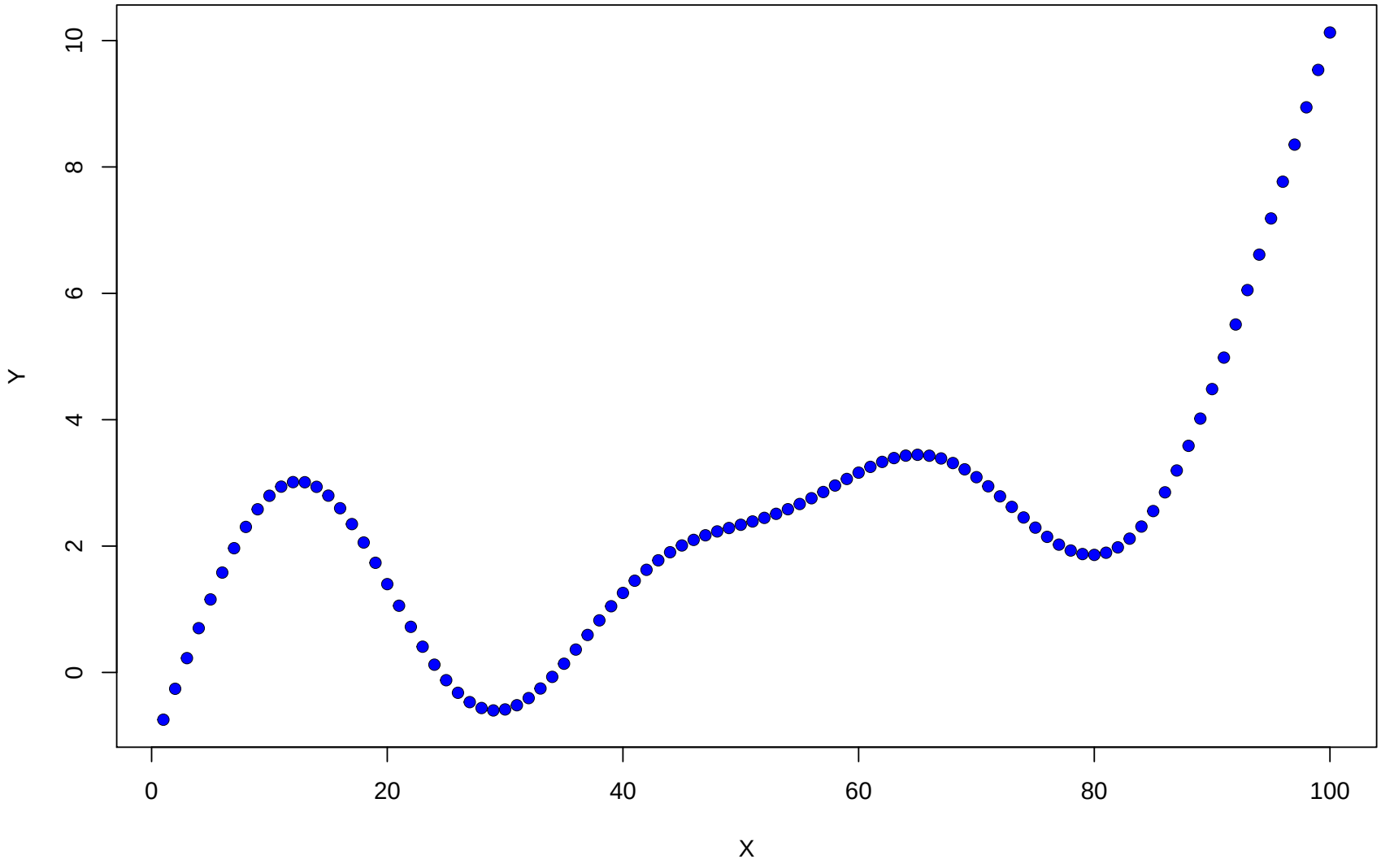
Gaussian processes for time series

We have some function we want to approximate



Gaussian processes for time series

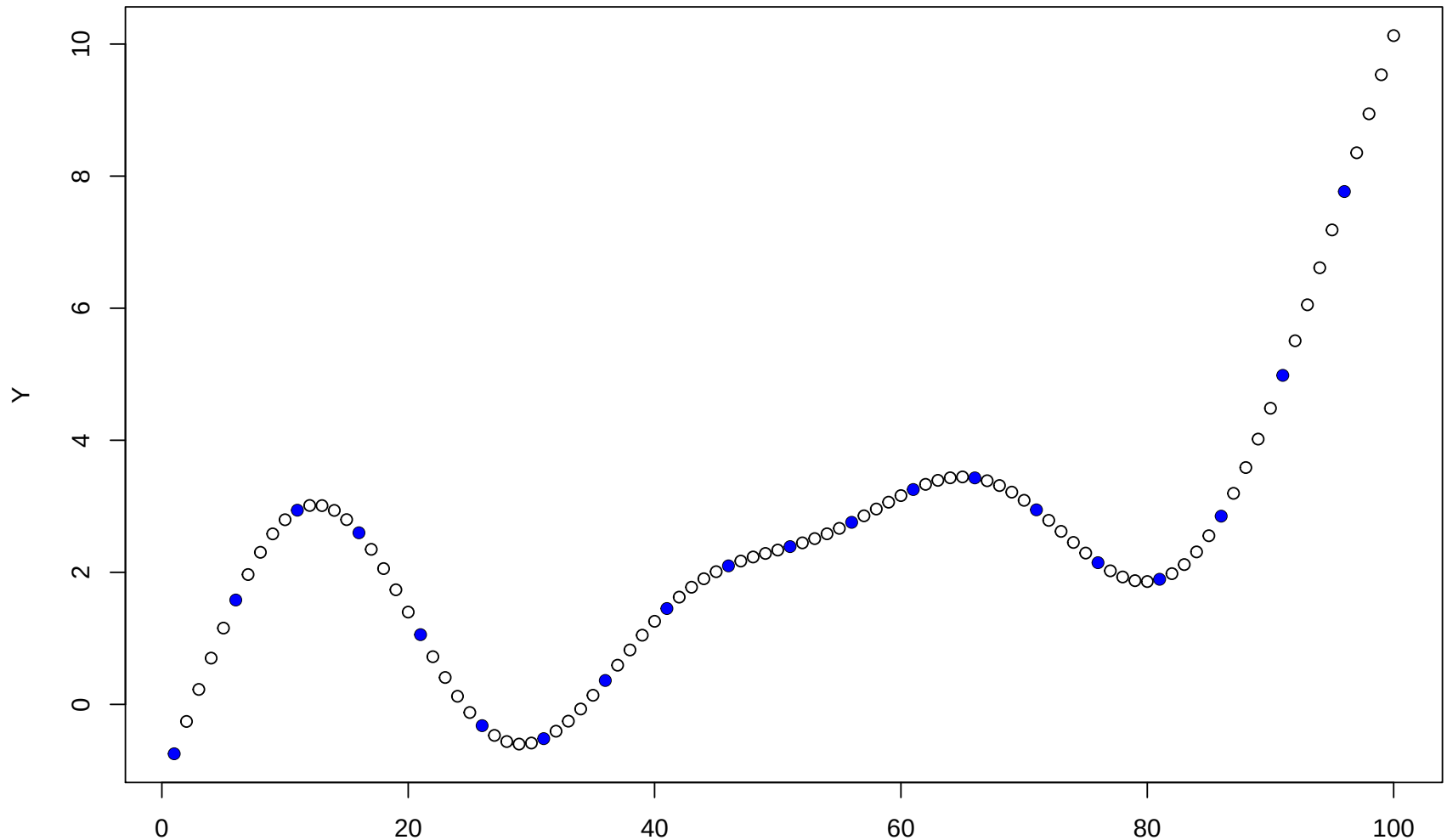
We could estimate the latent values at all observed locations *
What are the downsides to this?



Gaussian processes for time series

Instead, consider estimating them at a subset of points and extrapolating (aka Kriging)

- these locations are called the *knots*
- extrapolating to other locations = *predictive process model*



Gaussian processes for time series

Lots of applications in Fisheries and Ecology

- Munch et al. 2005 link
- Munch et al. 2018 link

Especially with applications to spatial models

- Latimer et al. 2009 link
- Finley et al. 2017 link
- Gelfand et al. 2018 link
- Anderson et al. 2018 link
- Shelton et al. 2014 link
- Ward et al. 2018 link

Gaussian processes for time series

Several options for estimating $f(x)$ at knot locations

- Common choice is random effects

Gaussian Process models use the covariance function, Σ

- e.g. Assume the random effects are MV Normal, e.g. $w \sim MVNormal(u, \Sigma)$

Gaussian processes for time series

We could estimate elements of Σ as unconstrained matrix (e.g. 'unconstrained' in MARSS)

- but that's a lot of parameters! $\sim$ m*(m+1)/2

We could try to zero out some elements of Σ

- but this will cause problems: if x_1 and x_2 are correlated, and x_1 and x_3 are correlated, x_2 and x_3 have to be correlated too

Gaussian processes for time series

Instead, we'll use a covariance function (aka kernel). Common choices are

- Exponential
- Squared-exponential (Gaussian)
- Matern
- Anisotropic functions

Gaussian processes for time series

For example with the exponential function,

$$\Sigma_{i,j} = \sigma^2 exp\left(-d_{i,j}/\tau\right)$$

- σ^2 is the variance parameter (estimated)
- $d_{i,j}$ is the distance between points, e.g. $|x_i - x_j|$
- distance could be distance in time, space, etc
- τ is a scaling parameter (estimated)

Gaussian processes for time series

Question:

For our exponential function, how do σ and τ control 'wiggliness'?

Gaussian processes for time series

For our exponential function, how do σ and τ control 'wiggliness'?

- Larger values of σ introduce more variability between $f(x)$ at knot locations
- Larger values of τ will make the 'exp(...)' term closer to 1

Gaussian processes for time series

Revisiting univariate state space models, what are some reasons the AR process is used?

$$x_t = x_{t-1} + w_t, \quad w_t \sim N(0, q)$$

$$y_t = x_t + v_t, \quad v_t \sim N(0, r)$$

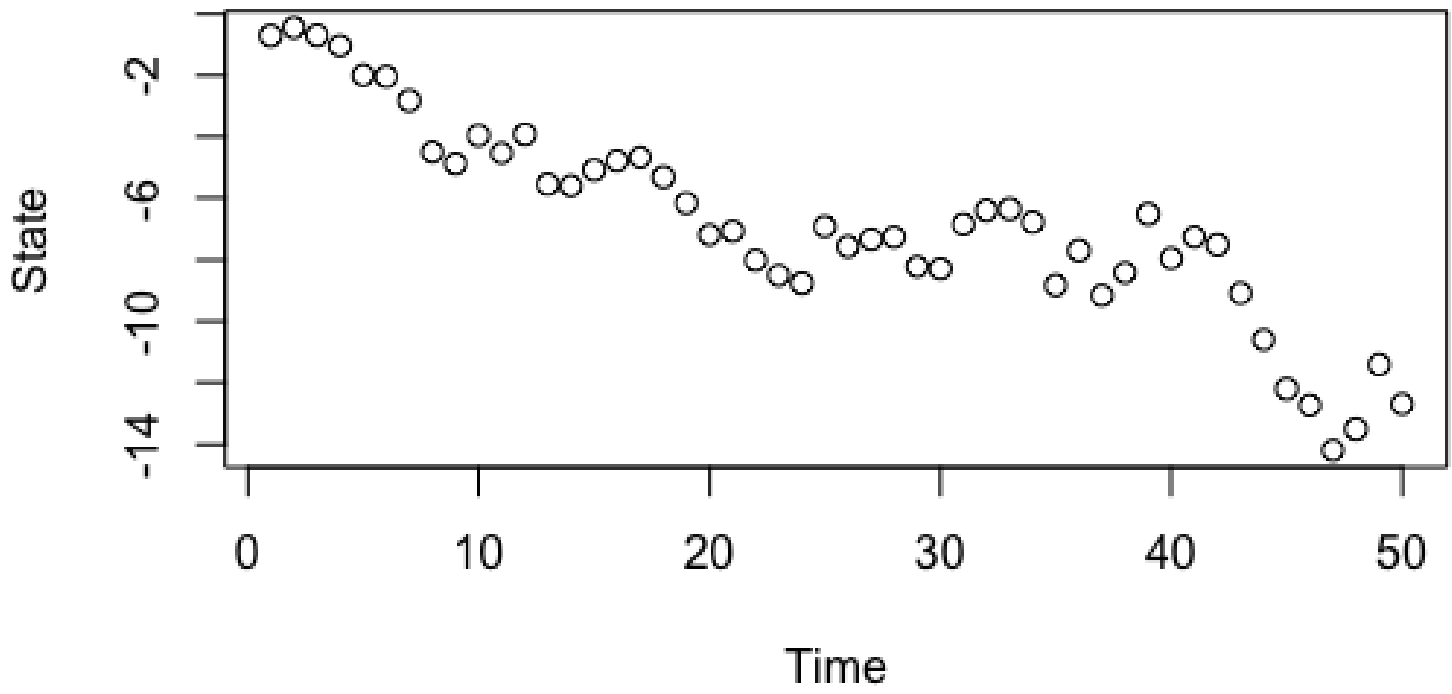
* Mechanism may be AR or RW **BUT** also * AR process is just one flavor of constraining estimation * Convenience / estimation of q and r

Gaussian processes for time series

Any of the univariate SS or multivariate models (DFA, MARSS) can be modified by swapping out an AR latent process for a GP one!

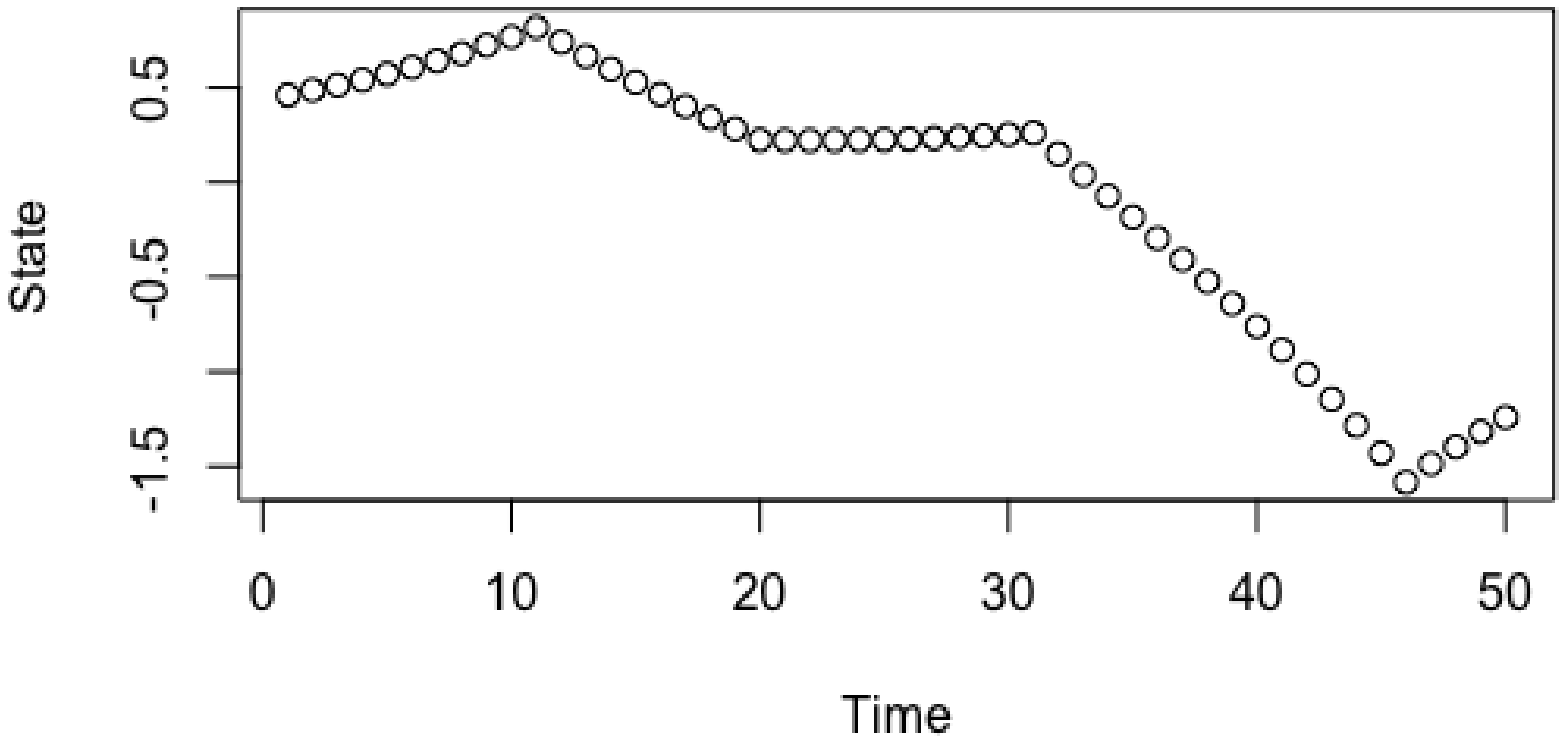
Example: Gaussian process DFA

- Simulated trend via AR process looks like this



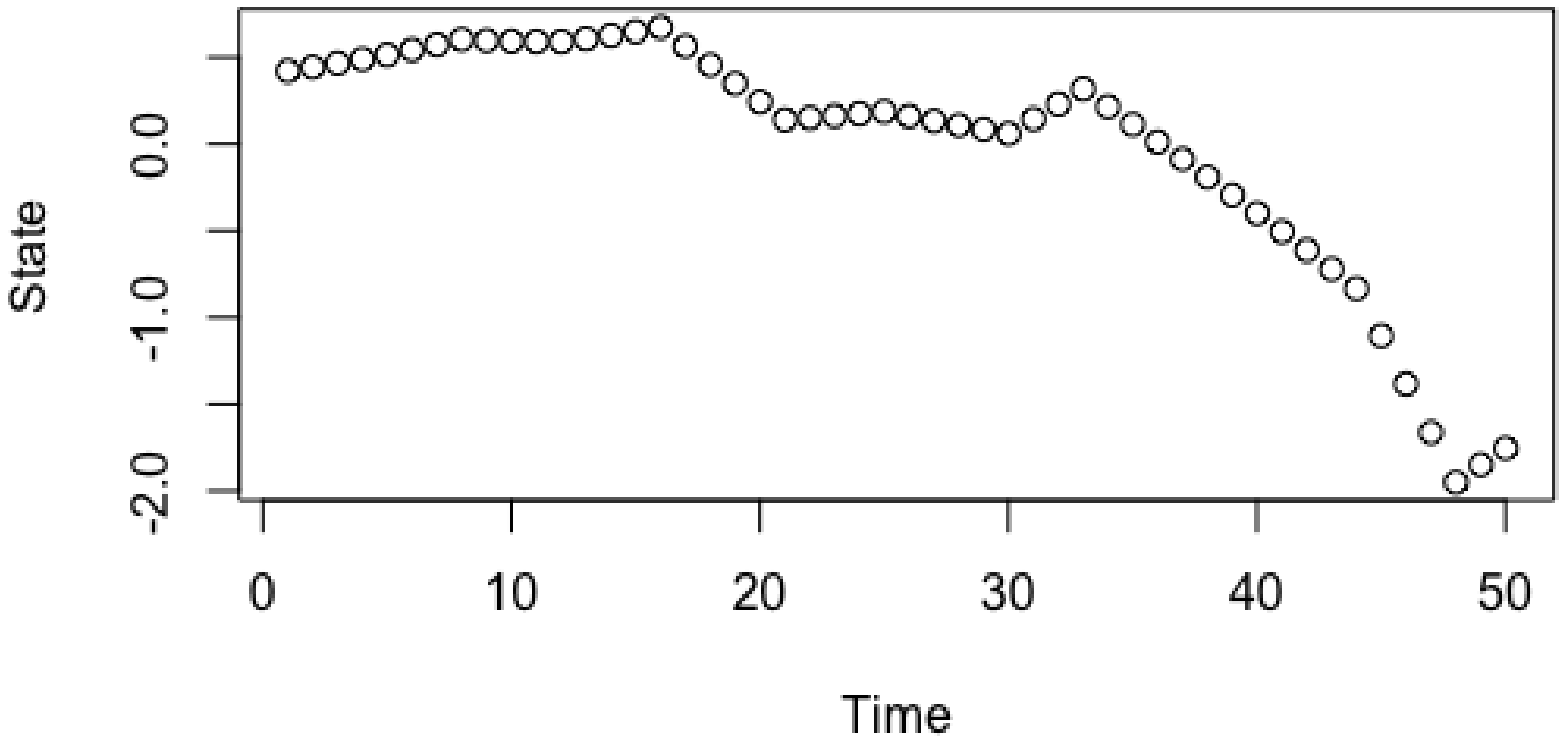
Gaussian processes for time series

Using a GP-DFA estimation model, we can see our ability to recover the process improve from 4 to 10 to 25 knots. 4 knots:



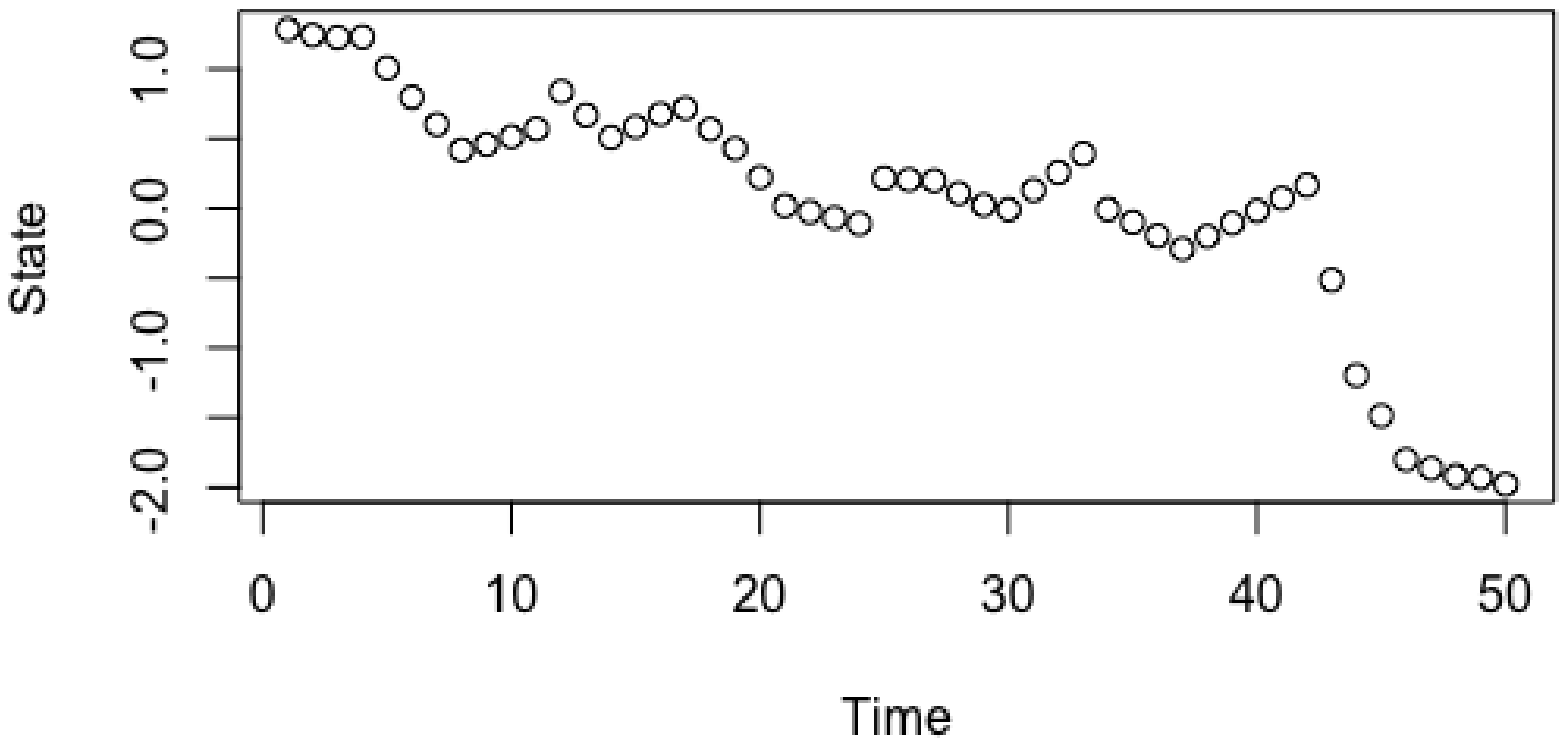
Gaussian processes for time series

Using a GP-DFA estimation model, we can see our ability to recover process improve from 4 to 10 to 25 knots. 10 knots:



Gaussian processes for time series

Using a GP-DFA estimation model, we can see our ability to recover the process improve from 4 to 10 to 25 knots. 25 knots:



Neural network time series models

Neural networks widely used in lots of fields. Not widely used in fisheries with a few examples:

Ward et al. 2014 link Coro et al. 2016 link

- Special applications to time series or data that are sequentially structured

Neural network time series models

Some NNet jargon:

- *Inputs* are predictors (including lagged data)
- *Hidden layer* are the latent variables / process
- *Neurons* control dimensionality of hidden layer (a collection of hidden neurons = hidden layer)
- *Output* is the predictions vailidated against observable data

Neural network time series models

Neural networks offer an advantage over many approaches we've seen in that they're non-linear

Example:

- We have a number of predictors for our time series. These are the inputs

X_1, X_2, X_3

- The neuron takes the inputs, and uses a function $f(...)$ to generate predictions. $f(...)$ is known as the *activation function* and is non-linear (sigmoid, exponential, etc)

Neural network time series models

Just like regression, the nueron estimates coefficients (aka *weights*) for each of the predictors.

$$E[Y] = f(b_0 + b_1 * X_1 + b_2 * X_2 + b_3 * X_3)$$

Note: b_0 is sometimes called the bias – but is similar to intercept in regression

Neural network time series models

Implementation in R

*We'll talk about examples in 2 packages

- *forecast*, *tsDyn*

Neural network time series models

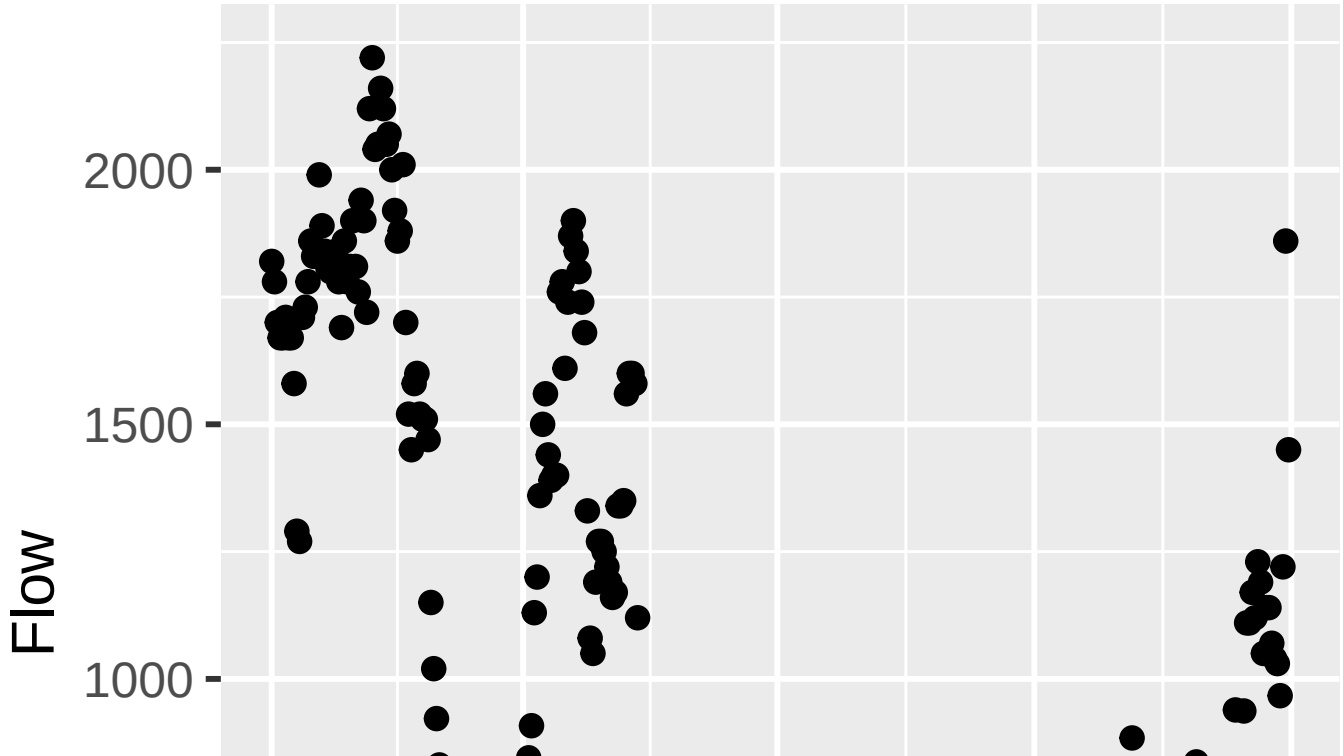
First the forecast package – function *nnetar*

- This package implements NNet models with *autoregression*, where this is defined as lagged values of the response time series Y
- Rob Hyndman has some great tutorials / vignettes for more in-depth info. more on nnetar here

Neural network time series models

We'll apply this to daily flow data from the Cedar River

```
library(waterData)
dat = importDVs(staid = "12119000")
library(lubridate)
dat = dat[which(year(date(dat$dates)) == 2018),]
ggplot(dat, aes(dates,val)) + geom_point() +
  ylab("Flow")
```



Neural network time series models

Using the 'nnetar' function, there's several important arguments to consider

```
mod = nnetar(y=dat$val, p=..., size=...)
```

- p represents the *embedding dimension* or number of lags to include
- size represents the dimension of the hidden layer (# neurons)

Each of these has defaults, but we'll do a couple sensitivities

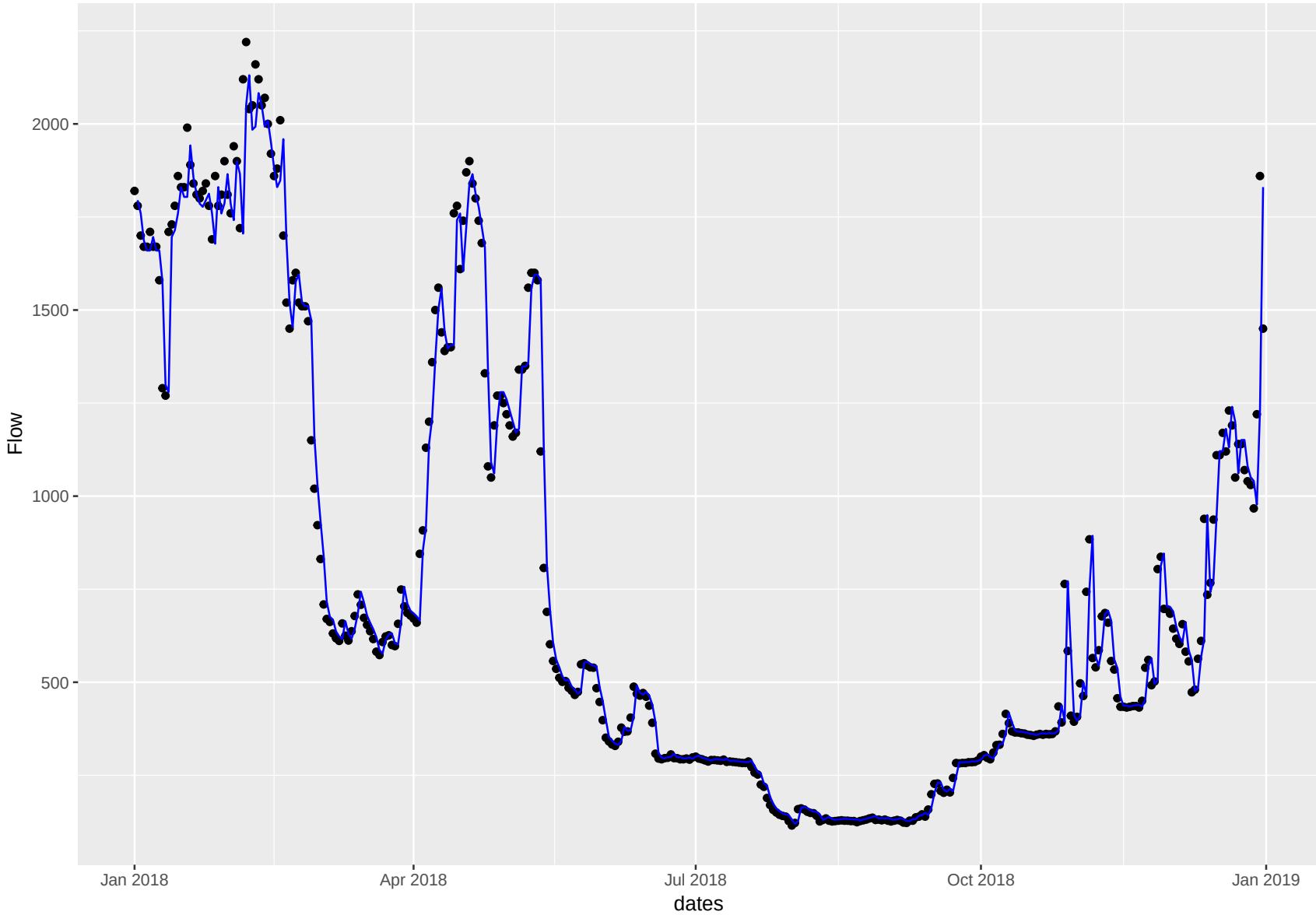
Neural network time series models

First, let's look at varying the number of lagged predictors

```
mod_1 = nnetar(y=dat$val, p=1, size=1)
mod_5 = nnetar(y=dat$val, p=5, size=1)
mod_15 = nnetar(y=dat$val, p=15, size=1)
```

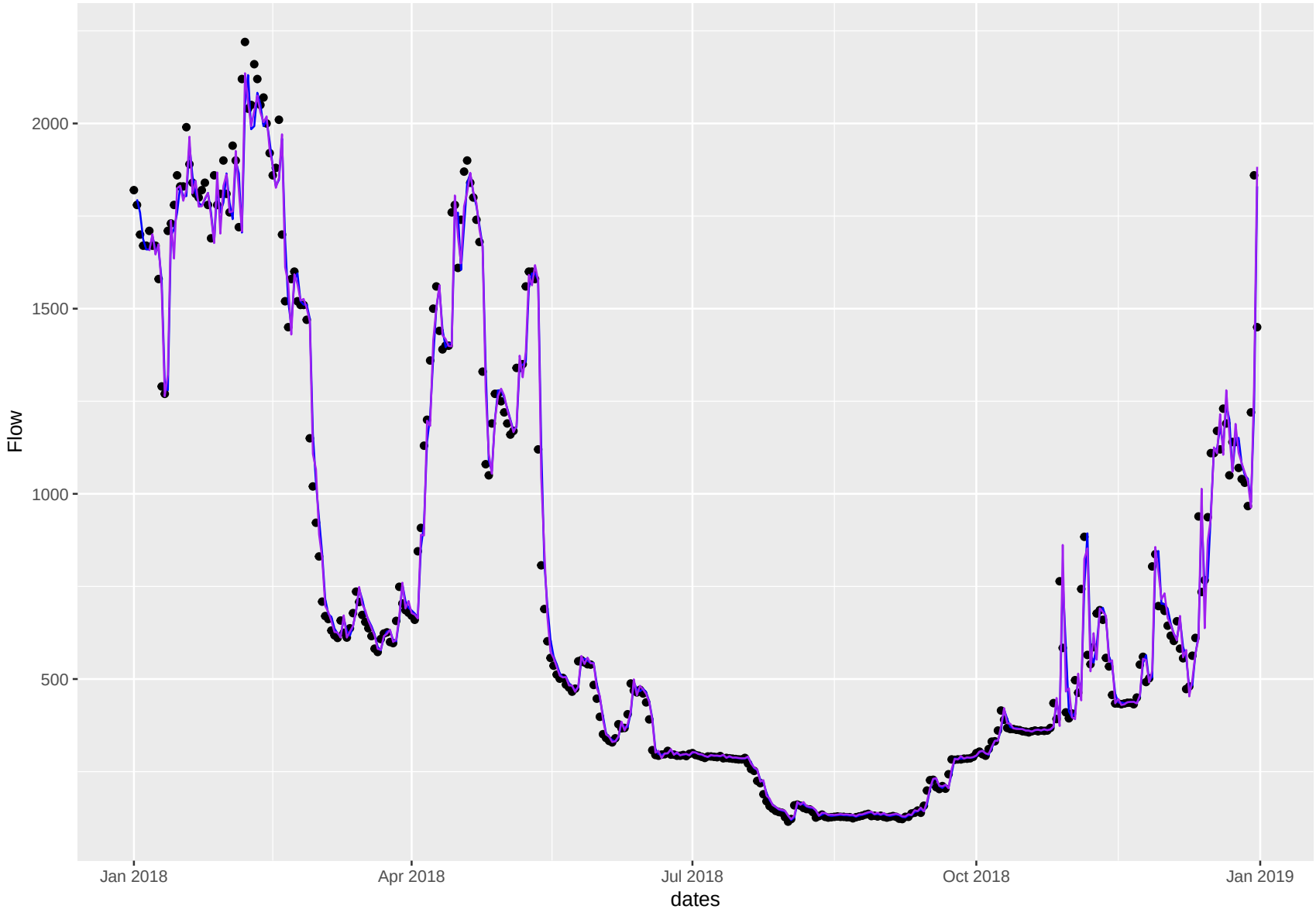
Neural network time series models

Even with embedding dimension = 1, predictions are pretty good



Neural network time series models

Only very slight differences here – slight ones in Feb/March for example



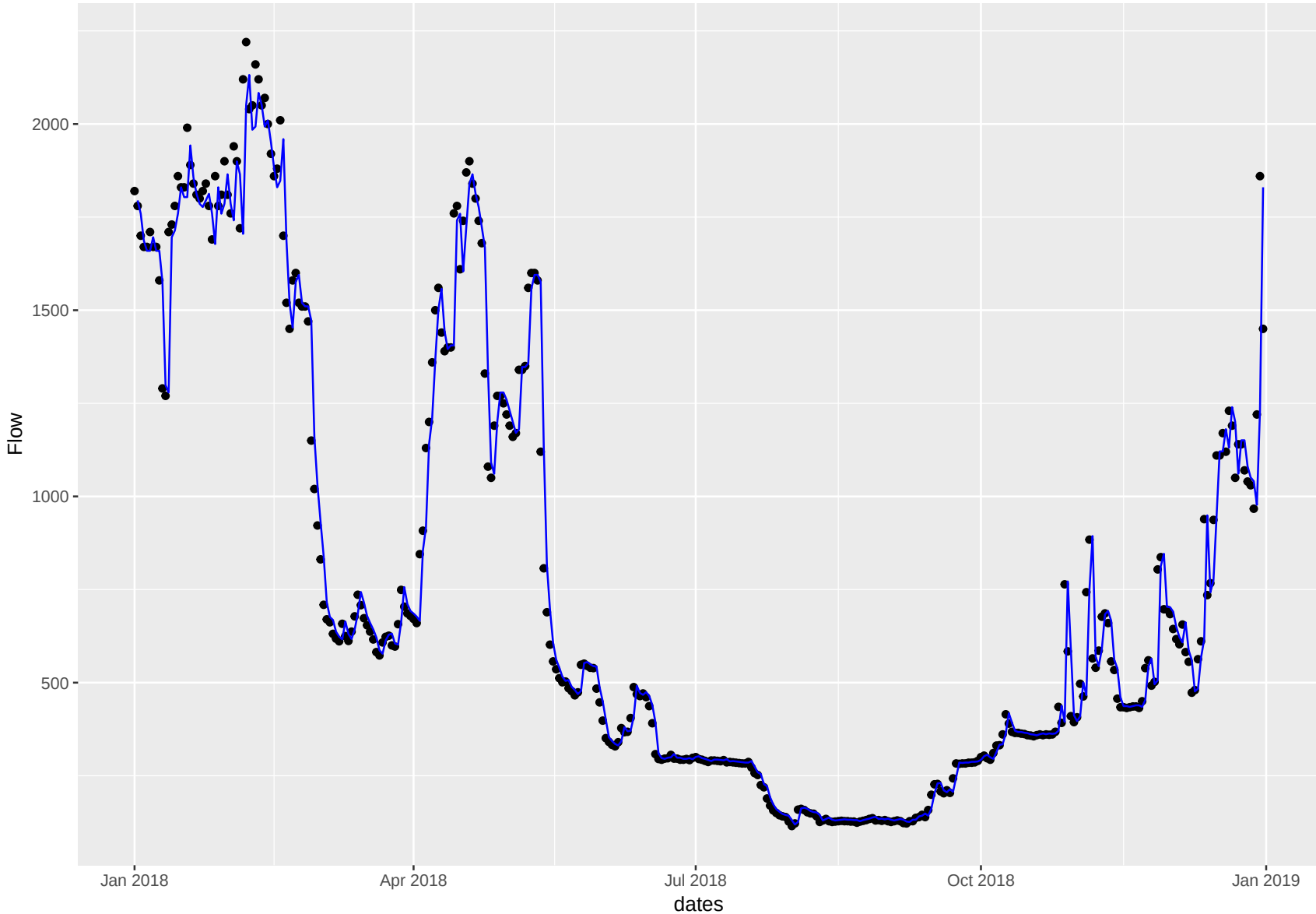
Neural network time series models

Ok, now a sensitivity to the size of the hidden layer

```
mod_1 = nnetar(y=dat$val, p=1, size=1)
mod_5 = nnetar(y=dat$val, p=1, size=5)
mod_15 = nnetar(y=dat$val, p=1, size=15)
```

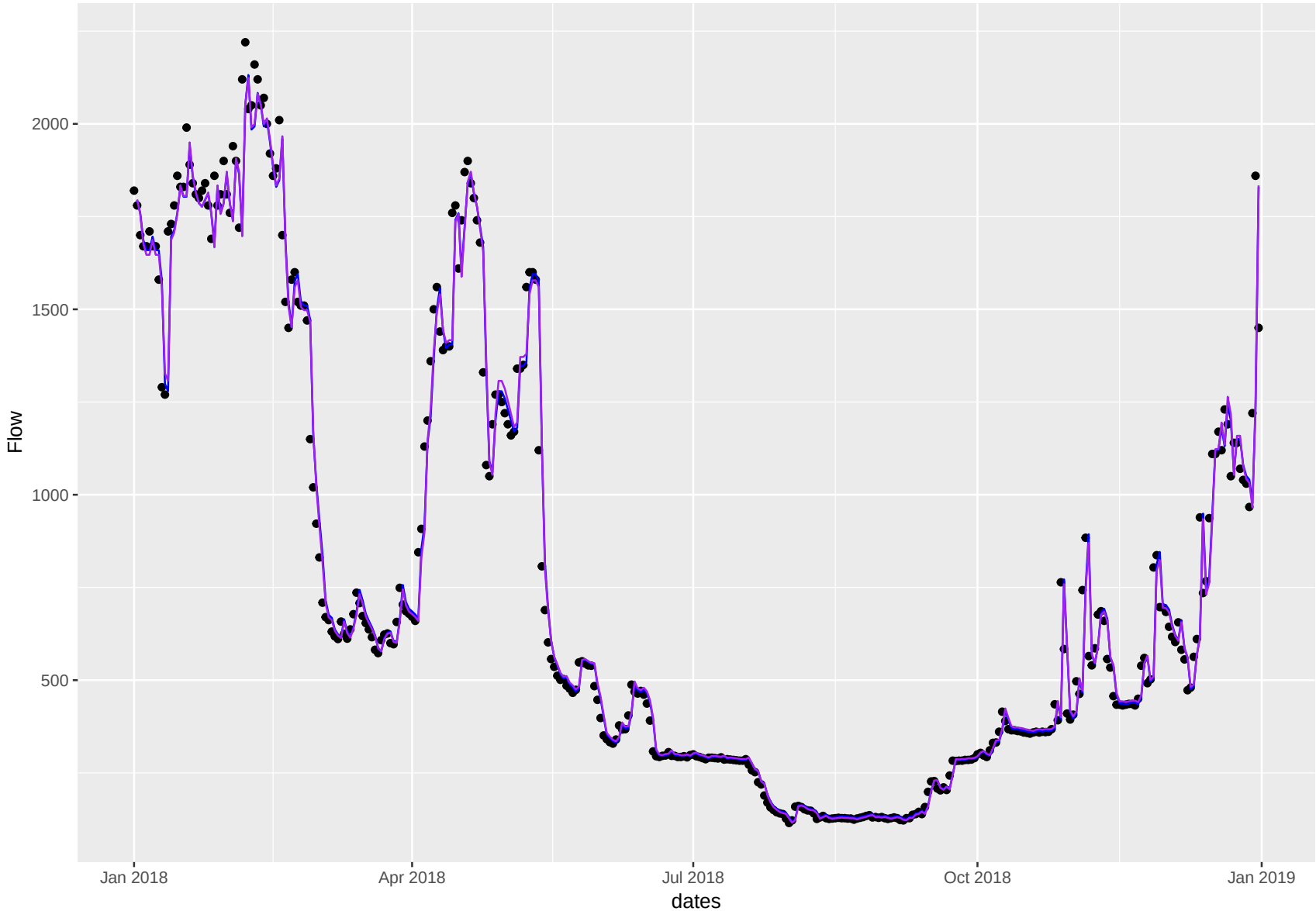
Neural network time series models

Again, the fit with 1 neuron looks pretty good



Neural network time series models

And there only appear to be slight differences as we add more neurons



Neural network time series models

Selecting the size of the network and number of lags (embedding dimension) can be tricky. Many estimation routines will do this for you.

- nnetar will do this for you

For our flow data for example, we can not specify p or *size*

```
mod = nnetar(y=dat$val)
```

Neural network time series models

Output here is as NNAR(p,k) with p equal to the embedding dimension, and k the hidden nodes

```
mod
```

```
## Series: dat$val
## Model:  NNAR(4,2)
## Call:   nnetar(y = dat$val)
##
## Average of 20 networks, each of which is
## a 4-2-1 network with 13 weights
## options were - linear output units
##
## sigma^2 estimated as 8136
```

Neural network time series models

Models are trained on 1-step ahead forecasts

- but this can be customized

Weights are randomized from lots of starting values and forecasts averaged

Point forecasts can be used from the fitted object as before,

```
f = forecast(mod, h = 10)
```

Neural network time series models

Alternative estimation routines also exist in 'tsDyn' package

```
nnetTs(x, m, d = 1, steps = d, size)
```

Just like 'nnetar',

- *m* is embedding dimension
- *size* is dimension of neural network

Empirical dynamic modeling for time series

Simplex link

S-Map link

Convergent cross-mapping link

Hao Ye's Vignette link Yair Daon's Vignette link Owen Petchey's Vignette link

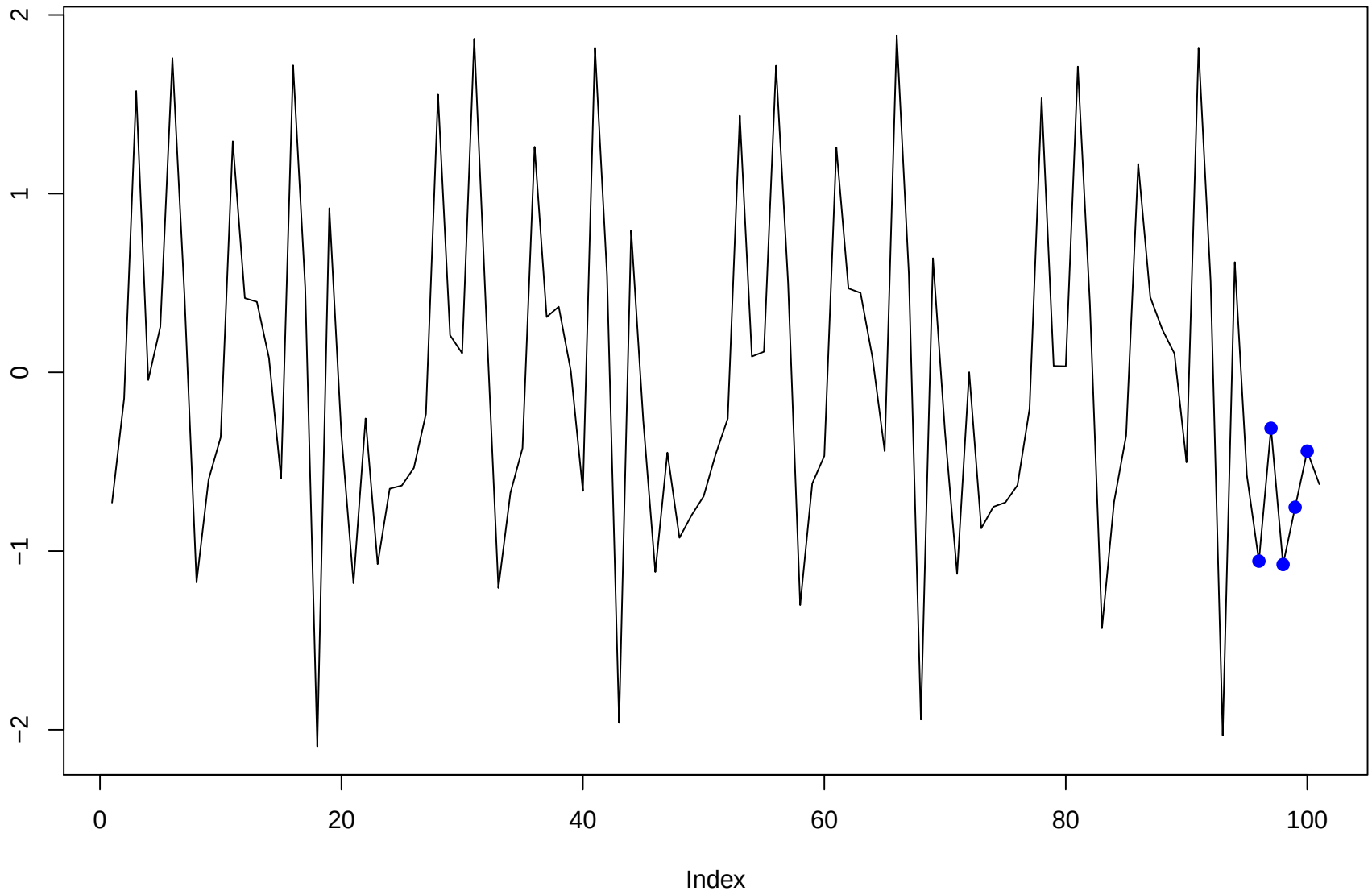
Empirical dynamic modeling for time series

These tools generally represent nearest neighbor forecasting (projecting) routines

- Like NNets, there is a lag (embedding dimension) that needs to be chosen
- Also need to specify the number of nearest neighbors (default Simplex = E+1)

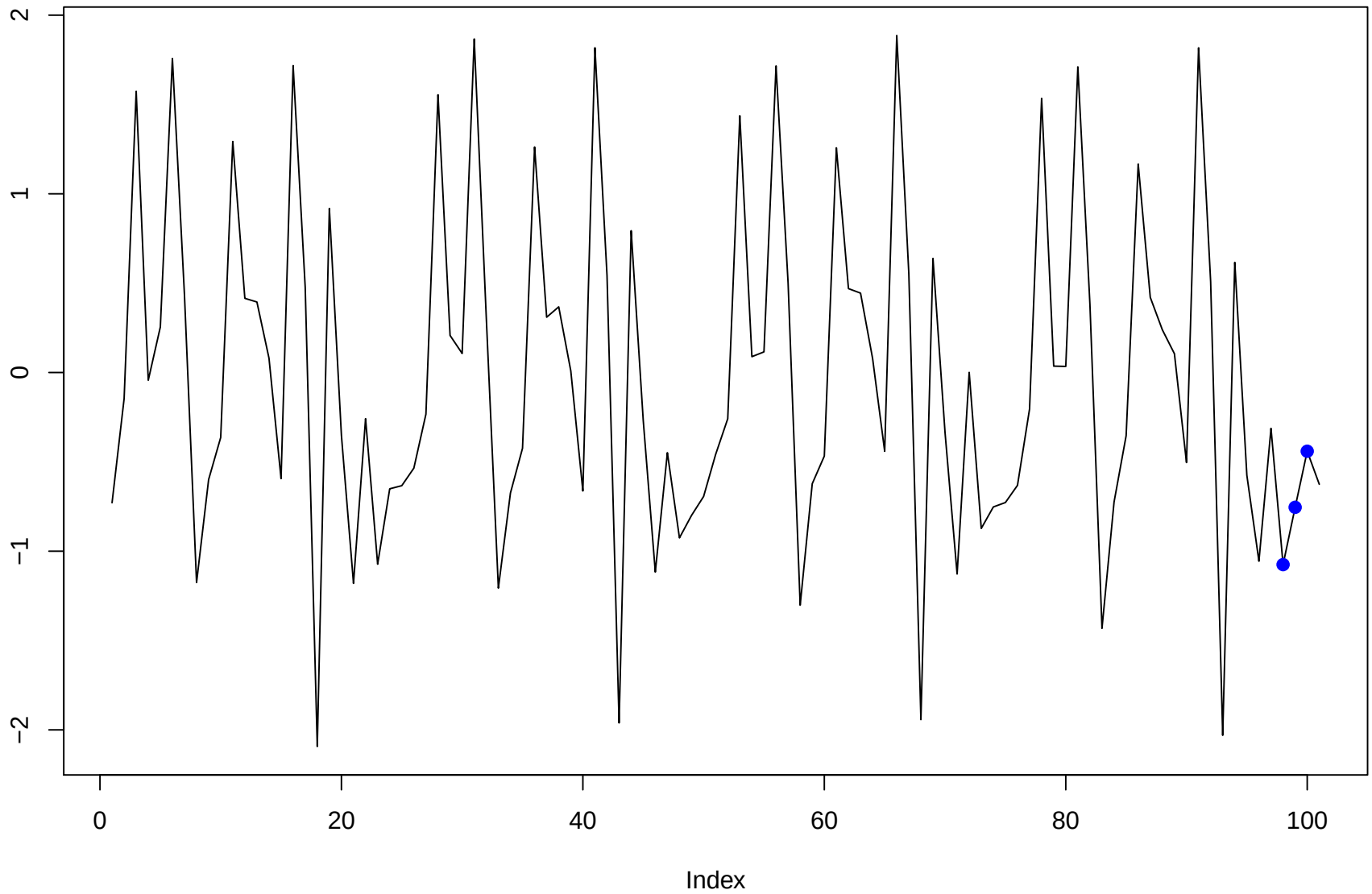
Empirical dynamic modeling for time series

First, the embedding dimension. We'll start with a lag / embedding dimension of $E = 5$



Empirical dynamic modeling for time series

Or we could use a value of $E = 3$



Empirical dynamic modeling for time series

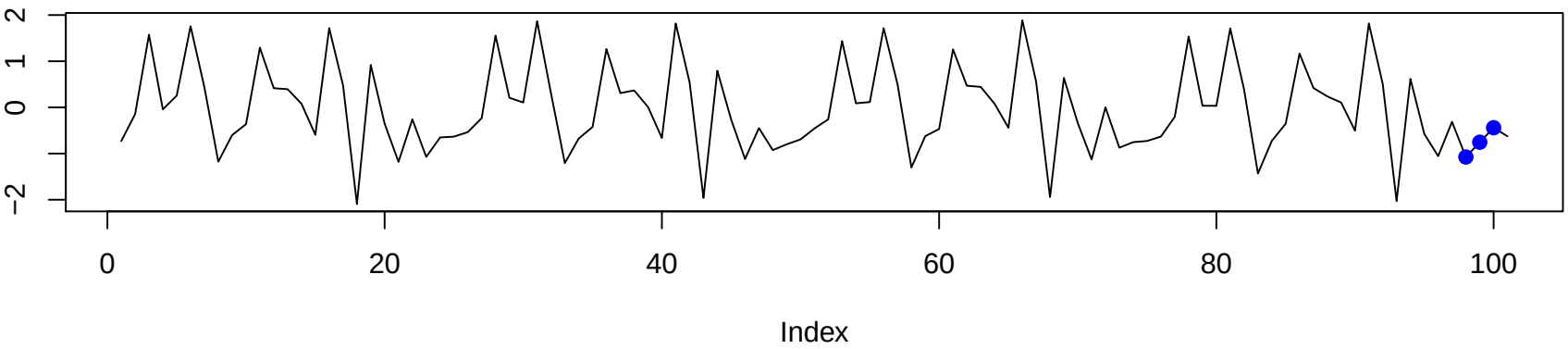
There's some optimal embedding dimension we can select

- predictions are likely affected strongly by recent dynamics
- it is less likely that conditions in the distant past are also useful at making projections
- as a result, predictability may increase slightly with greater values of E and then eventually decline

Empirical dynamic modeling for time series

Internally, forecasts will be made based on the library of predictors

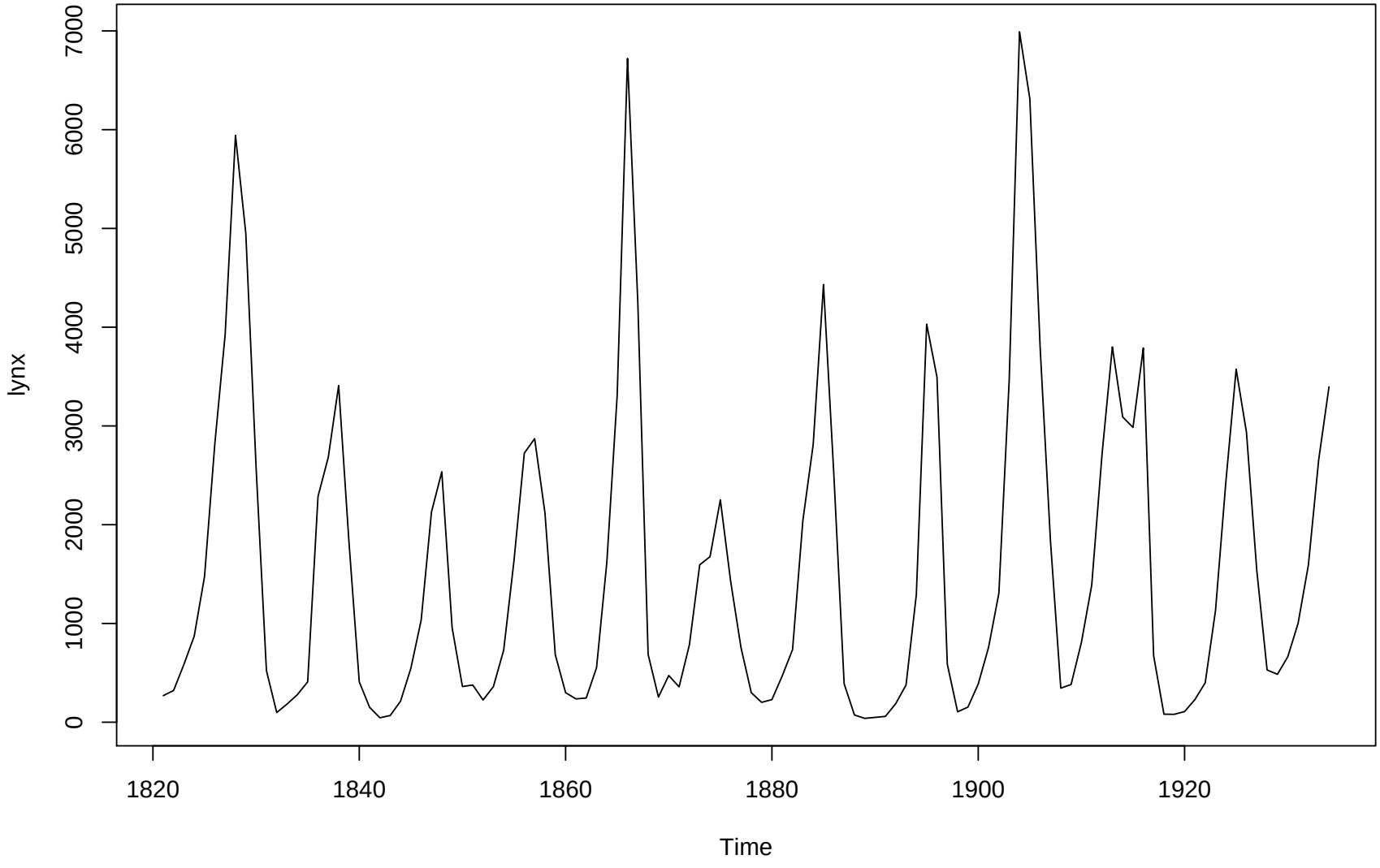
- This library is generated from previous dynamics that mirror the most recent time period
- Forecasts are then averaged + validated (cross - validation)



http://deepeco.ucsd.edu/simplex/

Empirical dynamic modeling for time series

Examples: let's start with the classic 'lynx' dataset



Empirical dynamic modeling for time series

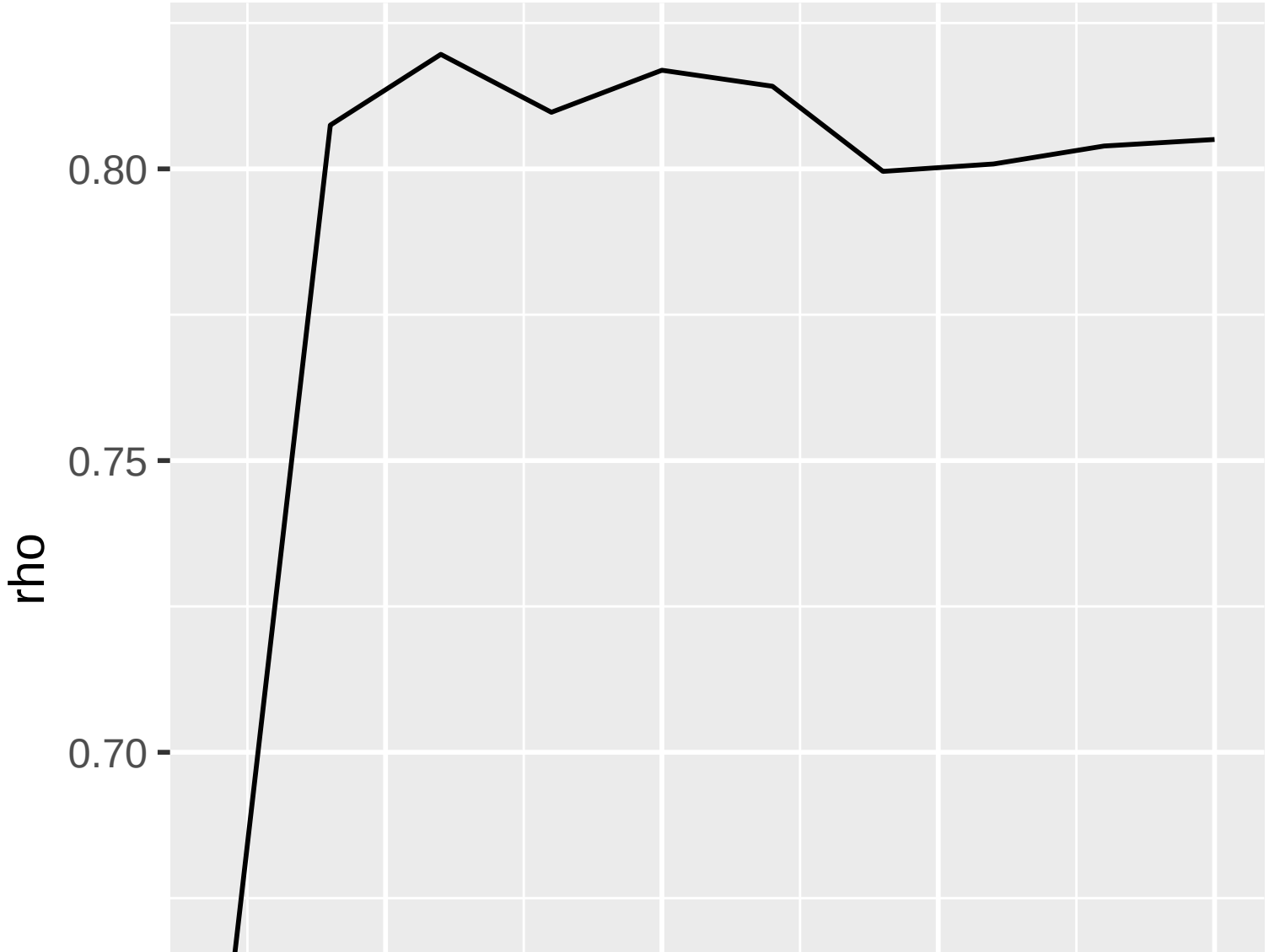
Examples: let's start with the classic 'lynx' dataset

```
mod = rEDM::simplex(as.numeric(lynx), E=1:10)
```

Empirical dynamic modeling for time series

Predictability increases a lot when E=2, but pretty flat after

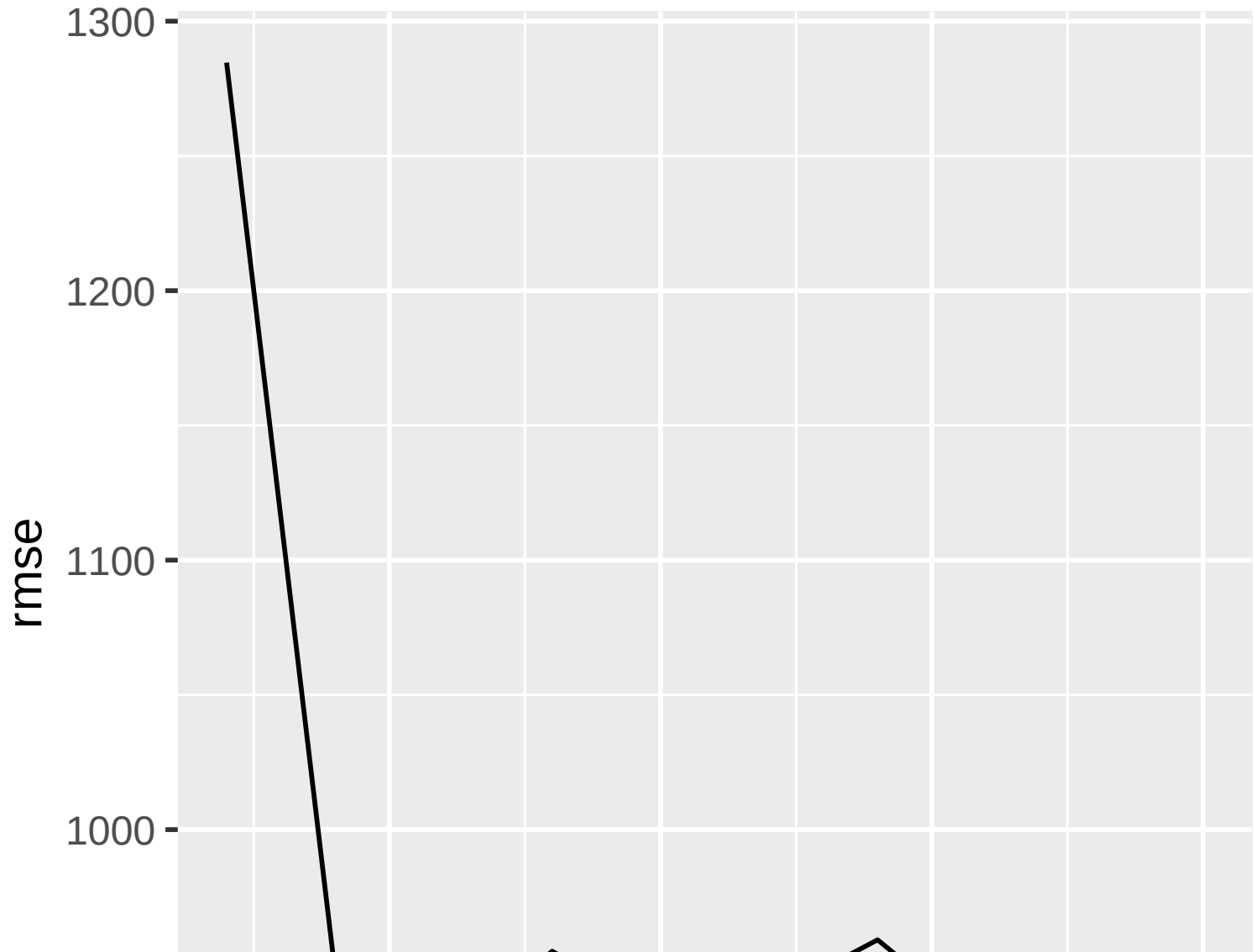
```
ggplot(mod, aes(E,rho)) + geom_line()
```



Empirical dynamic modeling for time series

Similar patterns with RMSE

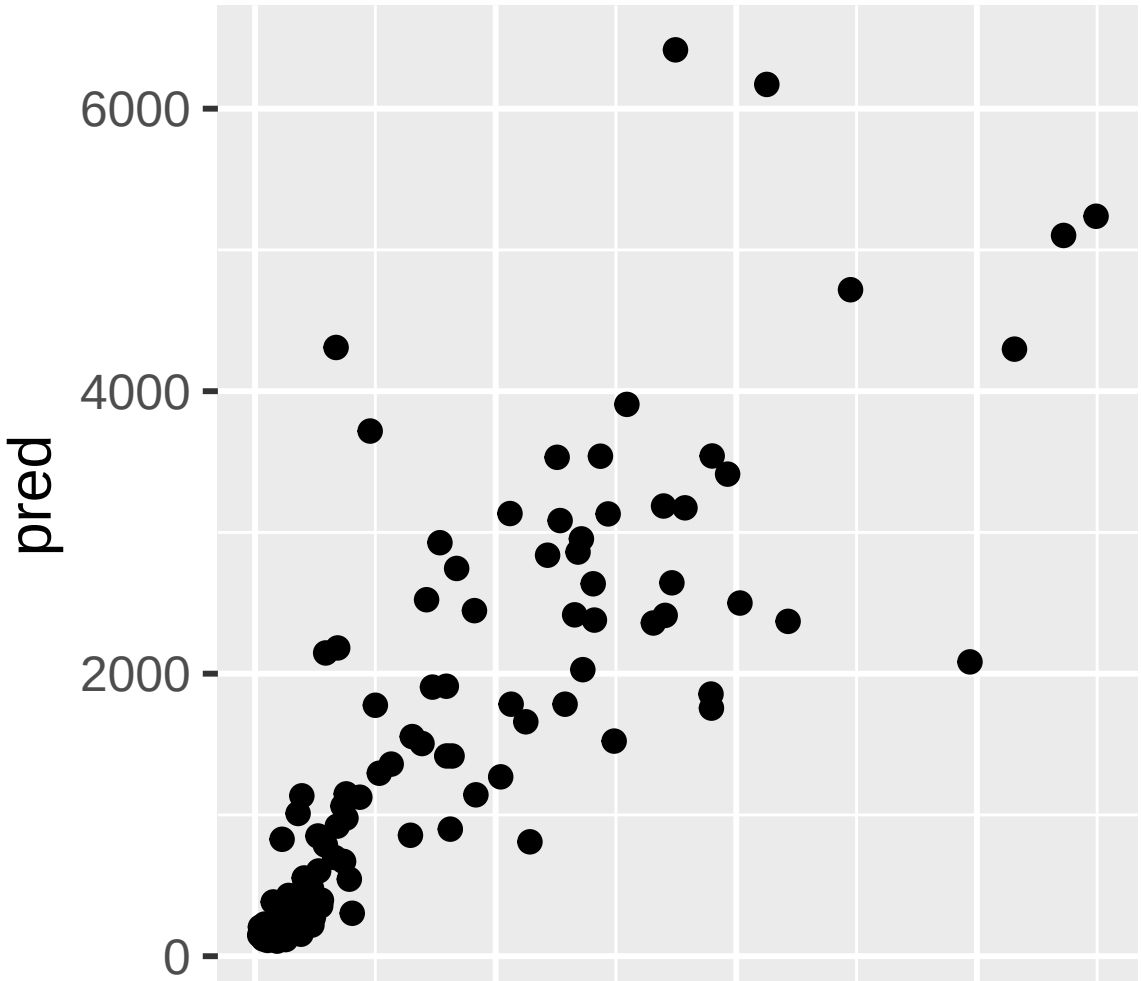
```
ggplot(mod, aes(E,rmse)) + geom_line()
```



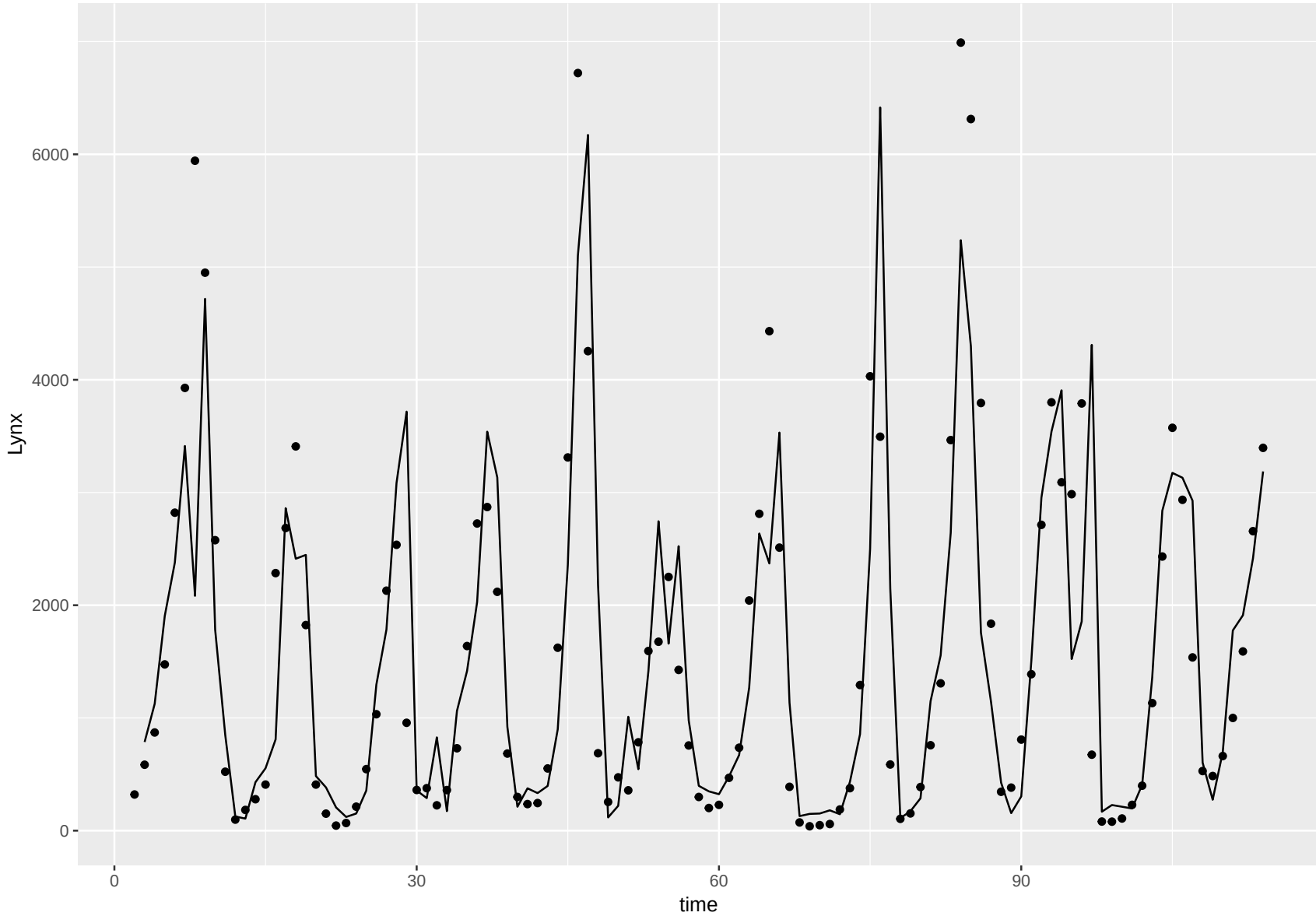
Empirical dynamic modeling for time series

We can also pull out predictions (off by default) with the 'stats_only' argument,

```
mod = rEDM::simplex(as.numeric(lynx), E=1:10, stats_only=FA
ggplot(mod[[2]]$model_output, aes(obs, pred)) + geom_point(
```



Empirical dynamic modeling for time series



Empirical dynamic modeling for time series

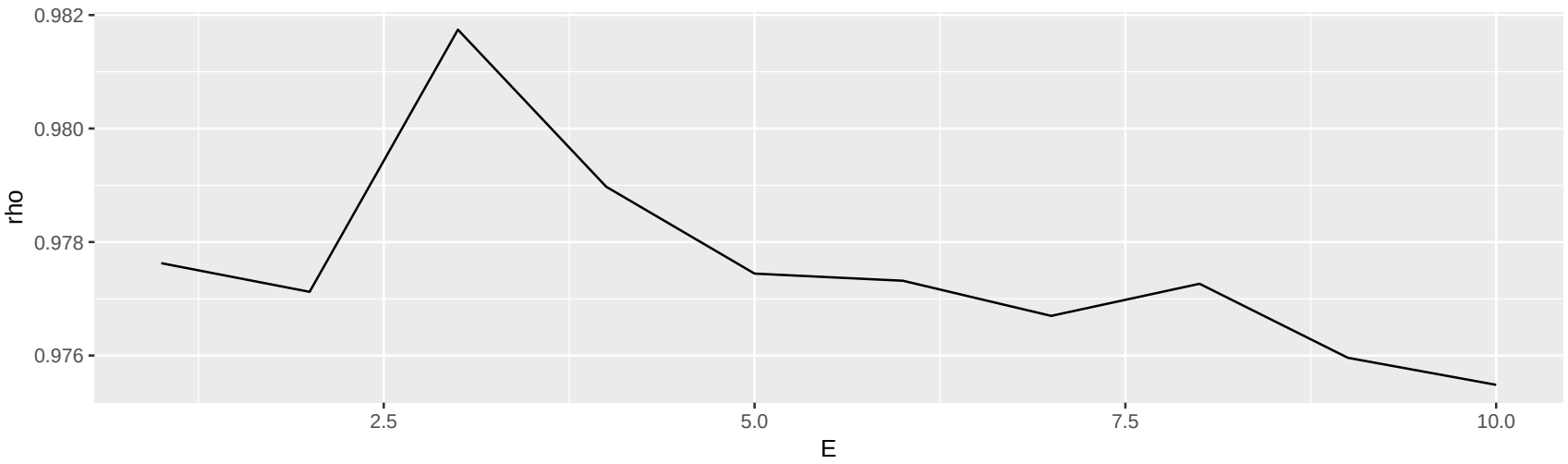
We can also play with out of sample forecasting by specifiying the data to be used in the library ('lib') and data to be used for prediction ('pred'). For example, to forecast the last 14 data points of the lynx series, we could use

```
mod = rEDM::simplex(as.numeric(lynx), E=1:10, stats_only=FA
      lib=c(1,100), pred=c(101,114))
```

Empirical dynamic modeling for time series

As a second example, let's fit this to the water data from the Cedar River.

```
mod = rEDM::simplex(dat$val, E=1:10)
ggplot(mod, aes(E,rho)) + geom_line()
```



Empirical dynamic modeling for time series

For this application, it's also interesting to maybe compare the Simplex fits against the neural network time series. Here, the 'forecast skill' (rho) is 0.9817 for the best model (E=3).

Fitting the nnet model yields a slightly higher correlation (0.988)

```
mod_nn = nnetar(y=dat$val)
```

Empirical dynamic modeling for time series

Beyond Simplex: in the interest of time, we haven't talked about SMAP or Cross Mapping

- Smap (rEDM::s_map) is similar to Simplex, but also estimates a non linear parameter θ
- Cross mapping (rEDM::ccm) models causality in multiple time series, using information in lags