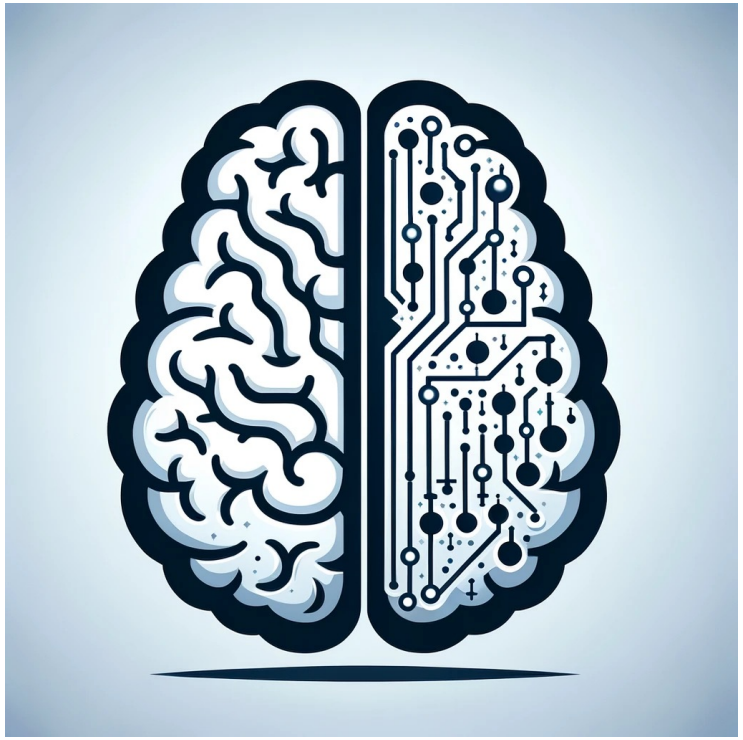
EN 601.473/601.673: Cognitive Artificial Intelligence (CogAI)

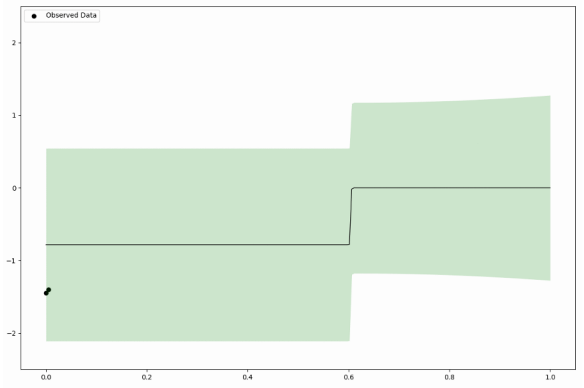


Lecture 15: Real-world applications

Tianmin Shu

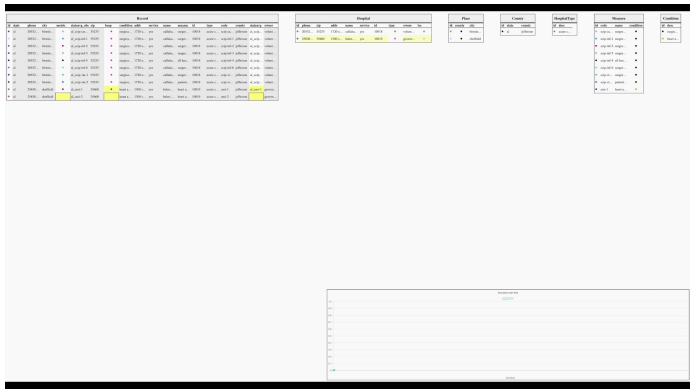
Applications where probabilistic programming outperforms SOTA machine learning

Automated data modeling



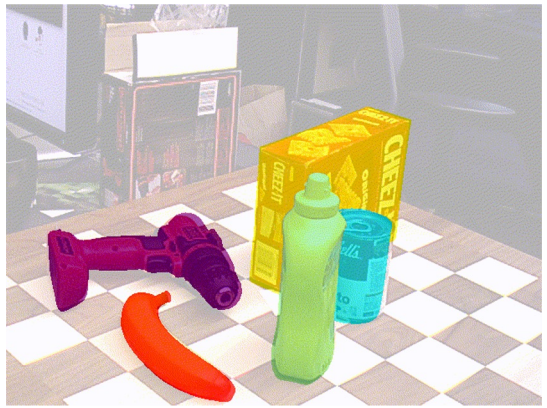
Saad, Cusumano-Towner, Schaechtle, Rinard, Mansinghka (POPL 2019)
Saad and Mansinghka (UAI 2021)
Schaechtle, Freer, Shelby, Saad, and Mansinghka (AutoML 2022)

Common-sense data cleaning



Lew, Agrawal, Sontag, and Mansinghka (AISTATS 2021)

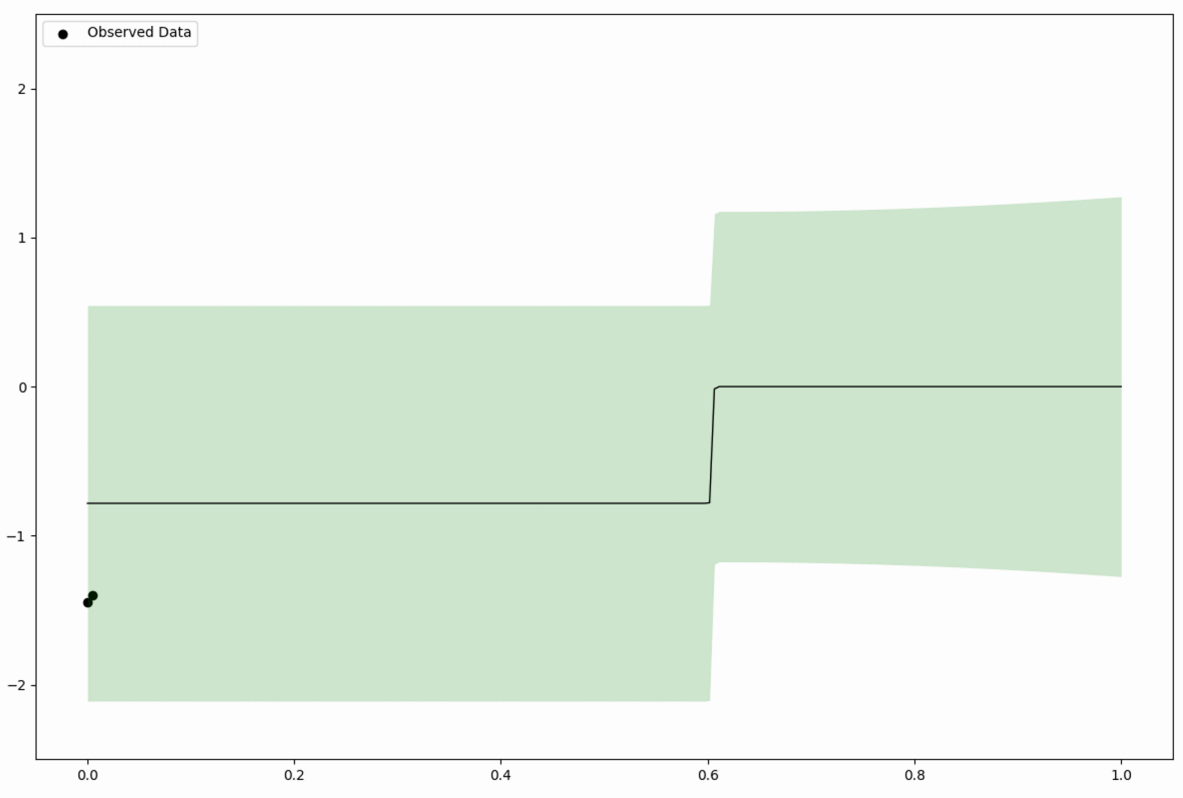
3D scene perception



Gothoskar et al. (NeurIPS 2021)

Automatic data modeling

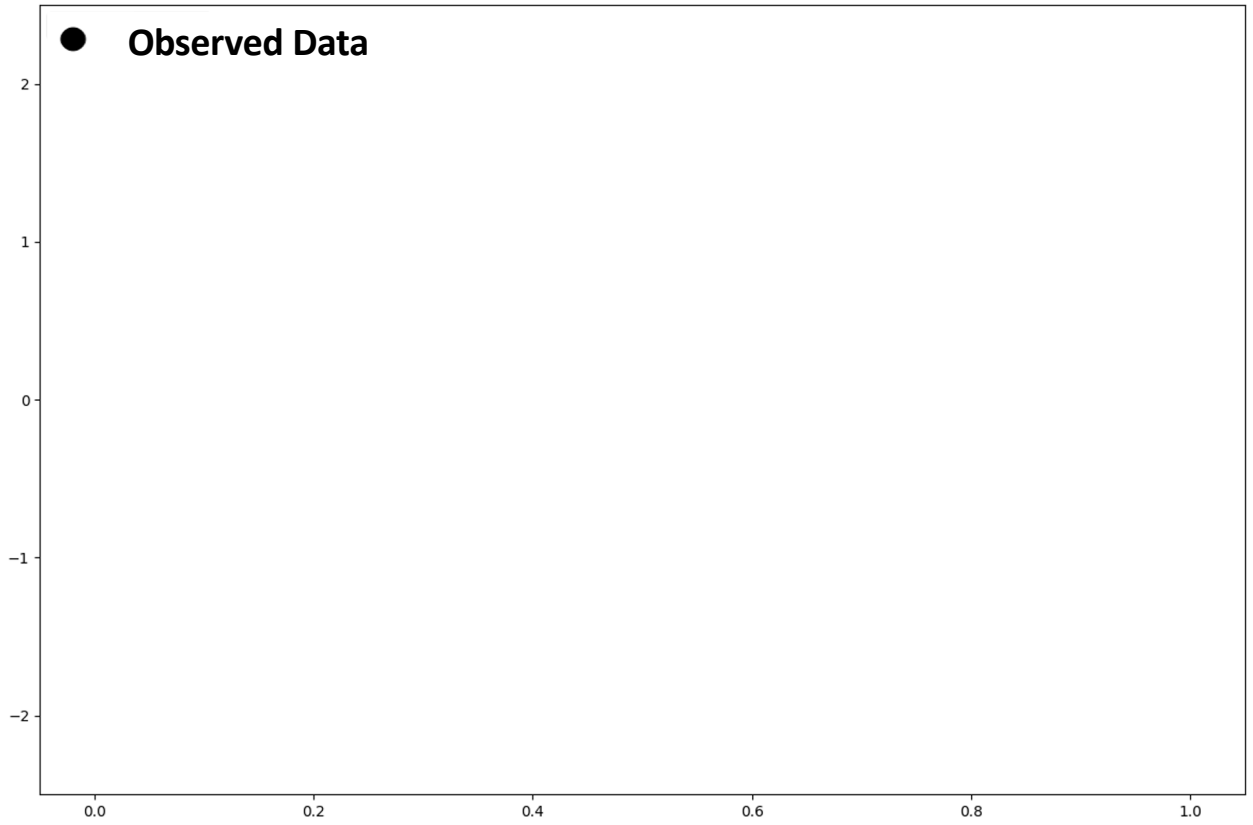
- Learning the structure of probabilistic programs from data
- Implementation: MCMC + SMC in Gen



[Slides adapted from Vikash Mansinghka]

Online learning in a time series DSL

- DSL: domain specific language



Our Goal

learn a probabilistic program that accurately models this data

P(function | data)

Preliminary: Gaussian Process Models

- Gaussian Processes (GPs) define a family of nonparametric regression models that are widely used for data modeling
- It is possible to use GPs to discover complex temporal patterns
- Let $D = (\boldsymbol{t}, \boldsymbol{x})$ be a dataset with *n* time points $\boldsymbol{t} = (t_1, \cdots, t_n)$ and observations $\boldsymbol{x} = (x_1, \cdots, x_n)$
- As the true function that generated the data is unknown, we use a GP function prior $f \sim GP(0, k)$
- A multivariate normal distribution:
  - $[f(t_1), \cdots, f(t_n)] \sim MultivariateNormal([0, \cdots, 0], \left[k(t_i, t_j)\right]_{1 \le i,j \le n})$
  - A kernel $k(t_i, t_j)$ defines the covariance matrix $\left[k(t_i, t_j)\right]_{1 \le i,j \le n}$
- We then assume an additive observation model, so that

$$x_i \sim Normal(f(t_i), \epsilon)$$

DSL: composing kernels to define a GP prior

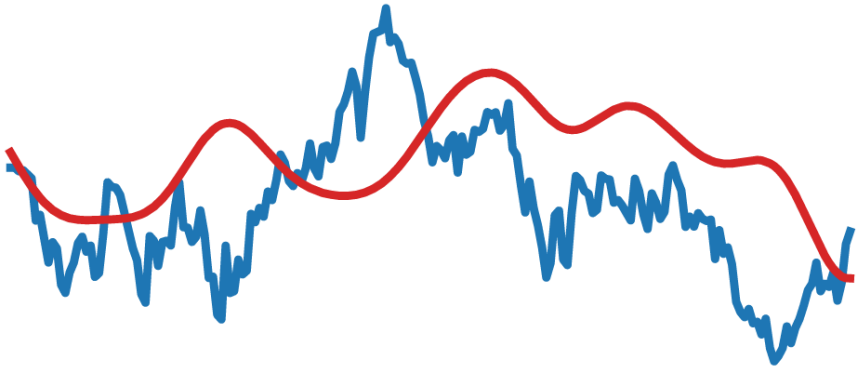
- $[f(t_1), \cdots, f(t_n)] \sim MultivariateNormal([0, \cdots, 0], \left[k\left(t_i, t_j\right)\right]_{1 \le i,j \le n})$
- $x_i \sim Normal(f(t_i), \epsilon)$
- Example kernels:
  - Linear kernel: $k_{Lin}\left(t_i, t_j\right) = \sigma_b^2 + \sigma_v^2 (t_i - c)\left(t_j - c\right)$
  - Squared Exponential kernel: $k_{SE}\left(t_i, t_j\right) = \sigma \exp(\frac{-\left(t_i - t_j\right)^2}{2\ell^2})$
- Each kernel has some parameters
- Basic kernels can be combined together through operations like sum, product, and changepoint (smoothly transitions between two kernels at some t-location)

$$K \in \text{Kernel} ::= (\texttt{C}\ v) \mid (\texttt{WN}\ v) \mid (\texttt{SE}\ v) \mid (\texttt{LIN}\ v) \mid (\texttt{PER}\ v_1\ v_2) \qquad \text{[BaseKernels]}$$
$$\mid (+\ K_1\ K_2) \mid (\times\ K_1\ K_1) \mid (\texttt{CP}\ v\ K_1\ K_2) \qquad \text{[CompositeKernels]}$$

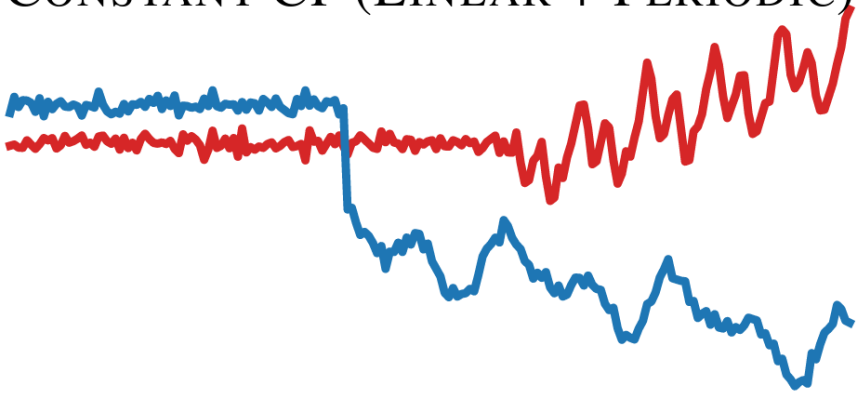
Goal of the probabilistic program

- Write a probabilistic program to infer the compositional kernel and its parameters that fit observed data points, $\mathrm{P}(k)$

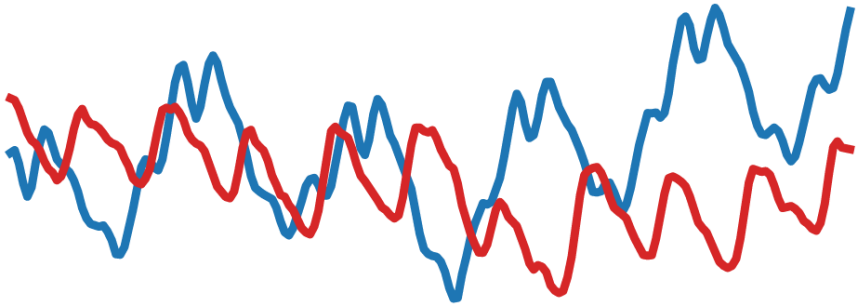
GAMMAEXP



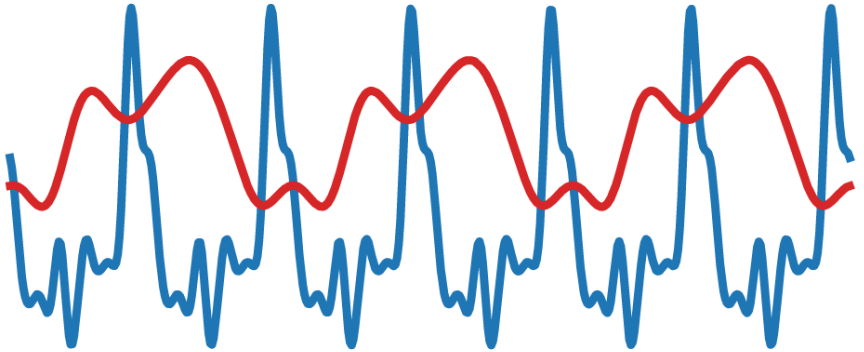
CONSTANT CP (LINEAR + PERIODIC)



(LINEAR * PERIODIC) + GAMMAEXP



(PERIODIC * PERIODIC)



A flexible class of time series probabilistic programs

Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

A flexible class of time series probabilistic programs

t

input time points

Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})
```

A flexible class of time series probabilistic programs

`t` **input time points**

`DSL_program` `Σ` **time series structure**

Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]
```

A flexible class of time series probabilistic programs

`t`	**input time points**
`DSL_program` `Σ`	**time series structure**
`ε`	**noise level**

Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)
```

A flexible class of time series probabilistic programs

`t`	**input time points**
`DSL_program` `Σ`	**time series structure**
`ε`	**noise level**
`x`	**time series values**

$$x[1] \sim \text{Normal}(0, \Sigma[1,1])$$

$$x[i] \mid x[1{:}i\text{-}1] \sim \text{Normal}(\mu_i, \sigma_i + \varepsilon)$$

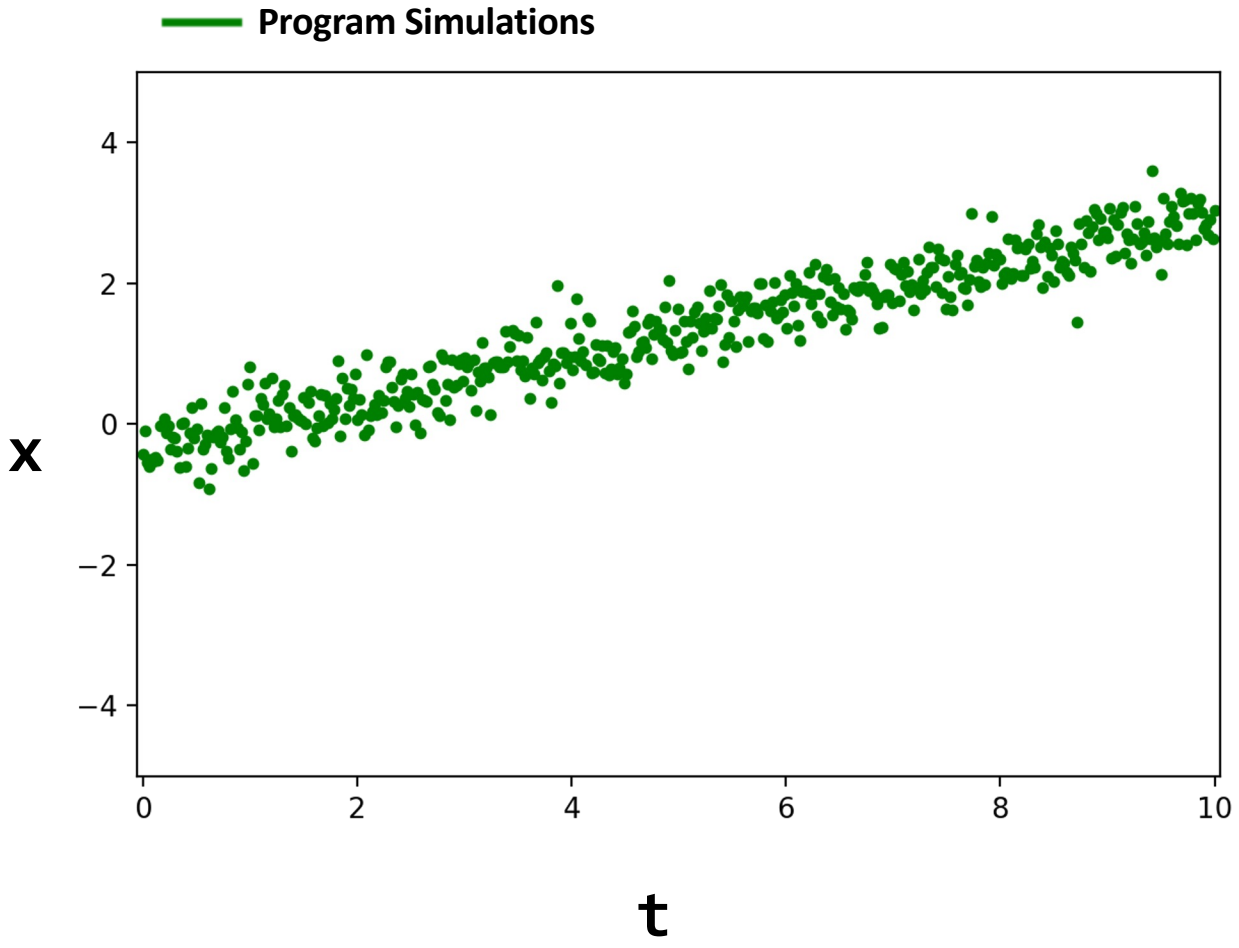
Property 1 Continuous-Time

Property 2 Time-Inhomogeneous

Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

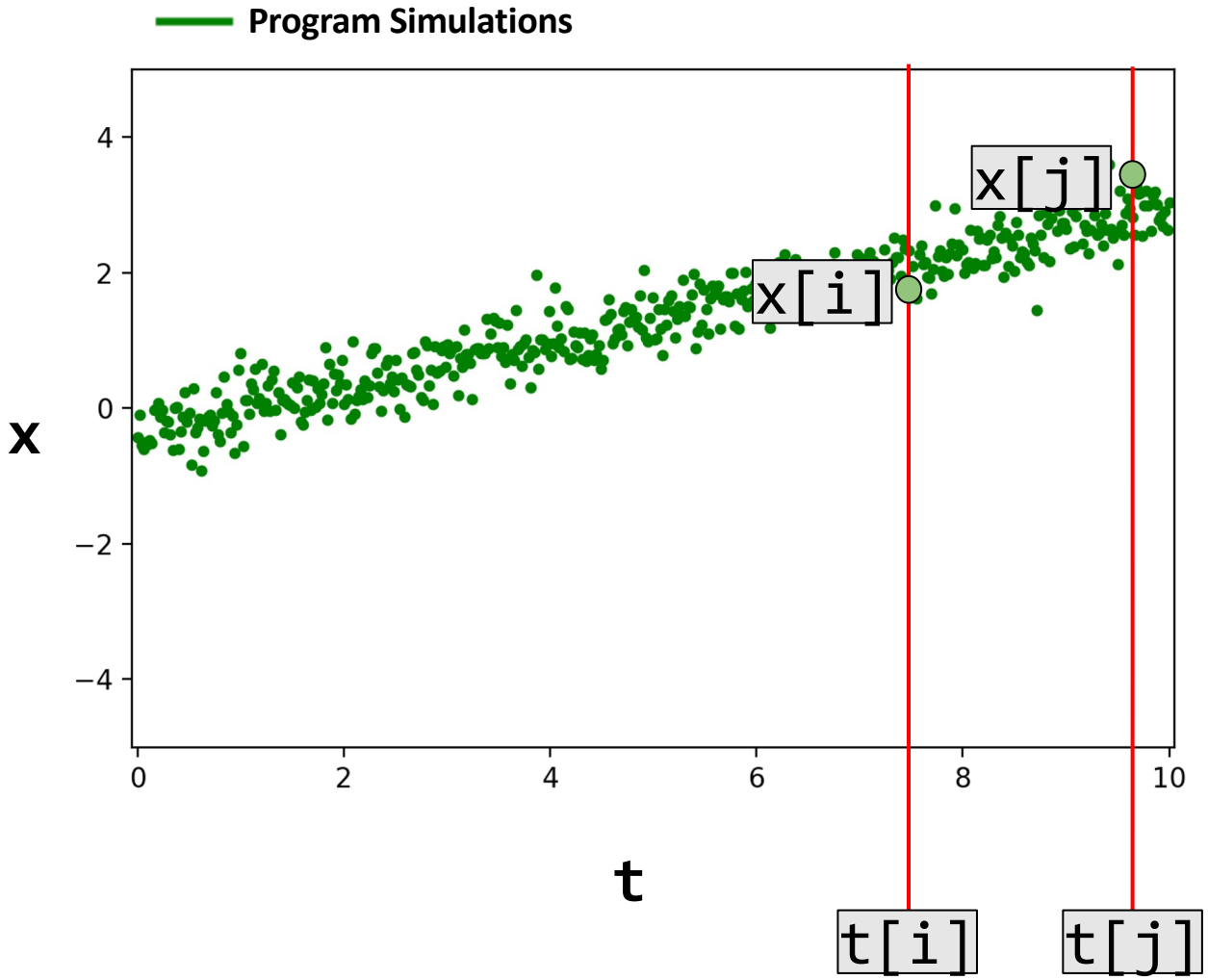
A flexible class of time series probabilistic programs



Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

A flexible class of time series probabilistic programs



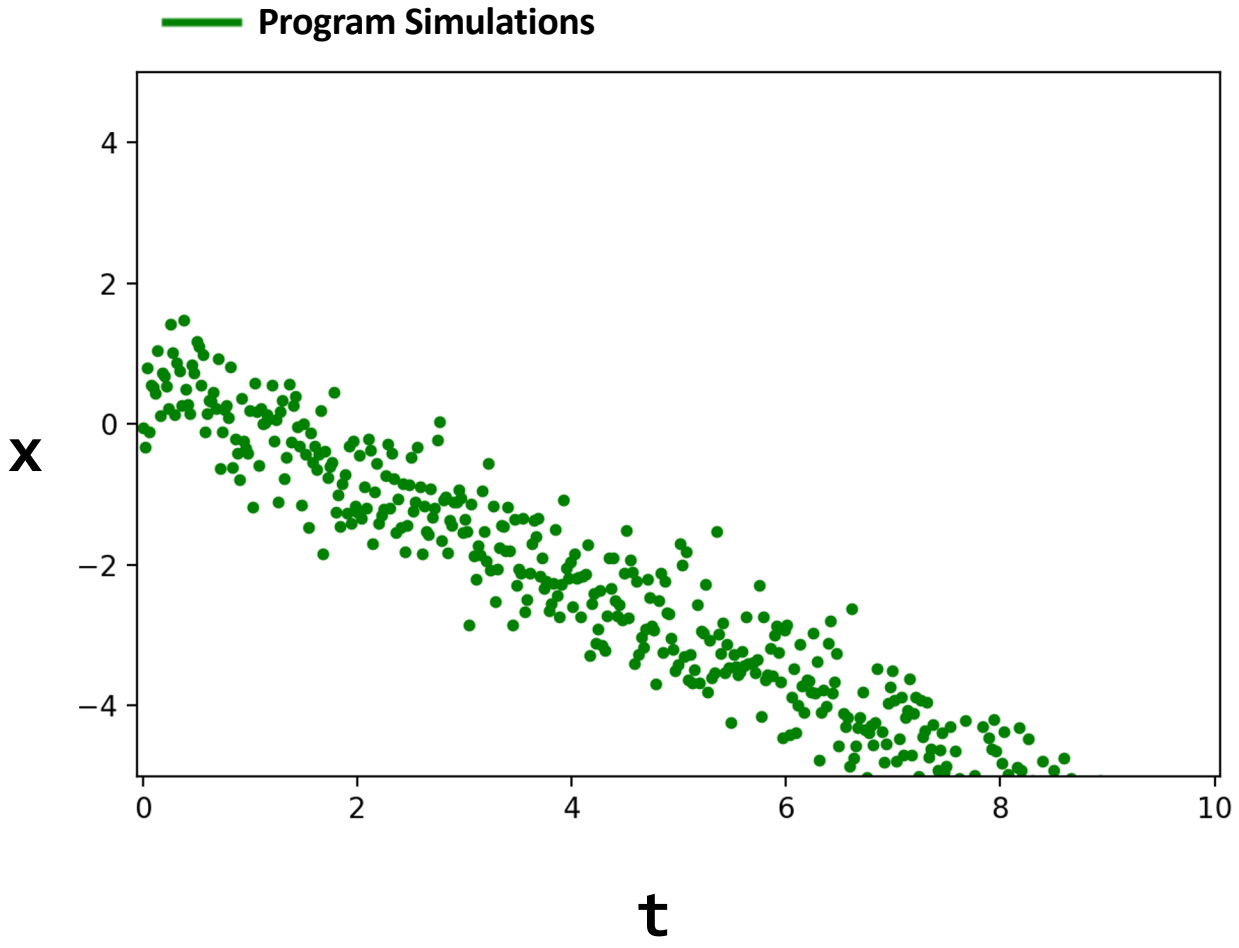
Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

execution trace

ϵ	0.28
x[0]	-0.33
x[1]	-0.32
…	…
x[N]	2.96

A flexible class of time series probabilistic programs



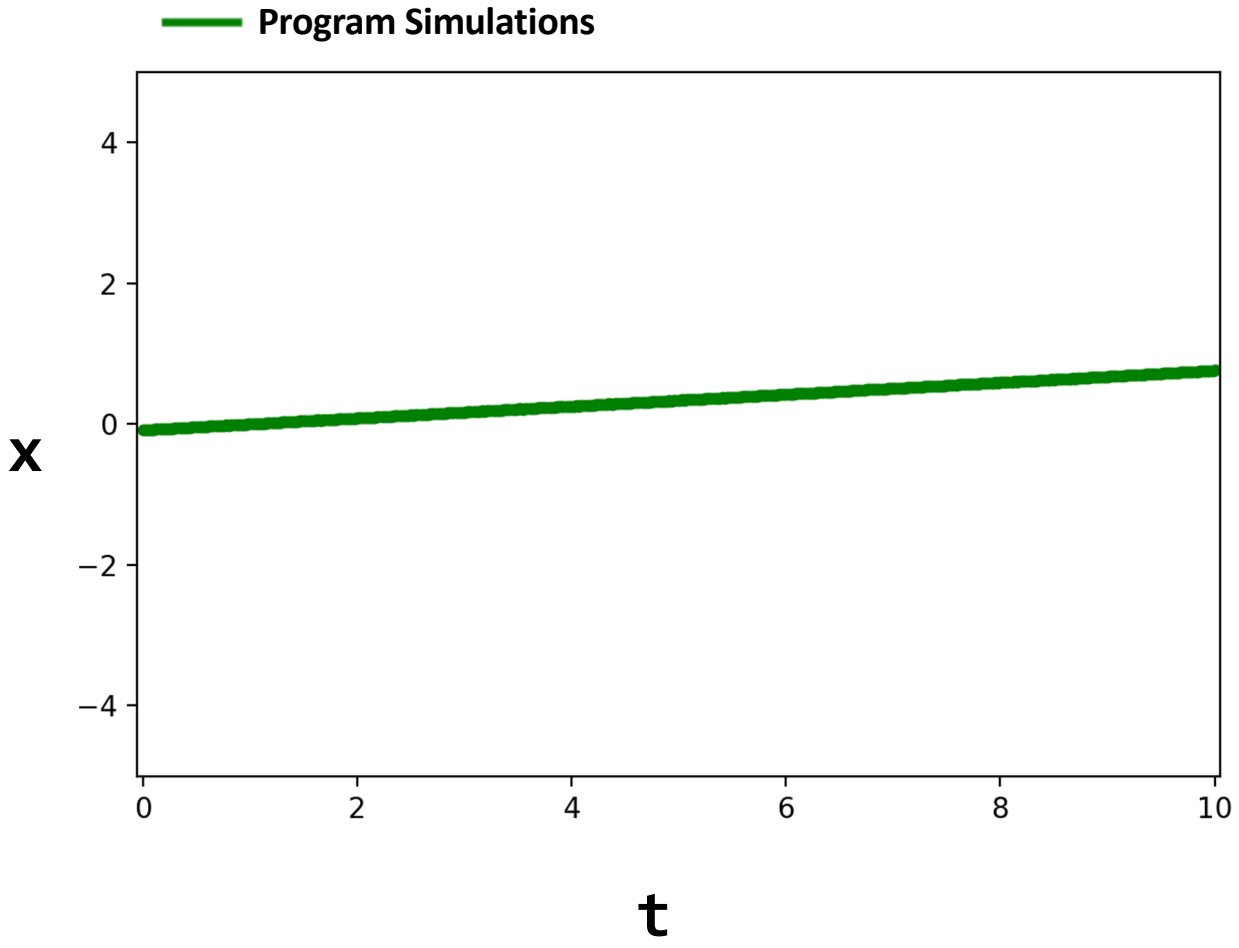
Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

execution trace

ϵ	0.65
x[0]	-0.27
x[1]	-0.13
…	…
x[N]	-4.91

A flexible class of time series probabilistic programs



Probabilistic Program

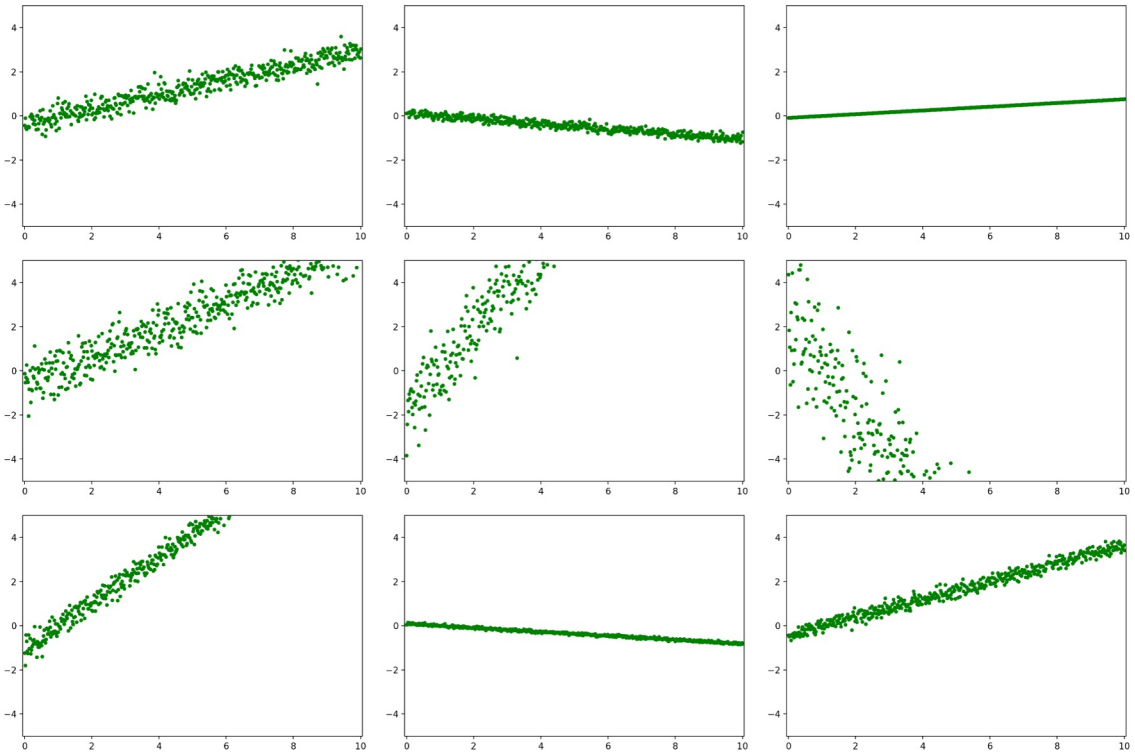
```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

execution trace

ϵ	0.001
x[0]	-0.02
x[1]	-0.01
…	…
x[N]	0.47

A flexible class of time series probabilistic programs

9 Independent Executions

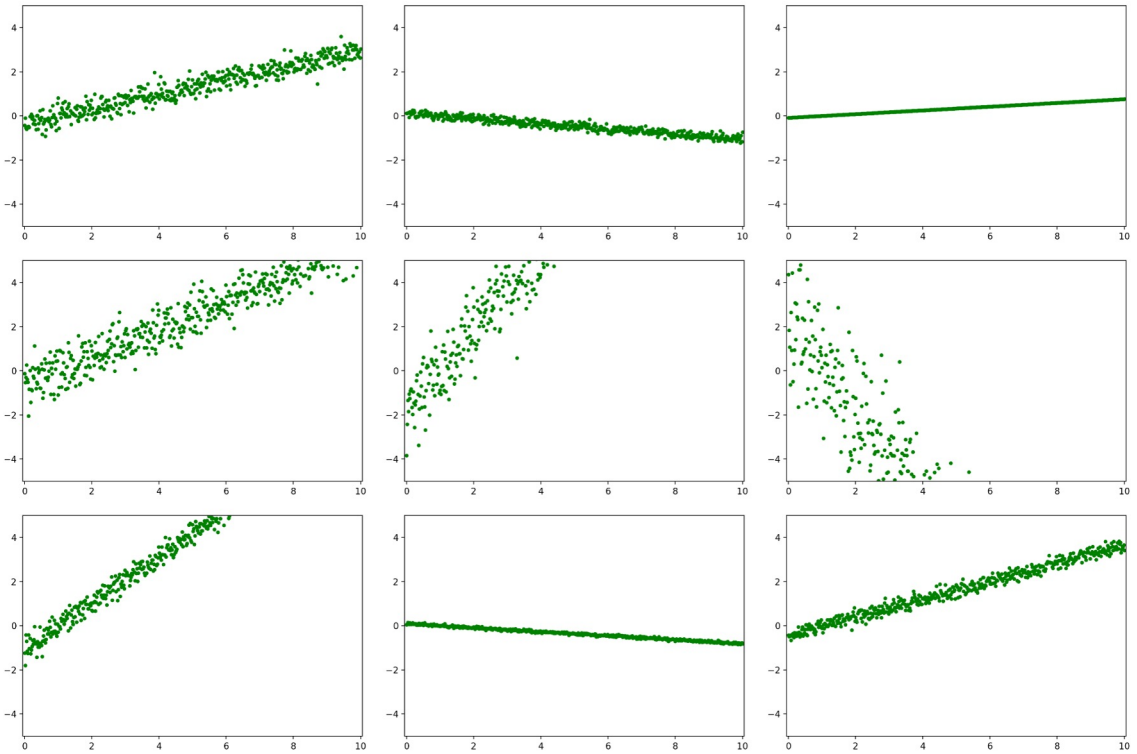


Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

A flexible class of time series probabilistic programs

9 Independent Executions



Probabilistic Program

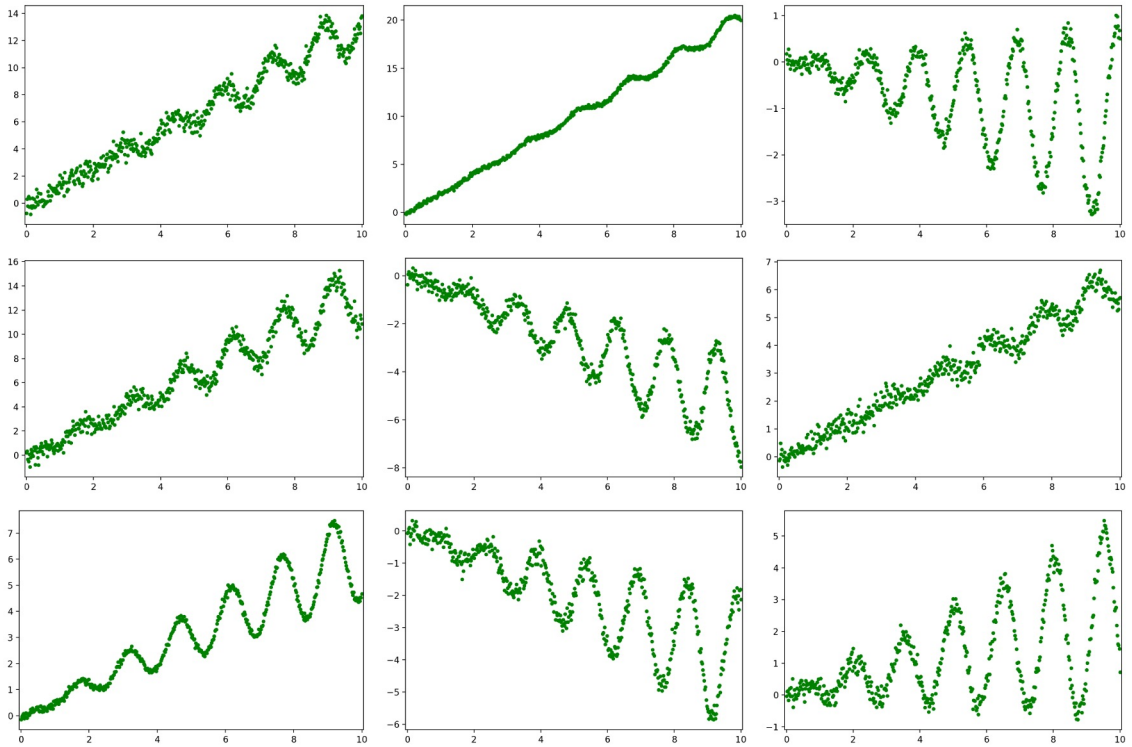
```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Linear(1.0)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
      for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

$$\Sigma[i,j] = (t[i]-1)(t[j]-1)$$

covariance between variables
`x[i]` and `x[j]` across executions

A flexible class of time series probabilistic programs

9 Independent Executions



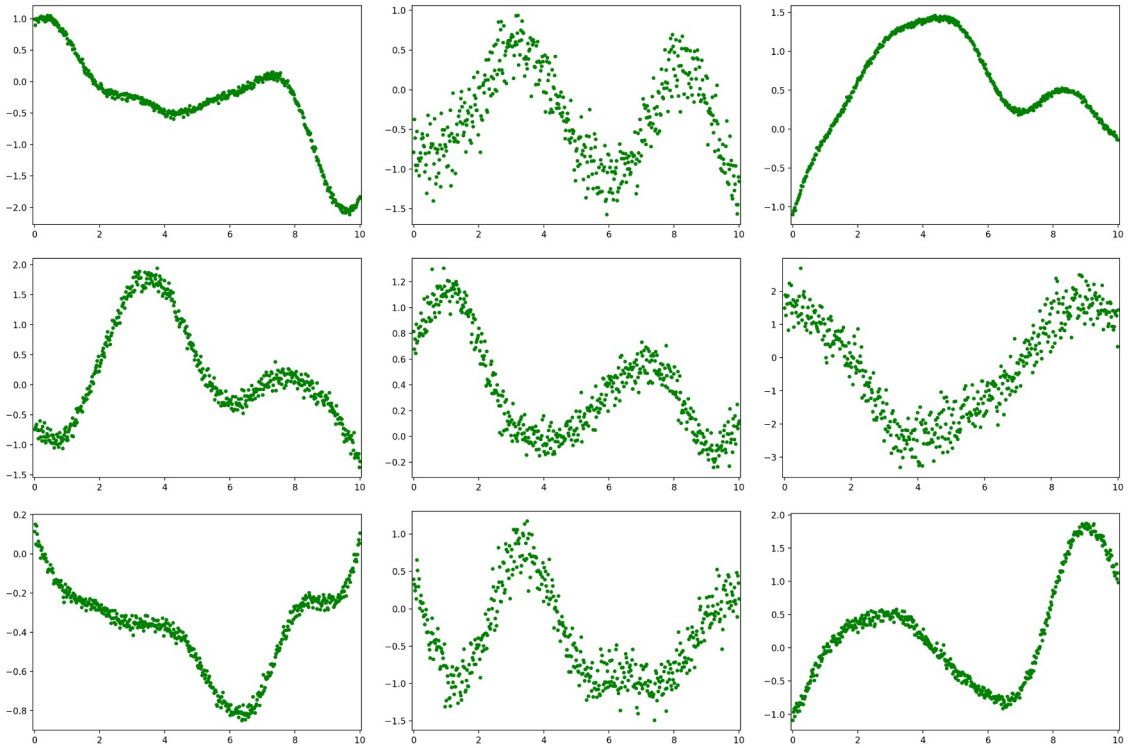
Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = (Periodic(60.0, 3.0) * Linear(0.1))

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

$$\Sigma[i,j] = \exp\left(-\frac{1}{3}\sin^2\left(\frac{2\pi}{60}\left|t[i]-t[j]\right|\right)\right)(t[i]-.1)(t[j]-.1)$$

A flexible class of time series probabilistic programs

9 Independent Executions



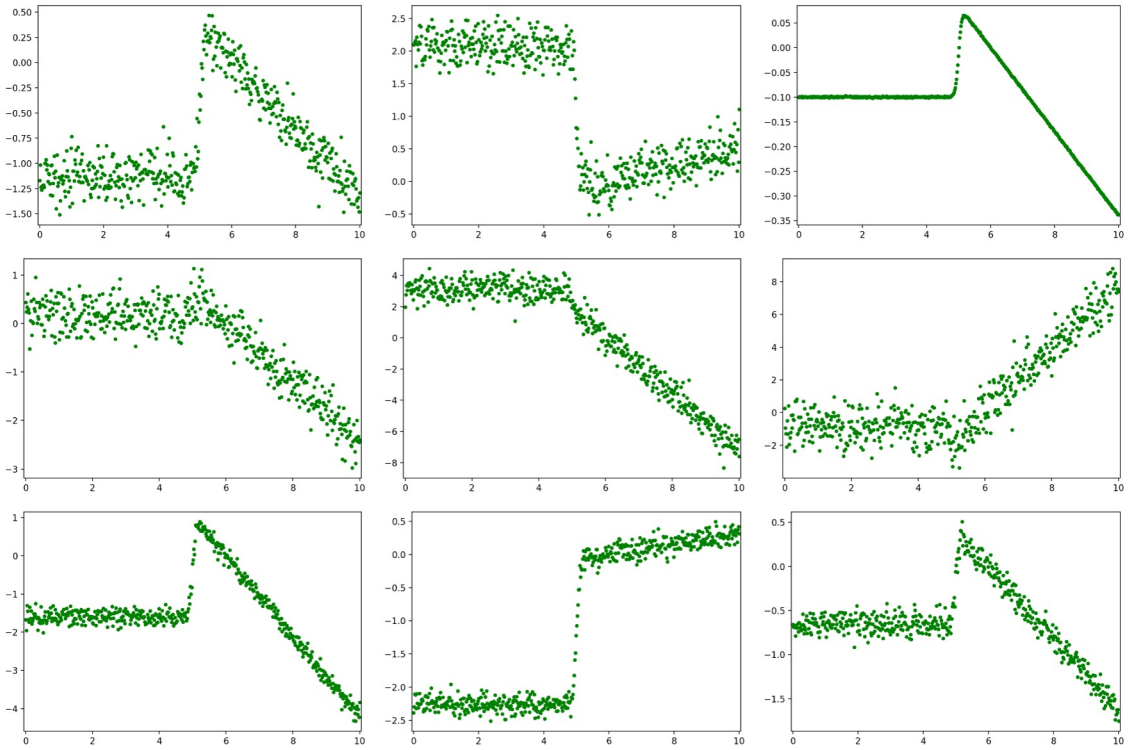
Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = SquaredExponential(3.4)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

$$\Sigma[i,j] = \exp\left(-\frac{1}{2}\frac{(t[i]-t[j])^2}{3.4}\right)$$

A flexible class of time series probabilistic programs

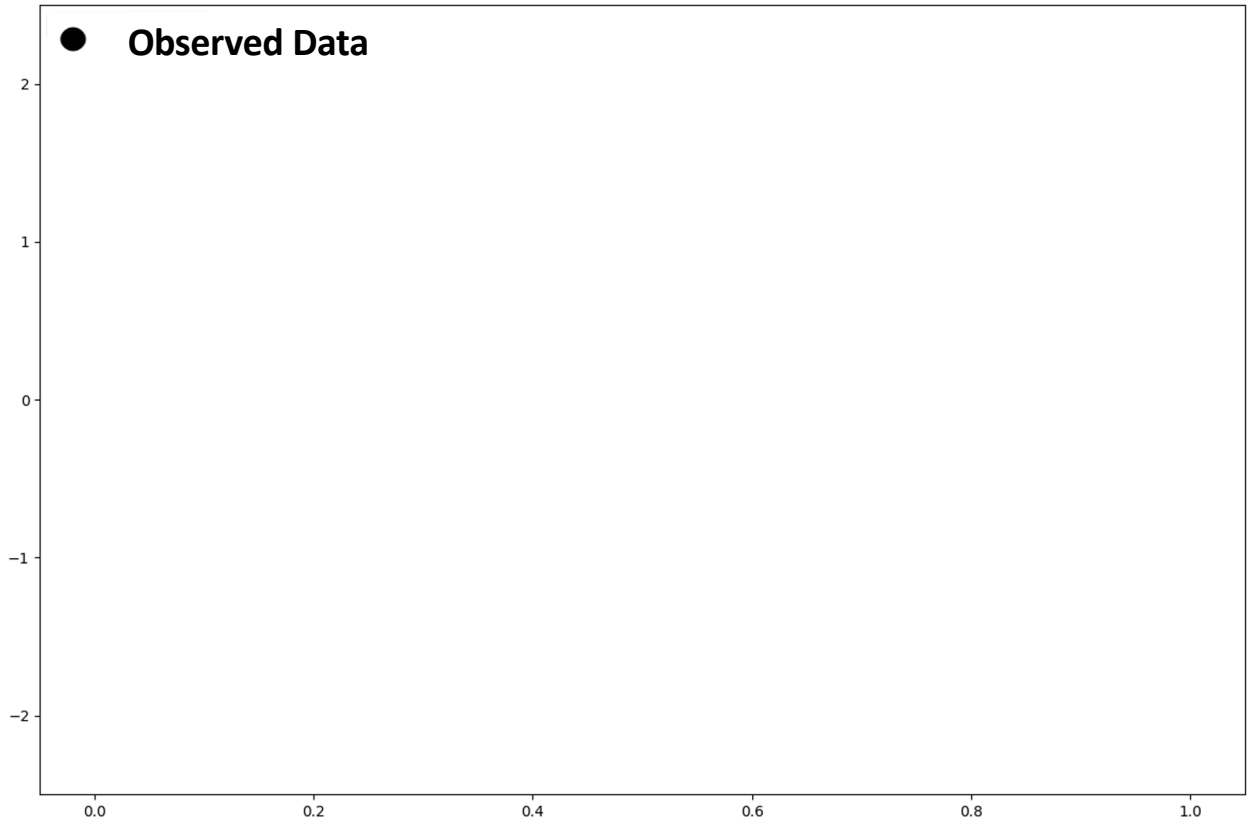
9 Independent Executions



Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = ChangePoint(5.0, Constant(2.0), Linear(6.0))

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ gamma(1,1)

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

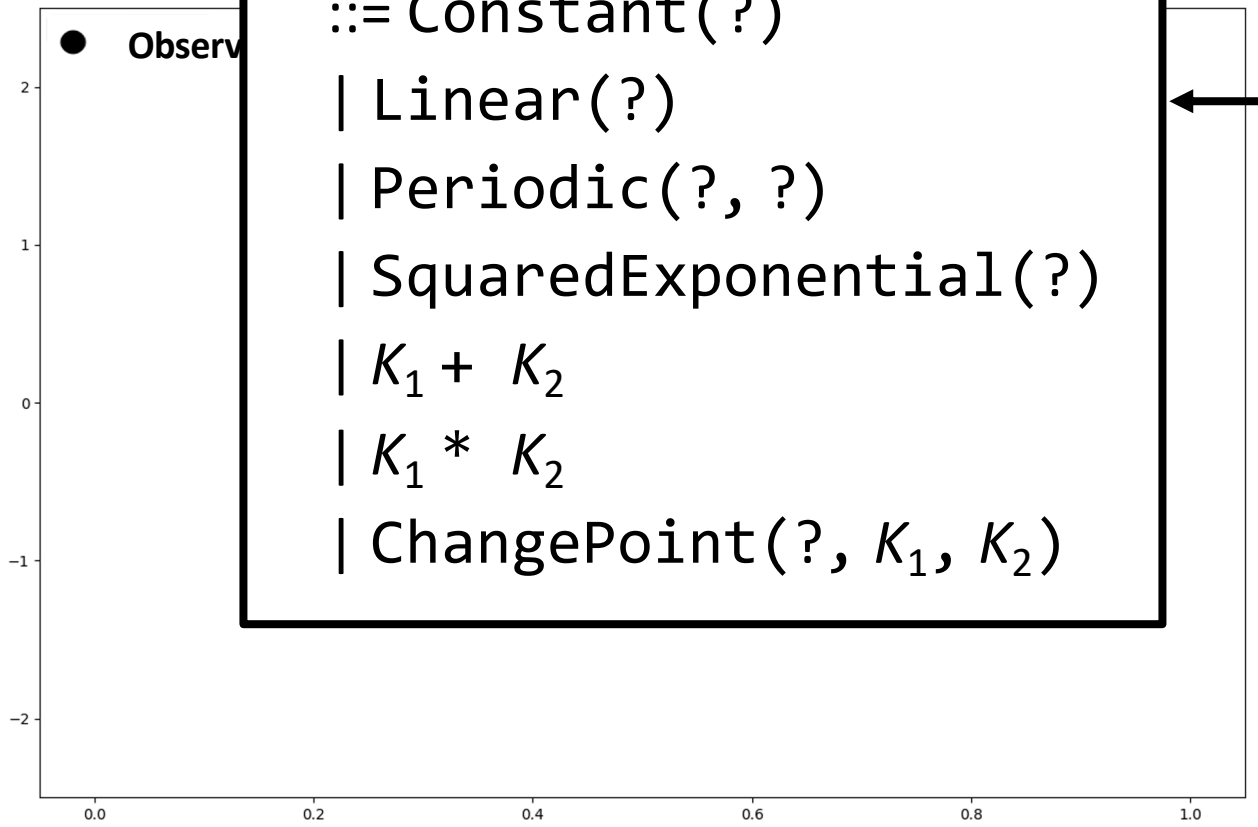
A flexible class of time series probabilistic programs



Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = ???????

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ ???

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

A flexible class of time series probabilistic programs



$K \in$ DSL_programs

::= Constant(?)

| Linear(?)

| Periodic(?, ?)

| SquaredExponential(?)

| K_1 + K_2

| K_1 * K_2

| ChangePoint(?, K_1, K_2)

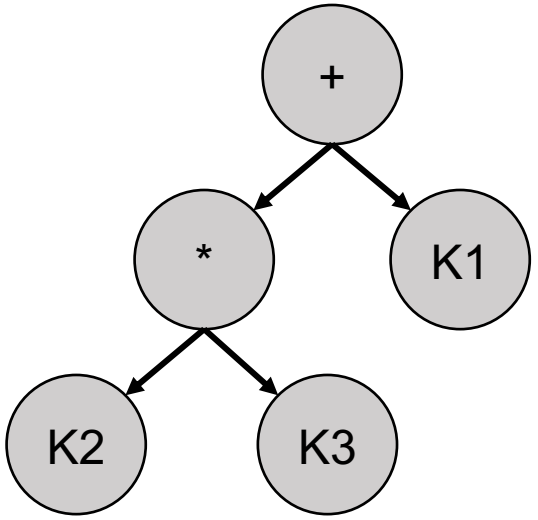
Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = ???????

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ ~ ???

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

The Kernel Cookbook. David Duvenaud.
https://www.cs.toronto.edu/~duvenaud/cookbook/

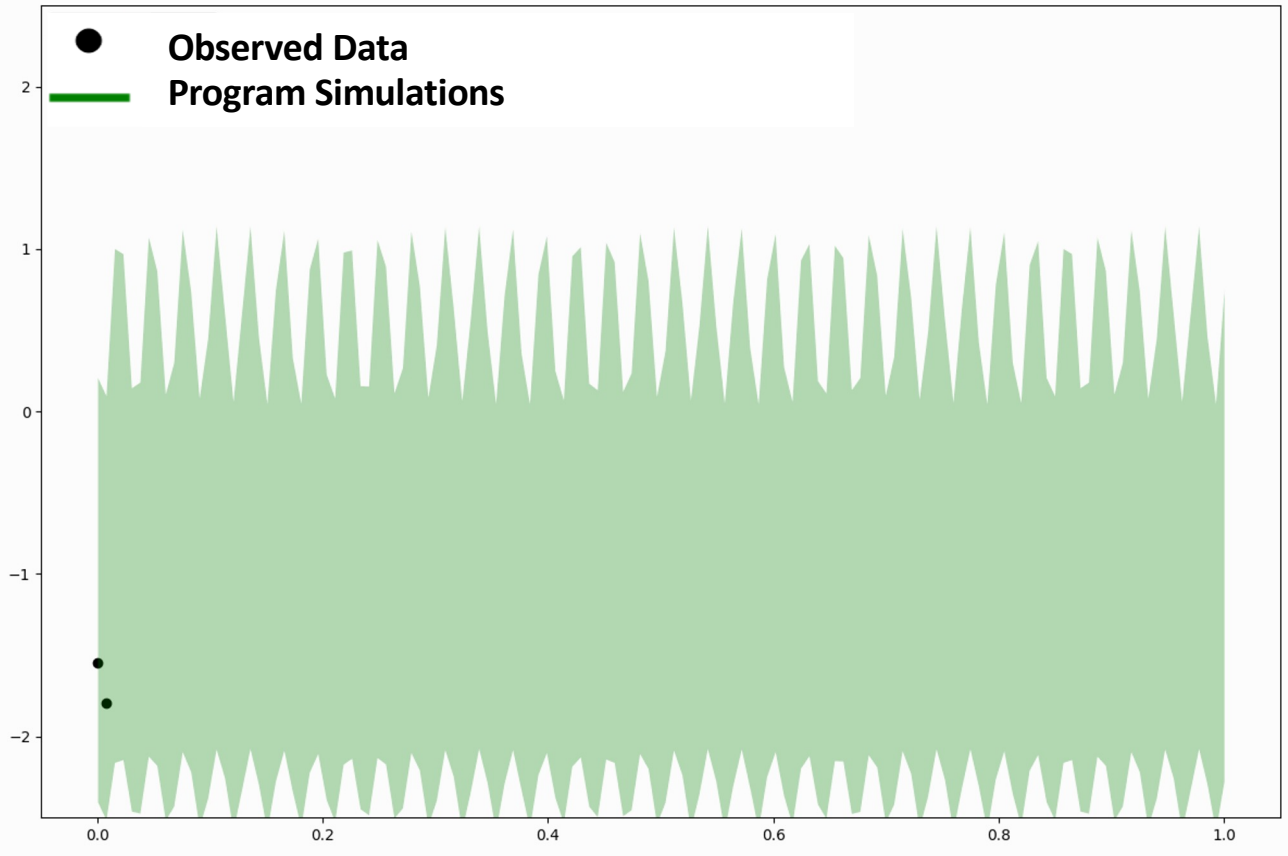
Sampling programs in a tree structure

Probabilistic context free grammar (PCFG) prior



K2 * K3 + K1

Online learning in a time series DSL

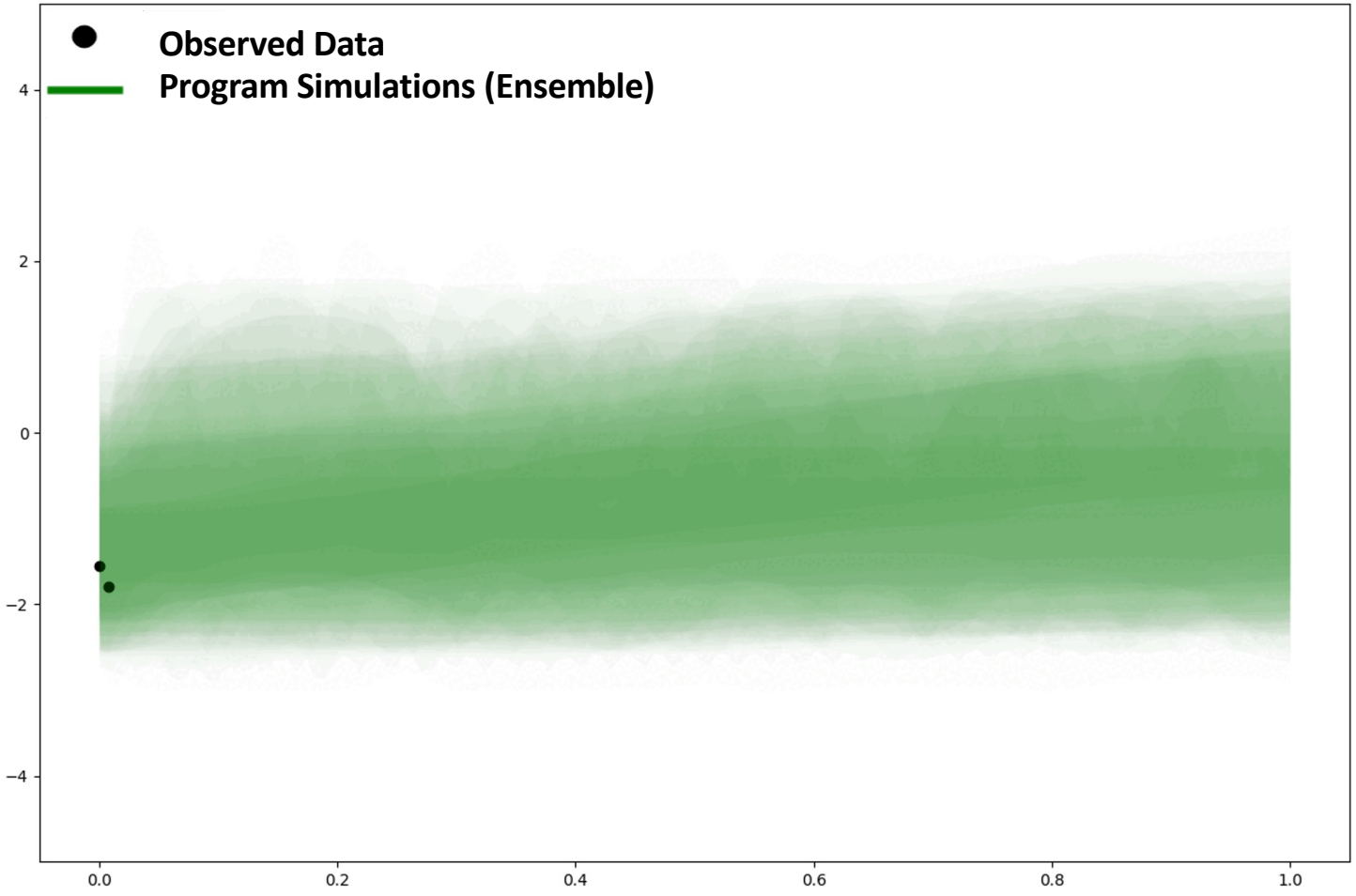


Learned Probabilistic Program

```
@gen function gaussian_process_DSL_model(t::Vector{Float64})

  DSL_program = Periodic(0.6945, 0.0203)

  Σ = [
    [DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)
  ]

  ϵ = 0.6724

  x = Vector{Float64}(undef, length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
    μ = Σ[i,1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * x[1:i-1]
    σ² = Σ[i,i] .- Σ[i, 1:i-1]' * Σ[1:i-1, 1:i-1]^-1 * Σ[i, 1:i-1]
    x[i] ~ normal(μ, σ² + ϵ)
  end

return x
end
```

Online learning in a time series DSL

Ensemble of Learned Probabilistic Programs



A well-known baseline for real-world data modeling

PROPHET

Docs GitHub

Forecasting at scale.

Prophet is a forecasting procedure implemented in R and Python. It is fast and provides completely automated forecasts that can be tuned by hand by data scientists and analysts.

INSTALL PROPHET | GET STARTED IN R | GET STARTED IN PYTHON | READ THE PAPER

THE AMERICAN STATISTICIAN
2018, VOL. 72, NO. 1, 37–45
https://doi.org/10.1080/00031305.2017.1380080

Forecasting at Scale

Sean J. Taylor and Benjamin Letham

Facebook, Menlo Park, CA

Citation: 2391

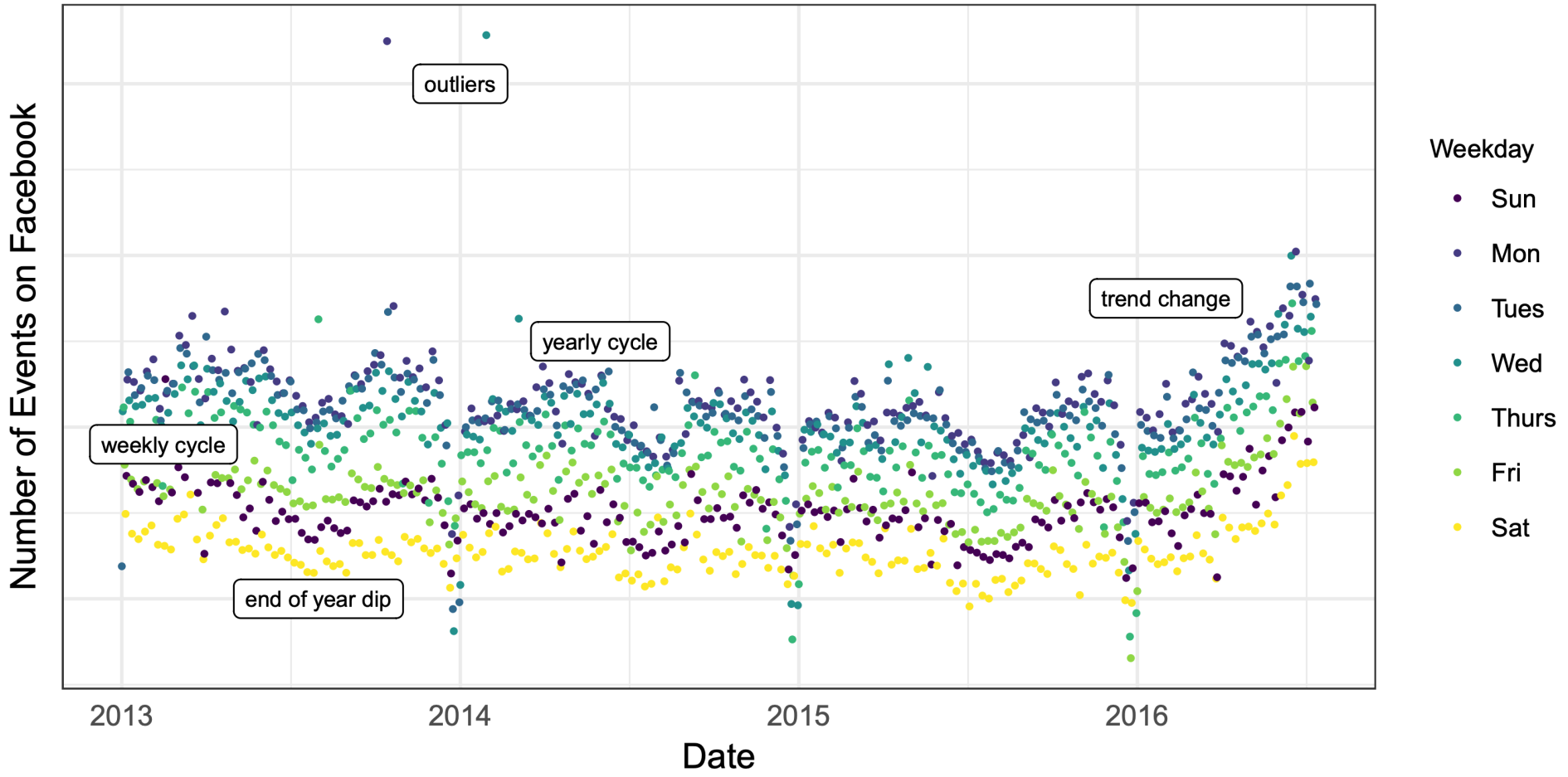
A well-known baseline for real-world data modeling

Trend (non-periodic change)

Seasonality (periodic change)

Effect of holidays

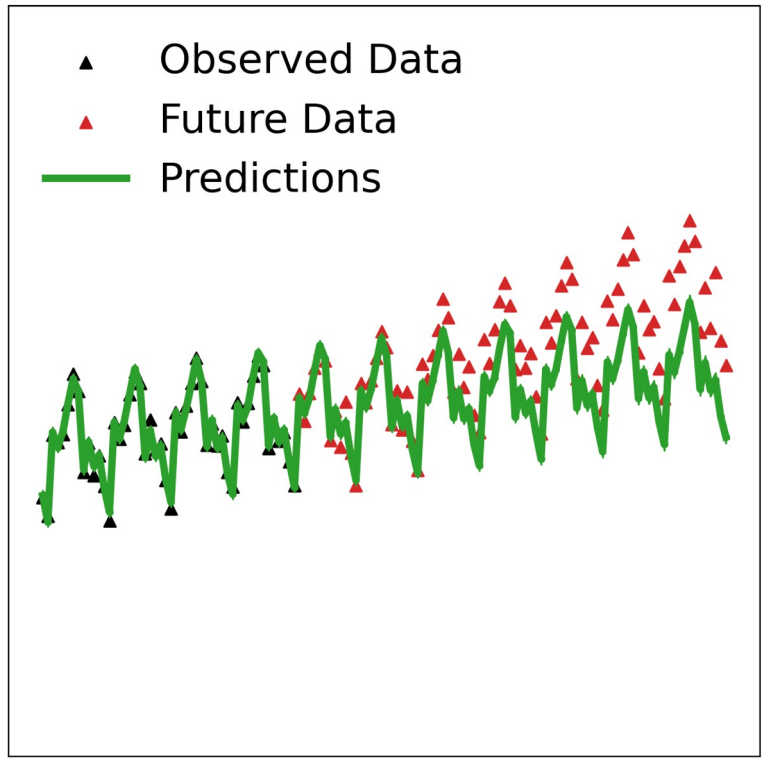
$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$



Structure learning improves robustness and captures uncertainty

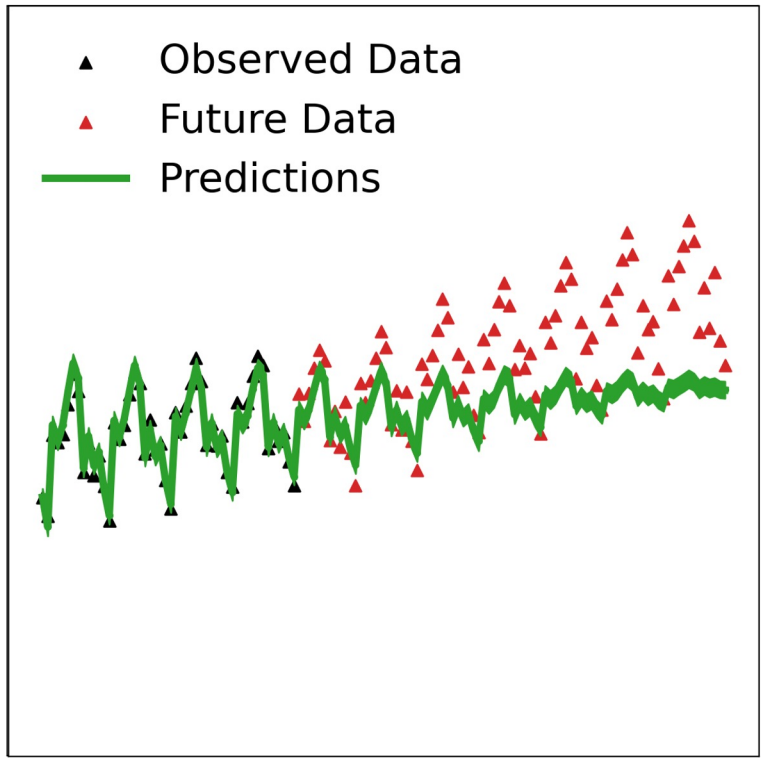
(amplitude x seasonality)

Facebook Prophet (Default)
Additive Seasonality



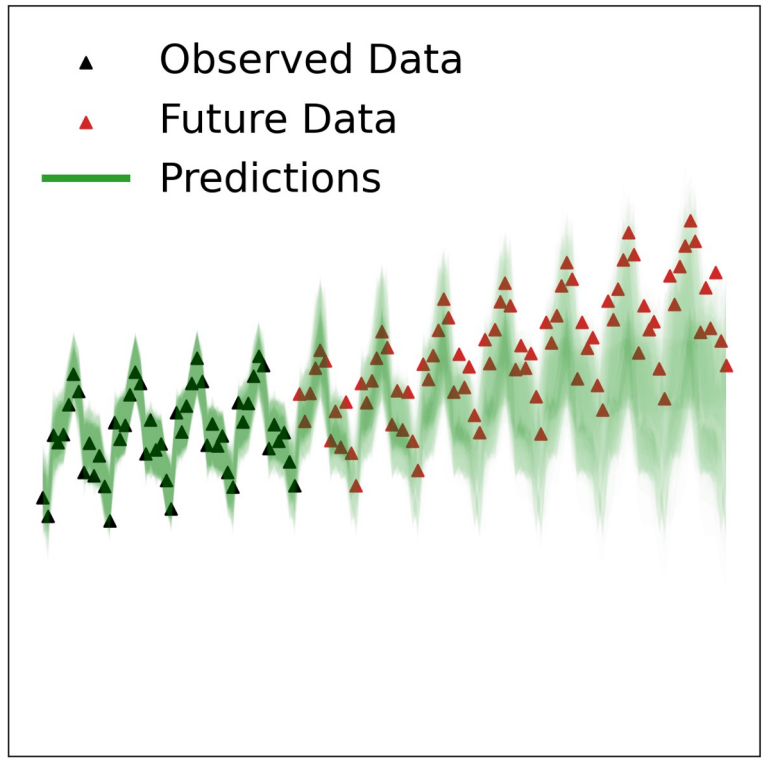
2.1 sec

Facebook Prophet (Custom)
Multiplicative Seasonality



2.1 sec

Probabilistic Programs
Learned Structure



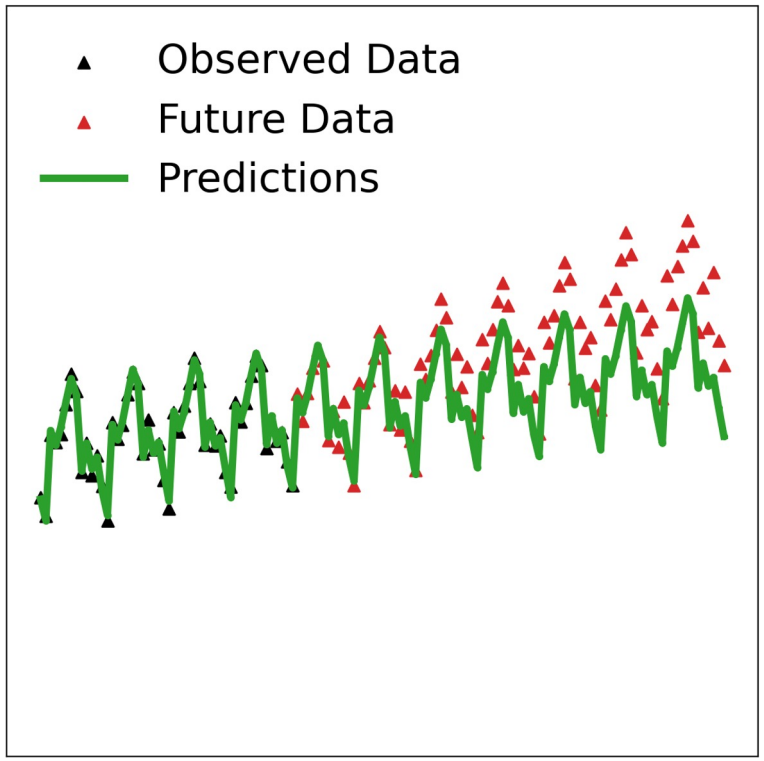
1.9 sec

facebook PROPHET Taylor and Letham. *Amer. Stat.* 2018

Structure learning improves robustness and captures uncertainty

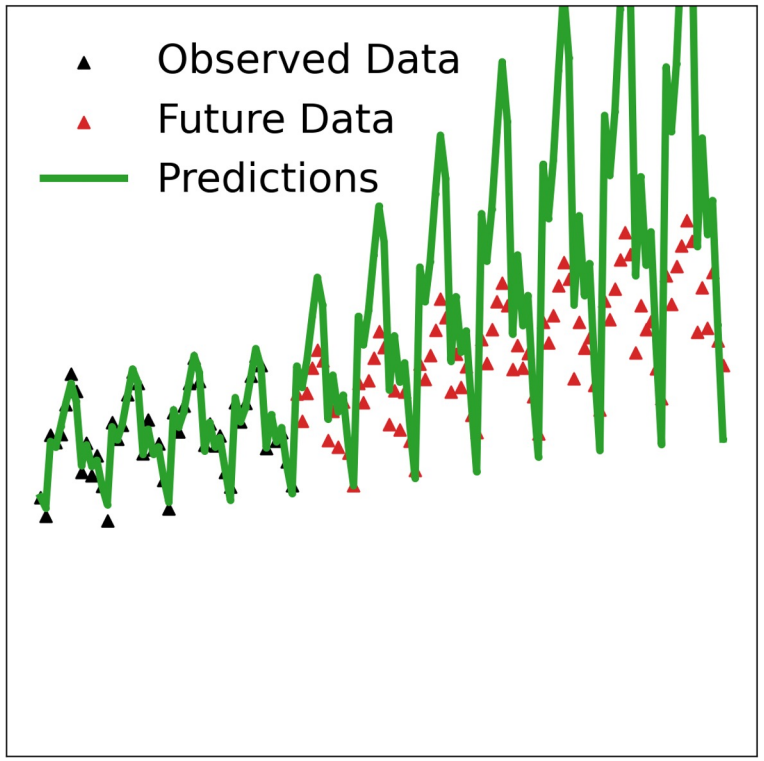
(amplitude x seasonality)

Neural Prophet (Default)
Additive Seasonality



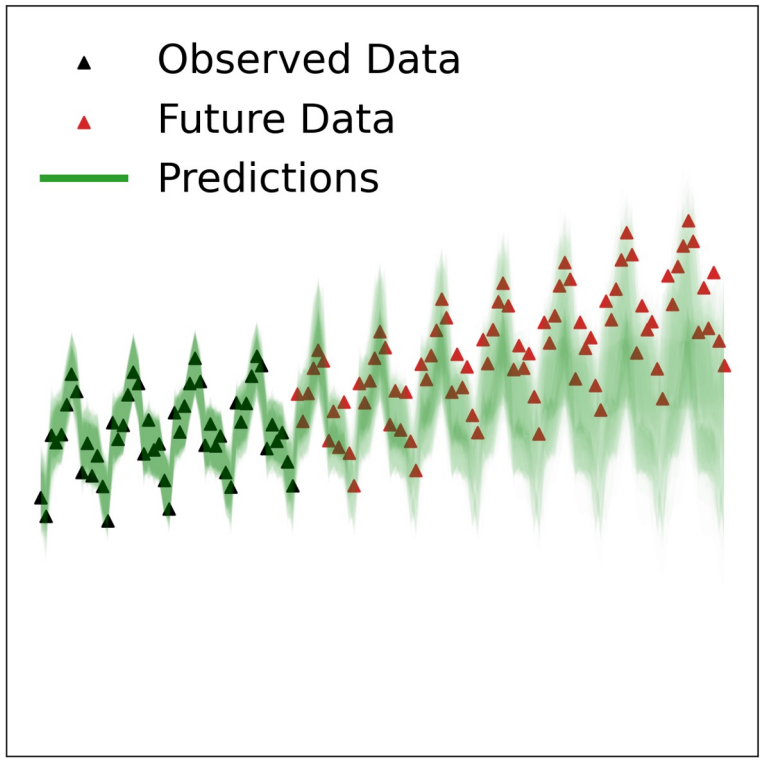
4.5 sec

Neural Prophet (Custom)
Multiplicative Seasonality



4.1 sec

Probabilistic Programs
Learned Structure

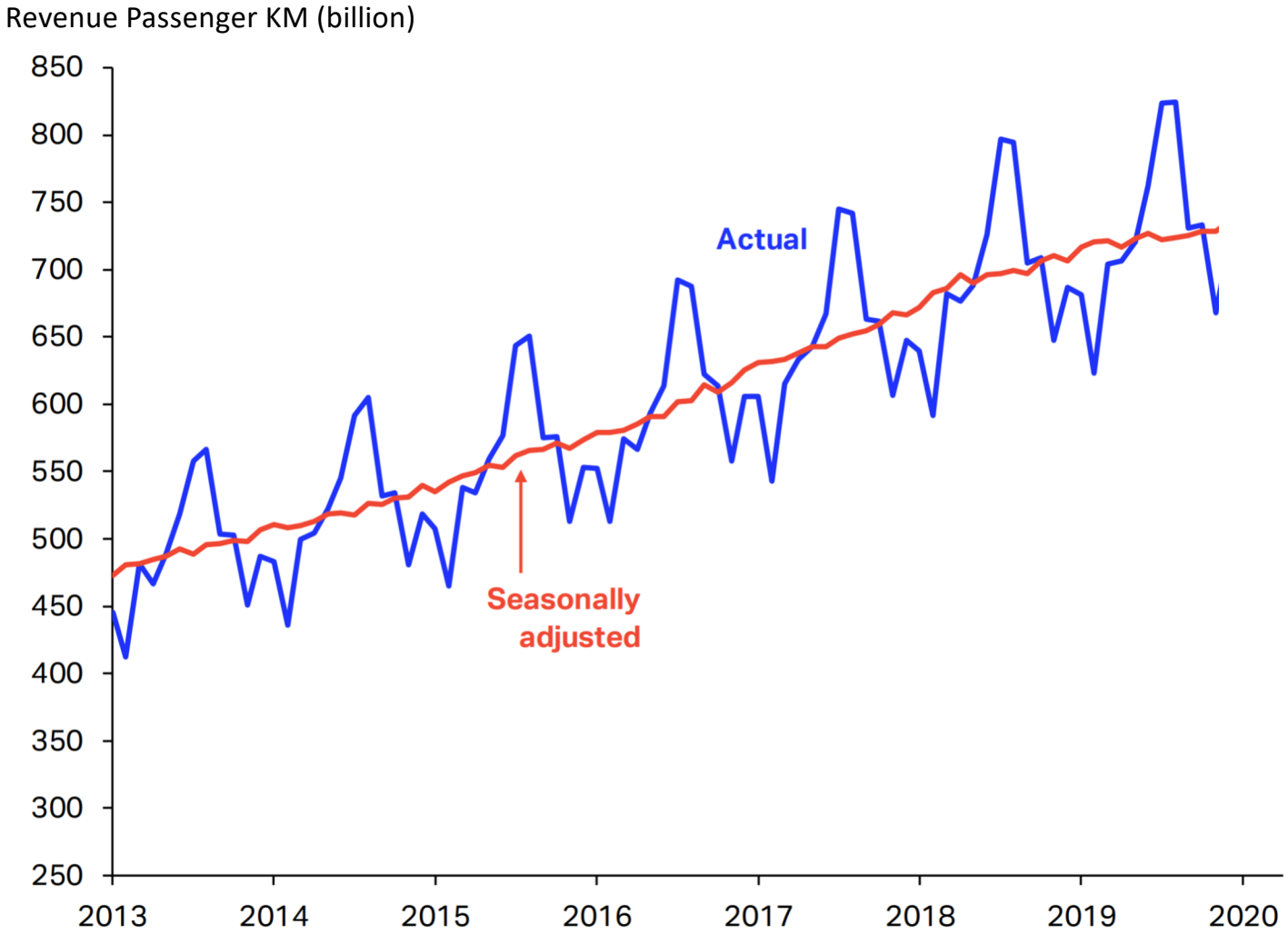


1.9 sec

Neural Prophet Triebe et al. 2021. https://neuralprophet.com

Learned dynamics of observed airline data

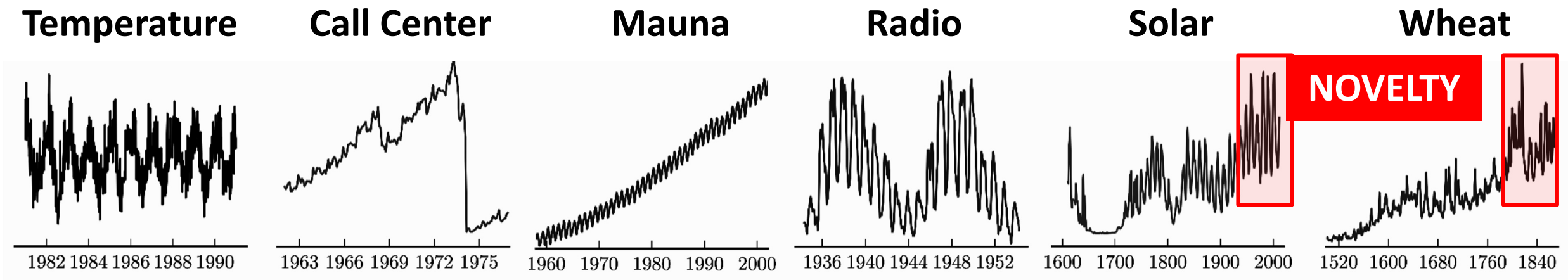
Airline Passenger Volume (2013 – 2020)



US Department of Transportation: Bureau of Transportation Statistics.

overall linear trend
periodic fluctuations
increasing amplitude

Learned time series programs produce accurate forecasts

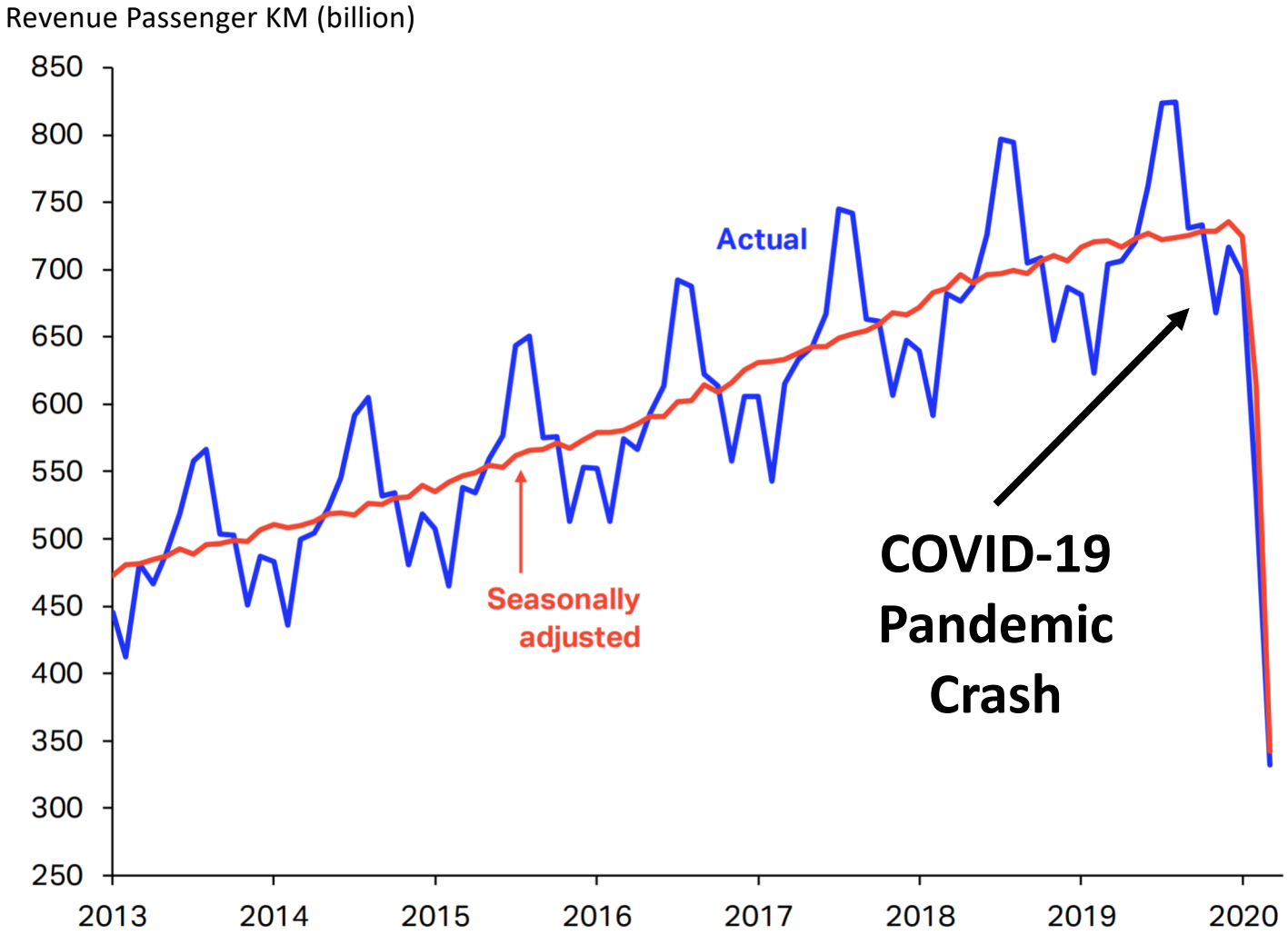


Standardized Root Mean Squared Error (RMSE)

	Airline	Temperature	Call	Mauna	Radio	Solar	Wheat
Learned Probabilistic Programs	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	1.47	1.50
Gaussian Process (Squared Exp. Kernel)	2.01	1.70	4.26	1.54	2.03	1.63	1.37
Autoregressive Integrated Moving Average	1.32	1.85	2.44	1.09	2.08	**1.0**	1.41
Facebook Prophet	1.83	2.00	5.61	1.23	3.09	1.73	1.29
Hierarchical HDP-Hidden Markov Model	4.61	1.77	2.26	14.77	1.19	3.49	1.89

Handling extreme novelty

Airline Passenger Volume (2013 – 2020)

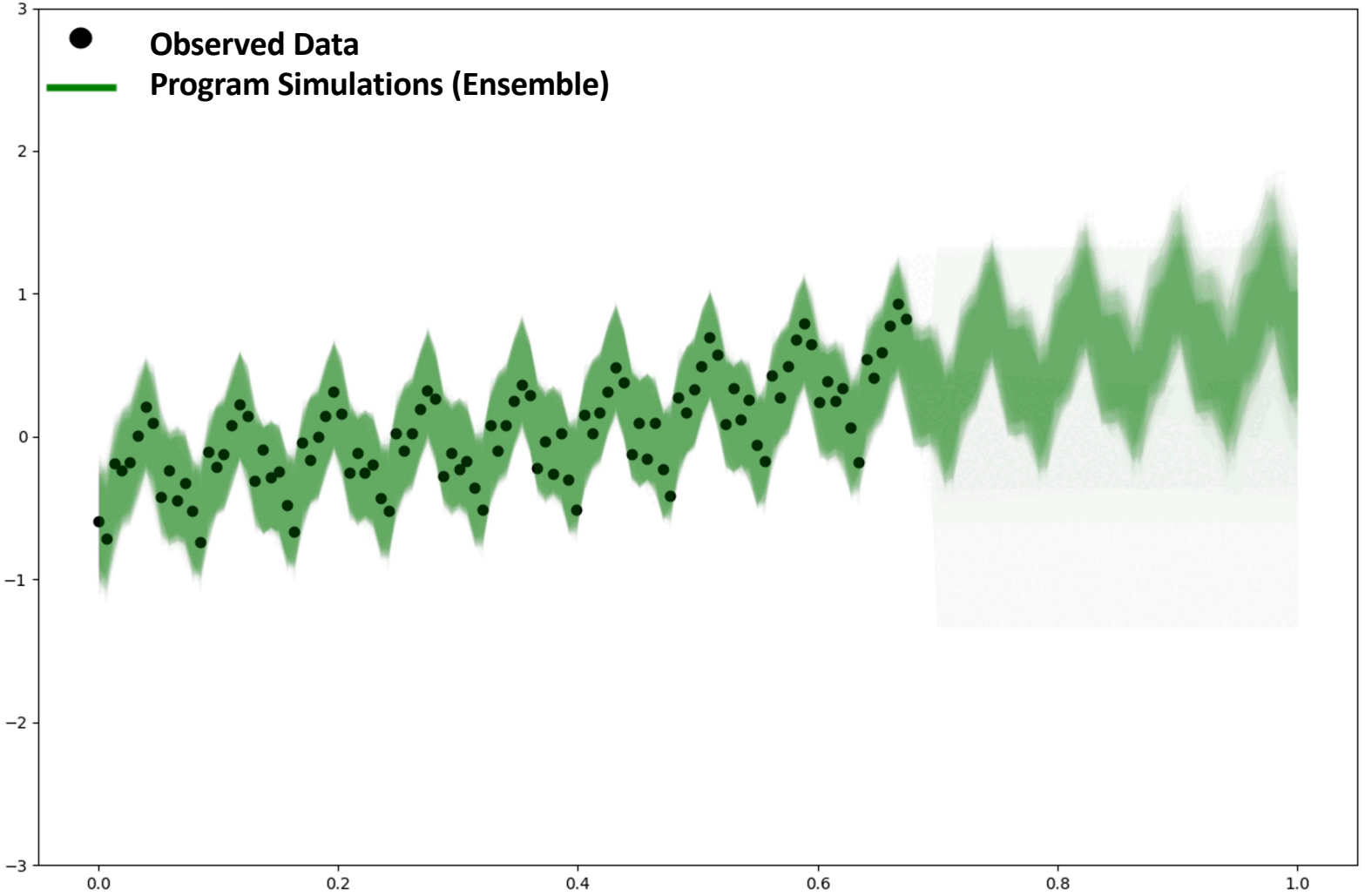


US Department of Transportation: Bureau of Transportation Statistics.

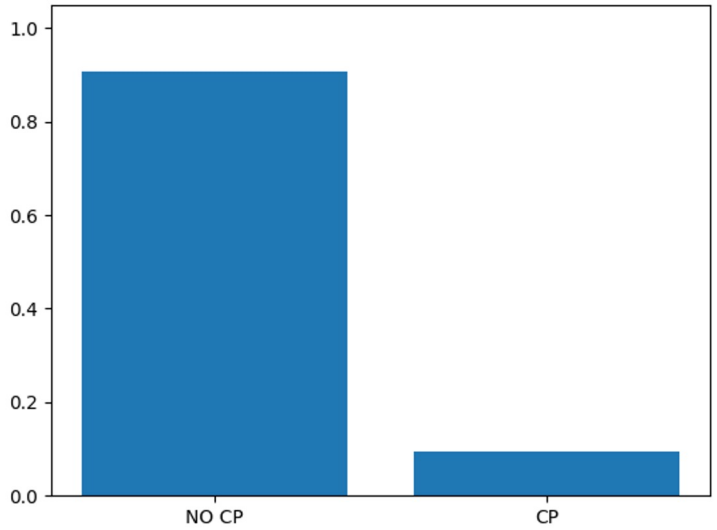
overall linear trend
periodic fluctuations
increasing amplitude

changepoint!

Handling extreme novelty

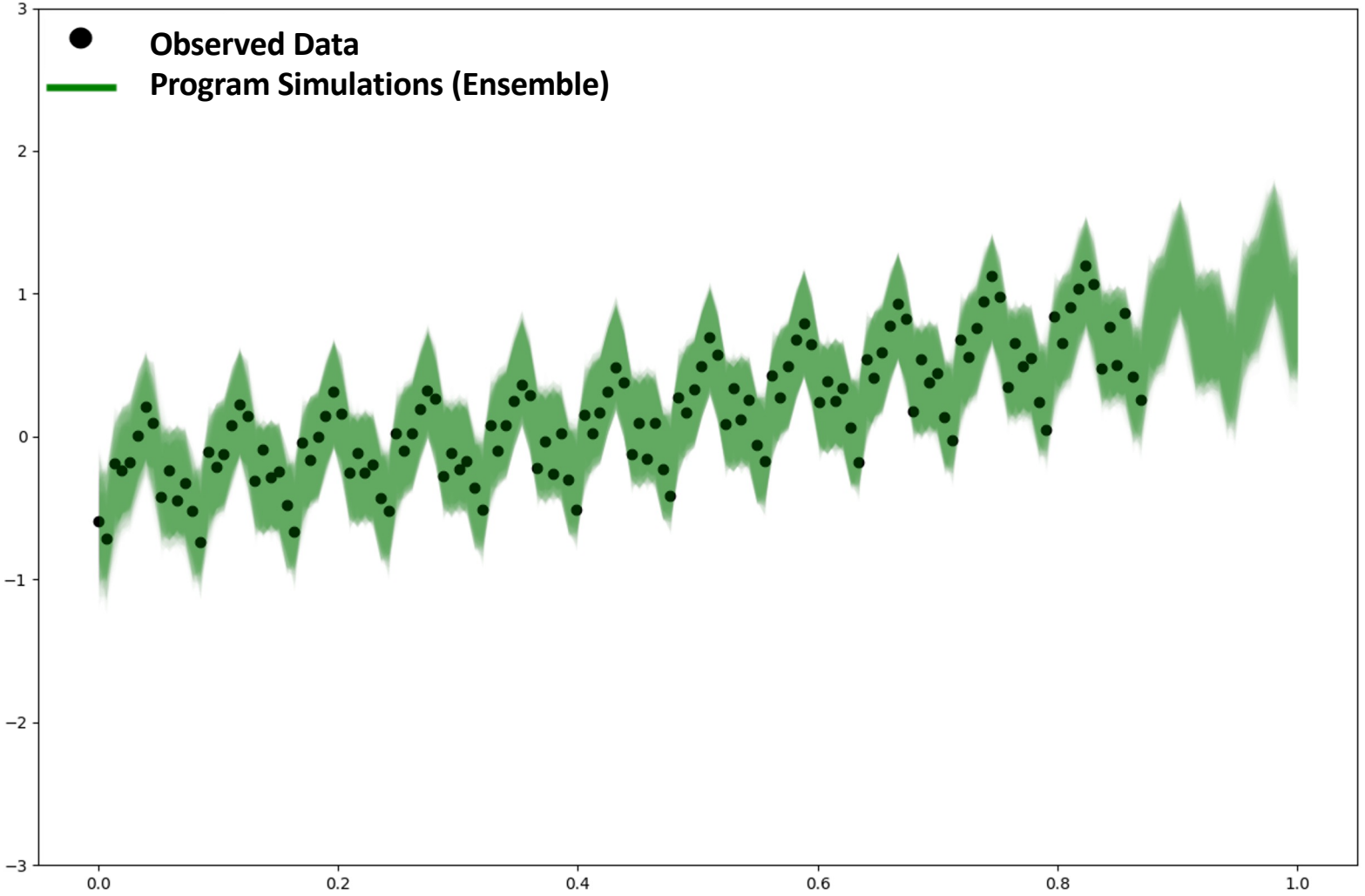


Changepoint Probability

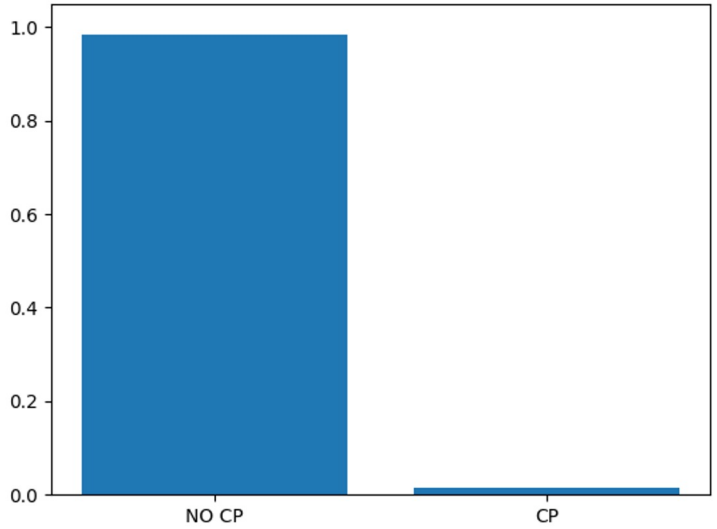


Source: US Department of Transportation: Bureau of Transportation Statistics.

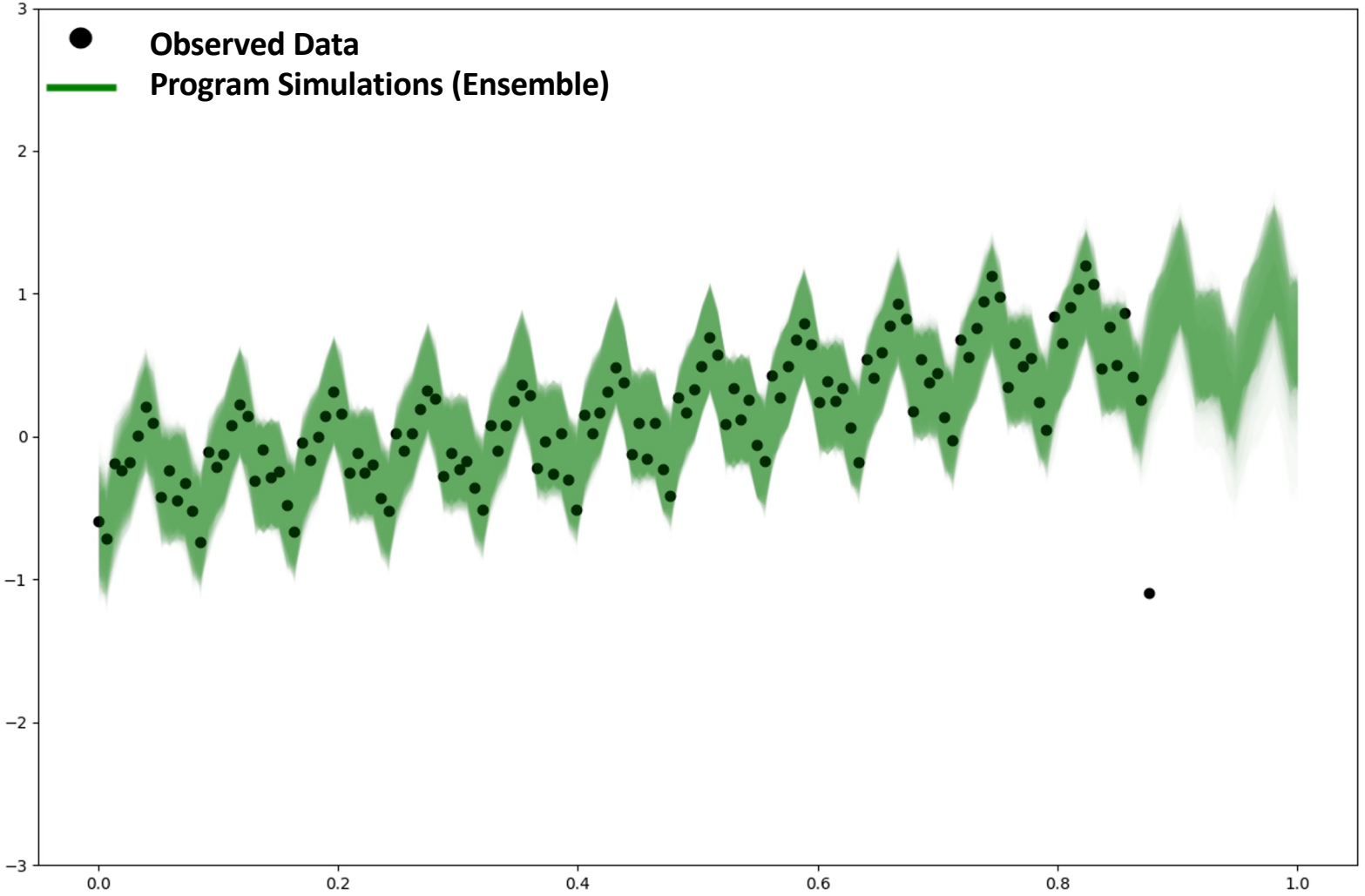
Handling extreme novelty



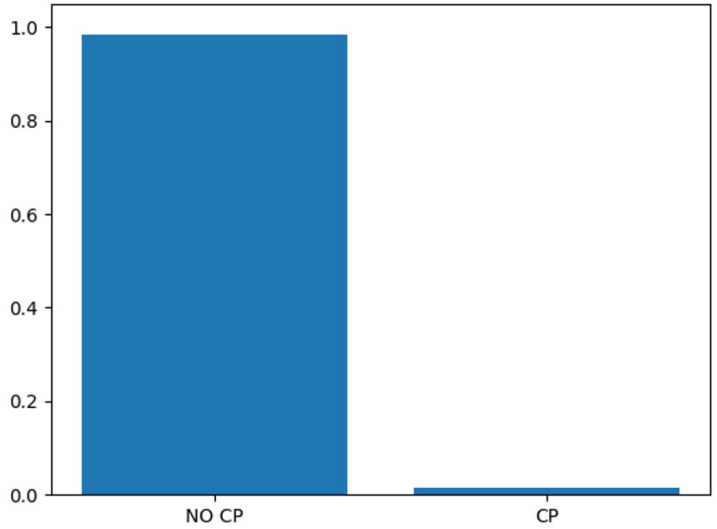
Changepoint Probability



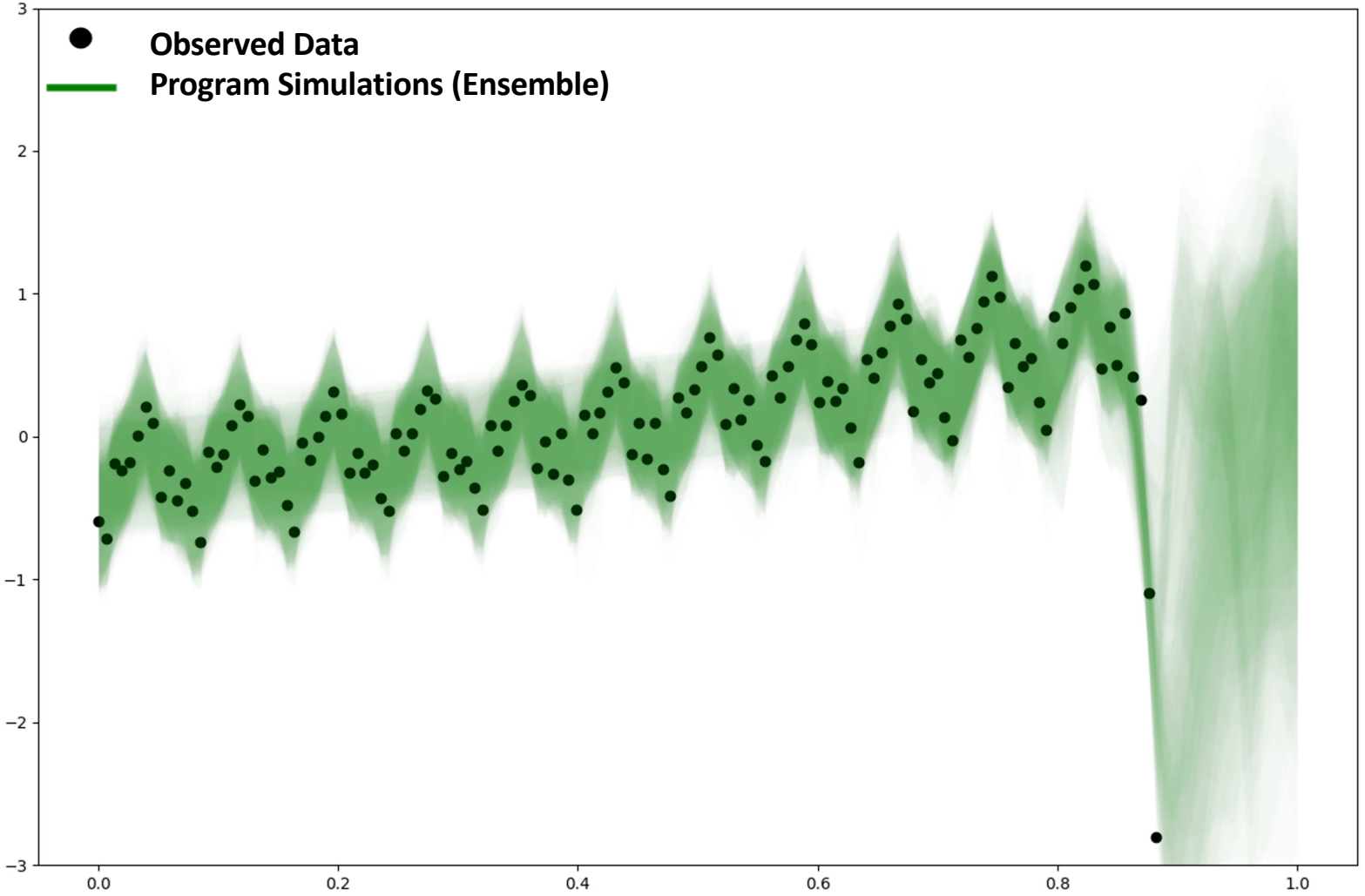
Handling extreme novelty



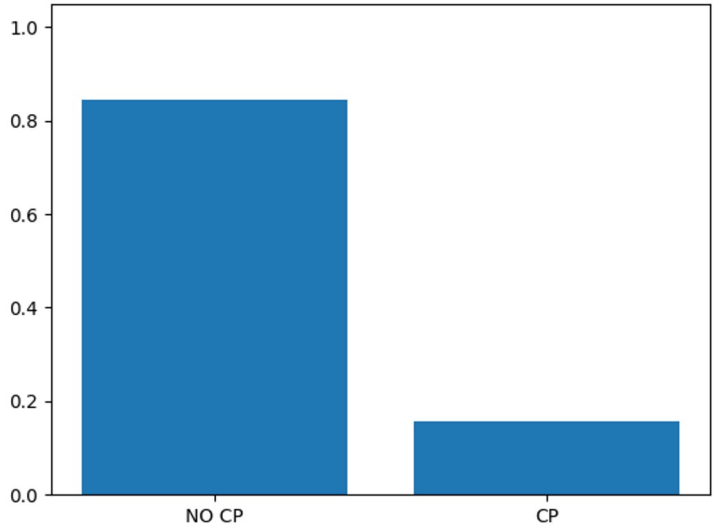
Changepoint Probability



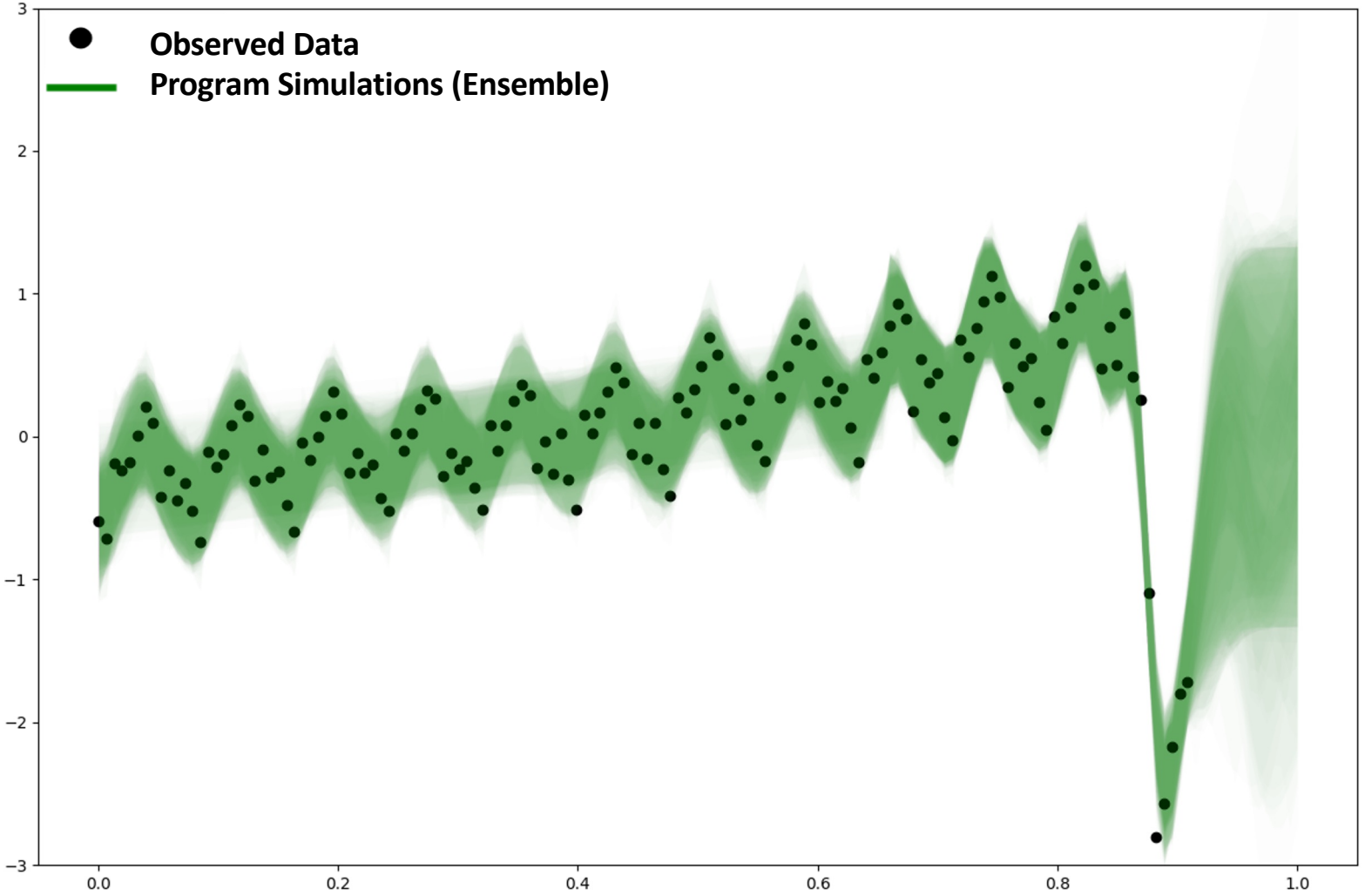
Handling extreme novelty



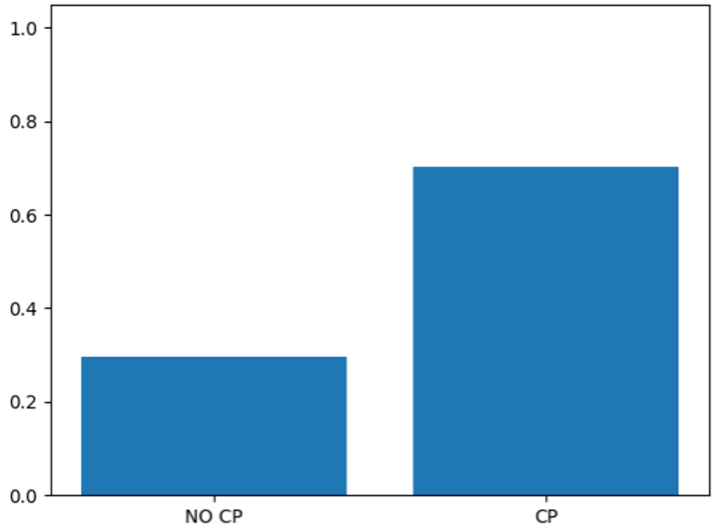
Changepoint Probability



Handling extreme novelty

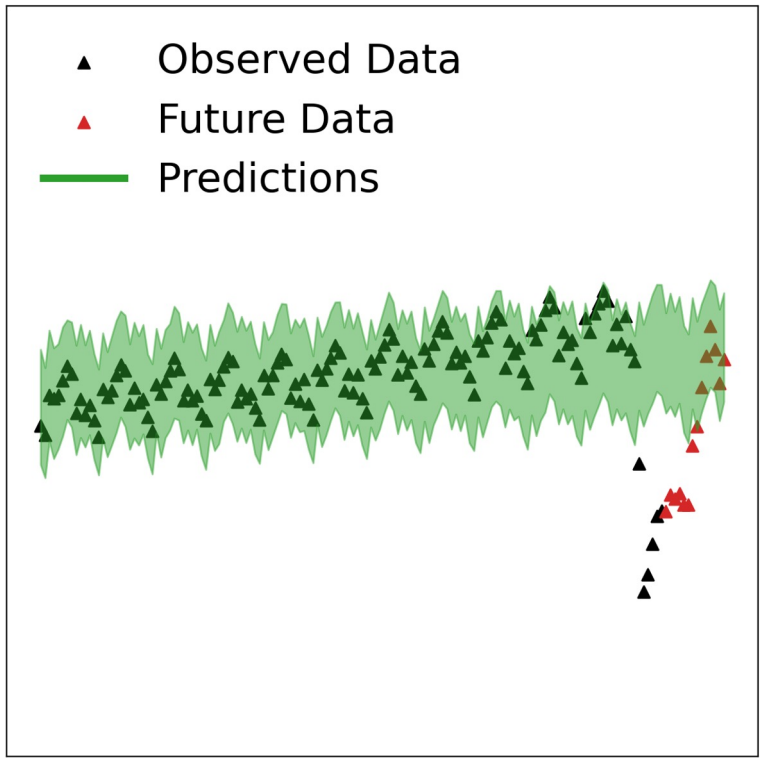


Changepoint Probability



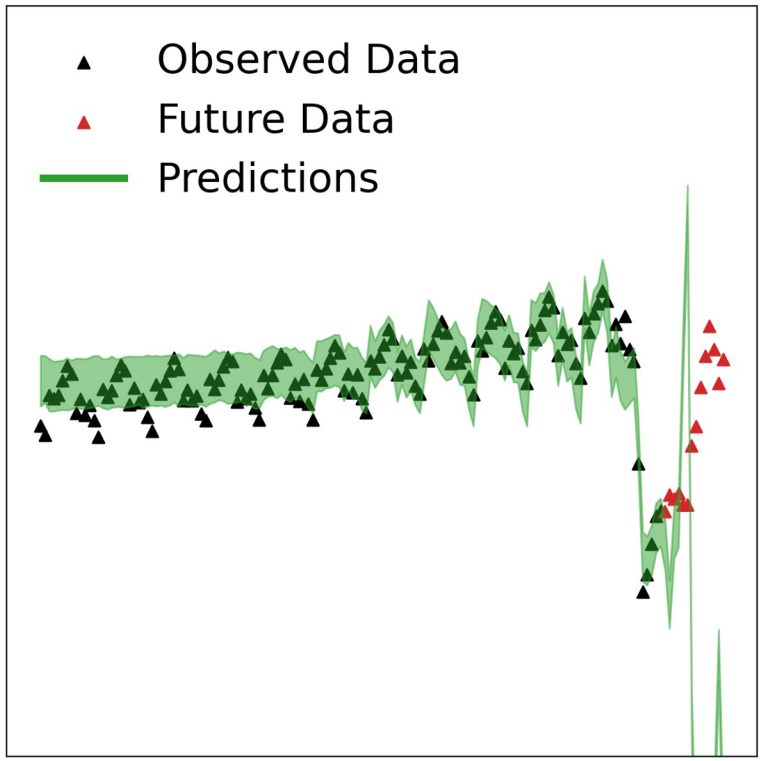
Online structure learning adapts to novelty

Facebook Prophet (Default)
No Change Point



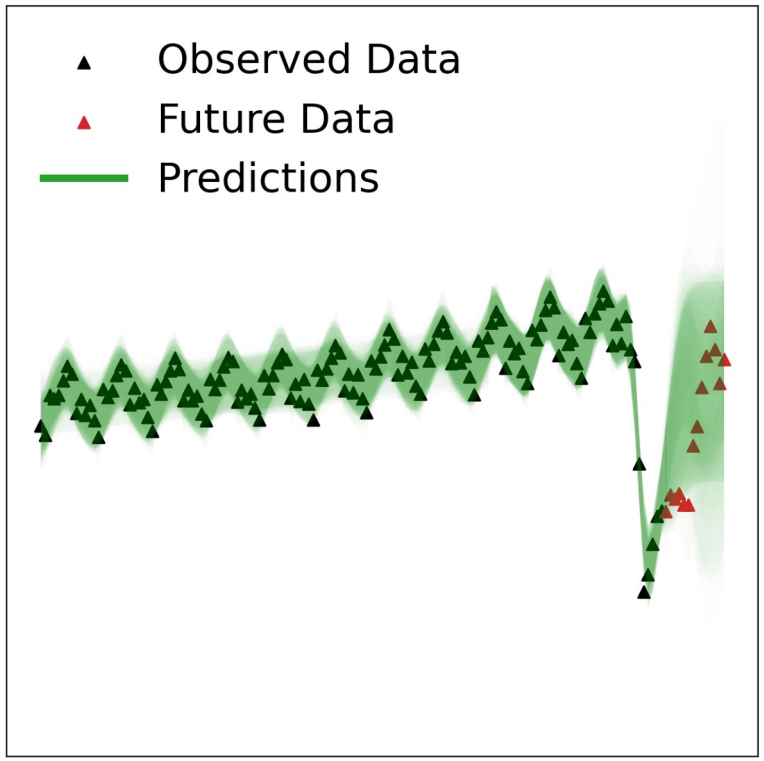
1.7 sec

Facebook Prophet (Custom)
With Change Point



2.3 sec

Probabilistic Programs
Learned Structure

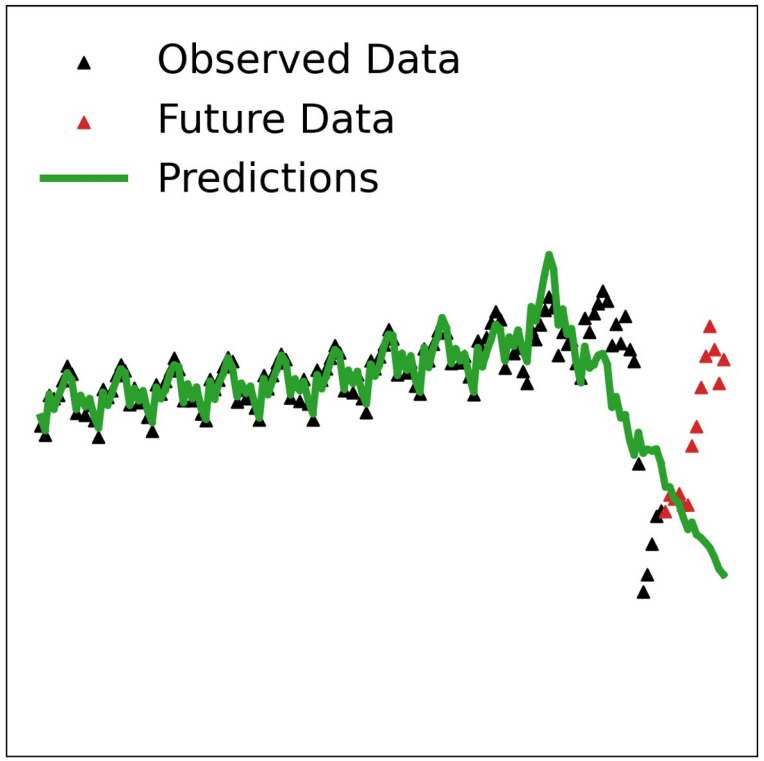


3.1 sec

facebook PROPHET Taylor and Letham. *Amer. Stat.* 2018

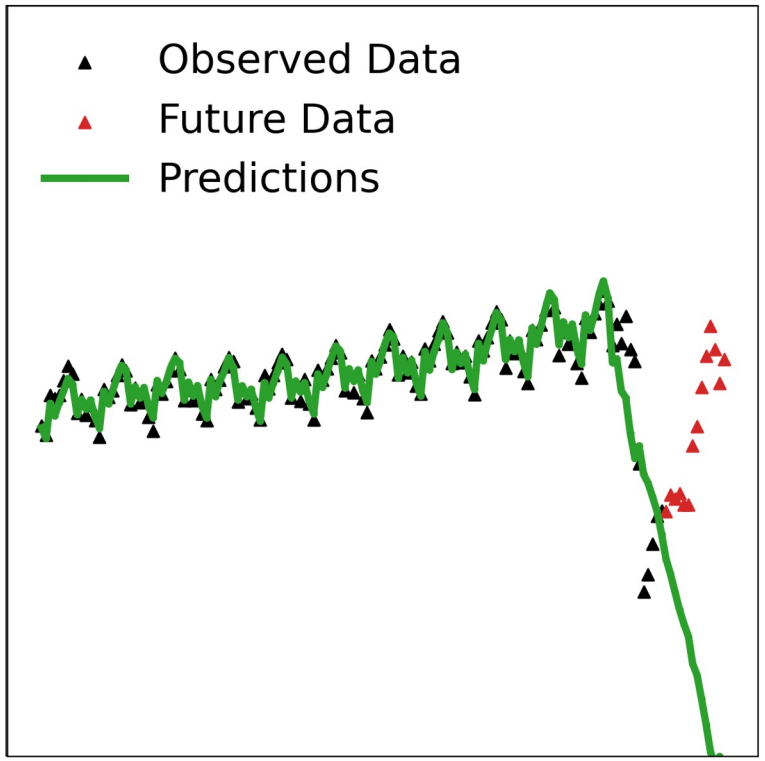
Online structure learning adapts to novelty

Neural Prophet (Default)
No Change Point



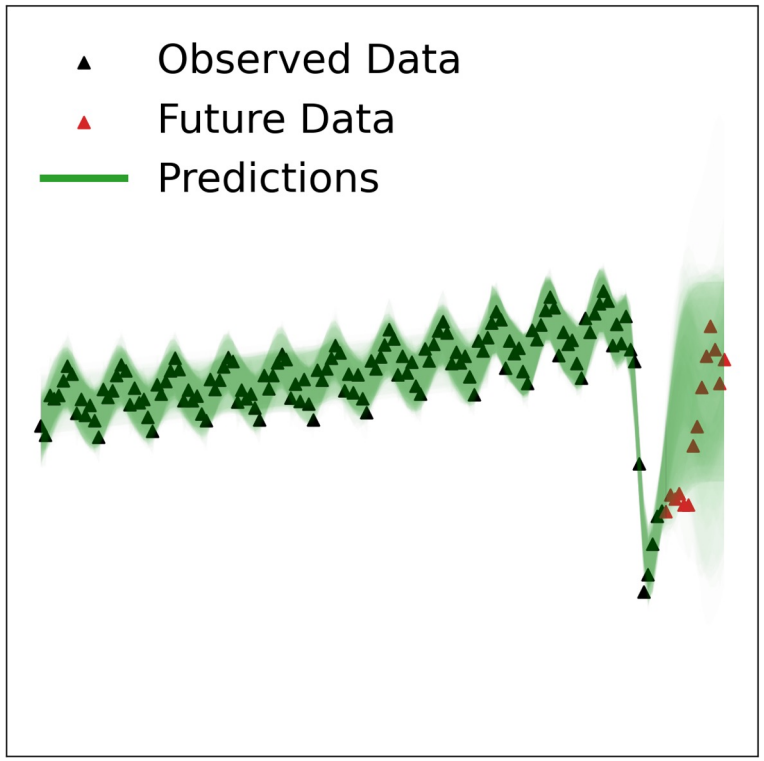
5.7 sec

Neural Prophet (Custom)
With Change Point



5.5 sec

Probabilistic Programs
Learned Structure



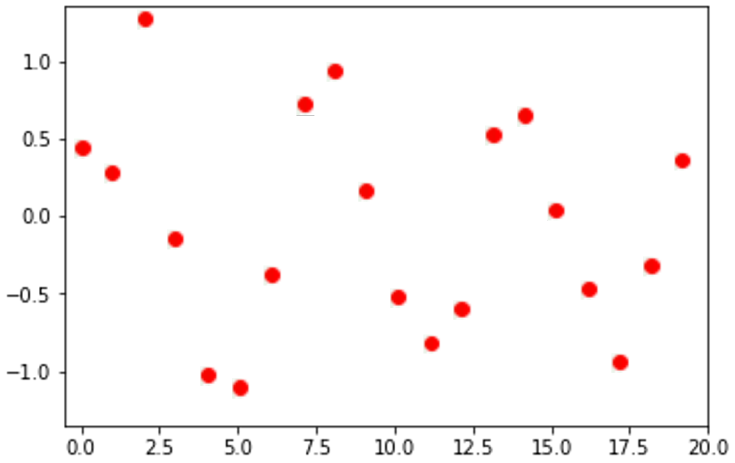
3.1 sec

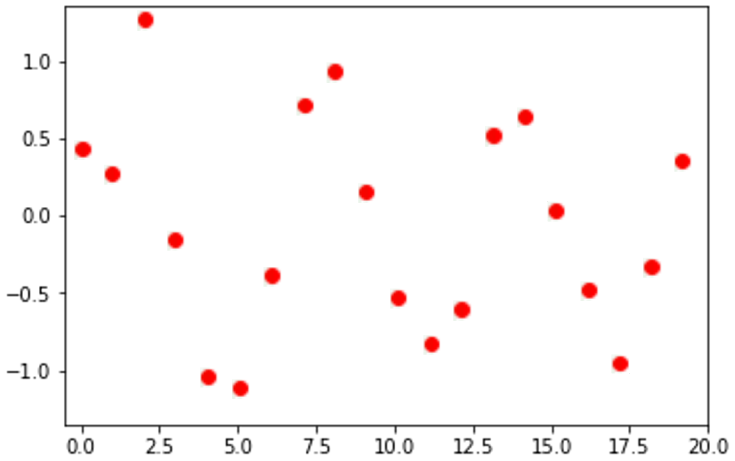
Neural Prophet Triebe et al. 2021. https://neuralprophet.com

Why maximum likelihood can fail

Suppose the goal is to find `DSL_program` that maximizes P(`data` | `DSL_program`)

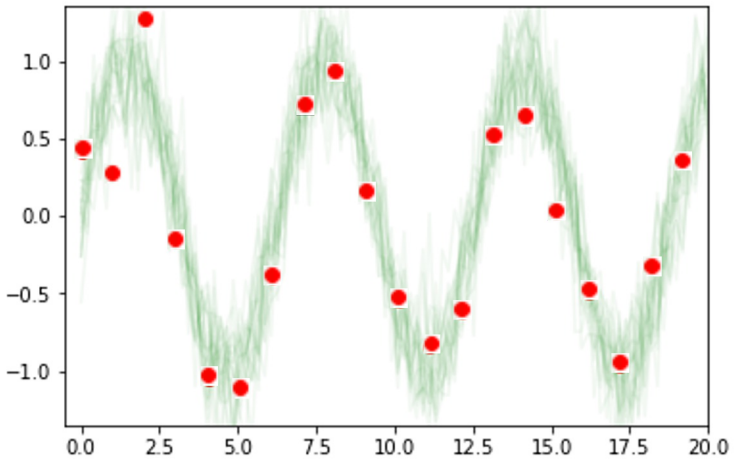
`DSL_program`

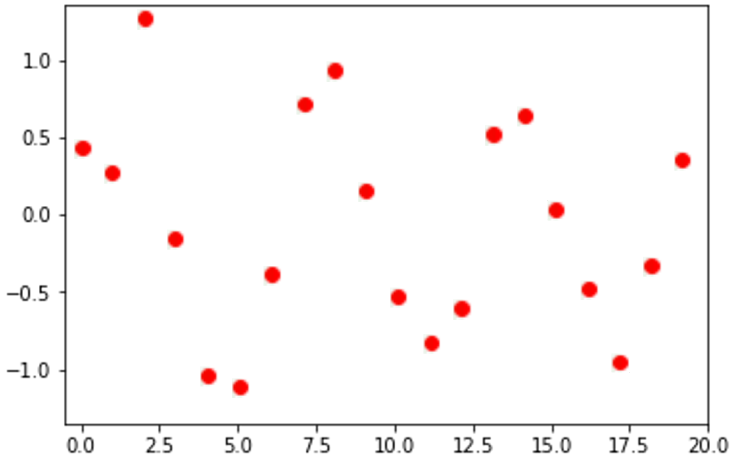




Why maximum likelihood can fail

Suppose the goal is to find `DSL_program` that maximizes P(`data` | `DSL_program`)



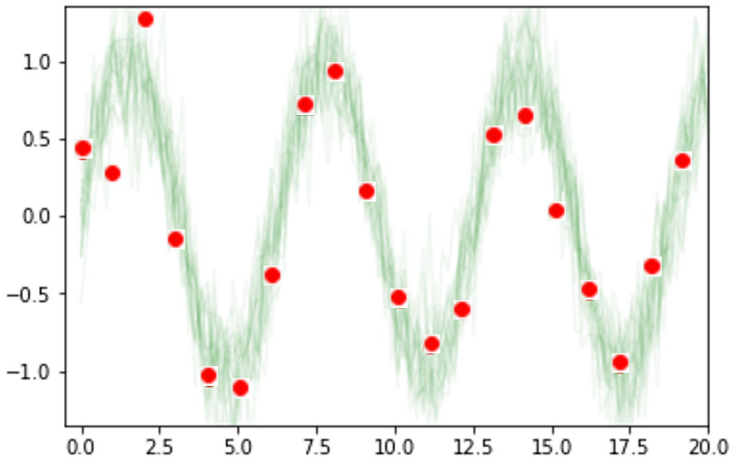


DSL_program

```
PERIODIC(2*pi, .5) + WHITE-
NOISE(.2)
```

Why maximum likelihood can fail

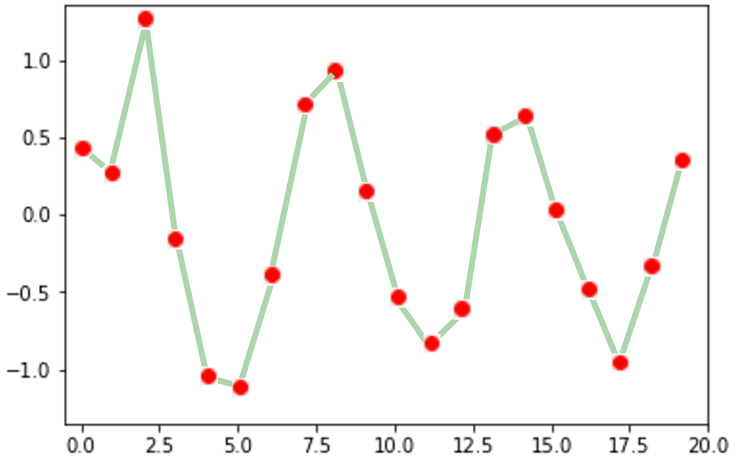
Suppose the goal is to find `DSL_program` that maximizes P(`data` | `DSL_program`)



DSL_program

```
PERIODIC(2*pi, .5) + WHITE-
NOISE(.2)
```

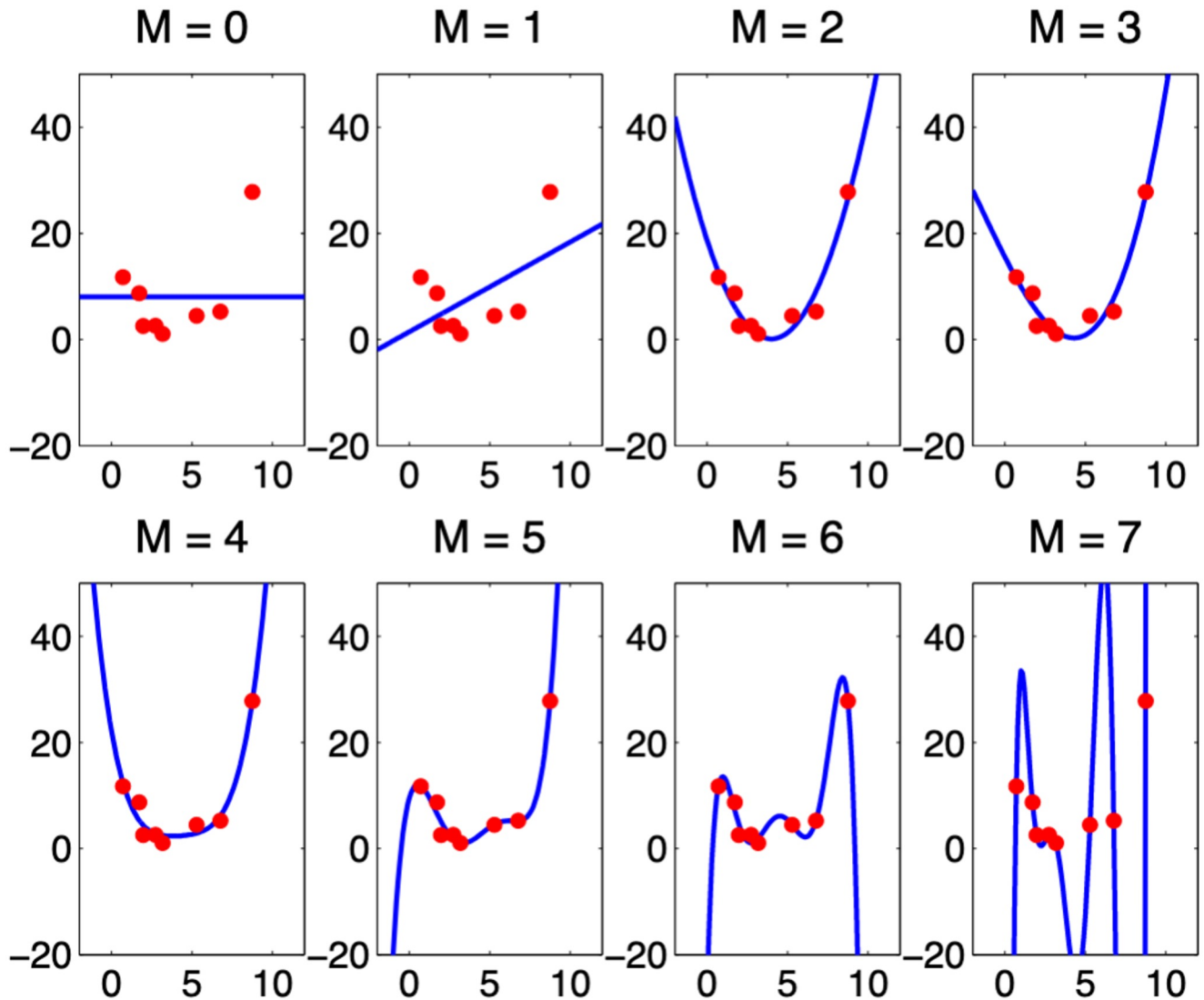
lower likelihood
useful model




```
CHANGEPOINT(t[1],
  LINEAR((x[2]-x[2])/(t[2]-t[1]) ...),
  CHANGEPOINT(t[2],
    LINEAR((x[3]-x[2])/(t[3]-t[2]) ...),
    CHANGEPOINT(t[3],
    ...
      LINEAR((x[n]-x[n-1])/(t[n]-t[n-1])))))
+ WHITE-NOISE(0.001)
```

higher likelihood
useless model

Model Selection



Bayesian Occam's Razor and Model Selection

Compare model classes, e.g. m and m', using posterior probabilities given $\mathcal{D}$:

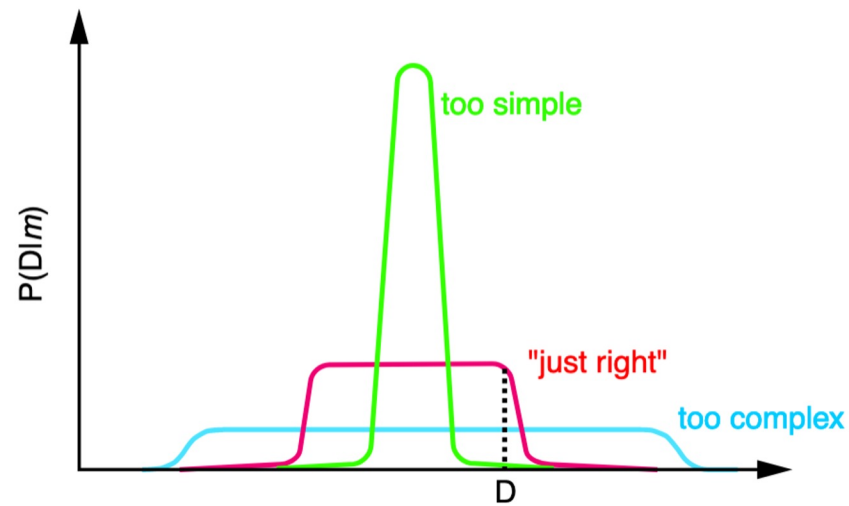
$$p(m|\mathcal{D}) = \frac{p(\mathcal{D}|m)\,p(m)}{p(\mathcal{D})}, \qquad p(\mathcal{D}|m) = \int p(\mathcal{D}|\boldsymbol{\theta}, m)\; p(\boldsymbol{\theta}|m)\; d\boldsymbol{\theta}$$

Interpretations of the Marginal Likelihood ("model evidence"):

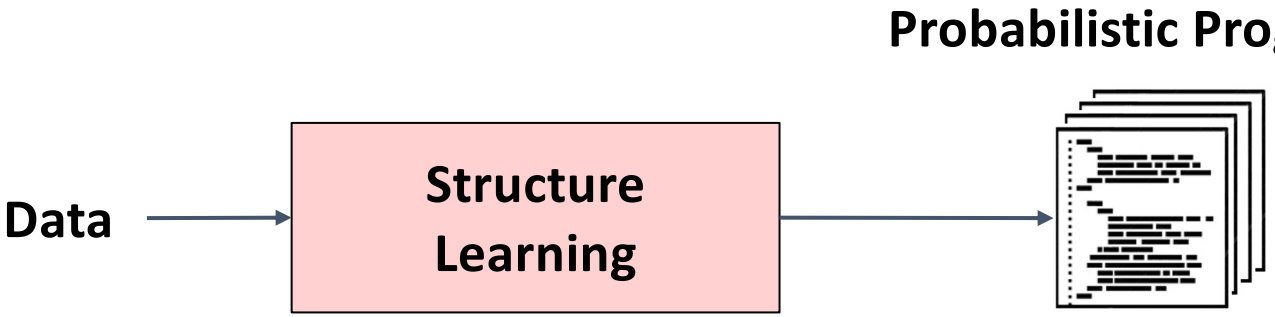
- The probability that *randomly selected* parameters from the prior would generate $\mathcal{D}$.
- Probability of the data under the model, *averaging* over all possible parameter values.
- $\log_2\left(\frac{1}{p(\mathcal{D}|m)}\right)$ is the number of *bits of surprise* at observing data $\mathcal{D}$ under model m.

Model classes that are too simple are unlikely to generate the data set.

Model classes that are too complex can generate many possible data sets, so again, they are unlikely to generate that particular data set at random.



Probabilistic program synthesis via Bayesian structure learning



L domain-specific language

↓

S DSL program structure

↓

θ DSL program parameters

↓

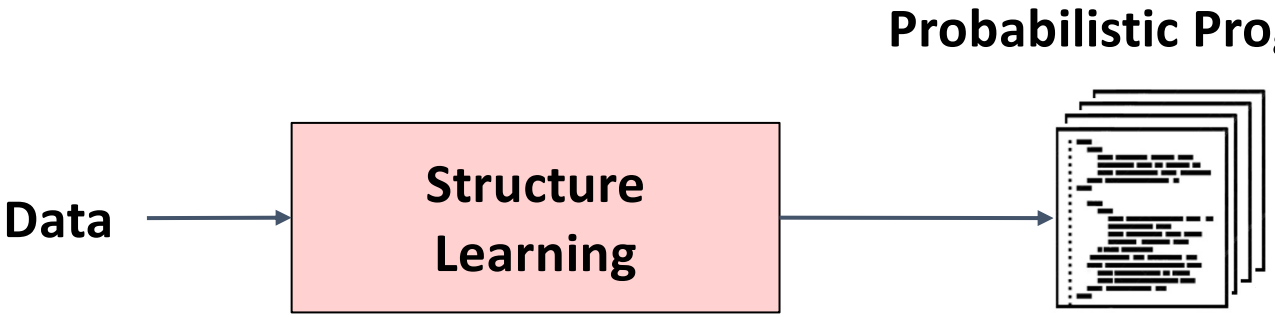
z latent variables

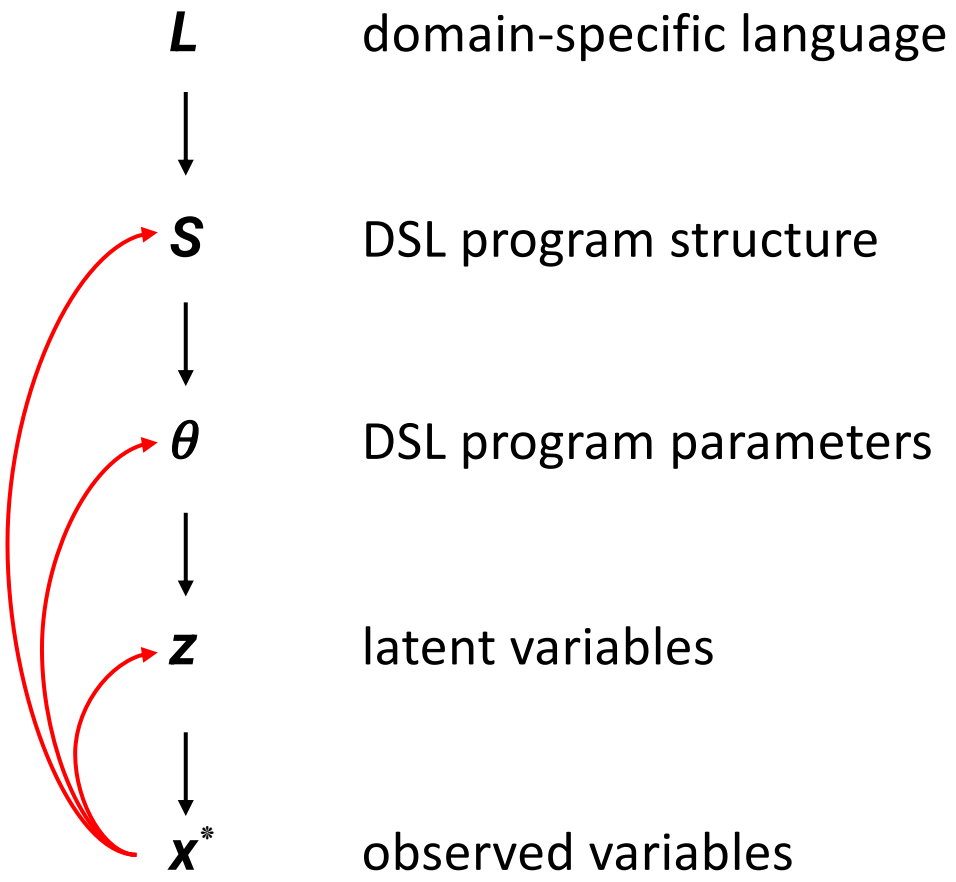
↓

x observed variables

$$P_L(S, \theta, z, x) = P_L(z, x \mid S, \theta) \times P_L(S, \theta)$$

Probabilistic program synthesis via Bayesian structure learning





$$P_L(\boldsymbol{S}, \boldsymbol{\theta}, \boldsymbol{z} \mid \boldsymbol{x}^*) \propto P_L(\boldsymbol{z}, \boldsymbol{x}^* \mid \boldsymbol{S}, \boldsymbol{\theta}) \times P_L(\boldsymbol{S}, \boldsymbol{\theta})$$

Implementing the prior as a probabilistic program

PCFG prior for DSL program $(\boldsymbol{S}, \boldsymbol{\theta})$

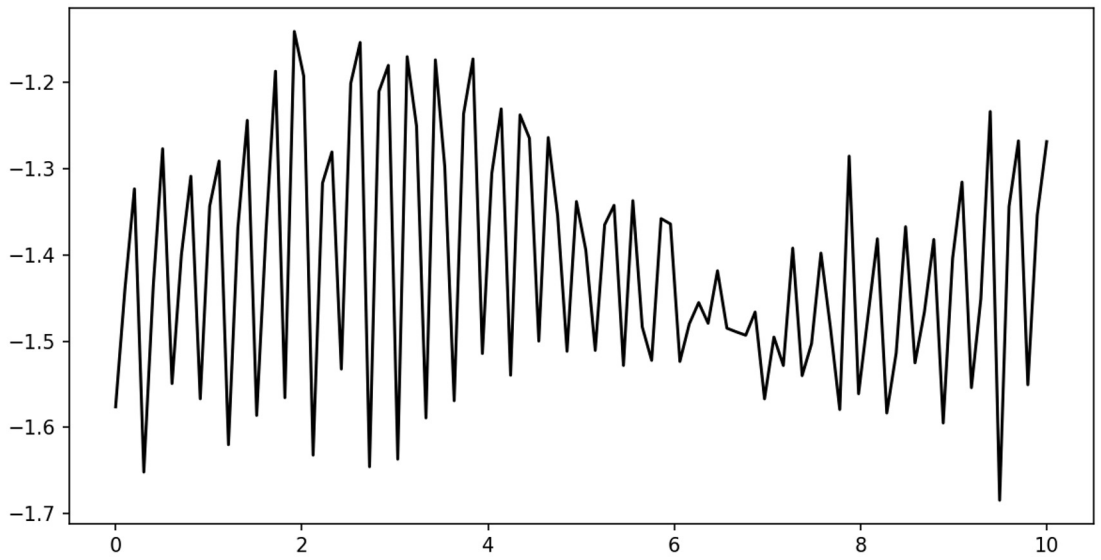
```
@gen function generate_random_dsl_program()
   node_type ~ categorical(production_rule_probs))
   if node_type == CONSTANT
      param ~ uniform(0, 1)
      …

   elseif node_type == PLUS
      left ~ generate_random_dsl_program()
      right ~ generate_random_dsl_program()
      return PlusNode(left, right)

   elseif node_type == CHANGEPOINT
      param ~ uniform(0,1)
      left ~ generate_random_dsl_program()
      right ~ generate_random_dsl_program()
      return ChangePoint(param, left, right)
   end

end
```

time series probabilistic program

```
@gen function gaussian_process_DSL_model(t::Vector)

  DSL_program ~ generate_random_dsl_program()
  Σ = [[DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)]

  ε ~ gamma(1, 1)

  x = Vector(length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
         μ = A(Σ, i) * x[1:i-1]
         σ = sigma(Σ, i)
         x[i] ~ normal(μ, σ + ε)
  end
return x
end
```

Implementing the prior as a probabilistic program

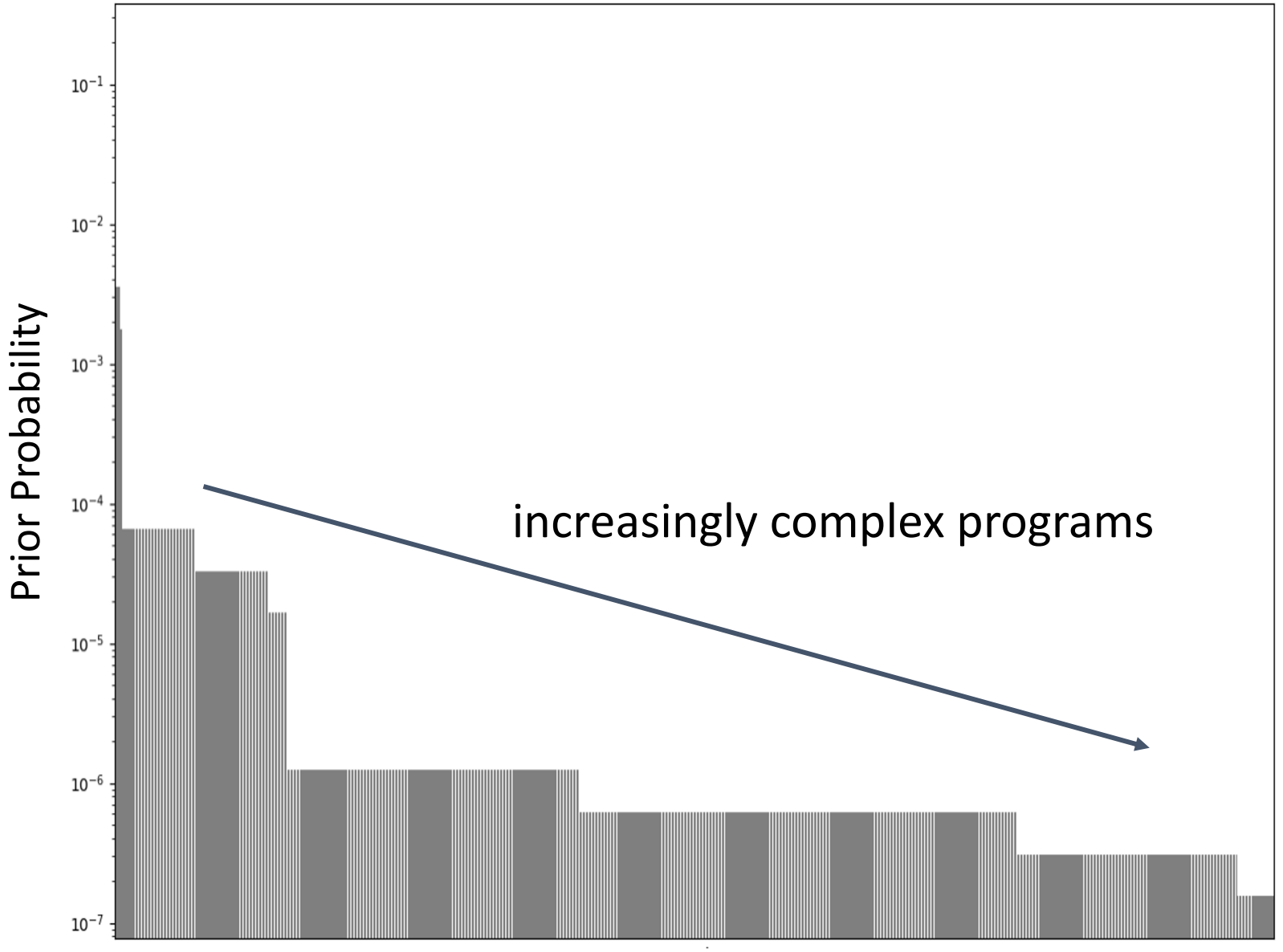
```
Periodic(0.78, 0.30)
```



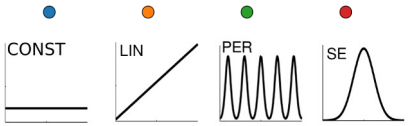
time series probabilistic program

```
@gen function gaussian_process_DSL_model(t::Vector)

  DSL_program ~ generate_random_dsl_program()
  Σ = [[DSL_program(t[i], t[j]) for j=1:length(t)]
    for i=1:length(t)]

  ε ~ gamma(1, 1)

  x = Vector(length(t))
  x[1] ~ normal(0, Σ[1,1])
  for i=2:length(t)
        μ = A(Σ, i) * x[1:i-1]
        σ = sigma(Σ, i)
        x[i] ~ normal(μ, σ + ε)
  end
return x
end
```

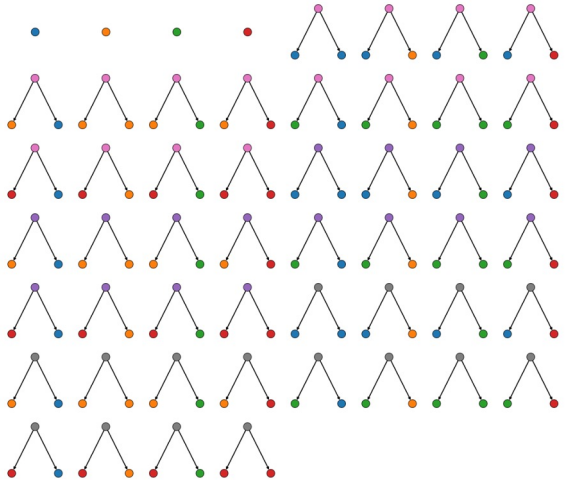
Broad “ignorance” prior over DSL programs



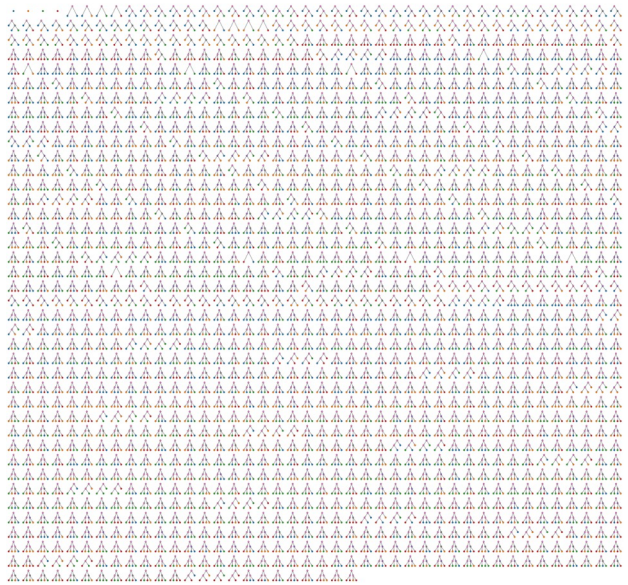
Broad “ignorance” prior over DSL programs



structures with 1 level



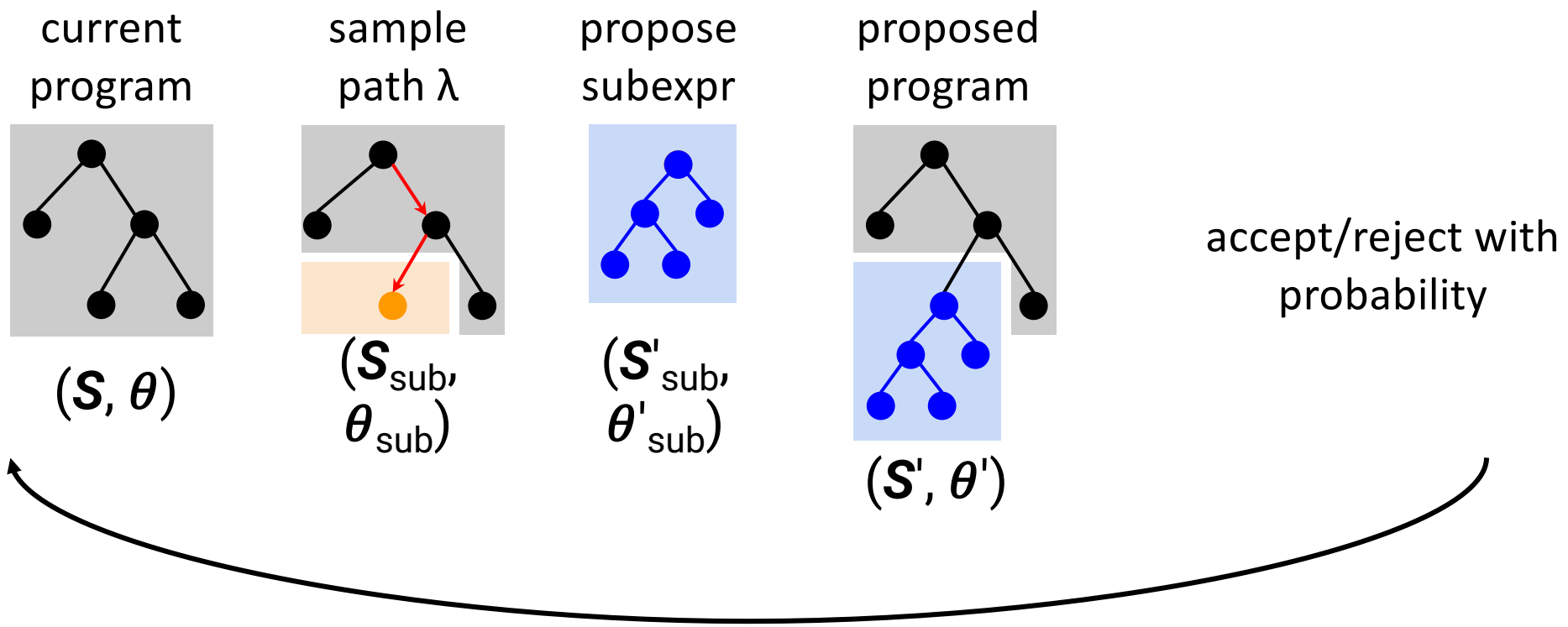
structures with ≤ 2 levels



structures with ≤ 3 levels

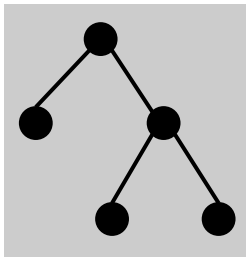
	# structures
1 level	4
2 levels	52
3 levels	8116
4 levels	199 944 576
5 levels	1.2×10^{17}
6 levels	4.3×10^{34}
7 levels	5.4×10^{72}
8 levels	9.4×10^{139}
9 levels	2.5×10^{280}

Posterior sampling



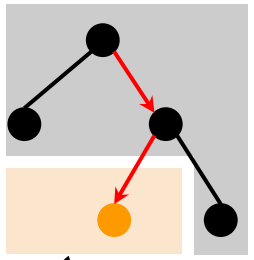
Posterior sampling

current program



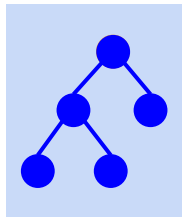
(S, θ)

sample path λ



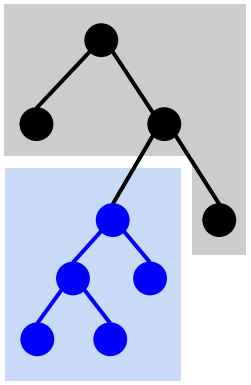
(S_{sub}, θ_{sub})

propose subexpr



$(S'_{sub}, \theta'_{sub})$

proposed program



(S', θ')

accept/reject with probability

Challenge

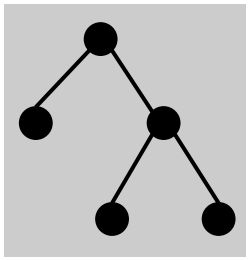
1. Hard to compute acceptance ratio

2. Expensive to compute acceptance ratio

3. Need many iterations to converge

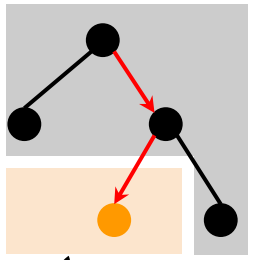
Three challenges

current program



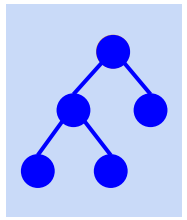
(S, θ)

sample path λ



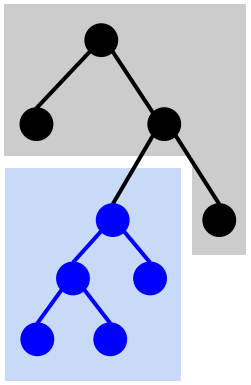
(S_{sub}, θ_{sub})

propose subexpr



$(S'_{sub}, \theta'_{sub})$

proposed program



(S', θ')

accept/reject with probability

Challenge

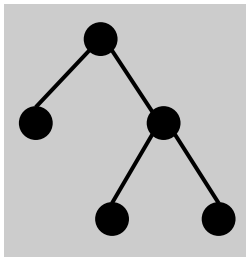
1. Hard to compute acceptance ratio

2. Expensive to compute acceptance ratio

3. Need many iterations to converge

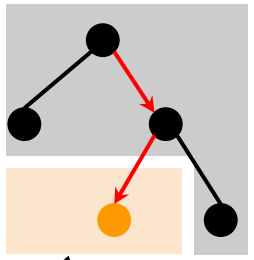
Solution 1: Automation via program tracing and AD

current program



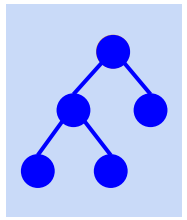
(S, θ)

sample path λ



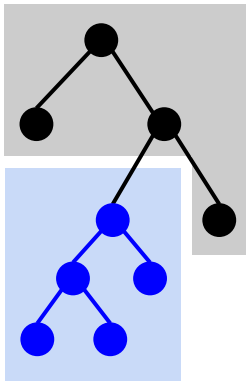
(S_{sub}, θ_{sub})

propose subexpr



$(S'_{sub}, \theta'_{sub})$

proposed program



(S', θ')

accept/reject with probability

derived automatically

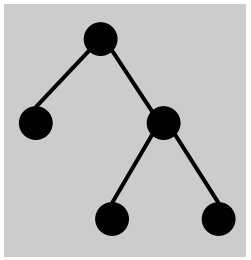
```
@gen function subtree_proposal(trace)
    path ~ walk_tree(trace[:DSL_program],
(:DSL_program,))
    new_subexpr ~ generate_random_program()
    return path
end
@gen function walk_tree(node::Node, path)
    if isa(node, LeafNode)
        done ~ bernoulli(1) && return path
    elseif ({:done} ~ bernoulli(0.5))
        return path
    elseif ({:recurse_left} ~ bernoulli(0.5))
        path = (path..., :left_node)
        return ({:left} ~ walk_tree(node.left, path))
    else
        path = (path..., :right_node)
        return ({:right} ~ walk_tree(node.right, path))
    end
end
```

+

```
@transform subtree_involution (model_in, aux_in)
to (model_out, aux_out) begin
  path = @read(aux_in[], :discrete)
  subtree_address = foldr(=>, path))
  @copy(model_in[subtree_address], aux_out[:new_subexpr])
  @copy(aux_in[:new_subexpr], model_out[subtree_address])
  @copy(aux_in[:path], aux_out[:path])
end
```

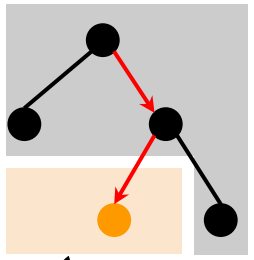
Solution 1: Automation via program tracing and AD

current program



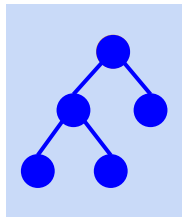
(S, θ)

sample path λ



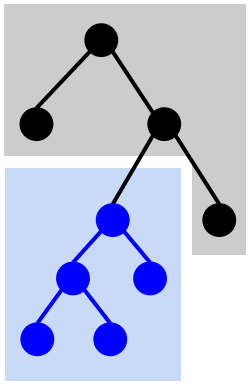
(S_{sub}, θ_{sub})

propose subexpr



$(S'_{sub}, \theta'_{sub})$

proposed program



(S', θ')

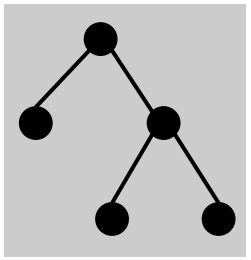
accept/reject with probability

Main Inference Loop (RJ-MCMC + HMC)

```
observations = choicemap((:x, x))
trace = generate(gaussian_process_DSL_model, t,
observations)
for iter=1:n_iters
    # RESAMPLE DSL PROGRAM STRUCTURE AND PARAMS
    trace, = reversible_jump_move(
            trace, subtree_proposal, subtree_involution)

    # RESAMPLE OBSERVATION NOISE
    trace, = hamiltonian_monte_carlo(trace, select(:ε))
```

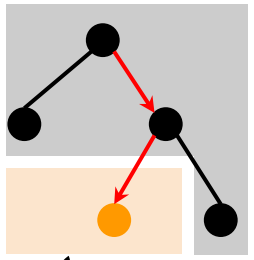
Three challenges

current program



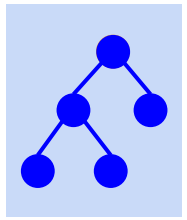
(S, θ)

sample path λ



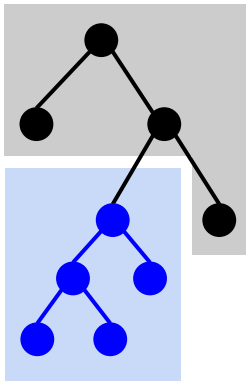
(S_{sub}, θ_{sub})

propose subexpr



$(S'_{sub}, \theta'_{sub})$

proposed program



(S', θ')

accept/reject with probability

Challenge

1. Hard to compute acceptance ratio
2. **Expensive to compute acceptance ratio**
3. Need many iterations to converge

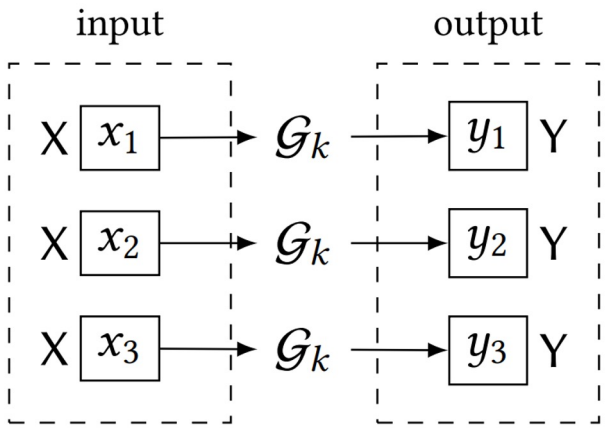
Solution

Automate via program tracing + auto-diff

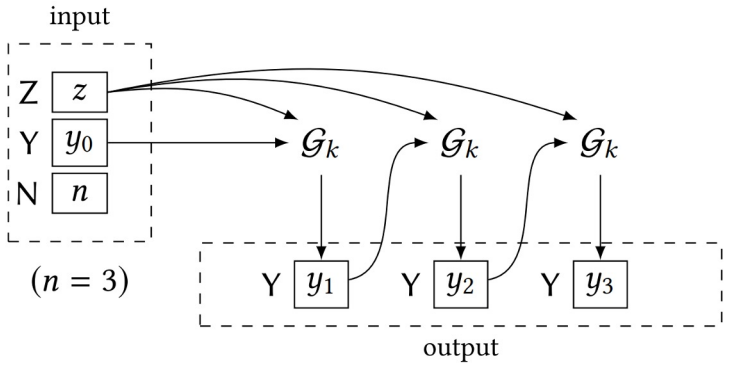
Solution 2: Exploit sparsity via static compilation and caching

	MCMC Runtime (ms/step)	
Gen PPL	2.57	
Julia (Hand coded)	4.73	**1.8x speedup**
Venture PPL	279	**110x speedup**

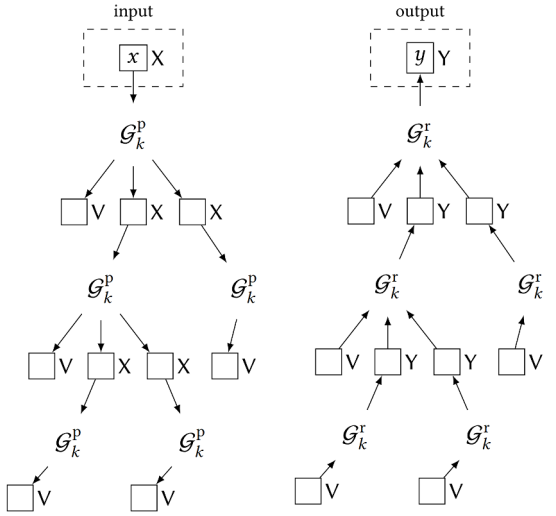
program combinators avoid recomputing entire expression from scratch at each step



Map



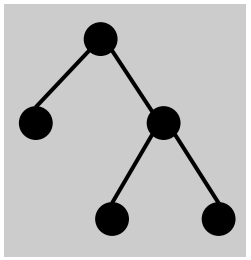
Unfold



Recurse

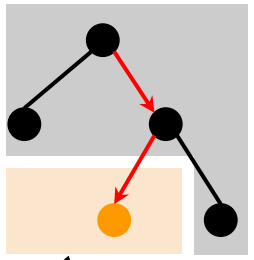
Three challenges

current program



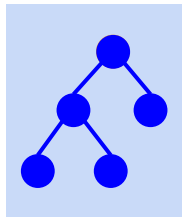
(S, θ)

sample path λ



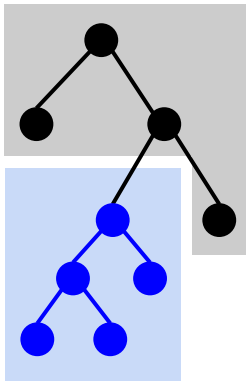
(S_{sub}, θ_{sub})

propose subexpr



$(S'_{sub}, \theta'_{sub})$

proposed program



(S', θ')

accept/reject with probability

Challenge

1. Hard to compute acceptance ratio
2. Expensive to compute acceptance ratio
3. **Need many iterations to converge**

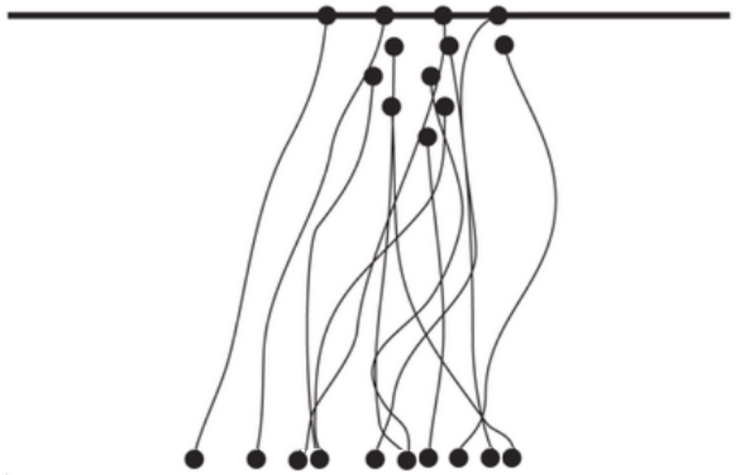
Solution

Automate via program tracing + auto-diff

Exploit sparsity via static compilation + caching

Solution 3: Scaling via sequential Monte Carlo

MCMC



random walk

Resample-Move SMC

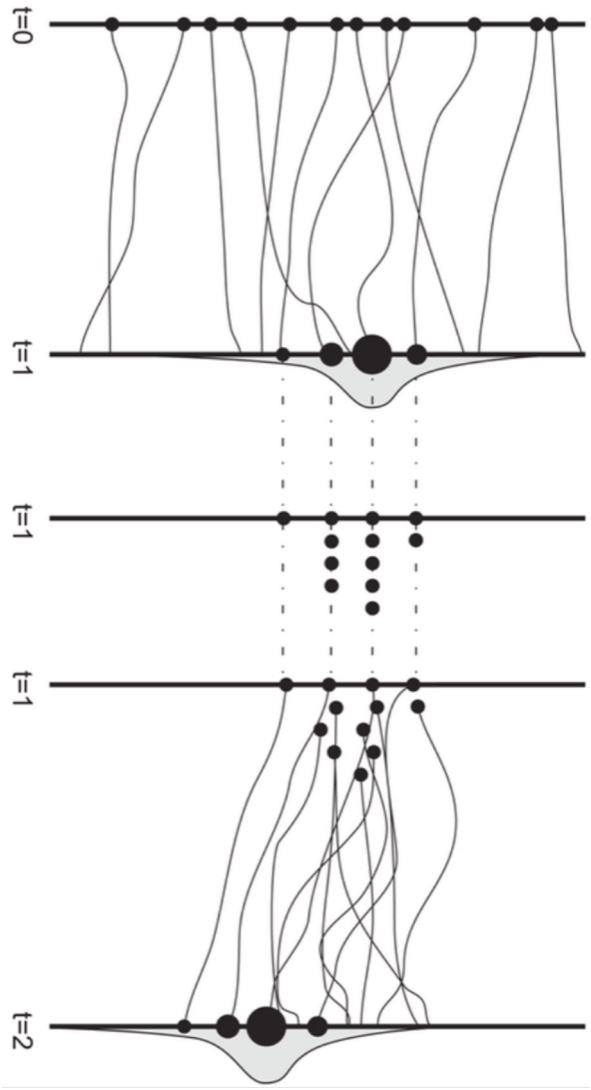
initialize particles

reweight particles

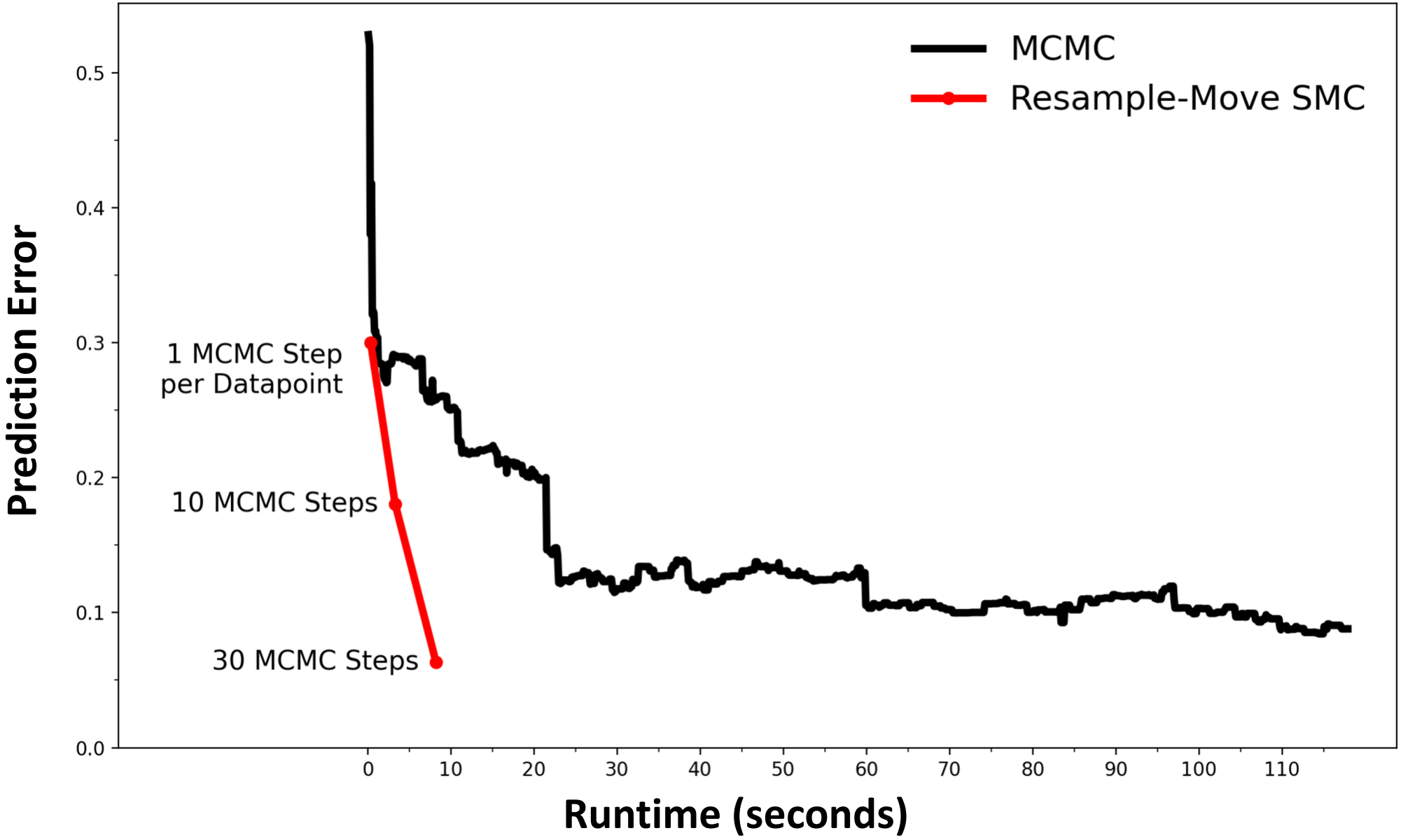
resample particles

move particles (MCMC)

reweight particles



Solution 3: Scaling via sequential Monte Carlo



Solution 3: Scaling via sequential Monte Carlo

Main Inference Loop (SMC + RJ-MCMC + HMC)

```
state = initialize_smc(
          model,
          (xs[1:1],),
          choicemap((:ys, ys[1:1])),
          n_particles)

Gen.smc_step!(state, (xs[1:1],), choicemap((:ys, ys[1:t])))

for t in 2:length(xs)

    Gen.maybe_resample!(state, ess_threshold=n_particles/2)

    Threads.@threads for i=1:n_particles
      local tr = state.traces[i]
      for iter=1:n_mcmc
        trace = reversible_jump_move(
             trace, subtree_proposal, subtree_involution)
        trace = hamiltonian_monte_carlo(trace, select(:ε))
       end
       state.traces[i] = tr
    end

    Gen.smc_step!(state, (xs[1:t],),
                  choicemap((:ys, ys[1:t])))
end
```

initialize particles

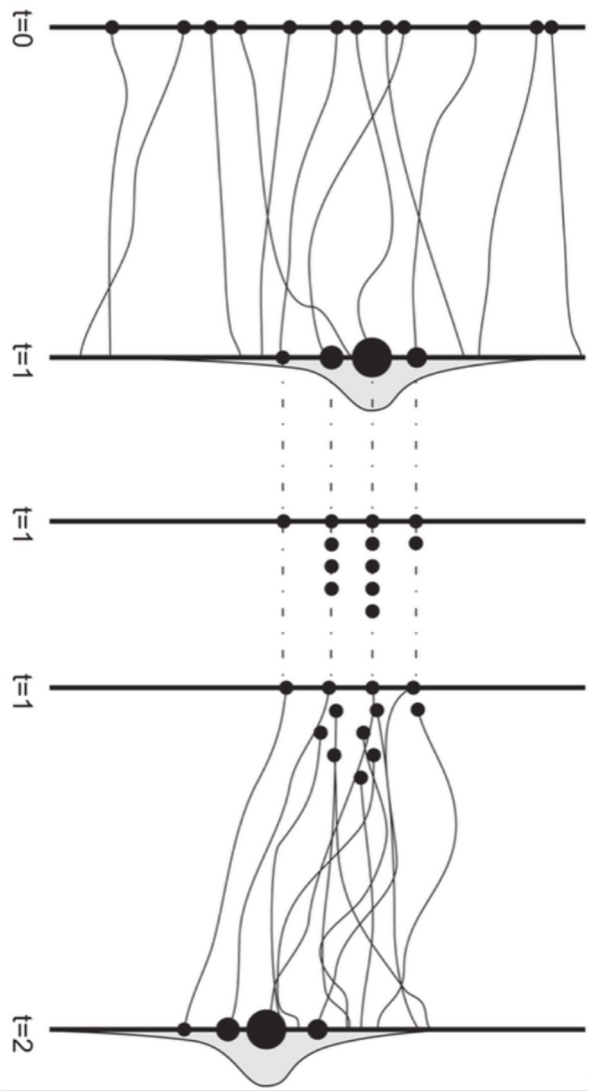
reweight particles

resample particles

move particles (MCMC)

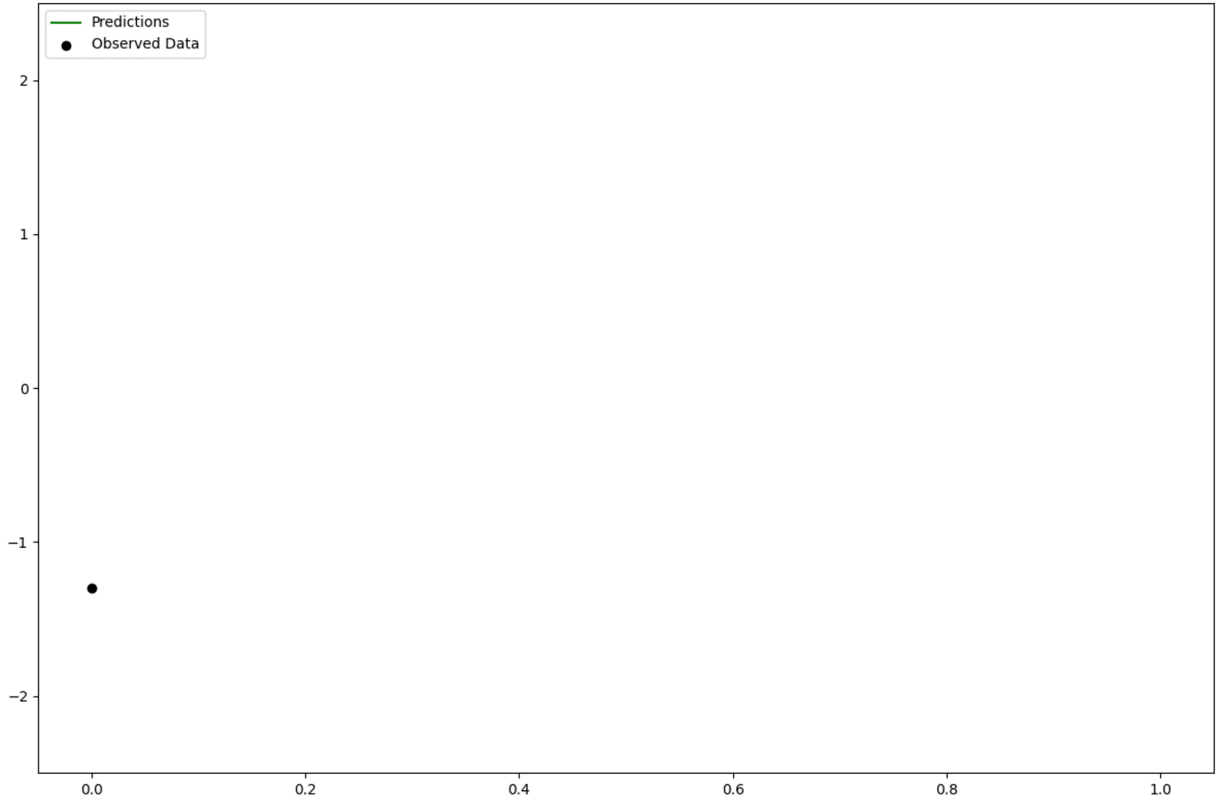
reweight particles

Resample-Move SMC

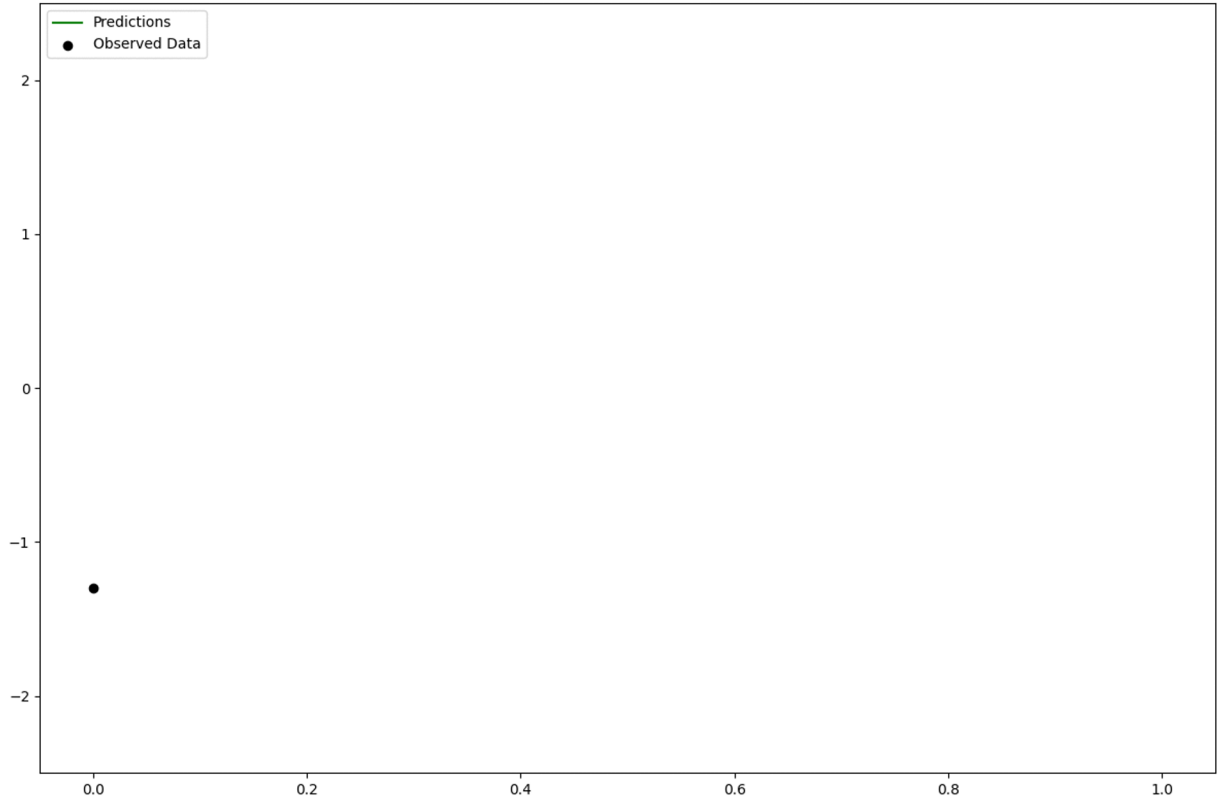


SMC particle rejuvenation is essential for inference quality

SMC with particle rejuvenation
particles resampled and improved

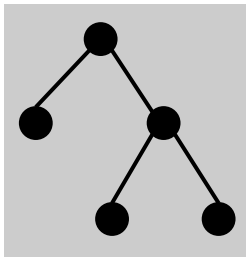


SMC without particle rejuvenation
collapses to particles that underfit



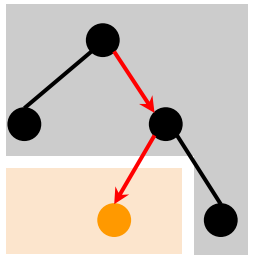
Stochastic sampling through the space of DSL programs

current program



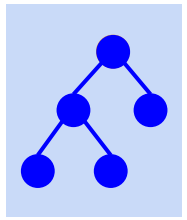
$(\boldsymbol{S}, \boldsymbol{\theta})$

sample path λ



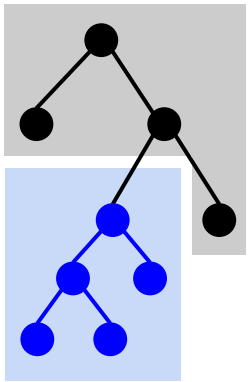
$(\boldsymbol{S}_{\text{sub}}, \boldsymbol{\theta}_{\text{sub}})$

sample subexpr



$(\boldsymbol{S}'_{\text{sub}}, \boldsymbol{\theta}'_{\text{sub}})$

proposed program



$(\boldsymbol{S}', \boldsymbol{\theta}')$

accept/reject with probability

$$\frac{P_L(\boldsymbol{S}', \boldsymbol{\theta}', \boldsymbol{x}^*)}{P_L(\boldsymbol{S}, \boldsymbol{\theta}, \boldsymbol{x}^*)} \frac{q(\lambda, \boldsymbol{S}_{\text{sub}}; \boldsymbol{S}', \boldsymbol{\theta}')}{q(\lambda, \boldsymbol{S}'_{\text{sub}}; \boldsymbol{S}, \boldsymbol{\theta})} \frac{g'(u')}{g(u)} \left| \frac{\partial(\boldsymbol{\theta}', u')}{\partial(\boldsymbol{\theta}, u)} \right|$$

Challenge

1. Hard to compute acceptance ratio
2. Expensive to compute acceptance ratio
3. Need many iterations to converge

Solution

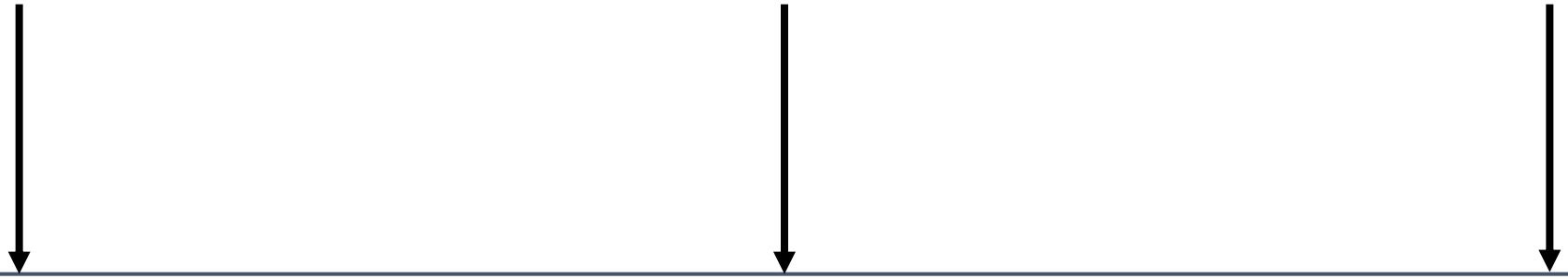
Automate via program tracing + AD

Exploit sparsity via static compilation + caching

Resample-move sequential Monte Carlo

Three solutions that enable scalable Bayesian structure learning

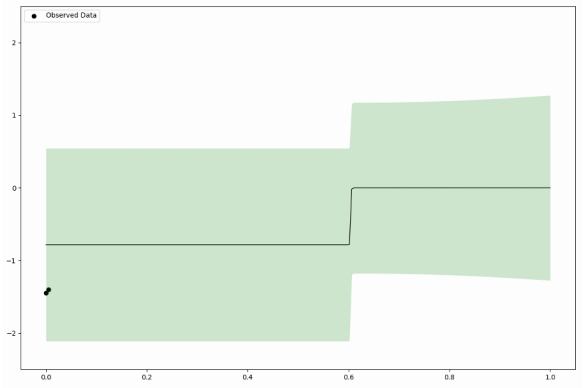
Resample-Move Sequential Monte Carlo	**Meta-Programming Techniques**	**Software Performance Engineering**
hybrid of SMC and MCMC for online learning and faster convergence	generate fast code for sparse importance weights, acceptance ratios, gradients	caching and data structure specialization for fast incremental state updates



new programming model for practical + scalable Bayesian structure learning

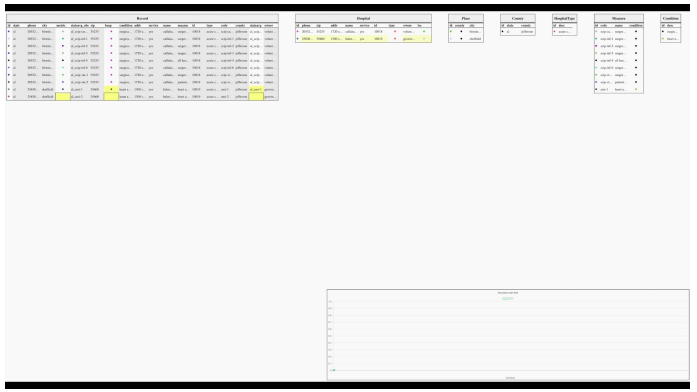
Applications where probabilistic programming outperforms SOTA machine learning

Automated data modeling



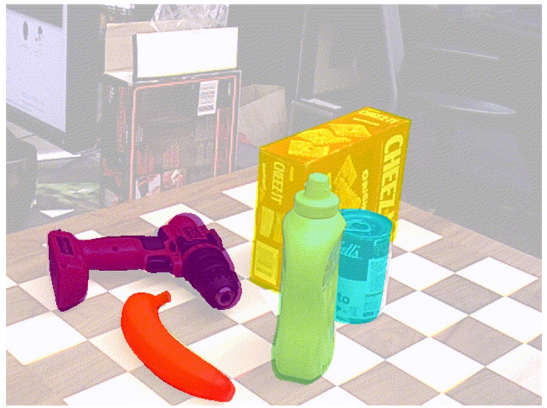
Saad, Cusumano-Towner, Schaechtle, Rinard, Mansinghka (POPL 2019)
Saad and Mansinghka (UAI 2021)
Schaechtle, Freer, Shelby, Saad, and Mansinghka (AutoML 2022)

Common-sense data cleaning



Lew, Agrawal, Sontag, and Mansinghka (AISTATS 2021)

3D scene perception



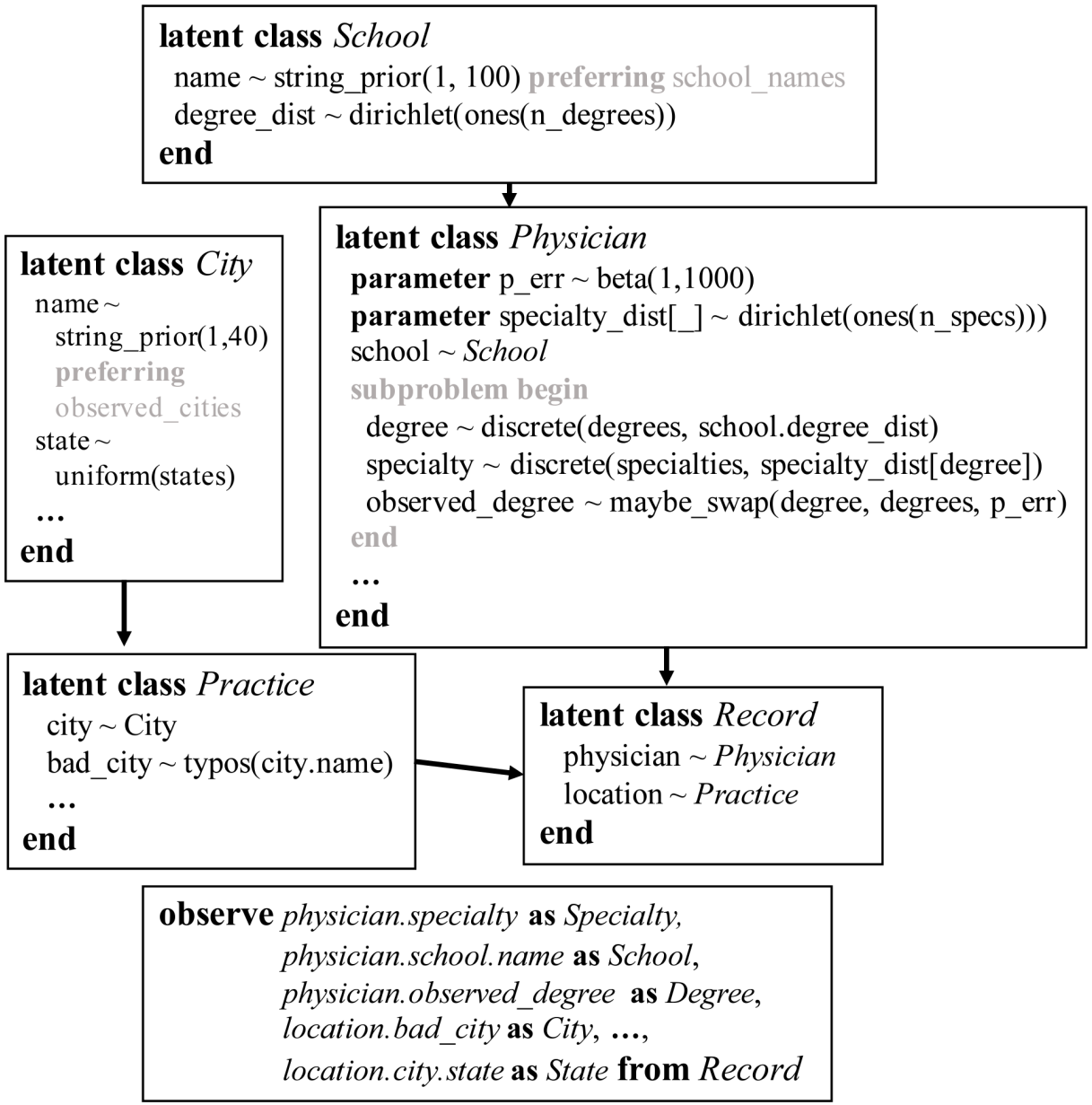
Gothoskar et al. (NeurIPS 2021)

PClean: Domain-specific PPL for Bayesian data cleaning

Dirty observations

Name	Specialty	Degree	School	Address	City	State	Zip
K. Ryan	Family Medicine		PCOM	6317 York Rd	Baltimore	MD	21212-2310
K. Ryan	Family Medicine		PCOM	100 Walter Ward Blvd	Abington	MD	21009-1285
S. Evans	Internal Medicine	MD	UMD	100 Walter Ward Blvd	Abington	MD	21009-1285
M. Grady	Physical Therapy		Other	3491 Merchants Blvd	Abingdon	MD	21009-2030
(2,183,988 more rows)							

Generative model as a PClean program



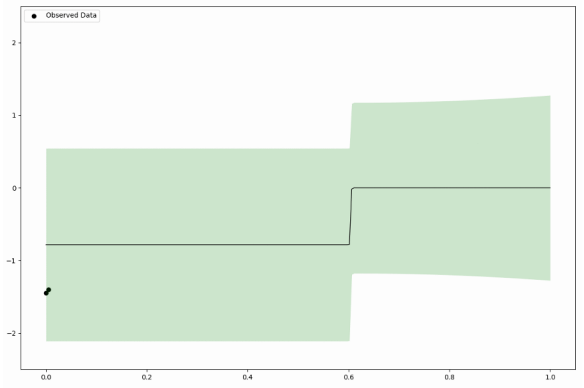
Results

Task	Metric	PClean	HoloClean (Unpublished)	HoloClean	NADEEF	NADEEF + Manual Java Heuristics
Flights	F_1	**0.90**	0.64	0.41	0.07	**0.90**
	Time	**3.1s**	45.4s	32.6s	9.1s	14.5s
Hospital	F_1	**0.91**	0.90	0.83	0.84	0.84
	Time	**4.5s**	1m 10s	1m 32s	27.6s	22.8s
Rents	F_1	**0.69**	0.48	0.48	0	0.51
	Time	1m 20s	20m 16s	13m 43s	13s	**7.2s**

Table 1: Results of PClean and various baseline systems on three diverse cleaning tasks.

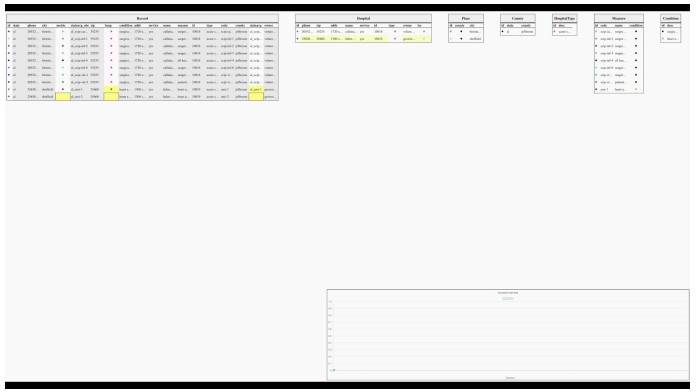
Applications where probabilistic programming outperforms SOTA machine learning

Automated data modeling



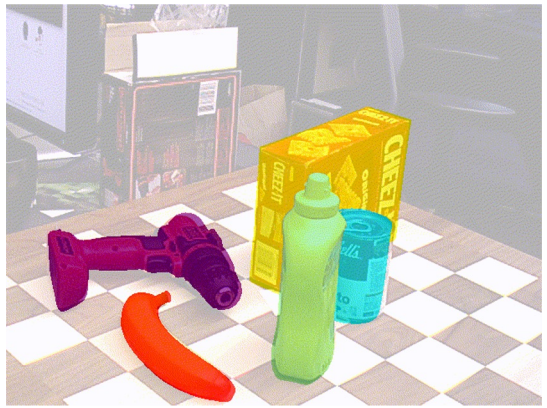
Saad, Cusumano-Towner, Schaechtle, Rinard, Mansinghka (POPL 2019)
Saad and Mansinghka (UAI 2021)
Schaechtle, Freer, Shelby, Saad, and Mansinghka (AutoML 2022)

Common-sense data cleaning



Lew, Agrawal, Sontag, and Mansinghka (AISTATS 2021)

3D scene perception



Gothoskar et al. (NeurIPS 2021)

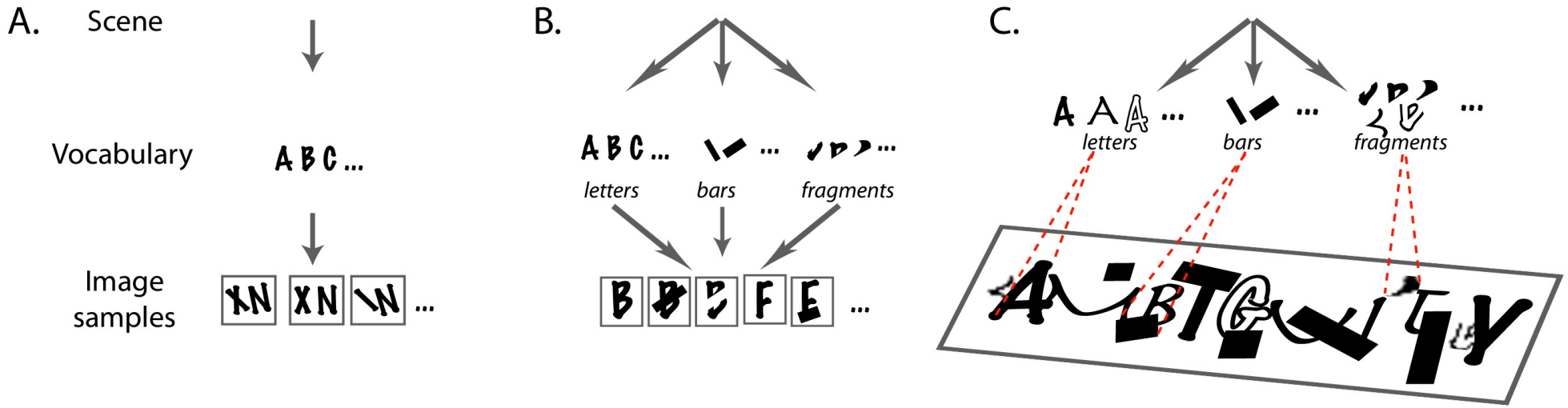
Vision as inverse graphics

Vision as Bayesian Inference: Analysis by Synthesis?

Alan Yuille[1] and Daniel Kersten[2].

Departments of Statistics UCLA[1], Psychology University of Minnesota [2].
emails: yuille@stat.ucla.edu, kersten@umn.edu.

Before CAPTCHA



Vision as inverse graphics

Science

A generative vision model that trains with high data efficiency and breaks text-based CAPTCHAs

DILEEP GEORGE, WOLFGANG LEHRACH, KEN KANSKY, MIGUEL LÁZARO-GREDILLA, [...], AND D. SCOTT PHOENIX +7 authors Authors Info & Affiliations

SCIENCE • 26 Oct 2017 • Vol 358, Issue 6368 • DOI: 10.1126/science.aag2612

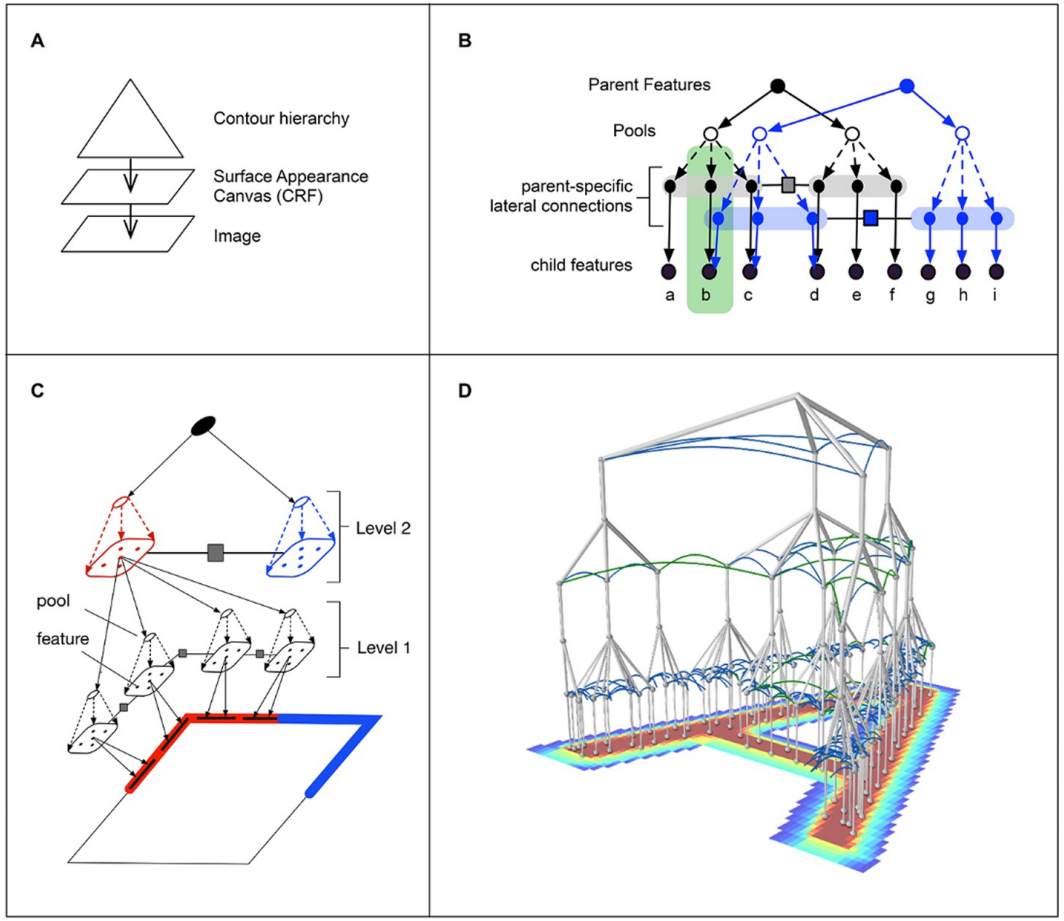
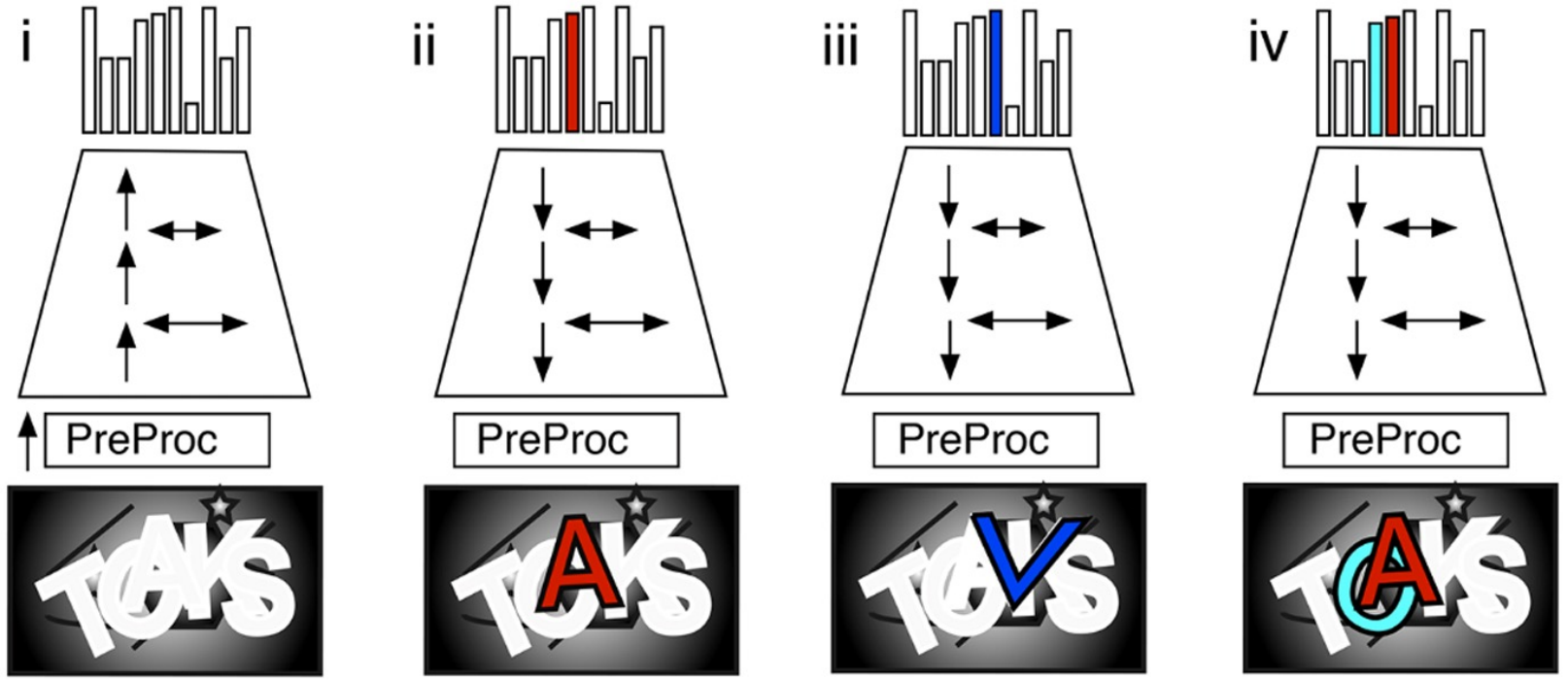


Fig. 2. Structure of the RCN. (A) A hierarchy generates the contours of an object, and a Conditional Random Field (CRF) generates its surface appearance. (B) Two subnetworks at the same level of the contour hierarchy keep separate lateral connections by making parent-specific copies of child features and connecting them with parent-specific laterals; nodes within the green rectangle are copies of the feature marked "e". (C) A three level RCN representing the contours of a square. Features at Level 2 represent the four corners, and each corner is represented as a conjunction of four line-segment features. (D) Four-level network representing an "A".

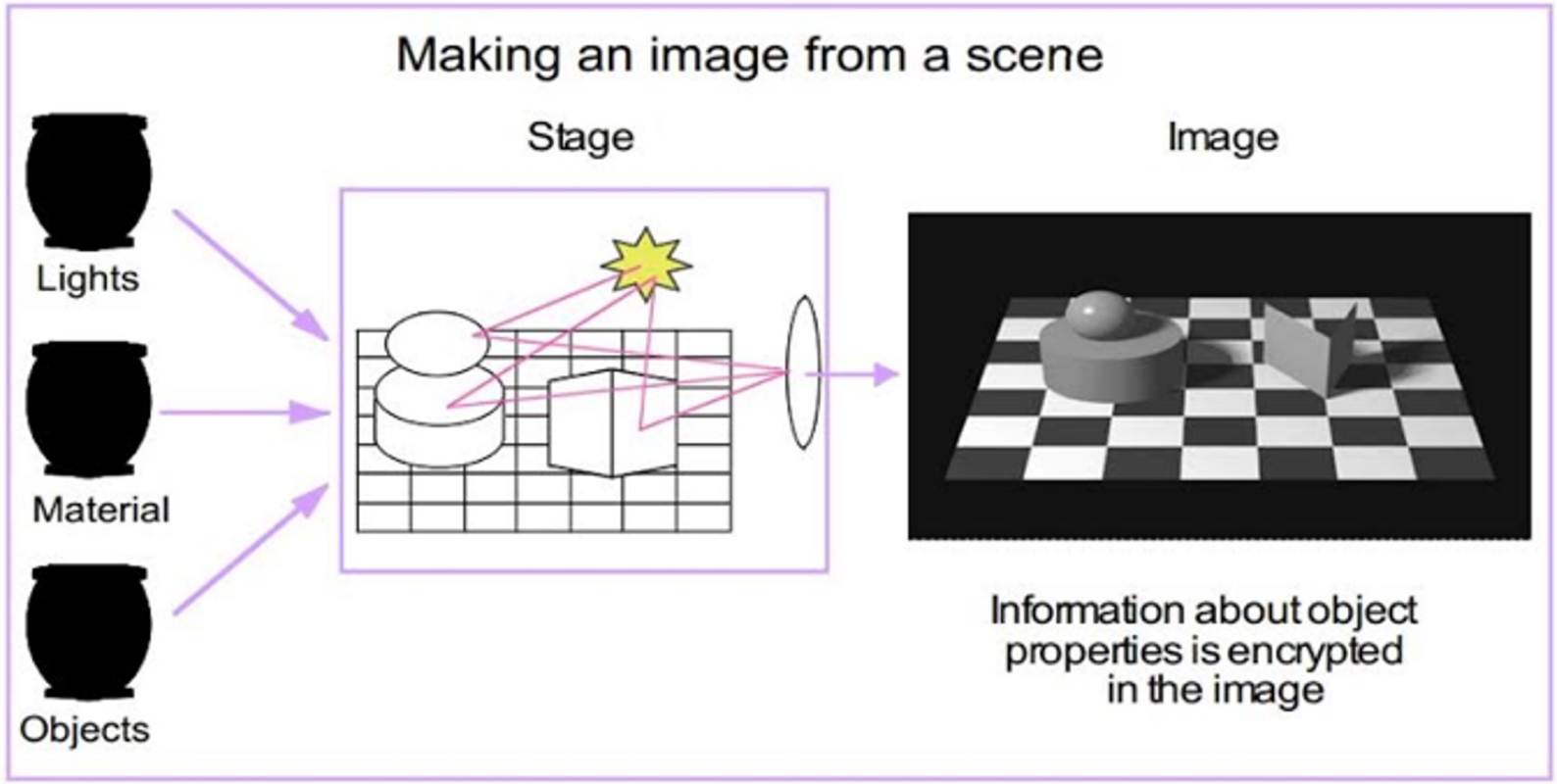
Vision as inverse graphics

Producing hypotheses



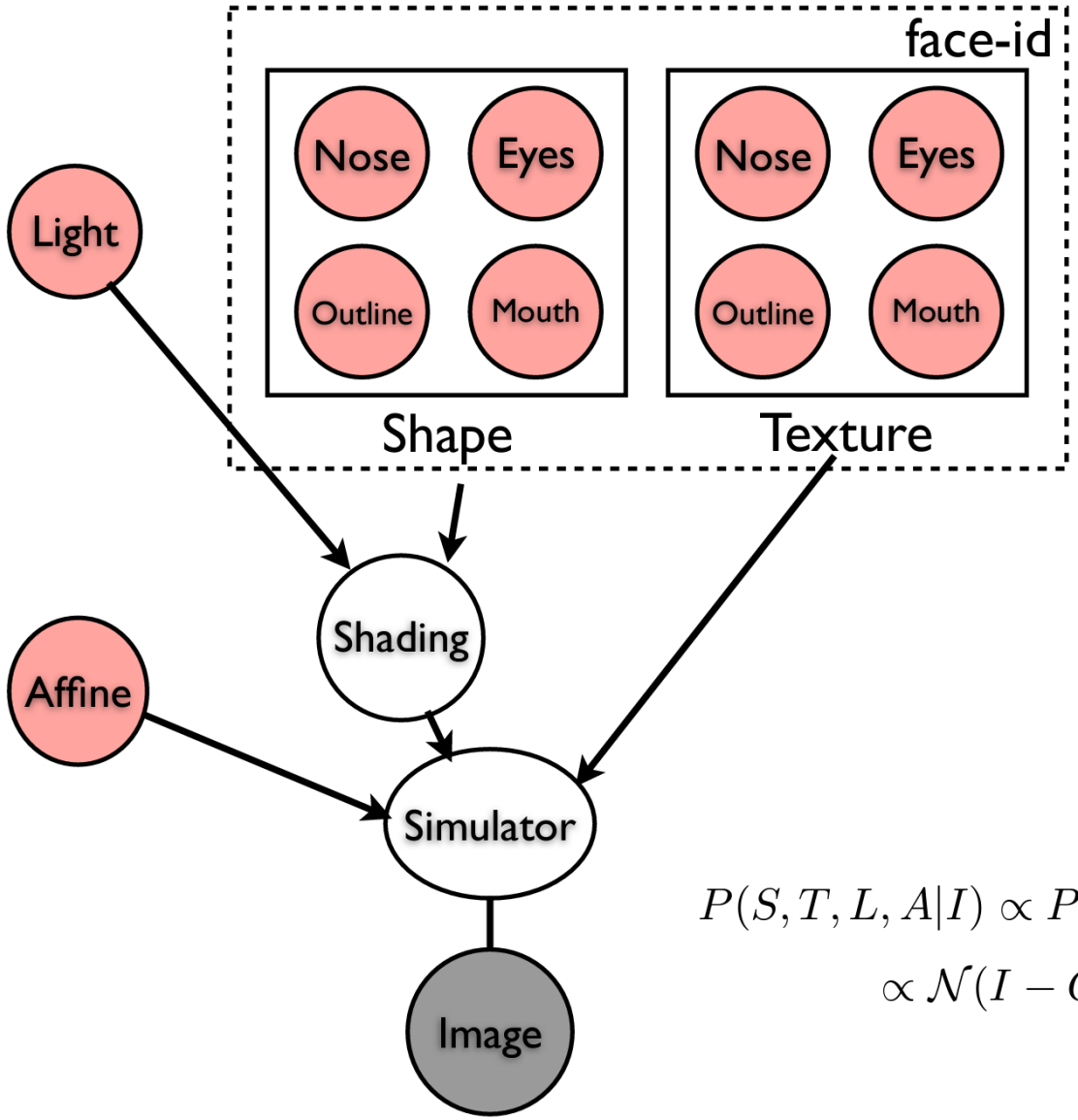
Vision as inverse graphics

3D scene reconstruction



[Kersten, NeurIPS 1998 Tutorial on Computational Vision]

Vision as inverse graphics



3D face reconstruction

Inference Problem:

$$P(S,T,L,A|I) \propto P(I|S,T,L,A)P(L)P(S)P(T)P(A)$$
$$\propto \mathcal{N}(I-O;0,0.1)P(L)P(A)\prod_i P(S_i)P(T_i)$$

[From Tejas Kulkarni]

Probabilistic 3D face reconstruction