A Changepoints package for Julia

Jamie Fairbrother, Lawrence Bardwell, Kaylea Haynes

January 5, 2016
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
- Good performance, approaching that of statically-compiled languages like C
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
- Good performance, approaching that of statically-compiled languages like C
- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
- Good performance, approaching that of statically-compiled languages like C
- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Lisp-like macros and other metaprogramming facilities
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
- Good performance, approaching that of statically-compiled languages like C
- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Lisp-like macros and other metaprogramming facilities
- Advanced package management
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
- Good performance, approaching that of statically-compiled languages like C
- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Lisp-like macros and other metaprogramming facilities
- Advanced package management
- Call C functions directly: no wrappers or special APIs
Why Julia? julialang.org

- Julia is a high-level, high-performance dynamic programming language for technical computing
- Good performance, approaching that of statically-compiled languages like C
- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Lisp-like macros and other metaprogramming facilities
- Advanced package management
- Call C functions directly: no wrappers or special APIs
- Efficient support for Unicode, including but not limited to UTF-8
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints

- Variety of parametric and non-parametric cost functions available

- Implementations of PELT, CROPS, and Binary Segmentation algorithms

- Macros to allow intuitive specification of changepoint model

- Functions for plotting changepoint output

- R-style documentation

- Rigorous unit tests
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints
- Variety of parametric and non-parametric cost functions available
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints
- Variety of parametric and non-parametric cost functions available
- Implementations of PELT, CROPS, and Binary Segmentation algorithms
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints
- Variety of parametric and non-parametric cost functions available
- Implementations of PELT, CROPS, and Binary Segmentation algorithms
- Macros to allow intuitive specification of changepoint model
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints
- Variety of parametric and non-parametric cost functions available
- Implementations of PELT, CROPS, and Binary Segmentation algorithms
- Macros to allow intuitive specification of changepoint model
- Functions for plotting changepoint output
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints
- Variety of parametric and non-parametric cost functions available
- Implementations of PELT, CROPS, and Binary Segmentation algorithms
- Macros to allow intuitive specification of changepoint model
- Functions for plotting changepoint output
- R-style documentation

Rigorous unit tests
Changepoints.jl Features

- Flexible macro for simulating time series with changepoints
- Variety of parametric and non-parametric cost functions available
- Implementations of PELT, CROPS, and Binary Segmentation algorithms
- Macros to allow intuitive specification of changepoint model
- Functions for plotting changepoint output
- R-style documentation
- Rigourous unit tests
Installing Changepoints.jl

- To install the latest stable release of Julia on Ubuntu:

  ```
sudo add-apt-repository ppa:staticfloat/juliareleases
sudo apt-get update
sudo apt-get install julia
  ```

- To install Changepoints and optional Winston plotting package in Julia:

  ```
Pkg.add("Changepoints")
Pkg.add("Winston")
  ```
Simulating Changepoints

```
using Winston, Changepoints
\[ \mu, \sigma = \text{Uniform}(-5.0, 5.0), 1.0 \]
\[ n = 1000 \quad # \text{Length of sample} \]
\[ \lambda = 10 \quad # \text{Frequency of changes} \]
\[ \text{data, cps = @changepoint_sampler n \ \lambda \ \text{Normal}(\mu, \sigma)} \]
\[ \text{plot(data, cps)} \]
```
Simulating Changepoints

using Winston, Changepoints

\( \mu, \sigma = \text{Uniform}(-5.0, 5.0), 1.0 \)

\( n = 1000 \quad \# \text{Length of sample} \)

\( \lambda = 10 \quad \# \text{Frequency of changes} \)

data, cps = @changepoint_sampler n \( \lambda \) Normal(\( \mu, \sigma \))

plot(data, cps)
Finding Changepoints

- Currently PELT and binary segmentation have been implemented
- Both algorithms have a uniform interface for all cost functions

```python
# Create cost function
seg_cost = NormalMeanSegment(data, σ)
# Run PELT on cost function
pelt_cps, pelt_cost = PELT(seg_cost, n)
# Run Binary segmentation on cost function
bs_cps, bs_cost = BS(seg_cost, n)
```
Creating cost functions

- A cost function is simply any function which takes two indices and returns the cost between

```plaintext
function NormalMeanSegment (data: Array{Float64}, σ: Real = 1.0)
   cd = [0, cumsum(data)]
   cd² = [0, cumsum(abs2(data))]
   function cost(s: Int64, t: Int64)
      (cd²[t + 1] - cd²[s + 1] - abs2(cd[t + 1] - cd[s + 1]) / (t - s)) / (σ²)
   end
end
```

Other functions to construct segment cost functions:
- NormalVarSegment, NormalMeanVarSegment
- PoissonSegment, ExponentialSegment, NonparametricSegment
- GammaRateSegment, GammaShapeSegment
Creating cost functions

- A cost function is simply any function which takes two indices and returns the cost between
- \( C(y_s:t) = -2 \log p(y_{s:t}|\hat{\theta}) \) for all \( s \) and \( t \).

```plaintext
function NormalMeanSegment (data : Array {Float 64}), σ : Real = 1.0 )
    cd = [0, cumsum(data)]
    cd2 = [0, cumsum(abs2(data))]
    function cost(s : Int64, t : Int64)
        return (cd2[t + 1] - cd2[s + 1] - abs2(cd[t + 1] - cd[s + 1]) / (t - s)) / (σˆ2)
    end
end
```

Other functions to construct segment cost functions:
- NormalVarSegment, NormalMeanVarSegment
- PoissonSegment, ExponentialSegment, NonparametricSegment
- GammaRateSegment, GammaShapeSegment
Creating cost functions

- A cost function is simply any function which takes two indices and returns the cost between
- \( C(y_{s:t}) = -2 \log p(y_{s:t} | \hat{\theta}) \) for all \( s \) and \( t \).
- Adding new cost functions is therefore very easy.
Creating cost functions

- A cost function is simply any function which takes two indices and returns the cost between
- $C(y_{s:t}) = -2 \log p(y_{s:t} | \hat{\theta})$ for all $s$ and $t$.
- Adding new cost functions is therefore very easy
Creating cost functions

- A cost function is simply any function which takes two indices and returns the cost between
- \( C(y_{s:t}) = -2 \log p(y_{s:t}|\hat{\theta}) \) for all \( s \) and \( t \).
- Adding new cost functions is therefore very easy

```plaintext
function NormalMeanSegment(data :: Array{Float64}, \sigma :: Real = 1.0)
    cd = [0, cumsum(data)]
    cd_2 = [0, cumsum(abs2(data))]
    function cost(s :: Int64, t :: Int64)
        (cd_2[t+1] - cd_2[s+1] - abs2(cd[t+1] - cd[s+1])/(t-s))/(\sigma^2)
    end
    return cost
end

- Other functions to construct segment cost functions:
  NormalVarSegment, NormalMeanVarSegment
  PoissonSegment, ExponentialSegment, NonparametricSegment
  GammaRateSegment, GammaShapeSegment
```
Changepoints for range of penalties (CROPS)

# Set penalty range
\( \beta_1, \beta_2 = 1.0, 10.0 \)

# Run CROPS
```python
crops_out = CROPS(seg_cost, n, (\beta_1, \beta_2))
plot(crops_out)
```
Changepoint macros

- Macros allow direct specification of Changepoint model

```plaintext
# Run PELT with default penalty
pelt_cps, pelt_cost = @PELT data Normal(?, \sigma)
# Run PELT with specified penalty
pelt_cps, pelt_cost = @PELT data Normal(?, \sigma) \beta_1
# Run CROPS
crops_output = @PELT data Normal(?, \sigma) \beta_1 \beta_2
```

<table>
<thead>
<tr>
<th>Changepoint Model</th>
<th>Cost function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal(?, \sigma)</td>
<td>NormalMeanSegment</td>
</tr>
<tr>
<td>Normal(\mu, ?)</td>
<td>NormalMeanVarSegment</td>
</tr>
<tr>
<td>Normal(?, ?)</td>
<td>NormalVarSegment</td>
</tr>
<tr>
<td>Gamma(? , \beta)</td>
<td>GammaShapeSegment</td>
</tr>
<tr>
<td>Gamma(\alpha, ?)</td>
<td>GammaRateSegment</td>
</tr>
<tr>
<td>Exponential(?)</td>
<td>ExponentialSegment</td>
</tr>
<tr>
<td>Poisson(?)</td>
<td>PoissonSegment</td>
</tr>
<tr>
<td>Nonparametric(k)</td>
<td>Nonparametric</td>
</tr>
</tbody>
</table>
Accessing documentation

```
 julia> Pkg.add("Lexicon")
 julia> using Lexicon
 julia> ?
 help?@PELT

Changepoints.@PELT(data, dist, args...)

Runs the PELT algorithm using a specified cost function and penalty to find the position and number of changepoints

Usage

1. @PELT data changepoint_model: Run PELT with default penalty value

2. @PELT data changepoint_model β: Run PELT at penalty value β

  ✔ From Julia v0.4 documentation will be directly accessible
```
Version 0.2 - Potential enhancements

- Add C version of PELT

Bayesian - Can calculate the posterior using a set of recursions (Forward-backward algorithm). Like the cost we calculate the marginal likelihood but we have a prior on any parameters of the model $\pi(\theta)$ and some prior distribution for the distance between changepoints.

What else?
Version 0.2 - Potential enhancements

- Add C version of PELT
- Implement recycling for CROPS

Bayesian - Can calculate the posterior using a set of recursions (Forward-backward algorithm). Like the cost we calculate the marginal likelihood but we have a prior on any parameters of the model \( \pi(\theta) \int p(y_s|\theta) \pi(\theta) d\theta \) and some prior distribution for the distance between changepoints.
Version 0.2 - Potential enhancements

- Add C version of PELT
- Implement recycling for CROPS
- Implement FPOP
Version 0.2 - Potential enhancements

- Add C version of PELT
- Implement recycling for CROPS
- Implement FPOP
- Bayesian - Can calculate the posterior using a set of recursions (Forward-backward algorithm). Like the cost we calculate the marginal likelihood but we have a prior on any parameters of the model $\pi(\theta)$

$$\int p(y_{s:t}|\theta)\pi(\theta)\,d\theta$$

and some prior distribution for the distance between changepoints.
Version 0.2 - Potential enhancements

- Add C version of PELT
- Implement recycling for CROPS
- Implement FPOP
- Bayesian - Can calculate the posterior using a set of recursions (Forward-backward algorithm). Like the cost we calculate the marginal likelihood but we have a prior on any parameters of the model $\pi(\theta)$

$$\int p(y_{s:t}|\theta)\pi(\theta)\,d\theta$$

and some prior distribution for the distance between changepoints.
- What else?
Getting involved

- Code can be viewed and downloaded from https://github.com/STOR-i/Changepoints.jl
- You will need a Github account to make changes to the package
Any Questions?
**Figure**: Benchmark times relative to C from julialang.org (smaller is better, C performance = 1.0)