RAPID PROTOTYPING WITH TREEPPL USING JUPYTER NOTEBOOK

Empowering Computational Biologists with Seamless TreePPL Integration

Miking Workshop 2024

Jan Kudlicka
jan.kudlicka@bi.no

Department of Data Science and Analytics
BI Norwegian Business School, Oslo, Norway

December 4, 2024

TREEPPL INTERFACES

Goal: Make TreePPL accessible by leveraging widely-used languages like Python and R.

We provide two interfaces for integration:

Language	Interface	GitHub Link
Python	`treeppl-python`	https://github.com/treeppl/treeppl-python
R	`treepplr`	https://github.com/treeppl/treepplr

Why Python and R?

- Python and R are widely adopted in **computational biology**.
- Extensive libraries for:
  - Support of various file formats
  - Data analysis
  - Visualization
  - Statistical modeling
- The interfaces hide complexity:
  - **No manual compilation**
  - **No input/output handling**

HOW TREEPPL INTERFACES WORK

The extensions follow a streamlined process:

1. **Compile the model**: Call the TreePPL compiler (`tpplc`).
2. **Prepare the input file**: Generate a JSON input file with model arguments.
3. **Run the program**: Execute the compiled model.
4. **Process output**: Parse results into Python/R-compatible data structures.

PYTHON INTEGRATION

Installation

1. Ensure TreePPL is installed and `tpplc` is in your `PATH`.
2. Install `treeppl-python` with pip:

```
pip install "git+https://github.com/treeppl/treeppl-python#egg=treeppl"
```

The package will be available on PyPI in the future.

In [1]:
```
import treeppl

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 6)
from IPython.display import clear_output
import seaborn as sns
sns.set_theme()

from Bio import Phylo
```

JUPYTER NOTEBOOK INTEGRATION

Why Jupyter?

- **Interactive model experimentation**: Test and modify TreePPL models in real-time.
- **Inline visualization**: Effortlessly post-process and visualize results within the notebook.
- **Integrated workflow**: Combine code, documentation, and results in one place for streamlined analysis.
- **Reproducibility & sharing**: Share notebooks to ensure consistent, repeatable research.

In [2]:
```
%load_ext treeppl.ipython
```

```
<IPython.core.display.Javascript object>
```

Key Features

- TreePPL magic: Use `%%treeppl` to define and compile models.
- Syntax highlighting: Available in Jupyter Notebook (not JupyterLab yet).

Example

In [ ]:
```
%%treeppl flip samples=10

model function flip(): Bool {
  assume p ~ Bernoulli(0.5);
  return p;
}
```

EXAMPLE: FAIR COIN FLIP

In [21]:
```
%%treeppl flip samples=10

model function flip(): Bool {
  assume p ~ Bernoulli(0.4);
  return p;
}
```

EXAMPLE: FAIR COIN FLIP

In [21]:
```
%%treeppl flip samples=10

model function flip(): Bool {
  assume p ~ Bernoulli(0.4);
  return p;
}
```

In [28]:
```
res = flip()
res.samples
```

Out[28]: [False, True, False, False, False, True, False, True, False, True]

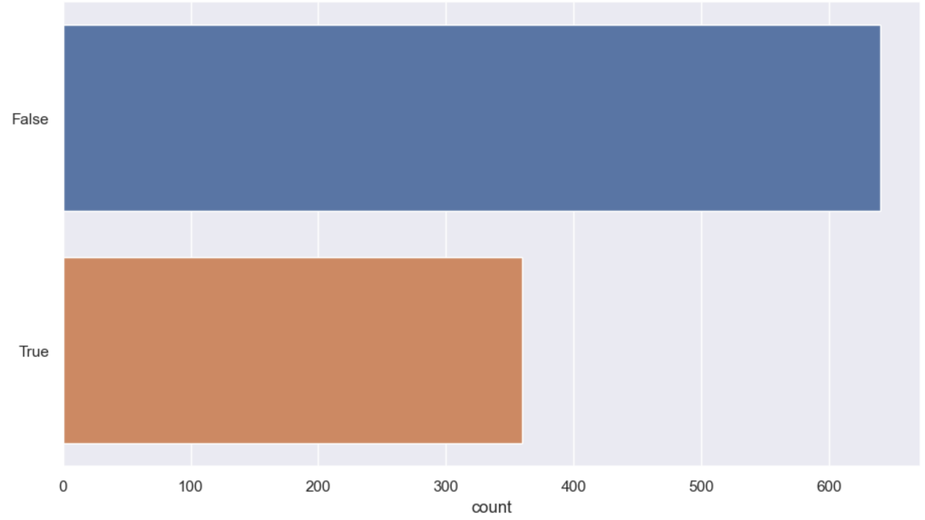
EXAMPLE: FAIR COIN FLIP

We can dynamically adjust the number of samples:

In [29]:
```
flip.set_samples(1000)
```

In [30]:
```
res = flip()
sns.countplot(y=res.samples)
```

Out[30]:
```
<Axes: xlabel='count'>
```



EXAMPLE: UNFAIR COIN

In [31]:

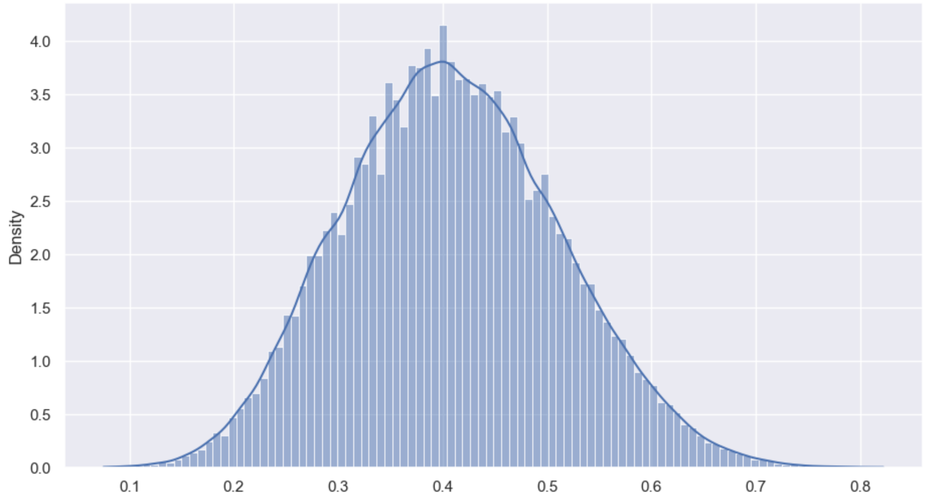
```
%%treeppl coin samples=100000

model function coin(outcomes: Bool[]): Real {
  assume p ~ Uniform(0.0, 1.0);
  for i in 1 to (length(outcomes)) {
    observe outcomes[i] ~ Bernoulli(p);
  }
  return p;
}
```

EXAMPLE: UNFAIR COIN

In [31]:
```
%%treeppl coin samples=100000

model function coin(outcomes: Bool[]): Real {
  assume p ~ Uniform(0.0, 1.0);
  for i in 1 to (length(outcomes)) {
    observe outcomes[i] ~ Bernoulli(p);
  }
  return p;
}
```

In [33]:
```
res = coin(
    outcomes=[
        True, True, True, False, True, False, False, True, True, False,
        False, False, True, False, True, False, False, False, False, False,
    ]
)
sns.histplot(
    x=res.samples, weights=res.nweights, bins=100, stat="density", kde=True
)
```

Out[33]: <Axes: ylabel='Density'>



EXAMPLE: GENERATING A CRBD TREE

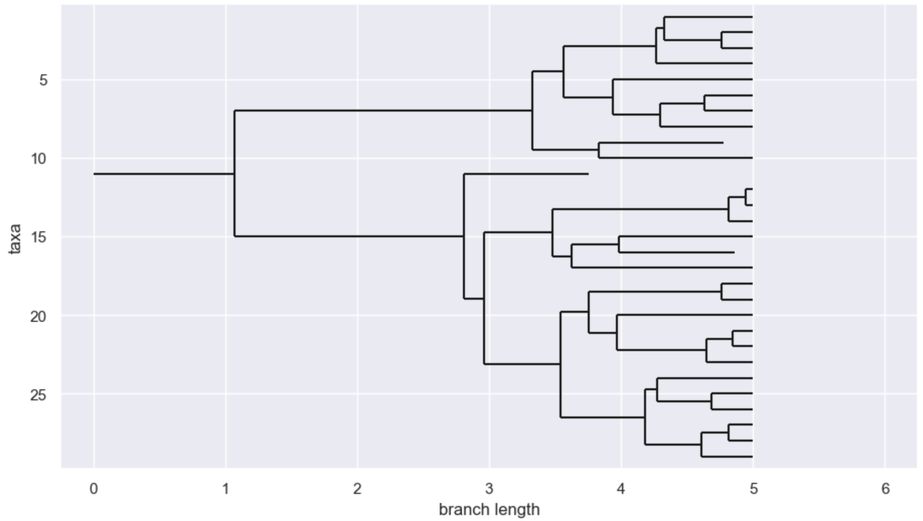
In [34]:

```
%%treeppl generative_crbd samples=1

model function generativeCrbd(time: Real, lambda: Real, mu: Real): Tree {
  assume waitingTime ~ Exponential(lambda + mu);
  let eventTime = time - waitingTime;
  if eventTime < 0.0 {
    return Leaf {age = 0.0};
  } else {
    assume isSpeciation ~ Bernoulli(lambda / (lambda + mu));
    if isSpeciation {
      return Node {
        left = generativeCrbd(eventTime, lambda, mu),
        right = generativeCrbd(eventTime, lambda, mu),
        age = eventTime
      };
    } else {
      return Leaf {age = eventTime};
    }
  }
}
```

EXAMPLE: GENERATING A CRBD TREE

In [34]:
```
%%treeppl generative_crbd samples=1

model function generativeCrbd(time: Real, lambda: Real, mu: Real): Tree {
  assume waitingTime ~ Exponential(lambda + mu);
  let eventTime = time - waitingTime;
  if eventTime < 0.0 {
    return Leaf {age = 0.0};
  } else {
    assume isSpeciation ~ Bernoulli(lambda / (lambda + mu));
    if isSpeciation {
      return Node {
        left = generativeCrbd(eventTime, lambda, mu),
        right = generativeCrbd(eventTime, lambda, mu),
        age = eventTime
      };
    } else {
      return Leaf {age = eventTime};
    }
  }
}
```

In [43]:
```
params = {
    "time": 5.0,
    "lambda": 1.0,
    "mu": 0.1
}

result = generative_crbd(**params)

tree = result.samples[0]
tree = Phylo.BaseTree.Clade(
    branch_length=params["time"] - tree.age,
    clades=[tree.to_biopython()]
)
Phylo.draw(tree)
```



EXAMPLE: INFERRING PARAMETERS OF A CRBD MODEL

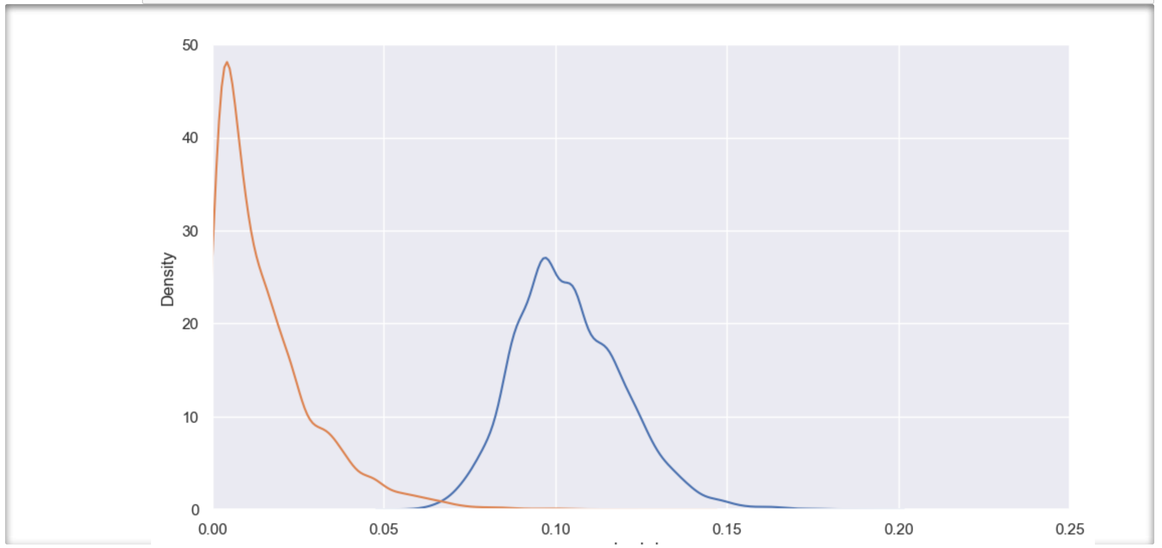
In [44]:

```
%%treeppl crbd samples=10000 subsamples=10

function simulateExtinctSubtree(time: Real, lambda: Real, mu: Real) {
  assume waitingTime ~ Exponential(lambda + mu);
  if waitingTime > time {
    weight 0.0; resample;
  } else {
    assume isSpeciation ~ Bernoulli(lambda / (lambda + mu));
    if isSpeciation {
      simulateExtinctSubtree(time - waitingTime, lambda, mu);
      simulateExtinctSubtree(time - waitingTime, lambda, mu);
    }
  }
}
function simulateUnobservedSpeciations(node: Tree, time: Real, lambda: Real, mu: Re
  assume waitingTime ~ Exponential(lambda);
  if time - waitingTime > node.age {
    simulateExtinctSubtree(time - waitingTime, lambda, mu);
    weight 2.0;
    simulateUnobservedSpeciations(node, time - waitingTime, lambda, mu);
  }
}
function walk(node: Tree, time:Real, lambda: Real, mu: Real) {
  simulateUnobservedSpeciations(node, time, lambda, mu);
  observe 0 ~ Poisson(mu * (time - node.age));
  if node is Node {
    observe 0.0 ~ Exponential(lambda);
    walk(node.left, node.age, lambda, mu);
    walk(node.right, node.age, lambda, mu);
  }
}
model function crbd(tree: Tree): Real[] {
  assume lambda ~ Gamma(1.0, 1.0);
  assume mu ~ Gamma(1.0, 0.5);
  walk(tree.left, tree.age, lambda, mu);
  walk(tree.right, tree.age, lambda, mu);
  return [lambda, mu];
}
```

EXAMPLE: INFERRING PARAMETERS OF A CRBD MODEL

In [44]:
```
%%treeppl crbd samples=10000 subsamples=10

function simulateExtinctSubtree(time: Real, lambda: Real, mu: Real) {
  assume waitingTime ~ Exponential(lambda + mu);
  if waitingTime > time {
    weight 0.0; resample;
  } else {
    assume isSpeciation ~ Bernoulli(lambda / (lambda + mu));
    if isSpeciation {
      simulateExtinctSubtree(time - waitingTime, lambda, mu);
      simulateExtinctSubtree(time - waitingTime, lambda, mu);
    }
  }
}
function simulateUnobservedSpeciations(node: Tree, time: Real, lambda: Real, mu: Re
  assume waitingTime ~ Exponential(lambda);
  if time - waitingTime > node.age {
    simulateExtinctSubtree(time - waitingTime, lambda, mu);
    weight 2.0;
    simulateUnobservedSpeciations(node, time - waitingTime, lambda, mu);
  }
}
function walk(node: Tree, time:Real, lambda: Real, mu: Real) {
  simulateUnobservedSpeciations(node, time, lambda, mu);
  observe 0 ~ Poisson(mu * (time - node.age));
  if node is Node {
    observe 0.0 ~ Exponential(lambda);
    walk(node.left, node.age, lambda, mu);
    walk(node.right, node.age, lambda, mu);
  }
}
model function crbd(tree: Tree): Real[] {
  assume lambda ~ Gamma(1.0, 1.0);
  assume mu ~ Gamma(1.0, 0.5);
  walk(tree.left, tree.age, lambda, mu);
  walk(tree.right, tree.age, lambda, mu);
  return [lambda, mu];
}
```

In [51]:
```
alcedinidae = treeppl.Tree.load("trees/Alcedinidae.phyjson", format="phyjson")
samples = None
for i in range(1000):
    try:
        res = crbd(tree=alcedinidae)
        samples = pd.concat([samples, pd.DataFrame({
            "lambda": res.items(0), "mu": res.items(1),
            "lweight": res.norm_const
        })])
        weights = np.exp(samples.lweight - samples.lweight.max())
        clear_output(wait=True)
        sns.kdeplot(data=samples, x="lambda", weights=weights)
        sns.kdeplot(data=samples, x="mu", weights=weights)
        plt.xlim(0, 0.25)
        plt.ylim(0, 50)
        plt.pause(0.05)
    except KeyboardInterrupt:
        break
```



TYPE BRIDGING: TREEPPL ↔ PYTHON

TreePPL custom types are mapped to Python classes, enabling intuitive data handling and model interaction.

TreePPL constructor ↔ Python class

TreePPL constructors are automatically mapped to Python classes during output processing.

If a corresponding class does not exist, new Python class is dynamically created:

In [46]:
```
%%treeppl ext_tree_model samples=1

type ExtTree =
  | ExtNode {left: ExtTree, right: ExtTree, age: Real, state: Int}
  | ExtLeaf {age: Real, state: Int}

model function ext_tree_model(): ExtTree {
  return ExtNode {
    left = ExtLeaf { age = 0.0, state = 0 },
    right = ExtLeaf { age = 0.5, state = 1 },
    age = 0.72,
    state = 1
  };
}
```

TYPE BRIDGING: TREEPPL ↔ PYTHON

TreePPL custom types are mapped to Python classes, enabling intuitive data handling and model interaction.

TreePPL constructor ↔ Python class

TreePPL constructors are automatically mapped to Python classes during output processing.

If a corresponding class does not exist, new Python class is dynamically created:

In [46]:
```
%%treeppl ext_tree_model samples=1

type ExtTree =
  | ExtNode {left: ExtTree, right: ExtTree, age: Real, state: Int}
  | ExtLeaf {age: Real, state: Int}

model function ext_tree_model(): ExtTree {
  return ExtNode {
    left = ExtLeaf { age = 0.0, state = 0 },
    right = ExtLeaf { age = 0.5, state = 1 },
    age = 0.72,
    state = 1
  };
}
```

In [48]:
```
tree = ext_tree_model().samples[0]

print("tree:", tree)
print()
print("left:", tree.left)
print("right:", tree.right)
print("age:", tree.age)
print("state:", tree.state)
```

```
tree: <treeppl.serialization.ExtNode object at 0x16886b790>

left: <treeppl.serialization.ExtLeaf object at 0x16886aa50>
right: <treeppl.serialization.ExtLeaf object at 0x16886bcd0>
age: 0.72
state: 1
```

CUSTOM CLASS DEFINITIONS

It is possible to define own Python classes for TreePPL custom types and link them to their constructors using the `@treeppl.constructor` decorator.

Example

In [49]:
```
@treeppl.constructor("ExtNode")
class ExtNode(treeppl.Object):
    def __repr__(self):
        return f"ExtNode(left={self.left!r}, right={self.right!r}, " \
               f"age={self.age}, state={self.state})"


@treeppl.constructor("ExtLeaf")
class ExtLeaf(treeppl.Object):
    def __repr__(self):
        return f"ExtLeaf(age={self.age}, state={self.state})"
```

CUSTOM CLASS DEFINITIONS

It is possible to define own Python classes for TreePPL custom types and link them to their constructors using the `@treeppl.constructor` decorator.

Example

In [49]:
```
@treeppl.constructor("ExtNode")
class ExtNode(treeppl.Object):
    def __repr__(self):
        return f"ExtNode(left={self.left!r}, right={self.right!r}, " \
               f"age={self.age}, state={self.state})"


@treeppl.constructor("ExtLeaf")
class ExtLeaf(treeppl.Object):
    def __repr__(self):
        return f"ExtLeaf(age={self.age}, state={self.state})"
```

In [50]:
```
tree = ext_tree_model().samples[0]
tree
```

Out[50]:
```
ExtNode(left=ExtLeaf(age=0.0, state=0), right=ExtLeaf(age=0.5, state=1), age=0.
72, state=1)
```

CUSTOM CLASS DEFINITIONS

It is possible to define own Python classes for TreePPL custom types and link them to their constructors using the `@treeppl.constructor` decorator.

Example

In [49]:
```
@treeppl.constructor("ExtNode")
class ExtNode(treeppl.Object):
    def __repr__(self):
        return f"ExtNode(left={self.left!r}, right={self.right!r}, " \
               f"age={self.age}, state={self.state})"


@treeppl.constructor("ExtLeaf")
class ExtLeaf(treeppl.Object):
    def __repr__(self):
        return f"ExtLeaf(age={self.age}, state={self.state})"
```

In [50]:
```
tree = ext_tree_model().samples[0]
tree
```

Out[50]:
```
ExtNode(left=ExtLeaf(age=0.0, state=0), right=ExtLeaf(age=0.5, state=1), age=0.
72, state=1)
```

When to Use Custom Classes

- Optional: For output processing, automatic class creation suffices.
- Required: If custom types are used for model arguments, define your own classes to bridge input.

THANKS FOR LISTENING

Try TreePPL, `treeppl-python` and the Jupyter extension today!

- http://treeppl.org
- https://github.com/treeppl