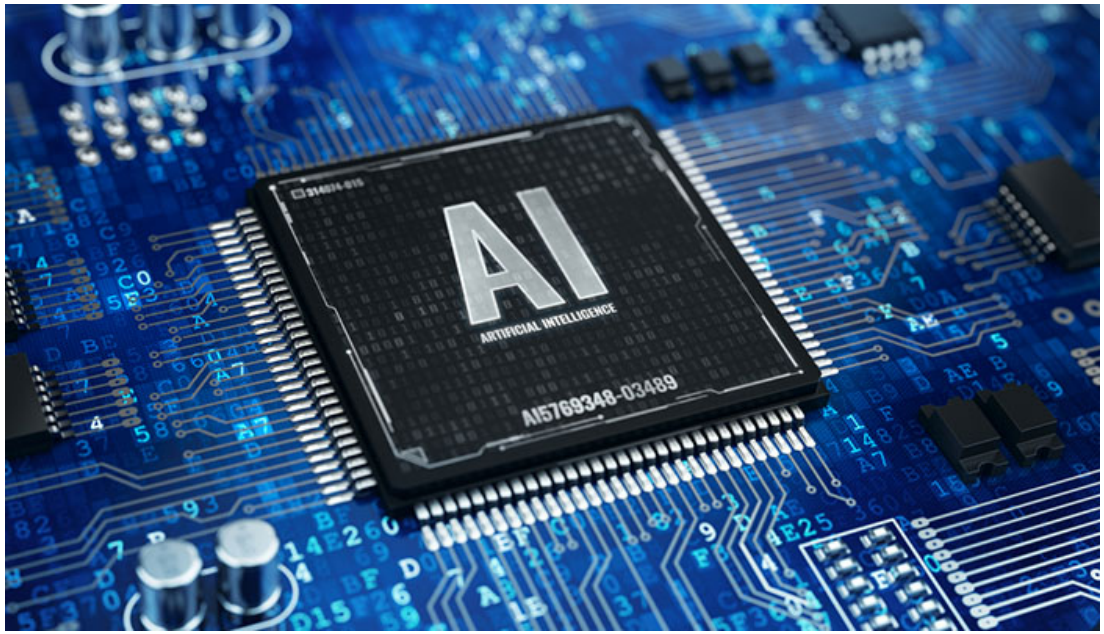
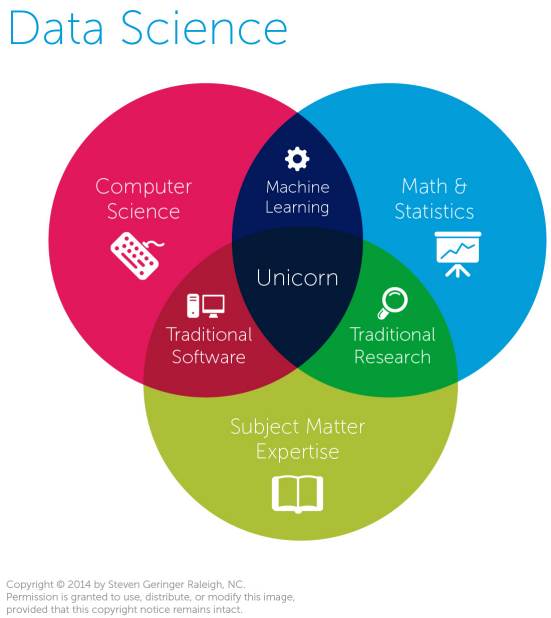
Introduction to probabilistic programming languages (PPLs)

Andrés Masegosa and Thomas D. Nielsen

Why PPLs?



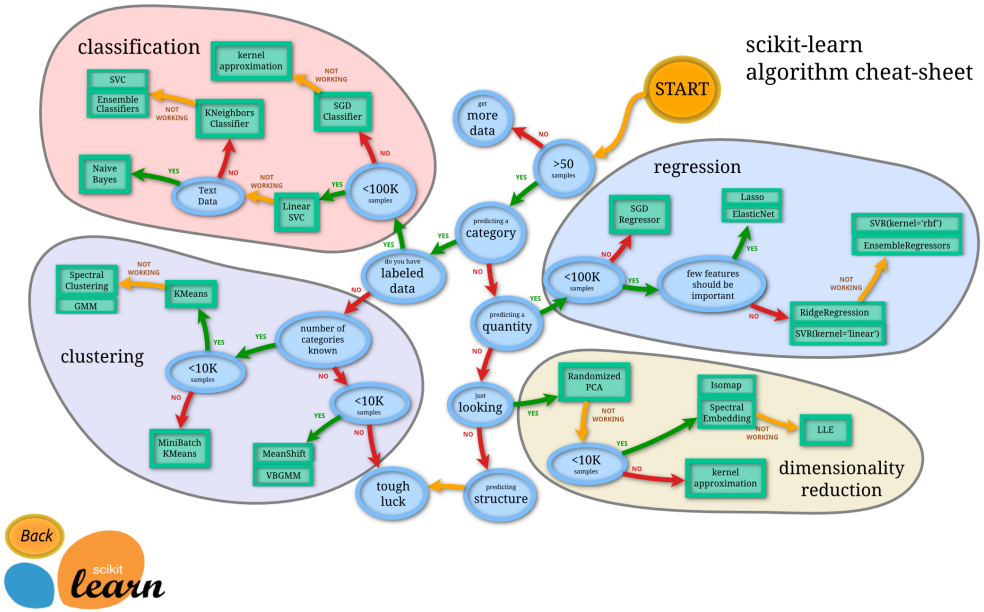
The development of **machine learning systems** requires enormous efforts.



The development of machine learning systems requires enormous efforts.

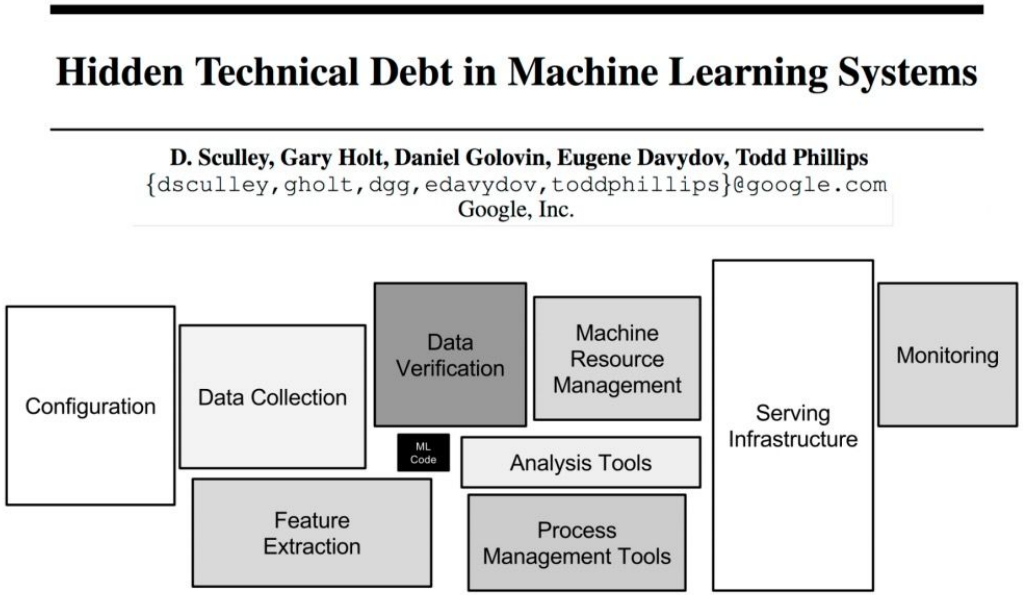
- **It requires of highly qualified experts.**

Why PPLs?



The development of machine learning applications requires enormous effort.

- It is necessary to have highly qualified experts.
- **It is difficult to find the ML model most suitable for an application.**

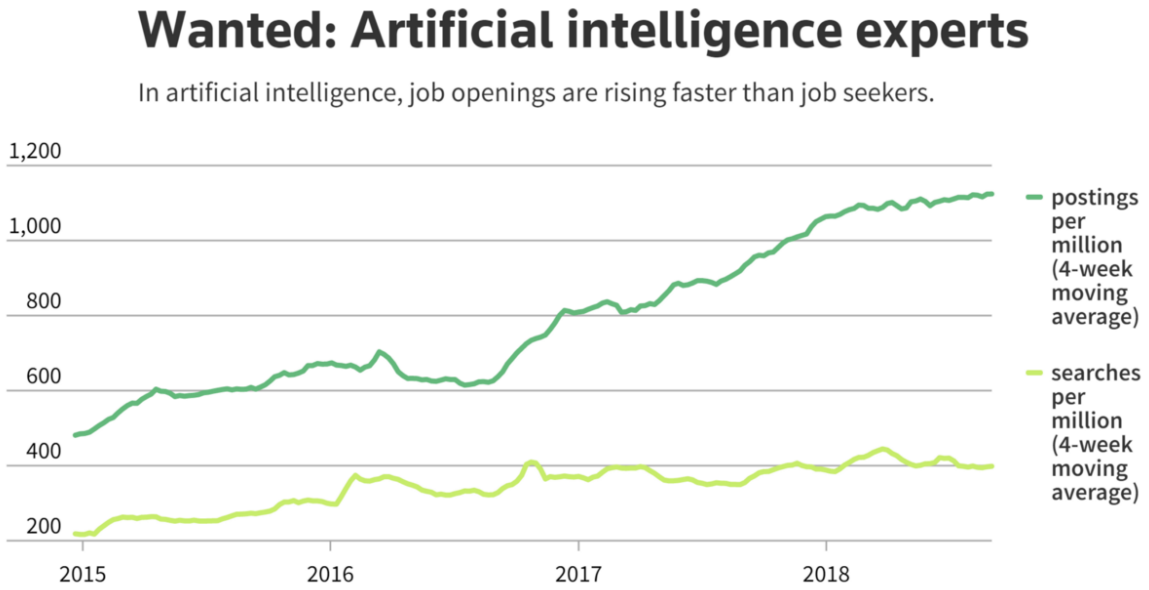


Hidden Technical Debt in Machine Learning Systems

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips
{dsculley,gholt,dgg,edavydov,toddphillips}@google.com
Google, Inc.

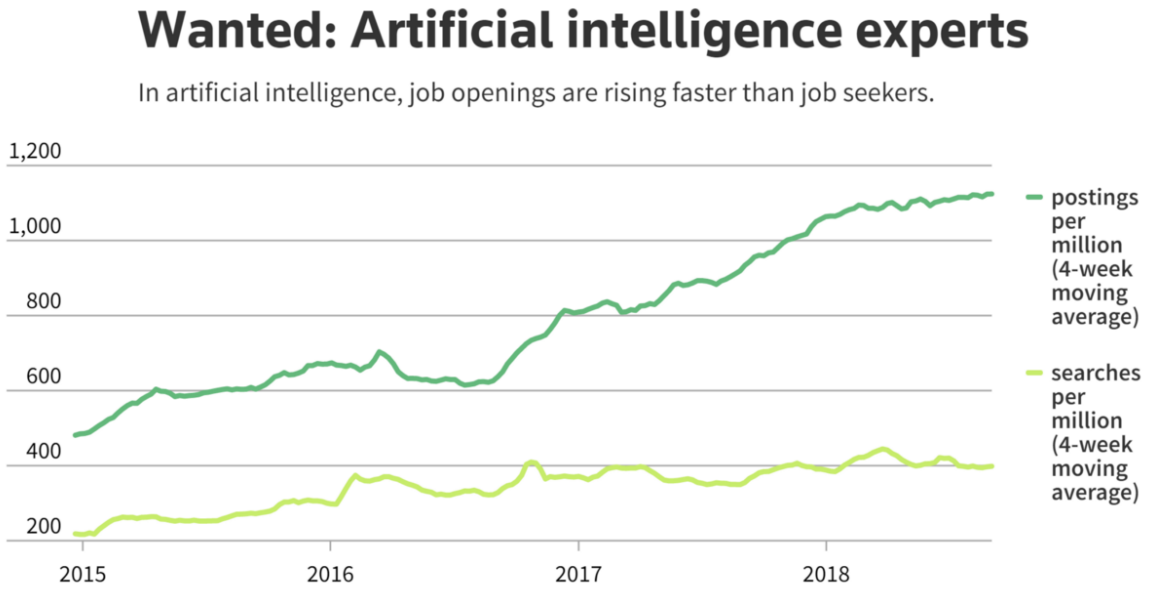
The development of machine learning applications requires enormous effort.

- It is necessary to have highly qualified experts.
- It is difficult to find the ML model most suitable for an application.
- **Programming a ML model is a complex task where many problems are intermingled**.



Consequences:

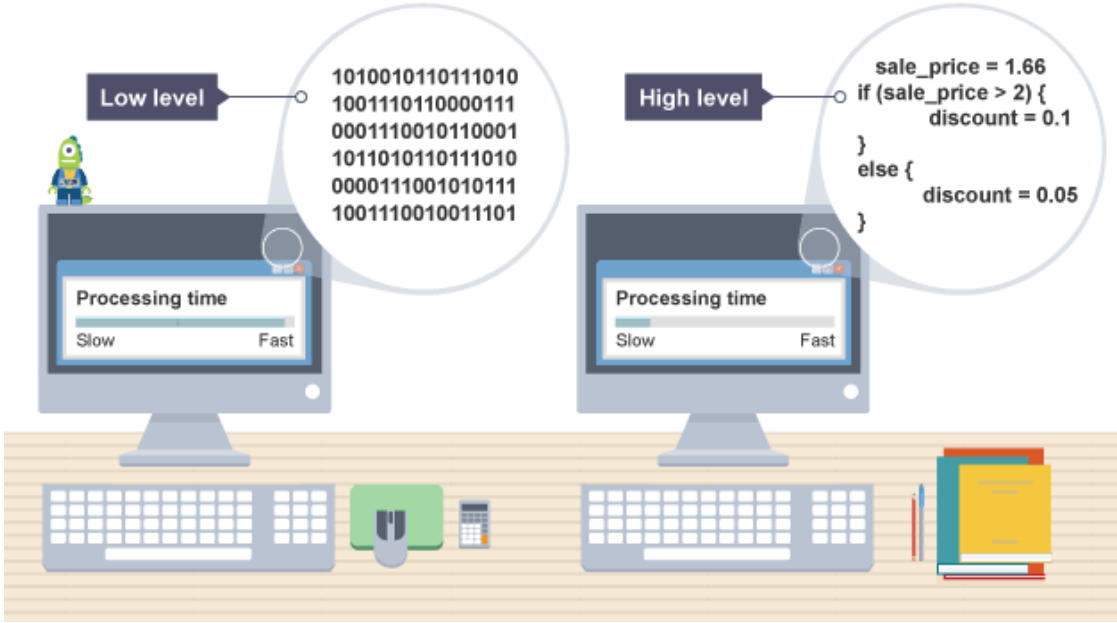
- Shortage of AI experts (and high salaries).



Consequences:

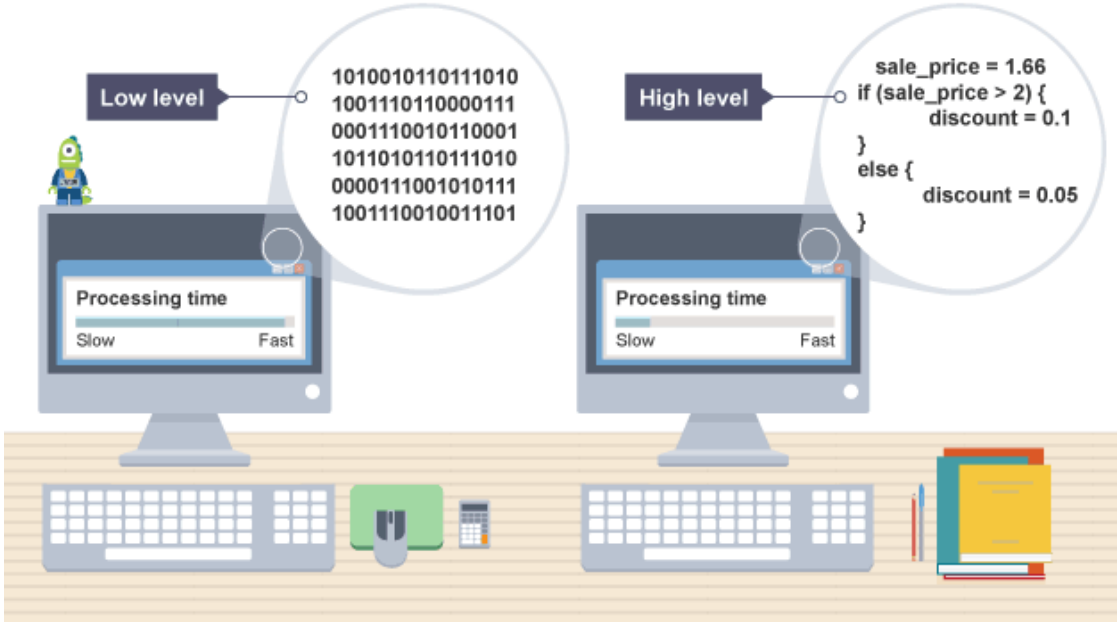
- Shortage of AI experts (and high salaries).
- Only big corporations have the resources for developing ML systems.

Why PPLs?



Similar situation than 50 years ago:

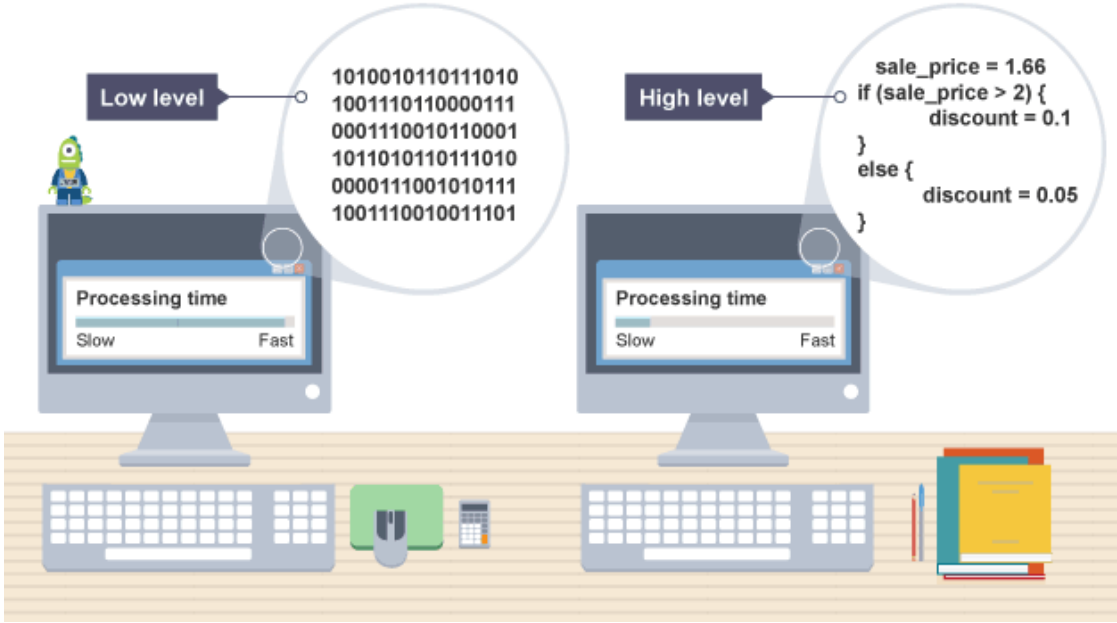
Why PPLs?



Similar situation than 50 years ago:

- People used to program in low-level programming languages.

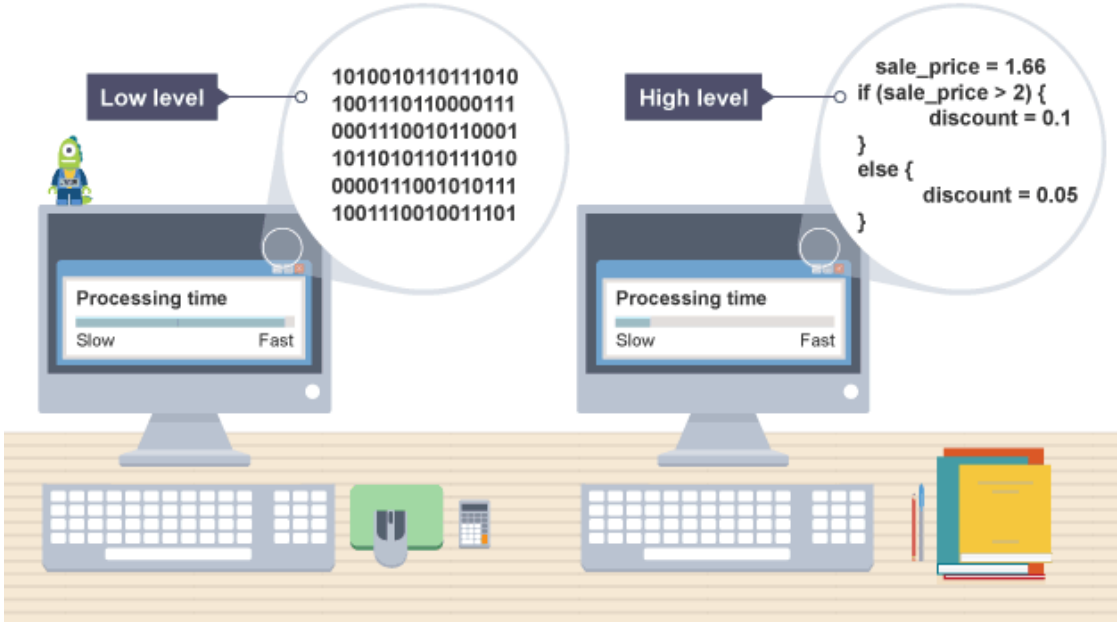
Why PPLs?



Similar situation than 50 years ago:

- People used to program in low-level programming languages.
- Programming was complex and demand high-expertise.

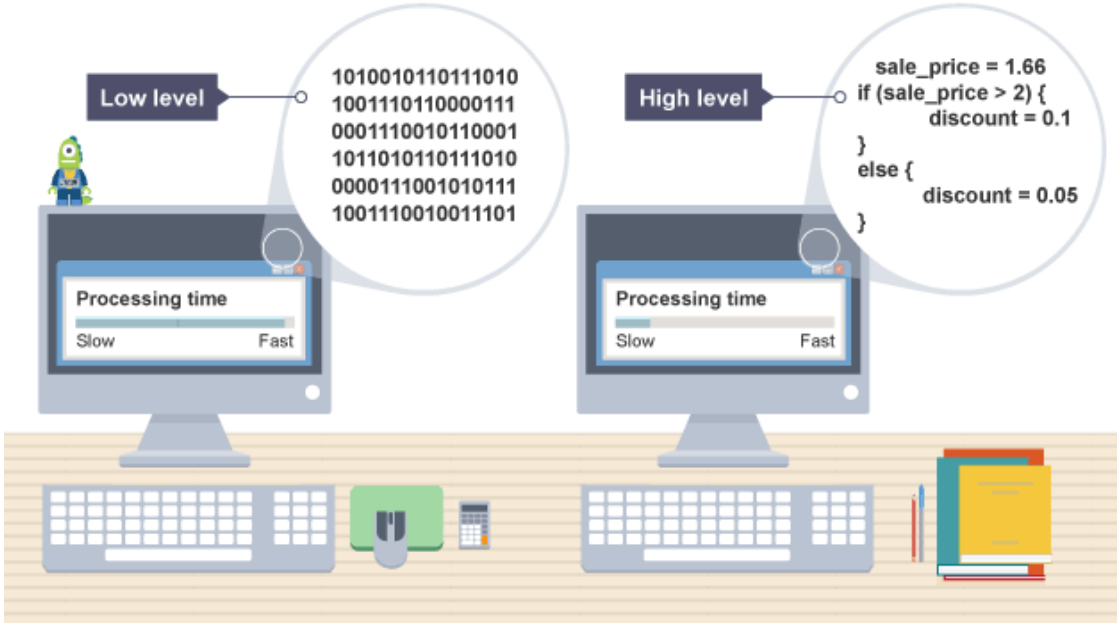
Why PPLs?



Similar situation than 50 years ago:

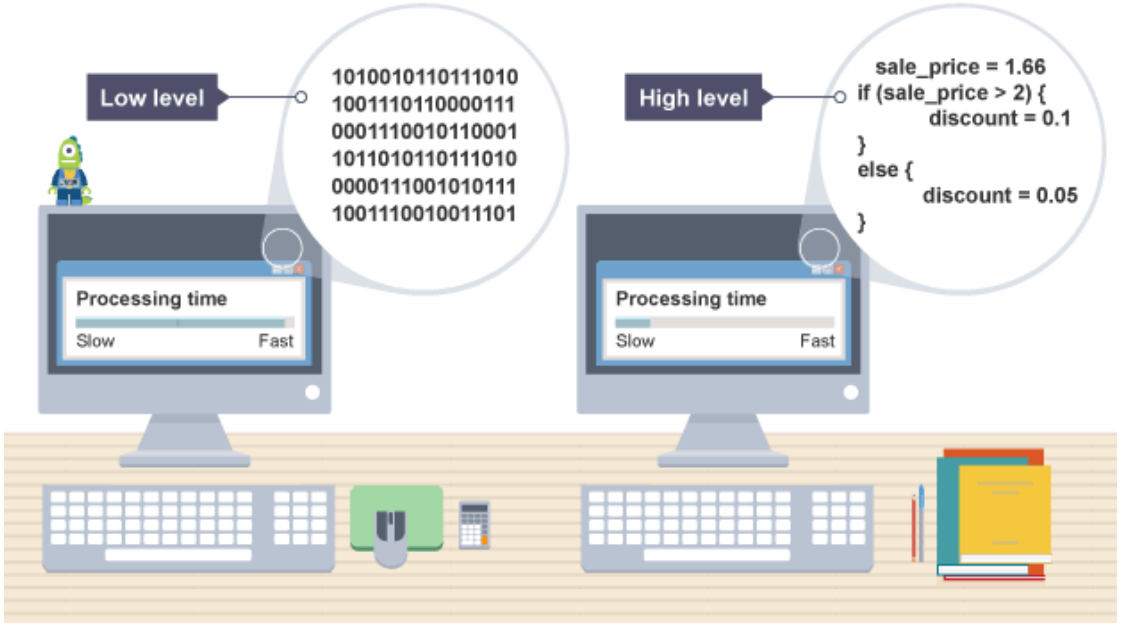
- People used to program in low-level programming languages.
- Programming was complex and demand high-expertise.
- Focus on application and low-level hardware details.

Why PPLs?



High-level programming languages brought many advantages:

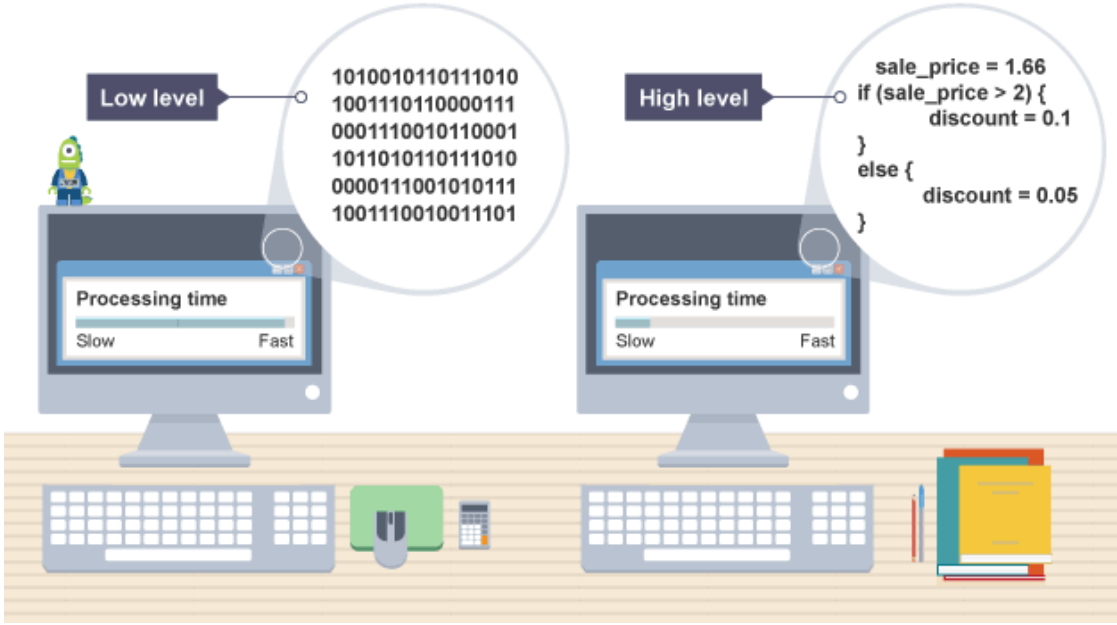
Why PPLs?



High-level programming languages brought many advantages:

- Programmers focused on the applications.

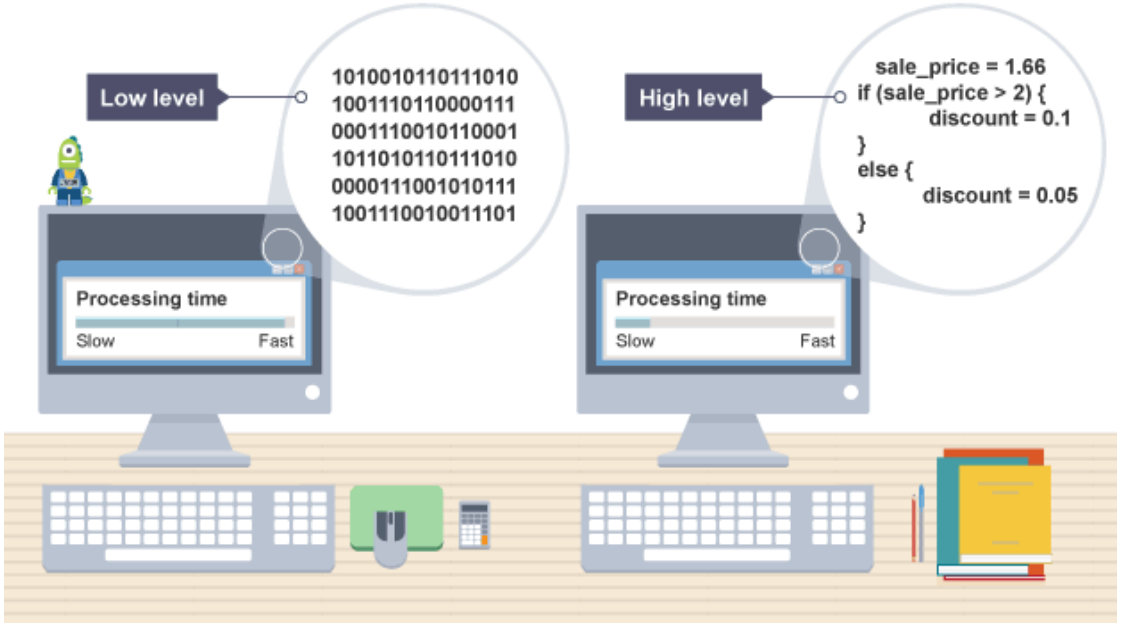
Why PPLs?



High-level programming languages brought many advantages:

- Programmers focused on the applications.
- Hardware Experts focused on compilers.

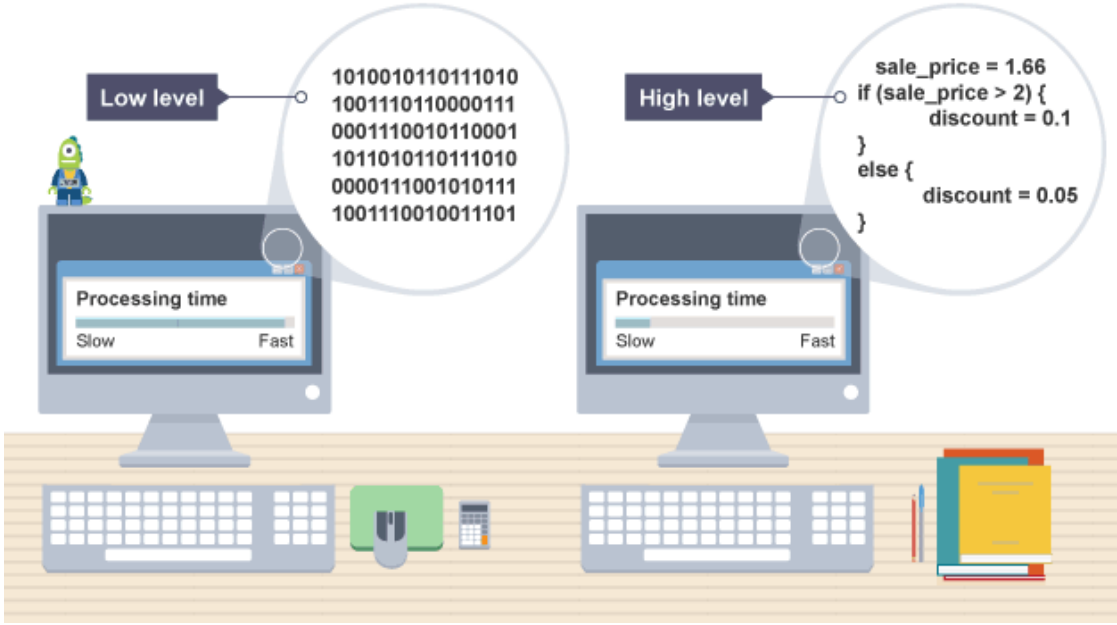
Why PPLs?



High-level programming languages brought many advantages:

- Programmers focused on the applications.
- Hardware Experts focused on compilers.
- High gains in productivity.

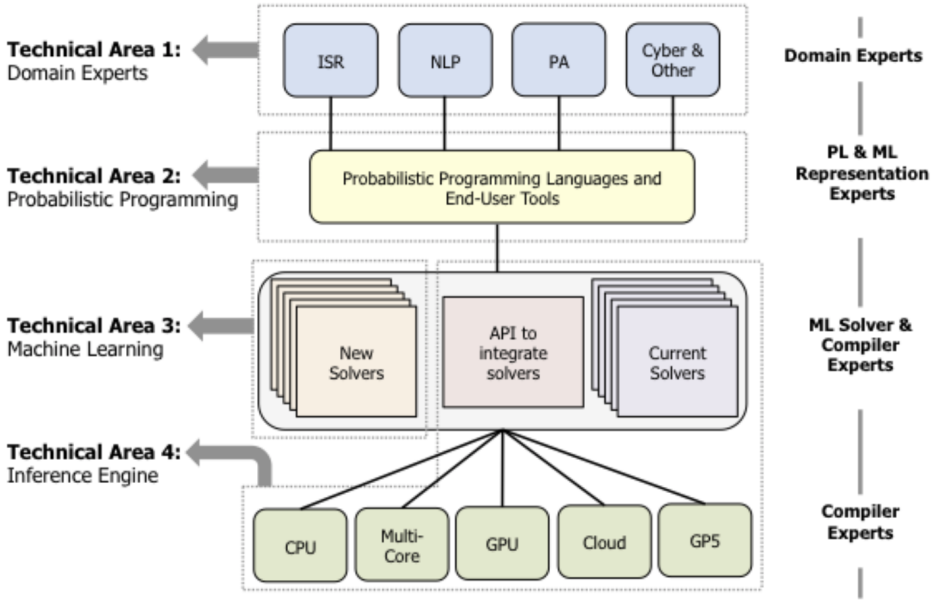
Why PPLs?



High-level programming languages brought many advantages:

- Programmers focused on the applications.
- Hardware Experts focused on compilers.
- High gains in productivity.
- "Democratization" of the software development.

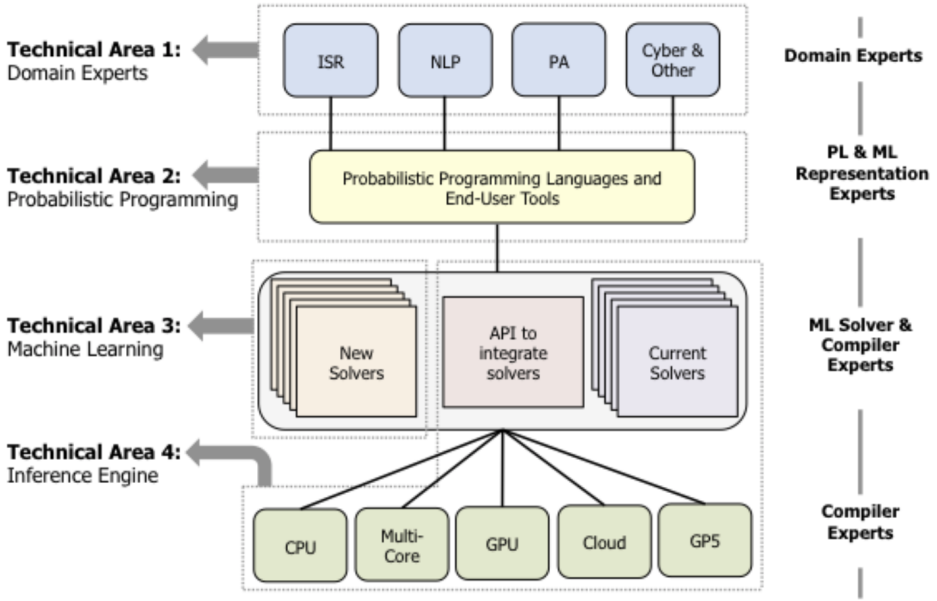
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture

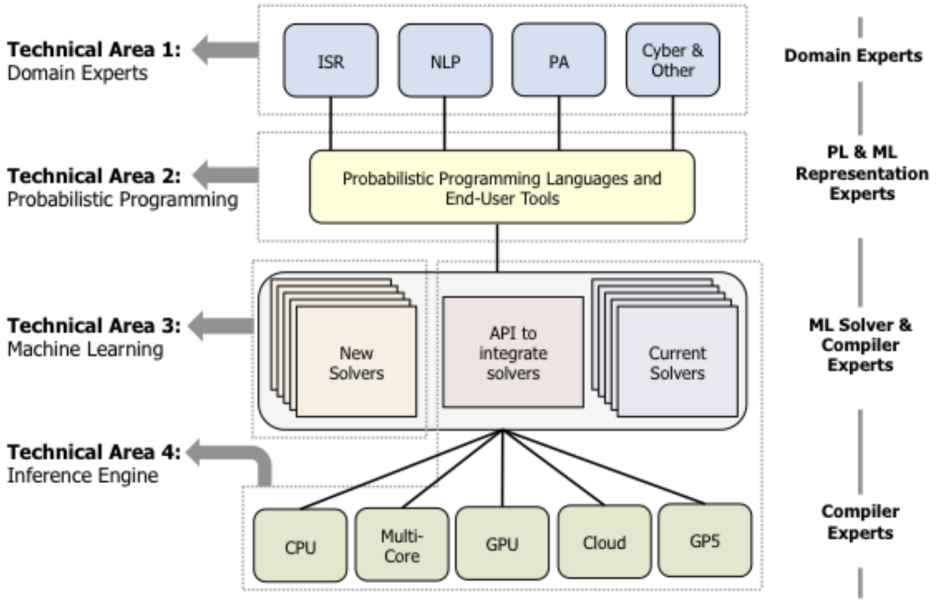
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture
- Different Domain Experts will code their models using the same language.

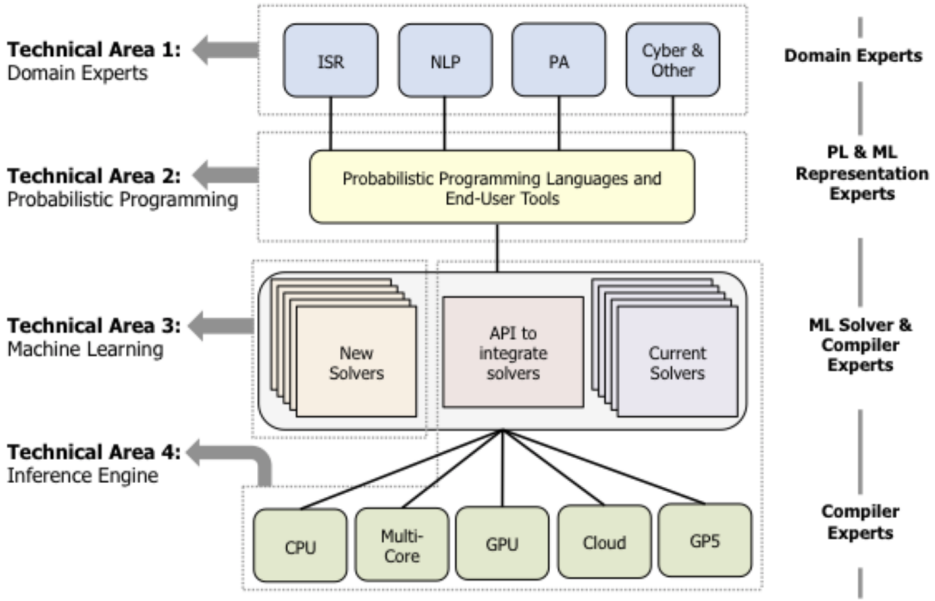
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture
- Different Domain Experts will code their models using the same language.
- ML experts will focus on the development of new ML solvers.

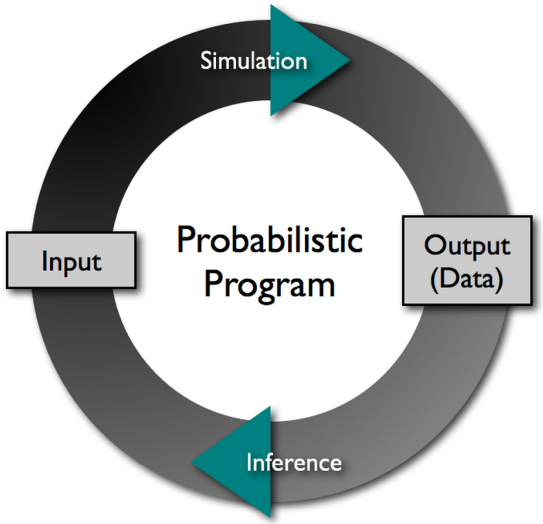
Why PPLs?



PPLs as high-level programming languages for **machine learning systems**:

- Stacked architecture
- Different Domain Experts will code their models using the same language.
- ML experts will focus on the development of new ML solvers.
- Compile experts will focus on running these ML solvers on specialized hardware.

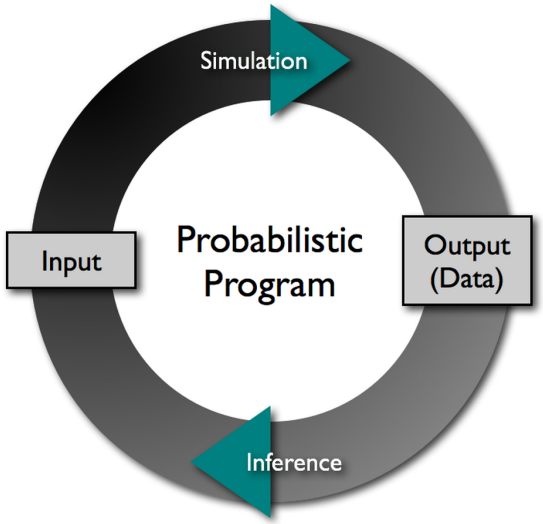
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.

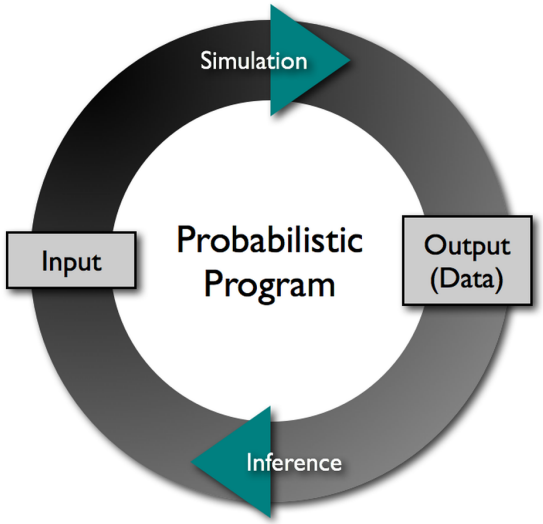
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.

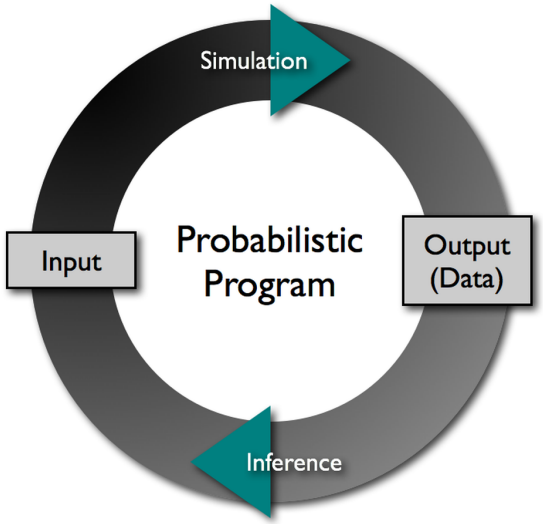
Why PPLs?



Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.
- Facilitate the construction of more sophisticated models.

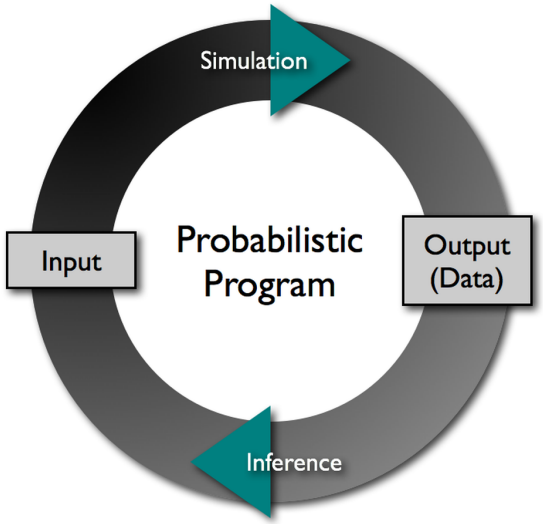
Why PPLs?



Benefits of PPLs:

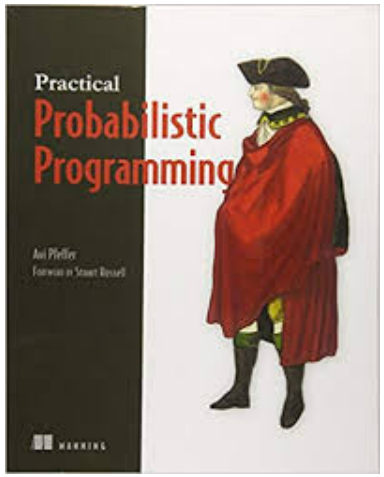
- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.
- Facilitate the construction of more sophisticated models.
- Reduce the necessary level of expertise.

Why PPLs?



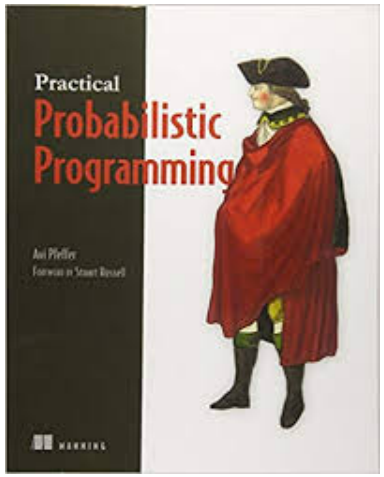
Benefits of PPLs:

- Simplify machine learning model code.
- Reduce development time and cost to encourage experimentation.
- Facilitate the construction of more sophisticated models.
- Reduce the necessary level of expertise.
- "Democratization" of the development of ML systems.



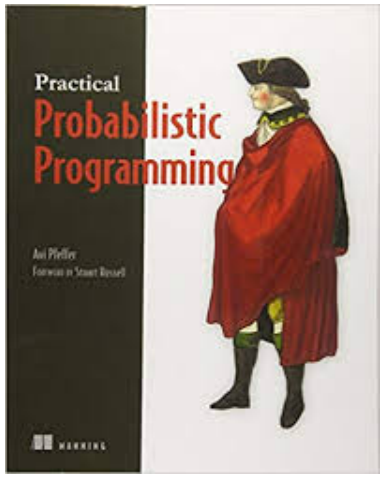
1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.



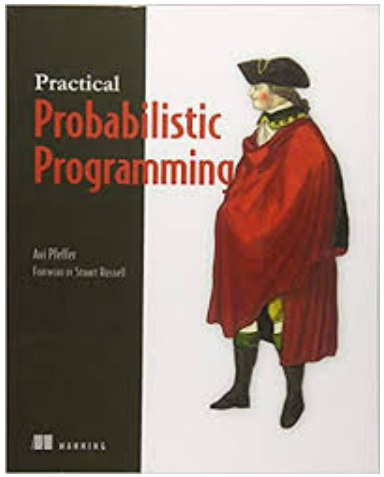
1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.
- Turing-complete probabilistic programming languages. (i.e. they can represent any computable probability distribution).



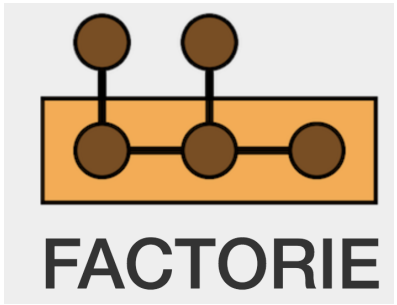
1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.
- Turing-complete probabilistic programming languages. (i.e. they can represent any computable probability distribution).
- Inference engine based on Monte Carlo methods.



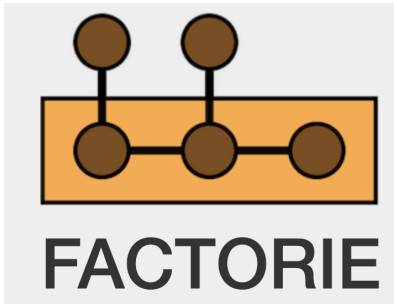
1st Generation of PPLs :

- Bugs, WinBugs, Jags, Figaro, etc.
- Turing-complete probabilistic programming languages. (i.e. they can represent any computable probability distribution).
- Inference engine based on Monte Carlo methods.
- They did not scale to large data samples/high-dimensional models.



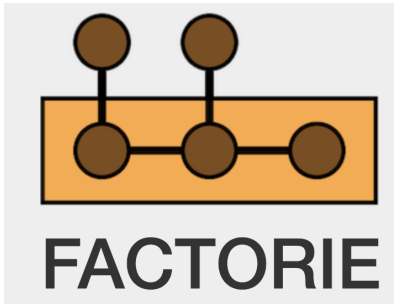
2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.



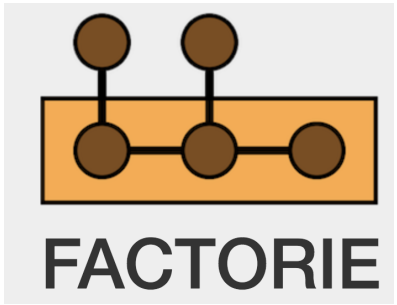
2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.
- Inference engine based on message passage algorithms and/or variational inference methods.



2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.
- Inference engine based on message passage algorithms and/or variational inference methods.
- They did scale to large data samples/high-dimensional models.



2nd Generation of PPLs :

- Infer.net, Factorie, Amidst, etc.
- Inference engine based on message passage algorithms and/or variational inference methods.
- They did scale to large data samples/high-dimensional models.
- Restricted probabilistic model family (i.e. factor graphs, conjuage exponential family, etc.)

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- Black Box Variational Inference and Hamiltonian Monte-Carlo.

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- Black Box Variational Inference and Hamiltonian Monte-Carlo.
- They did scale to large data samples/high-dimensional models.



3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- Black Box Variational Inference and Hamiltonian Monte-Carlo.
- They did scale to large data samples/high-dimensional models.
- Turing-complete probabilistic programming languages.

3rd Generation of PPLs :

- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- Black Box Variational Inference and Hamiltonian Monte-Carlo.
- They did scale to large data samples/high-dimensional models.
- Turing-complete probabilistic programming languages.
- Rely on deep learning frameworks (TensorFlow, Pytorch, Theano, etc).



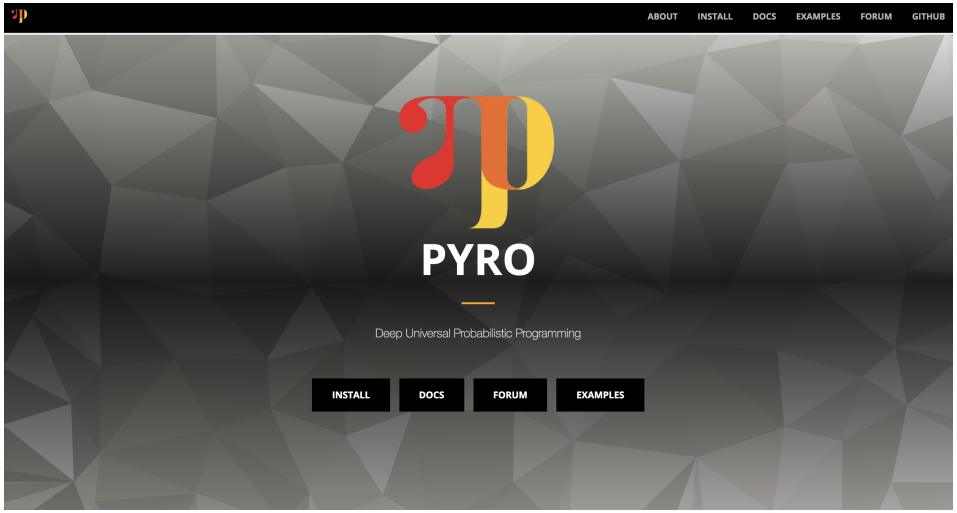
3rd Generation of PPLs :

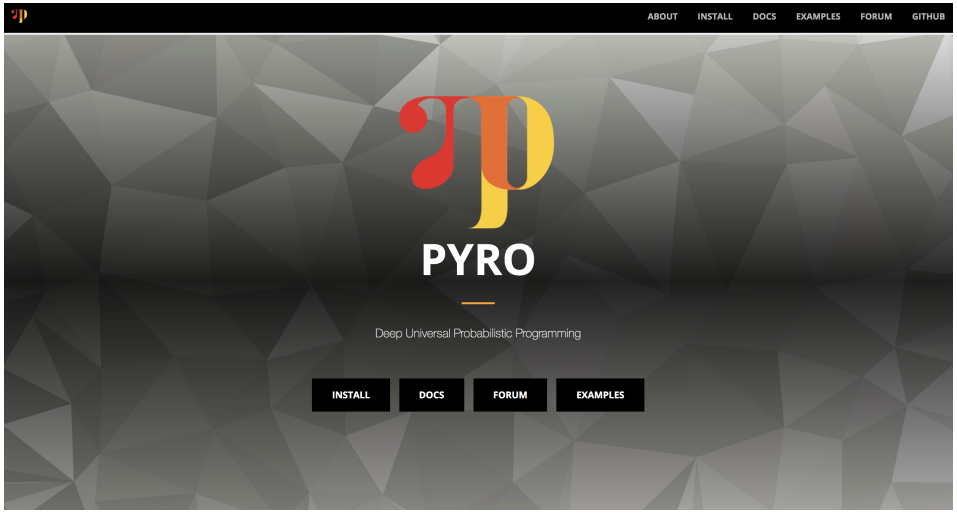
- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- Black Box Variational Inference and Hamiltonian Monte-Carlo.
- They did scale to large data samples/high-dimensional models.
- Turing-complete probabilistic programming languages.
- Rely on deep learning frameworks (TensorFlow, Pytorch, Theano, etc).
  - Specialized hardware like GPUs, TPUs, etc.



3rd Generation of PPLs :

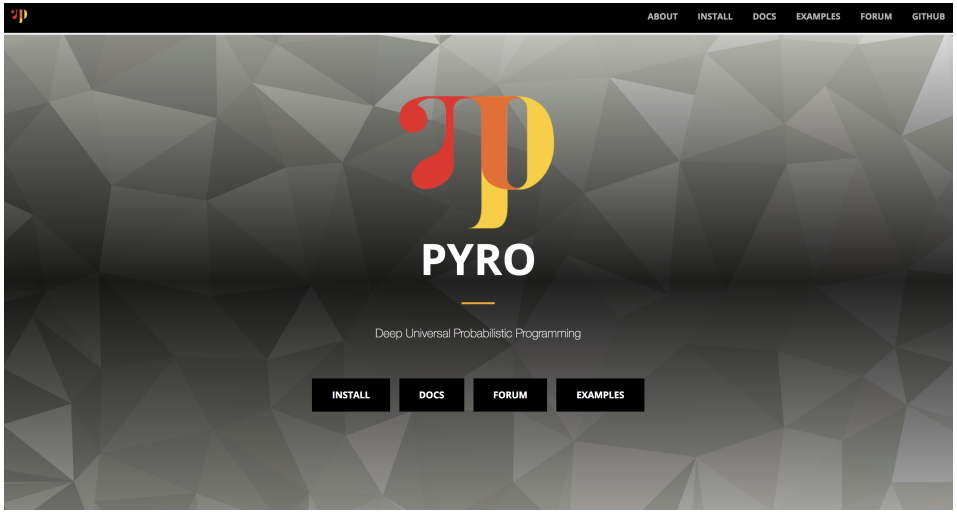
- TensorFlow Probability, Pyro, PyMC3, InferPy, etc.
- Black Box Variational Inference and Hamiltonian Monte-Carlo.
- They did scale to large data samples/high-dimensional models.
- Turing-complete probabilistic programming languages.
- Rely on deep learning frameworks (TensorFlow, Pytorch, Theano, etc).
  - Specialized hardware like GPUs, TPUs, etc.
  - Automatic differentiation methods.


ABOUT
INSTALL
DOCS
EXAMPLES
FORUM
GITHUB
PYRO
Deep Universal Probabilistic Programming
INSTALL
DOCS
FORUM
EXAMPLES



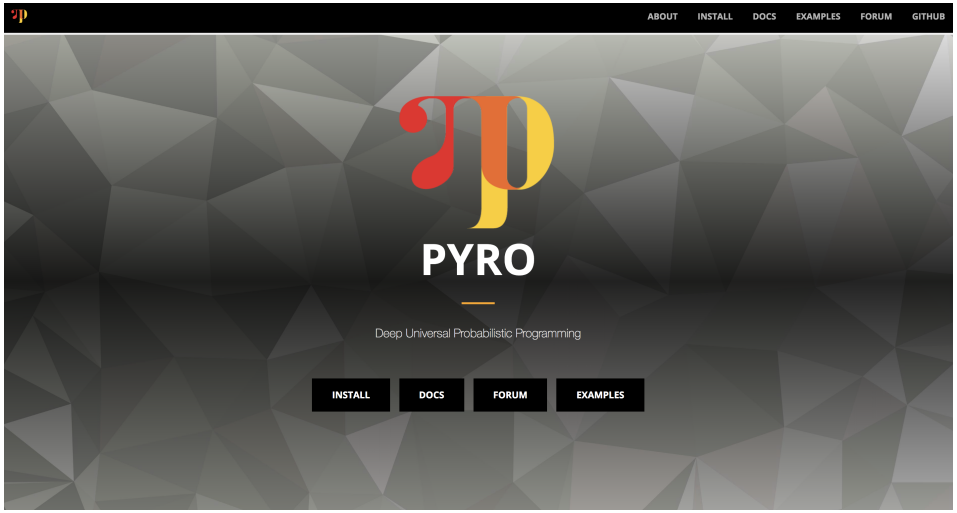
Pyro's main features (`www.pyro.ai`) :

- Developed by UBER (the car riding company).



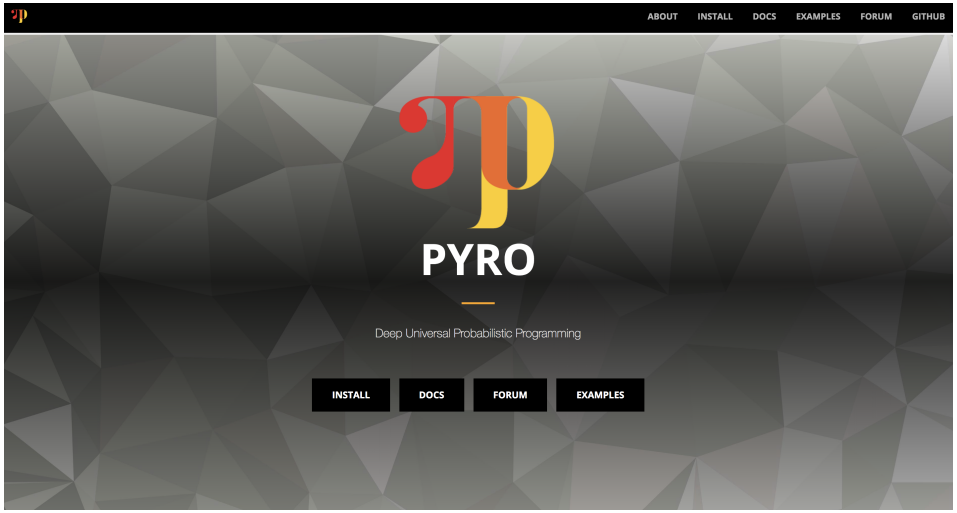
Pyro's main features (`www.pyro.ai`) :

- Developed by UBER (the car riding company).
- Focus on deep generative models (Day 4).



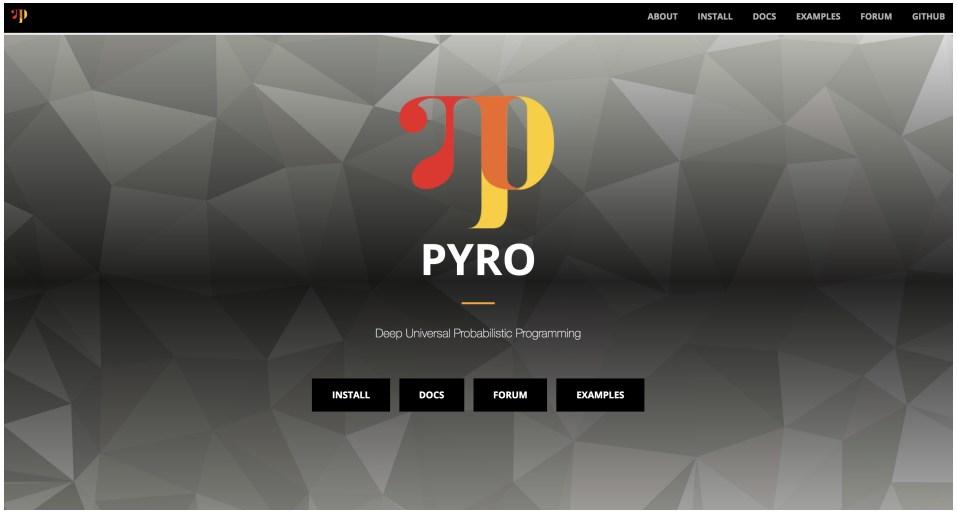
Pyro's main features (`www.pyro.ai`) :

- Developed by UBER (the car riding company).
- Focus on deep generative models (Day 4).
- Rely on Pytorch (Deep Learning Framework).



Pyro's main features (`www.pyro.ai`) :

- Developed by UBER (the car riding company).
- Focus on deep generative models (Day 4).
- Rely on Pytorch (Deep Learning Framework).
- Enable GPU accelaration and distributed learning.



Pyro's main features (`www.pyro.ai`) :

- Developed by UBER (the car riding company).
- Focus on deep generative models (Day 4).
- Rely on Pytorch (Deep Learning Framework).
- Enable GPU accelaration and distributed learning.

`Day1/students_PPLs_Intro.ipynb`

```
In [3]: normal = dist.Normal(0,1)
        normal

Out[3]: Normal(loc: 0.0, scale: 1.0)
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.

```
In [15]: sample = normal.sample()
         sample

Out[15]: tensor(0.4908)
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).

```
In [17]: sample = normal.sample(sample_shape=[3,4,5])
         sample.shape

Out[17]: torch.Size([3, 4, 5])
```

Pyro's distributions (http://docs.pyro.ai/en/stable/distributions.html) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).

```
In [19]: torch.sum(normal.log_prob(sample))

Out[19]: tensor(-85.1003)
```

Pyro's distributions (http://docs.pyro.ai/en/stable/distributions.html) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).

```
In [9]: normal = dist.Normal(torch.tensor([1.,2.,3.]),1.)
        normal

Out[9]: Normal(loc: torch.Size([3]), scale: torch.Size([3]))
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).
- Multiple distributions can be embedded in single object (to define efficient vectorized operations).

```
In [10]: normal.sample()

Out[10]: tensor([2.0592, 2.4035, 3.1918])
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).
- Multiple distributions can be embedded in single object (to define efficient vectorized operations).

```
In [11]: normal.log_prob(normal.sample())
Out[11]: tensor([-0.9402, -1.2113, -2.3214])
```

Pyro's distributions (`http://docs.pyro.ai/en/stable/distributions.html`) :

- Wide range of distributions: Normal, Beta, Cauchy, Dirichlet, Gumbel, Poisson, Pareto, etc.
- Samples from the distributions are Pytorch's Tensor objects (i.e. multidimensional arrays).
- Operations, like log-likelihood, are defined over tensors (with GPU acceleration powered by Pytorch).
- Multiple distributions can be embedded in single object (to define efficient vectorized operations).

Now it is your turn!

Open the Notebook and Play around

- Test you have installed the basic packages.
- Test you can run the first lines of code.
- Play a bit with the code.

`Day1/students_PPLs_Intro.ipynb`

In [12]:
```
def model():
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    return temp

print(model())
print(model())
```

```
tensor(12.3926)
tensor(22.5272)
```

Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A probabilistic model is defined as a **stochastic function**.

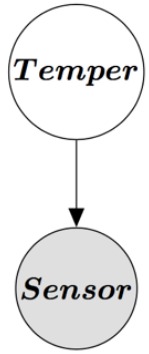
In [12]:
```
def model():
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    return temp

print(model())
print(model())
```

```
tensor(12.3926)
tensor(22.5272)
```

Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A probabilistic model is defined as a **stochastic function**.
- Each random variable is associated to a **primitive stochastic function** using the construct **pyro.sample(...)**.

```
In [21]: def model():
             temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
             sensor = pyro.sample('sensor', dist.Normal(temp, 1.0))
             return (temp, sensor)

         out1 = model()
         out1

Out[21]: (tensor(15.8576), tensor(16.9907))
```

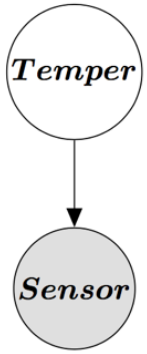


Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A **stochastic function** can be defined as a composition of **primitive stochastic functions**.

```
In [21]: def model():
             temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
             sensor = pyro.sample('sensor', dist.Normal(temp, 1.0))
             return (temp, sensor)

         out1 = model()
         out1
```

```
Out[21]: (tensor(15.8576), tensor(16.9907))
```



Pyro's models (http://pyro.ai/examples/intro_part_i.html) :

- A **stochastic function** can be defined as a composition of **primitive stochastic functions**.
- We define a **joint probability distribution**:

$$p(sensor, temp) = p(sensor|temp)p(temp)$$

In [22]:
```
#The observatons
obs = {'sensor': torch.tensor(18.0)}

def model(obs):
    temp = pyro.sample('temp', dist.Normal(15.0, 2.0))
    sensor = pyro.sample('sensor', dist.Normal(temp, 1.0),obs=obs['sensor'])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- We can introduce observations (e.g. sensor $= 18.0$).

In [27]:
```
#Run inference
svi(model,guide,obs, plot=True)

#Print results
print("P(Temperature|Sensor=18.0) = ")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- We can introduce observations (e.g. sensor $= 18.0$).
- We can query the **posterior probability distribution**:

$$p(temp|sensor = 18) = \frac{p(sensor = 18|temp)p(temp)}{\int p(sensor = 18|temp)p(temp)dtemp}$$

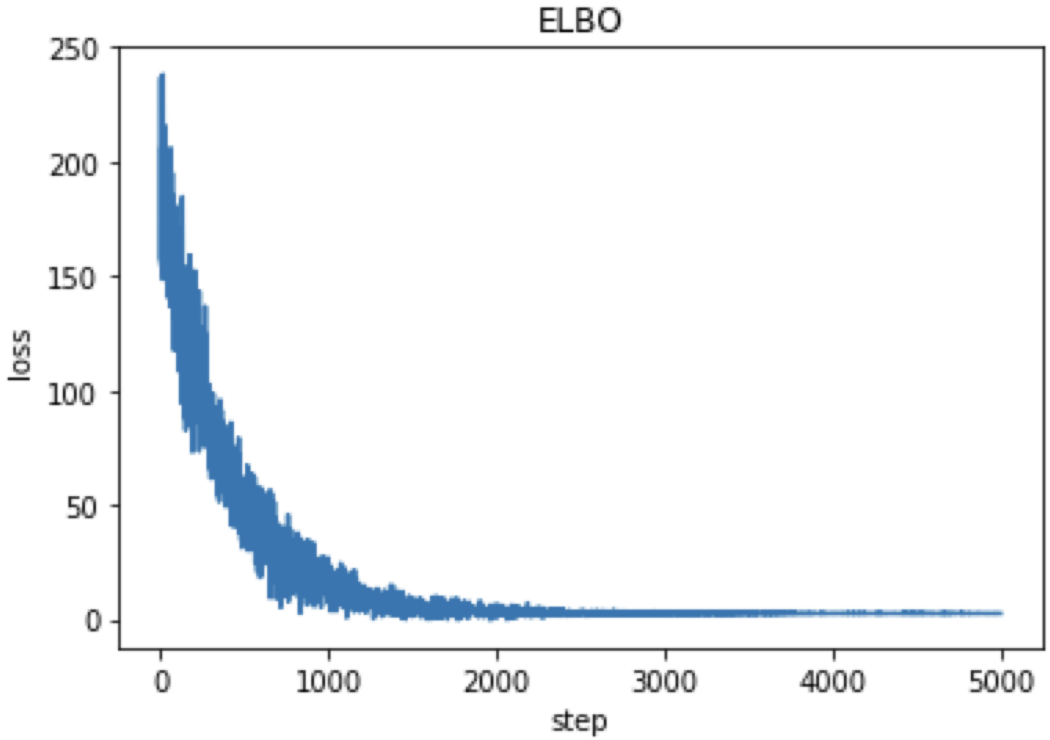
In [27]:
```
#Run inference
svi(model,guide,obs, plot=True)

#Print results
print("P(Temperature|Sensor=18.0) = ")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

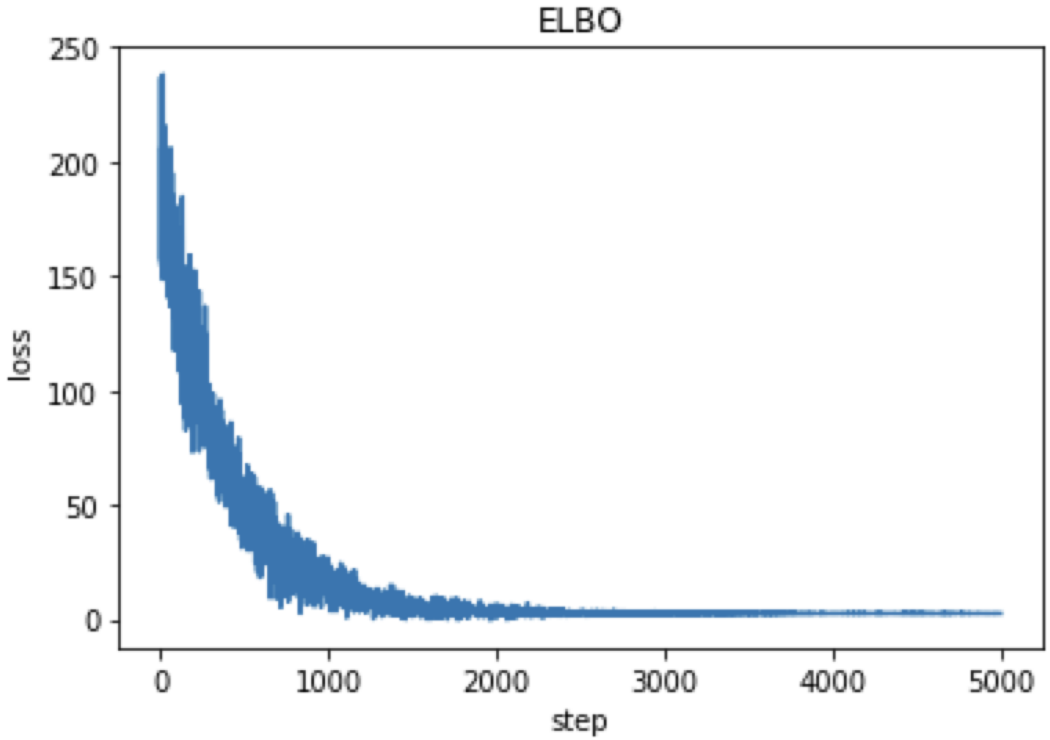
- We can introduce observations (e.g. sensor $= 18.0$).
- We can query the **posterior probability distribution**:

$$p(temp|sensor = 18) = \frac{p(sensor = 18|temp)p(temp)}{\int p(sensor = 18|temp)p(temp)dtemp}$$

- Guide is an auxiliary method needed for inference (more details on Day 3).




```
P(Temperature|Sensor=18.0) =
Normal(loc: 17.39859390258789, scale: 0.9089401960372925)
```




```
P(Temperature|Sensor=18.0) =
Normal(loc: 17.39859390258789, scale: 0.9089401960372925)
```

Details on the inference method will be given on Day 3.

In [ ]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(15.0, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if we have a **bunch of observations**, $\mathbf{s} = \{s_1, \ldots, s_n\}$

In [ ]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(15.0, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if we have a **bunch of observations**, $\mathrm{s} = \{s_1, \ldots, s_n\}$
- A random variable is created for each observation (using a **for-loop**).

In [ ]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(15.0, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if **we do not know the average temperature**?

In [28]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.param('mean_temp', torch.tensor(15.0))
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if **we do not know the average temperature**?
- We can introduce a **parameter** using **pyro.param** construct.

```
In [32]: #Run inference
svi(model, guide, obs, num_steps=1000)

#Print results
print("Estimated Mean Temperature")
print(pyro.param("mean_temp").item())
```

```
Estimated Mean Temperature
19.129146575927734
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if **we do not know the average temperature**?
- We can introduce a **parameter** using **pyro.param** construct.
- And perform **maximum likelihood estimation**:

$$\mu_t = \arg\max_{\mu} \ln p(s_1, \ldots, s_n|\mu) = \arg\max_{\mu} \prod_i \int_{t_i} p(s_i|t_i)p(t_i|\mu)dt_i$$

- Which is performed with the same general inference algorithm.

In [28]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.param('mean_temp', torch.tensor(15.0))
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

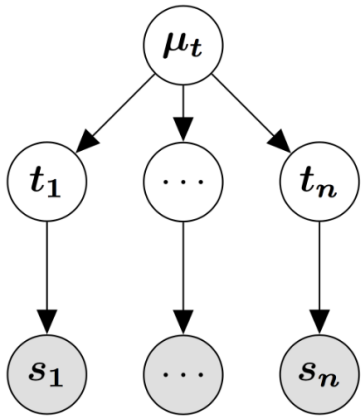
Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if **we want to capture uncertainty about the estimation of the average temperature**?

In [176]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    for i in range(obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- What if **we want to capture uncertainty about the estimation of the average temperature**?
- We can model this parameter with a **random variable**.



Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- What if **we want to capture uncertainty about the estimation of the average temperature**?
- We can model this parameter with a **random variable**.
- We build this probabilistic graphical model.

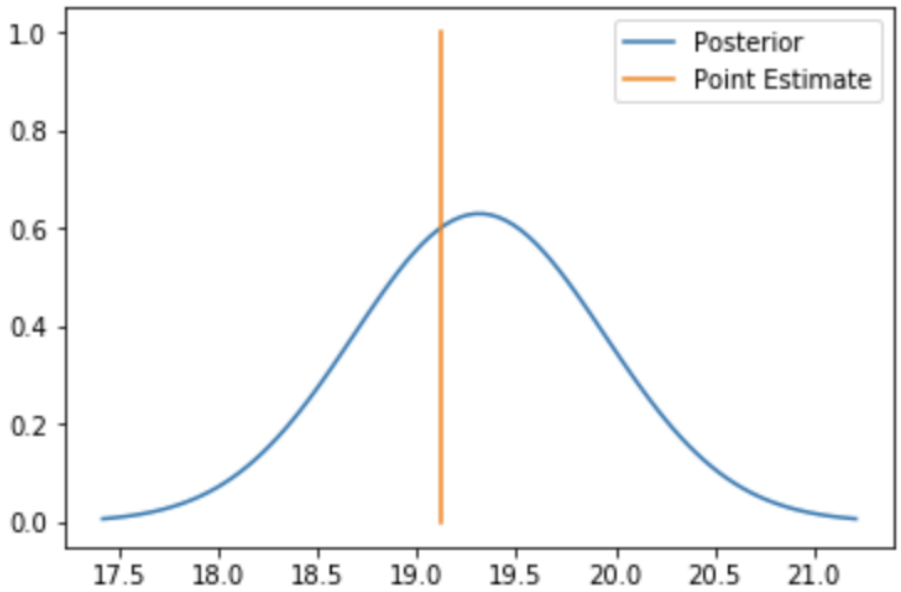
In [162]:
```
import time

#Run inference
start = time.time()
svi(model, guide, obs, num_steps=1000)

#Print results
print("P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
end = time.time()
print(f"{(end - start)} seconds")
```

```
P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =
Normal(loc: 19.199871063232422, scale: 0.6046891212463379)

10.298431873321533 seconds
```

Pyro's inference (http://pyro.ai/examples/intro_part_ii.html) :

- And perform **Bayesian inference**:

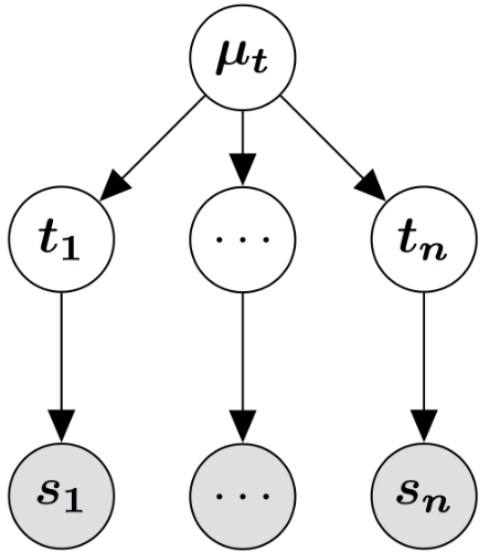
$$p(\mu_t|\mathbf{s}) = \frac{\prod_i \int p(s_i|t_i)p(t_i|\mu_t)dt_i p(\mu_t)}{\int \prod_i p(s_i|\mu_t)p(\mu_t)d\mu}$$



Pyro's inference (`http://pyro.ai/examples/intro_part_ii.html`) :

- And perform **Bayesian inference**:

$$p(\mu_t|\mathbf{s}) = \frac{\prod_i \int p(s_i|t_i)p(t_i|\mu_t)dt_i p(\mu_t)}{\int \prod_i p(s_i|\mu_t)p(\mu_t)d\mu}$$



Pyro's cond. independeces (`http://pyro.ai/examples/svi_part_ii.html`) :

- Sensor variables are **independent** given temperature mean, μ_t.

In [176]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    for i in range(obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

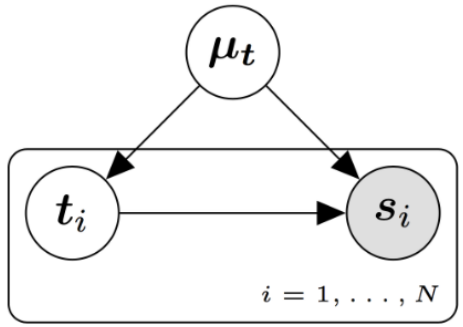
Pyro's cond. independeces (http://pyro.ai/examples/svi_part_ii.html) :

- Sensor variables are **independent** given temperature mean, μ_t.

In [37]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

Pyro's cond. independeces (http://pyro.ai/examples/svi_part_ii.html) :

- Sensor variables are **independent** given temperature mean, μ_t.
- We can use **Pyro's plate construct** to introduce this independence.



Pyro's cond. independeces (`http://pyro.ai/examples/svi_part_ii.html`) :

- Sensor variables are **independent** given temperature mean, μ_t.
- We can use **Pyro's plate construct** to introduce this independence.
- The underlying probabilistic graphical model.

In [165]:
```
#Run inference
start = time.time()
svi(model, guide, obs, num_steps=1000)

#Print results
print("P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
end = time.time()
print(f"{(end - start)} seconds")
```

```
P(mean_temp|Sensor=[18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]) =
Normal(loc: 19.300748825073242, scale: 0.6379732489585876)

2.81210994720459 seconds
```

Pyro's cond. independeces (http://pyro.ai/examples/svi_part_ii.html) :

- We get large gains in efficiency due to **vectorized operations**.
- Execution time without *plate* is over 10s.

Now it is your turn!

In [37]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

Now it is your turn!

In [37]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

Exercise: The role of the prior in learning

- What happens if we change the **mean of the prior**?
- What happens if we change the **scale of the prior**?
- What happens to the posterior if the **number of data samples** decreases and increases?
- Go and play with the prior in the **notebook**

`students_PPLs_Intro.ipynb`

Toy Example

Defining Machine Learning models with PPLs:

- We have an ice-cream shop and we record the ice-cream sales and the average temperature of the day.
- We know temperature affects the sales of ice-creams.
- We want to precisely find out how temperature affects ice-cream sales.

In [49]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]),
       'sales': torch.tensor([46., 47., 49., 44., 50., 54., 51., 52., 49., 53.])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))

    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
```

Ice-cream Shop Model:

- We have observations from temperature and sales.

In [43]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]),
       'sales': torch.tensor([46., 47., 49., 44., 50., 54., 51., 52., 49., 53.])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))

    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
        sales = pyro.sample('sales', dist.Poisson(??????), obs=obs['sales'])
```

Ice-cream Shop Model:

- We have observations from temperature and sales.
- Sales are modeled with a **Poisson distribution**.

In [44]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1]),
       'sales': torch.tensor([46., 47., 49., 44., 50., 54., 51., 52., 49., 53.])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    alpha = pyro.sample('alpha', dist.Normal(0.0, 100.0))
    beta = pyro.sample('beta', dist.Normal(0.0, 100.0))

    with pyro.plate('a', obs['sensor'].shape[0]):
        temp = pyro.sample('temp', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample('sensor', dist.Normal(temp, 1.0), obs=obs['sensor'])
        sales = pyro.sample('sales', dist.Poisson(alpha + beta*temp), obs=obs['sales'])
```

Ice-cream Shop Model:

- We have observations from temperature and sales.
- Sales are modeled with a **Poisson distribution**.
- The rate of the Poisson **linearly depends of the real temperature**.

In [48]:
```
#Run inference
svi(model, guide, obs, num_steps=1000)

#Print results
print("Posterior Temperature Mean")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
print("Posterior Alpha")
print(dist.Normal(pyro.param("alpha_mean").item(), pyro.param("alpha_scale").item()))
print("")
print("Posterior Beta")
print(dist.Normal(pyro.param("beta_mean").item(), pyro.param("beta_scale").item()))
```

```
Posterior Temperature Mean
Normal(loc: 19.311052322387695, scale: 0.6258021593093872)

Posterior Alpha
Normal(loc: 19.773971557617188, scale: 1.8541947603225708)

Posterior Beta
Normal(loc: 1.5178951025009155, scale: 0.1155082955956459)
```

Ice-cream Shop Model:

- We run the **(variational) inference engine** and get the results.

In [48]:
```
#Run inference
svi(model, guide, obs, num_steps=1000)

#Print results
print("Posterior Temperature Mean")
print(dist.Normal(pyro.param("mean").item(), pyro.param("scale").item()))
print("")
print("Posterior Alpha")
print(dist.Normal(pyro.param("alpha_mean").item(), pyro.param("alpha_scale").item()))
print("")
print("Posterior Beta")
print(dist.Normal(pyro.param("beta_mean").item(), pyro.param("beta_scale").item()))
```

```
Posterior Temperature Mean
Normal(loc: 19.311052322387695, scale: 0.6258021593093872)

Posterior Alpha
Normal(loc: 19.773971557617188, scale: 1.8541947603225708)

Posterior Beta
Normal(loc: 1.5178951025009155, scale: 0.1155082955956459)
```

Ice-cream Shop Model:

- We run the **(variational) inference engine** and get the results.
- With PPLs, we only care about modeling, **not about the low-level details** of the machine-learning solver.

Exercise 2: Introduce Humidity in the Icecream shop model

- Assume we also have a bunch of humidity senor measurements.
- Assume the sales are also linearly influenced by the humidity.
- **Extend the above model in order to integrate all of that**.

`students_PPLs_Intro.ipynb`

In [28]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.param('mean_temp', torch.tensor(15.0))
    for i in range(0,obs['sensor'].shape[0]):
        temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
```

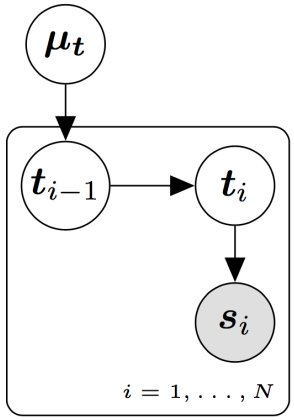
Temporal Models in Pyro:

- The current real temperature must be **similar to the real temperature in the previous time step**.

In [103]:
```
#The observatons
obs = {'sensor': torch.tensor([18., 18.7, 19.2, 17.8, 20.3, 22.4, 20.3, 21.2, 19.5, 20.1])}

def model(obs):
    mean_temp = pyro.sample('mean_temp', dist.Normal(15.0, 2.0))
    for i in range(0,obs['sensor'].shape[0]):
        if i==0:
            temp = pyro.sample(f'temp_{i}', dist.Normal(mean_temp, 2.0))
        else:
            temp = pyro.sample(f'temp_{i}', dist.Normal(prev_temp, 2.0))
        sensor = pyro.sample(f'sensor_{i}', dist.Normal(temp, 1.0), obs=obs['sensor'][i])
        prev_temp = temp
```

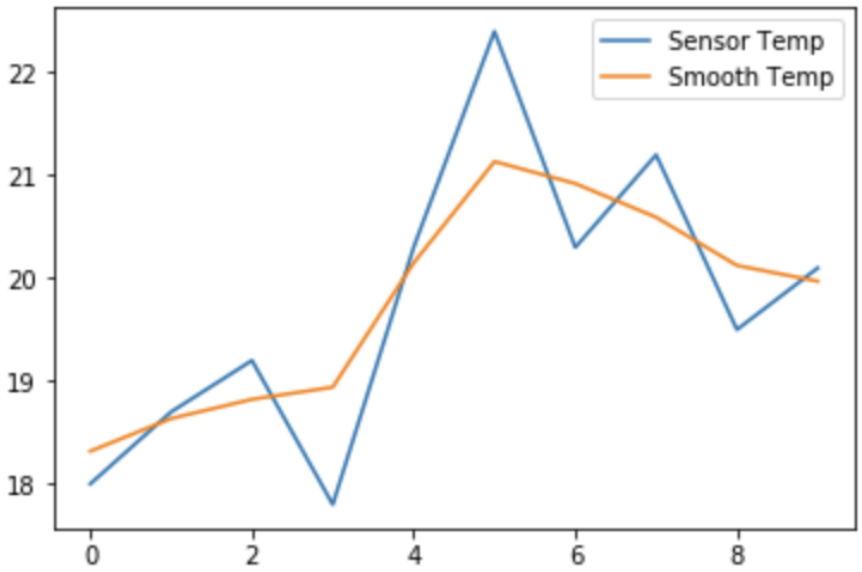
Temporal Models in Pyro:

- The current real temperature must be **similar to the real temperature in the previous time step**.
- This temporal dependency can easily model as follows using a **for-loop**.



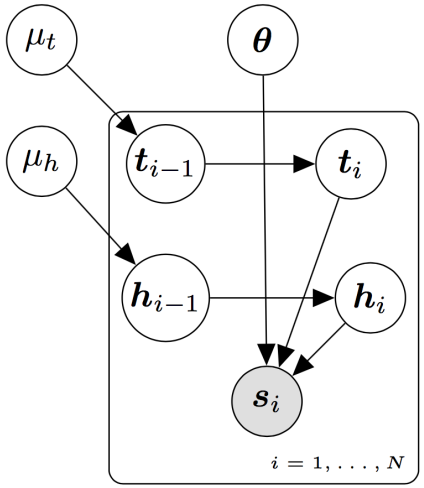
Temporal Models in Pyro:

- The current real temperature must be **similar to the real temperature in the previous time step**.
- This temporal dependency can easily model as follows using a **for-loop**.
- Graphical representation.



Temporal Models in Pyro:

- Pyro's inference engine takes care of the inference.
- By querying the local hidden we can **smooth** the temperature.

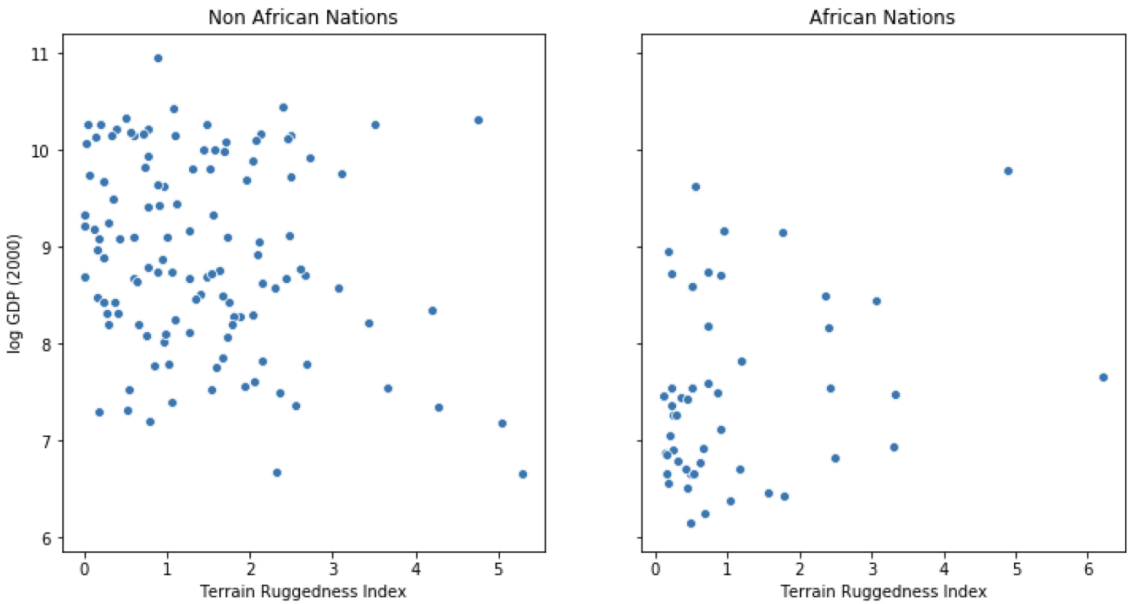


Exercise 3: Temporal Extension of the Iceacream shop model

- Assume temperature depends of the temperature in the previous day.
- Assume humidity depends of the humidity in the previous day.
- Assume sales depends on the current temperature and humidity.

`students_PPLs_Intro.ipynb`

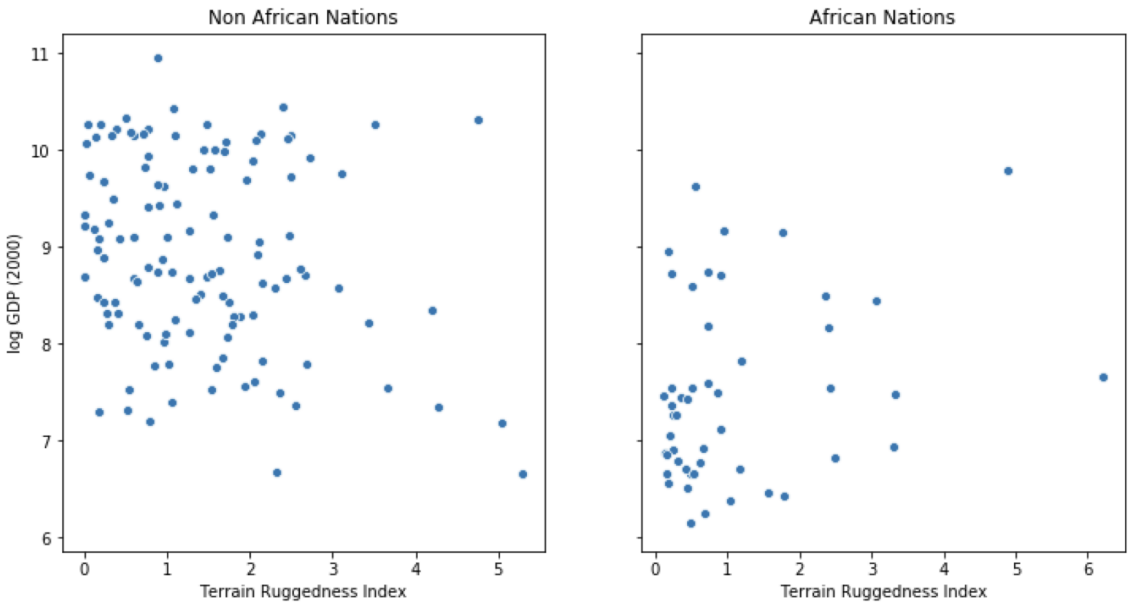
Real Data Example



Relationship between topographic heterogeneity and GDP per capita

- Terrain ruggedness or bad geography is related to poorer economic performance outside of Africa.

Real Data Example

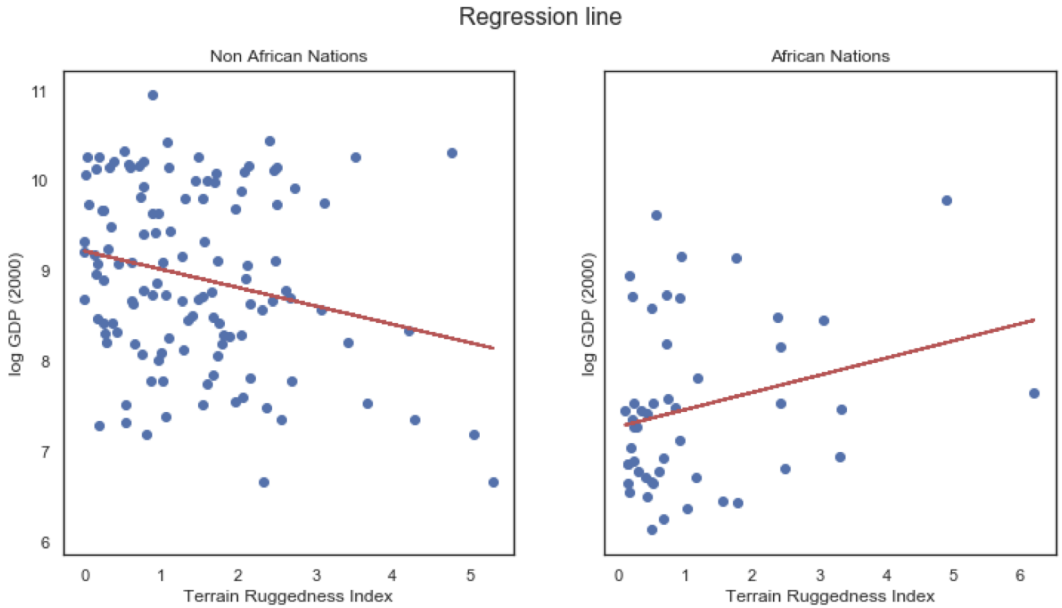


Relationship between topographic heterogeneity and GDP per capita

- Terrain ruggedness or bad geography is related to poorer economic performance outside of Africa.
- Tut rugged terrains have had a reverse effect on income for African nations

`students_Bayesian_regression.ipynb`

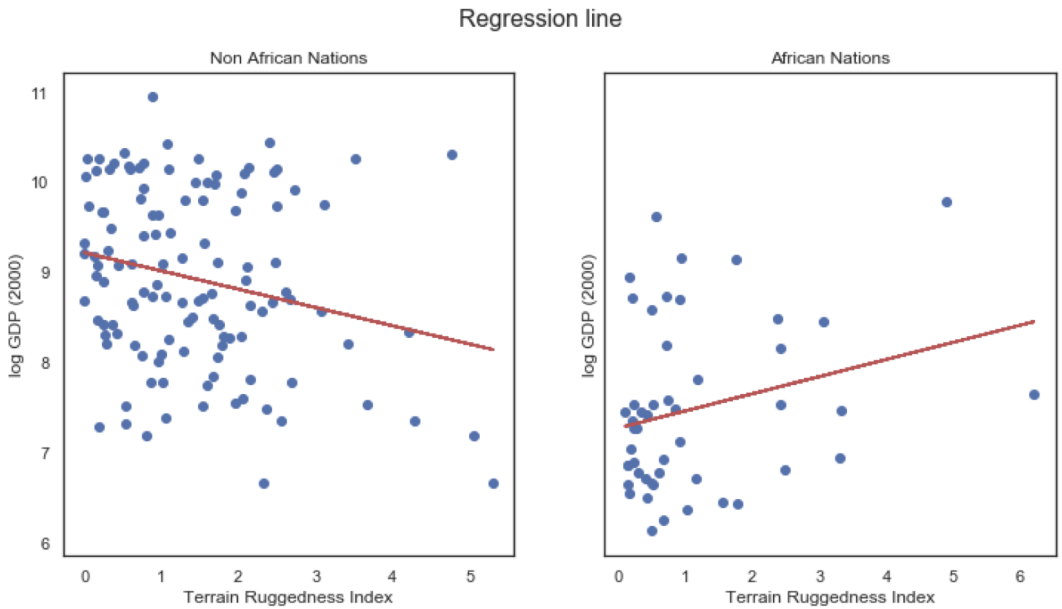
Real Data Example



Linear Regression Model

- Negative slope for Non African Nations.
- Positive slope for African Nations.

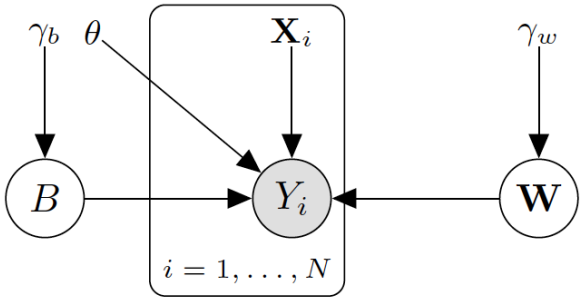
Real Data Example



Bayesian Linear Regression Model

- Modeling data noise (aleatoric uncertainty)
- Modeling uncertainty about the linear coefficients (epistemic uncertainty).

The Bayesian linear regression model



- Num. of data dim: M
- Num. of data inst: N
- $Y_i \mid \{\mathbf{w}, \mathbf{x}_i, \theta\} \sim \mathcal{N}(\mathbf{w}^\mathsf{T}\mathbf{x}_i + b, 1/\theta)$
- $\mathbf{W} \sim \mathcal{N}(\mathbf{0}, \gamma_w^{-1}\mathbf{I}_{M\times M})$
- $B \sim \mathcal{N}(0, \gamma_b^{-1})$

The probability model

$$p(\cdot \mid \mathbf{w}, \theta, \gamma_w, \gamma_b) = \prod_{i=1}^{N} p(y_i \mid \mathbf{x}_i, \mathbf{w}, \theta) p(\mathbf{w} \mid \gamma_w) p(b \mid \gamma_b)$$

Bayesian Linear Regression Model

- Modeling data noise (aleatoric uncertainty)
- Modeling uncertainty about the linear coefficients (epistemic uncertainty).

Real Data Example

In [334]:
```
def model(x_data, y_data):
    # weight and bias priors
    w = pyro.sample("w", Normal(torch.zeros(1, 3), torch.ones(1, 3)).to_event(1))
    b = pyro.sample("b", Normal(0., 1000.))

    precision = pyro.sample("precision", Gamma(1., 1.))
    with pyro.plate("map", len(x_data)):
        # run the nn forward on data
        prediction_mean = (b + torch.mm(x_data,torch.t(w))).squeeze(-1)
        # condition on the observed data
        pyro.sample("obs", Normal(prediction_mean, 1./torch.sqrt(precision)), obs=y_data)
```

Bayesian Linear Regression Model in Pyro

- Modeling data noise by placing a Normal distributions in the observations.

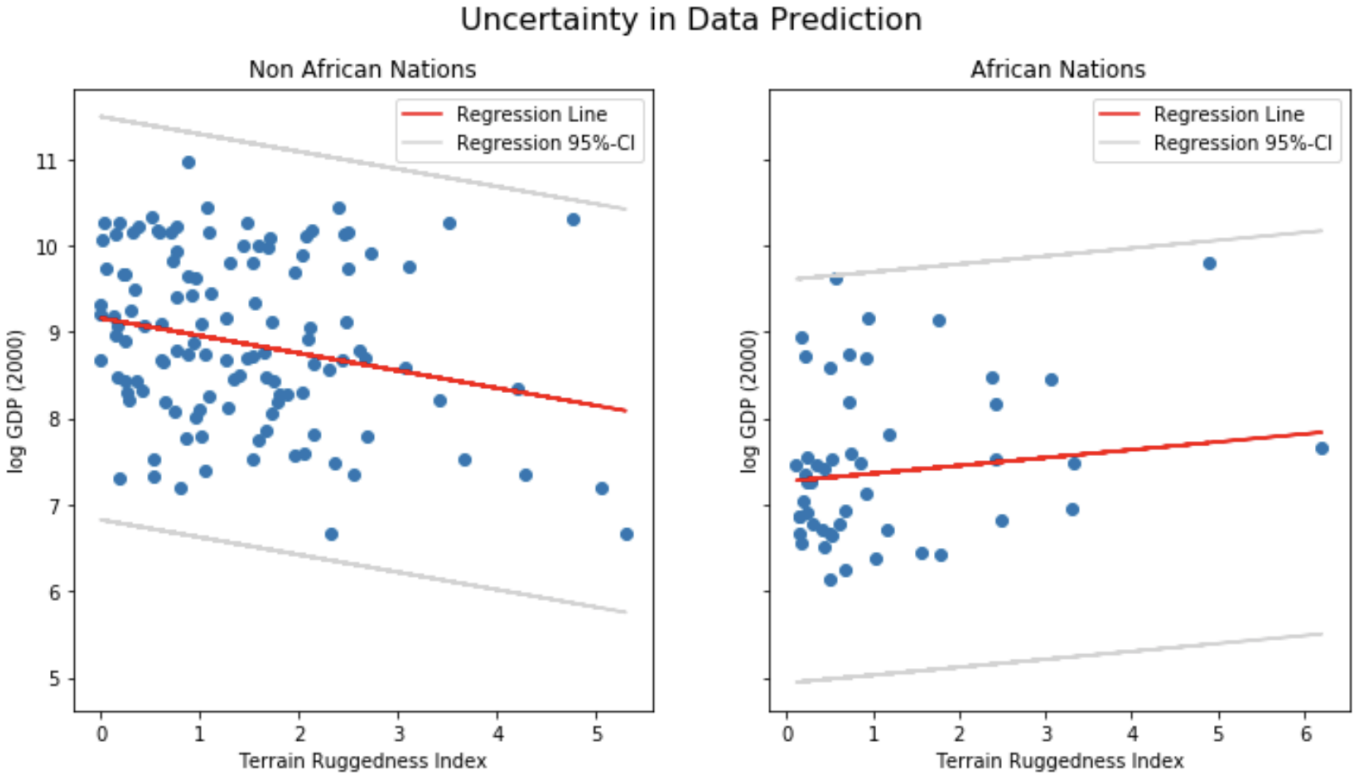
Real Data Example

In [334]:
```
def model(x_data, y_data):
    # weight and bias priors
    w = pyro.sample("w", Normal(torch.zeros(1, 3), torch.ones(1, 3)).to_event(1))
    b = pyro.sample("b", Normal(0., 1000.))

    precision = pyro.sample("precision", Gamma(1., 1.))
    with pyro.plate("map", len(x_data)):
        # run the nn forward on data
        prediction_mean = (b + torch.mm(x_data,torch.t(w))).squeeze(-1)
        # condition on the observed data
        pyro.sample("obs", Normal(prediction_mean, 1./torch.sqrt(precision)), obs=y_data)
```

Bayesian Linear Regression Model in Pyro

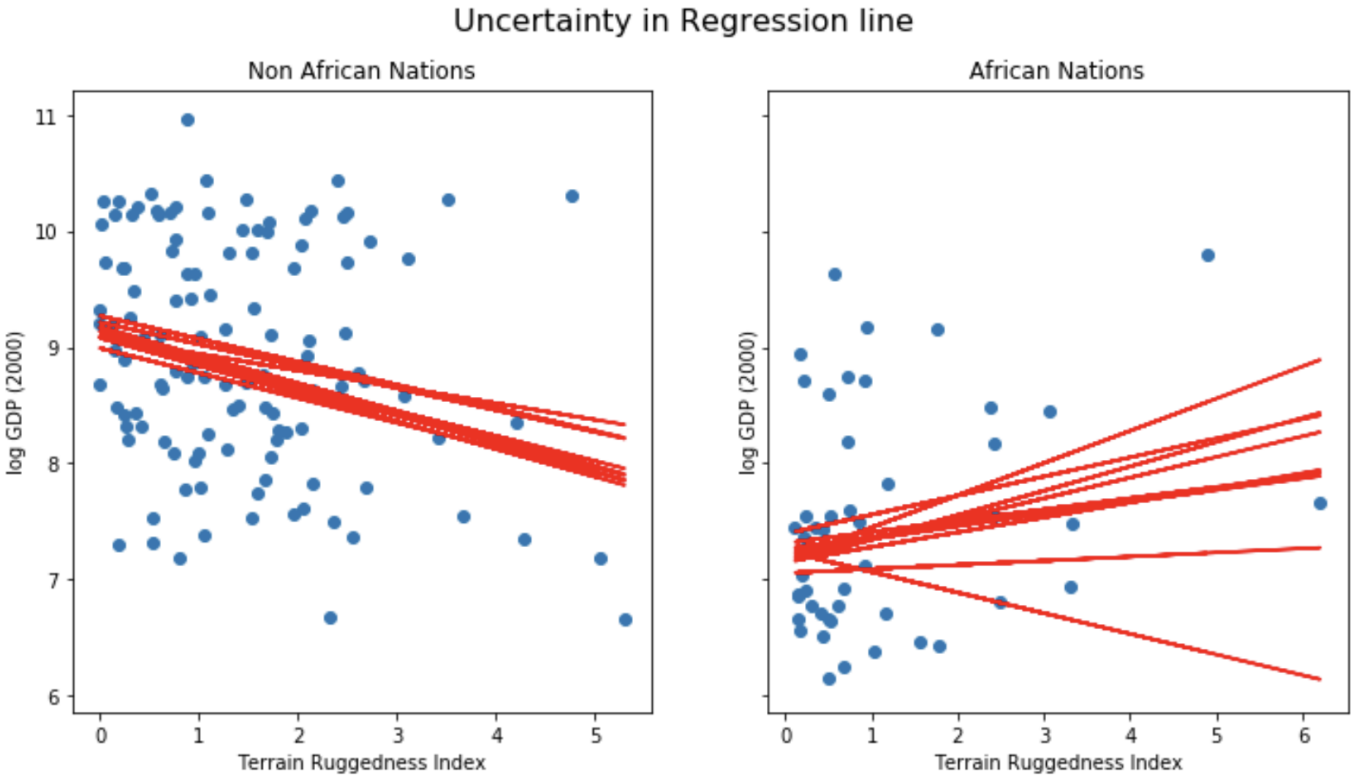
- Modeling data noise by placing a Normal distributions in the observations.
- Modeling uncertainty about the linear coefficients by placing Normal distributions in the parameters.

Real Data Example



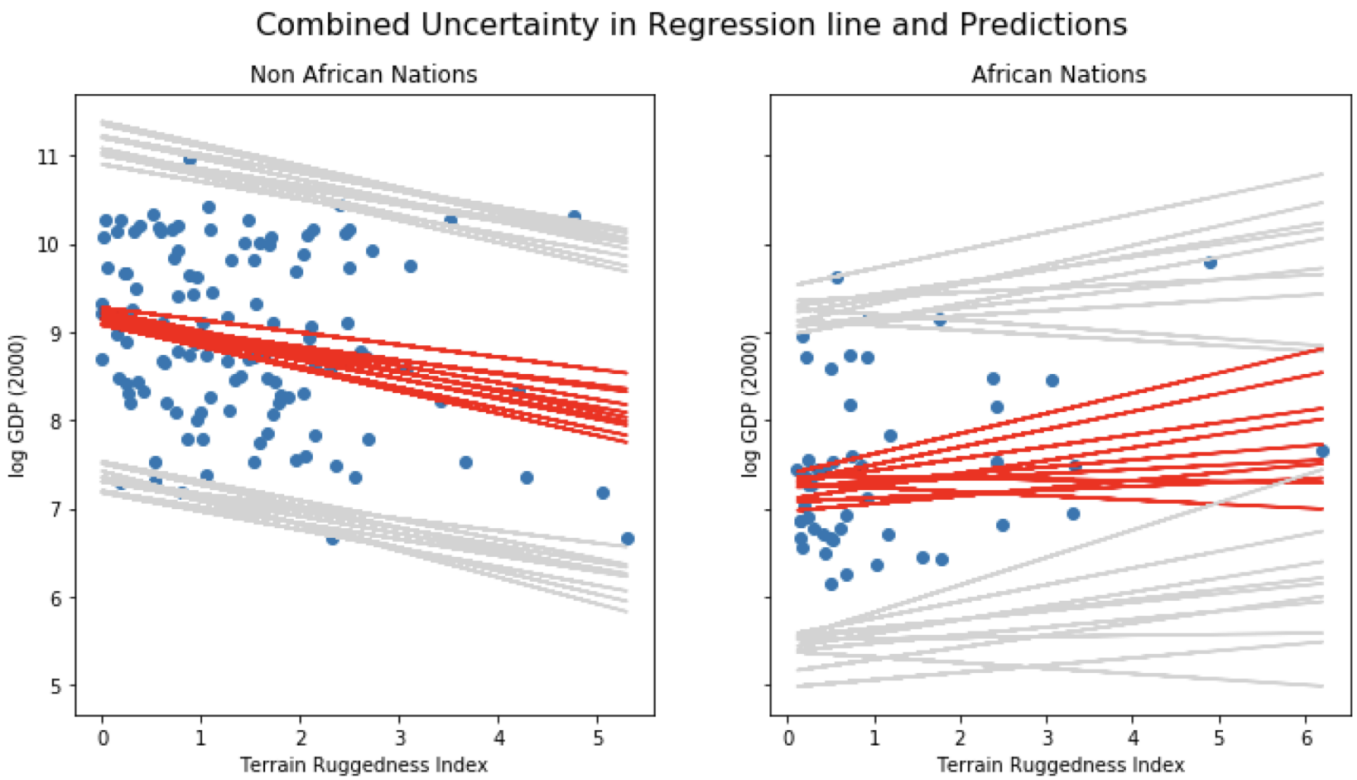
Bayesian Linear Regression Model in Pyro: Uncertainty in Data Predictions

- Now a prediction gives a Normal distribution, not a point estimate.
- We can build confidence intervals.



Bayesian Linear Regression Model in Pyro: Model's uncertainty

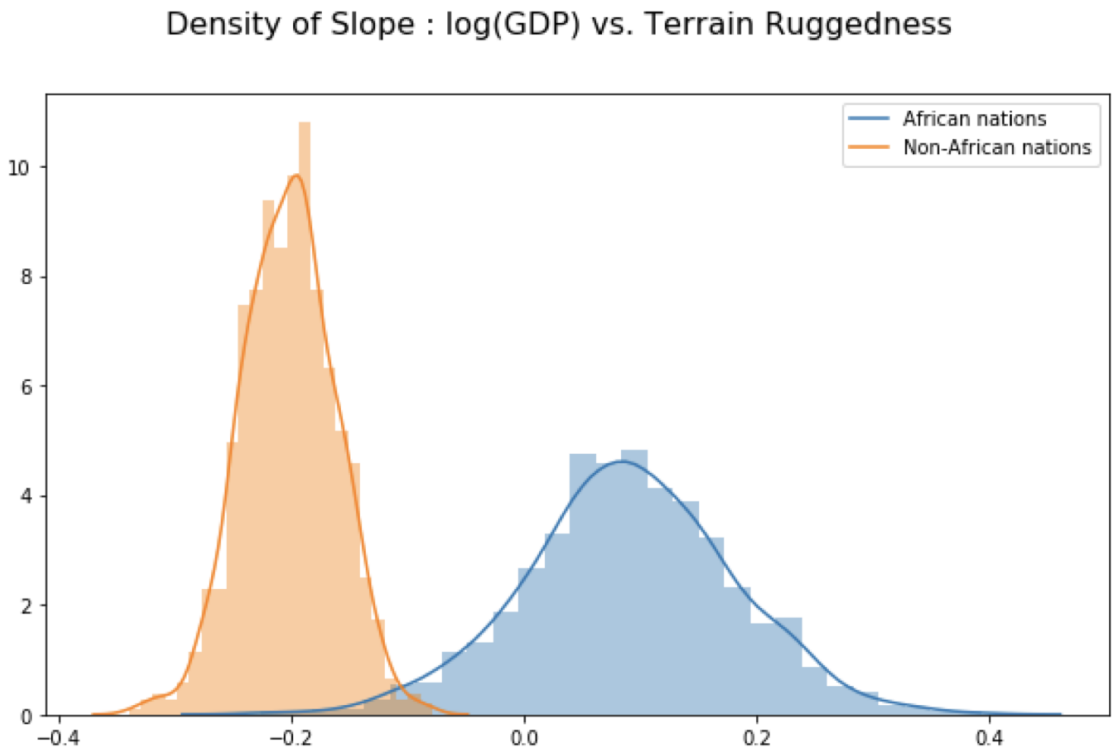
- We have a distribution over (an ensemble of) linear regression models.
- More uncertainty in low density regions.



Bayesian Linear Regression Model in Pyro: Combining uncertainty

- Predictions uncertainty can be combined with model uncertainty.
- More uncertainty in low density regions.

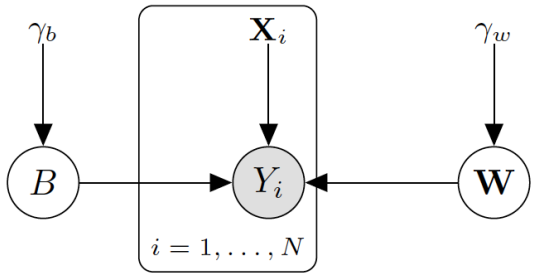
Real Data Example



Bayesian Linear Regression Model in Pyro: Ruggedness Slope

- We have a posterior over the Ruggedness's slope.
- Non-negligible probability that this slope is negative.

The Bayesian logistic regression model



- Num. of data dim: M
- Num. of data inst: N
- $Y_i \,|\, \{\mathbf{w}, \mathbf{x}_i, \theta\} \sim \mathcal{B}(\mathbf{w}^\intercal \mathbf{x}_i + b)$
- $\mathbf{W} \sim \mathcal{N}(\mathbf{0}, \gamma_w^{-1} \mathbf{I}_{M \times M})$
- $B \sim \mathcal{N}(0, \gamma_b^{-1})$

Exercise: Bayesian Logistic Regression

- Predicts whether a country is African or not based on ruggedness and GDP.

`students_Bayesian_Regression.ipynb`