# Homework 2

Kevin Yang - 50244152

October 20, 2021

# 1 Code Snippets

## 1.1 evaluation_based_sampling.py

```python
functions = {}

def evaluate_program(orig_ast):
    """Evaluate a program as desugared by daphne, generate a sample from the prior
    Args:
        ast: json FOPPL program
    Returns: sample from the prior of ast
    """

    variable_bindings = {}

    if type(orig_ast[0]) is list and orig_ast[0][0] == 'defn':
        function_expression = orig_ast[0][2:]
        functions[orig_ast[0][1]] = function_expression
        ast = orig_ast[1]
    else:
        ast = orig_ast[0]
    return evaluate_program_helper(ast, variable_bindings)


def evaluate_program_helper(ast, variable_bindings):

    # print(ast)

    if type(ast) is not list:
        if ast in math_operations:
            return ast
        if ast in data_structure_operations:
            return ast
        if ast in matrix_operations:
            return ast
        if ast in complex_operations:
            return ast
        if ast in my_distributions:
            return ast
        if type(ast) is torch.Tensor:
            return ast
        if type(ast) in [int, float]:
            return torch.tensor(ast)
        if ast in list(variable_bindings.keys()):
            return variable_bindings[ast]
        if ast in list(functions.keys()):
            return functions[ast]
        if ast is None:
            return None
        else:
            raise RuntimeError('Invalid Function', ast)
```

```
    if type(ast) is list:
        if ast[0] == 'let':
            # evaluate the expression that the variable will be bound to
            binding_obj = evaluate_program_helper(ast[1][1], variable_bindings)

            # the variable name is found in let_ast[1][0]
            # update variable_bindings dictionary
            variable_bindings[ast[1][0]] = binding_obj

            # evaluate the return expression
            return evaluate_program_helper(ast[2], variable_bindings)
        if ast[0] in my_distributions:
            curr = [evaluate_program_helper(elem, variable_bindings) for elem in ast]
            return Distribution(dist_type=curr[0], params=curr[1:])
        if ast[0] in math_operations:
            curr = [evaluate_program_helper(elem, variable_bindings) for elem in ast]
            return evaluate_math_operation(curr)
        if ast[0] in data_structure_operations:
            curr = [evaluate_program_helper(elem, variable_bindings) for elem in ast]
            return evaluate_data_structure_operation(curr)
        if ast[0] in complex_operations:
            curr = [evaluate_program_helper(elem, variable_bindings) for elem in ast]
            return evaluate_complex_operation(curr)
        if ast[0] in matrix_operations:
            curr = [evaluate_program_helper(elem, variable_bindings) for elem in ast]
            return evaluate_matrix_operation(curr)
        if ast[0] in list(functions.keys()):
            inputs = [evaluate_program_helper(elem, variable_bindings) for elem in ast[1:]]
            body = functions[ast[0]]

            for idx, param in enumerate(body[0]):
                variable_bindings[param] = inputs[idx]

            return evaluate_program_helper(body[1], variable_bindings)
```

## 1.2 graph_based_sampling.py

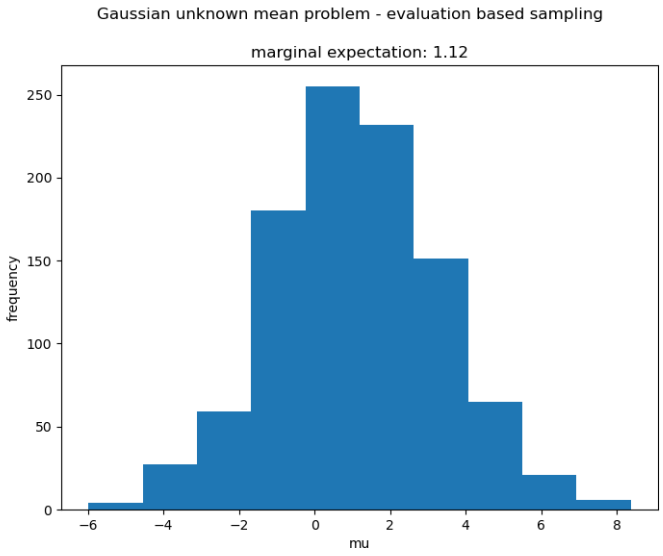
```
def sample_from_joint(graph):
    "This function does ancestral sampling starting from the prior."

    g = Graph(graph[1]['V'])
    for key, values in graph[1]['A'].items():
        for child in values:
            g.addEdge(key, child)
    sampling_order = g.topologicalSort()

    for vertex in sampling_order:
        # substitute parent nodes with their sampled values
        raw_expression = graph[1]['P'][vertex]
        variable_bindings = graph[1]['Y']
        expression = substitute_sampled_vertices(raw_expression, variable_bindings)

        graph[1]['Y'][vertex] = deterministic_eval(expression)

    # substitute return nodes with sampled values
    raw_expression = graph[2]
    variable_bindings = graph[1]['Y']
    expression = substitute_sampled_vertices(raw_expression, variable_bindings)
    return deterministic_eval(expression)
```

```
import primitives
from graph import Graph
from tests import is_tol, run_prob_test,load_truth
from utils import load_ast, substitute_sampled_vertices

# Put all function mappings from the deterministic language environment to your
# Python evaluation context here:
env = {'normal': dist.Normal,
       'beta': dist.Beta,
       'exponential': dist.Exponential,
       'uniform': dist.Uniform,
       'discrete': dist.Categorical,
       '+': primitives.add,
       '*': primitives.multiply,
       '-': primitives.minus,
       '/': primitives.divide,
       'sqrt': torch.sqrt,
       'vector': primitives.vector,
       'sample*': primitives.sample,
       'observe*': primitives.observe,
       'hash-map': primitives.hashmap,
       'get': primitives.get,
       'put': primitives.put,
       'first': primitives.first,
       'second': primitives.second,
       'rest': primitives.rest,
       'last': primitives.last,
       'append': primitives.append,
       '<': primitives.less_than,
       '>': primitives.greater_than,
       'mat-transpose': primitives.mat_transpose,
       'mat-tanh': primitives.mat_tanh,
       'mat-mul': primitives.mat_mul,
       'mat-add': primitives.mat_add,
       'mat-repmat': primitives.mat_repmat,
       'if': primitives.conditional
       }


def deterministic_eval(exp):
    "Evaluation function for the deterministic target language of the graph based representation."
    if type(exp) is list:
        op = exp[0]
        args = exp[1:]
        return env[op](*map(deterministic_eval, args))
    elif type(exp) is int or type(exp) is float:
        # We use torch for all numerical objects in our evaluator
        return torch.tensor(float(exp))
    elif type(exp) is torch.Tensor:
        return exp
    else:
        raise("Expression type unknown.", exp)
```

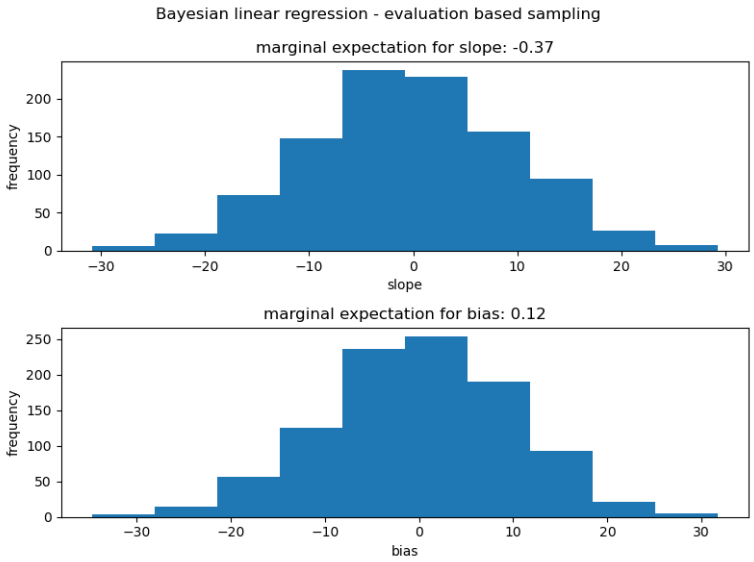
# 2 Plots

## 2.1 Evaluation based

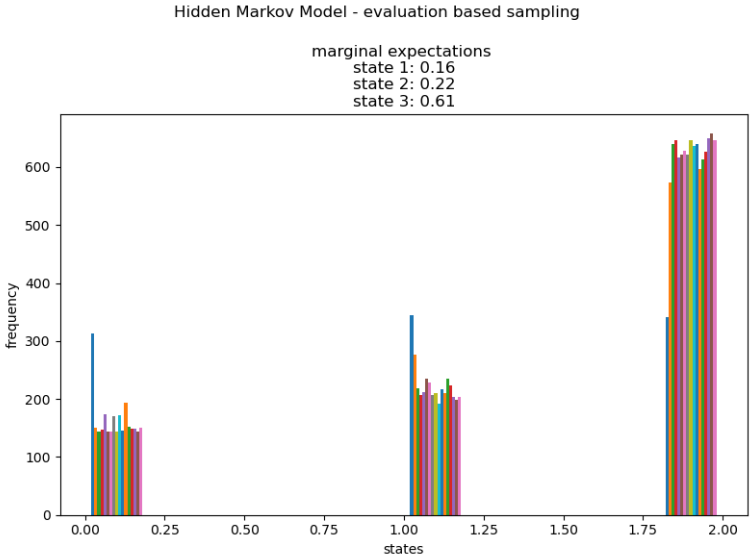
### 2.1.1 Gaussian unknown mean



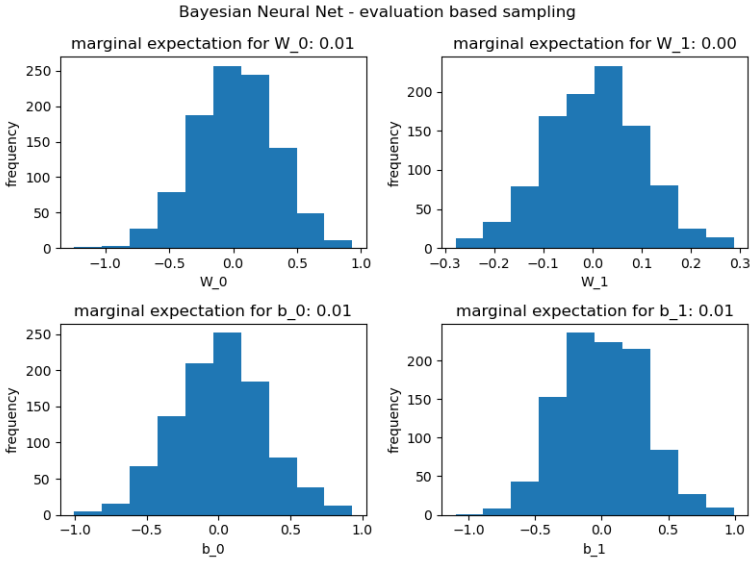

### 2.1.2 Bayesian linear regression problem



### 2.1.3 Hidden Markov Model

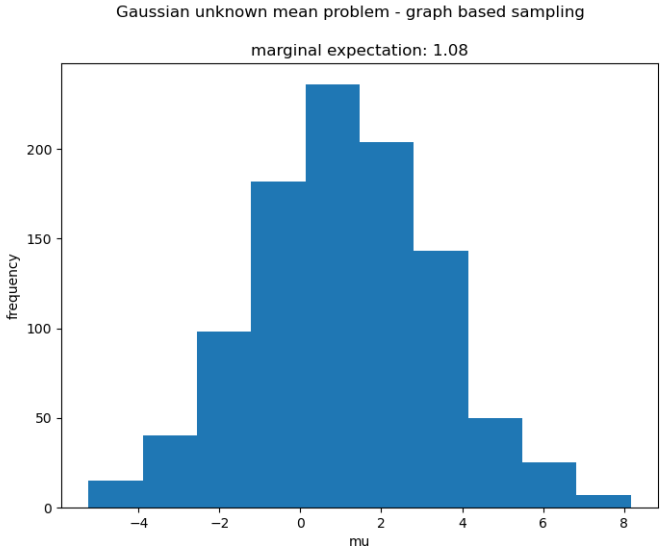



### 2.1.4 Bayesian Neural Network Learning

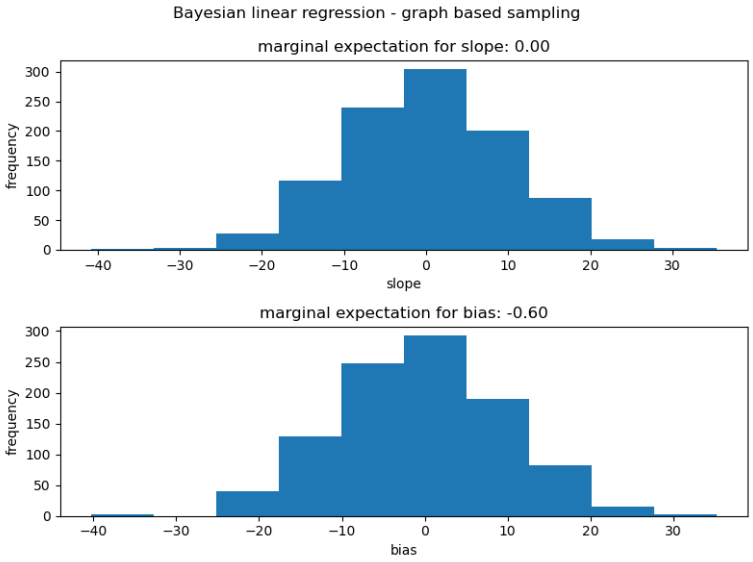


## 2.2 Graph based

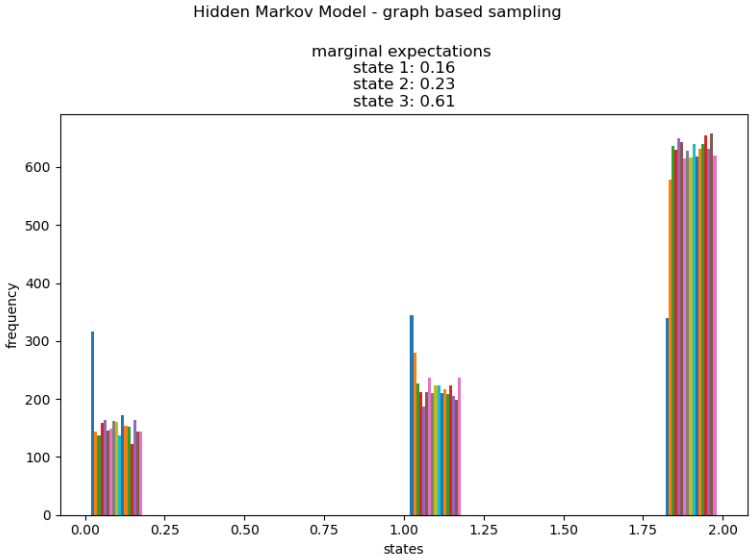
### 2.2.1 Gaussian unknown mean




### 2.2.2 Bayesian linear regression problem



### 2.2.3 Hidden Markov Model




### 2.2.4 Bayesian Neural Network Learning