CPSC 532W Assignment 2

Ali Seyfi - 97446637

1 Evaluation Based Sampling

Write a FOPPL evaluator that evaluates the FOPPL as described in lecture and in the book (Alg. 6)

1.1 Code

Provide code snippets that document critical aspects of your implementation sufficient to allow us to quickly determine whether or not you individually completed the assignment. Based on what was mentioned in the assignment description, I ignored the observe statements for this assignment.

1.1.1 primitives

```
baseprimitives = {
    '+': lambda x: x[0] + x[1],
    '-': lambda x: x[0] - x[1],
    '*': lambda x: x[0] * x[1],
    '/': lambda x: x[0] / x[1],
    '>': lambda x: x[0] > x[1],
    '>=': lambda x: x[0] >= x[1],
    '<': lambda x: x[0] < x[1],
    '<=': lambda x: x[0] <= x[1],
    '==': lambda x: x[0] == x[1],
    'sqrt': lambda x: torch.sqrt(x[0]),
    'exp': lambda x: torch.exp(x[0]),
    'log': lambda x: torch.log(x[0]),
    'vector': vector,
    'list': list,
    'get': get,
    'put': put,
    'hash-map': hash_map,
    'first': lambda x: x[0][0],
    'last': lambda x: x[0][-1],
    'nth': lambda x: x[0][int(x[1].item())],
    'second': lambda x: x[0][1],
    'rest': lambda x: x[0][1:],
    'append': append,
    'cons': lambda x: append([x[1],x[0]]),
    'conj': append,
    'mat-add': lambda x: x[0] + x[1],
    'mat-mul': lambda x: torch.matmul(x[0], x[1]),
    'mat-transpose': lambda x: x[0].T,
    'mat-tanh': lambda x: x[0].tanh(),
    'mat-repmat': lambda x: x[0].repeat((int(x[1].item()), int(x[2].item())))
}
```

Listing 1: primitives.py - Base primitives

```
def vector(x):
    try:
        vector = torch.stack(x)
    except:
        vector = x
    return vector

def list(x):
    try:
        list = torch.stack(x)
    except:
        list = x
    return list

def get(x):
    if type(x[0]) == dict:
        value = x[0][x[1].item()]
    else:
        value = x[0][x[1].long()]
    return value

def put(x):
    if type(x[0]) == dict:
        x[0][x[1].item()] = x[2]
    else:
        x[0][x[1].long()] = x[2]
    return x[0]

def hash_map(x):
    keys = x[::2]
    value = x[1::2]
    new_keys = []
    for key in keys:
        try:
            new_keys.append(key.item())
        except:
            new_keys.append(key)
    result = dict(zip(new_keys, value))
    return result

def append(x):
    first = x[0]
    second = x[1]

    if first == 'vector':
        first = torch.tensor([])
    elif first.dim() == 0:
        first = first.unsqueeze(0)
    if second == 'vector':
        second = torch.tensor([])
    if second.dim() == 0:
        second = second.unsqueeze(0)
    return torch.cat((first, second))
```

Listing 2: primitives.py - Functions

```
import torch
from torch import distributions

class Dist:
    def __init__(self, name, distribution, num_par, *par):
        self.name = name
        self.distribution = distribution
        self.num_par = num_par
        self.pars = []
        for i in range(num_par):
            self.pars.append(par[i])

    def sample(self):
        return self.distribution.sample()

class normal(Dist):
    def __init__(self, pars):
        mean = pars[0]
        var = pars[1]
        super().__init__('normal', distributions.Normal(mean, var), 2, mean, var)

class beta(Dist):
    def __init__(self, pars):
        alpha = pars[0]
        betta = pars[1]
        super().__init__('beta', distributions.Beta(alpha, betta), 2, alpha, betta)

class exponential(Dist):
    def __init__(self, par):
        lamda = par[0]
        super().__init__('exponential', distributions.Exponential(lamda), 1, lamda)

class uniform(Dist):
    def __init__(self, pars):
        a = pars[0]
        b = pars[1]
        super().__init__('uniform', distributions.Uniform(a, b), 2, a, b)

class discrete(Dist):
    def __init__(self, pars):
        prob = pars[0]
        super().__init__('discrete', distributions.Categorical(prob), 0)

class bernoulli(Dist):
    def __init__(self, pars):
        p = pars[0]
        super().__init__('bernoulli', distributions.Bernoulli(p), 1, p)
```

Listing 3: primitives.py - Distributions

```
distlist = {
    'normal' : normal,
    'beta' : beta,
    'exponential' : exponential,
    'uniform' : uniform,
    'discrete' : discrete,
    'bernoulli': bernoulli
}
```

Listing 4: primitives.py - distlist

1.1.2 evaluate program

```python
from primitives import baseprimitives, distlist


def evaluate_program(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
    """
    funcs = {}
    final_ast = ast
    if isinstance(ast, list):
        if isinstance(ast[0], list):
            if ast[0][0] == 'defn':
                for statement in ast:
                    if statement[0]== 'defn':
                        funcs[statement[1]] = (statement[1], statement[2], statement[3])
                        final_ast = final_ast[1:]
                    else:
                        result = eval(statement, {}, {}, funcs)
            if final_ast[0][0] != 'defn':
                result = eval(final_ast[0], {}, {}, funcs)
        else:
            result = eval(ast, {}, {}, funcs)
    else:
        result = eval(ast, {}, {}, funcs)
    return result[0]
```

Listing 5: evaluation_based_sampling.py - evaluate_program

1.1.3 eval

```python
def eval(x, sigma, l, funcs):
    "Evaluate an expression in an environment."
    if isinstance(x, list) and len(x) == 1:
        x = x[0]
    if not isinstance(x, list):
        if isinstance(x, int) or isinstance(x, float):
            result = torch.tensor(x, dtype=float)
        elif x in baseprimitives or torch.is_tensor(x) or x in funcs or x in distlist:
            result = x
        else:
            result = l[x]
    elif x[0] == 'if':
        cond_result, sigma = eval(x[1], sigma, l, funcs)
        if cond_result:
            result = x[2]
        else:
            result = x[3]
    elif x[0] == 'let':
        result_temp, sigma = eval(x[1][1], sigma, l, funcs)
        for e in x[2:-1]:
            _, sigma = eval(e, sigma, {**l, **{x[1][0]: result_temp}}, funcs)
        result, sigma = eval(x[-1], sigma, {**l, **{x[1][0]: result_temp}}, funcs)
    elif x[0] == 'sample':
        dist, sigma = eval(x[1], sigma, l, funcs)
        result = dist.sample()
    elif x[0] == 'observe':
        result = None
    else:
        statements = []
        for expression in x:
```

```
            statement, sigma = eval(expression, sigma, l, funcs)
            statements.append(statement)

        first_statemnt, other_statements = statements[0], statements[1:]
        if first_statemnt in baseprimitives:
            result = baseprimitives[first_statemnt](other_statements)
        elif first_statemnt in distlist:
            result = distlist[first_statemnt](other_statements)

        elif first_statemnt in funcs:
            _, variables, process = funcs[first_statemnt]
            assignment = {key:value for key, value in zip(variables, other_statements)}
            result, sigma = eval(process, sigma, {**l, **assignment}, funcs)
        else:
            result = torch.tensor(statements)
    return result, sigma
```

Listing 6: evaluation_based_sampling.py - evaluate_program

1.2 Results

1.2.1 Tests Results

Results of Tests
Here are the results of the test files:
Deterministic
tensor(7., dtype=torch.float64)
Test passed
tensor(1.4142, dtype=torch.float64)
Test passed
tensor(24., dtype=torch.float64)
Test passed
tensor(0.2500, dtype=torch.float64)
Test passed
tensor(0.1802, dtype=torch.float64)
Test passed
tensor([2., 3., 4., 5.], dtype=torch.float64)
Test passed
tensor(4., dtype=torch.float64)
Test passed
tensor([2., 3., 3., 5.], dtype=torch.float64)
Test passed
tensor(2., dtype=torch.float64)
Test passed
tensor(5., dtype=torch.float64)
Test passed
tensor([2.0000, 3.0000, 4.0000, 5.0000, 3.1400], dtype=torch.float64)
Test passed
tensor(5.3000, dtype=torch.float64)
Test passed
6.0: tensor(2., dtype=torch.float64), 1.0: tensor(3.2000, dtype=torch.float64)
Test passed
All deterministic tests passed

Probabilistic ('normal', 5, 1.4142136)
p value 0.0012557687750447137
Test 1 passed
('beta', 2.0, 5.0)
p value 0.7075526288252392
Test 2 passed
('exponential', 0.0, 5.0)
p value 0.09598741374360209
Test 3 passed
('normal', 5.3, 3.2)
p value 0.016072128150235995
Test 4 passed
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.3676183768222948
Test 5 passed
('normal', 0, 1.44)
p value 0.1907566727266461
Test 6 passed
All probabilistic tests passed

1.2.2 Marginal Expectation

Draw 1000 samples for each of the task programs and report marginal expectations for all return value dimensions.
Definitely the result changes each time we draw the samples, and in the following I have provided the result of the only one run of the code.

1.2.2.1 Task 1

Marginal Expectation of mu is: **0.9896**

1.2.2.2 Task 2

Marginal Expectation of slope is: **-0.1806**
Marginal Expectation of bias is: **0.0168**

1.2.2.3 Task 3

Marginal Expectation of each step is these numbers by order:
1.8824,1.6470,1.7058,1.6471,1.7059,0.9412,1.1176,1.2941,
1.2353,1.7647,1.5294,1.11765,1.11765,1.3529,1.3529,1.5882

1.2.2.4 Task 4

Marginal Expectation of w_1 elements are:

$$\begin{bmatrix} -0.1264 \\ -0.3671 \\ 0.0419 \\ 0.0767 \\ 0.1846 \\ -0.1250 \\ 0.2340 \\ 0.2118 \\ -0.1531 \\ 0.6157 \\ -0.1788 \\ 0.1203 \\ 0.8325 \\ -0.1807 \\ 0.0838 \\ -0.2090 \\ -0.0974 \end{bmatrix}$$

Marginal Expectation of b_1 elements are:

$$\begin{bmatrix} 0.2112 \\ 0.5417 \\ 0.048 \\ 0.2376 \\ -0.0978 \\ -0.0586 \\ 0.3023 \\ -0.4403 \\ -0.5109 \\ -0.0517 \\ -0.1594 \\ -0.1185 \\ 0.7249 \\ 0.0214 \\ 0.0119 \\ -0.3293 \\ 0.0979 \end{bmatrix}$$

Marginal Expectation of w_2 elements are:

$$\begin{bmatrix} -0.0354 & 0.0195 & -0.0046 & -0.0041 & -0.0542 & 0.028 & -0.0106 & 0.0013 & 0.0318 & 0.0063 \\ 0.0385 & -0.0414 & -0.0328 & -0.0524 & 0.0058 & -0.0672 & -0.0242 & -0.0182 & -0.0357 & 0.0026 \\ -0.0005 & 0.0281 & -0.0467 & -0.0054 & -0.0168 & 0.0125 & -0.0054 & 0.0226 & 0.0022 & -0.023 \\ -0.0065 & 0.0267 & 0.03 & 0.0366 & 0.0354 & 0.046 & 0.0115 & -0.0267 & 0.0344 & -0.004 \\ 0.0507 & -0.0629 & 0.0439 & 0.0585 & 0.0141 & -0.0153 & -0.0057 & 0.0291 & -0.0244 & 0.0308 \\ -0.0106 & -0.0203 & 0.0332 & 0.0199 & 0.0279 & -0.0208 & 0.0066 & 0.0153 & 0.0396 & -0.0323 \\ -0.0101 & -0.0367 & 0.0427 & 0.0245 & -0.0546 & 0.0185 & 0.0037 & -0.0173 & 0.0014 & 0.0599 \\ -0.0261 & 0.0029 & -0.0164 & -0.0138 & 0.0364 & 0.0285 & 0.0475 & -0.0012 & -0.0091 & -0.052 \\ -0.0146 & -0.0539 & -0.0297 & -0.0349 & 0.0144 & -0.0121 & -0.0167 & -0.0114 & 0.0126 & 0.0041 \\ 0.0027 & -0.0295 & -0.0509 & 0.0139 & 0.0441 & 0.0168 & -0.0433 & -0.0706 & 0.0254 & 0.0179 \end{bmatrix}$$

Marginal Expectation of b_2 elements are:

$$\begin{bmatrix} -0.3279 \\ 0.2553 \\ 0.4049 \\ 0.1395 \\ 0.1667 \\ -0.3218 \\ 0.3551 \\ 0.1866 \\ 0.1888 \\ 0.0709 \\ -0.0666 \\ -0.0663 \\ 0.0215 \\ 0.3254 \\ 0.198 \\ -0.1191 \\ -0.1661 \end{bmatrix}$$

1.2.3 Histograms

1.2.3.1 Task 1

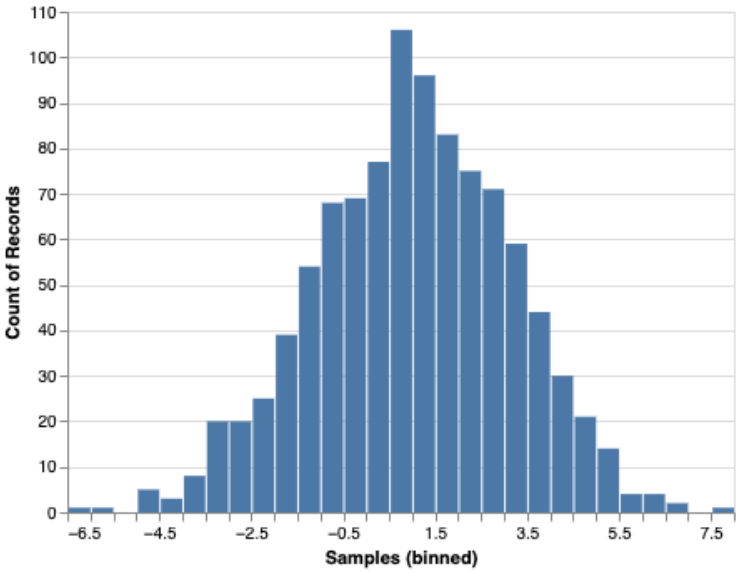

Figure 1: Histogram of mu

1.2.3.2 Task 2

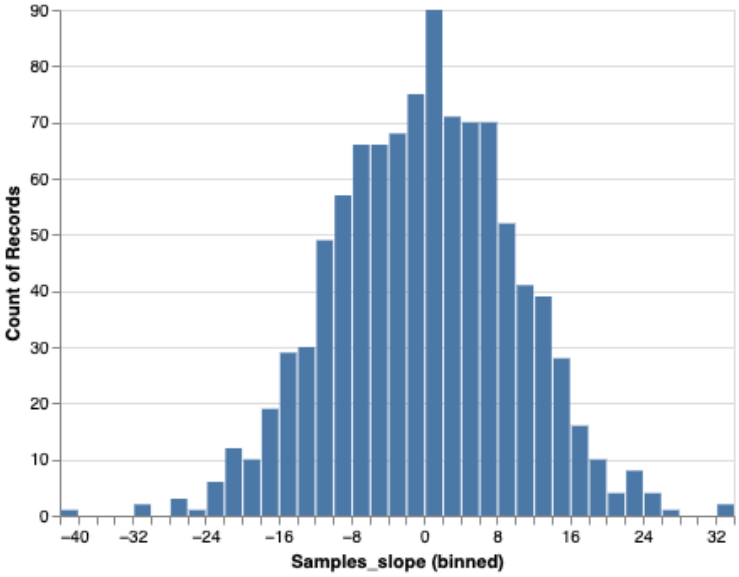

Figure 2: Histogram of slope

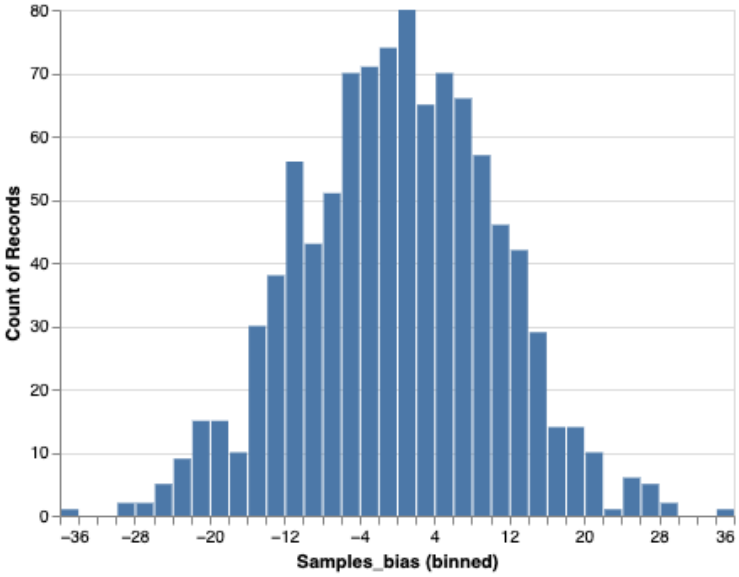

Figure 3: Histogram of bias

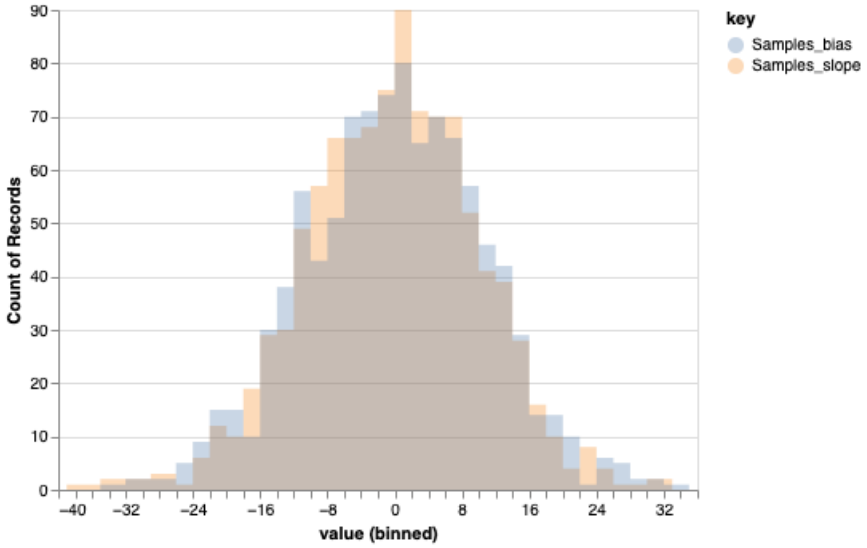

Figure 4: Histogram of slope and bias together

1.2.3.3 Task 3

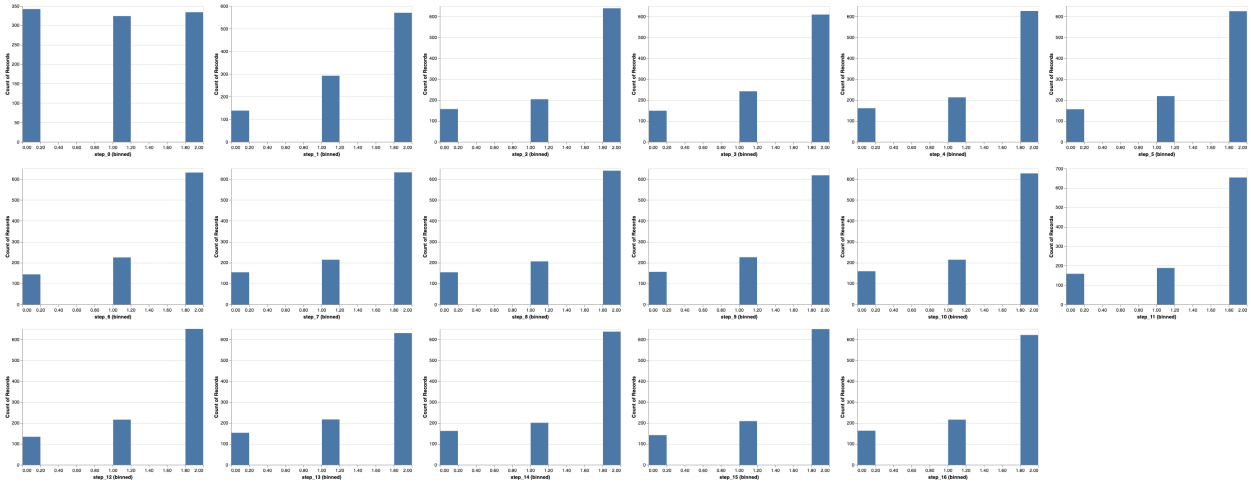

Figure 5: Histogram of states in each step

1.2.3.4 Task 4

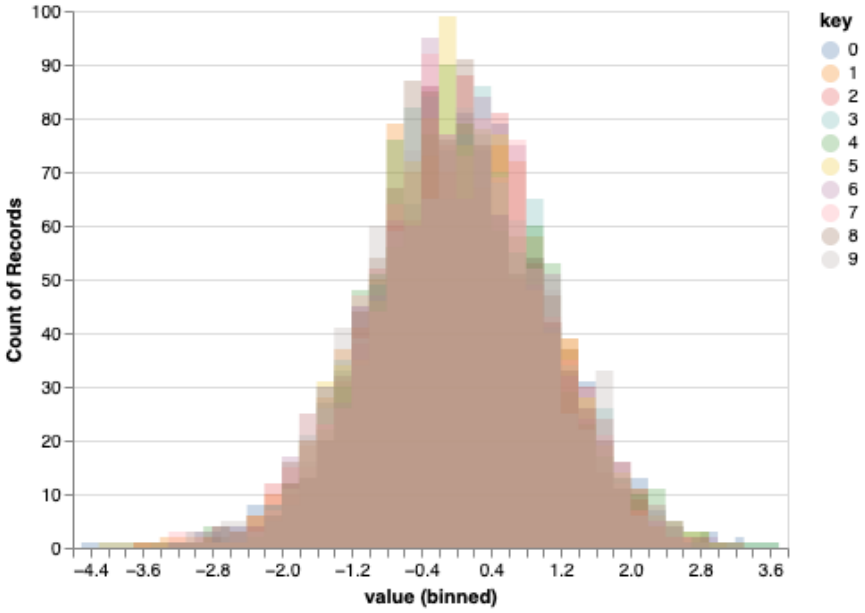

Figure 6: Histogram of W_1 elements together

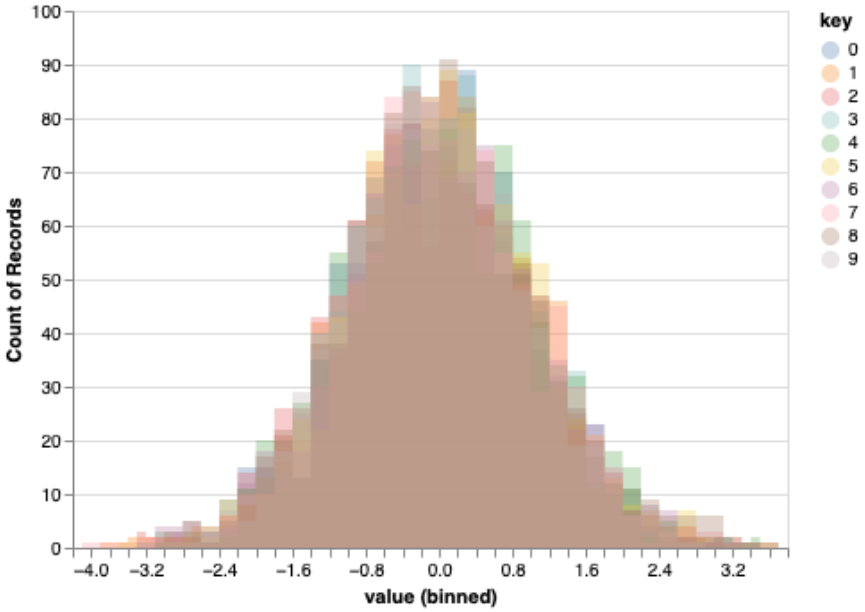

Figure 7: Histogram of b_1 elements together

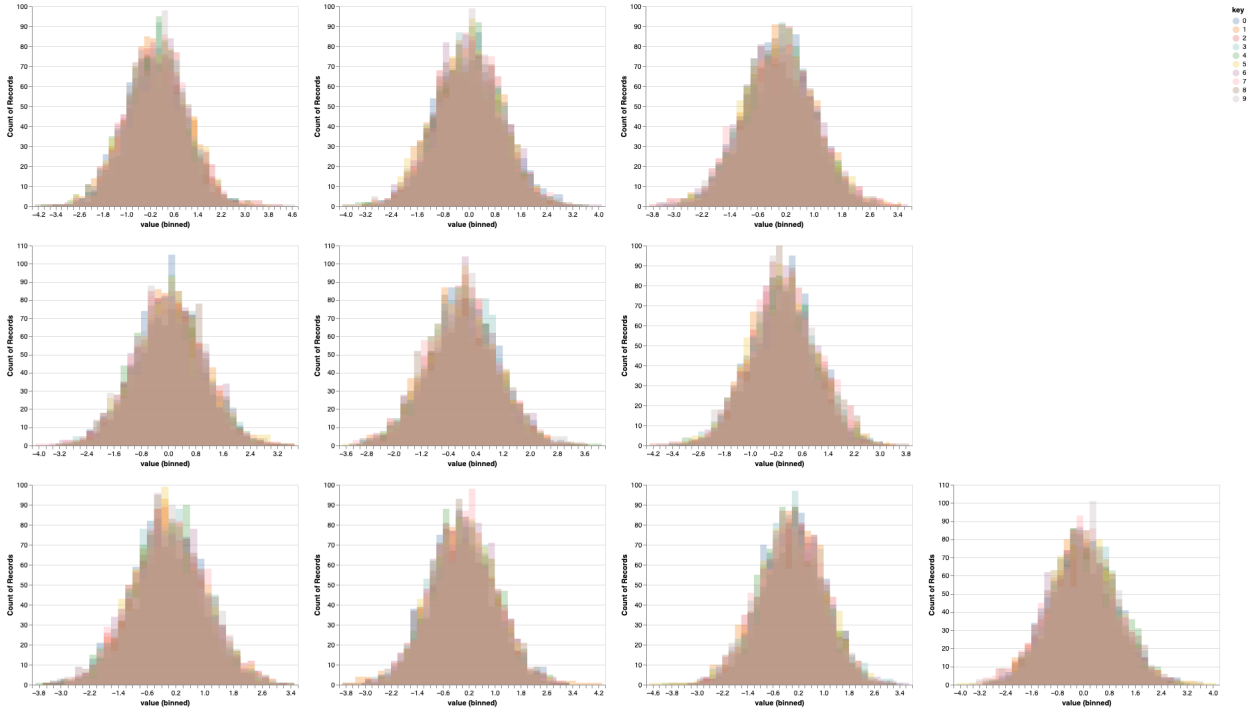

Figure 8: Histogram of W_2 elements together

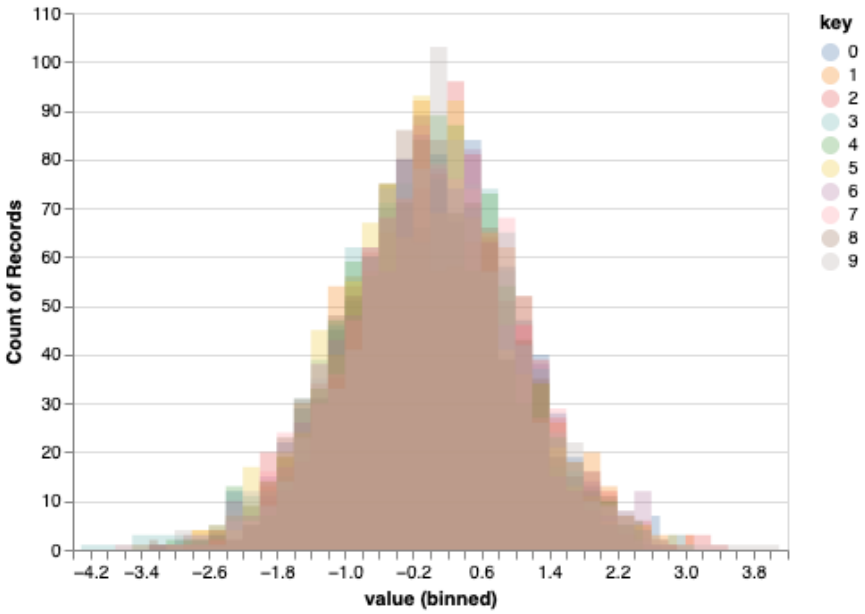

Figure 9: Histogram of b_2 elements together

2 Graph Based Sampling

Write a FOPPL evaluator that evaluates the FOPPL program via ancestral sampling in the graphical model.

2.1 Code

Provide code snippets that document critical aspects of your implementation sufficient to allow us to quickly determine whether or not you individually completed the assignment.

2.1.1 primitives

This part is exactly same as 1.1.1

2.1.2 topological sort

```
def topological_sort(graph):
    nodes = graph[1]['V']
    edges = graph[1]['A']
    is_visited = dict.fromkeys(nodes, False)
    node_stack = []
    node_order_reverse = []
    for node in nodes:
        if not is_visited[node]:
            node_stack.append((node, False))
        while len(node_stack) > 0:
            node, flag = node_stack.pop()
            if flag:
                node_order_reverse.append(node)
                continue
            is_visited[node] = True
            node_stack.append((node, True))
            if node not in edges:
                continue
            children = edges[node]
            for child in children:
                if not is_visited[child]:
                    node_stack.append((child, False))
    return node_order_reverse[::-1]
```

Listing 7: graph_based_sampling.py - topological_sort

2.1.3 environment

```
env = {**baseprimitives, **distlist}
```

Listing 8: graph_based_sampling.py - environment

2.1.4 deterministic eval

```
def deterministic_eval(exp):
    "Evaluation function for the deterministic target language of the graph based
    representation."
    if isinstance(exp, list):
        if exp[0] == 'hash-map':
            exp = ['hash-map'] + [value for expression in exp[1:] for value in expression]
    return evaluate_program(exp)
```

Listing 9: graph_based_sampling.py - deterministic_eval

2.1.5 value substitution

```
def value_subs(expressions, variables):
    if isinstance(expressions, list):
        result = []
        for expression in expressions:
            result.append(value_subs(expression, variables))
    else:
        if expressions in variables:
            result = variables[expressions]
        else:
            result = expressions
    return result
```

Listing 10: graph_based_sampling.py - value_subs

2.1.6 sample from joint

```
def sample_from_joint(graph):
    "This function does ancestral sampling starting from the prior."
    node_order = topological_sort(graph)
    results = {}
    for node in node_order:
        first_statement, *other_statements = graph[1]['P'].get(node)
        if first_statement == 'sample*':
            dist = deterministic_eval(value_subs(other_statements, results))
            result = dist.sample()
        if first_statement == 'observe*':
            result = graph[1]['Y'].get(node)
        results[node] = result
    return deterministic_eval(value_subs(graph[2], results))
```

Listing 11: graph_based_sampling.py - sample_from_joint

2.2 Results

2.2.1 Tests Results

Results of Tests
Here are the results of the test files:
Deterministic
tensor(7., dtype=torch.float64)
Test passed
tensor(1.4142, dtype=torch.float64)
Test passed
tensor(24., dtype=torch.float64)
Test passed
tensor(0.2500, dtype=torch.float64)
Test passed
tensor(0.1802, dtype=torch.float64)
Test passed
tensor([2., 3., 4., 5.], dtype=torch.float64)
Test passed
tensor(4., dtype=torch.float64)
Test passed
tensor([2., 3., 3., 5.], dtype=torch.float64)
Test passed

tensor(2., dtype=torch.float64)
Test passed
tensor(5., dtype=torch.float64)
Test passed
tensor([2.0000, 3.0000, 4.0000, 5.0000, 3.1400], dtype=torch.float64)
Test passed
tensor(5.3000, dtype=torch.float64)
Test passed
All deterministic tests passed
Probabilistic ('normal', 5, 1.4142136)
p value 0.5663323302205264
('beta', 2.0, 5.0)
p value 0.3206785223599047
('exponential', 0.0, 5.0)
p value 0.32984046631598296
('normal', 5.3, 3.2)
p value 0.5410338117003856
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.03844514904336682
('normal', 0, 1.44)
p value 0.6072736562958746
All probabilistic tests passed

2.2.2 Marginal Expectation

Draw 1000 samples for each of the task programs and report marginal expectations for all return value dimensions.
Definitely the result changes each time we draw the samples, and in the following I have provided the result of the only one run of the code.

2.2.2.1 Task 1

Marginal Expectation of mu is: **1.0183**

2.2.2.2 Task 2

Marginal Expectation of slope is: **-0.1693**
Marginal Expectation of bias is: **0.1126**

2.2.2.3 Task 3

Marginal Expectation of each step is these numbers by order:
1.2353 1.3529 1.5294 1.5882 0.8824 1.3529 1.9412 1.3529
1.4118 1.5294 1.6471 1.2353 1.2941 1.1176 1.2941 1.5294 1.2353

2.2.2.4 Task 4

Marginal Expectation of w_1 elements are:

$$\begin{bmatrix} -0.0116 \\ -0.1254 \\ 0.3629 \\ -0.2237 \\ 0.2997 \\ -0.0427 \\ -0.0172 \\ -0.2322 \\ 0.0637 \\ -0.2389 \\ 0.172 \\ -0.1198 \\ 0.2513 \\ -0.3007 \\ -0.1563 \\ -0.0328 \\ -0.1635 \end{bmatrix}$$

Marginal Expectation of b_1 elements are:

$$\begin{bmatrix} -0.213 \\ 0.1301 \\ -0.3042 \\ 0.1645 \\ -0.2951 \\ -0.0824 \\ 0.5833 \\ -0.1653 \\ 0.1151 \\ -0.1317 \\ 0.1903 \\ -0.1795 \\ -0.4252 \\ 0.7888 \\ -0.0638 \\ 0.3291 \\ -0.2595 \end{bmatrix}$$

Marginal Expectation of w_2 elements are:

$$\begin{bmatrix} -0.0142 & -0.0052 & -0.0494 & 0.0185 & -0.0629 & -0.0547 & 0.009 & -0.0188 & 0.0162 & -0.004 \\ -0.0064 & 0.0193 & -0.0407 & 0.0317 & 0.0051 & -0.0126 & -0.0207 & -0.0025 & 0.0251 & -0.0516 \\ -0.057 & -0.0017 & 0.0374 & 0.0038 & -0.0176 & -0.0082 & 0.0151 & -0.0269 & -0.0085 & -0.0243 \\ 0.0122 & -0.0005 & -0.0448 & -0.0564 & 0.0005 & 0.0345 & 0.0718 & 0.0308 & 0.005 & -0.0089 \\ -0.0819 & -0.0145 & -0.0286 & -0.0325 & -0.0565 & 0.044 & 0.0178 & -0.0043 & 0.0271 & 0.0003 \\ 0.0219 & 0.0345 & 0.0251 & 0.0254 & -0.0139 & 0.047 & 0.0051 & 0.0255 & 0.0469 & 0.0204 \\ 0.0492 & -0.0257 & -0.003 & 0.0182 & -0.035 & -0.0264 & 0.0609 & 0.0106 & 0.0003 & -0.0122 \\ -0.019 & 0.0187 & -0.0308 & -0.0158 & 0.0189 & -0.0126 & 0.0241 & -0.0395 & -0.0088 & 0.0434 \\ -0.0166 & -0.0015 & -0.0674 & 0.0265 & -0.0046 & 0.0763 & -0.0201 & 0.0072 & 0.0231 & -0.004 \\ -0.0346 & -0.026 & 0.0248 & -0.0126 & -0.0444 & 0.0066 & 0.0662 & -0.0013 & -0.0439 & 0.0088 \end{bmatrix}$$

Marginal Expectation of b_2 elements are:

$$\begin{bmatrix} 0.0754 \\ -0.044 \\ 0.1044 \\ -0.3383 \\ 0.115 \\ -0.0623 \\ -0.6844 \\ 0.255 \\ 0.0021 \\ 0.3057 \\ 1.155 \\ 0.1589 \\ 0.0446 \\ -0.1136 \\ -1.0082 \\ -0.3659 \\ -0.0531 \end{bmatrix}$$

2.2.3 Histograms

2.2.3.1 Task 1

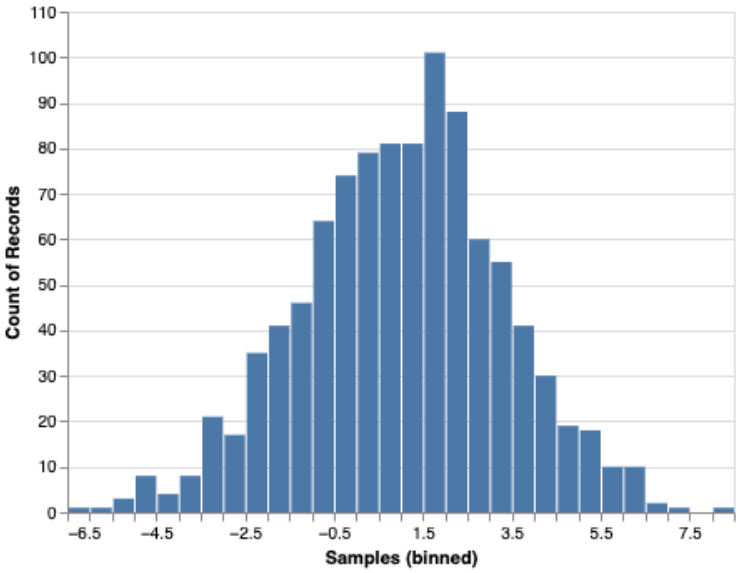

Figure 10: Histogram of mu

2.2.3.2 Task 2

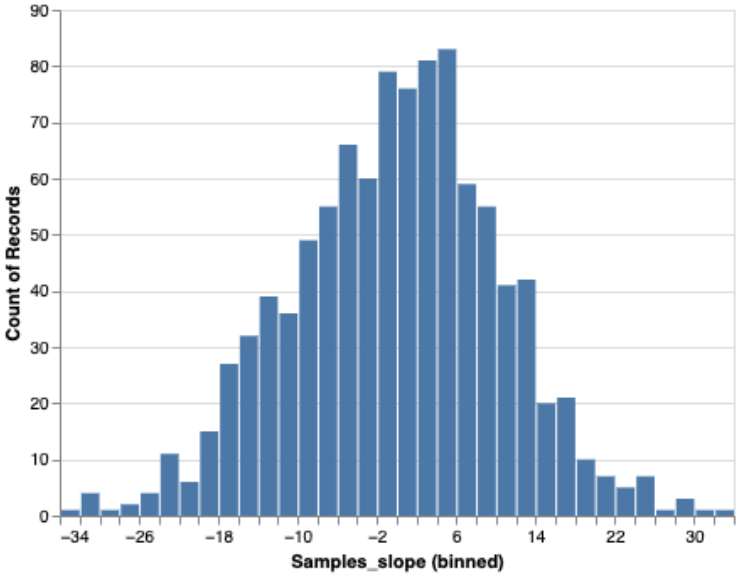

Figure 11: Histogram of slope

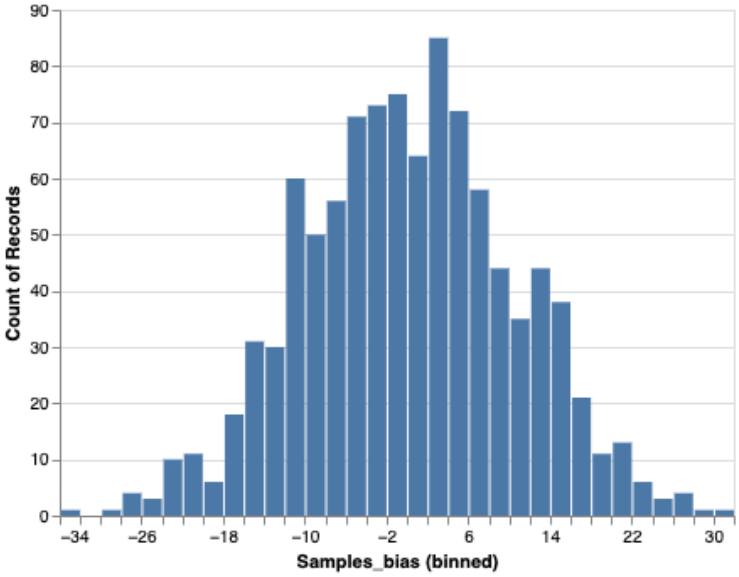

Figure 12: Histogram of bias

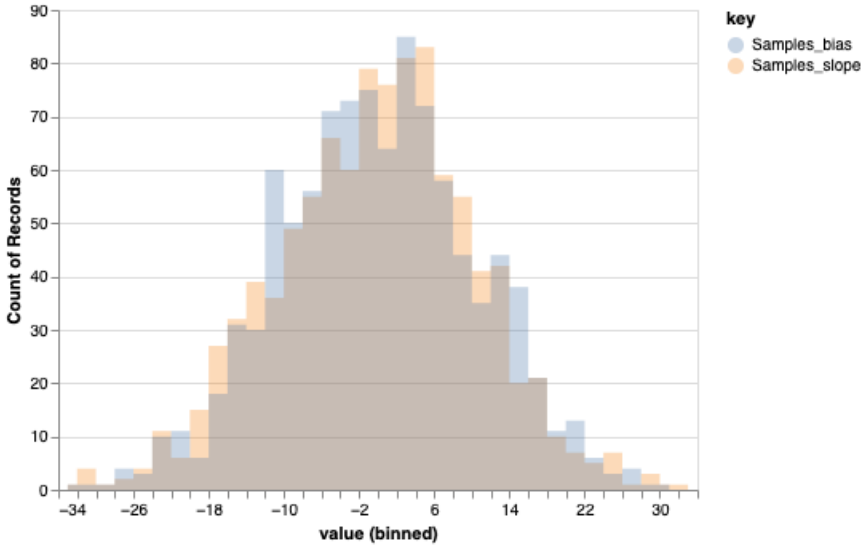

Figure 13: Histogram of slope and bias together

2.2.3.3 Task 3

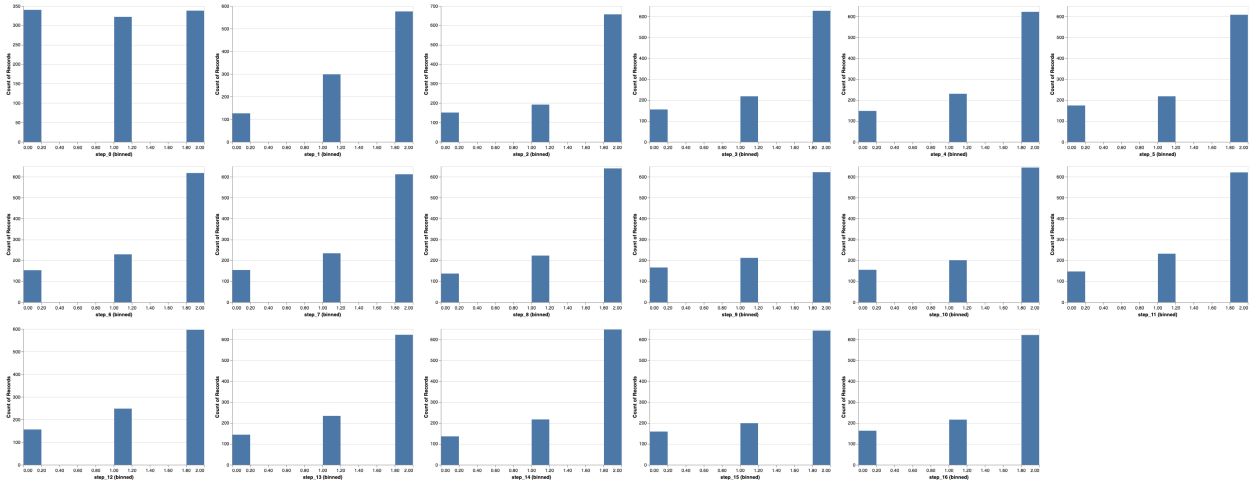

Figure 14: Histogram of states in each step

2.2.3.4 Task 4

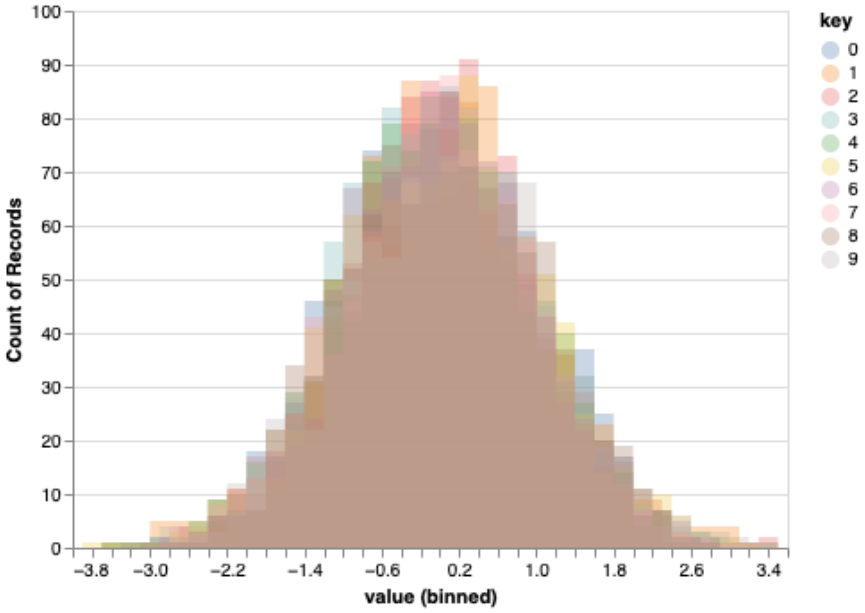

Figure 15: Histogram of W_1 elements together

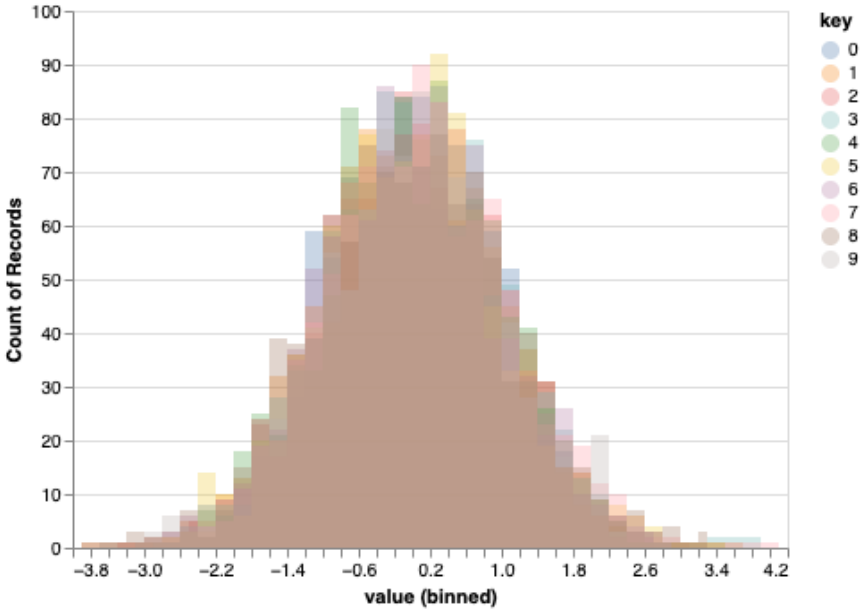

Figure 16: Histogram of b_1 elements together

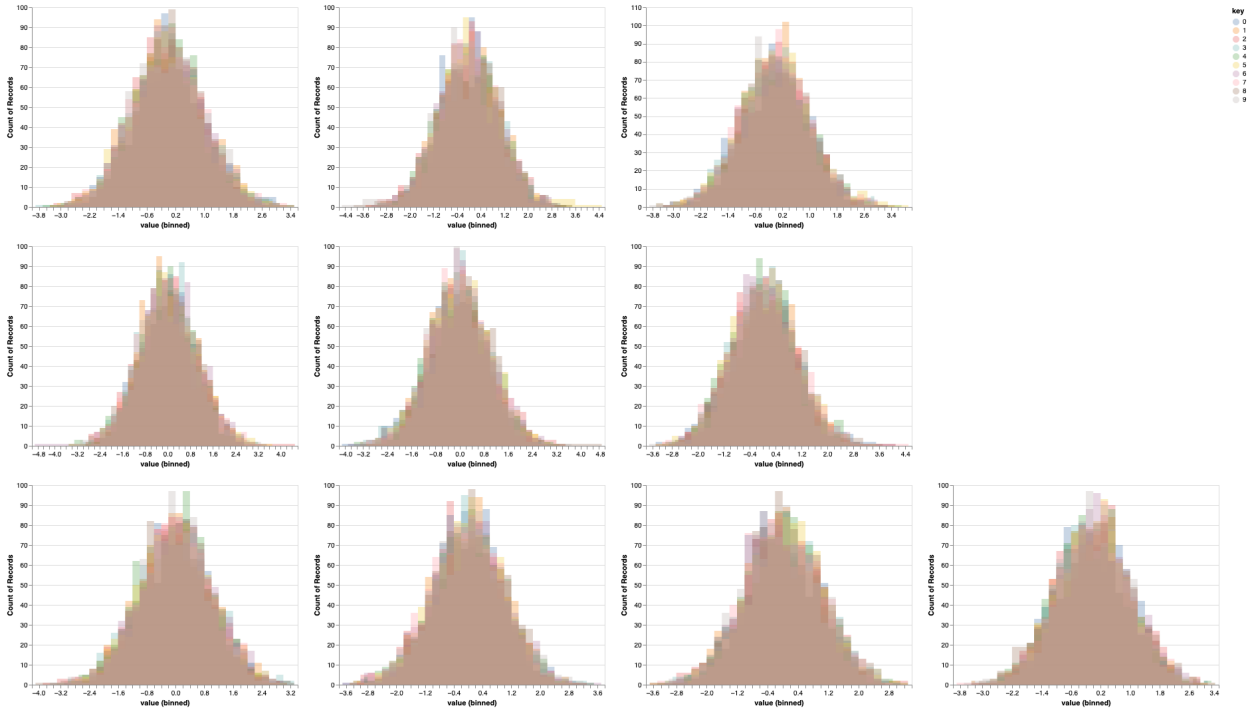

Figure 17: Histogram of W_2 elements together

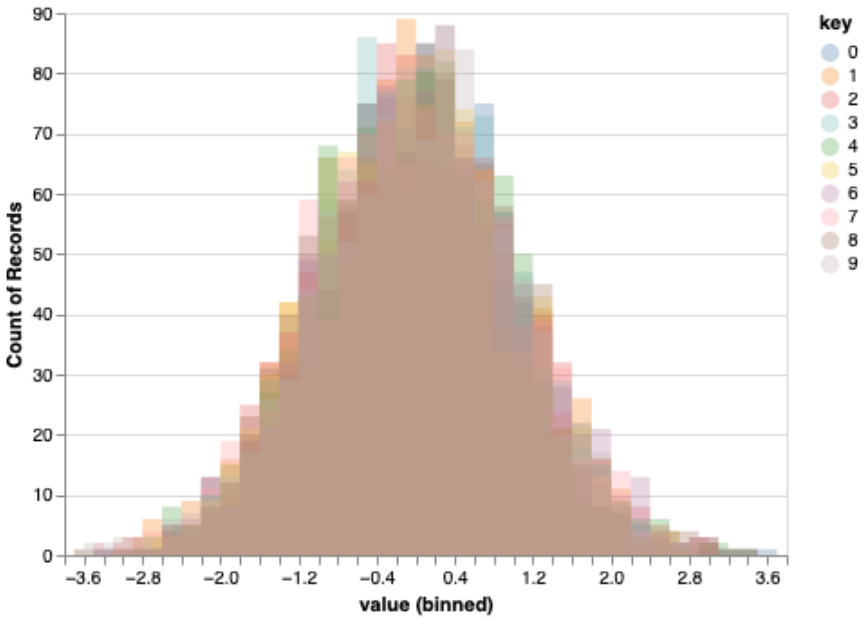

Figure 18: Histogram of b_2 elements together