

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.

1.1.1 primitives

```
1 baseprimitives = {
2     '+' : lambda x: x[0] + x[1],
3     '-' : lambda x: x[0] - x[1],
4     '*' : lambda x: x[0] * x[1],
5     '/' : lambda x: x[0] / x[1],
6     '>' : lambda x: x[0] > x[1],
7     '>=' : lambda x: x[0] >= x[1],
8     '<' : lambda x: x[0] < x[1],
9     '<=' : lambda x: x[0] <= x[1],
10    '==' : lambda x: x[0] == x[1],
11    'sqrt': lambda x: torch.sqrt(x[0]),
12    'exp': lambda x: torch.exp(x[0]),
13    'log': lambda x: torch.log(x[0]),
14    'vector': vector,
15    'list': list,
16    'get': get,
17    'put': put,
18    'hash-map': hash_map,
19    'first': lambda x: x[0][0],
20    'last': lambda x: x[0][-1],
21    'nth': lambda x: x[0][int(x[1].item())],
22    'second': lambda x: x[0][1],
23    'rest': lambda x: x[0][1:],
24    'append': append,
25    'cons': lambda x: append([x[1], x[0]]),
26    'conj': append,
27    'mat-add': lambda x: x[0] + x[1],
28    'mat-mul': lambda x: torch.matmul(x[0], x[1]),
29    'mat-transpose': lambda x: x[0].T,
30    'mat-tanh': lambda x: x[0].tanh(),
31    'mat-repmat': lambda x: x[0].repeat((int(x[1].item()), int(x[2].item())))
32 }
```

Listing 1: primitives.py - Base primitives

```

1 def vector(x):
2     try:
3         vector = torch.stack(x)
4     except:
5         vector = x
6     return vector
7
8 def list(x):
9     try:
10        list = torch.stack(x)
11    except:
12        list = x
13    return list
14
15 def get(x):
16     if type(x[0]) == dict:
17         value = x[0][x[1].item()]
18     else:
19         value = x[0][x[1].long()]
20     return value
21
22 def put(x):
23     if type(x[0]) == dict:
24         x[0][x[1].item()] = x[2]
25     else:
26         x[0][x[1].long()] = x[2]
27     return x[0]
28
29 def hash_map(x):
30     keys = x[:, :2]
31     value = x[:, 2:]
32     new_keys = []
33     for key in keys:
34         try:
35             new_keys.append(key.item())
36         except:
37             new_keys.append(key)
38     result = dict(zip(new_keys, value))
39     return result
40
41 def append(x):
42     first = x[0]
43     second = x[1]
44
45     if first == 'vector':
46         first = torch.tensor([])
47     elif first.dim() == 0:
48         first = first.unsqueeze(0)
49     if second == 'vector':
50         second = torch.tensor([])
51     if second.dim() == 0:
52         second = second.unsqueeze(0)
53     return torch.cat((first, second))

```

Listing 2: primitives.py - Functions

```

1 import torch
2 from torch import distributions
3
4 class Dist:
5     def __init__(self, name, distribution, num_par, *par):
6         self.name = name
7         self.distribution = distribution
8         self.num_par = num_par
9         self.pars = []
10        for i in range(num_par):
11            self.pars.append(par[i])
12
13    def sample(self):
14        return self.distribution.sample()
15
16 class normal(Dist):
17     def __init__(self, pars):
18         mean = pars[0]
19         var = pars[1]
20         super().__init__('normal', distributions.Normal(mean, var), 2, mean, var)
21
22 class beta(Dist):
23     def __init__(self, pars):
24         alpha = pars[0]
25         betta = pars[1]
26         super().__init__('beta', distributions.Beta(alpha, betta), 2, alpha, betta)
27
28 class exponential(Dist):
29     def __init__(self, par):
30         lamda = par[0]
31         super().__init__('exponential', distributions.Exponential(lamda), 1, lamda)
32
33 class uniform(Dist):
34     def __init__(self, pars):
35         a = pars[0]
36         b = pars[1]
37         super().__init__('uniform', distributions.Uniform(a, b), 2, a, b)
38
39 class discrete(Dist):
40     def __init__(self, pars):
41         prob = pars[0]
42         super().__init__('discrete', distributions.Categorical(prob), 0)
43
44 class bernoulli(Dist):
45     def __init__(self, pars):
46         p = pars[0]
47         super().__init__('bernoulli', distributions.Bernoulli(p), 1, p)

```

Listing 3: primitives.py - Distributions

```

1 distlist = {
2     'normal' : normal,
3     'beta' : beta,
4     'exponential' : exponential,
5     'uniform' : uniform,
6     'discrete' : discrete,
7     'bernoulli': bernoulli
8 }

```

Listing 4: primitives.py - distlist

1.1.2 evaluate program

```
1 from primitives import baseprimitives, distlist
2
3
4 def evaluate_program(ast):
5     """Evaluate a program as desugared by daphne, generate a sample from the prior
6     Args:
7         ast: json FOPPL program
8     Returns: sample from the prior of ast
9     """
10    funcs = {}
11    final_ast = ast
12    if isinstance(ast, list):
13        if isinstance(ast[0], list):
14            if ast[0][0] == 'defn':
15                for statement in ast:
16                    if statement[0] == 'defn':
17                        funcs[statement[1]] = (statement[2], statement[3], statement[3])
18                        final_ast = final_ast[1:]
19                    else:
20                        result = eval(statement, {}, {}, funcs)
21                if final_ast[0][0] != 'defn':
22                    result = eval(final_ast[0], {}, {}, funcs)
23            else:
24                result = eval(ast, {}, {}, funcs)
25    else:
26        result = eval(ast, {}, {}, funcs)
27    return result[0]
```

Listing 5: evaluation-based_sampling.py - evaluate_program

1.1.3 eval

```
1 def eval(x, sigma, l, funcs):
2     "Evaluate an expression in an environment."
3     if isinstance(x, list) and len(x) == 1:
4         x = x[0]
5     if not isinstance(x, list):
6         if isinstance(x, int) or isinstance(x, float):
7             result = torch.tensor(x, dtype=float)
8         elif x in baseprimitives or torch.is_tensor(x) or x in funcs or x in distlist:
9             result = x
10        else:
11            result = l[x]
12    elif x[0] == 'if':
13        cond_result, sigma = eval(x[1], sigma, l, funcs)
14        if cond_result:
15            result = x[2]
16        else:
17            result = x[3]
18    elif x[0] == 'let':
19        result_temp, sigma = eval(x[1][1], sigma, l, funcs)
20        for e in x[2:-1]:
21            _, sigma = eval(e, sigma, {**l, **{x[1][0]: result_temp}}, funcs)
22        result, sigma = eval(x[-1], sigma, {**l, **{x[1][0]: result_temp}}, funcs)
23    elif x[0] == 'sample':
24        dist, sigma = eval(x[1], sigma, l, funcs)
25        result = dist.sample()
26    elif x[0] == 'observe':
27        result = None
28    else:
29        statements = []
30        for expression in x:
```

```

31         statement, sigma = eval(expression, sigma, l, funcs)
32         statements.append(statement)
33
34         first_statemnt, other_statements = statements[0], statements[1:]
35         if first_statemnt in baseprimitives:
36             result = baseprimitives[first_statemnt](other_statements)
37         elif first_statemnt in distlist:
38             result = distlist[first_statemnt](other_statements)
39
40         elif first_statemnt in funcs:
41             _, variables, process = funcs[first_statemnt]
42             assignment = {key:value for key, value in zip(variables, other_statements)}
43             result, sigma = eval(process, sigma, {**l, **assignment}, funcs)
44         else:
45             result = torch.tensor(statements)
46     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:

-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

1.2.3 Histograms

1.2.3.1 Task 1

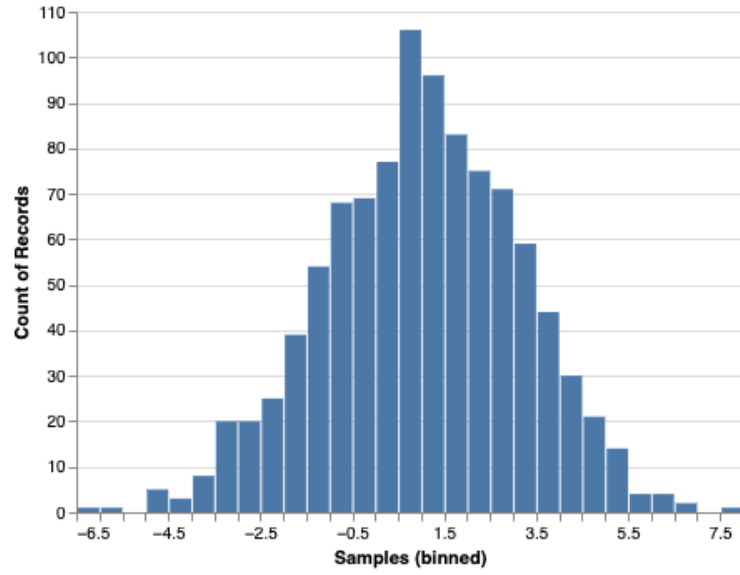


Figure 1: Histogram of μ

1.2.3.2 Task 2

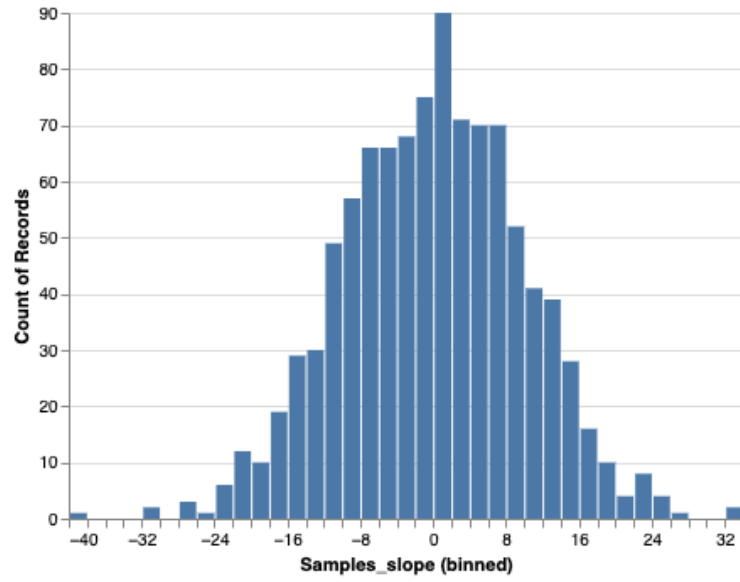


Figure 2: Histogram of slope

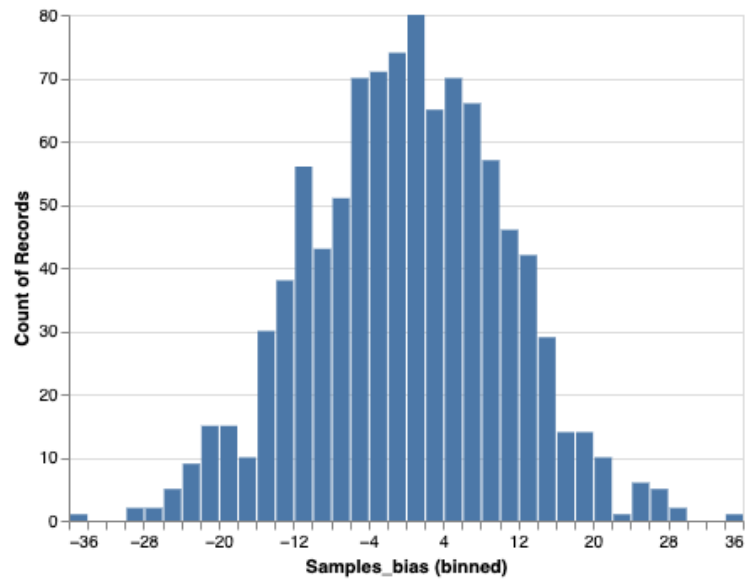


Figure 3: Histogram of bias

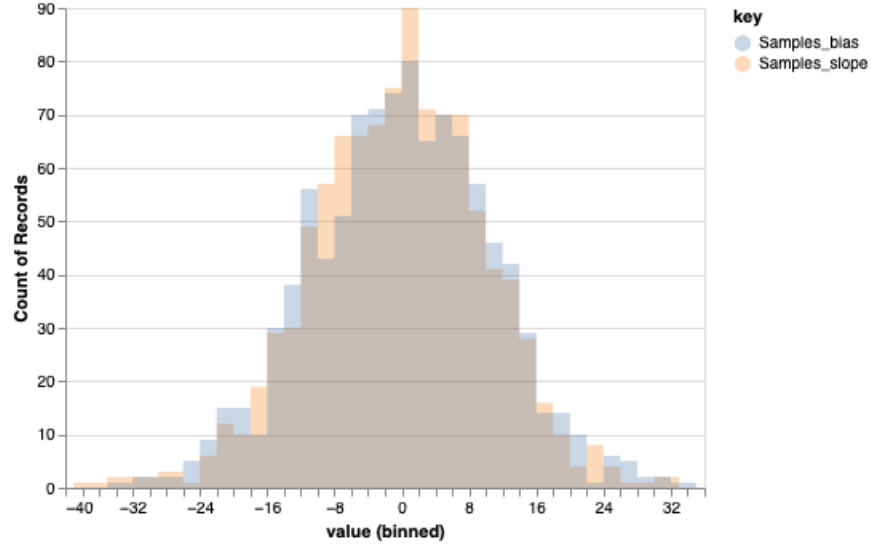


Figure 4: Histogram of slope and bias together

1.2.3.3 Task 3

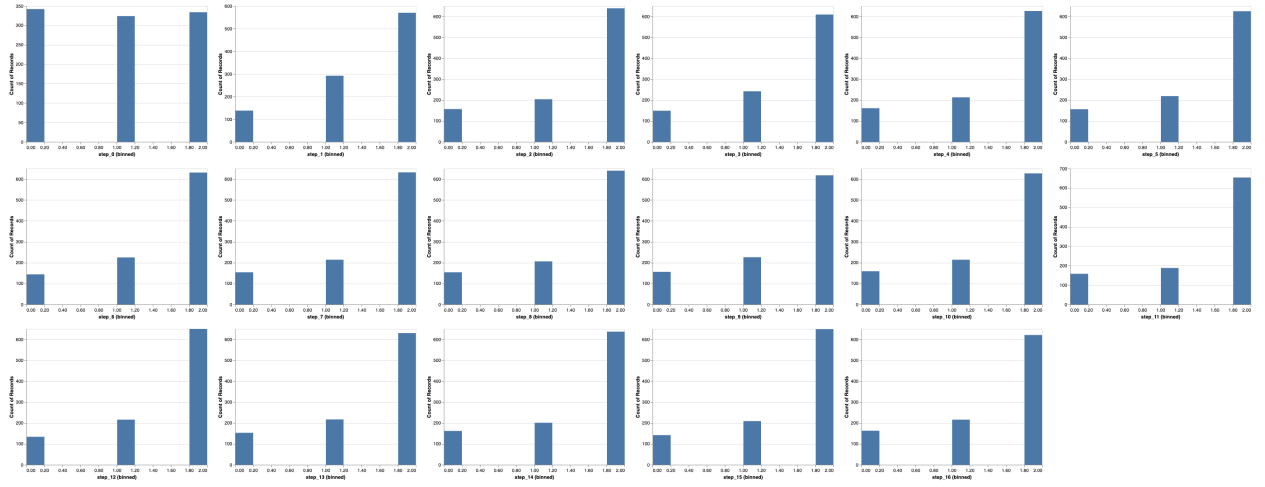


Figure 5: Histogram of states in each step

1.2.3.4 Task 4

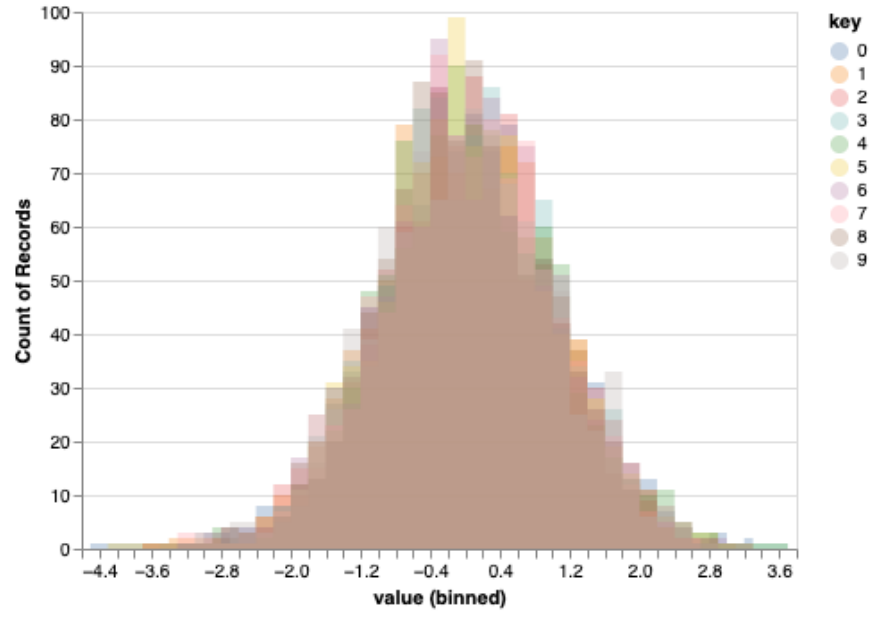


Figure 6: Histogram of W_1 elements together

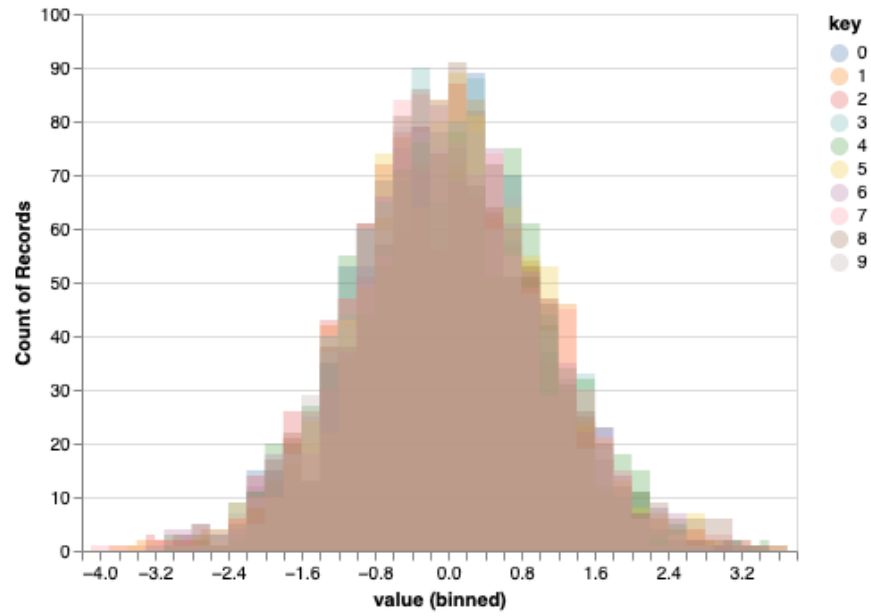


Figure 7: Histogram of b_1 elements together

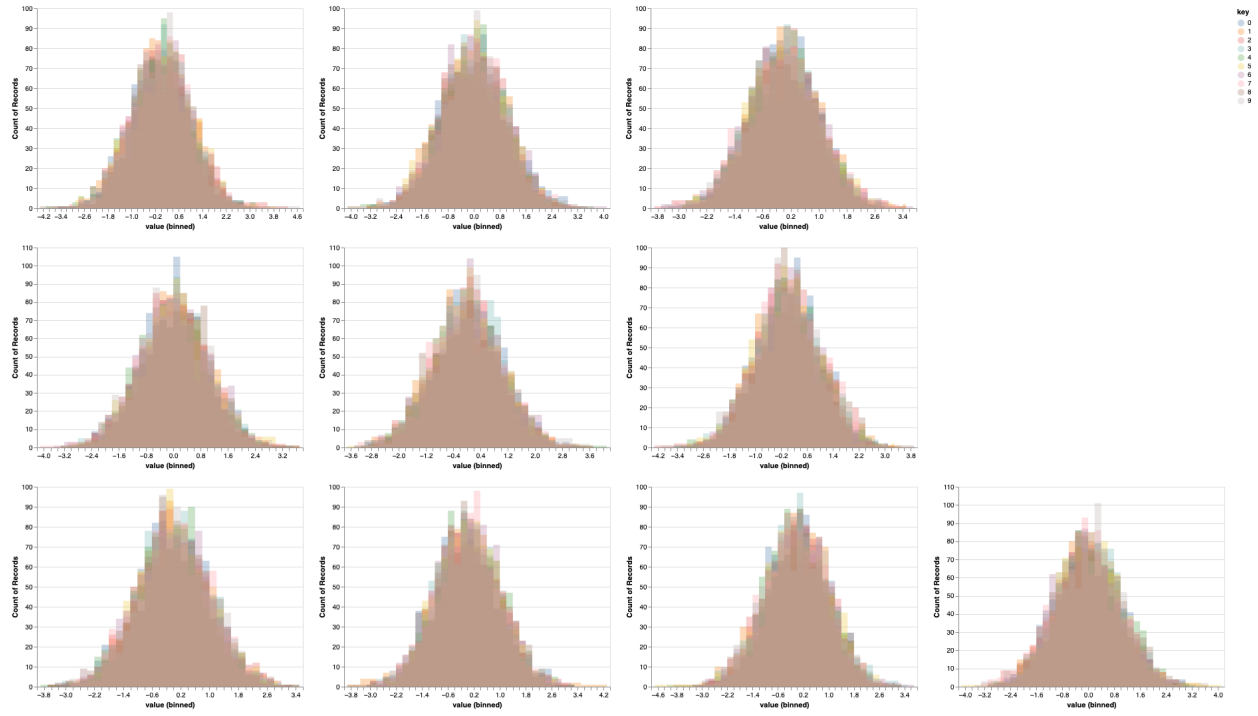


Figure 8: Histogram of W_2 elements together

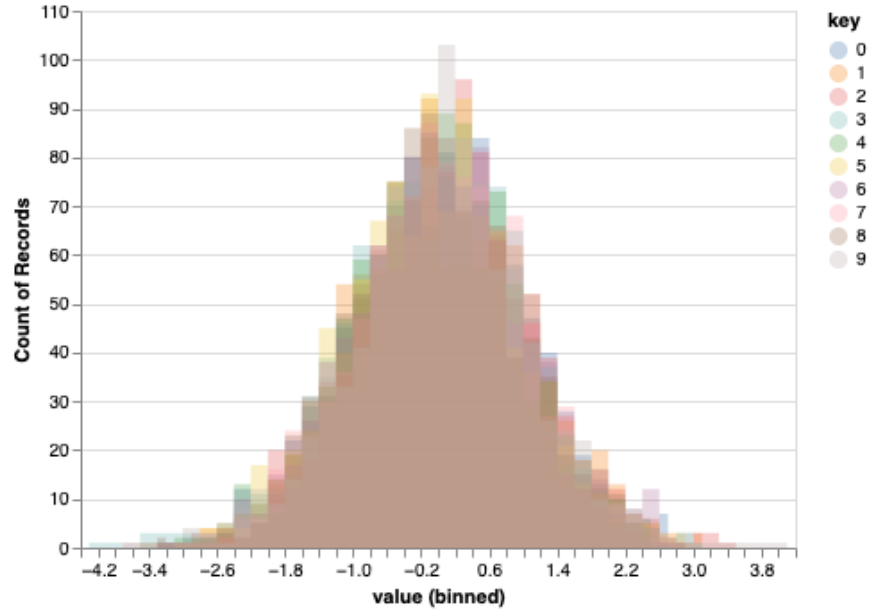


Figure 9: Histogram of b_2 elements together

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

```
1 def topological_sort(graph):
2     nodes = graph[1]['V']
3     edges = graph[1]['A']
4     is_visited = dict.fromkeys(nodes, False)
5     node_stack = []
6     node_order_reverse = []
7     for node in nodes:
8         if not is_visited[node]:
9             node_stack.append((node, False))
10        while len(node_stack) > 0:
11            node, flag = node_stack.pop()
12            if flag:
13                node_order_reverse.append(node)
14                continue
15            is_visited[node] = True
16            node_stack.append((node, True))
17            if node not in edges:
18                continue
19            children = edges[node]
20            for child in children:
21                if not is_visited[child]:
22                    node_stack.append((child, False))
23    return node_order_reverse[::-1]
```

Listing 7: graph_based_sampling.py - topological_sort

2.1.3 environment

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

Listing 8: graph_based_sampling.py - environment

2.1.4 deterministic eval

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

Listing 9: graph_based_sampling.py - deterministic_eval

2.1.5 value substitution

```
1 def value_subs(expressions, variables):
2     if isinstance(expressions, list):
3         result = []
4         for expression in expressions:
5             result.append(value_subs(expression, variables))
6     else:
7         if expressions in variables:
8             result = variables[expressions]
9         else:
10            result = expressions
11    return result
```

Listing 10: graph_based_sampling.py - value_subs

2.1.6 sample from joint

```
1 def sample_from_joint(graph):
2     "This function does ancestral sampling starting from the prior."
3     node_order = topological_sort(graph)
4     results = {}
5     for node in node_order:
6         first_statement, *other_statements = graph[1]['P'].get(node)
7         if first_statement == 'sample*':
8             dist = deterministic_eval(value_subs(other_statements, results))
9             result = dist.sample()
10        if first_statement == 'observe*':
11            result = graph[1]['Y'].get(node)
12        results[node] = result
13    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:

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

2.2.3 Histograms

2.2.3.1 Task 1

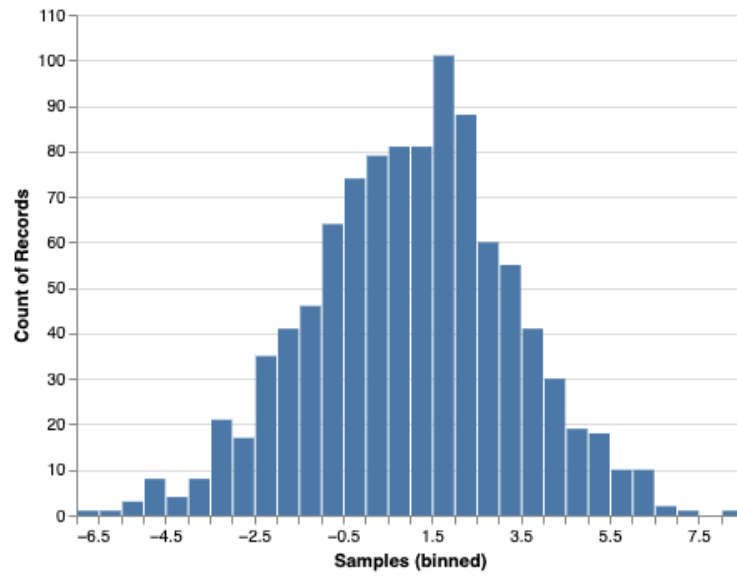


Figure 10: Histogram of μ

2.2.3.2 Task 2

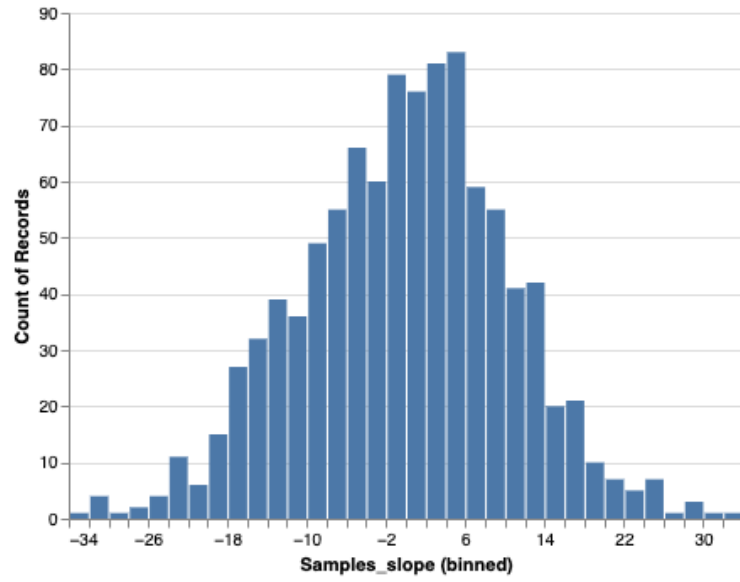


Figure 11: Histogram of slope

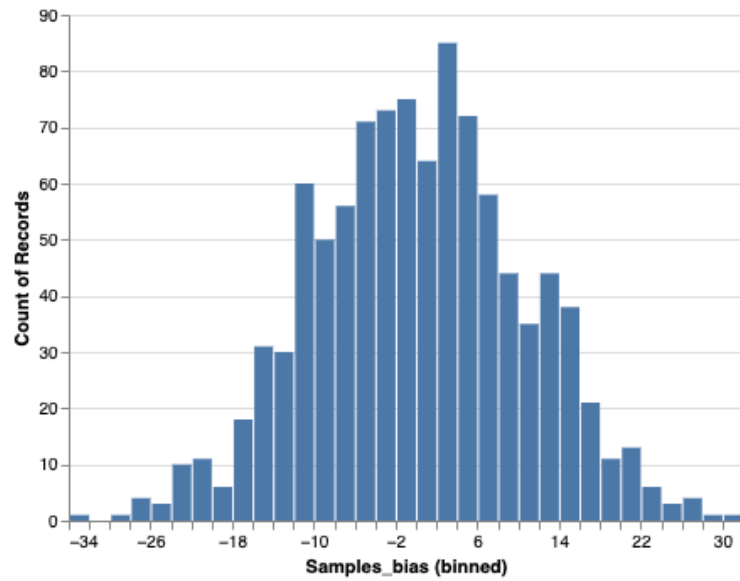


Figure 12: Histogram of bias

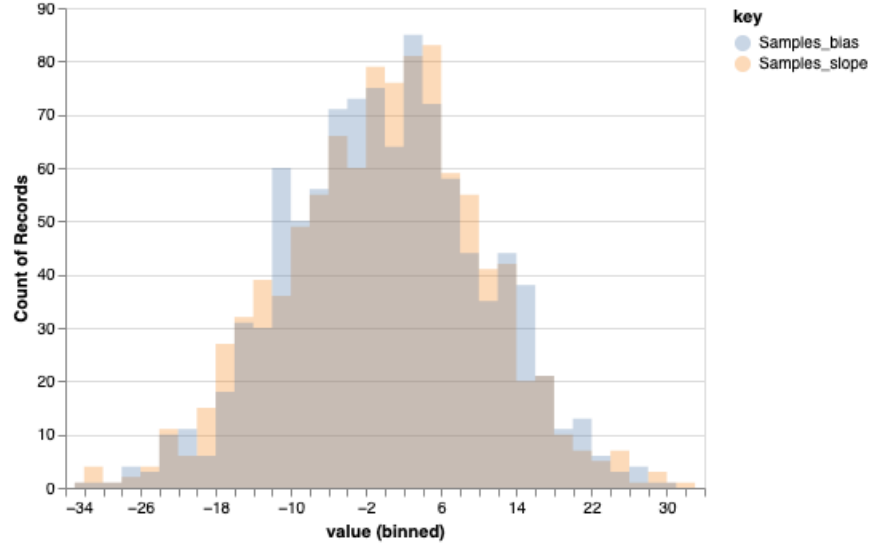


Figure 13: Histogram of slope and bias together

2.2.3.3 Task 3

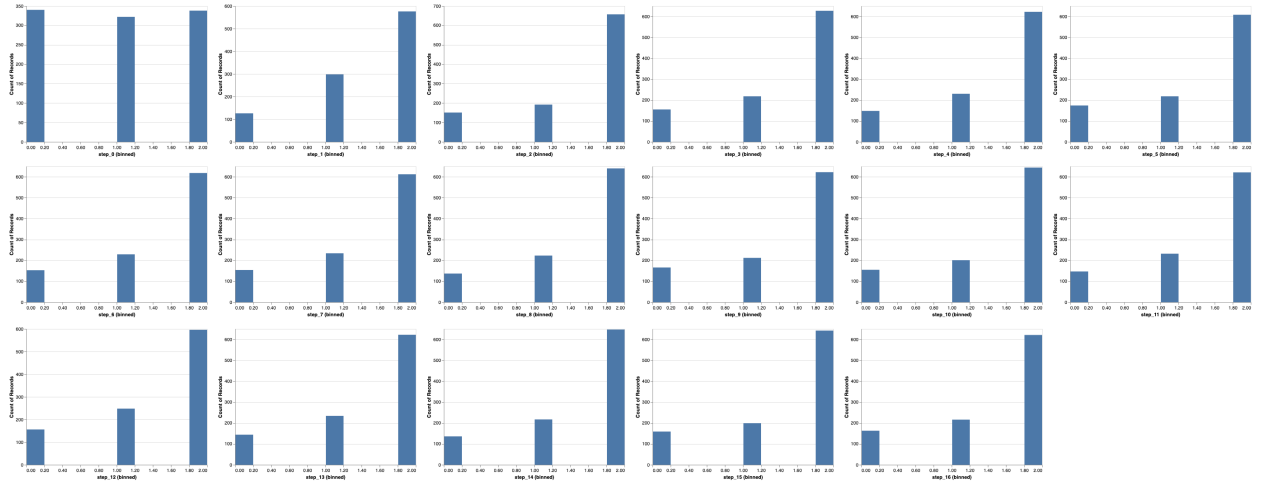


Figure 14: Histogram of states in each step

2.2.3.4 Task 4

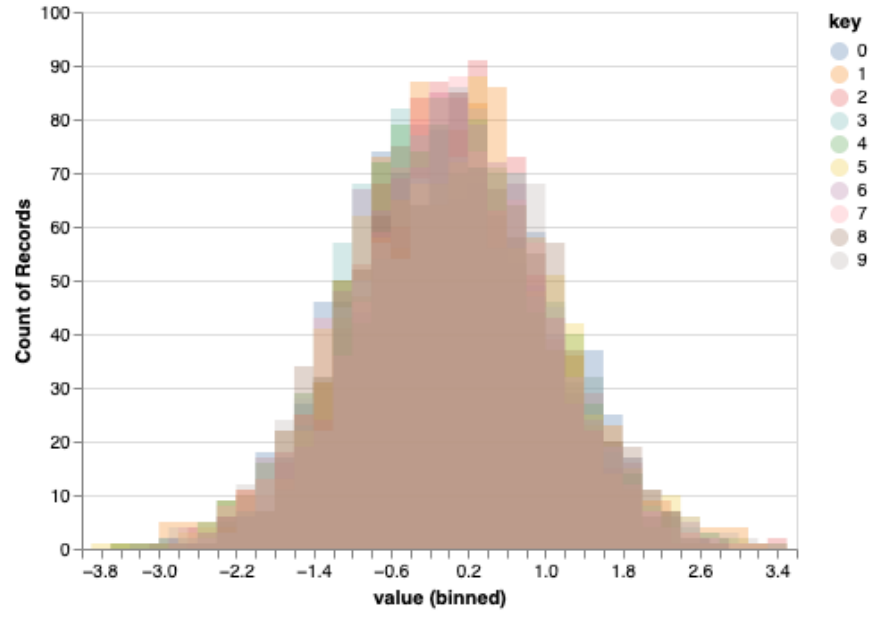


Figure 15: Histogram of W_1 elements together

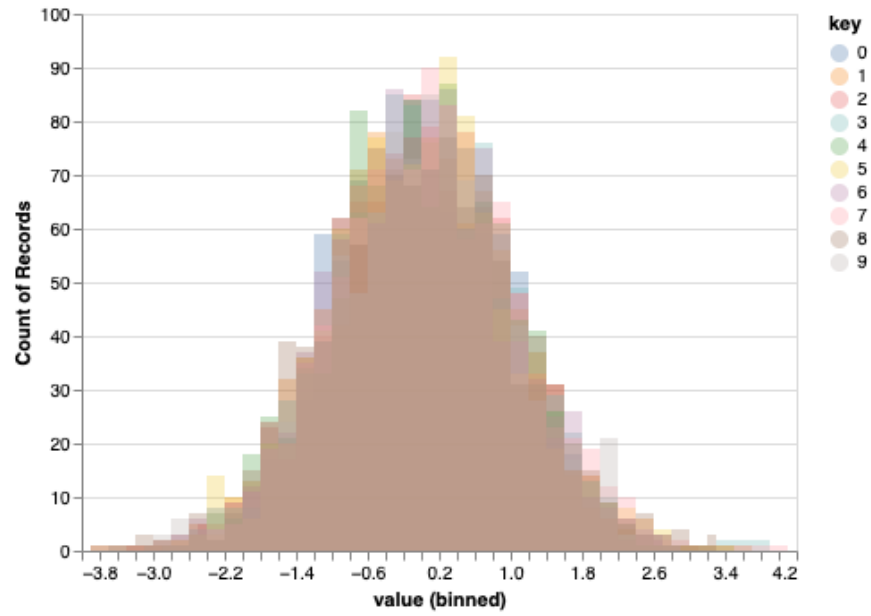


Figure 16: Histogram of b_1 elements together

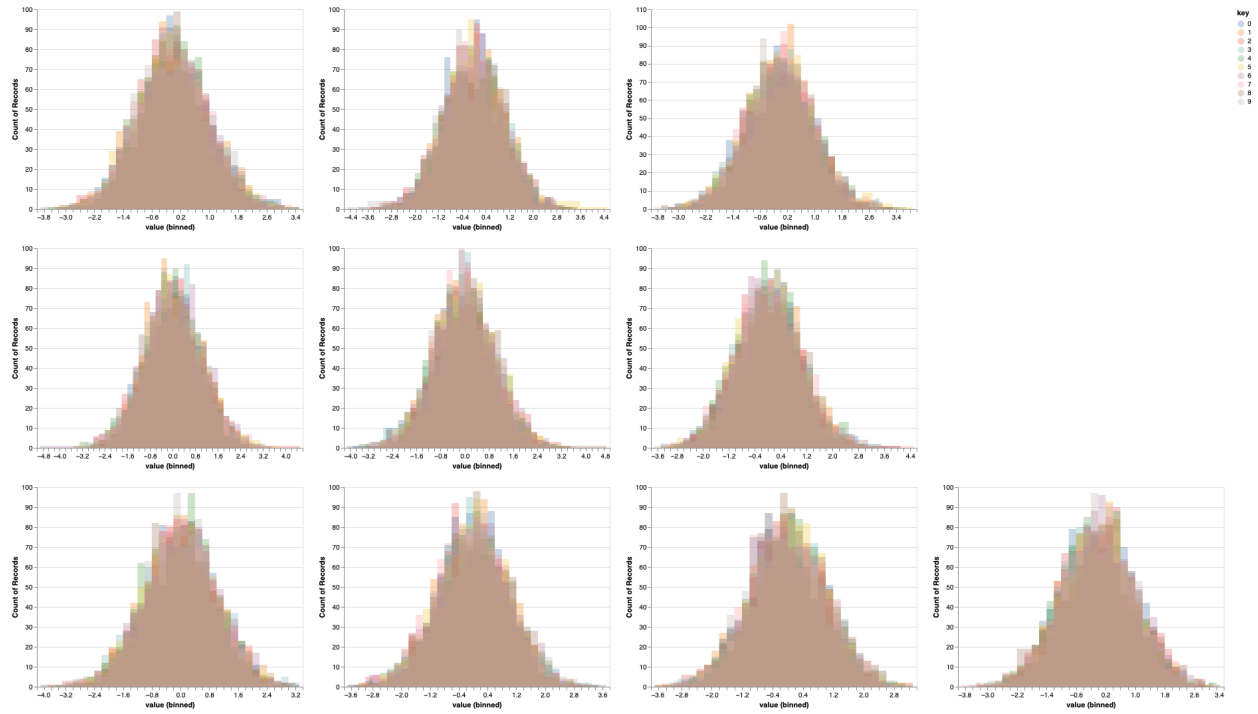


Figure 17: Histogram of W_2 elements together

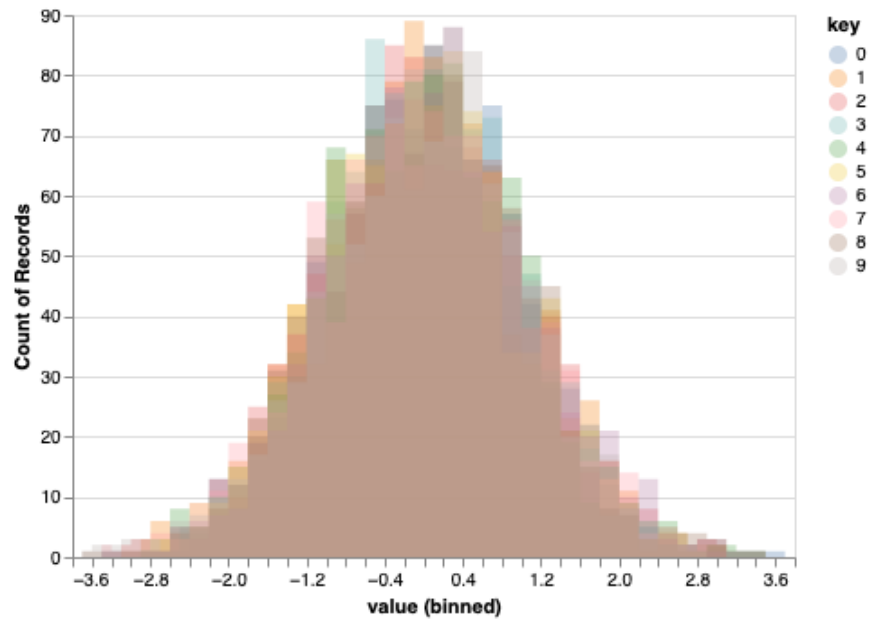


Figure 18: Histogram of b_2 elements together