MIT 6.S890 Topics in Multiagent Learning Fall 2024

Homework 3

Released on: Nov. 12, 2024 **Due on: Nov. 24, 2024, 11:59pm ET**

Instructions This homework contains three problems (each split into smaller tasks), for a total of 60 points.

Collaboration policy You are welcome to discuss the problems with other students. However you should write up the solutions by yourself.

1 The 0/1-Polyhedral Kernel (20 points)

As we discussed in class, the 0/1-polyhedral kernel introduces a computationally tractable way to handle the exponentially large set $\mathcal{V} \subseteq \{0,1\}^d$. Specifically, we define the 0/1-polyhedral feature map $\phi_{\mathcal{V}} : \mathbb{R}^d \to \mathbb{R}^{\mathcal{V}}$ as follows:

$$\phi_{\mathcal{V}}(x)[v] := \prod_{k:v[k]=1} x[k], \quad \forall v \in \mathcal{V}.$$

The 0/1-polyhedral kernel $K_{\mathcal{V}} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is defined as the inner product of the feature map:

$$K_{\mathcal{V}}(x,y) := \langle \phi_{\mathcal{V}}(x), \phi_{\mathcal{V}}(y) \rangle_{\mathcal{V}} = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} x[k]y[k].$$

The kernel method enables handling the exponentially large space $\mathbb{R}^{\mathcal{V}}$ while requiring access only to the relatively small space $\mathbb{R}^d$. In this question, you will derive some properties of the 0/1-polyhedral kernel.

Problem 1.1 (2 points). Someone gives you a kernel $K_{\mathcal{V}}$ but doesn't tell you what $\mathcal{V}$ is. How do you find out the cardinality of $\mathcal{V}$?

Proof. To find the cardinality of $\mathcal{V}$, compute $|\mathcal{V}| = K_{\mathcal{V}}(x,y)$, where $x = y = \mathbf{1} \in \mathbb{R}^d$. We know that

$$K_{\mathcal{V}}(\mathbf{1},\mathbf{1}) = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} \mathbf{1}[k]\mathbf{1}[k] = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} 1 = \sum_{v \in \mathcal{V}} 1 = |\mathcal{V}|$$

Therefore $K_{\mathcal{V}}(\mathbf{1},\mathbf{1})$ is the cardinality of $\mathcal{V}$. □

Problem 1.2 (7 points). Someone gives you a kernel $K_{\mathcal{V}}$ but doesn't tell you what $\mathcal{V}$ is. How do you find out how many elements $v \in \mathcal{V}$ have the first bit hot, that is, $v[1] = 1$?

Proof. Let $x = [0,1,1,\cdots,1] \in \mathbb{R}^d$ (i.e., x only have 0 in the first position) and $y = \mathbf{1} \in \mathbb{R}^d$. We can compute $K_{\mathcal{V}}(x,y)$ and $|V| - K_{\mathcal{V}}(x,y)$ counts the number of elements $v \in \mathcal{V}$ that have the first bit hot.

We know that

$$K_{\mathcal{V}}(x,y) = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} x[k]\mathbf{1}[k] = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} x[k]$$

By the definition of x, we know that $x[k] = 0$ only when $k = 1$. Therefore, $x[k] = 0$ only if the first

position of v is 1, which means

$$\prod_{k:v[k]=1} x[k] = 0 \iff v[k] = 1$$

So we can write

$$K_{\mathcal{V}}(x, y) = \sum_{v\in\mathcal{V}, v[k]\neq 1} 1 = \#v \in V, v[1] \neq 1$$

i.e., $K_{\mathcal{V}}(x, y)$ counts the number of $v \in \mathcal{V}$ where $v[1] \neq 1$. So, $|V| - K_{\mathcal{V}}(x, y)$ counts the number of v where $v[1] = 1$. By the earlier part, we can also write this number as

$$K_{\mathcal{V}}(y, y) - K_{\mathcal{V}}(x, y)$$

□

Problem 1.3 (8 points). Someone gives you a kernel $K_{\mathcal{V}}$ but doesn't tell you what $\mathcal{V}$ is. How do you find out how many elements $v \in \mathcal{V}$ have the first two bits hot, that is, $v[1] = v[2] = 1$?

Proof. Define

$$x_1 = [0, 1, 1, \cdots, 1] \in \mathbb{R}^d \qquad x_2 = [1, 0, 1, \cdots, 1] \in \mathbb{R}^d x_3 = [0, 0, 1, \cdots, 1] \in \mathbb{R}^d \qquad y = \mathbf{1} \in \mathbb{R}^d$$

i.e., $x_1[1] = x_2[2] = x_3[1] = x_3[2]$ only, all other positions are 1's. We argued before that $K_{\mathcal{V}}(x_1, y)$ counts the number of $v \in cV$ such that $v[1] \neq 1$. By a similar argument, we can prove that

$$K_{\mathcal{V}}(x_2, y)$$

counts the number of $v \in cV$ such that $v[2] \neq 1$. We further know that

$$K_{\mathcal{V}}(x_3, y) = \sum_{v\in\mathcal{V}} \prod_{k:v[k]=1} x[k]\mathbf{1}[k] = \sum_{v\in\mathcal{V}} \prod_{k:v[k]=1} x_3[k]$$

We know that $x_3[k] = 0$ only if $k = 1$ or $k = 2$. Therefore,

$$K_{\mathcal{V}}(x_3, y) = \sum_{v\in\mathcal{V}, v[1]=v[2]=0} 1$$

i.e., $K_{\mathcal{V}}(x_3, y)$ counts the number of $v \in \mathcal{V}$ where $v[1] = v[2] = 0$.

Recall that

$$|A \cup B| = |A| + |B| - |A \cap B|$$

So the total number of $v \in \mathcal{V}$ where either $v[1] = 0$ or $v[2] = 0$ is

$$K_{\mathcal{V}}(x_1, y) + K_{\mathcal{V}}(x_2, y) - K_{\mathcal{V}}(x_3, y)$$

So the total number of $v \in \mathcal{V}$ where $v[1] = v[2] = 1$ is

$$|\mathcal{V}| - (K_{\mathcal{V}}(x_1, y) + K_{\mathcal{V}}(x_2, y) - K_{\mathcal{V}}(x_3, y))$$

which is

$$K_{\mathcal{V}}(y, y) - K_{\mathcal{V}}(x_1, y) - K_{\mathcal{V}}(x_2, y) + K_{\mathcal{V}}(x_3, y)$$

□

Problem 1.4 (3 points). Show that, given oracle access to the kernel function $K_{\mathcal{V}}(x, 1)$ for any $x \in \mathbb{R}^d$, one can compute $K_{\mathcal{V}}(u, v)$ for any $u, v \in \mathbb{R}^d$ in polynomial time.

Proof. Define $m \in \mathbb{R}^d$, where $m[k] = u[k]v[k]$. By definition

$$K_{\mathcal{V}}(m, 1) = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} m[k]\mathbf{1}[k] = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} u[k]v[k]\mathbf{1}[k] = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} u[k]v[k] = K_{\mathcal{V}}(u, v)$$

Note that the last step is by definition. Since computing m is in $O(d)$, which is polynomial in d, and we can query the oracle $K_{\mathcal{V}}(m, 1)$, we can compute $K_{\mathcal{V}}(u, v)$ in polynomial time. □

2 Kernel on Combinatorial Games (30 points)

In this problem, you will compute the 0/1 polyhedral kernel for certain combinatorial game instances. As some games are large, you may want to compute the kernel operations programmatically.

2.1 m-set (10 points)

An m-set refers to the decision problem of choosing m distinct actions from d available actions simultaneously. The pure strategy of the decision problem can be represented as a vector of length d, where the entries corresponding to the m chosen actions are set to 1. In this way, the vertices of an m-set are given by $\mathcal{V} := \{v \in \{0, 1\}^d : \|v\|_1 = m\}$.

Problem 2.1 (5 points). For an m-set instance, explain how one can evaluate the kernel function in time polynomial in m and d.

Proof. We can consider a polynomial in γ:

$$P(\gamma) = \prod_{i=1}^{d} (1 + x[i]y[i] \cdot \gamma)$$

And the coefficient on γ^m will be $K_{\mathcal{V}}(x, y)$. To show that, note that to obtain γ^m, we need to pick the term $x[i]y[i] \cdot \gamma$ exactly m times during the expansion. Let S be the set of valid such combinations (i.e., all possible combinations from selecting m elements from the set $[\![d]\!]$ with d elements) and $s \in S$ be s contain m non-repeating indices from $[\![d]\!]$. Then the coefficient on γ^m would be

$$\sum_{s \in S} \prod_{s_i \in s} x[s_i]y[s_i]$$

Note that S is a set that contains all valid combinations of selecting m elements from the set $[\![d]\!]$ with d elements. This is exactly the same as the definition of the m set. Therefore, we could write the notations slightly:

$$\sum_{s \in S} \prod_{s_i \in s} x[s_i]y[s_i] = \sum_{v \in \mathcal{V}} \prod_{k:v[k]=1} x[k]y[k] := K_{\mathcal{V}}(x, y)$$

Therefore the coefficient on γ^m is equal to $K_{\mathcal{V}}(x, y)$ by definition. Note that in the above derivation, we changed the notation slightly, by changing s which contains m elements to v, which contains the bit-vector form of s (i.e., $v[k] = 1$ where $k \in s$). This allows us to write the expression in terms of $\mathcal{V}$ and does not change the result.

So, to compute $K_{\mathcal{V}}(x, y)$, we need to compute the coefficient of γ^m, which can be done in $O(d)$ times by dynamic programming on $P(\lambda)$ □

Problem 2.2 (2 points). For an m-set instance with $d = 5$ and $m = 2$, compute the kernel function $K_{\mathcal{V}}(x, 1)$ for the vector $x[i] = i/d$ for $i \in [\![d]\!]$.

Proof. We evaluated

$$K_{\mathcal{V}}(x, 1) = 3.4$$

Please see the code attached □

Problem 2.3 (3 points). For an m-set instance with $d = 40$ and $m = 10$, compute the kernel function $K_{\mathcal{V}}(x, 1)$ for the vector $x[i] = i/d$ for $i \in [\![d]\!]$.

Proof. We evaluated

$$K_{\mathcal{V}}(x, 1) = 714702.9738912641$$

Please see the code attached □

```
def all_coefficients(x, y, m):
    dp = [0] * (len(x) + 1)
    dp[0] = 1
    for xi, yi in zip(x,y):
        for k in range(len(x), 0, -1):
            dp[k] += xi * yi * dp[k - 1]
    return dp

def compute_kv(d, m):
    x = [i / d for i in range(1, d+1)]
    y = [1] * d
    dp = all_coefficients(x, y, m)
    return dp[m]

if __name__ == "__main__":
    print(compute_kv(d = 5, m = 2))
    print(compute_kv(d = 40, m = 10))

## output
## 3.4000000000000004
## 714702.9738912641
```

2.2 Colonel Blotto (10 points)

The Colonel Blotto game is a type of allocation game in which two players each write down m *ordered positive* integers that sum to a pre-specified number S. Subsequently, the two players reveal their choices and compare the corresponding numbers. The player who has more numbers higher than the corresponding ones of the opponent wins the game.

The pure strategy of each player's decision problem can be represented as a vector of length $m \times S$, where the entry (i, j) (using a two-dimensional index for clarity) is set to 1 when the i-th number is j. Of course, only bit strings that denote valid allocations are considered.

Problem 2.4 (4 points). For an Colonel Blotto instance with $S = 6$ and $m = 3$, compute the kernel function $K_{\mathcal{V}}(x, 1)$ for the vector $x[i, j] = i/m + j/S$ for $i \in [\![m]\!]$ and $j \in [\![S]\!]$. Explain how your algorithm works.

Proof. The kernel is

$$K_{\mathcal{V}}(x, 1) = \sum_{v \in V} \prod_{(i,k):v[i,j]=1} x[i, j]$$

Define $\mathcal{V}_S^m$ to be the set of all valid resource allocations that allocate S resource to m tasks. Then $\mathcal{V}_{S-k}^{m-1}$ is the set of all valid resource allocations that allocate $S - k$ resources on $m - 1$ tasks. i.e., $\mathcal{V}_{S-k}^{m-1}$ also implicitly denotes the set of all valid resource allocations of S resources on m tasks that put k resource on the last task. So, we can write the kernel function as

$$K_{\mathcal{V}_S^m}(x, 1) = \sum_{k=1}^{S-1} \sum_{v \in \mathcal{V}_{S-K}^{m-1}} \prod_{(i,k):v[i,j]=1} x[i, j]x[m, k] = \sum_{k=1}^{S-1} x[m, k] \sum_{v \in \mathcal{V}_{S-k}^{m-1}} \prod_{(i,k):v[i,j]=1} x[i, j]$$

Note that by definition $\sum_{v \in \mathcal{V}_{S-k}^{m-1}} \prod_{(i,k):v[i,j]=1} x[i, j] = K_{\mathcal{V}_{S-k}^{m-1}}(x, 1)$. Also note that $K_{\mathcal{V}_k^1}(x, 1) = x[1, k]$. So can compute the kernel function via dynamic programming using the following recurrence relations:

$$K_{\mathcal{V}_k^1}(x, 1) = x[1, k]$$
$$K_{\mathcal{V}_j^i}(x, 1) = \sum_{k=1}^{j-1} x[m, k] K_{\mathcal{V}_{j-k}^{i-1}}(x, 1)$$

We also provide pseudo-code at the end. For $m = 3, S = 6$, we evaluate

$$K_{\mathcal{V}}(x, 1) = 8.499999999999998$$

```
// pseudo-code
Initialize V: shape [m, S], filled with 0.
V[1, *] <-- x[1, *]
for i = 1, ..., m-1:
    for j  = 1, ..., S:
        for k = 1, ..., S:
            next = j + k
            if next <= S:
                V[i+1][next] += V[i, j] * x[i+1, k]
kernel = V[m, S] #result
// end
```

□

Problem 2.5 (6 points). For an Colonel Blotto instance with $S = 40$ and $m = 10$, compute the kernel function $K_{\mathcal{V}}(x, 1)$ for the vector $x[i, j] = i/m + j/S$ for $i \in [\![m]\!]$ and $j \in [\![S]\!]$. Explain how your algorithm works. Does it always run in time polynomial in m and S?

Proof. The algorithm is the same. For $S = 40$ and $m = 10$, we evaluate

$$K_{\mathcal{V}}(x, 1) = 759669.923410581$$

This algorithm always runs in polynomial time in m and S. More specifically, we need to run m iterations to get to $\mathcal{V}^m$. For every iteration i, we need to compute S elements $\mathcal{V}_j^i$. To compute $\mathcal{V}_j^i$, we run $j-1 < S$ operations to compute the sum. So, the overall run time is in $O(m \cdot S^2)$, which is polynomial in m, S □

```python
import numpy as np

def blotto(m, S, x):
    weights = np.zeros((m, S))
    weights[0] = x[0]
    for i in range(0, m-1):
        for j in range(S):
            for k in range(1, S+1):
                next_resource = j + k
                if next_resource < S:
                    weights[i+1][next_resource] += weights[i, j] * x[i+1, k-1]
    return weights[-1, -1]

def get_x(m, S):
    x = np.zeros((m, S))
    for i in range(m):
        for j in range(S):
            x[i, j] = (i+1) / m + (j+1) / S
    return x

if __name__ == "__main__":
    m, S = 3, 6
    x = get_x(m, S)
    # ans_1 = get_blotto_result(m, S, x)
    ans_1 = blotto(m, S, x)
    # print(x)
    print(f"Answer 1 is {ans_1}")
    m, S = 10, 40
    x = get_x(m, S)
    ans_2 = blotto(m, S, x)
    print(f"Answer 2 is {ans_2}")
```

2.3 Online Shortest Path (10 points)

Online shortest path refers to the decision problem where an agent chooses a path from the source to the sink in a directed graph $G = (V, E)$. The adversary generates a weight assignment for each of the edges simultaneously, and the agent receives a loss equal to the summation of the weights of the chosen edges.

The pure strategy of the decision problem can be encoded using a vector of length equal to the number of edges, $|E|$, where the entries corresponding to the chosen edges are set to 1. In this way, the feasible solutions of the online shortest path problem are given by

$$\mathcal{V} := \{v \in \{0,1\}^{|E|} : \text{edges } e \text{ with } v[e] = 1 \text{ form a path from the source to the sink}\}.$$

Problem 2.6 (3 points). For the online shortest path instance in Figure 1, compute the kernel function $K_{\mathcal{V}}(x, 1)$ for the vector $x[(u, v)] = (v - u)/2$ for each directed edge $(u, v) \in E$. The source vertex is 1, and the sink is 4.

Explain how your algorithm works.

Proof. Note that computing the kernel, by definition, is equivalent to computing the sum of the weights of all possible paths from the source to the sink, where the weight is defined as the product of all the edges in the path.

To compute this, we can use dynamic programming. Define the sum of weights of all possible paths from u to v to be $w(u, v)$. Then note that

$$w(u, v) = \sum_{(j,v)\in E} w(u, j)$$

where E is the set of edges. i.e., the sum of weights of all paths from u to v is equal to the sum of the weights of paths from u to j times the weight on edge (j, v), where such edge exists. Further, we know the base case:

$$w(u, u) = 0$$

Since the graph is a DAG, we can write the following pseudo code:

```
// pseudo-code
V: vertex set, |V|: number of vertices, source: 1, sink: |V|
w(u, v): computes the weight of edge (u, v)
Initialize W: {i: 0 for in in |V|}
W[1] = 1
for u = 1, ...., |V|:
    for v s.t., (u, v) exists:
        W[v] += w(u, v) * W[u]
kernel = W[|V|] # result
// end
```

For 4 vertices, we evaluate

$$K_{\mathcal{V}}(x, 1) = 2.625$$

□

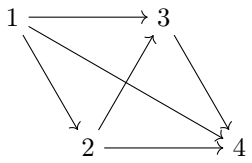

Figure 1: Online shortest path instance with source vertex 1 and sink vertex 4.

Problem 2.7 (7 points). For the online shortest path instance in Figure 2, compute the kernel function $K_{\mathcal{V}}(x, 1)$ for the vector $x[(u, v)] = (v - u)/2$ for each directed edge $(u, v) \in E$. The source vertex is 1, and the sink is 40.

Explain how your algorithm works. If the graph pattern were to be repeated, would your algorithm run in polynomial time in the number of vertices?

Proof. The algorithm is the same. For 40 vertices, the kernel evaluates to

$$K_{\mathcal{V}}(x, 1) = 2264174.3669983726$$

The algorithm runs in $O(|E| \cdot |V|)$, where V is the set of vertices and E is the set of edges. In the pseudo-code, we see that we need to do $|V|$ iterations, and in total, we need to visit all edges. So the

runtime is in $O(|E| \cdot |V|)$. In this case, since we know the shape of the graph and the number of edges is bounded by the number of vertices (i.e., $|E| < 10|V|$ for example), we can also conclude that this algorithm runs in polynomial time of the number of vertices (in the general DAG $|E| < |V|^2$, which also allows us to make this statement, but the bound is looser though it is still in polynomial time). □

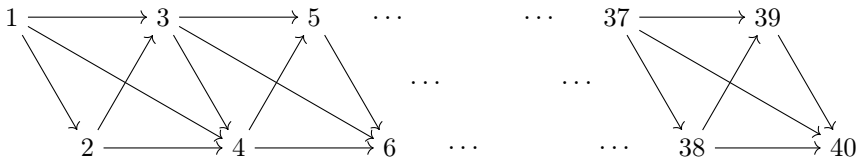

Figure 2: Graph with 40 vertices, showing two rows of vertices with directed edges. The first row consists of vertices 1, 3, 5, ..., 39, and the second row consists of vertices 2, 4, 6, ..., 40.

```python
def compute_x(u, v):
    return (v - u) / 2

def get_neighbor(u):
    if u % 2 == 1:
        return [u+1, u+2, u+3]
    else:
        return [u+1, u+2]


def dag(v):
    result = {i: 0 for i in range(1, v+1)}
    result[1] = 1
    for i  in range(1, v+1):
        neighbors = get_neighbor(i)
        for n in neighbors:
            if n in result:
                result[n] += compute_x(i, n) * result[i]
    return result[v]

if __name__ == "__main__":
    print(dag(4))
    print(dag(40))
```

3 Project Progress Report (10 points)

Problem 3.1 (10 points)**.** Give a progress report on your course project. Briefly describe the progress you have made so far. Additionally, outline any issues you have encountered (if any), and explain how you addressed or plan to address them.

6.S890 Topics in Multiagent Learning

Fall 2024