Name: Weiyi Zheng

Problem 1

- Let $b_1, \ldots, b_n$ be real numbers representing positions on a number line. Let $w_1, \ldots, w_n$ be positive numbers representing the importance of these positions. Define the quadratic function: $f(x) = \frac{1}{2} \sum_{i=1}^{n} w_i (x - b_i)^2$. What value x minimizes $f(x)$? You can think about this problem as trying to find the point x that's not too far away from the points b_i's. Over time, hopefully you'll appreciate how nice quadratic functions are to minimize.

Answer: The derivative of the quadratic function is $f'(x) = \sum_{i=1}^{n} w_i (x - b_i)$, $f(x)$ is minimized when the linear distance when the distance between x and highly weighted b_i are minimized

---

- In this class, there will be a lot of sums and maxes. Let's see what happens if we switch the order. Let $f(x) = \max_{a \in \{1,-1\}} \sum_{j=1}^{d} a x_j$ and $g(x) = \sum_{j=1}^{d} \max_{a \in \{1,-1\}} a x_j$, where $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$ is a real vector. Does $f(x) \leq g(x)$, $f(x) = g(x)$, or $f(x) \geq g(x)$ hold for all x? Prove it.

Answer:For All x $f(x) \leq g(x)$ holds.

Prove: $\forall b \in \mathbb{R}$, $\max_{a \in \{1,-1\}} ab$ equals to b when $b \geq 0$, $-b$ when $b < 0$. Thus we can treat $f(x)$ as $f(x) = |\sum_{j=1}^{d} x_j|$, $g(x) = \sum_{j=1}^{d} |x_j|$.

If all $x_j \geq 0$, Then $f(x) = g(x)$. If not, $f(x) < g(x)$

- Suppose you repeatedly roll a fair six-sided dice until the number of dots is 3 or fewer (and then you're done). Every time the dice turns up with a 6, you earn r points. What is the expected number of total points (as a function of r) that you will earn before you're done?

Answer: If dice roll is n, The probability of rolling a 6 should be $\mathbb{P}(\text{rolling a 6 per game that can continue}) = \mathbb{P}(n = 6 | n = \{4, 5, 6\}) = 1/3$. The total expected probability of being able to end the game at nth roll is $\frac{1}{2}^{n-1}\frac{1}{2} = \frac{1}{2}^{n}$. So the expected value

$\mathbb{E}(\text{points}) = r \times P(\text{4-6 for n-1 rolls}) \times P(6|\{4,5,6\}) \times P(\text{1-3 at nth roll})$

$= r \times (n-1) \times \frac{1}{3} \times \frac{1}{2}^{n}$

- Suppose the probability of a coin turning up heads is $0 < p < 1$, and that we flip it 5 times and get $\{\text{H}, \text{T}, \text{T}, \text{H}, \text{H}\}$. We know the probability (likelihood) of obtaining this sequence is $L(p) = p(1-p)(1-p)pp = p^3(1-p)^2$. Now let's go backwards and ask the question: what is the value of p that maximizes $L(p)$? Hint: consider taking the derivative of $\log L(p)$.

To maximize the probability of $L(p)$ is the same as maximizing $\log L(p)$, we take its derivative

$\frac{d \log L(p)}{dp} = 3/p + 2/(p-1)$

When the derivative reachs 0 we will have a maximum. So solve

$0 = 3/p + 2/(p-1), 0 < p < 1$

$p = 0.6$

- Let's practice taking gradients, which is a key operation for being able to optimize continuous functions. For $\mathbf{w} \in \mathbb{R}^d$ and constants $a_i, b_j \in \mathbb{R}^d$ and $\lambda \in \mathbb{R}$, define the scalar-valued function

$$f(\mathbf{w}) = \sum_{i=1}^{n}\sum_{j=1}^{n}(a_i^\top \mathbf{w} - b_j^\top \mathbf{w})^2 + \lambda \|\mathbf{w}\|_2^2,$$

where the vector is $\mathbf{w} = (w_1, \ldots, w_d)$ and $\|\mathbf{w}\|_2 = \sqrt{\sum_{j=1}^{d} w_j^2}$ is known as the L2 norm. Compute the gradient $\nabla f(\mathbf{w})$, which is a vector-valued function (i.e., for each $\mathbf{w} \in \mathbb{R}^d$, $\nabla f(\mathbf{w}) \in \mathbb{R}^d$).

Answer:

$$\frac{d\sum_{j=1}^{d} w_j^2}{dw_c} = \begin{cases} 2w_j & j = c \\ 0 & j \neq c \end{cases}$$

so

$$\frac{d\|\mathbf{w}\|_2^2}{d\mathbf{w}} = 2\mathbf{w}$$

We can use rule derivatives of sum equals the sum of derivatives to solve the first part. So the final equation is

$$\nabla f(\mathbf{w}) = \sum_{i=1}^{n}\sum_{j=1}^{n} 2(a_i^\top \mathbf{w} - b_j^\top \mathbf{w})(a_i - b_i) + 2\lambda\mathbf{w}$$

Problem 2

- Suppose we have an image of a human face consisting of $n \times n$ pixels. In our simplified setting, a face consists of two eyes and one mouth, each represented as an arbitrary axis-aligned rectangle. As we'd like to handle Picasso portraits too, there are no constraints on the location or size of the rectangles. How many possible faces (choice of its component rectangles) are there? In general, we only care about asymptotic complexity, so give your answer in the form of $O(n^c)$ or $O(c^n)$ for some integer c.

Answer: Since the eyes and mouth can overlap and no dimensionality constrains, we can assume finding one of the three rectangles that belongs to a face is independent of the other rectangles of the same face. Let's look at how many rectangles we can find in a $n \times n$ matrix.

Finding a rectangle of dimension $a \times b$, where $0 < a, b \leq n$, we should be able to find $(n - a + 1)(n - b + 1)$ rectangles with either side has a width > 0. So for all possible values of a and b, Total number of rectangles is

$$\sum_{a=1}^{n}\sum_{b=1}^{n}(n - a + 1)(n - b + 1)$$

Asymptoticlly, the equation above is equivalent to $O(n^4)$ (Each summation provides a $n(n-1)/2$ possible solution, which is in the order of $O(n^2)$)

Since we have to pick 3 of such rectangles of each face, the total amount of possibility should be $O(n^{4\times 3}) = O(n^{12})$

- Suppose we have n cities on the number line: $1, 2, \ldots, n$. Define a function $c(i, j)$ which returns the cost of going from city i to city j, and assume it takes constant time to compute. We want to travel from 1 to n via a set of intermediate cities, but only moving forwards. We can compute the minimum cost of doing this by defining the following recurrence: $f(j) = \min_{1<i<j}[c(i, j) + f(i)]$ for $j = 1, \ldots, n$. Give an algorithm for computing $f(n)$ for a fixed n in the most efficient way. What is the runtime (just give the big-O)?

Anwser: The answer is similar to use Dijkstra's algorithm to find the shortest path.

- let's define $g(j)$ to be the current cost of going from city 1 to j, and initialize all values to infinity.
- Add city 1 to a queue.
- Pop the first element off the queue, calculate the cost to reach all of its neighbors, which takes a max of O(n) to compute. Update $g(j) = min(g(i) + c(i, j), g(j))$. Add all the unvisited neighbors into the queue.
- Repeat the step above until we have no more elements in the queue. The $g(j)$ value is the minimal cost to reach from 1 to n Since there are a total of n vertex to visit, our algorithm will run in $O(n^2)$ time.

Suppose we have an $n \times n$ grid. How many ways are there to get from the upper-left corner to the lower-right corner if at each step you are only allowed to move down or right? Give your answer as a function of n. For example, if $n = 3$, then the answer is 6.

Anwser: In order to reach the bottom right corder, there have to be n right and n down in the total of 2n steps.

This is equivalent to picking the combination

$$\binom{n}{2n} = \frac{(2n)!}{n!n!} = \frac{(2n)!}{2n!}$$

Consider the scalar-valued function $f(\mathbf{w})$ from Problem 1e. Devise a strategy that first does preprocessing in $O(nd^2)$ time, and then for any given vector $\mathbf{w}$, takes $O(d^2)$ time instead to compute $f(\mathbf{w})$. Hint: refactor the algebraic expression; this is a classic trick used in machine learning.

Answer: Inside the summation, we have
$(a_i^T w - b_j^T w)^2 = w^T(a_i^T - b_j^T)^T(a_i^T - b_j^T)w = w^T(a_i a_i^T - a_i b_j^T - b_j a_i^T + b_j b_j^T)w$

1. Since the summation of $\sum_{i=1}^n \sum_{j=1}^n$ does not affect w, we can pull w out of the summation by associative rules
2. Summing over i does not affect j, vice versa, so we can simplify $\sum_{i=1}^n \sum_{j=1}^n (a_i b_j^T) = \sum_{i=1}^n a_i \sum_{j=1}^n b_j^T$
3. $\sum_{j=1}^n b_j^T = (\sum_{j=1}^n b_j)^T$, this also applies to $\sum_{i=1}^n a_i$

According to the three reasonings above, we can simplify

$$\sum_{i=1}^n \sum_{j=1}^n (a_i^\top \mathbf{w} - b_j^\top \mathbf{w})^2 = \sum_{i=1}^n \sum_{j=1}^n w^T(a_i a_i^T - a_i b_j^T - b_j a_i^T + b_j b_j^T)w$$

$$= w^T(\sum_{i=1}^n \sum_{j=1}^n a_i a_i^T - a_i b_j^T - b_j a_i^T + b_j b_j^T)w$$

$$= w^T(\sum_{i=1}^n a_i a_i^T - \sum_{i=1}^n a_i \sum_{j=1}^n b_j^T - (\sum_{i=1}^n a_i \sum_{j=1}^n b_j^T)^T + \sum_{j=1}^n b_j b_j^T)w$$

The preprocessing step is then to compute the $d \times d$ matrix generated by the terms inside the parenthese. Runtime for each term is

- $O(nd^2)$ for $\sum_{i=1}^n a_i a_i^T$ and $\sum_{j=1}^n b_j b_j^T$
- $O(2nd + d^2)$ for $\sum_{i=1}^n a_i \sum_{j=1}^n b_j^T$

To compute
$\mathbf{M} = (\sum_{i=1}^n a_i a_i^T - \sum_{i=1}^n a_i \sum_{j=1}^n b_j^T - (\sum_{i=1}^n a_i \sum_{j=1}^n b_j^T)^T + \sum_{j=1}^n b_j b_j^T), \mathbf{M} \in \mathbf{R}^{d \times d}$ is $O(nd^2)$

Then we can change the formula of $f(w)$

$$f(\mathbf{w}) = \mathbf{w}^T \mathbf{M} \mathbf{w} + \lambda \|\mathbf{w}\|_2^2$$

Multiplying vector of size d with matrix of size $d \times d$, the runtime is $O(d^2)$.