CS221 Autumn 2016 Homework [1]

SUNet ID: [06033811]
Name: [Chao Xu]

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my original work.

Problem 1: Optimization and Probability

(a) Define the quadratic function: $f(x) = \frac{1}{2}\sum_{i=1}^{n} w_i(x - b_i)^2$

Take the derivative of f(x) with respect to x:

$$\frac{\partial f(x)}{\partial x} = \sum_{i=1}^{n} w_i(x - b_i)$$

To minimize the value of f(x), set the derivative of f(x) equals to zero. Thus, we have:

$$\sum_{i=1}^{n} w_i x = \sum_{i=1}^{n} w_i b_i$$

Therefore:

$$\text{x} = \frac{\sum_{i=1}^{n} w_i b_i}{\sum_{i=1}^{n} w_i}$$

This value of x can minimize the value of $f(x)$.

If some w_i's are negative, there could have situations that no x exists can minimize f(x), as $\sum_{i=1}^{n} w_i$ could equal to zero which is invalid for solving x.

(b) $f(x) = max_{a\in\{-1,1\}} \sum_{j=1}^{d} ax_j$ and $g(x) = \sum_{j=1}^{d} max_{a\in\{-1,1\}} ax_j$ where $x = \{x_1, \ldots, x_d\} \in R^d$

f(x) can be expressed as following:

$$f(x) = max_{a\in\{-1,1\}} a \sum_{j=1}^{d} x_j = max_{a\in\{-1,1\}} a(x_1 + \cdots + x_d)$$

g(x) can be expressed as following:

$$g(x) = max_{a\in\{-1,1\}} ax_1 + \cdots + max_{a\in\{-1,1\}} ax_d$$

x is a real vector in the domain of all real numbers, thus it can be positive or negative. g(x) can maximize or take the absolute value of each element in the vector independently using a. On the other hand, f(x) can only maximize the summation of vector x. Therefore, if there is a negative element in the x vector, g(x) will be larger than f(x). If all the elements in the x vector are positive or zero, then g(x) will be equal to f(x). Therefore, the answer is:

$$f(x) \leq g(x)$$

(c) Suppose you repeatedly roll a fair six-sided dice until the number of dots is 2 or fewer (and then you're done). Every time the dice turns up with a 4, 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?

The game ends when the dots is 2 or 1. Therefore, for each dice roll there is $\frac{1}{3}$ probability the game will ends. Therefore, after n^th roll, the probability that the game continues will be $\frac{2}{3}^n$. The expected number of total points (as a function of r) that I will earn after n rolls before I'm done should be:

$$\text{E[X]} = \frac{2}{3}^n \frac{1}{4} r$$

(d) Suppose the probability of a coin turning up heads is 0 <p <1, and that we flip it 5 times and get {H,T,H,T,T}. We know the probability (likelihood) of obtaining this sequence is $L(p) = p(1-p)p(1-p)(1-p) = p^2(1-p)^3$. 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 logL(p). Taking the derivative of L(p) works too, but it is cleaner and more natural to differentiate logL(p). You can verify for yourself that the value of p which minimizes logL(p) must also minimize L(p).

$$logL(p) = log(p^2(1-p)^3)$$

Take the derivative of logL(p) with respect to p and set it equals to 0:

$$\frac{\partial logL(p)}{\partial p} = \partial(log(p^2) + log(1-p)^3)/\partial p = \partial(2log(p) + 3log(1-p))/\partial p$$

$$= 2\frac{1}{pln(10)} + 3\frac{1}{-(1-p)ln(10)} = 0$$

Therefore, p should equal to:

$$p = 0.4$$

(e) Define the scalar-valued function:

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

where the vector is $\mathbf{w} = (w_1, \ldots, w_d)$ and $\|\mathbf{w}\|_2 = \sqrt{\sum_{j=1}^{d} \mathbf{w}_j^2}$ is known as the L_2 norm. Compute the gradient $\nabla f(w)$.

$$\|\mathbf{w}\|_2^2 = \sum_{j=1}^{d} \mathbf{w}_j^2$$

$$\frac{\partial \sum_{j=1}^{d} w_j^2}{\partial w_i} = \sum_{j=1}^{n} \frac{\partial w_j^2}{\partial w_i} = 0 \quad when \quad i \neq j$$

$$\frac{\partial \sum_{j=1}^{d} w_j^2}{\partial w_i} = \sum_{j=1}^{n} \frac{\partial w_j^2}{\partial w_i} = 2w_j \quad when \quad i = j$$

Therefore:

$$\nabla \|\mathbf{w}\|_2^2 = 2\mathbf{w}$$

Then

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

Problem 2: Complexity

(a) 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, two ears, and one mouth, each represented as an arbitrary axis-aligned rectangle (i.e. the axes of the rectangle are aligned with the axes of the image). 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.

The total rectangles with the size, x pixels by y pixels, exist in a $n \times n$ pixels area is:

$$(n - x + 1) \times (n - y + 1)$$

Since the number of pixels on each side can range from 1 to n. Therefore, the total number of rectangles should be :

$$\sum_{x=1}^{n} \sum_{y=1}^{n} (n - x + 1) \times (n - y + 1)$$

The asymptotic complexity for the above equation is $O(n^4)$. Thus, for five different rectangles that can overlap in a $n \times n$ pixels area, the asymptotic complexity should be:

$$O(n^{4\times5}) = O(n^{20})$$

(b) 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: $f(j) = min_{1\leq i<j}[c(i,j) + f(i)]$ for j=1,...,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)?
The algorithm should pick the unvisited cities with the lowest cost, calculates the cost through it to each unvisited neighbor cities, and updates the neighbor city's cost if lower:

$$g(j) = min_{1\leq i<j}[c(i,j) + g(j), g(j)]$$

Remove the city from the list if this city has been selected in the path. Running the algorithm again until it reaches the destination, n. This algorithm would provide the minimum cost to travel from 1 to n.

(c) 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.
First, all the path has $2 \times n$ steps, as you need to go right n times and go down n times as well. Furthermore, for these 2n steps,they are the combination of going right or going down. The problem can treated as the number of combination of n steps arrangement in 2n available steps.

$$\binom{2n}{n} = \frac{2n!}{n!\times n!}$$

(d) 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. Again, you may find it helpful to work out the scalar case first.

Define the scalar-valued function:

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

It can be expressed as:

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

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

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

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

Let $A = \sum_{i=1}^{n} \sum_{j=1}^{n} (a_i a_i^\top - a_i b_j^\top - b_j a_i^\top + b_j b_j^\top)$, therefore, $f(\mathbf{w}) = \mathbf{w}^\top A \mathbf{w} + \lambda \|\mathbf{w}\|_2^2$. To define A, the preprocessing would take $O(nd^2)$. And once matrix A with size $d \times d$ has been define from the above equation, it will take $O(d^2)$ time to compute $f(\mathbf{w})$.