CS221 Fall 2015 Homework [foundations]

Problem 1: Optimization and Probability

1a.

Given: $f(x)=\frac{1}{2}\sum_{1=1}^{n} w_i(x-b_i)^2$

Therefore $f'(x)=\sum_{1=1}^{n} w_i(x-b_i)$

Furthermore

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

The value of x that minimizes $f(x)$ will occur when $f'(x)$ is equal to zero.

This means that

$$x\sum_{1=1}^{n} w_i - \sum_{1=1}^{n} w_i b_i = 0$$

Therefore $x=\frac{\sum_{i=1}^{n} w_i b_i}{\sum_{i=1}^{n} w_i}$ is the answer.

Finally, confirm that $f''(x)$ is greater than zero with $f''(x)=\sum_{1=1}^{n} w_i$ which proves it is a minimum.

1b.

Given

$$f(x)=max_{a\in\{1,-1\}}\sum_{j=1}^{d} ax_j$$

Therefore *f(x)* is only one of two possibilities:

$$f_1(x)=\sum_{j=1}^{d} x_j \quad \text{or} \quad f_2(x)=-\sum_{j=1}^{d} x_j$$

And given

$$g(x)=\sum_{j=1}^{d} max_{a\in\{1,-1\}} ax_j$$

We can state that $max_{a\in\{1,-1\}} ax_j$ is equal to $|x_j|$

Therefore $g(x)=\sum_{j=1}^{d} max_{a\in\{1,-1\}} ax_j$ is equivalent to: $g(x)=\sum_{j=1}^{d} |x_j|$

Thus we know that $g(x) \geq f_1(x)$ because $|x_j| \geq x_j$ for any x_j

Similarly we know that $g(x) \geq f_2(x)$ because $|x_j| \geq -x_j$ for any x_j

Therefore we have proven that $f(x) \leq g(x)$ for all of x.

1c.
First, on a roll event, the “terminating event” probability is ½ or 50% chance that the fair six sided dice will roll a 3, 2 or 1. Conversely, on a single roll, the “continue event” probability, i.e. the probability that the dice will roll a 4, 5, or 6 is also ½ or 50%.

Second, the probability of achieving a reward ***r*** given that a “continue event” has occurred is $\frac{1}{3}$

The expected reward for a continue event is therefore $r\frac{1}{3}$

The probability of terminating on the ***n****th* throw is $\frac{1}{2^n}$ and the expected reward in this case is

$\frac{r}{3}(n-1)$ because the first *n-1* throws are continue events.

Therefore:

$$f(r)=\frac{r}{3}\sum_{n=1}^{\infty}\frac{(n-1)}{2^n}$$

Now consider the infinite series portion: $\sum_{n=1}^{\infty}\frac{(n-1)}{2^n}$

Let's assign S equal to the infinite series: $S=\sum_{n=1}^{\infty}\frac{(n-1)}{2^n}$ (such that $f(r)=\frac{r}{3}S$ holds true).

This results in: $S=0+\frac{1}{4}+\frac{2}{8}+\frac{3}{16}+\frac{4}{32}...$

Then, divide both sides by 2: $\frac{S}{2}=\frac{1}{8}+\frac{2}{16}+\frac{3}{32}+\frac{4}{64}...$

and subtract the two equations $S-\frac{S}{2}=\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}...=\frac{1}{2}$

Thus we determine that $S=1$ therefore:

$f(r)=\frac{r}{3}S$ is equal to $f(r)=\frac{r}{3}$ which is the answer.

1d.
Given: $L(p)=p^3(1-p)^2$
And converting this into
$\log L(p)=3\log p+2\log(1-p)$

The value of p that maximizes *log L(p)* also maximizes *L(p)* because *log* is a monotonic function.

To determine the value of p that maximizes *log L(p)* we must equate the derivative of *log L(p)* to zero. We can do this using the suggested method of taking the derivative of *log L(p)* as follows:

$\frac{d(\log L(p))}{dp}=3\frac{1}{p}+2\frac{1}{(1-p)}*(-1)$ which when set to zero implies:

$\frac{3}{p}-\frac{2}{1-p}=0$ and therefore: $p=\frac{3}{5}$ is the answer.

To ensure this is a maximum, we can check that the second order derivative $-\frac{3}{p^2}-\frac{2}{(1-p)^2}<0$

1e.
Given

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

Rewritten as:

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

The gradient of *f(w)* defined as $\nabla f(w)$ can be determined by taking the derivative of the function *f(w)* with respect to the vector w :

Therefore: $\nabla f(w)=\sum_{i=1}^{n}\sum_{j=1}^{n}2(a_i^T w-b_j^T w) * (a_i - b_j) + 2\lambda w$ is the answer.

Problem 2: Complexity

2a.
Within an $n\times n$ pixel image, each feature represented as an unconstrained arbitrary axis-aligned rectangle can be uniquely specified using 2 diagonally opposite points to define a feature's bounding box. Therefore, asymptotically the number of ways of choosing a single rectangle is:

$$\binom{n^2}{2}$$

Which is:

$$O(n^4)$$

Thus for three features, the complexity is:

$$O(n^4 * n^4 * n^4)$$

As observed, the answer is in the $O(n^c)$ form, where c = 12
Thus the asymptotic complexity is $O(n^{12})$ which is the answer.

2b.
Given the following recurrence: $f(j) = min_{1\le i<j}[c(i,j)+f(i)]$
The algorithm for computing *f(n)* using the above recurrence can be implemented using dynamic programming. The dynamic programming algorithm calculates *f(1), f(2), f(3) … f(n)* in that order. To compute *f(j)* we need to examine all previous terms $c(i,j)+f(i)$ where i is less than j . Therefore *j-1* terms has to be examined for computing *f(j)*. The run-time for computing *f(n)* is therefore $O(\sum_{j=1}^{n}(j-1))$ which is equal to $O(n^2)$ which is the answer.

2c.

For any $n \times n$ grid with moves constrained as specified at each step, the following illustration explains the problem for any number of n :

	n=1	n=2	n=3	n=4	n=5	n=6	... n=d
n=1	0	1	1	1	1	1	
n=2	1	2 $\frac{(2(n-1))!}{(n-1)!\ (n-1)!}$	3	4	5	6	
n=3	1	3	6 $\frac{(2(n-1))!}{(n-1)!\ (n-1)!}$	10	15	21	
n=4	1	4	10	20 $\frac{(2(n-1))!}{(n-1)!\ (n-1)!}$	35	56	
n=5	1	5	15	35	70 $\frac{(2(n-1))!}{(n-1)!\ (n-1)!}$	126	
n=6	1	6	21	56	126	252 $\frac{(2(n-1))!}{(n-1)!\ (n-1)!}$	
⋮ n=d							

The number of ways 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 can be determined by computing the total number of moves from 2(n-1) available moves using the binomial coefficient for number of outcomes as follows:

$$\binom{2(n-1)}{(n-1)} = \frac{(2(n-1))!}{(n-1)!\ (n-1)!}$$

The answer is now given as a function of n:

$$f(n) = \frac{(2(n-1))!}{(n-1)!\ (n-1)!}$$

2d.

In the equation

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

$\lambda\|w\|_2^2$ is computable in O(d) time so we can ignore it for this problem.

Now this $\sum_{i=1}^{n}\sum_{j=1}^{n}(a_i^T w - b_j^T w)^2$ can be refactored as $\sum_{i=1}^{n}\sum_{j=1}^{n}(a_i^T w - b_j^T w)^T(a_i^T w - b_j^T w)$

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

Let A be the $d\times n$ matrix equal to $A=(a_1 a_2 a_3 a_4 \ldots a_n)$ where each of the columns represents the ***a*** vectors. Similarly let B be the $d\times n$ matrix $B=(b_1 b_2 b_3 b_4 \ldots b_n)$ which represents the ***b*** vectors.

Now the strategy is to represent the terms from the summation expression as matrix multiplication, such that: $f(w)=w^T C w$ for some $d\times d$ matrix C. This will make run-time computation of $f(w)$ happen in $O(d^2)$ time.

Strategically speaking, an example AB^T has equivalent terms to the expanded form of $f(w)$. Therefore computing C is equivalent to AB^T in compute time. Computing C can be done in $O(nd^2)$ preprocessing time since A and B are both $d\times n$ matrices that yield a $d\times d$ matrix result similar to C.