XCS221 Assignment 0 — Foundations **(Solutions)**

Guidelines

1. These questions require thought, but do not require long answers. Please be as concise as possible.
2. If you have a question about this homework, we encourage you to post your question on our Slack channel, at `http://xcs221-scpd.slack.com/`
3. Familiarize yourself with the collaboration and honor code policy before starting work.
4. For the coding problems, you may not use any libraries except those defined in the provided started code. In particular, ML-specific libraries such as `scikit-learn` are not permitted.

Submission Instructions

Written Submission: All students must submit an electronic PDF containing solutions to the written questions. As long as the submission is legible and well-organized, the course staff has no preference between a handwritten and a typeset LaTeXsubmission. Students wishing to typeset their documents should follow these recommendations:

- Type responses only in `submission.tex`.
- If you choose to submit a typeset document, please submit the compiled PDF, **not** `submission.tex`.
- Use the commented recommendations within the Makefile and README.md to get started.

Coding Submission: All assignment code is in the `src/` subirectory. You will submit only the `src/submission.py` file. Please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` will be used to autograde your submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
python grader.py
```

There are two types of unit tests used by our autograders:

- `basic`: These unit tests will verify only that your code runs without errors on obvious test cases.
- `hidden`: These unit tests will verify that your code produces correct results on complex inputs and tricky corner cases. In the student version of `src/grader.py`, only the setup and inputs to these unit tests are provided. When you run the autograder locally, these test cases will run, but the results will not be verified by the autograder.

For debugging purposes, a single unit test can be run locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
python grader.py 3a-0-basic
```

Before beginning this course, we highly recommend you walk through our Anaconda tutorial to familiarize yourself with our coding environment. Please use the env defined in `src/environment.yml` to run your code. This is the same environment used by our autograder.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

THIS ASSIGNMENT IS COMPLETELY OPTIONAL. YOU WILL RECEIVE NO CREDIT FOR ANY PART OF THIS ASSIGNMENT.

Introduction

Welcome to your first XCS221 assignment! The goal of this assignment is to sharpen your math and programming skills needed for this class. If you meet the prerequisites, you should find these problems relatively innocuous. Some of these problems will occur again as subproblems of later homeworks, so make sure you know how to do them.

1. **Optimization and Probability**

In this class, we will cast a lot of AI problems as optimization problems, that is, finding the best solution in a rigorous mathematical sense. At the same time, we must be adroit at coping with uncertainty in the world, and for that, we appeal to tools from probability.

(a) **[1 point (Written)] Optimization**

Let $x_1, \dots, x_n$ be real numbers representing positions on a number line. Let $w_1, \dots, w_n$ be positive real numbers representing the importance of each of these positions. Consider the quadratic function: $f(\theta) = \frac{1}{2}\sum_{i=1}^{n} w_i(\theta - x_i)^2$. What value of θ minimizes $f(\theta)$? What problematic issues could arise if some of the w_i's are negative?

Note: You can think about this problem as trying to find the point θ that's not too far away from the x_i's. Over time, hopefully you'll appreciate how nice quadratic functions are to minimize.

A necessary condition for θ to be a minimizer of a differentiable function $f(\theta)$ is that its derivative $\frac{d}{d\theta}f(\theta) = 0$. Thus, let's take the derivative and set it to zero:

$$\frac{d}{d\theta}f(\theta) = \left(\sum_{i=1}^{n} w_i\right)\theta - \sum_{i=1}^{n} w_i x_i = 0$$

We can know solve for θ. Doing some basic algebra yields

$$\theta = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

.

To double check that θ is a true minimizer, we check that the second derivative is positive. The second derivative is $\frac{d^2}{d\theta^2}f(\theta) = \sum_{i=1}^{n} w_i$, which is positive because of the assumption that the w_i's are positive real numbers.

Note that θ is just the weighted average of the x_i's, which is nice and interpretable. If some w_i's are negative, then $f(\theta)$ may not be a convex function (based on the second derivative), so a minimizer for this function may not exist.

(b) **[1 point (Written)] Optimization**

In this class, there will be a lot of sums and maxes. Let's see what happens if we switch the order. Let $f(\mathbf{x}) = \sum_{i=1}^{d} \max_{s \in \{1,-1\}} s x_i$ and $g(\mathbf{x}) = \max_{s \in \{1,-1\}} \sum_{i=1}^{d} s x_i$, where $\mathbf{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$ is a real vector. Does $f(\mathbf{x}) \le g(\mathbf{x})$, $f(\mathbf{x}) = g(\mathbf{x})$, or $f(\mathbf{x}) \ge g(\mathbf{x})$ hold for all $\mathbf{x}$? Please provide a formal proof explaining your answer.

Hint: You may find it helpful to refactor the expressions so that they are maximizing the same quantity over different sized sets.

Intuitively, for $f(\mathbf{x})$, we can select a different a for each i, whereas for $g(\mathbf{x})$, we have to select a single a for all j. More formally, to make $f(\mathbf{x})$ and $g(\mathbf{x})$ look more similar for comparison, let's rewrite them:

$$f(\mathbf{x}) = \max_{s_1,\dots,s_n} \sum_{i=1}^{d} s_j x_i$$

$$g(\mathbf{x}) = \max_{s_1=\dots=s_n} \sum_{i=1}^{d} s_j x_i$$

Now it's obvious that both functions are maximizing over the same quantity, except that $f(\mathbf{x})$ is maximizing over a larger set. Therefore, we have that $f(\mathbf{x}) \ge g(\mathbf{x})$ for all $\mathbf{x}$.

(c) **[1 point (Written)] Probability**

Suppose you repeatedly roll a fair six-sided die until you roll a 1 (and then you stop). Every time you roll a 2, you lose a points, and every time you roll a 6, you win b points. You do not win or lose any points if you roll a 3, 4, or a 5. What is the expected number of points (as a function of a and b) you will have when you stop?

Hint: We recommend defining a recurrence.

Let V be the expected number of points you will earn. We can define the recurrence by considering the possible outcomes of the dice:

$$V = \frac{1}{6}(-a) + \frac{1}{6}b + \frac{5}{6}V$$

Solving for V yields $b - a$. These types of calculations will show up in Markov decision processes, where we consider iterated games of chance.

(d) **[1 point (Written)] Probability**

Suppose the probability of a coin turning up heads is $0 < p < 1$, and that we flip it 7 times and get $\{\mathrm{H}, \mathrm{H}, \mathrm{T}, \mathrm{H}, \mathrm{T}, \mathrm{T}, \mathrm{H}\}$. We know the probability (likelihood) of obtaining this sequence is $L(p) = pp(1-p)p(1-p)(1-p)p = p^4(1-p)^3$. What value of p maximizes $L(p)$?

What is an intuitive interpretation of this value of p?

Hint: Consider taking the derivative of $\log L(p)$. *You can also directly take the derivative of* $L(p)$, *but it is cleaner and more natural to differentiate* $\log L(p)$. *You can verify for yourself that the value of* p *which maximizes* $\log L(p)$ *must also maximize* $L(p)$ *(you are not required to prove this in your solution).*

We have $\log L(p) = 4 \log p + 3 \log(1-p)$. Taking the derivative yields $\nabla \log L(p) = \frac{4}{p} - \frac{3}{1-p}$. Setting this to zero and solving yields $p = \frac{4}{7}$. Checking that the second derivative is negative for $0 < p < 1$:

$$\nabla \log L(p) = -4/p^2 - 3/(1-p)^2 < 0$$

The optimal p has a very intuitive interpretation: it's just the fraction of heads. This is a classic derivation of the maximum likelihood estimator in statistics.

(e) **[1 point (Written)] Linear Algebra**

Let's practice taking gradients, which is a key operation for being able to optimize continuous functions. For $\mathbf{w} \in \mathbb{R}^d$ (represented as a column vector) and constants $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ (also represented as column vectors) and $\lambda \in \mathbb{R}$, define the scalar-valued function

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

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

Recall: the gradient is a d*-dimensional vector of the partial derivatives with respect to each* w_i*:*

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \ldots \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top.$$

If you're not comfortable with vector calculus, first warm up by working out this problem using scalars in place of vectors and derivatives in place of gradients. Not everything for scalars goes through for vectors, but the two should at least be consistent with each other (when $d = 1$*). Do not write out summation over dimensions, because that gets tedious.*

$$\nabla f(\mathbf{w}) = \nabla(\sum_{i=1}^{n} \sum_{j=1}^{n} (\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w})^2) + \nabla(\lambda \|\mathbf{w}\|_2^2)$$

We know $\nabla \|\mathbf{w}\|_2^2 = \nabla(\mathbf{w}^\top \mathbf{w}) = 2\mathbf{w}$. Hence as λ is a constant,

$$\nabla f(\mathbf{w}) = \nabla(\sum_{i=1}^{n} \sum_{j=1}^{n} (\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w})^2) + 2\lambda \mathbf{w}$$

$\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ are constants.

$$\nabla f(\mathbf{w}) = \sum_{i=1}^{n} \sum_{j=1}^{n} \nabla(\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w})^2 + 2\lambda \mathbf{w}$$

Applying the chain rule,

$$\nabla f(\mathbf{w}) = 2\sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w})(\mathbf{a}_i - \mathbf{b}_j) + 2\lambda\mathbf{w}$$

2. **Complexity**

When designing algorithms, it's useful to be able to do quick back of the envelope calculations to see how much time or space an algorithm needs. Hopefully, you'll start to get more intuition for this by being exposed to different types of problems.

(a) **[1 point (Written)] Complexity**

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, one nose, 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.

There are $O(n)$ coordinates, $O(n^2)$ points (each defined by two coordinates), and $O(n^4)$ possible rectangles in the image (each defined by two corner points). Since we need to select 6 rectangles, the total number of faces is $O((n^4)^6) = O(n^{24})$.

(b) **[1 point (Written)] Dynamic Programming**

Suppose we have an $n \times n$ grid. We start in the upper-left corner (position $(1, 1)$), and we would like to reach the lower-right corner (position (n, n)) by taking single steps down and right. Define a function $c(i, j)$ to be the cost of touching position (i, j), and assume it takes constant time to compute. Note that $c(i, j)$ can be negative. Give an algorithm for computing the minimum cost in the most efficient way. What is the runtime (just give the big-O)?

We can compute the minimum cost of doing this by defining the following recurrence: $f(i, j) = c(i, j) + \min(f(i-1, j), f(i, j-1))$ for $i, j = 1, \ldots, n$. The initial state is $f(1, 1) = c(1, 1)$, and our solution will be found as $f(n, n)$. To handle the top and left edges, we can simply define $f(i, j) = \infty$ whenever $i = 0$ or $j = 0$. For each $i, j = 1, \ldots, n$, we need to compute $f(i, j)$. If we loop through i and j in increasing order, we will have already computed $f(i-1, j)$ and $f(i, j-1)$, so we can compute $f(i, j)$ in $O(1)$ time. Thus, the total running time is $O(n^2)$. Note that this is a basic case where we are memoizing the result of $f(j)$; if we didn't do that, then it would take time exponential in n, which is too slow.

(c) **[1 point (Written)] Recursion**

Suppose we have a staircase with n steps (we start on the ground, so we need n total steps to reach the top). We can take as many steps forward at a time, but we will never step backwards. How many ways are there to reach the top?

Give your answer as a function of n. For example: if $n = 3$, then the answer is 4. The four options are the following:

1) take one step, take one step, take one step
2) take two steps, take one step
3) take one step, take two steps
4) take three steps.

We have to take n steps in total, and we just have to decide where to break up our steps into blocks. There are $n-1$ spots where we can break up the steps (one spot in between each of the n steps). For each of these spots, we can either split the step or not, so we have $n-1$ binary choices. There are 2^{n-1} ways to do this. There are exponentially many paths!

(d) **[1 point (Written)] Complexity**

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.

$$f(\mathbf{w}) = \sum_{i=1}^{n} \sum_{j=1}^{n} (\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w})^2 + \lambda \|\mathbf{w}\|_2^2.$$

The second term is λ times square of the L2 norm of $\mathbf{w}$. It takes $O(d)$ to compute. The first term can be written as

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w})^\top(\mathbf{a}_i^\top \mathbf{w} - \mathbf{b}_j^\top \mathbf{w}) &= \sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{w}^\top \mathbf{a}_i \mathbf{a}_i^\top \mathbf{w} + \mathbf{w}^\top \mathbf{b}_j \mathbf{b}_j^\top \mathbf{w} - \mathbf{w}^\top \mathbf{a}_i \mathbf{b}_j^\top \mathbf{w} - \mathbf{w}^\top \mathbf{b}_j \mathbf{a}_i^\top \mathbf{w} \\ &= \mathbf{w}^\top X \mathbf{w}\end{aligned}$$

where $X = \sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{a}_i\mathbf{a}_i^\top + \mathbf{b}_j\mathbf{b}_j^\top - \mathbf{a}_i\mathbf{b}_j^\top - \mathbf{b}_j\mathbf{a}_i^\top)$ and $\mathbf{w} : d \times 1; X : d \times d$.
Computing the first term takes $O(d^2)$ for any given $\mathbf{w}$ if we pre-compute X.

$$\begin{aligned}X &= n(\sum_{i=1}^{n}\mathbf{a}_i\mathbf{a}_i^\top + \sum_{j=1}^{n}\mathbf{b}_j\mathbf{b}_j^\top) - \sum_{i=1}^{n}\sum_{j=1}^{n}\mathbf{a}_i\mathbf{b}_j^\top + \mathbf{b}_j\mathbf{a}_i^\top \\ &= n(\sum_{i=1}^{n}\mathbf{a}_i\mathbf{a}_i^\top + \sum_{j=1}^{n}\mathbf{b}_j\mathbf{b}_j^\top) - ((\sum_{i=1}^{n}\mathbf{a}_i)(\sum_{j=1}^{n}\mathbf{b}_j)^\top + (\sum_{j=1}^{n}\mathbf{b}_j)(\sum_{i=1}^{n}\mathbf{a}_i)^\top)\end{aligned}$$

Hence, it can be seen that it takes $O(nd^2)$ to compute X.

3. **Optimization and Probability**

In this problem, you will implement a bunch of short functions. The main purpose of this exercise is to familiarize yourself with Python, but as a bonus, the functions that you will implement will come in handy in subsequent homeworks.

If you are new to Python, the following provide pointers to various tutorials and examples for the language:

- Python for Programmers
- Example programs of increasing complexity

(a) [**3 points (Coding)**]

Implement `findAlphabeticallyLastWord` in `submission.py`.

(b) [**2 points (Coding)**]

Implement `euclideanDistance` in `submission.py`.

(c) [**3 points (Coding)**]

Implement `mutateSentences` in `submission.py`.

(d) [**2 points (Coding)**]

Implement `sparseVectorDotProduct` in `submission.py`.

(e) [**2 points (Coding)**]

Implement `incrementSparseVector` in `submission.py`.

(f) [**3 points (Coding)**]

Implement `findSingletonWords` in `submission.py`.

(g) [**5 points (Coding)**]

Implement `computeLongestPalindromeLength` in `submission.py`.