HW 0

Kevin Lin

1/12/2026

1

(a) Let $x_1, \ldots, x_n$ be real values. Then for the quadratic function $f(\theta) = \sum_{i=1}^{n} w_i(x_i - \theta)^2$ where $w_i > 0$ for all i, the optimal solution θ^* denoted by $\theta^* = \arg\min_\theta f(\theta)$ can be calculated as follows:

$$\begin{aligned}
\frac{d}{d\theta} f(\theta) &= \frac{d}{d\theta} \sum_{i=1}^{n} w_i(x_i - \theta)^2 \\
&= \sum_{i=1}^{n} w_i \cdot 2(x_i - \theta) \cdot (-1) \\
&= -2 \sum_{i=1}^{n} w_i(x_i - \theta) \\
&= -2 \left(\sum_{i=1}^{n} w_i x_i - \theta \sum_{i=1}^{n} w_i \right)
\end{aligned}$$

Setting the derivative to zero to find the minimum:

$$\begin{aligned}
-2 \left(\sum_{i=1}^{n} w_i x_i - \theta \sum_{i=1}^{n} w_i \right) &= 0 \\
\sum_{i=1}^{n} w_i x_i - \theta \sum_{i=1}^{n} w_i &= 0 \\
\theta \sum_{i=1}^{n} w_i &= \sum_{i=1}^{n} w_i x_i \\
\theta^* &= \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}
\end{aligned}$$

Thus, the optimal solution is the weighted average of the x_i's. If some weights are negative, the function may not be convex, and the solution may not correspond to a minimum.

(b) (i) Given $2n$ kids are randomly divied into two equal subgroups, the probability that the two tallest kids end up in the the same subgroup can be calculated as follows:

$$\begin{aligned}
P(\text{tallest in same group}) &= P(\text{both in group 1}) + P(\text{both in group 2}) \\
&= \frac{\binom{2n-2}{n-2}}{\binom{2n}{n}} + \frac{\binom{2n-2}{n-2}}{\binom{2n}{n}} \\
&= 2 \cdot \frac{\binom{2n-2}{n-2}}{\binom{2n}{n}} \\
&= 2 \cdot \frac{\frac{(2n-2)!}{(n-2)!(n)!}}{\frac{(2n)!}{(n)!(n)!}} \\
&= 2 \cdot \frac{(2n-2)!n!}{(n-2)!(2n)!} \\
&= 2 \cdot \frac{n(n-1)}{(2n)(2n-1)} \\
&= \frac{n(n-1)}{(2n-1)(n)} \\
&= \frac{n-1}{2(2n-1)}
\end{aligned}$$

(ii) The probabilty that the two tallest kids end up in different subgroups is:

$$\begin{aligned}
P(\text{tallest in different groups}) &= 1 - P(\text{both tallest in same group}) \\
&= 1 - \frac{n-1}{2(2n-1)} \\
&= \frac{2(2n-1) - (n-1)}{2(2n-1)} \\
&= \frac{4n - 2 - n + 1}{2(2n-1)} \\
&= \frac{3n-1}{2(2n-1)}
\end{aligned}$$

(c) We know $P(\text{knows answer}) = p$, and $P(\text{doesn't know answer}) = 1 - p$. Also, $P(\text{correct}|\text{knows answer}) = 0.99$ and $P(\text{correct}|\text{doesn't know answer}) = 1/k$. Then $P(\text{knows answer}|\text{correct})$ can be calculated using Bayes' Theorem. Let $P(\text{knows answer}) = P(K)$, $P(\text{doesn't know answer}) =$

$P(DK)$, and $P(\text{correct}) = P(C)$ for simplicity.

$$
\begin{aligned}
P(\text{knows answer}|\text{correct}) = P(K|C) &= \frac{P(C|K)\cdot P(K)}{P(C)} \\
&= \frac{P(C|K)\cdot P(K)}{P(C|K)\cdot P(K) + P(C|DK)\cdot P(DK)} \\
&= \frac{0.99\cdot p}{0.99\cdot p + \frac{1}{k}\cdot(1-p)}
\end{aligned}
$$

(d) Given $L(p) = p^6(1-p)^4$, we can find the value of p that maximizes the likelihood function by taking the derivative and setting it to zero:

$$
\begin{aligned}
\frac{d}{dp}L(p) &= \frac{d}{dp}\left(p^6(1-p)^4\right) \\
&= 6p^5(1-p)^4 + p^6\cdot 4(1-p)^3\cdot(-1) \\
&= p^5(1-p)^3\left(6(1-p) - 4p\right) \\
&= p^5(1-p)^3(6-10p)
\end{aligned}
$$

Setting the derivative to zero:

$$
p^5(1-p)^3(6-10p) = 0
$$

The solutions are $p = 0$, $p = 1$, and $p = \frac{6}{10} = 0.6$. Since p must be in the interval (0, 1), the value of p that maximizes the likelihood function is $p = 0.6$.

(e) Given $F(w) = \sum_{i=1}^n (x_i^T w - y_i)^2 + \lambda \sum_{i=1}^d w_i^2$, $w \in \mathbb{R}^d$, $x_1 \ldots x_n \in \mathbb{R}^d$ column vectors, and y_i scalars, we can find the gradient $\nabla_w F(w)$ by calculating the partial derivatives with respect to each variable w_i:

$$
\begin{aligned}
\frac{\partial}{\partial w_j}F(w) &= \frac{\partial}{\partial w_j}\left(\sum_{i=1}^n (x_i^T w - y_i)^2 + \lambda\sum_{i=1}^d w_i^2\right) \\
&= \sum_{i=1}^n \frac{\partial}{\partial w_j}(x_i^T w - y_i)^2 + \lambda\cdot 2w_j \\
&= \sum_{i=1}^n 2(x_i^T w - y_i)x_i + 2\lambda w_j
\end{aligned}
$$

Thus, the gradient vector is:

$$
\nabla_w F(w) = \begin{bmatrix} \sum_{i=1}^n 2(x_i^T w - y_i)x_i + 2\lambda w_1 \\ \sum_{i=1}^n 2(x_i^T w - y_i)x_i + 2\lambda w_2 \\ \vdots \\ \sum_{i=1}^n 2(x_i^T w - y_i)x_i + 2\lambda w_d \end{bmatrix}
$$

(f) Given the softmax function $f(x_1, \ldots, x_n) = log \sum_{i=1}^{n} e^{x_i}$, we can find the gradient of f with respect to vector x as follows:

$$\begin{aligned}
\frac{\partial}{\partial x_j} f(x_1, \ldots, x_n) &= \frac{\partial}{\partial x_j} \log \sum_{i=1}^{n} e^{x_i} \\
&= \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \frac{\partial}{\partial x_j} \sum_{i=1}^{n} e^{x_i} \\
&= \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot e^{x_j} \\
&= \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}
\end{aligned}$$

Thus, the gradient vector is:

$$\nabla_x f(x_1, \ldots, x_n) = \begin{bmatrix} \frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}} \\ \frac{e^{x_2}}{\sum_{i=1}^{n} e^{x_i}} \\ \vdots \\ \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}} \end{bmatrix}$$

(g) For the previous softmax function, $\max_i x_i \leq f(x_1, \ldots, x_n) \leq \max_i x_i + \log n$. We can show this as follows:

$$f(x_1, \ldots, x_n) = \log \sum_{i=1}^{n} e^{x_i} \geq \log e^{\max_i x_i} = \max_i x_i$$

and

$$f(x_1, \ldots, x_n) = \log \sum_{i=1}^{n} e^{x_i} \leq \log \left(n e^{\max_i x_i}\right) = \log n + \max_i x_i$$

hw0_code

January 10, 2026

1 2

1.1 a

[3]:
```python
import numpy as np

# matrix A, 5 by 4, values 1 to 20 left to right, top to bottom
A = np.arange(1, 21).reshape(5, 4)
print("Matrix A:")
print(A)

# matrix B, 4 by 3, values 1 to 12 top to bottom, left to right
B = np.arange(1, 13).reshape(3, 4).T
print("Matrix B:")
print(B)

def matrix_multiply(A, B):
    return A @ B

print("Matrix Product AB:")
print(matrix_multiply(A, B))
```

```
Matrix A:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]
Matrix B:
[[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]
Matrix Product AB:
[[ 30  70 110]
 [ 70 174 278]
 [110 278 446]
 [150 382 614]
 [190 486 782]]
```

1.2 b

[4]:
```
import scipy.sparse as sp

# worst time complexity of multiplying n x n dense matrix with dense vector R^n
# is O(n^2) because we need to compute n dot products, each taking O(n) time.

# worst time complexity of multiplying n x n sparse matrix with dense vector␣
↪R^n,
# where the sparse matrix has nnz(A) non-zero entries, is O(nnz(A)) because we
# only need to compute dot products for the non-zero entries.

# Create sparse matrix A of size (n-1) x n such that for any x in R^n,
# Ax gives the vector of differences between consecutive elements of x.
def p2b(n):
    row_indices = np.arange(n - 1)
    col_indices = np.arange(n - 1)
    data = np.ones(n - 1)

    row_indices = np.concatenate([row_indices, np.arange(n - 1)])
    col_indices = np.concatenate([col_indices, np.arange(1, n)])
    data = np.concatenate([data, -data])

    A = sp.coo_matrix((data, (row_indices, col_indices)), shape=(n - 1, n))
    return A.toarray()

n = 5
A_sparse = p2b(n)
print(f"Sparse matrix A of size ({n-1}, {n}):")
print(A_sparse)

# random vector x of integers in R^n
x = np.random.randint(1, 10, size=n)
print("Vector x:")
print(x)
print("Ax:")
print(A_sparse @ x)
```

```
Sparse matrix A of size (4, 5):
[[ 1. -1.  0.  0.  0.]
 [ 0.  1. -1.  0.  0.]
 [ 0.  0.  1. -1.  0.]
 [ 0.  0.  0.  1. -1.]]
Vector x:
[9 5 7 1 8]
Ax:
[ 4. -2.  6. -7.]
```

1.3 c

[5]:
```python
# read in text from 'data_example.txt' line by line

with open('data_example.txt', 'r') as f:
    lines = f.readlines()

# naive unique word count by just splitting on spaces and keeping track of
# seen words and their count using a dictionary

words = {}
for line in lines:
    for word in line.strip().split():
        words[word] = words.get(word, 0) + 1
print(f"Naive unique word count: {len(words)}")

# the time complexity of the naive approach is O(n), where n is the total number
# of words in the text, because we need to process each word once.
# the space complexity is O(2m), where m is the number of unique words,␣
↪multiplied
# by 2 because for each unique word, we store the word itself and its count.
```

```
Naive unique word count: 1394
```

1.4 d

[6]:
```python
# naive Fibonacci
def fib_naive(n):
    if n <= 1:
        return n
    return fib_naive(n - 1) + fib_naive(n - 2)

# recursive Fibonacci w/ bookkeeping using a global dictionary
fib_dict = {0: 0, 1: 1}

def fib(n):
    if n not in fib_dict:
        fib_dict[n] = fib(n - 1) + fib(n - 2)
    return fib_dict[n]

# time complexity of book keeping Fibonacci is O(n) because each Fibonacci␣
↪number
# from 0 to n is computed only once and stored in the dictionary.

print("Naive Fibonacci of 10:", fib_naive(10))
print("Bookkeeping Fibonacci of 10:", fib(10))
```

```
Naive Fibonacci of 10: 55
Bookkeeping Fibonacci of 10: 55
```