NYU CS-GY 6923: Machine Learning Fall 2024

Homework 3

Name: Rohan Mahesh Gore (N19332535)

Problem 1: Convexity Warm-up

Collaborators: None.

a. Proof of Convexity for Sum of Convex Functions

We know that a function L(β) is convex if for any two points β_1 and β_2, and for any $\lambda \in [0, 1]$, if the following inequality holds:

$$L(\lambda.\beta_1 + (1-\lambda)\beta_2) \leq \lambda.L(\beta_1) + (1-\lambda).L(\beta_2)$$

We are given that f(β) and g(β) is convex. Therefore, to prove that f(β) + g(β) is convex let's consider:

any two points β_1 and β_2, and any $\lambda \in [0, 1]$.

For f(β), we have:

$$f(\lambda\beta_1 + (1-\lambda)\beta_2) \leq (\beta_1) + (1-\lambda)f(\beta_2)$$

For g(β), we have:

$$g(\lambda\beta_1 + (1-\lambda)\beta_2) \leq (\beta_1) + (1-\lambda)g(\beta_2)$$

Addings above inequalities:

$$f(\lambda\beta_1 + (1-\lambda)\beta_2) + g(\lambda\beta_1 + (1-\lambda)\beta_2) \leq (\beta_1) + (1-\lambda)f(\beta_2) + (\beta_1) + (1-\lambda)g(\beta_2)$$

The left side of this inequality is (f + g)($\lambda\beta_1$ + (1-λ)β_2).

The right side can be rearranged as: $\lambda(f(\beta_1) + g(\beta_1)) + (1-\lambda)(f(\beta_2) + g(\beta_2))$

which becomes: $\lambda(f+g)(\beta_1) + (1-\lambda)(f+g)(\beta_2)$

Therefore, we have shown that:

$$(f+g)(\lambda\beta_1 + (1-\lambda)\beta_2) \leq \lambda(f+g)(\beta_1) + (1-\lambda)(f+g)(\beta_2)$$

This inequality satisfies the definition of convexity for the function (f + g)(β).

Thus, we have proved that the sum of two convex functions is convex.

Hence, proved.

b. Proof of Convexity for L2 and L1 Regularization

L2 regularization term is given by:

$$f(\boldsymbol{\beta}) = \lambda \|\boldsymbol{\beta}\|_2^2 = \lambda \sum_{i=1}^{n} \beta_i^2$$

- The function $g(\beta_i) = \beta_i^2$ is convex because its second derivative $g''(\beta_i) = 2$ is positive for all β_i, indicating that the function is convex.
- Since $\lambda > 0$, multiplying a convex function by a positive scalar preserves convexity. Also, we know that sum of convex functions is convex. Therefore, $\lambda \|\boldsymbol{\beta}\|_2^2$ is convex.

L1 regularization term is given by:

$$h(\boldsymbol{\beta}) = \lambda \|\boldsymbol{\beta}\|_1 = \lambda \sum_{i=1}^{n} |\beta_i|$$

- The function $g(\beta_i) = |\beta_i|$ is convex. This can be shown using the definition of convexity or by noting that its sub-gradient exists everywhere which makes it a convex function.
- Again, since $\lambda > 0$, multiplying a convex function by a positive scalar preserves convexity. Therefore, $\lambda \|\boldsymbol{\beta}\|_1$ is convex.

Conclusion: Given that both $\lambda \|\boldsymbol{\beta}\|_2^2$ and $\lambda \|\boldsymbol{\beta}\|_1$ are convex, we can conclude that the regularized regression objectives, which combine these terms with the loss function, are also convex.

Specifically, if the loss function is least squares loss, given as $\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$, which is convex, then from problem 1, part (a) we know that adding either of these regularization terms would overall result in a convex objective function.

Thus, both L2 and L1 regularized regression objectives are convex, allowing for the use of optimization techniques like gradient descent to find their global minimum.

Problem 2: Complexity of Hypothesis Classes

Collaborators: None.

a. Sample Complexity for PAC-Learning Two Hypothesis Classes

We aim to determine tight upper bounds on the number of training examples required to PAC-learn the following functions with accuracy ϵ and success probability $1 - \delta$.

Part 1: Decision Lists

Step 1: Hypothesis Complexity

The number of decision lists grows with d, as y_i can be constructed from any of d variables or their negations. This results in:
- $O(2^d)$ possible y_i conditions,
- $O(2^{2^d})$ possible decision lists (a finite but extremely large number).

Step 2: VC Dimension

The VC dimension of decision lists is $O(d)$. Intuitively:
- The number of linearly independent conditions is bounded by d, as the y_i-terms depend only on d variables.

Step 3: PAC-Sample Complexity

- The PAC-sample complexity for learning a hypothesis class with VC dimension $\text{VC}(H)$ is:

$$m \geq \frac{\text{VC}(H)}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}.$$

- Substituting $\text{VC}(H) = O(d)$, the final sample complexity that we get is:

$$m = O\left(\frac{d}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}\right).$$

Part 2: Binary Linear Threshold Functions

Step 1: Hypothesis Complexity

The number of binary linear threshold functions depends on:
1. The number of binary weight vectors w, which is 2^d.
2. The placement of λ, which divides the input space into regions based on the weighted sum.

Step 2: VC Dimension

The VC dimension of binary linear threshold functions is $d + 1$ which means that the model can shatter $d + 1$ points in general position, but no more.

Step 3: PAC-Sample Complexity

Using the PAC-sample complexity formula:

$$m \geq \frac{\mathrm{VC}(H)}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}.$$

Substituting $\mathrm{VC}(H) = d + 1$, the final sample complexity is:

$$m = O\left(\frac{(d+1)}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}\right).$$

Therefore from Part 1 and Part 2 we have final sample complexities as

1. Decision Lists:

$$m = O\left(\frac{d}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}\right).$$

2. Binary Linear Threshold Functions:

$$m = O\left(\frac{(d+1)}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}\right).$$

b. Tight Upper Bound for Model Selection

- We know that PAC-sample complexity for a single hypothesis class H_i can be given as:

$$m_i = O\left(\frac{\mathrm{VC}(H_i)}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta}\right),$$

  where:
  - $\mathrm{VC}(H_i)$ is the VC dimension of H_i.
  - ϵ is the target error.
  - δ is the failure probability.

- When multiple hypothesis classes are considered:

  1. We need to account for the union bound over q hypothesis classes to ensure success probability $1 - \delta$.

  2. The failure probability for each class is reduced to δ/q.

  Therefore, the sample complexity for q hypothesis classes becomes:

$$m = \max_i O\left(\frac{\mathrm{VC}(H_i)}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{q}{\delta}\right).$$

- To find the best hypothesis h^*, the number of samples required is determined by the hypothesis class with the largest VC dimension, $\mathrm{VC}_{\max} = \max_i \mathrm{VC}(H_i)$.

  Thus the sample complexity required for model selection across q hypothesis classes is:

$$m = O\left(\frac{\mathrm{VC}_{\max}}{\epsilon} \log \frac{1}{\epsilon} + \frac{1}{\epsilon} \log \frac{q}{\delta}\right),$$

  where $\mathrm{VC}_{\max}$ is the maximum VC dimension of the hypothesis classes.

Problem 3: Kernels of Shifted Images

Collaborators: None.

a. Kernel Matrix Calculation Using the Gaussian Kernel

We need to compute the 4×4 kernel matrix K for images I_1, I_2, I_3, I_4 using the Gaussian kernel.

The Gaussian kernel is defined as:

$$k_G(I_i, I_j) = e^{-\|x_i - x_j\|_2^2},$$

where x_i and x_j are the vectorized representations of images I_i and I_j.

Each given image is a 5×5 matrix, which has been vectorized into a 25-dimensional binary vector $x_i \in \{0, 1\}^{25}$.

Step 1: Compute Pairwise Squared Euclidean Distances

The squared Euclidean distance between two vectorized images x_i and x_j is:

$$\|x_i - x_j\|_2^2 = \sum_{k=1}^{25} (x_i^{(k)} - x_j^{(k)})^2.$$

For example:

- $\|x_1 - x_1\|_2^2 = 0$ (self-similarity).

- $\|x_1 - x_2\|_2^2 =$ sum of squared differences of corresponding entries in x_1 and x_2.

Step 2: Apply the Gaussian Kernel

After calculating $\|x_i - x_j\|_2^2$ for all i, j, we apply the Gaussian kernel:

$$k_G(I_i, I_j) = e^{-\|x_i - x_j\|_2^2}.$$

The resulting values form the entries of the kernel matrix K.

Kernel Matrix K using Python The 4×4 kernel matrix K, computed using the Gaussian kernel, is as follows:

```
[[1.00000000e+00 9.11881966e-04 3.35462628e-04 6.73794700e-03]
 [9.11881966e-04 1.00000000e+00 6.73794700e-03 2.47875218e-03]
 [3.35462628e-04 6.73794700e-03 1.00000000e+00 1.67017008e-05]
 [6.73794700e-03 2.47875218e-03 1.67017008e-05 1.00000000e+00]]
```

This can be written as

$$K = \begin{bmatrix} 1.0000 & 0.0009 & 0.0003 & 0.0067 \\ 0.0009 & 1.0000 & 0.0067 & 0.0025 \\ 0.0003 & 0.0067 & 1.0000 & 0.00002 \\ 0.0067 & 0.0025 & 0.00002 & 1.0000 \end{bmatrix}.$$

Kernel Entries Explanation

1. Diagonal entries: $K[i,i] = 1$, as the Gaussian kernel applied to an image with itself is always 1.

2. Off-diagonal entries : Represent the similarity between different images:

- $K[1,2] = 0.0009$: I_1 and I_2 are very dissimilar.

- $K[3,4] = 0.00002$: I_3 and I_4 are extremely dissimilar.

This matrix provides the similarity scores between all pairs of images under the Gaussian kernel.

b. Training and Testing Similarities

Based on the kernel matrix from question 3, part (a) we have the following Training-Test similarities:
- $k_G(I_3, I_1) = 0.0003$, $k_G(I_3, I_2) = 0.0067$.
- $k_G(I_4, I_1) = 0.0067$, $k_G(I_4, I_2) = 0.0025$.

We can identify the most similar training images as compared to the given test images as:

1. For I_3 most similar training image is I_2 as we have $k_G(I_3, I_2) = 0.0067$.

2. For I_4 most similar training image is I_1 as we have $k_G(I_4, I_1) = 0.0067$

- As can be clearly seen I_3 and I_2 are predicted to be similar images but they are not (Zero and One respectively).

  Same situation is with I_4 and I_1 as they belong to different classes (One and Zero respectively).

- The Gaussian kernel only accounts for raw pixel values and does not handle shifts well.

- The above statement is verified by the incorrect predictions and therefore we can conclude that in general the Gaussian kernel may struggle with images affected by shifts.

c. Kernel Matrix Using the Shift Kernel

The shift kernel $k_{\text{shift}}(I_i, I_j)$ is defined as:

$$k_{\text{shift}}(I_i, I_j) = k_G(I_i^{\text{right}}, I_j^{\text{right}}) + k_G(I_i^{\text{left}}, I_j^{\text{left}}) + k_G(I_i^{\text{right}}, I_j^{\text{left}}) + k_G(I_i^{\text{left}}, I_j^{\text{right}}),$$

where:
- I_i^{right}: Image I_i with its far-right column removed.
- I_i^{left}: Image I_i with its far-left column removed.

This can be computed in following 2 steps:

Step 1: Create Shifted Images
We construct the "right-shifted" and "left-shifted" versions of each image.

Step 2: Compute k_{shift}
We compute k_{shift} for all image pairs using the given formula, summing the Gaussian kernel similarities between the shifted images.

The above steps were executed via a python program and the output kernel matrix was:

```
array([[2.00067093, 0.01529966, 1.00798529, 0.01440448],
       [0.01529966, 2.0049575 , 0.14332057, 1.00743626],
       [1.00798529, 0.14332057, 2.01347589, 0.00742557],
       [0.01440448, 1.00743626, 0.00742557, 2.0049575 ]])
```

Therefore the 4×4 kernel matrix computed using the shift kernel k_{shift} is as follows the which can be written as

$$K_{\text{shift}} = \begin{bmatrix} 2.0007 & 0.0153 & 1.0080 & 0.0144 \\ 0.0153 & 2.0050 & 0.1433 & 1.0074 \\ 1.0080 & 0.1433 & 2.0135 & 0.0074 \\ 0.0144 & 1.0074 & 0.0074 & 2.0050 \end{bmatrix}.$$

Just an observation: This matrix reflects similarity more effectively than the Gaussian kernel by accounting for left-right shifts.

d. Training and Testing Similarities Using the Shift Kernel

Based on the kernel matrix from question 3, part (c) we have the following Training-Test similarities:
- $k_{\text{shift}}(I_3, I_1) = 1.0080$, $k_{\text{shift}}(I_3, I_2) = 0.1433$.
- $k_{\text{shift}}(I_4, I_1) = 0.0144$, $k_{\text{shift}}(I_4, I_2) = 1.0074$.

We can identify the most similar training images as compared to the given test images as:

1. For I_3 most similar training image is I_1 as we have $k_{\text{shift}}(I_3, I_1) = 1.0080$

2. For I_4 most similar training image is I_2 as we have $k_{\text{shift}}(I_4, I_2) = 1.0074$

- As we can see that unlike Gussian kernel, the Shift kernel has correctly classified the images according to their classes.

- The shift kernel accounts for left-right shifts, leading to higher similarity scores for I_3 with I_1 and I_4 with I_2.

- A kernel classifier like 1-nearest neighbor assigns the label of the most similar training image to each test image.

- This kernel is more robust for classification in cases where test images are shifted versions of training images.

e. Proving k_{shift} is Positive Semi-Definite

We know that a kernel $k(x, y)$ is positive semi-definite (PSD) if for any set of points $\{x_1, x_2, \ldots, x_n\}$, the kernel matrix K with entries $K_{ij} = k(x_i, x_j)$ satisfies:

$$z^\top K z \geq 0 \quad \text{for all } z \in \mathbb{R}^n.$$

Step 1: Decomposition of k_{shift}:

The shift kernel $k_{\text{shift}}(I_i, I_j)$ is defined as:

$$k_{\text{shift}}(I_i, I_j) = k_G(I_i^{\text{right}}, I_j^{\text{right}}) + k_G(I_i^{\text{left}}, I_j^{\text{left}}) + k_G(I_i^{\text{right}}, I_j^{\text{left}}) + k_G(I_i^{\text{left}}, I_j^{\text{right}}).$$

- Each term $k_G(I_a, I_b)$ is a Gaussian kernel, which is known to be PSD.

- **A sum of PSD kernels is also PSD**.

Step 2: Proving PSD Property for k_{shift}:

Let $K^{\text{right-right}}, K^{\text{left-left}}, K^{\text{right-left}}, K^{\text{left-right}}$ be the kernel matrices corresponding to each term in k_{shift}. Then:

$$K_{\text{shift}} = K^{\text{right-right}} + K^{\text{left-left}} + K^{\text{right-left}} + K^{\text{left-right}}.$$

- Since each K^{term} is PSD (as it comes from k_G), their sum K_{shift} is also PSD.

Therfore the shift kernel k_{shift} is positive semi-definite, as it is a sum of PSD Gaussian kernels.

Problem 4: Steepest Descent

Collaborators: None.

a. Steepest search direction for L1 norm constraint

We aim to solve the maximization problem:

$$\max_{\|v\|_1=1} \langle \nabla L(\beta), v \rangle$$

where $\|v\|_1 = \sum_{i=1}^{n} |v_i|$.

This can be rewritten as:

$$\max_{v} \sum_{i=1}^{d} v_i \cdot \nabla L(\beta) \quad \text{subject to} \quad \|v\|_1 = 1$$

- The ℓ_1-norm constraint $\|v\|_1 = 1$ implies v_i can be positive or negative but the absolute sum of all components equals 1.
- To maximize the simplified equation, each v_i should align with the sign of $\nabla L(\beta)_i$. Specifically:
  - If $\nabla L(\beta)_i > 0$, choose $v_i = 1$ (positive direction).
  - If $\nabla L(\beta)_i < 0$, choose $v_i = -1$ (negative direction).

For $\|v\|_1 = 1$, the optimal v focuses the entire "weight" (value of 1) on the component of $\nabla L(\beta)$ with the largest magnitude, as this maximizes the dot product which therfore give us:

$$v_i^* = \begin{cases} \text{sign}(\nabla L(\beta)_i) & \text{if } i = \arg\max_j |\nabla L(\beta)_j| \\ 0 & \text{otherwise.} \end{cases}$$

- v will be a sparse vector with a single nonzero entry corresponding to the largest magnitude component of g.
- The nonzero entry is assigned a value of 1 (or -1) depending on the sign of the corresponding gradient component. The coordinate with the maximum absolute gradient determines the steepest ascent or descent direction under this norm.

Therefore, the optimal search direction is:

$$v^* = \text{sign}(\nabla L(\beta)_{j^*}) e_{j^*},$$

where, $j^* = \arg\max_j |\nabla L(\beta)_j|$, and e_{j^*} is the unit vector in the j^*-th direction.

b. Steepest search direction for L∞ norm constraint

We aim to solve the maximization problem:

$$\max_{\|v\|_\infty=1} \langle \nabla L(\beta), v \rangle,$$

where $\|v\|_\infty = \max_i |v_i|$ (the infinity norm).

This can be re-written as:

$$\max_v \sum_{i=1}^{d} v_i \cdot \nabla L(\beta) \quad \text{subject to} \quad |v_i| \leq 1 \text{ for all } i.$$

- The ℓ_∞-norm constraint allows v_i to take any value between -1 and 1 for all i.
- To maximize $\langle \nabla L(\beta), v \rangle$, v_i should match the sign of $\nabla L(\beta)_i$, i.e., $v_i = \text{sign}(\nabla L(\beta)_i)$.

To maximize the contribution of each $g_i \cdot v_i$, v_i should take the same sign as g_i. Therefore:

$$v_i = \begin{cases} 1 & \text{if } g_i > 0 \\ -1 & \text{if } g_i < 0. \end{cases}$$

This ensures that each $\nabla L(\beta) \cdot v_i$ is maximized.

- Since $|v_i| \leq 1$, the maximum contribution of each $\nabla L(\beta)_i v_i$ to the dot product occurs when v_i is either 1 (if $\nabla L(\beta)_i > 0$) or -1 (if $\nabla L(\beta)_i < 0$).

Therefore, the optimal choice for v_i is:

$$v_i^* = \text{sign}(\nabla L(\beta)_i).$$

Here, the ℓ_∞-norm does not limit the number of nonzero components in v. as we saw in the case of ℓ_1-norm. Instead, all components v_i can independently take their maximum possible values (± 1) depending on the sign of the gradient at each coordinate.

Therefore, the optimal search direction is:

$$v^* = \text{sign}(\nabla L(\beta)).$$

which maximizes the alignment with the gradient under the ℓ_∞-norm constraint.

c. Prove the Objective Decreases with Steepest Descent

We aim to prove that the update rule

$$\beta \leftarrow \beta - \eta v^*,$$

where v^* is either of the steepest descent directions derived in parts (a) and (b), decreases the objective value $L(\beta)$ as the step size $\eta \to 0$. Specifically, we need to show:

$$\lim_{\eta \to 0} \frac{L(\beta - \eta v^*) - L(\beta)}{\eta} < 0.$$

where:
- β is the current parameter vector.
- v^* is the steepest descent direction derived in parts (a) or (b) (ℓ_1- or ℓ_∞-norm based)
- η is the step size.

This is equivalent to showing that for small η, $L(\beta - \eta v^*) < L(\beta)$, meaning the loss function decreases.

We can prove this in a few steps as belows:

Step 1: Taylor Expansion of $L(\beta)$

- The objective $L(\beta)$ can be approximated using a first-order Taylor expansion around β:

$$L(\beta - \eta v^*) \approx L(\beta) - \eta \langle \nabla L(\beta), v^* \rangle.$$

- For small η that is $\eta \to 0$, the higher-order terms (quadratic and beyond) become negligible, leaving:

$$L(\beta - \eta v^*) - L(\beta) \approx -\eta \langle \nabla L(\beta), v^* \rangle.$$

Step 2: Compute the Limit

- Divide the change in $L(\beta)$ by η:

$$\frac{L(\beta - \eta v^*) - L(\beta)}{\eta} \approx -\langle \nabla L(\beta), v^* \rangle.$$

- Incorporating limits:

$$\lim_{\eta \to 0} \frac{L(\beta - \eta v^*) - L(\beta)}{\eta} = -\langle \nabla L(\beta), v^* \rangle < 0.$$

- The sign of $\langle \nabla L(\beta), v^* \rangle$ determines whether the objective decreases:

- By construction of v^*, we maximize $\langle \nabla L(\beta), v^* \rangle$ under the chosen norm constraint ($\|v^*\|_1 = 1$ or $\|v^*\|_\infty = 1$).

- Thus, $\langle \nabla L(\beta), v^* \rangle > 0$, ensuring:

$$-\langle \nabla L(\beta), v^* \rangle < 0.$$

Justifications for ℓ_1 and ℓ_∞ norms

1. For the ℓ_1-norm:

- v^* focuses on the coordinate $j^* = \arg\max_j |\nabla L(\beta)_j|$, aligning with the steepest descent direction.
- This guarantees $\langle \nabla L(\beta), v^* \rangle$ is maximized under the ℓ_1-norm constraint, and the objective decreases.

2. For the ℓ_∞-norm:

   - $v^* = \text{sign}(\nabla L(\beta))$, which aligns fully with the gradient direction component-wise.
   - Again, $\langle \nabla L(\beta), v^* \rangle$ is maximized under the ℓ_∞-norm constraint, ensuring the objective decreases.

Therefore, for small η, the steepest descent update $\beta \leftarrow \beta - \eta v^*$ always decreases the loss function $L(\beta)$. This holds for both ℓ_1- and ℓ_∞-norm-based steepest descent directions.

Problem 5: Randomized Coordinate Descent

Collaborators: None.

a. Expectation of Parameter Updates in Randomized Coordinate Descent (RCD)

From looking at the pseudocode we can tell a few points about the update rule of RCD:

- At each iteration i, RCD updates only one coordinate β_j, chosen uniformly at random, based on the gradient:
$$\beta_j \leftarrow \beta_j - \eta \cdot g_j,$$
where $g_j = \nabla L_j(\beta)$ is the partial derivative of $L(\beta)$ wrt β_j.
- All other coordinates remain unchanged. Therefore, for $k \neq j$, β_k remains the same. Therefore we will have $\beta_k^{(i)} = \beta_k^{(i-1)}$.

We can prove this is a few steps as below:

Step 1: Formulate the Update in Vector Form

The update for β can be written as:

$$\beta^{(i)} = \beta^{(i-1)} - \eta g_j e_j,$$

where: - e_j is the unit vector with a 1 in the j-th position and 0 elsewhere.

For the change in β:

$$\beta^{(i)} - \beta^{(i-1)} = -\eta g_j e_j.$$

Step 2: Taking Expectation

Since j is chosen uniformly at random from $\{1, 2, \ldots, d\}$, the probability of selecting each coordinate is $\frac{1}{d}$.

Expectation of $\beta^{(i)} - \beta^{(i-1)}$:

$$\mathbb{E}[\beta^{(i)} - \beta^{(i-1)}] = \mathbb{E}[-\eta g_j e_j].$$

By linearity of expectation:

$$\mathbb{E}[\beta^{(i)} - \beta^{(i-1)}] = -\eta \cdot \mathbb{E}[g_j e_j].$$

Step 3: Expanding $\mathbb{E}[g_j e_j]$ and Re-substituting:

For any coordinate j:

$$\mathbb{E}[g_j e_j] = \frac{1}{d} \sum_{j=1}^{d} g_j e_j.$$

Since e_j is the unit vector, this is equivalent to:

$$\mathbb{E}[g_j e_j] = \frac{1}{d} \cdot \nabla L(\beta).$$

Substituting back in the previous equation, we get:

$$\mathbb{E}[\beta^{(i)} - \beta^{(i-1)}] = -\eta \cdot \frac{1}{d} \nabla L(\beta^{(i-1)}).$$

Step 4: Represent in the terms of $-c\nabla L(\beta)$:

$$\mathbb{E}[\beta^{(i)} - \beta^{(i-1)}] = -c\nabla L(\beta^{(i-1)}), \quad \text{where } c = \frac{\eta}{d}.$$

where,

- $c = \frac{\eta}{d}$ is a +ve scalar value
- $\eta > 0$ (learning rate)
- d is the dimension of β:

Hence Proved.

b. Proving Objective Decrease of the function

Step 1: Taylor Expansion of $L(\beta^{(i)})$

- By using a first-order Taylor expansion around $\beta^{(i-1)}$ we get,

  $$L(\beta^{(i)}) \approx L(\beta^{(i-1)}) + \langle \nabla L(\beta^{(i-1)}), \beta^{(i)} - \beta^{(i-1)} \rangle + \frac{1}{2}(\beta^{(i)} - \beta^{(i-1)})^\top H (\beta^{(i)} - \beta^{(i-1)}),$$

  where,
  - H is the Hessian (matrix of second derivatives of $L(\beta)$).

- Subtract $L(\beta^{(i-1)})$:

  $$L(\beta^{(i)}) - L(\beta^{(i-1)}) \approx \langle \nabla L(\beta^{(i-1)}), \beta^{(i)} - \beta^{(i-1)} \rangle + \frac{1}{2}(\beta^{(i)} - \beta^{(i-1)})^\top H (\beta^{(i)} - \beta^{(i-1)}).$$

Step 2: Substitute the update rule for RCD

- The update rule for RCD is:

  $$\beta^{(i)} = \beta^{(i-1)} - \eta g_j e_j,$$

  where:
  - $g_j = \nabla_j L(\beta^{(i-1)})$
  - e_j is the unit vector in the j-th direction.

- Change in β is:
$$\beta^{(i)} - \beta^{(i-1)} = -\eta g_j e_j.$$

- Re-substituting it into the objective difference:
$$L(\beta^{(i)}) - L(\beta^{(i-1)}) \approx -\eta \langle \nabla L(\beta^{(i-1)}), g_j e_j \rangle + \frac{\eta^2}{2} g_j^2.$$

Step 3: Simplify

- Simplifying first term:
Since $g_j = \nabla_j L(\beta^{(i-1)})$, the inner product reduces to:
$$\langle \nabla L(\beta^{(i-1)}), g_j e_j \rangle = g_j^2.$$
making the complete first term as:
$$-\eta g_j^2.$$

- Therefore we have,
$$L(\beta^{(i)}) - L(\beta^{(i-1)}) \approx -\eta g_j^2 + \frac{\eta^2}{2} g_j^2.$$

Step 4: Taking Expectation

- The coordinate j is chosen uniformly at random, so the probability of selecting each coordinate is $\frac{1}{d}$. Taking the expectation over j we have:
$$\mathbb{E}[L(\beta^{(i)}) - L(\beta^{(i-1)})] = \mathbb{E}\left[-\eta g_j^2 + \frac{\eta^2}{2} g_j^2\right].$$

- Therefore, by linearity of expectation:
$$\mathbb{E}[L(\beta^{(i)}) - L(\beta^{(i-1)})] = -\frac{\eta}{d} \|\nabla L(\beta^{(i-1)})\|_2^2 + \frac{\eta^2}{2d} \|\nabla L(\beta^{(i-1)})\|_2^2.$$

But we knw that for sufficiently small η, the η^2 term becomes negligible compared to the η term. Therefore:
$$\mathbb{E}[L(\beta^{(i)}) - L(\beta^{(i-1)})] \approx -\frac{\eta}{d} \|\nabla L(\beta^{(i-1)})\|_2^2.$$

Since $\|\nabla L(\beta^{(i-1)})\|_2^2 \geq 0$, this shows: $\mathbb{E}[L(\beta^{(i)}) - L(\beta^{(i-1)})] \leq 0$.

Therfore for sufficiently small η:
$$\lim_{\eta \to 0} \mathbb{E}[L(\beta^{(i)}) - L(\beta^{(i-1)})] \leq 0.$$

which proves that RCD decreases the objective value in expectation.

Hence Proved.

c. Proving that each iteration of RCD for Least Squares Regression can be implemented in $O(n)$

- The gradient of the least squares loss is:

$$\nabla L(\beta) = 2X^\top(X\beta - y).$$

- For implementing a single iteration of randomized coordinate descent (RCD) in $O(n)$ time, we need to leverage precomputed values.
- AS taught in the lecture, we know that one of the most expensive and recurrent calculation in the gradient is the dot-product of $X\beta$.
- Therefore, we will precompute $X\beta$ at the start of RCD as,

  $z = X\beta$, where $z \in \mathbb{R}^n$

  which costs $O(nd)$ for initialization/first iteration.
- For upcoming iterations we can update the stored z iteratively instead of recomputing it from scratch.

After computing z, the further loop can be implemented with a few modifications (for z) and will also include updating z at the end of the loop. These minor changes would look as:

- Gradient Calculation: $\nabla L_j(\beta) = 2 \cdot X_j^\top(z - y)$
- Update β_j like: $\beta_j \leftarrow \beta_j - \eta \cdot \nabla_j(\beta)$.
- Update z: $z \leftarrow z - \Delta\beta_j \cdot X_j$,

A Python Psuedocode for the same is given below:

```
# Inputs: X (n x d), y (n,), beta (d,), eta (learning rate), num_iterations (T)

z = X @ beta  # Precompute X @ beta in O(nd)

for i in range(num_iterations):
    j = np.random.randint(d)             # Select random coordinate
    grad_j = 2 * X[:, j].T @ (z - y)     # Compute partial gradient in O(n)
    delta_beta_j = eta * grad_j          # Compute update
    beta[j] -= delta_beta_j              # Update beta
    z -= delta_beta_j * X[:, j]          # Update z in O(n)
```

Time Complexity Analysis for the above pseudocode:

- Precomputation of $z = X\beta$: $O(nd)$ (one-time cost).
- Gradient Computation $\nabla_j L(\beta)$: $O(n)$ per iteration.

- Update z: $O(n)$ per iteration.
- Total Cost Per Iteration: $O(n)$.

AS the total cost per iteration achieved is $O(n)$, we can conclude that each iteration of RCD can be implemented in $O(n)$ time after an $O(nd)$ initialization by reusing precomputed values for $z = X\beta$ and updating only the necessary components iteratively.