CS 240 - Lab 2

TAs: Yash J. & Abhi Jain

Spring 2025

Welcome to the second graded lab of this course!!

As discussed in the last lab, the in-lab portion would be based on/similar to the outlab of the previous lab. Whereas, the outlab portion will build on different techniques iterative optimization techniques like gradient descent, stochastic gradient descent, Newton's second order method, etc.

They are arranged in increasing order of difficulty, and you are encouraged to attempt them sequentially. Throughout the lab, focus on understanding what is happening and why, and feel free to pause and ask questions rather than mechanically completing the tasks and moving on.

Instructions:

- This lab has 3 questions. Only the first question is graded in-lab. We have provided you with the following files:

```
lab2.pdf
algos/
|__ grad_descent.py
|__ LSLR_algo1.py
|__ LSLR_algo2.py
|__ LSLR_algo3.py
|__ optim.py

functions/
|__ func.py

metrics/
|__ metrics.py

q1/
|__ q1_main.py
|__ q1_solve.py
|__ q1_train.csv
|__ q1_test.csv

q2/
|__ q2.py
|__ readme.txt
```

```
q3/
|__ A/
|   |__ A.py
|   |__ readme.txt
|__ B/
|   |__ B.py
|   |__ readme.txt
|__ C/
|   |__ C.py
|   |__ readme.txt
|__ data/
    |__ datasets/
        |__ A/
        |   |__ X.npy
        |   |__ y.npy
        |__ D/
            |__ X.npy
            |__ y.npy
```

- Create a folder on your Desktop with the following directory structure. The folder must contain only the one specified file:

  ```
  submission_<roll_number>/
  |-- q1_solve.py
  ```

- Once you have completed the work, open the terminal, type the following command, and then call a TA: `check-cs240`
- The TA will run `submit-cs240` to submit your work to the server. **It is your responsibility to ensure your work is submitted.** If you fail to inform us to submit your work, you will not be graded.

Graded In-Lab Begins :)

Q1. In-Lab: Ridge Regression and Hyperparameter Optimization

You are a data scientist at SunGrid*, a renewable energy startup that claims to be "AI-powered" because NumPy is installed on the server. Your task is to predict* ***daily energy demand*** *using environmental signals such as temperature and humidity, and a few extra features added by* Dr. Jhatka *during an enthusiastic brainstorming session. No one remembers what these features mean, but deleting them is considered "too risky for production".*

Last month, Motu *trained an ordinary least squares regression model. The training error was almost zero. The test error was so bad that the battery system briefly attempted to violate the laws of physics.* Patlu *suggested using* ***Ridge Regression*** *to fix the model*

You are now tasked with fixing the model as per Patlu's *suggestions and tuning the parameters using cross-validation.*

The dataset is small and the features are highly correlated. Using Ordinary Least Squares (OLS), the model fits the training data well but performs poorly on unseen data.

Your task is to stabilize the model using **Ridge Regression**, select the regularization parameter λ using **cross-validation** and compare simple optimization strategies for hyperparameter tuning.

Data Format

You are given two CSV files:

- `q1_train.csv`
- `q1_test.csv`

Each row has the form: $(x_1, x_2, \ldots, x_d, y)$ where the last column is the target value y.

Linear Regression Review

Given a design matrix $X \in \mathbb{R}^{N\times d}$ and targets $y \in \mathbb{R}^N$, Ordinary Least Squares solves:

$$\min_w \|y - Xw\|_2^2$$

If $X^T X$ is invertible, the solution is:

$$w_{\text{OLS}} = (X^T X)^{-1} X^T y$$

OLS is unstable when features are correlated, or $X^T X$ is ill-conditioned. You have studied Ridge Regression now, and hence the graded tasks for your lab are:

Task 1 Implementing Ridge Regression (20 Points)

Ridge regression adds an ℓ_2 penalty:

$$\min_w \|y - Xw\|_2^2 + \lambda \|w\|_2^2$$

The closed-form solution is:

$$w = (X^T X + \lambda I)^{-1} X^T y$$

Task 1.1 Feature Standardization

Before applying ridge regression, each feature is standardized:

$$x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}$$

where μ_j and σ_j are computed on the training set.

Task 1.2 Handling Bias Term

When a bias (intercept) term is included by adding a column of ones to X, it should *not* be regularized. This is handled by using:

$$D = \text{diag}(0, 1, 1, \ldots, 1)$$

and solving:

$$(X^T X + \lambda D)w = X^T y$$

Task 2 Implementing Cross-Validation (15 Points)

To estimate generalization performance, you will use k-fold cross-validation; i.e. split the data into k folds, train on $k - 1$ folds, validate on the remaining fold and average this MSE over all folds.

Task 3 Optimizing Hyperparameters (15 Points)

Task 3.1 Grid Search

Grid search evaluates a fixed set of λ values (log-spaced): $\lambda \in \{10^{-4}, \ldots, 10^4\}$

It is simple and exhaustive but can be computationally inefficient.

Task 3.2 Random Search

Instead of fixed points, random search samples: $\lambda = 10^u, \quad u \sim \text{Uniform}(a, b)$

Random search often finds good hyperparameters faster than grid search, especially when the search space is large.

What You Must Implement?

In `q1_solve.py`, implement all the non-implemented functions.

DO NOT:

- Don't use scikit-learn in `q1_solve.py`.
- Don't make any changes in `q1_main.py`. We will change it our autograding and only `q1_solve.py` has to be submitted.

Graded In-Lab Ends :)

Exercises on Optimization Begin :O

Note: This contains the ungraded in-lab and the graded out-lab.

Preliminaries and Setup

This section establishes the formal notation and mathematical properties required to solve the assignment. You are expected to interpret these definitions to implement the algorithms and analyze their convergence behaviors.

1. Least Squares Linear Regression

We consider the **Least Squares** problem for Linear Regression.

- **Dataset:** Let $X \in \mathbb{R}^{n\times d}$ be the feature matrix containing n samples and d features. Let $y \in \mathbb{R}^n$ be the target vector.
- **Parameters:** Let $w \in \mathbb{R}^d$ be the weight vector we wish to optimize.
- **Objective Function:** We minimize the Mean Squared Error (MSE): $f(w) = \frac{1}{2n}\|Xw - y\|_2^2$

2. Linear Algebra Foundations

Understanding the geometry of X is crucial for analyzing optimization performance.

2.1 Singular Value Decomposition (SVD)

Any matrix $X \in R^{n\times d}$ can be decomposed as $X = U\Sigma V^\top$, where:

- $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{d\times d}$ are orthonormal matrices, i.e. $U^TU = I$ and $V^TV = I$.
- The matrix $\Sigma \in \mathbb{R}^{n\times d}$ is diagonal (possibly rectangular), with diagonal entries

$$\Sigma_{ii} = \sigma_i, \qquad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(n,d)} \geq 0,$$

  and $\Sigma_{ij} = 0$ for $i \neq j$. The values $\{\sigma_i\}$ are called the **singular values** of X.
- **Geometric interpretation:** X maps the unit sphere to an ellipsoid whose principal axes are given by the right singular vectors (columns of V) and whose axis lengths are $\{\sigma_i\}$.
- **Numpy:** U, Σ, V^T = np.linalg.svd(X, full_matrices=False)

Computational Cost: Computing the SVD or inversion for the pseudoinverse scales as $\mathcal{O}(nd^2)$ or $\mathcal{O}(d^3)$. For large d (e.g., $d > 10,000$), this is **intractable**, motivating the use of iterative methods.

2.2 Eigenvalues and The Hessian

For Least Squares, the curvature of the loss surface is determined by the hessian matrix $H = X^\top X$.

- The eigenvalues of H satisfy $\lambda_i(X^\top X) = \sigma_i(X)^2$. This is because the SVD diagonalizes the Gram matrices:

$$X^\top X = V\Sigma^\top\Sigma V^\top, \qquad XX^\top = U\Sigma\Sigma^\top U^\top.$$

  Hence, the squared singular values $\{\sigma_i^2\}$ are the eigenvalues of both $X^\top X$ and $XX^\top$.

- Curvature of the loss surface along direction v_i (eigenvector of $X^\top X$) is proportional to σ_i^2.
- **Numpy:** `eigenvalues = np.linalg.eigvals(X)` (Use `eigvalsh` for symmetric matrices)

2.3 Condition Number (κ)

The condition number measures how "stretched" the loss landscape is (elliptical vs. spherical).

$$\kappa(X) = \frac{\sigma_{\max}(X)}{\sigma_{\min}(X)} \implies \kappa(X^\top X) = \frac{\lambda_{\max}(X^\top X)}{\lambda_{\min}(X^\top X)} = \kappa^2(X)$$

- **Numpy:** `np.linalg.cond(X)`

2.4 Stable Rank (sr)

The stable rank provides a continuous notion of effective dimensionality:

$$\mathrm{sr}(X) = \frac{\sum \sigma_i^2}{\sigma_{\max}^2}$$

3. Relevant Numpy Functions Cheat Sheet

Mathematical Concept	Numpy Implementation	Note
Transpose	`A.T`	View, no copy
Matrix Multiplication AB	`A @ B`	
Solve $Ax = b$ (Exact)	`np.linalg.solve(A, b)`	Requires A square/full rank
Solve Least Squares	`np.linalg.lstsq(X, y, rcond=None)`	The "intractable" baseline
Pseudo-inverse	`np.linalg.pinv(X)`	SVD-based
Singular values	`np.linalg.svd(X)`	
Eigenvalues (general)	`np.linalg.eigvals(A)`	May be complex
Eigenvalues (symmetric)	`np.linalg.eigvalsh(A)`	Faster, real-valued
Condition number	`np.linalg.cond(X)`	Ill-conditioning measure
Matrix rank	`np.linalg.matrix_rank(X)`	Numerical rank
Gradient $\nabla f(w)$ for OLS	`X.T @ (X @ w - y)`	Vectorized implementation

Q2. Gradient Descent: The First Example [Ungraded In-Lab]

We will implement gradient descent on the following miniature functions:

1. Rosenbrock Function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

   take $a = 1$ and $b = 100$

2. Rotated Anisotropic Quadratic

$$f(x) = x^T Q x - b^T x$$

You would do the following:

Task 1 Rosenbrock function

For Rosenbrock function, fill the corresponding rosenbrock callable class in func.py in functions folder. Note the MSE obtained and the minima point x^* at the end of the run. Compare it with the analytic solution obtained by putting $\nabla_x f(x) = 0$

Task 2 Rotated Anisotropic Quadratic

You repeat the same thing as above for $Q = U\Sigma V^T$, where all $U, V,$ and Σ are 5x5 matrices.

$$U = \begin{bmatrix} -0.5615 & -0.0142 & 0.3363 & -0.7292 & 0.1992 \\ 0.6419 & -0.1191 & -0.2419 & -0.4509 & 0.5585 \\ 0.4427 & 0.3283 & 0.8318 & 0.0191 & -0.0630 \\ 0.0350 & -0.9228 & 0.3452 & 0.1621 & 0.0432 \\ 0.2748 & -0.1621 & -0.1317 & -0.4882 & -0.8016 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.3582 & -0.7321 & 0.2857 & 0.4717 & 0.1777 \\ 0.2772 & 0.3230 & 0.0332 & -0.0691 & 0.9017 \\ 0.1081 & 0.4481 & 0.8425 & 0.1824 & -0.2108 \\ 0.0002 & 0.3732 & -0.3938 & 0.8382 & -0.0550 \\ 0.8850 & 0.1404 & -0.2289 & -0.1918 & -0.3286 \end{bmatrix},$$

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 \\ 0 & 0 & 0 & \sigma_4 & 0 \\ 0 & 0 & 0 & 0 & \sigma_5 \end{bmatrix}.$$

The σ_is are deliberatly left empty so that you tinker with it. Try observing MSE at 10, 100, 1000 and 10000 epochs in the following cases:

1. when σ_{max} is very small or 0.
2. when for some j, $\sigma_j = 100$ and otherwise $\sigma_i = 1 \quad \forall i \neq j$.
3. when all σ_i are equal, and say equal to 10.

In all the above experiments, maintain b = 0.

Also experiment with the learning rate. Try to observe how MSE and convergence of the algorithm changes. Try to see if there is any correlation with condition number (κ) or stable rank (sr). Also compare x^* obtained with the analytic solution.

Running q2.py outputs only metrics_params.json for the given parameters of a, b, Q, and b^T, which function and epochs/iterations to run. It does not generate plots, you have to look into the metrics_params.json file in the q2 folder or make your own plots.

Also, note that all code is vectorised in the repository. So code accordingly.

Update!! Some people complained about convergence challenges for part B. You have write functions in func.py as if $U \neq V$. But in the code we have set $U = V$ now to ensure strong convexity. Please do not change U and V values in the code.

Q3. Least Squares Linear Regression

Tasks A and B of this question are the graded outlab, while Task C is there for your exploration and is ungraded.

Problem Description:

We aim to solve the classic unconstrained Least Squares optimization problem:

$$\min_{w \in \mathbb{R}^d} f(w) = \frac{1}{n}\|Xw - y\|_2^2$$

While the analytical solution (in case of full rank), $w^* = (X^\top X)^{-1} X^\top y$ is mathematically elegant, it is computationally intractable for large-scale datasets due to the cubic complexity of matrix inversion or singular value decomposition. Consequently, Machine Learning relies heavily on **iterative optimization algorithms** that approximate the solution by traversing the loss landscape.

In this Question, you will investigate the behavior of Deterministic (Full-Batch), Stochastic, Randomised Kacmarz, and Second-Order algorithms. You will not only implement these methods but also rigorously analyze their performance trade-offs.

The central theme of this lab is the distinction between **Algorithmic Convergence** (error reduction per iteration) and **Computational Efficiency** (error reduction per unit of wall-clock time). You will demonstrate that the "best" algorithm depends entirely on the spectral properties of the dataset (Condition Number, Stable Rank) and the compute budget. For a given dataset (X, y), the convergence behavior of optimization algorithms is dictated by the parameters like stable rank and condition number as defined in the preliminaries.

Task A: Getting Started [Graded]

Before implementing optimization routines, you must establish the motivations, and what geometrical factors of the dataset affect convergence.

1. The Cost of Exactness:
Implement the exact_solve_function function that computes the exact solution using the Moore-Penrose pseudoinverse. Your function must return:

- the optimal solution $w^\star$,
- the wall-clock time required to compute it.

This will serve as a reference point for evaluating iterative methods.

2. Vanilla Gradient Descent:
The following formal update rule defines the deterministic Full-Batch Gradient Descent. Let η_t denote the step size at iteration t. The update rule is

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla f(w^{(t)}).$$

Fill in the `LSLR` (Least Squares Linear Regression) callable class with these definitions for the derivative and Hessian. Further implement full-batch Gradient Descent in the `GradientDescent` class in the `grad_descent.py` file.

Other metrics are autorecorded and plotted for your reference.

Task B: The Algorithm Leaderboard [Graded]

In this task, you need to implement three of the iterative optimization algorithms described below. We will have 2 private and 2 public datasets, and for each dataset, we will evaluate you based on the time taken to achieve convergence under a certain precision, selecting the best out of the 3 you implement.

Note that an *algorithm* includes all design choices, such as step-size schedule, batch size, and sampling strategy.

Step-Size Schedule

GD and SGD require different step-size schedules. There are certain conditions under which we can give different kinds of guarantees of convergence. This part is left open for your exploration. You will be graded based on how well your algorithms perform on different datasets.

As a hint, for stochastic methods, decaying step sizes are typically required for convergence to high precision, while variance-reduced methods may allow constant step sizes.

Analysis and Plots:
must generate two distinct types of convergence plots:

- **MSE vs. Iterations:** This demonstrates the *algorithmic* convergence rate.
- **MSE vs. Wall-Clock Time:** This demonstrates *computational* efficiency. You should observe that despite requiring more iterations, SGD may reduce the error faster in the early stages compared to GD.

Note that you will be mainly graded based on MSE recorded at different Wall Clock times and not iterations. For the same compute your algorithm should have lower MSE.

Algorithm Specifications

The following is the standard step rule for stochastic gradient descent where $G(w,\gamma)$ is the stochastic derivative over the random variable γ such that $E(G(w,\gamma)) = \nabla f(w)$ (unbiasedness condition)

$$w^{(t+1)} = w^{(t)} - \eta_t G(w^{(t)}, \gamma^{(t)}),$$

Hence, we replace the gradient with an unbiased estimator of the gradient in order to perform stochastic gradient descent. This unbiased estimator can take many forms, as discussed in the techniques below. In each case please convince yourself that the estimate is unbiased.

1. Vanilla Stochastic Gradient Descent (SGD) The following formal update rule defines standard Stochastic Gradient Descent. Let η_t denote the step size at iteration t. Let $G(w,i)$ be a stochastic estimator of the gradient using a random data point index $i \in \{1,\ldots,n\}$ such that

$$G(w,i) = \nabla f_i(w) \qquad \text{where } f_i \text{ is loss due to } i^{th} \text{ point}$$

The update rule is

$$w^{(t+1)} = w^{(t)} - \eta_t\, G(w^{(t)}, i^{(t)}).$$

For Least Squares, $G(w,i) = x_i(x_i^\top w - y_i)$, where x_i is the i-th row.

2. Mini-Batch Gradient Descent At iteration t, sample a subset of data indices $B_t = \{i_1, \dots, i_b\} \subset \{1, \dots, n\}$ of size b. The update approximates the full gradient using this batch:

$$G(w, B) = \frac{1}{b} \sum_{i \in B} \nabla f_i(w)$$

The update rule is

$$w^{(t+1)} = w^{(t)} - \eta_t G(w^{(t)}, B^{(t)})$$

3. Coordinate Gradient Descent The following formal update rule defines coordinate gradient descent. Let η_t denote the step size at iteration t. Let $G(w, \gamma)$ be a stochastic estimator of the gradient using a random feature index $\gamma \in \{1, \dots, d\}$ such that:

$$G(w, \gamma) = k_\gamma \, e_\gamma \big(\nabla f(w)\big)_\gamma$$

where e_γ is the standard basis vector and k_γ is a scaling factor chosen to ensure unbiasedness. Notice how $k_\gamma = 1/p_\gamma$ where p_γ is the probability distribution of the random variable γ.

For OLS, the partial derivative with respect to coordinate γ is

$$(\nabla f(w))_\gamma = \frac{1}{n}(X_{:\gamma})^\top (Xw - y).$$

The update rule is

$$w^{(t+1)} = w^{(t)} - \eta_t \, G(w^{(t)}, \gamma^{(t)})$$

3.1 Standard Coordinate Descent (Uniform Sampling)

$$p_\gamma = \frac{1}{d}, \qquad k_\gamma = d$$

The unbiased estimator is

$$G_{\text{uni}}(w, \gamma) = d \, e_\gamma \big(\nabla f(w)\big)_\gamma.$$

3.2 Randomized Kaczmarz (Non-Uniform Sampling)

$$p_\gamma = \frac{\|X_{:\gamma}\|_2^2}{\|X\|_F^2}, \qquad k_\gamma = \frac{\|X\|_F^2}{\|X_{:\gamma}\|_2^2}$$

where $\|.\|_F$ is the Frobenius norm (read online).

The unbiased estimator is

$$G_{\text{RK}}(w, \gamma) = \frac{\|X\|_F^2}{\|X_{:\gamma}\|_2^2} \, e_\gamma \big(\nabla f(w)\big)_\gamma.$$

Note: Usually, a learning rate of $\eta_t = 1$ works well due to non-uniform sampling.

4. Mini-Batch Coordinate Descent At iteration t, sample a subset of feature indices

$$B_t = \{\gamma_1, \gamma_2, \ldots, \gamma_b\} \subset \{1, \ldots, d\}$$

of size b. The update rule is

$$w^{(t+1)} = w^{(t)} - \eta_t \left(\frac{1}{b} \sum_{\gamma \in B_t} G_{\text{uni}}(w^{(t)}, \gamma) \right)$$

where the same unbiased estimator G_{uni} from uniform sampling is used.

5. Stochastic Variance Reduced Gradient (SVRG) SVRG is an epoch-based stochastic optimization method that reduces the variance of stochastic gradients by periodically computing a full gradient snapshot.

The SVRG gradient estimator is defined as

$$G(w; \bar{w}, i) = \nabla f_i(w) - \nabla f_i(\bar{w}) + \nabla f(\bar{w}),$$

where $\bar{w}$ is a snapshot point that is fixed within an epoch.

Algorithm. Initialize $w^{(0)}$. For epochs $s = 0, 1, 2, \ldots$:

1. Set the snapshot $\bar{w} \leftarrow w^{(s)}$ and compute the full gradient
$$\nabla f(\bar{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\bar{w}).$$
2. Set $w_{\text{in}}^0 \leftarrow \bar{w}$.
3. For $t = 0, \ldots, N-1$:
$$\begin{aligned} &\text{Sample } i_t \sim \text{Uniform}(\{1, \ldots, n\}), \\ &G\big(w_{\text{in}}^t; \bar{w}, i_t\big) = \nabla f_{i_t}(w_{\text{in}}^t) - \nabla f_{i_t}(\bar{w}) + \nabla f(\bar{w}), \\ &w_{\text{in}}^{t+1} = w_{\text{in}}^t - \eta_t G\big(w_{\text{in}}^t; \bar{w}, i_t\big)\,. \end{aligned}$$
4. Set $w^{(s+1)} = w_{\text{in}}^N$.

Other Optimization Techniques

Here are some advanced optimization methods frequently used in Deep Learning and Convex Optimization that you may wish to explore further:

- **Momentum (Heavy Ball):** Accelerates SGD by accumulating a "velocity" vector from past gradients to dampen oscillations and push through flat regions.
- **Nesterov Accelerated Gradient (NAG):** A refined momentum method that calculates the gradient at a "lookahead" position rather than the current position to correct the course faster.
- **Adaptive Methods (Adam, RMSProp):** Algorithms that automatically adapt the learning rate for each parameter individually based on the variance of historical gradients.
- **Newton's Method:** A second-order method that uses the inverse Hessian matrix (curvature) to take a direct path to the minimum, though computationally expensive ($O(d^3)$).

Some Instructions

You have to fill the files in the algos folder - LSLR_algo1.py, LSLR_algo2.py, LSLR_algo3.py with your algorithms. Unlike how in q2 we were giving function to be optimised as a parameter, the function eval(...), grad(...), stoch_grad(...) will be defined in the optimiser class itself, and of course these are to be filled with respect to Least Squares loss function.

B.py in the q3 folder is mainly for your own tinkering and experimentation. Will not be used in grading, we will be importing your defined algorithm classes in our autograders. We will be grading based on relative thresholds of performance, so be competitive :)

Task C: The UNO Reverse [Ungraded]

The datasets in part A and B are generated by us to have specific properties so that distinctions between algorithms can be observed well. Now its your job to generate them!

This is an exploratory and open task where you have to generate datasets based on a condition. This task tests your deep understanding of how data geometry affects optimization. You cannot simply use provided data; you must synthetically generate matrices X and vectors y to prove specific hypotheses.

In this part, your knowledge of Q2 will help, as the stable rank and condition number affect which of Randomised Kacmarz or full batch GD works better.

Your job is to construct datasets with $X \in R^{20000 \times 50}$ and $y \in R^{20000 \times 1}$ for the LSLR problem with a given stable rank and condition number approximately. You will have to figure what stable rank and condition number to choose based on manual tinkering.

The Challenge: Iterative algorithms behave differently depending on the **Stable Rank** and **Condition Number** of the feature matrix. You must generate two distinct datasets to demonstrate the following phenomena:

1. **Scenario 1: The Victory of Determinism**
   Generate a dataset where **Full-Batch Gradient Descent** significantly outperforms Stochastic variants (in terms of stability and final error) within a fixed time budget.
2. **Scenario 2: The Triumph of Stochastic**
   Generate a dataset where **Stochastic variants** (SGD or Kaczmarz) significantly outperform Full-Batch GD in terms of wall-clock time to reach a specific low MSE.

Some Instructions

Fill the functions in C.py for dataset generation. Generate the dataset and run the task B code on it. You can change the dataset_path variable in the B.py code.