Assignment 5: Logistic Regression

1 Instructions

You will build a Logistic Regression classifier from scratch to predict binary outcomes on a high-dimensional dataset and then perform GPU computations via JAX.

1.1 Conceptual Background

Logistic Regression is a probabilistic model that estimates $P(y|x)$. To handle non-linear decision boundaries, we often increase the **expressivity** of our model by creating synthetic features (like polynomials). However, increasing expressivity introduces the risk of **overfitting**—where the model memorizes the training noise rather than the underlying signal.

Regularization (specifically L2 regularization) serves as a way to control the complexity of the model. By penalizing large weights, we effectively limit the model's capacity, pulling it back from the overfitting regime to a solution that generalizes better to unseen data.

For more information, you can read the "Machine Learning Basics" covered in Chapter 5 of the *Deep Learning Book (ML chapter)*.

You will move beyond "getting it to work" and focus on "getting it to work fast and accurately" by leveraging:

1. **Feature Engineering:** Expanding tabular data into thousands of features to capture non-linear interactions (increasing Model Capacity).
2. **Regularization:** Preventing overfitting in high-dimensional spaces using L2 penalties.
3. **Hardware Acceleration:** Using Google's **JAX** library to run your NumPy code on a **GPU**, achieving massive speedups for large matrix operations.

1.2 Submission Requirements

You must submit two components to Gradescope:

1. **The Report:** A plaintext Markdown document containing the 5 sections described in Part 2.
    - **No Images:** Describe your findings with text and data tables.
    - **Character Limit:** 7,500 characters.

2. **Supplemental Material:** Your `.ipynb` notebook containing all code, plots, and raw results.

---

2 Task 1: Implementation (The Notebook)

You will author a Jupyter Notebook. It is **highly recommended** that you use **Google Colab** for this assignment to access the free T4 GPU for Task 3. When you get to the JAX/GPU part, you will need to change Colab's runtime to a GPU/TPU. Read **Google Colab's** FAQs *here.*. If you do not use Colab, you must have access to a GPU to do a timing comparison. *Note:* We suggest using CPU runtime for Tasks 1 and 2, and then switching to GPU runtime for Task 3 to avoid unnecessary GPU usage during development.

Global Setup:

- Fix the random state of numpy. You can globally due this via `np.random.seed(42)` or by creating a `RandomState` object and passing it around.
- Load the **Forest Cover Type** dataset (`sklearn.datasets.fetch_covtype`).
  - *Simplification:* Select only the first two classes (Spruce-Fir vs Lodgepole Pine) to make this a **binary classification** problem.
- *Subsampling:* Select a random subset of **25,000 samples** total.

2.1 Task 1: Data Setup and Preprocessing

2.1.1 Task 1.1: Feature Engineering (Raw vs. Expanded)

Linear models like Logistic Regression are "high bias" models—they assume the decision boundary is a flat hyperplane. To capture complex relationships (like the interaction between "Elevation" and "Slope"), we can create synthetic features.

You will prepare **two** distinct datasets for this assignment to compare the power of feature engineering.

1. **Dataset A with Raw Features:** The dataset has 54 columns (10 quantitative, 44 binary). Keep this version stored as `X_raw`.
2. **Dataset B with Feature Expansion:** Create **pairwise interaction features** based on `X_raw`. For every pair of features x_i, x_j, add a new feature $x_{new} = x_i \cdot x_j$.
   - *Implementation Hint:* You can use `sklearn.preprocessing.PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)`. `interaction_only=True` means that it will not include x_i^2 terms.
   - *Check:* If you started with ~ 54 features, you should end up with over 1,400 features. Store this as `X_expanded`.

3. **Data Splitting (Crucial):** For **BOTH** `X_raw` and `X_expanded`:
    - **Test Set:** Hold out **5,000 samples** for the final evaluation. (Do not touch these until Task 2.2).
    - **Training Set:** Use the remaining **20,000 samples** for training and hyperparameter tuning.

2.1.2 Task 1.2: Feature Normalization

Gradient Descent is sensitive to the scale of features. If one feature ranges from 0 to 1 and another from 0 to 10,000 (like Elevation), then the parameter values corresonding to these features may vary widely. Perhaps more importantly, the scale of the features will significantly affect the parameter regularization we will introduce later. Thus, we will normalize all features to have zero mean and unit variance.

1. **Calculate Statistics:** Compute the feature-wise means μ_j and standard deviations σ_j of the **Training Set**.
    - *Note:* You must do this separately for the Raw dataset and the Expanded dataset.
    - You should compute all means and stds in a vectorized way without explicit loops.
2. **Standardize:** Transform the features of both Training and Test sets: $x'_j = \frac{x_j - \mu_j}{\sigma_j}$.
    - *Implementation Note:* Add a small epsilon ($\epsilon = 1e-8$) to the denominator to avoid division by zero for constant columns.
    - *Crucial:* Use the training set statistics to normalize the test set. Never "leak" test set statistics.
    - You should use broadcasting to do this efficiently without explicit loops.

Notebook Output:

- Print the shape of `X_raw` and `X_expanded` (e.g., "Raw: 54 features, Expanded: 1485 features").
- Print the mean and std of the first 5 features of `X_expanded` after normalization (should be close to 0 and 1).

2.2 Task 2: Logistic Regression from Scratch

2.2.1 Concept: The Mathematical Model

You will implement Logistic Regression using **NumPy only**. Do not use `sklearn.linear_model`. You may not use automatic differentiation libraries for this part (e.g., JAX's `grad`). You must implement the gradient explicitly.

Notation Clarification: In lecture slides, you may see the hypothesis as $h(x) = \sigma(\theta^T x)$. In code, we vectorize this for m samples using a design matrix X of shape (m, n). To make the vector math $X\theta$ work, you must handle the **bias term** (intercept). **Instruction:** Manually concatenate

a column of **ones** to your X matrix (both Train and Test). Your parameter vector θ will now have size $n+1$.

1. The Hypothesis (Prediction): The model predicts the probability that $y = 1$ using the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$.

$$\hat{y} = \sigma(X\theta) = \frac{1}{1 + e^{-X\theta}}$$

2. The Objective (Loss Function): We minimize the **Binary Cross Entropy** loss, which is equivalent to the **negative log-likelihood** discussed in class. To handle our exploded feature space, we add **L2 Regularization** (Ridge) to prevent overfitting.

$$\begin{aligned} J(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\sigma(x^{(i)}\theta)) + (1-y^{(i)})\log(1-\sigma(x^{(i)}\theta))\right] + \frac{\lambda}{2m}\sum_{j=0}^{n}\theta_j^2 \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=0}^{n}\theta_j^2 \end{aligned}$$

- m: Number of samples.
- λ (lambda): Regularization strength.
- *Note on vectorization:* The loss function is written in a summation form for clarity, but you should implement it in a fully vectorized way without explicit loops. Use matrix operations to compute the predictions and the loss efficiently.
- *Note on Regularizing Bias:* By convention, we often exclude the bias term (θ_0) from the regularization sum. However, for this assignment, **regularizing all theta values is acceptable** to keep the vectorization code simple (hence the summation starting at $j = 0$ above).
- *Note on numerical stability:* For numerical stability, you may want to use a numeric function that combines the log and sigmoid functions together into the same function call: scipy.special.log_expit. This can help avoid issues with log(0) or overflow when computing the loss.

3. The Gradient (Update Rule): To minimize $J(\theta)$, we take the derivative with respect to θ.

$$\begin{aligned} \nabla J(\theta) &= \frac{1}{m}X^T(\sigma(X\theta) - y) + \frac{\lambda}{m}\theta \\ &= \frac{1}{m}X^T(\hat{y} - y) + \frac{\lambda}{m}\theta \end{aligned}$$

2.2.2 Task 2.1: Implementation

Implement a class or set of functions:

1. `sigmoid(z)`: Returns probabilities.
2. `compute_loss(X, y, theta, lambda_reg)`: Returns the float value of $J(\theta)$.
3. `compute_gradient(X, y, theta, lambda_reg)`: Returns the vector $\nabla J(\theta)$.
4. `train(X, y, lr, epochs, lambda_reg)`: The gradient descent loop.
    - Initialize θ with small random values.

- Loop for `epochs` iterations:
  - Calculate gradient.
  - Update: $\theta := \theta - \eta \nabla J(\theta)$ (where η is learning rate).
- (Optional) Print loss every 100 epochs.

2.2.3 Task 2.2: Hyperparameter Tuning (Comparative Study)

You must find the "Goldilocks" zone for the learning rate (η) and regularization (λ). You will perform this search **twice**: once for the Raw data and once for the Expanded data.

1. **Validation Split:** Split your **20,000 Training samples** into **Sub-Train** (16,000) and **Validation** (4,000).
2. **Grid Search:** Try the following combinations:
   - Learning Rates (η): `[0.1, 0.01, 0.001, 0.0001]`
   - Regularization (λ): `[0, 0.1, 1, 10, 100]`
3. **Experiment A (Raw Features):**
   - Run the grid search on `X_raw`. Train on Sub-Train (16k), evaluate on Validation (4k).
   - Select the best $(\eta_{raw}, \lambda_{raw})$ based on validation accuracy.
4. **Experiment B (Expanded Features):**
   - Run the grid search on `X_expanded`. Train on Sub-Train (16k), evaluate on Validation (4k).
   - Select the best $(\eta_{exp}, \lambda_{exp})$ based on validation accuracy.
5. **Final Training & Evaluation:**
   - **Model A:** Retrain on the full 20k Raw Training set using best $(\eta_{raw}, \lambda_{raw})$. Report final Train and Test accuracy.
   - **Model B:** Retrain on the full 20k Expanded Training set using best $(\eta_{exp}, \lambda_{exp})$. Report final Train and Test accuracy.

Notebook Output:

- **Plot:** Loss vs. Epochs for the best Model B (Expanded) during the final refit.
- **Metric:** Final Train and Test Accuracy for Model A (Raw).
- **Metric:** Final Train and Test Accuracy for Model B (Expanded).

2.3 Task 3: Entering the Matrix (JAX & GPU)

2.3.1 Concept: Hardware Acceleration

For this task, you will **NOT** use JAX's automatic differentiation. You will simply swap `numpy` for `jax.numpy`. **Important:** *You must switch to a GPU runtime (e.g., Google Colab) to see the speedup.* JAX will automatically run your code on the GPU if it detects one. The speedup comes

from the fact that matrix operations (like $X\theta$) are highly optimized on GPUs, and JAX can leverage this without any code changes. The only difference is that you will be using `jnp` instead of `np`. You will still implement the same gradient descent algorithm, but it will run much faster on the GPU for large datasets.

2.3.2 Task 3.1: Porting to JAX

1. **Import:** `import jax.numpy as jnp`
2. **Data Transfer:** Move your **Expanded** dataset (`X_expanded`, `y`) to the GPU. (The raw dataset is too small to see meaningful GPU gains).
    - `X_gpu = jnp.array(X_expanded)`
    - `y_gpu = jnp.array(y)`
    - *Note:* JAX usually handles this lazily, but explicit creation ensures it's on the accelerator device.
    - *Tip:* You can check if the data is on the GPU/TPU by inspecting `X_gpu.device()`.
3. **Rewrite:** Copy your gradient descent functions and replace `np.` with `jnp.`. Remember that JAX arrays are immutable; use out-of-place updates (`theta = theta - lr * grad`).

2.3.3 Task 3.2: The Speed Test

Compare the training time of your **NumPy (CPU)** implementation vs. your **JAX (GPU)** implementation using the **Expanded Dataset**.

1. **Setup:** Fix hyperparameters ($\eta = 0.01, \lambda = 10, \text{epochs} = 2000$).
2. **Run CPU:** Time the execution of the pure NumPy training loop.
3. **Run GPU:** Time the execution of the JAX training loop.
    - *Critical:* JAX is asynchronous. To get accurate timing, you must block until the result is ready.
    - `final_theta.block_until_ready()` must be called before stopping the timer.
    - *Performance Note:* Real-world JAX uses `@jax.jit` (Just-In-Time compilation) to fuse operations, which provides the massive speedups you see in benchmarks. Here, we are just measuring the raw matrix multiplication speedup on the GPU.
4. **Scale It Up:** Increase the dataset features or size (if memory allows) or simply run for more epochs to observe where the GPU advantage becomes significant.

Notebook Output:

- **Table:** Comparison of "Time per 1000 epochs" for CPU vs GPU.
- **Value:** Calculate the speedup factor ($Time_{CPU}/Time_{GPU}$).

---

3 Part 2: Content Requirements (The Report)

Your report must be organized into **exactly five sections** with Markdown headers.

3.1 Section 1: Executive Summary

- **One-Sentence Takeaway:** Summarize the most impactful result (e.g., "Feature expansion improved accuracy by X%, while GPU acceleration improved training speed by Yx").
- **Summary Paragraph:** Briefly outline the problem, the methods (Raw vs Expanded comparison), and key findings regarding accuracy and performance.

3.2 Section 2: Mathematical Derivation & Normalization

- **The Gradient:** Write out the vectorized equation you used for the gradient step.
- **Normalization Defense:** Explain **why** normalization was important for this dataset.

3.3 Section 3: Feature Expansion & Model Complexity

This is the core analysis of the assignment.

- **Comparison Table:** Create a Markdown table summarizing the results:

Model	Best η	Best λ	Final Train Acc	Final Test Acc
Raw Features (54)	...	...	...	...
Expanded Features (~1.4k)	...	...	...	...

- **Analysis:**
  - Did feature expansion improve the Test Accuracy? Why or why not?
  - Compare the gap between Train/Test accuracy for Raw vs. Expanded. Did the expanded model show signs of higher variance (overfitting)?
  - Did the Expanded model require stronger regularization (higher λ) than the Raw model? Explain why this might be the case.

3.4 Section 4: High-Performance Computing (JAX)

- **Speedup Analysis:** Present a small Markdown table comparing the Wall-Clock time for NumPy (CPU) vs JAX (GPU) on the **Expanded** dataset.
- **Report the Speedup Factor.**
- **Implementation Experience:** Describe the process of porting to JAX. Did you encounter issues with immutability? Why does the GPU perform better for this specific workload (Matrix-Vector multiplication)?

3.5 Section 5: Reflection

- **Learning Rate Sensitivity:** Describe what happened when you chose a learning rate that was **too high** or **too low**.
- **The "Black Box":** Now that you have implemented Logistic Regression from scratch, how does this change your perspective when calling `sklearn.linear_model.LogisticRegression`?
- Give any other insightful observations or reflections on the assignment.

4 Grading Rubric

Each of the five sections is weighted equally (**20% each**).

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 1: Executive Summary	Takeaway is specific, quantitative, and captures the core ideas. Summary clearly outlines the problem and key findings.	Takeaway is clear. Summary covers the main comparison points (Raw vs. Expanded).	Takeaway is generic. Summary is present but vague or misses key findings.	Summary misses key elements of the analysis.	Missing.
Section 2: Math & Normalization	Gradient equation is correctly vectorized. Explanation of normalization demonstrates deep understanding of its impact on convergence and regularization.	Equation is correct. Good explanation of why normalization is needed.	Equation is present. Explanation is generic (e.g., "it helps training").	Equation is incorrect or explanation is missing.	Missing.

Criterion	Excellent (5)	Good (4)	Satisfactory (3)	Okay (2)	Poor (1)
Section 3: Feature Expansion	Comparison table is complete. Analysis provides deep insight into why expanded features require stronger regularization (λ) and how they impact generalization.	Table is complete. Analysis correctly identifies the need for regularization.	Table present. Analysis states results (e.g., "accuracy went up") without explaining "why".	Table missing or values are clearly incorrect.	Missing or fails to implement expansion.
Section 4: JAX & Speedup	Speedup factor is reported accurately. insightful discussion on *why* GPUs excel at this specific workload (matrix mult) and the experience of JAX immutability.	Speedup factor present. Good description of the porting process.	Speedup reported. Discussion is minimal or generic.	Speedup missing or implausible.	Missing or fails to implement JAX.
Section 5: Reflection	Specific, empirically-grounded observations on Learning Rate sensitivity (e.g., divergence vs. slow convergence). Thoughtful reflection on the "Black Box" of sklearn.	Good observations on hyperparameters and libraries.	Generic reflection (e.g., "LR is important").	Minimal effort.	Missing.