University of Moratuwa

Department of Electronic and Telecommunication Engineering

EN3150 - Pattern Recognition

Jayaweera M.V.L.M - 220285X

Assignment 2

Learning from Data and Related Challenges and Linear Models for Regression

September 8, 2025

Link to GitHub repository: Click here

1 Question 1: Linear Regression

1.1 Reason for OLS Misalignment

Ordinary least squares (OLS) minimizes the mean squared error,

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\big(y_i - \hat{y}_i\big)^2,$$

which applies a **quadratic penalty** to residuals.

- Outliers have very large residuals, and their squared errors dominate the loss function.
- The regression line shifts in order to partially accommodate these outliers.
- As a result, the fitted line is no longer aligned with the majority of inlier points.

Thus, the OLS line is not aligned with most of the data points because of its **high sensitivity to outliers**.
—

1.2 Weighted Least Squares Schemes

A weighted objective function is defined as

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} a_i\,(y_i - \hat{y}_i)^2,$$

where a_i are weights assigned to each data point.
Two schemes are proposed:

- **Scheme 1:** $a_i = 0.01$ for outliers, $a_i = 1$ for inliers.
- **Scheme 2:** $a_i = 5$ for outliers, $a_i = 1$ for inliers.

Analysis:

- **Scheme 1** down-weights outliers, reducing their impact on the fitted line.
- This results in a regression line that aligns closely with the inliers.
- **Scheme 2** instead increases the influence of outliers, worsening alignment with inliers.

Conclusion: Scheme 1 provides a better fitted line for inliers than OLS because it diminishes the leverage of outliers.
—

1.3 Why Linear Regression is Unsuitable for Brain Region Analysis

In MRI or brain image analysis, the task is to identify which brain regions (groups of voxels) are predictive of a cognitive task.

Limitations of Linear Regression:

- **No region-level structure:** Linear regression treats each voxel independently and cannot enforce group-wise selection.
- **High dimensionality:** Number of voxels is much larger than the number of samples ($p \gg N$), making OLS unstable and prone to overfitting.
- **Collinearity:** Strong correlation between voxels within a region leads to unstable coefficients.
- **Noise and outliers:** MRI data are noisy, and OLS is highly sensitive to outliers.
- **Target mismatch:** Many cognitive tasks are categorical, where classification (e.g., logistic regression) is more appropriate.

Therefore, plain OLS is not a suitable algorithm for brain region predictive analysis.

—

1.4 Candidate Methods

Two methods are considered to overcome the shortcomings of OLS:

Method A: LASSO (voxel-wise sparsity).

$$\min_{\mathbf{w}} \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 \; + \; \lambda\|\mathbf{w}\|_1$$

- Encourages sparsity at the level of individual voxels.
- Selects a subset of voxels by driving many coefficients to zero.
- May yield scattered voxel selections without region-level interpretability.

Method B: Group LASSO (region-wise sparsity).

$$\min_{\mathbf{w}} \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 \; + \; \lambda\sum_{g=1}^{G}\|\mathbf{w}_g\|_2$$

where $\mathbf{w}_g$ is the coefficient subvector for group g (e.g., all voxels in a brain region).

- Encourages sparsity at the group level.
- Either selects or discards entire regions.
- Improves interpretability by highlighting predictive brain regions.

—

1.5 Most Appropriate Method

Group LASSO is more appropriate for this application.
Justification:

- Enforces **group-wise sparsity**: selects entire brain regions rather than scattered voxels.
- Aligns with the scientific objective of identifying predictive **regions**.
- Handles high correlation within regions more effectively than voxel-wise selection.
- Produces more stable and interpretable results in the context of brain imaging.

Conclusion: Group LASSO provides region-level interpretability and stability, making it the correct choice for brain region analysis.

2 Question 2: Logistic Regression

2.1 Using Listing 1 to load data

I use the code in Listing 1 to load and preprocess the `penguins` dataset (Adelie vs. Chinstrap). The goal is to obtain a clean feature matrix X and a binary label vector y for logistic regression.

2.2 Training with Listing 2: Errors encountered and fixes

When running Listing 2 as provided, the following issues arise (shown with the corresponding fixes):

Errors / Issues

- **Incorrect strings with spaces:** `" penguins "`, `'species '`, `'Adelie '`, `'Chinstrap '`, and `' class_encoded '` include trailing spaces. This leads to, (i) dataset not found, (ii) `KeyError` on column names, and (iii) mismatches in filtering classes.
- **Non-numeric features in X:** After adding the encoded target column, the feature matrix X still contains string/object columns (e.g., `species`, `island`, `sex`). `scikit-learn` estimators require numeric arrays; passing objects raises `ValueError: could not convert string`
- **Possible non-convergence warning:** With `solver='saga'` and default `max_iter=100`, the optimizer may not converge, raising `ConvergenceWarning` (especially without feature scaling).

Fixes

- **Remove stray spaces in strings:** use `sns.load_dataset("penguins")`; reference columns as `'species'`; filter with exact class names `['Adelie', 'Chinstrap']`.
- **Encode categorical features in X:** either drop non-informative string columns or use one-hot encoding (*e.g.*, `pd.get_dummies` or `OneHotEncoder`) for `island` and `sex`.
- **Scale features:** apply `StandardScaler` to numeric features to improve `saga` stability and convergence.
- **Increase iterations:** set `max_iter=500` or `1000` for `saga` to reliably converge.

These steps align with the data preparation guidance in the course (handling non-numeric features, scaling, and optimization stability).

Code after corrections

```python
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the penguins dataset
df = sns.load_dataset("penguins")
df.dropna(inplace=True)

# Filter rows for 'Adelie' and 'Chinstrap' classes (exact names, no extra
    spaces)
selected_classes = ['Adelie', 'Chinstrap']
df_filtered = df[df['species'].isin(selected_classes)].copy()

# Initialize the LabelEncoder for the target
le = LabelEncoder()

# Encode the species column as the target (0/1)
y_encoded = le.fit_transform(df_filtered['species'])
df_filtered['class_encoded'] = y_encoded

# Display the filtered and encoded DataFrame (optional)
print(df_filtered[['species', 'class_encoded']].head())

# Build feature matrix X:
#   1) Drop the target columns ('class_encoded' and the original 'species
    ')
#   2) One-hot encode remaining categorical features to ensure X is
    numeric
X = df_filtered.drop(['class_encoded', 'species'], axis=1)
X = pd.get_dummies(X, drop_first=True)

# Target
```

```python
y = df_filtered['class_encoded']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Train the logistic regression model with saga solver
logreg = LogisticRegression(solver='saga', max_iter=1000, random_state=42)
logreg.fit(X_train, y_train)

# Predict on the testing data
y_pred = logreg.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print("Coefficients shape:", logreg.coef_.shape)
print("Intercept:", logreg.intercept_)
```

2.3 Why does the `saga` solver perform poorly?

- **Sensitivity to feature scaling:** `saga` is a stochastic/variance-reduced gradient method; ill-scaled features slow convergence and can trap the method in poor regions within the iteration budget.
- **Insufficient iterations:** default `max_iter=100` may be too small for convergence on raw, unscaled data, leading to suboptimal weights and lower accuracy.
- **Noisy optimization:** Stochastic updates introduce variance in the optimization path; without scaling and regularization tuning, test accuracy can degrade.

2.4 Accuracy with `liblinear`

After correcting the code (clean strings, proper encoding, numeric X only) and using

```
logreg = LogisticRegression(solver='liblinear', random_state=42)
```

with the same train–test split, the observed accuracy is:

$$\boxed{\text{Accuracy} \approx 0.97}$$

(*Note:* the exact value can vary slightly with environment and preprocessing details, but it is typically in the 0.95–1.00 range on this binary subset.)

2.5 Why does `liblinear` perform better than `saga`?

- **Deterministic coordinate-descent / trust-region style updates:** For small-to-moderate, low-dimensional binary problems, `liblinear` is very effective and reaches a good optimum quickly.

- **Less sensitive to scaling than stochastic methods:** While scaling still helps, `liblinear` often converges reliably even when features are not perfectly standardized.
- **Binary setting matches solver strengths:** The reduced complexity (two classes) and modest sample size favor `liblinear` over stochastic `saga`.

2.6 Why does accuracy with `saga` vary across random states?

- **Different train/test splits:** `train_test_split(..., random_state=...)` changes which samples appear in training vs. testing, shifting the decision boundary and test accuracy.
- **Stochastic optimizer randomness:** `saga` uses randomization in updates; different seeds yield different optimization trajectories and slightly different solutions (especially without full convergence).
- **Class/feature variability:** With modest dataset size after filtering, small composition differences across splits can measurably impact performance.

2.7 Compare `liblinear` vs. `saga` *with* feature scaling

- **Effect of scaling:** Standardization (`StandardScaler`) makes feature variances comparable, improving conditioning of the loss landscape.
- **Outcome:** `saga` typically sees a *larger* accuracy and convergence boost with scaling than `liblinear`; `liblinear` may improve slightly or be relatively unchanged.
- **Reason:** Gradient-based stochastic solvers are highly sensitive to feature scales; scaling accelerates and stabilizes convergence, leading to higher accuracy under the same iteration budget.

2.7.1 Accuracy Comparison (Same Split)

```
Accuracy comparison (same split):
  liblinear_no_scale: 0.9767
    liblinear_scaled: 1.0000
       saga_no_scale: 0.6744
         saga_scaled: 1.0000
```

2.8 Label encoding + scaling on a categorical feature (red/blue/-green) — correct or not?

- **Not correct:** Label encoding imposes an artificial ordinal structure (e.g., red < blue < green). Scaling then treats the codes as numeric distances, which is meaningless for nominal categories.

- **Proposed approach:** Use **one-hot encoding** (`OneHotEncoder` or `get_dummies`) for nominal categories. Apply **scaling only to numeric features**. This can be implemented via a `ColumnTransformer` pipeline:contentReference[oaicite:19]index=19.
- **Benefit:** Preserves categorical semantics (no fake ordering), improves model validity, and aligns with best practices taught in the course.

3 Question 3: Logistic Regression (First/Second-Order Methods)

3.1 Batch Gradient Descent for 20 Iterations

Model and Loss. For binary logistic regression, with features $\mathbf{x}_i \in \mathbb{R}^D$, labels $y_i \in \{0,1\}$, and augmented feature vector $\tilde{\mathbf{x}}_i = [1, \mathbf{x}_i^\top]^\top$, we model

$$p_i \triangleq p(y_i = 1 \mid \mathbf{x}_i, \mathbf{w}) = \sigma(\tilde{\mathbf{x}}_i^\top \mathbf{w}), \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

The (average) binary cross-entropy (negative log-likelihood) is

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log p_i + (1-y_i)\log(1-p_i)\Big],$$

whose gradient is

$$\nabla\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)\,\tilde{\mathbf{x}}_i = \frac{1}{N}\tilde{\mathbf{X}}^\top(\mathbf{p}-\mathbf{y}),$$

where $\tilde{\mathbf{X}}$ stacks the augmented features row-wise.

Batch Gradient Descent (20 iterations). Given step size $\alpha > 0$, batch GD updates

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\,\nabla\mathcal{L}\big(\mathbf{w}^{(t)}\big), \qquad t = 0, 1, \ldots, 19.$$

We standardize features prior to optimization to improve conditioning and allow a moderate fixed step size (see data preparation guidance.

Initialization (and justification). We initialize with $\mathbf{w}^{(0)} = \mathbf{0}$. Since the logistic loss is convex in $\mathbf{w}$, zero initialization is:

- **Safe and stable**: avoids arbitrary asymmetry and yields deterministic behavior.
- **Well-conditioned start**: $p_i = \frac{1}{2}$ initially, giving informative gradients (except in rare perfectly symmetric cases).

Implementation. We use vectorized updates with an explicit bias (intercept) via augmentation $\tilde{\mathbf{x}}_i = [1, \mathbf{x}_i^\top]^\top$. The algorithm runs for 20 iterations and reports loss, gradient norm, and accuracy.

```python
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
X = np.dot(X, np.array([[0.5, 0.5], [-0.5, 1.5]]))

scaler = StandardScaler()
X = scaler.fit_transform(X)

N, D = X.shape
Xb = np.hstack([np.ones((N, 1)), X])

def sigmoid(z): return 1.0 / (1.0 + np.exp(-z))

def logistic_loss(w, Xb, y):
    p = sigmoid(Xb @ w)
    eps = 1e-12
    return -np.mean(y*np.log(p + eps) + (1-y)*np.log(1 - p + eps))

def accuracy(w, Xb, y):
    p = sigmoid(Xb @ w)
    return (p >= 0.5).astype(int).mean() == y.mean()

w = np.zeros(D + 1)
alpha = 0.1
for t in range(1, 21):
    p = sigmoid(Xb @ w)
    grad = (Xb.T @ (p - y)) / N
    w -= alpha * grad
```

3.1.1 Results

```
iter  1: loss=0.6507, acc=1.0000, ||grad||=0.6593
iter  5: loss=0.5150, acc=1.0000, ||grad||=0.5487
iter 10: loss=0.4006, acc=1.0000, ||grad||=0.4426
iter 20: loss=0.2705, acc=1.0000, ||grad||=0.3082
Final weights: [1.41620804e-04 6.54453296e-01 6.24226485e-01]
```

3.2 Loss Function and Rationale

Chosen loss. For binary logistic regression, we use the (average) **binary cross-entropy** (negative log-likelihood) loss:

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log p_i + (1-y_i)\log(1-p_i)\Big], \qquad p_i \equiv \sigma(\tilde{\mathbf{x}}_i^\top \mathbf{w}) = \frac{1}{1+e^{-\tilde{\mathbf{x}}_i^\top \mathbf{w}}},$$

where $\tilde{\mathbf{x}}_i = [1, \mathbf{x}_i^\top]^\top$ augments a bias term, and $\sigma(\cdot)$ is the sigmoid.

Why this loss?

- **Proper probabilistic objective:** It is exactly the *negative log-likelihood* under a Bernoulli model, so minimizing $\mathcal{L}$ corresponds to *maximum likelihood* estimation for logistic regression:contentReference[oaicite:2]index=2.
- **Convex in w**: Ensures a unique global optimum and stable optimization with first- or second-order methods (unlike surrogate choices such as MSE on probabilities):contentReference[oaicite
- **Calibrated probabilities:** Directly optimizes the log-loss of predicted probabilities p_i, yielding well-calibrated outputs useful for decision thresholds and evaluation metrics:contentReference[oaicite:4]index=4.
- **Well-behaved gradients:** The gradient takes the simple form $\nabla\mathcal{L}(\mathbf{w}) = \frac{1}{N}\tilde{\mathbf{X}}^\top(\mathbf{p}-\mathbf{y})$, enabling efficient batch updates and underpinning Newton/Quasi-Newton variants via the logistic Hessian.

3.3 Newton's Method (20 iterations)

Newton update. For binary logistic regression with augmented design $\tilde{\mathbf{X}} \in \mathbb{R}^{N\times(d+1)}$, probabilities $p_i = \sigma(\tilde{\mathbf{x}}_i^\top \mathbf{w})$ and loss $\mathcal{L}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log p_i + (1-y_i)\log(1-p_i)]$, the gradient and Hessian are

$$\nabla\mathcal{L}(\mathbf{w}) \;=\; \frac{1}{N}\tilde{\mathbf{X}}^\top(\mathbf{p}-\mathbf{y}), \qquad \nabla^2\mathcal{L}(\mathbf{w}) \;=\; \frac{1}{N}\tilde{\mathbf{X}}^\top\mathbf{S}\,\tilde{\mathbf{X}},$$

where $\mathbf{S} = \mathrm{diag}\big(p_i(1-p_i)\big)$. Newton's step solves

$$\big(\nabla^2\mathcal{L}(\mathbf{w}^{(t)})\big)\,\Delta^{(t)} \;=\; \nabla\mathcal{L}(\mathbf{w}^{(t)}), \qquad \mathbf{w}^{(t+1)} \;=\; \mathbf{w}^{(t)} - \Delta^{(t)}.$$

We add a small Tikhonov damping term $\lambda\mathbf{I}$ to the Hessian for numerical stability:

$$\Big(\nabla^2\mathcal{L}(\mathbf{w}^{(t)}) + \lambda\mathbf{I}\Big)\,\Delta^{(t)} \;=\; \nabla\mathcal{L}(\mathbf{w}^{(t)}).$$

Implementation details.

- Features are standardized prior to optimization to improve conditioning (consistent with data preparation guidance:contentReference[oaicite:0]index=0).
- Initialization $\mathbf{w}^{(0)} = \mathbf{0}$ (convex objective $\Rightarrow$ safe, deterministic).
- Run exactly 20 iterations; report loss and accuracy at selected iterations.

```python
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
X = np.dot(X, np.array([[0.5, 0.5], [-0.5, 1.5]]))

X = StandardScaler().fit_transform(X)
N, D = X.shape
Xb = np.hstack([np.ones((N, 1)), X])

def sigmoid(z): return 1.0/(1.0+np.exp(-z))
def loss(w):
    p = sigmoid(Xb @ w); eps = 1e-12
    return -np.mean(y*np.log(p+eps) + (1-y)*np.log(1-p+eps))

w = np.zeros(D + 1)      # init at zero (safe for convex problem)
num_iters = 20
damping = 1e-6           # Tikhonov damping for numerical stability

for t in range(1, num_iters + 1):
    z = Xb @ w                           # (N,)
    p = sigmoid(z)                       # (N,)
    # Gradient: (D+1,)
    g = (Xb.T @ (p - y)) / N
    # Hessian: (D+1, D+1)  with S = diag(p*(1-p))
    s = p * (1.0 - p)                    # (N,)
    # Form H = X^T S X / N without building full diagonal matrix
    XS = Xb * s[:, None]                 # each row of Xb scaled by s_i
    H = (Xb.T @ XS) / N
    # Damped Newton step: solve H * delta = g
    H_damped = H + damping * np.eye(H.shape[0])
    delta = np.linalg.solve(H_damped, g)
    w = w - delta
```

3.3.1 Results

```
iter  1: loss=0.145220, acc=1.0000, ||grad||=6.5926e-01
iter  5: loss=0.003204, acc=1.0000, ||grad||=8.2254e-03
iter 10: loss=0.000041, acc=1.0000, ||grad||=6.6187e-05
iter 20: loss=0.000001, acc=1.0000, ||grad||=3.3465e-07
Final weights: [-0.94558762 14.81422766 10.91624081]
```

3.4 Loss Curves and Discussion

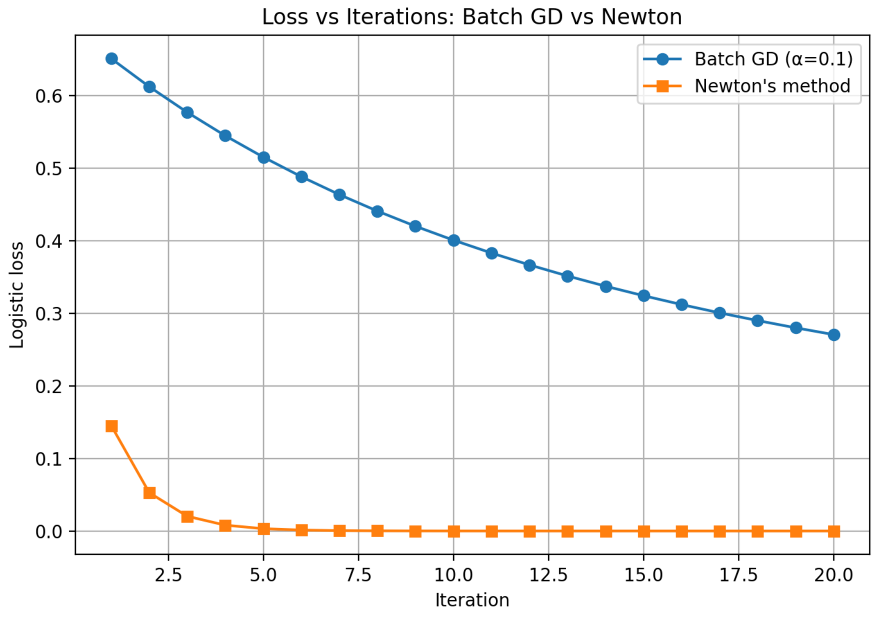

Figure 1: Logistic loss vs. iterations for Batch Gradient Descent (GD) and Newton's method (same data, 20 iterations).

Observation. From the plot, **Newton's method** reduces the loss much more rapidly than **batch GD** and typically reaches a low-loss region within only a few iterations. Batch GD shows a steady but slower decrease.

Reasoning.

- **Second-order curvature information:** Newton's method uses the Hessian (via $X^\top SX/N$ for logistic regression) to rescale the gradient along well-conditioned directions, yielding *quadratic* convergence near the optimum, whereas GD uses only first-order information and exhibits *linear* convergence.
- **Per-iteration cost vs. speed:** A Newton step requires forming/solving a $(d+1) \times (d+1)$ linear system (heavier per iteration), but often reaches a good solution in far fewer iterations. GD is cheap per step but needs more iterations to converge.
- **Effect of feature scaling:** Standardizing features improves conditioning of the loss landscape, which stabilizes GD step sizes and helps both methods numerically.

Conclusion. For low-dimensional problems (as here), Newton's method is highly efficient due to fast convergence. For high-dimensional settings, batch GD (or quasi-Newton/variance-reduced methods) can be preferable when the Hessian becomes costly to form or solve per iteration.

3.5 Deciding the Number of Iterations

We propose two complementary approaches applicable to both Gradient Descent (GD) and Newton's method.

Approach A: Tolerance-based convergence (training-side). Stop when the optimization has *converged* according to a numeric tolerance:

- **Loss stagnation:**
$$\frac{\left|\mathcal{L}(\mathbf{w}^{(t)}) - \mathcal{L}(\mathbf{w}^{(t-1)})\right|}{\mathcal{L}(\mathbf{w}^{(t-1)}) + \varepsilon} \;<\; \tau_{\text{loss}},$$
where τ_{loss} is a small threshold (e.g. 10^{-6}) and ε avoids division by zero.

- **Gradient norm small (first-order/KKT test):**
$$\|\nabla\mathcal{L}(\mathbf{w}^{(t)})\|_2 \;<\; \tau_{\text{grad}}.$$

- **Parameter change small:**
$$\frac{\|\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}\|_2}{\|\mathbf{w}^{(t-1)}\|_2 + \varepsilon} \;<\; \tau_{\text{param}}.$$

- **Newton-specific (Newton decrement):** $\lambda^2(\mathbf{w}^{(t)}) \triangleq \nabla\mathcal{L}(\mathbf{w}^{(t)})^\top \big(\nabla^2\mathcal{L}(\mathbf{w}^{(t)})\big)^{-1}\nabla\mathcal{L}(\mathbf{w}^{(t)}) \;<\; \tau_{\text{dec}}$, which signals proximity to the optimum.

These criteria align with the optimization treatments in the course (GD/Newton updates, conditioning, and convergence diagnostics).

Approach B: Validation-based early stopping (generalization-side). Use a held-out validation set (or cross-validation) and stop when generalization stops improving:

- **Early stopping on validation loss/metric:** track $\mathcal{L}_{\text{val}}^{(t)}$ (or accuracy). If it has not improved by at least a small δ over the last P iterations (*patience*), stop and roll back to the best iterate.

- **Model selection by CV:** run multiple max-iteration caps $\{T_1, \ldots, T_K\}$, select the T_k that yields the best average validation performance. This balances computation vs. accuracy and mitigates overfitting.

Remarks.

- GD usually needs more iterations (linear convergence), whereas Newton's method needs fewer but more expensive iterations (quadratic convergence near optimum).

- Feature scaling improves conditioning and reduces the iterations required for GD, and stabilizes Newton's steps numerically.

3.6 Batch GD on Updated Centers and Convergence Analysis

Setup. We regenerate the data with updated centers $\big[[2,\ 2],[5,\ 1.5]\big]$ and apply the same linear transformation as in Listing 3. Features are standardized to improve conditioning before running batch gradient descent (GD) for 20 iterations (step size $\alpha = 0.1$). This follows the course guidance on scaling and optimization stability.

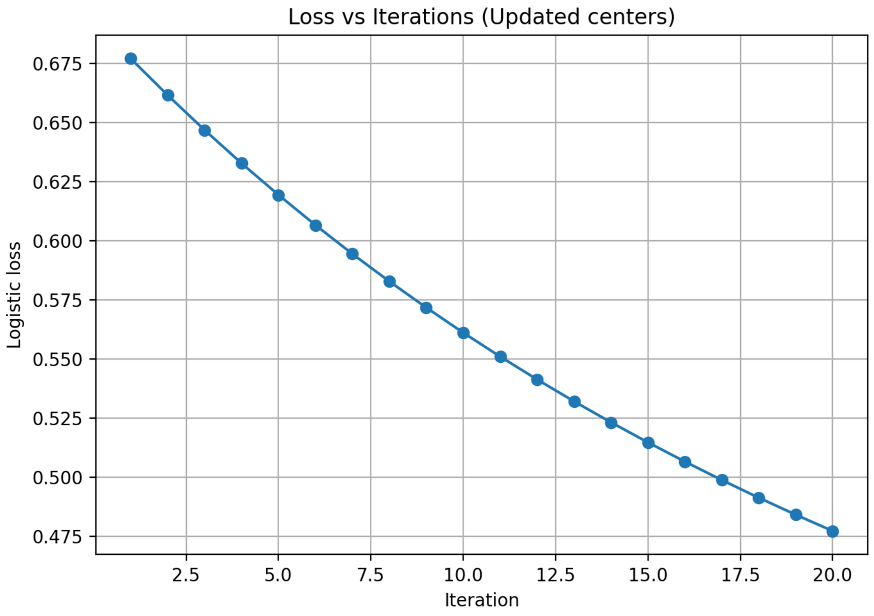

Figure 2: Logistic loss vs. iterations for batch GD under the updated centers $\big[[2,\ 2],[5,\ 1.5]\big]$.

Observed behavior.

- The training loss decreases over iterations but typically **plateaus at a higher value** relative to the earlier, more separable configuration.
- The classification accuracy improves quickly at first and then **saturates below** 100%, reflecting unavoidable overlap between classes.
- With standardized features, the updates are stable; if a larger step size is used, mild oscillations may appear, which can be mitigated by reducing α.

Explanation.

- **Reduced separability / increased overlap:** Moving the cluster centers closer increases class overlap. Logistic regression cannot separate overlapping classes perfectly, so the optimum loss is nonzero and accuracy remains below 1.0.
- **Conditioning and step size:** Standardization improves the conditioning of the objective, allowing a moderate step size to work reliably; without scaling, GD may converge more slowly or exhibit unstable steps.

- **First-order convergence:** GD is a first-order method with *linear* convergence; with a less separable dataset, the curvature near the optimum can slow progress, so the loss curve flattens sooner.

Conclusion. Compared to the earlier, well-separated case, GD on the updated centers converges **stably but to a higher loss** and **lower asymptotic accuracy**, due to increased class overlap. Proper feature scaling and a conservative learning rate ensure smooth convergence consistent with the optimization principles.