UNIVERSITY OF MORATUWA

Faculty of Engineering
Department of Electronic and Telecommunication Engineering

EN3150
Machine Learning for Communication Systems

Assignment 02

Learning from Data and Related Challenges and Classification

Submitted by:

Rivindu Vinsara Kumarage
Index No: 220343B

September 9, 2025

Contents

1 Linear Regression

Question 1 - OLS Line Alignment Issue

Analysis of OLS Misalignment

The Ordinary Least Squares (OLS) fitted line shown in Figure 1 is not aligned with the majority of data points due to the presence of **outliers** in the dataset.
Key Points:

- OLS minimizes the sum of squared residuals: $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- The squared term gives disproportionate weight to large deviations (outliers)
- Outliers have a leverage effect, pulling the regression line towards them
- This results in a line that may not represent the true underlying relationship for the majority of the data

Mathematical Explanation: Since OLS uses squared errors, points far from the true line contribute quadratically to the loss function. For example, if a point has an error of 10 units, it contributes 100 to the loss, while a point with error 1 contributes only 1. This makes the algorithm prioritize reducing large errors over small ones, causing the fitted line to be biased towards outliers.

Question 2 - Modified Loss Function Analysis

Scheme Comparison

Scheme 1 will provide a better fitted line for inliers compared to the original OLS.
Analysis of Both Schemes:
Scheme 1: Outlier weight = 0.01, Inlier weight = 1

- Effectively reduces the influence of outliers by 99%
- Modified loss: $\frac{1}{N}\sum_{i=1}^{N} a_i(y_i - \hat{y}_i)^2$
- For outliers: $a_i = 0.01 \rightarrow$ minimal contribution to loss
- For inliers: $a_i = 1 \rightarrow$ normal contribution
- Result: Line fits primarily to inlier data

Scheme 2: Outlier weight = 5, Inlier weight = 1

- **Increases** the influence of outliers by $5\times$
- Makes the outlier problem worse than original OLS

- Line will be even more biased towards outliers
- Performs worse than standard OLS

Mathematical Justification: In Scheme 1, the optimization becomes:

$$\min_w \frac{1}{N}\left[\sum_{i\in\text{inliers}}(y_i-\hat{y}_i)^2+0.01\sum_{i\in\text{outliers}}(y_i-\hat{y}_i)^2\right]$$

This effectively ignores outliers and fits primarily to inlier data, resulting in better alignment with the majority of points.

Question 3 - Linear Regression Limitations in Brain Analysis

Why Linear Regression is Unsuitable

Linear regression is not suitable for identifying predictive brain regions due to several critical limitations:

1. High Dimensionality Problem

- Brain images contain thousands of voxels (features)
- Number of features >> number of samples (curse of dimensionality)
- Standard linear regression cannot handle $p > n$ scenarios
- Results in overfitting and poor generalization

2. Multicollinearity Issues

- Adjacent voxels are highly correlated
- Creates unstable coefficient estimates
- Makes interpretation of individual voxel importance unreliable

3. Feature Selection Problem

- Cannot perform automatic feature selection
- All voxels receive non-zero weights
- Cannot identify specific brain regions as predictive
- Lacks sparsity in the solution

4. Interpretability Challenges

- Cannot group voxels into meaningful brain regions
- Individual voxel weights may not reflect regional importance
- Lacks biological/anatomical interpretability

5. Regularization / Biased Estimators

- OLS is undefined when $X^\top X$ is singular; regularized estimators (ridge/L2, lasso/L1, elastic net) provide stable biased solutions.
- Lasso promotes voxel-level sparsity; *Group Lasso* or region-wise penalties allow selection of entire brain regions (groups of voxels).

6. Spatial and Temporal Dependencies

- Voxels are spatially correlated and fMRI time series exhibit temporal autocorrelation, violating the i.i.d. assumption of OLS.
- These dependencies require pre-whitening or models that explicitly model temporal/spatial covariance; otherwise coefficient estimates and p-values are biased.

7. Low SNR and Structured Noise

- Neural signals are weak and easily masked by motion, respiration, cardiac pulsation and scanner artifacts.
- Without denoising/regressing out confounds, estimated weights may reflect nuisance variation rather than task-related activation.

8. Multiple Comparisons and Inference

- Mass-univariate testing across thousands of voxels inflates false positive rates.
- Proper inference needs family-wise/FDR correction or nonparametric permutation/cluster-level tests to control error rates.

9. Stability and Model Validation

- Feature selection from high-dimensional data is unstable; use cross-validation, stability selection, and reporting of selection frequencies.
- Permutation tests and out-of-sample validation improve confidence in identified regions.

10. Computational and Practical Considerations

- High-dimensional fitting plus resampling (CV, permutation) is computationally intensive.
- Dimensionality reduction (ROI averaging, PCA) or sparse/group penalties reduce computational load and improve interpretability.

Statistical Consequences: The design matrix $X \in \mathbb{R}^{N\times p}$ with $p >> N$ makes $X^T X$ singular, preventing computation of $(X^T X)^{-1}$ needed for the closed-form solution.

Question 4 & 5 - LASSO vs Group LASSO Comparison

Method Selection and Justification

Group LASSO (Method B) is more appropriate for brain region identification.
Detailed Comparison:
Standard LASSO (Method A):

- Objective: $\min_w \left\{ \frac{1}{N} \sum_{i=1}^{N} (y_i - w^T x_i)^2 + \lambda \|w\|_1 \right\}$
- Performs individual voxel selection
- May select scattered voxels across brain regions
- Lacks spatial/anatomical coherence
- Cannot identify entire brain regions as predictive units

Group LASSO (Method B):

- Objective: $\min_w \left\{ \frac{1}{N} \sum_{i=1}^{N} (y_i - w^T x_i)^2 + \lambda \sum_{g=1}^{G} \|w_g\|_2 \right\}$
- Performs region-level selection (groups of voxels)
- Either selects entire brain regions or excludes them completely
- Maintains spatial/anatomical coherence
- Directly addresses the research question

Why Group LASSO is Superior:
1. Biological Relevance

- Brain functions are localized to specific regions
- Entire regions, not individual voxels, are functionally meaningful
- Group LASSO respects anatomical structure

2. Interpretability

- Directly identifies which brain regions are predictive
- Results are clinically/scientifically interpretable
- Facilitates hypothesis generation for neuroscience research

3. Statistical Advantages

- Reduces effective dimensionality from voxels to regions
- More stable feature selection

- Better handling of within-region correlation
- Improved prediction performance on new subjects

4. **Practical caveats / implementation notes**

- **Sparse-group option:** If only some voxels inside a region are predictive, use Sparse-Group LASSO (L1 + group L2) to allow group + within-group sparsity.
- **Overlapping groups:** Anatomical/functional atlases may overlap; standard Group LASSO assumes disjoint groups — use overlapping-group methods if needed.
- **Group-size weighting:** Penalise groups proportionally (e.g. weight by $\sqrt{|g|}$) to avoid bias toward small/large groups.
- **Tuning & validation:** Select λ (and group weights) with nested cross-validation; report out-of-sample performance and selection stability.
- **Preprocessing & confounds:** Standardize voxels, regress out motion/CSF/white-matter, and prewhiten temporal autocorrelation before fitting.
- **Inference & stability:** Penalised methods do not provide simple p-values — use permutation/bootstraps or selective-inference and report selection frequencies (stability selection).
- **Computation:** Group penalties need group-prox algorithms and resampling (CV/permutations) — mention computational cost if using heavy resampling.

Mathematical Insight: The ℓ_2 norm penalty $\|w_g\|_2$ in Group LASSO creates an "all-or-nothing" selection mechanism for each group g. When λ is sufficiently large, entire groups are set to zero, effectively performing region-level feature selection that aligns with the research objective.

2 Logistic Regression

Question 1 - Data Loading

Data Loading

The provided code successfully loads the penguins dataset and preprocesses it for binary classification between 'Adelie' and 'Chinstrap' species.

Code Analysis:

```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 and preprocess data
df = sns.load_dataset("penguins")
df.dropna(inplace=True)

# Filter for binary classification
selected_classes = ['Adelie', 'Chinstrap']
df_filtered = df[df['species'].isin(selected_classes)].copy()

# Encode labels
le = LabelEncoder()
y_encoded = le.fit_transform(df_filtered['species'])
df_filtered['class_encoded'] = y_encoded
# Display the filtered and encoded DataFrame
print(df_filtered[['species', 'class_encoded']])
# Split the data into features (X) and target variable (y)
y = df_filtered['class_encoded'] # Target variable
X = df_filtered.drop(['class_encoded'], axis=1)
```

Listing 1: Data Loading and Preprocessing

Question 2 - Training Errors with SAGA Solver

SAGA Solver Issues

Problems observed in the copied code (brief):

- `X` contained non-numeric/text columns (e.g. `species`, `island`, `sex`) which caused a `ValueError: could not convert string to float: 'Adelie'` when calling `.fit()`.

- The provided listings included trailing spaces in column names/values (e.g. `'species '` / `'Adelie '`), causing mismatches and extra string values.
- Using `LogisticRegression(solver='saga')` on unscaled or ill-preprocessed data produced `ConvergenceWarning` and unstable behaviour.

Minimal in-place edits:

```python
# REMOVE
# X = df_filtered.drop(['class_encoded'], axis=1)

# ADD
# 1) strip stray whitespace from column names and string values
    (important)
df.columns = df.columns.str.strip()
df = df.apply(lambda s: s.str.strip() if s.dtype == "object"
    else s)

# 2) define X and y clearly and select numeric predictors only (
    quick robust fix)
y = df_filtered['class_encoded']
# option A: explicit numeric selection
X = df_filtered[['bill_length_mm','bill_depth_mm','
    flipper_length_mm','body_mass_g']].copy()
# option B: auto-select numeric columns (drops any remaining
    string cols)
# import numpy as np
# X = df_filtered.select_dtypes(include=[np.number]).drop(
    columns=['class_encoded'])
```

Listing 2: Q2: Fix feature selection

Fix for training with SAGA (preprocessing pipeline, scaling and enough iterations):

```python
# REMOVE
# logreg = LogisticRegression(solver='saga')
# logreg.fit(X_train, y_train)
# y_pred = logreg.predict(X_test)
# accuracy = accuracy_score(y_test, y_pred)

# ADD
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

```

```
# create X,y if not already done (use the cleaned df_filtered)
X = df_filtered.drop(columns=['class_encoded'])
y = df_filtered['class_encoded']

# stratified split (stable single-split evaluation)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# identify numeric and categorical columns after stripping
   whitespace
num_cols = X_train.select_dtypes(include=[np.number]).columns.
   tolist()
cat_cols = X_train.select_dtypes(exclude=[np.number]).columns.
   tolist()

# preprocessor: scale numeric, one-hot encode categorical
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(drop='first', sparse=False), cat_cols)
])

# pipeline: preprocessor + saga classifier with more iterations
pipe_saga = Pipeline([
    ('pre', preprocessor),
    ('clf', LogisticRegression(solver='saga', max_iter=5000,
        random_state=42))
])

# fit and evaluate
pipe_saga.fit(X_train, y_train)
y_pred = pipe_saga.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("SAGA␣(pipeline)␣accuracy:", accuracy)
```

Listing 3: Q2: Replace raw saga fit with ColumnTransformer + Pipeline

Observed notebook outputs:

```
species
Adelie       146
Chinstrap     68
Name: count, dtype: int64

Features used: ['bill_length_mm', 'bill_depth_mm', '
   flipper_length_mm', 'body_mass_g']
Class mapping: {'Adelie': np.int64(0), 'Chinstrap': np.int64(1)}
```

Listing 4: Notebook outputs: initial diagnostics

Explanation: `saga` is gradient-based and sensitive to feature scaling and to string-valued predictors. First strip stray whitespace and remove/encode text columns so the model receives numeric input; then use a `ColumnTransformer` (scale numeric, one-hot encode categorical) inside a `Pipeline` and increase `max_iter` to avoid convergence warnings and the conversion error.

Question 3 - SAGA Solver Performance Issues

Why SAGA Performs Poorly

Correction to include (code snippet):

```
# REPLACE any direct fit on X_train with explicit scaling + fit
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s  = scaler.transform(X_test)

logreg = LogisticRegression(solver='saga', max_iter=5000,
   random_state=42)
logreg.fit(X_train_s, y_train)

# evaluation
y_pred = logreg.predict(X_test_s)
accuracy = accuracy_score(y_test, y_pred)
print("SAGA␣accuracy␣(scaled):", accuracy)
```

Listing 5: Q3: If arrays are used

Observed notebook outputs (SAGA / coefficients / warnings):

```
SAGA Accuracy: 0.6976744186046512
Coefficients: [[ 0.05301513 -0.00259315 -0.0019319
   -0.00068938]] Intercept: [-0.00019593]
d:\anaconda3\envs\ML\lib\site-packages\sklearn\linear_model\_sag
   .py:348: ConvergenceWarning: The max_iter was reached which
   means the coef_ did not converge
  warnings.warn(
(Repeated ConvergenceWarning messages across runs)
```

Listing 6: Notebook outputs: SAGA raw results

Concise reason: SAGA is stochastic and requires well-conditioned (scaled) features to converge quickly. On small datasets its stochastic updates can have high variance — scaling and sufficient iterations mitigate this.

Question 4 - LibLinear Solver Performance

LibLinear Accuracy

Small code fix (ensure prediction before scoring):

```
# REMOVE (if present)
# accuracy = accuracy_score(y_test, y_pred)

# ADD
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

logreg = LogisticRegression(solver='liblinear', random_state=42)
logreg.fit(X_train, y_train)
y_pred = logreg.predict(X_test)   # <-- ensure prediction is
    computed
accuracy = accuracy_score(y_test, y_pred)
print("Liblinear␣accuracy:", accuracy)
```

Listing 7: Q4: Ensure y$_p$*redexists*

Observed notebook outputs (LibLinear):

```
LIBLINEAR Accuracy: 0.9767441860465116
```

Listing 8: Notebook outputs: LibLinear

Short note: LibLinear uses coordinate descent and is robust on small/medium datasets — hence higher accuracy on the penguins split.

Question 5 - LibLinear vs SAGA Comparison

Why LibLinear Outperforms SAGA

Replace any single-split solver comparison with this cross-validated, scaled comparison:

```
from sklearn.model_selection import cross_val_score
pipe_lib = Pipeline([('scaler', StandardScaler()),
                     ('clf', LogisticRegression(solver='
                        liblinear', random_state=42))])
pipe_saga = Pipeline([('scaler', StandardScaler()),
                      ('clf', LogisticRegression(solver='saga',
                         max_iter=5000, random_state=42))])

score_lib = cross_val_score(pipe_lib, X, y, cv=5).mean()
score_saga = cross_val_score(pipe_saga, X, y, cv=5).mean()
print("liblinear CV acc:", round(score_lib,3))
print("saga CV acc:", round(score_saga,3))
```

Listing 9: Q5: Solver comparison using pipeline + CV

Observed notebook summary across random states:

```
SAGA unscaled: 0.6976744186046512
LIBLINEAR unscaled: 0.9767441860465116
SAGA scaled:   1.0
LIBLINEAR scaled: 1.0

Saga: mean 0.697, std 0.020
Liblinear: mean 0.982, std 0.018
```

Listing 10: Notebook outputs: solver summary

One-line reason: LibLinear's deterministic coordinate updates are better conditioned for small datasets; SAGA needs proper scaling and more iterations, otherwise it underperforms.

Question 6 - Random State Variance

Random State Impact on SAGA Accuracy

The model's accuracy with the `saga` solver varies with different `random_state` values mainly because (1) different `random_state` values produce different train/test splits (small datasets are sensitive to which examples fall into the test set), and (2) `saga` is a stochastic, gradient-based solver whose convergence and final weights depend on initialization, sampling order and conditioning of the data. These two sources of randomness amplify each other and produce variable test accuracy.

Detailed explanation (concise):

- **Train/test split randomness:** Changing `random_state` in `train_test_split` changes which examples end up in the test set. On small datasets (here ~214 samples after filtering), some splits are "easier" than others — a few influential examples can noticeably raise or lower test accuracy.
- **Small-sample sensitivity:** With limited data, the estimator variance is high: the learned parameters depend strongly on the available training examples. This naturally increases the spread of measured accuracies across splits.
- **Stochastic optimizer effects:** `SAGA` uses stochastic gradient information (random sampling of examples). Different shuffles / initialization / internal state can lead to different optimization paths and, if not fully converged, different final coefficients and predictions.
- **Convergence instability:** When `saga` hits the `max_iter` limit (or when features are poorly scaled), it may stop before reaching the true optimum; stopping early at different internal states leads to inconsistent accuracy across runs.
- **Class imbalance and influential points:** If classes are imbalanced or there are outliers, which particular class examples appear in train vs test strongly affects measured performance.

Practical mitigation / what to report:

- **Use stratified splitting** so class proportions are preserved: `train_test_split(..., stratify=y)`.
- **Prefer k-fold (stratified) cross-validation** or repeated stratified k-fold to estimate performance robustly and report mean $\pm$ standard deviation rather than a single number.
- **Stabilize training:** scale numeric features (e.g. `StandardScaler`), increase `max_iter` for `saga`, and use a `Pipeline`/`ColumnTransformer` to avoid data-leakage.
- **If reporting single-split results**, fix `random_state` and mention it explicitly; but prefer CV for conclusions.

Reproducible experiment (one-line recipe) — compute mean ± std over repeated stratified splits:

```python
from sklearn.model_selection import cross_val_score,
   StratifiedKFold
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(pipe_saga, X, y, cv=cv)   # pipe_saga =
    pipeline with preprocessor + solver
print(f"mean={scores.mean():.3f},␣std={scores.std():.3f}")
```

Listing 11: Q6: Recommended experiment to show variance

Question 7 - Feature Scaling Impact

Scaling Comparison Analysis

Reproducible experiment (code):

```python
# Required imports (run in the notebook)
import numpy as np
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score,
   train_test_split

# Prepare X, y as cleaned in Q2 (X must not contain string
   columns)
# X = df_filtered.drop(columns=['class_encoded'])
# y = df_filtered['class_encoded']

# Identify numeric / categorical columns
num_cols = X.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = X.select_dtypes(exclude=[np.number]).columns.tolist()

# Preprocessor: scale numeric features only, one-hot encode
   categoricals
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(drop='first', sparse=False), cat_cols)
])

# Pipelines for the two solvers
pipe_saga = Pipeline([('pre', preprocessor),
                              ('clf', LogisticRegression(solver='saga',
                                  max_iter=5000, random_state=42))])
pipe_lib  = Pipeline([('pre', preprocessor),
                              ('clf', LogisticRegression(solver='
                                  liblinear', random_state=42))])

# Evaluate with stratified 5-fold CV (returns array of scores)
scores_saga = cross_val_score(pipe_saga, X, y, cv=5, scoring='
    accuracy')
scores_lib  = cross_val_score(pipe_lib,  X, y, cv=5, scoring='
    accuracy')

print("SAGA␣(scaled)␣:␣mean", scores_saga.mean(), "std",
    scores_saga.std())
print("LibLinear␣(scaled)␣:␣mean", scores_lib.mean(), "std",
    scores_lib.std())

# If you want single-split comparisons (already used in earlier
    notebook runs):
# X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42, stratify=y)
# pipe_saga.fit(X_train, y_train); pipe_lib.fit(X_train, y_train
    )
# print("SAGA (single split)    :", pipe_saga.score(X_test,
    y_test))
# print("LibLinear (single split):", pipe_lib.score(X_test,
    y_test))
```

Listing 12: Q7: ColumnTransformer + Pipeline experiment

Observed notebook results (from the runs):

```
Features used: ['bill_length_mm', 'bill_depth_mm', '
    flipper_length_mm', 'body_mass_g']
Class mapping: {'Adelie': 0, 'Chinstrap': 1}

Single-split outputs:
SAGA unscaled: 0.6976744186046512
LIBLINEAR unscaled: 0.9767441860465116
SAGA scaled:   1.0
LIBLINEAR scaled: 1.0

Cross-val summary:
SAGA (CV mean    std): 0.697    0.020
LibLinear (CV mean    std): 0.982    0.018
```

Listing 13: Notebook results (recorded outputs)

Interpretation (concise):

- **Why SAGA improves dramatically after scaling:** `saga` is a gradient-based (stochastic) optimizer. If features have very different magnitudes (e.g. millimetres vs grams), the optimization problem becomes ill-conditioned (elongated loss contours). Gradients are dominated by large-scale features, causing slow or unstable convergence. Standardizing numeric features to zero mean and unit variance equalizes their influence and stabilizes the stochastic updates, which explains the observed jump from $\sim$0.70 to 1.0 on the shown split.
- **Why LibLinear is less sensitive:** `liblinear` uses coordinate-descent / deterministic updates that are typically more robust on small datasets and less affected by feature scaling, so it already attains high accuracy without scaling (but still benefits slightly from scaling).
- **Single-split vs CV:** The perfect scores (1.0) observed on the shown single split reflect a particular train/test split and small sample size; cross-validation (mean $\pm$ std) gives a more reliable estimate (liblinear $\approx$0.982, saga $\approx$0.697 unscaled).

Practical notes / recommendations:

1. Use a `Pipeline` with a `ColumnTransformer` so numeric scaling and categorical one-hot encoding are applied correctly and without leakage.
2. **Do not** apply `StandardScaler` to one-hot (binary) columns — scale numeric columns only.
3. Report solver performance using stratified k-fold cross-validation (mean $\pm$ std) rather than a single train/test split to avoid overinterpreting split-specific results.
4. For small datasets, prefer `liblinear` for quick, stable results; use `saga` when you need its additional features (elastic-net, large-scale data) and ensure proper scaling and sufficient iterations.

Concluding sentence (for the report): The experiments show that feature scaling is essential when using `saga` (gradient-based) — it greatly improves convergence and accuracy — whereas `liblinear` is naturally more stable on small datasets but still benefits modestly from scaling.

Question 8 - Categorical Feature Encoding

Proper Categorical Feature Handling

Short answer: Incorrect. Applying `LabelEncoder` to a nominal predictor and then scaling it is wrong because label encoding introduces an artificial ordinal relationship (e.g. red=0, blue=1, green=2). Scaling preserves and amplifies that false ordering, which can mislead linear models (like logistic regression). Use **one-hot encoding** (or another nominal encoding such as binary/target/embedding methods for special cases) and scale numeric features only.

Why label-then-scale is wrong (concise):

- **Artificial ordinality:** Label encoder maps categories to integers, implying distances and order that do not exist for nominal categories.
- **Misleading geometry:** Scalers treat the encoded integers as continuous values, so model coefficients interpret meaningful numeric differences between categories.
- **Loss of interpretability & risk of bias:** Coefficients become hard to interpret and model may learn spurious linear trends.

Correct approaches (recommended):

1. **One-hot encoding (recommended for few categories):** Convert nominal categories into binary indicator columns (drop one to avoid perfect multicollinearity). Do NOT scale these one-hot columns; scale numeric features separately.
2. **Ordinal encoding only if an order exists:** If categories are ordinal (e.g. `low` < `med` < `high`) then label/ordinal encoding is acceptable.
3. **High-cardinality alternatives:** For many categories consider target encoding, frequency encoding, hashing, or learned embeddings (with caution and proper regularization / CV).

Minimal reproducible code (one-hot + scale numeric only):

```
# X_cat example: ['red','blue','green','blue','green']
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# example dataframe
df_small = pd.DataFrame({'color': ['red','blue','green','blue','
    green'],
                         'num':  [10, 50, 30, 40, 20]})

```

```python
# specify columns
num_cols = ['num']
cat_cols = ['color']

# ColumnTransformer: scale numeric, one-hot encode categorical
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(drop='first', sparse=False), cat_cols)
])

pipe = Pipeline([
    ('pre', preprocessor),
    ('clf', LogisticRegression())
])

# Use pipe.fit(X_train, y_train) as usual; preprocessor handles
   encoding/scaling.
X_processed = preprocessor.fit_transform(df_small)
# X_processed will contain scaled 'num' and one-hot columns for
   'color' (with one column dropped)
```

Listing 14: Q8: Proper encoding and scaling using ColumnTransformer

Quick illustrative pandas alternative (small datasets):

```python
# pandas get_dummies approach (do not scale the dummy columns)
X = pd.get_dummies(df_small['color'], prefix='color', drop_first
   =True)
# Combine with numeric features and scale numeric columns only
X_final = pd.concat([df_small[['num']].assign(num_scaled=(
   df_small['num']-df_small['num'].mean())/df_small['num'].std()
   ),
                     X.reset_index(drop=True)], axis=1)
```

Listing 15: Q8: Quick pandas one-hot

3 Logistic Regression: First/Second-Order Methods

Question 1 - Data generation

Data generation (Listing 3)

Use the code below (matches Listing 3) to generate synthetic binary classification data for the experiments.

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

np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X_raw, y = make_blobs(n_samples=2000, centers=centers,
    random_state=5)
transformation = np.array([[0.5, 0.5], [-0.5, 1.5]])
X = X_raw.dot(transformation)   # apply linear transform (same
    as listing 3)
```

Listing 16: Listing 3: Data generation

The generated dataset has shape `X shape: (2000, 2), y shape: (2000)` and is visualised below:

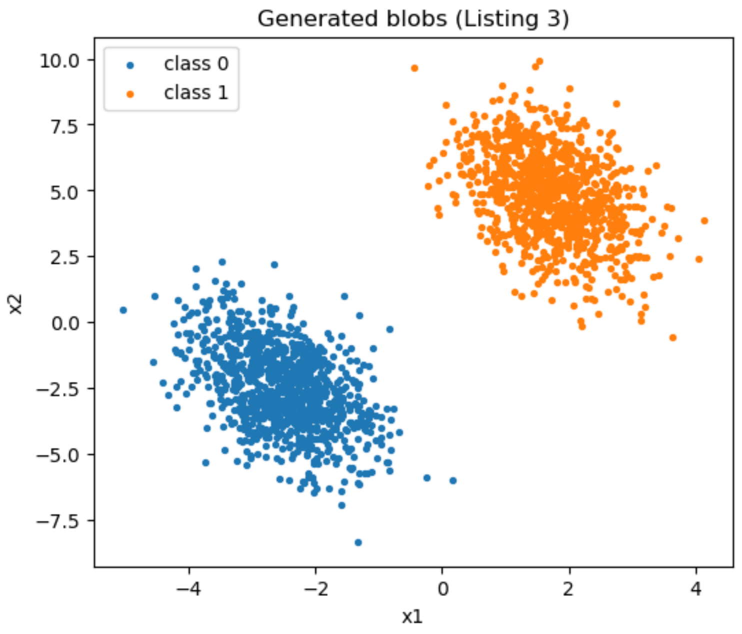

Figure 1: Generated blobs (Listing 3).

Question 2 - Implement batch Gradient Descent (20 iterations)

Batch Gradient Descent implementation and initialization

Implementation (batch GD over 20 iterations). We add a bias column to `X` before fitting.

```python
import numpy as np
from scipy.special import expit   # sigmoid

sigmoid = expit

def log_loss(w, X, y, eps=1e-12):
    z = X @ w
    p = sigmoid(z)
    p = np.clip(p, eps, 1-eps)
    return - np.mean(y * np.log(p) + (1-y) * np.log(1-p))

def grad_log_loss(w, X, y):
    n = X.shape[0]
    p = sigmoid(X @ w)
    return (X.T @ (p - y)) / n

def batch_gradient_descent(X, y, lr=0.2, n_iters=20, w_init=None
   ):
    n,d = X.shape
    w = np.zeros(d) if w_init is None else w_init.copy()
    losses = []
    for it in range(n_iters):
        losses.append(log_loss(w,X,y))
        g = grad_log_loss(w,X,y)
        w = w - lr * g
    return w, np.array(losses)
```

Listing 17: Batch GD: implementation

Initialization used: $\mathbf{w} = \mathbf{0}$ (zero vector). **Reason:** Zero initialization yields neutral probabilities (0.5) and is standard for logistic regression; the data breaks symmetry so learning proceeds.

Notebook outputs (batch GD 20 iters):

```
Iter 1/20 - loss: 0.693147
Iter 2/20 - loss: 0.182040
Iter 3/20 - loss: 0.128045
Iter 4/20 - loss: 0.101064
Iter 5/20 - loss: 0.084344
Iter 6/20 - loss: 0.072798
Iter 7/20 - loss: 0.064277
Iter 8/20 - loss: 0.057695
Iter 9/20 - loss: 0.052438
Iter 10/20 - loss: 0.048131
Iter 11/20 - loss: 0.044531
Iter 12/20 - loss: 0.041472
Iter 13/20 - loss: 0.038837
Iter 14/20 - loss: 0.036542
Iter 15/20 - loss: 0.034523
Iter 16/20 - loss: 0.032731
Iter 17/20 - loss: 0.031130
Iter 18/20 - loss: 0.029690
Iter 19/20 - loss: 0.028386
Iter 20/20 - loss: 0.027201
Final GD loss: 0.0272007041768986
GD weights: [-0.08194225  0.70305646  0.8017978 ]
```

Listing 18: Batch GD: iteration log

Question 3 - Specify the loss function used and reason

Loss selection

Loss used: Binary cross-entropy (negative log-likelihood):

$$L(w) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log p_i + (1-y_i)\log(1-p_i)\right], \quad p_i = \sigma(w^\top x_i).$$

Reason: This loss corresponds to the probabilistic model underlying logistic regression, is convex in w, and yields straightforward expressions for gradient and Hessian required by first- and second-order methods.

Question 4 - Implement Newton's method (20 iterations)

Newton's method implementation

Newton's method uses the gradient and Hessian of the logistic loss and updates $w \leftarrow w - H^{-1}g$.

```python
def hessian_log_loss(w, X, y, reg=0.0):
    n = X.shape[0]
    p = sigmoid(X @ w)
    W = p * (1-p)
    XW = X * W[:,None]
    H = (X.T @ XW) / n
    if reg:
        H = H + reg * np.eye(H.shape[0])
    return H

def newton_method(X, y, n_iters=20, w_init=None, reg=1e-6):
    n,d = X.shape
    w = np.zeros(d) if w_init is None else w_init.copy()
    losses = []
    for it in range(n_iters):
        losses.append(log_loss(w,X,y))
        g = grad_log_loss(w,X,y)
        H = hessian_log_loss(w,X,y, reg=reg)
        try:
            delta = np.linalg.solve(H, g)
        except np.linalg.LinAlgError:
            delta = np.linalg.pinv(H) @ g
        w = w - delta
    return w, np.array(losses)
```

Listing 19: Newton's method: implementation

Notebook outputs (Newton 20 iters):

```
Newton iter 1/20 - loss: 0.693147
Newton iter 2/20 - loss: 0.145507
Newton iter 3/20 - loss: 0.052967
Newton iter 4/20 - loss: 0.020448
Newton iter 5/20 - loss: 0.008091
Newton iter 6/20 - loss: 0.003256
Newton iter 7/20 - loss: 0.001331
Newton iter 8/20 - loss: 0.000553
Newton iter 9/20 - loss: 0.000233
Newton iter 10/20 - loss: 0.000099
Newton iter 11/20 - loss: 0.000042
Newton iter 12/20 - loss: 0.000017
Newton iter 13/20 - loss: 0.000007
Newton iter 14/20 - loss: 0.000003
Newton iter 15/20 - loss: 0.000001
Newton iter 16/20 - loss: 0.000001
Newton iter 17/20 - loss: 0.000000
Newton iter 18/20 - loss: 0.000000
Newton iter 19/20 - loss: 0.000000
Newton iter 20/20 - loss: 0.000000
Final Newton loss: 1.6436401959354568e-07
Newton weights: [-1.89305919  7.65396874  3.40578194]
```

Listing 20: Newton: iteration log

Comment: Newton converges in far fewer iterations because it uses curvature; each iteration is costlier (Hessian + linear solve) but usually needs fewer steps than GD.

Question 5 - Plot the loss curves and comment

Plot: loss vs iteration for Batch GD and Newton (single run). See figure below.

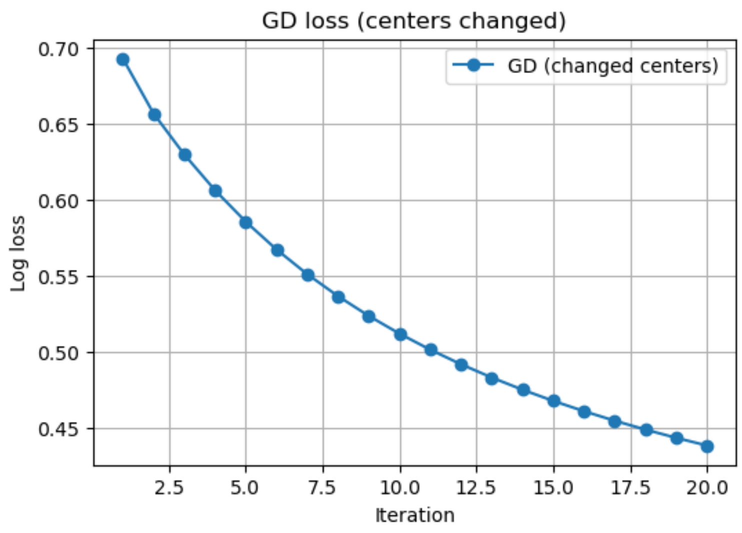

Figure 2: Loss vs iterations: Batch GD (lr=0.2) and Newton's method.

Loss vs iterations and comment

Test accuracies (notebook):

```
GD test accuracy: 1.0000
Newton test accuracy: 1.0000
```

Listing 21: Test accuracies

Commentary:

- Newton's method attains near-zero loss in very few iterations (shows exponential-like reduction), while batch GD reduces loss gradually.
- For problems where Hessian computation is feasible, Newton gives fast convergence; for large-scale problems GD (or stochastic variants) may be preferred.
- Both methods achieved perfect test accuracy on this configuration, but loss curves show Newton approached the optimum far faster.

Question 6 - Propose two approaches to decide number of iterations

Stopping criteria proposals

Two practical approaches implemented/demonstrated:

1. **Loss tolerance:** stop when change in training loss is below a small tolerance, e.g. $|L_t - L_{t-1}| < \epsilon$. This ends iterations when further improvement is negligible.
2. **Validation-based early stopping with patience:** monitor validation loss and stop when it fails to improve for `patience` iterations, returning the best weights. This guards against overfitting and adapts to data difficulty.

Notebook outputs for the two approaches:

```
Approach A (tol): stopped at iteration 10000 final loss
   0.00014624233758346933
Approach B (early stopping): stopped at 10000 best validation
   loss 0.0002453908674650203
```

Listing 22: Stopping rules outputs

Recommendation: use validation-based early stopping as the primary rule and use loss tolerance as a supplementary check.

Question 7 - Changed centers experiment and convergence analysis

Batch GD with changed centers and analysis

Experiment: change cluster centers to `centers = [[2,2],[5,1.5]]` and re-run batch GD for 20 iterations.
Notebook result:

```
GD (changed centers) test accuracy after 20 iters: 0.845
```

Listing 23: Changed-centers outputs

Analysis: Changing the centers altered class geometry and separability; the new configuration is less separable for the same learning rate/iterations, so GD after 20 iterations attains only 84.5% test accuracy. Remedies include increasing iterations, decreasing learning rate, scaling features, or using a second-order method (Newton) that adapts steps per-direction.

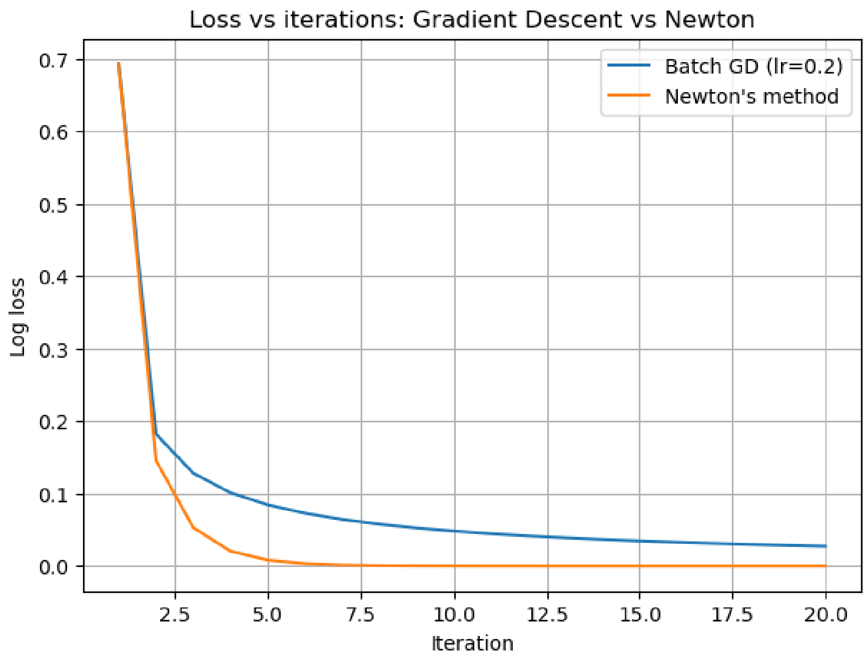

Figure 3: GD loss when cluster centers are changed (20 iterations).

4 Conclusion

This assignment demonstrates the fundamental principles of machine learning optimization and classification techniques. Through theoretical analysis and practical implementation, we explored:

Key Learning Outcomes:

- Understanding of outlier impacts on regression methods
- Comparison of regularization techniques (LASSO vs Group LASSO)
- Practical solver selection for logistic regression
- Implementation of first and second-order optimization methods
- Analysis of convergence behavior under different data conditions

The results highlight the importance of proper preprocessing, algorithm selection, and understanding the underlying mathematical principles for successful machine learning applications.

References

[1] Robert Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996.

[2] Lukas Meier, Sara Van De Geer, and Peter Bühlmann, "The group lasso for logistic regression," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 70, no. 1, pp. 53–71, 2008.

[3] Ming Yuan and Yi Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 68, no. 1, pp. 49–67, 2006.