Department of Electronic & Telecommunication Engineering,
University of Moratuwa, Sri Lanka.

Assignment 2
Learning from Data and Related Challenges and Classification

Wickramasinghe S.D.
220700T

Submitted in partial fulfillment of the requirements for the module
EN3150 Pattern Recognition

9/9/2025

Contents

1 Introduction

This assignment explores fundamental concepts in learning from data, focusing on linear and logistic regression, handling outliers, and optimization methods for classification. The first part investigates the effects of outliers on ordinary least squares (OLS) regression and introduces modified loss functions to improve the fit for inliers. In addition, the assignment examines the limitations of linear regression in high-dimensional settings such as fMRI brain imaging and compares standard LASSO and group LASSO methods for selecting predictive brain regions. The second part focuses on logistic regression using the penguins dataset, addressing challenges related to solver selection, random initialization, feature scaling, and categorical encoding. Finally, first and second-order optimization methods, batch Gradient Descent, and Newton's Method are applied to synthetic datasets to analyze convergence behavior and the influence of data distribution on training efficiency.

2 Linear Regression

1. A set of data points (x_i, y_i) is known to form a line. The ordinary least squares (OLS) is performed on this dataset. The OLS minimizes a loss function, which is

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

with y_i and $\hat{y}_i$ are true and OLS outputs, respectively. The fitted OLS line and data points are shown in Figure 1. It is observed that the OLS fitted line is not aligned with the majority of data points. What is the reason behind this?

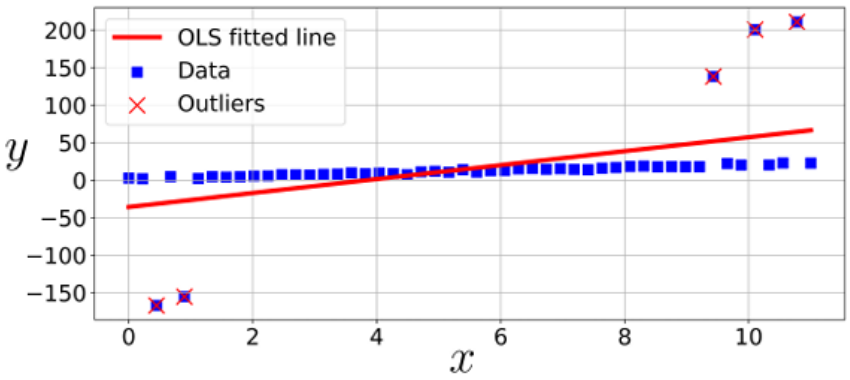

Figure 1: Ordinary least squares fit on data

Ordinary Least Squares (OLS) minimizes the sum of squared errors across all data points. Since squared error gives a higher weight to larger deviations, the presence of five outliers in the dataset heavily influences the regression line. To reduce the large errors caused by these outliers, OLS shifts the fitted line towards them, which results in a deviation from the main cluster of points. Consequently, the OLS fitted line does not align well with the majority of the data points.

2. To reduce the impact of outliers, a modified loss function is introduced. It is given as

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

There are two schemes proposed for setting a_i:

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

Under which scheme do you expect a better-fitted line for inliers than the OLS fitted line in Figure 1? Justify your answer.

Under this new loss function, each point's influence on the fitted line is proportional to its weight a_i times the squared error $(y_i - \hat{y}_i)^2$.

- Scheme 1: outliers have $a_i = 0.01$. This strongly down-weights their contribution, so the optimization effectively ignores most of the large residuals caused by outliers and fits primarily to the inliers.
- Scheme 2: outliers have $a_i = 5$. This up-weights their contribution, making their already-large squared residuals dominate the loss even more than in ordinary OLS (where $a_i = 1$), so the fitted line will be pulled further toward the outliers and away from the inliers.

Therefore, Scheme 1 reduces the influence of outliers and is expected to produce a fitted line that better represents the majority (inliers).

3. In brain image analysis (e.g., fMRI), the brain is divided into multiple regions, as shown in Figure 2, each consisting of many voxels (pixels). A researcher wants to identify which brain regions are most predictive of a specific cognitive task. Why is linear regression not a suitable algorithm for the above task?

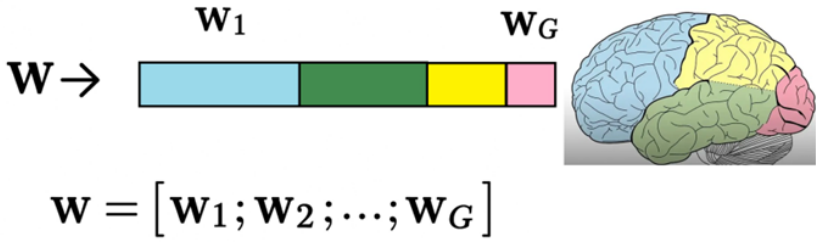

Figure 2: Simple brain segmentation

Linear regression is not a suitable algorithm for this type of brain image analysis task because of the nature of the data and the research goal. In fMRI or similar brain imaging studies, the data typically involve thousands of voxels across many brain regions, but the number of observations (subjects or trials) is usually much smaller. This creates a high-dimensional problem where the number of features far exceeds the number of samples, making linear regression prone to severe overfitting. In addition, voxels within and across regions are often highly correlated, which leads to multicollinearity, causing instability in the regression coefficients and making it difficult to interpret which brain regions are truly predictive of the cognitive task.

4. **Next, the following two methods are being considered:**

   - **Method A: Standard LASSO, which selects individual voxels independently. The LASSO objective is to minimize:**

$$\min_w \frac{1}{N}\sum_{i=1}^{N}(y_i - w^\top x_i)^2 + \lambda\|w\|_1$$

   - **Method B: Group LASSO. The Group LASSO objective is to minimize:**

$$\min_w \frac{1}{N}\sum_{i=1}^{N}(y_i - w^\top x_i)^2 + \lambda\sum_{g=1}^{G}\|w_g\|_2$$

     where w_g is the sub-vector of weights corresponding to group g, and G is the number of groups (e.g., brain regions).

Which method (LASSO or Group LASSO) is more appropriate in this setting, and why?

Group LASSO is more appropriate for identifying predictive brain regions than standard LASSO.

This is because it enforces sparsity at the group level, selecting or discarding entire regions of voxels rather than individual ones. This approach aligns with the research goal of identifying region-level predictors and respects the spatial and functional structure of brain data, where voxels within a region are often highly correlated. Compared to standard LASSO, which can produce scattered and unstable selections, Group LASSO offers more stable and interpretable results. Additionally, by applying the L2 norm to each group, Group LASSO handles multicollinearity within regions more effectively, avoiding overemphasis on single voxels and providing a more meaningful interpretation at the region level.

3 Logistic regression

1. Load Data

The following code was used to load the dataset,

```
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

df = sns.load_dataset("penguins")
df.dropna(inplace=True)

# Filter rows for 'Adelie' and 'Chinstrap' classes
selected_classes = ['Adelie', 'Chinstrap']
df_filtered = df[df['species'].isin(selected_classes)].copy()

# Make a copy to avoid the warning
# Initialize the LabelEncoder
le = LabelEncoder()

# Encode the species column
y_encoded = le.fit_transform(df_filtered['species'])
df_filtered['class_encoded'] = y_encoded
```

```
22
23 # Display the filtered and encoded DataFrame
24 print(df_filtered[['species', 'class_encoded']])
25 # Split the data into features (X) and target variable (y)
26 y = df_filtered['class_encoded'] # Target variable
27 X = df_filtered.drop(['class_encoded'], axis=1)
```

The obtained output is as follows.

```
        species  class_encoded
0        Adelie              0
1        Adelie              0
2        Adelie              0
4        Adelie              0
5        Adelie              0
..          ...            ...
215  Chinstrap              1
216  Chinstrap              1
217  Chinstrap              1
218  Chinstrap              1
219  Chinstrap              1

[214 rows x 2 columns]
```

Figure 3: Output after loading data

2. Train a logistic regression model. Here, did you encounter any errors? If yes, what were they, and how would you go about resolving them?

The following code was initially used to train the model.

```
1  # Split the data into training and testing sets
2  X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.2, random_state=42)
3
4  # Train the logistic regression model. Here we are using the saga solver to learn weights.
5  logreg = LogisticRegression(solver='saga')
6  logreg.fit(X_train, y_train)
7
8  # Predict on the testing data
9  y_pred = logreg.predict(X_test)
10
11 # Evaluate the model
12 accuracy = accuracy_score(y_test, y_pred)
13 print("Accuracy:", accuracy)
14 print(logreg.coef_, logreg.intercept_)
```

But then I encountered the following error.

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-1532930499.py in <cell line: 0>()
      4 # Train the logistic regression model. Here we are using saga solver to learn weights.
      5 logreg = LogisticRegression(solver='saga')
----> 6 logreg.fit(X_train, y_train)
      7
      8 # Predict on the testing data

                                  6 frames
/usr/local/lib/python3.12/dist-packages/pandas/core/generic.py in __array__(self, dtype, copy)
   2151     ) -> np.ndarray:
   2152         values = self._values
-> 2153         arr = np.asarray(values, dtype=dtype)
   2154         if (
   2155             astype_is_view(values.dtype, arr.dtype)

ValueError: could not convert string to float: 'Adelie'
```

Figure 4: Encountered error

This error occurs because LogisticRegression in scikit-learn requires numeric input, but the dataset contains non-numeric (string) columns. I used the following code line to check the data types of the features.

```
print(X.dtypes)
```

The output I got is given below.

```
species              object
island               object
bill_length_mm      float64
bill_depth_mm       float64
flipper_length_mm   float64
body_mass_g         float64
sex                  object
dtype: object
```

Figure 5: Features and their data types

According to that, three columns in my dataset are objects. They caused an error because a string cannot be converted to a float data type. Among them species column contains the target variable, and island and sex are features. I removed the target variable column because it is not needed in the training set, and converted island and sex into numeric using the following code.

```
# Select features (drop target columns)
X = df_filtered.drop(['species'], axis=1)

# One-hot encode categorical features
X = pd.get_dummies(X, drop_first=True)
```

Then I trained the model using the same code as above, and it gave the following result.

```
Accuracy: 0.5813953488372093
[[ 2.75514365e-03 -8.11743284e-05  4.73445486e-04 -2.87084510e-04
   3.06576060e-04  1.85164705e-04 -1.04571737e-04  1.09434453e-05]] [-8.49013702e-06]
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_sag.py:348: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
```

Figure 6: Output for the saga solver

3. Reasons for the saga solver performing poorly

In the given penguins dataset, the saga solver performs poorly because the dataset is relatively small and the features, such as bill length, flipper length, and body mass, are on very different scales. Since Saga relies on stochastic updates, it is more effective on large, high-dimensional datasets, but in this case, it may struggle to converge efficiently and can give unstable results. Without scaling the features or increasing the maximum iterations, the solver cannot properly optimize the model weights, leading to lower accuracy.

4. Changing the solver to liblinear

I have changed the solver to liblinear by replacing 'saga' with 'liblinear' as follows.

```
#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)

# Train the logistic regression model. Here we are using liblinear to learn weights.
logreg = LogisticRegression(solver='liblinear')
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(logreg.coef_, logreg.intercept_)
```

The output I obtained is as follows.

```
Accuracy: 1.0
[[ 1.32621379 -1.25548308 -0.12756343 -0.00312309  1.24947049  0.7210066
  -0.55671955 -0.22369071]] [-0.08354891]
```

Figure 7: Output for the liblinear solver

5. Why does the "liblinear" solver perform better than the "saga" solver?

In the penguins dataset, the liblinear solver performs better than the saga solver mainly because of the dataset's size and structure. The penguins dataset is relatively small, with only a few hundred samples and low-dimensional features. saga is designed for large-scale, high-dimensional problems and uses stochastic gradient updates, which can be inefficient and unstable on small datasets. In contrast, liblinear uses a coordinate descent optimization method that is deterministic and well-suited for small to medium-sized, low-dimensional, and binary classification problems like yours.

Additionally, the features in the penguins dataset are not standardized by default, and saga is more sensitive to scaling differences, while liblinear handles these cases more robustly. As a result, liblinear converges faster, provides more stable solutions, and achieves higher accuracy on this dataset compared to saga.

6. Why does the model's accuracy (with saga solver) vary with different random state values ?

The model's accuracy with the saga solver varies with different random state values because saga is a stochastic optimizer. It updates the model weights using random subsets of the data, so the optimization path depends on the random initialization and the random order of samples. With small datasets like the penguins data, this randomness has a stronger impact, leading to different

local solutions and hence variations in accuracy across runs. By contrast, solvers like liblinear, which are deterministic, give consistent results regardless of the random state.

7. Comparison of "liblinear" and "saga" solvers with feature scaling

The following code was used to scale the features and perform logistic regression using liblinear and saga solvers. Standard scaling was used to scale the features.

```
y = y.reshape(-1, 1)

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

# Filter for two classes
df_filtered = df[df['species'].isin(['Adelie', 'Chinstrap'])].copy()
y = df_filtered['species'].map({'Adelie':0, 'Chinstrap':1})  # simple encoding
X = pd.get_dummies(df_filtered.drop('species', axis=1), drop_first=True)

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train liblinear
logreg_lib = LogisticRegression(solver='liblinear')
logreg_lib.fit(X_train_scaled, y_train)
y_pred_lib = logreg_lib.predict(X_test_scaled)
acc_lib = accuracy_score(y_test, y_pred_lib)

# Train saga
logreg_saga = LogisticRegression(solver='saga', max_iter=1000)
logreg_saga.fit(X_train_scaled, y_train)
y_pred_saga = logreg_saga.predict(X_test_scaled)
acc_saga = accuracy_score(y_test, y_pred_saga)

print("Accuracy with liblinear:", acc_lib)
print("Accuracy with saga:", acc_saga)
```

The obtained accuracies are as follows.

```
Accuracy with liblinear: 1.0
Accuracy with saga: 1.0
```

Figure 8: Accuracy after feature scaling

After feature scaling using the standard scaler, both models gave the same accuracy (1.0).

Without scaling, saga solver performed poorly because the gradient steps are inconsistent across features with different magnitudes, leading to slower convergence and suboptimal weights. liblinear is less affected by scaling, so its accuracy has not been changed.

8. Suppose you have a categorical feature with the categories 'red', 'blue', 'green', 'blue', 'green'. After encoding this feature using label encoding, you then apply a feature scaling method such as Standard Scaling or Min-Max Scaling. Is this approach correct? or not?. What do you propose?

This approach is not correct.

Label encoding transforms categorical values into integer labels, such as 0, 1, and 2. These integers implicitly suggest an ordinal relationship (e.g., $2 > 1 > 0$), which does not exist in the original data (For example, colors like red, blue, and green have no inherent order). Applying Standard Scaling or Min-Max Scaling to these encoded integers treats them as continuous numerical values, potentially misleading the model into interpreting the categories as having meaningful numerical differences. This can negatively impact model performance and lead to incorrect conclusions.

One-hot encoding can be used instead of this. This converts the categorical feature into binary dummy variables, one per category. Then each feature is binary, and there's no ordinal assumption. Scaling is optional for one-hot encoded features, since they are already in the 0–1 range.

4 Logistic regression First/Second-Order Methods

1. Generating data (y - class labels, X - Feature values)

The data was generated using the following code.

```
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_blobs

# Generate synthetic data
np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)
```

2. Implementing batch Gradient descent

The following code is used to implement batch Gradient descent to update the weights for the given dataset over 20 iterations.

```
y = y.reshape(-1, 1)

# Add bias column
X_bias = np.hstack([X, np.ones((X.shape[0], 1))])

# Initialize weights and bias
n_features = X.shape[1]
w = np.random.randn(n_features, 1) * 0.01
b = 0.0

# Learning rate and iterations
lr = 0.01
iterations = 20

# Store loss for visualization
loss_bgd = []

# Batch Gradient Descent
```

```python
for i in range(iterations):
    # Predictions (linear regression)
    y_hat = np.dot(X, w) + b

    # Compute Mean Squared Error Loss
    loss = np.mean((y_hat - y)**2)
    loss_bgd.append(loss)

    # Compute gradients
    dw = (2 / X.shape[0]) * np.dot(X.T, (y_hat - y))
    db = (2 / X.shape[0]) * np.sum(y_hat - y)

    # Update weights and bias
    w -= lr * dw
    b -= lr * db

    print(f"Iteration {i+1}, Loss: {loss:.4f}")
```

The obtained losses for each iteration are as follows.

```
Iteration 1, Loss: 0.4531
Iteration 2, Loss: 0.2692
Iteration 3, Loss: 0.1996
Iteration 4, Loss: 0.1707
Iteration 5, Loss: 0.1566
Iteration 6, Loss: 0.1479
Iteration 7, Loss: 0.1415
Iteration 8, Loss: 0.1360
Iteration 9, Loss: 0.1312
Iteration 10, Loss: 0.1267
Iteration 11, Loss: 0.1225
Iteration 12, Loss: 0.1186
Iteration 13, Loss: 0.1149
Iteration 14, Loss: 0.1115
Iteration 15, Loss: 0.1082
Iteration 16, Loss: 0.1052
Iteration 17, Loss: 0.1023
Iteration 18, Loss: 0.0995
Iteration 19, Loss: 0.0969
Iteration 20, Loss: 0.0944
```

Figure 9: Loss of each iteration for Gradient descent

For the batch gradient descent implementation, the weights were initialized with small random values near zero.

This

- Breaks symmetry, ensuring that each feature's weight updates differently during gradient descent.
- If all weights were initialized to zero, all features would receive the same update in the first iteration, preventing the model from learning unique contributions of each feature.
- Keeps the initial predictions close to zero, which stabilizes training and prevents large gradients that could cause divergence in batch gradient descent.

3. Specify the loss function you have used and state the reason for your selection.

Mean Squared Error (MSE)

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

MSE calculates the average squared difference between the predicted values and the true targets, which is appropriate since we are predicting continuous outputs in this dataset. This is particularly suitable for regression tasks, because it provides a smooth and differentiable function that can be minimized using gradient descent. Additionally, squaring the errors emphasizes larger deviations, ensuring that the model focuses on reducing significant prediction mistakes.

4. Implementing Newton's method to update the weights

The following code is used to implement Newton's method for this dataset.

```
y = y.reshape(-1, 1)

# Add bias column
X_bias = np.hstack([X, np.ones((X.shape[0], 1))])

# Add a column of ones for bias term
X_bias = np.hstack([X, np.ones((X.shape[0], 1))])

# Initialize weights (including bias)
n_features = X_bias.shape[1]
w = np.random.randn(n_features, 1) * 0.01

# Store loss for visualization
loss_newton = []

# Newton's Method iterations
for i in range(20):
    # Predictions
    y_hat = np.dot(X_bias, w)

    # Compute MSE loss
    loss = np.mean((y_hat - y)**2)
    loss_newton.append(loss)

    # Gradient
    grad = (2 / X_bias.shape[0]) * np.dot(X_bias.T, (y_hat - y))

    # Hessian
    H = (2 / X_bias.shape[0]) * np.dot(X_bias.T, X_bias)

    # Update weights
    w -= np.linalg.inv(H).dot(grad)

    print(f"Iteration {i+1}, Loss: {loss:.4f}")
```

The obtained losses for each iteration are as follows.

```
Iteration 1, Loss: 0.4434
Iteration 2, Loss: 0.0099
Iteration 3, Loss: 0.0099
Iteration 4, Loss: 0.0099
Iteration 5, Loss: 0.0099
Iteration 6, Loss: 0.0099
Iteration 7, Loss: 0.0099
Iteration 8, Loss: 0.0099
Iteration 9, Loss: 0.0099
Iteration 10, Loss: 0.0099
Iteration 11, Loss: 0.0099
Iteration 12, Loss: 0.0099
Iteration 13, Loss: 0.0099
Iteration 14, Loss: 0.0099
Iteration 15, Loss: 0.0099
Iteration 16, Loss: 0.0099
Iteration 17, Loss: 0.0099
Iteration 18, Loss: 0.0099
Iteration 19, Loss: 0.0099
Iteration 20, Loss: 0.0099
```

Figure 10: Loss of each iteration for Newton's method

The results show that Newton's Method converged extremely quickly. The loss drops sharply from 0.4434 in the first iteration to 0.0099 in the second iteration. From iteration 2 onward, the loss remains almost constant at 0.0099, indicating that the weights have reached the minimum of the MSE loss function.

This behavior is expected because Newton's Method uses second-order derivative (Hessian) information, which captures the curvature of the loss function. As a result, it can take larger, well-informed steps toward the optimum compared to batch gradient descent, which only uses the gradient. The plateau after the second iteration confirms that the method has effectively found the global minimum very quickly, demonstrating the rapid convergence of Newton's Method for convex problems like linear regression.

The equation for Newton's method is as follows:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - H^{-1}\nabla L$$

where $\nabla L = \frac{2}{N}X^T(X\mathbf{w} - \mathbf{y})$ (gradient), $H = \frac{2}{N}X^TX$ (Hessian)

5. Plot the loss with respect to the number of iterations for batch Gradient descent and Newton methods in a single plot. Comment on your results.

The following code was used to plot the loss for both methods in a single plot.

```python
# ---------------- Plotting ----------------
plt.figure(figsize=(8,5))
plt.plot(range(1, iterations+1), loss_bgd, marker='o', label='Batch Gradient Descent')
plt.plot(range(1, iterations+1), loss_newton, marker='x', label="Newton's Method")
plt.xlabel('Iteration')
plt.ylabel('MSE Loss')
plt.title('Loss vs Iterations: BGD vs Newton')
plt.legend()
plt.grid(True)
plt.show()
```

The obtained plot is as follows.

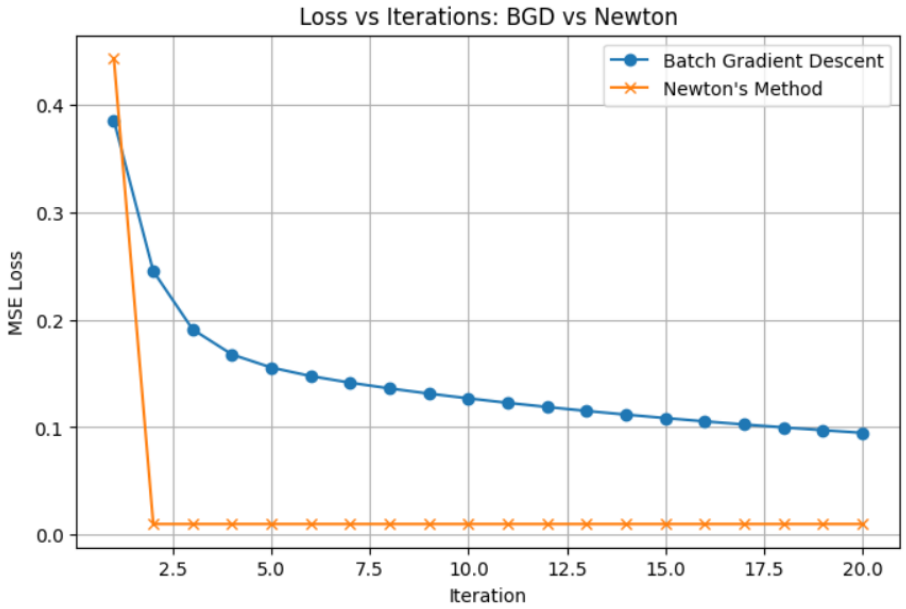

Figure 11: Loss Vs. iteration no. for batch Gradient descent and Newton's methods

According to the plot, Newton's Method converges extremely fast, usually reaching near the minimum within 1–2 iterations. Batch Gradient Descent converges gradually, reducing loss slowly over all 20 iterations. This happens because Newton's Method uses the second-order derivative (Hessian), which accounts for curvature, allowing larger, well-informed steps toward the optimum. Batch Gradient Descent uses only the first-order gradient, so it requires many small steps to reach the minimum.

6. Propose two approaches to decide the number of iterations for Gradient descent and Newton's method.

Convergence-Based Stopping Criterion
Stop the iterations when the change in loss between consecutive iterations becomes very small, i.e.,

$$|\text{Loss}_k - \text{Loss}_{k-1}| < \epsilon$$

where ϵ is a small threshold (e.g., 1×10^{-6}). This ensures that the algorithm stops when it has effectively reached the minimum, avoiding unnecessary iterations.

Predefined Iteration Limit
This sets a maximum number of iterations to ensure the algorithm does not run indefinitely. This is often combined with a convergence check (hybrid approach).

The maximum can be based on:

1. Experimental results (observe how many iterations it usually takes to converge).
2. Problem size and solver type (Newton's Method often requires fewer iterations than Gradient Descent).

7. Suppose the centers in listing 3 are changed to centers = [[2, 2], [5, 1.5]]. Use batch Gradient descent to update the weights for this new configuration. Analyze the convergence behavior of the algorithm with this updated data, and explain the convergence behavior.

The new code for data generation is as follows.

```
# New centers
np.random.seed(0)
centers = [[2, 2], [5, 1.5]]  # updated centers
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)
```

Batch gradient descent was implemented using the same code as above. The resulting losses for each iteration are as follows.

```
Iteration 1, Loss: 0.4427
Iteration 2, Loss: 0.2665
Iteration 3, Loss: 0.2134
Iteration 4, Loss: 0.1945
Iteration 5, Loss: 0.1853
Iteration 6, Loss: 0.1790
Iteration 7, Loss: 0.1737
Iteration 8, Loss: 0.1689
Iteration 9, Loss: 0.1643
Iteration 10, Loss: 0.1600
Iteration 11, Loss: 0.1560
Iteration 12, Loss: 0.1521
Iteration 13, Loss: 0.1485
Iteration 14, Loss: 0.1450
Iteration 15, Loss: 0.1418
Iteration 16, Loss: 0.1387
Iteration 17, Loss: 0.1358
Iteration 18, Loss: 0.1330
Iteration 19, Loss: 0.1304
Iteration 20, Loss: 0.1279
```

Figure 12: Loss of each iteration for batch Gradient descent using new data

I plotted loss Vs. iteration for both instances in the same plot, and got the following output.

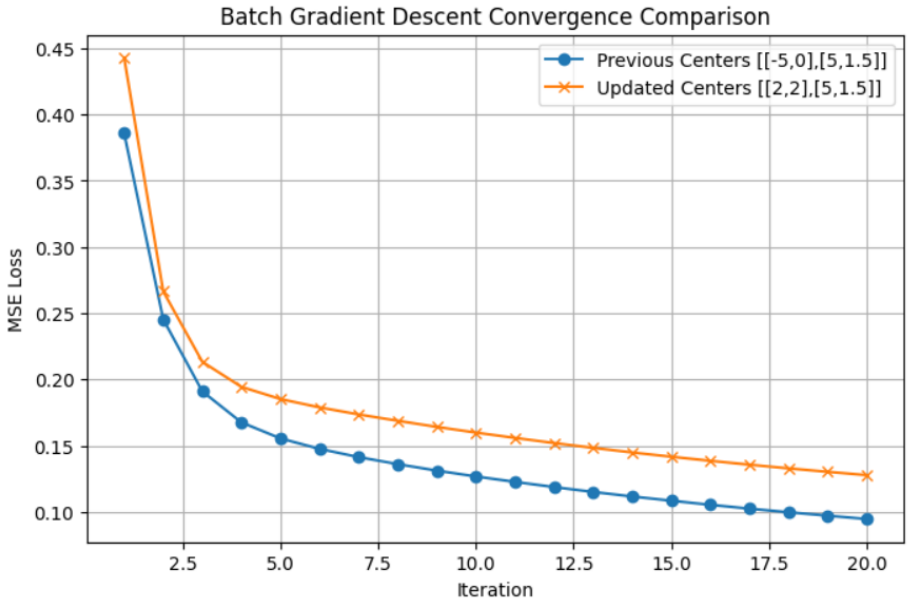

Figure 13: Loss Vs. iteration no. for old and new datasets

When using the previous dataset, the loss decreased from 0.3858 to 0.0948 over 20 iterations. The decrease was relatively steady, showing consistent gradient updates. In the updated Dataset, the loss decreased from 0.4427 to 0.1279 over the same 20 iterations. The decrease is slower and more gradual, and the final loss is higher than in the previous dataset.

This happens because the updated centers are closer together, so the initial errors between predictions and targets are smaller for most points. As the gradient magnitude is proportional to the prediction error, smaller errors result in smaller updates per iteration, leading to slower convergence. In contrast, the previous dataset had widely separated clusters, producing larger initial errors, larger gradients, and faster reduction in loss.

In conclusion, it is understood that the convergence of Batch gradient descent is sensitive to the spread and distribution of data. Closer clusters reduce the gradient magnitudes, slowing convergence, while widely separated clusters produce faster convergence due to larger gradients.

5 Conclusion

This assignment highlights the importance of choosing appropriate models, solvers, and loss functions based on data characteristics. Modified loss functions effectively mitigate the influence of outliers in linear regression, while group LASSO better captures region-level patterns in high-dimensional brain data compared to standard LASSO. In logistic regression, the liblinear solver consistently outperforms saga for small datasets, and feature scaling further stabilizes convergence. Batch Gradient Descent demonstrates steady, gradual loss reduction, whereas Newton's Method achieves rapid convergence due to second-order information. Changes in data distribution, such as shifting cluster centers, significantly affect convergence speed, emphasizing the interplay between optimization algorithms and dataset geometry. Overall, the assignment underscores the critical role of algorithm selection, data preprocessing, and iterative optimization in achieving accurate and interpretable models.