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

Assignment 02 - Learning from data and related challenges and classification

https://github.com/SasikaA073/pattern-recognition-assignments-m

Y.W.S.P Amarasinghe 210035A

Submitted in partial fulfillment of the requirements for the module
EN 3150 Pattern Recognition

2nd of October 2024

0.1 Logistic Regression

0.1.1 Question 1

Q1.2) The purpose of `y_encoded = le.fit_transform(df_filtered['species'])`:
To represent categorical values for the "species" feature as numerical values (integers).

Q1.3) The purpose of `X = df.drop(['species', 'island', 'sex'], axis=1)`:
To delete the columns (columns are vertical and axis=1) with the names "species", "island", and "sex".

Q1.4) Why can't we use the "island" and "sex" features?
Our target variable is to predict the species of a penguin. Penguin species is a biological factor, so "island" is irrelevant, and the species does not depend on the "sex" of each penguin.

Q1.6) What is the usage of `random_state=42`?
Splitting the dataset into train and test sets is done randomly. If we run `train_test_split()` multiple times, we will get different results. Setting `random_state` to an integer ensures reproducibility of the split.

Q1.7) Why is the accuracy low? Why does the `saga` solver perform poorly?
The SAGA (Stochastic Average Gradient Augmented) solver is better suited for very large datasets. Here, the dataset is not very large and is not sparse, which limits the solver's effectiveness.

Q1.8) What is the classification accuracy using the `liblinear` solver?
The classification accuracy using the `liblinear` solver is 1.0.

Q1.9) Why does the `liblinear` solver perform better than the `saga` solver?
`liblinear` tends to outperform `saga` on smaller datasets due to its efficient handling of fewer data points, leading to quicker convergence. Both solvers support L1 regularization; however, `liblinear`'s efficiency in smaller datasets often gives it an edge in accuracy and speed over `saga`.

Q1.10) Why is there a significant difference in accuracy with and without feature scaling using the `saga` and `liblinear` solvers?
After applying the standard scaler to the dataset, the accuracy increases with the `saga` solver due to reduced variance in feature values. This helps the solver focus on learning the underlying patterns instead of being affected by disparities in feature scales.

Q1.11) What is the problem in the code given in Listing 3, and how can it be solved?
We should remove the features like `sex` and `island`, which do not contribute to determining the penguin species.

Q1.12) 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? What do you propose?
Categorical values are non-numerical, and after label encoding, they become discrete values. The model might interpret label-encoded numbers as ordinal, which is not ideal. After scaling, these numbers will become non-integer values, which could distort the categories.

I propose using one-hot encoding to encode categorical values. This produces binary values [0, 1], and scaling is not necessary.

0.1.2 Question 2

$$w_0 = -5.9$$
$$w_1 = 0.06$$
$$w_2 = 1.5$$

Part A

$$y = \Pr(A^+)$$
$$x_1 = \text{number of hours studied}$$
$$x_2 = \text{undergraduate GPA}$$

$$x_1 = 50, \quad x_2 = 3.6$$

$$z = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2$$
$$\Pr(A^+) = \frac{1}{1 + e^{-z}}$$
$$\Pr(A^+) = 0.9241418199787566$$

Part B

$$\Pr(A^+) = p$$
$$p = 0.6$$
$$z = -\log\left(\frac{1}{p} - 1\right)$$

Subject x_1 :

$$x_1 = \frac{z - w_0 - w_2 \cdot x_2}{w_1}$$
$$x_1 = 15.091 \text{ hours}$$

0.2 Logistic regression on real world data

Q2.1) UCI Machine Learning Repository dataset I used
Dataset Information: MAGIC Gamma Telescope

- **UCI ID:** 159
- **Name:** MAGIC Gamma Telescope
- **Repository URL:** `https://archive.ics.uci.edu/dataset/159/magic+gamma+telescope`
- **Data URL:** `https://archive.ics.uci.edu/static/public/159/data.csv`
- **Abstract:** Data are MC generated to simulate registration of high energy gamma particles in an atmospheric Cherenkov telescope.
- **Area:** Physics and Chemistry
- **Tasks:** Classification
- **Characteristics:** Multivariate
- **Number of Instances:** 19,020
- **Number of Features:** 10
- **Feature Types:** Real
- **Target Column:** `class`
- **Has Missing Values:** No
- **Year of Dataset Creation:** 2004
- **Last Updated:** Tue Dec 19 2023
- **DOI:** `10.24432/C52C8B`
- **Creators:** R. Bock

Additional Information:

The data are MC generated to simulate registration of high energy gamma particles in a ground-based atmospheric Cherenkov gamma telescope using the imaging technique. Cherenkov gamma telescopes observe high-energy gamma rays by capturing the radiation emitted by charged particles in electromagnetic showers. These showers, initiated by the gamma rays, develop in the atmosphere, and the Cherenkov radiation (visible to UV wavelengths) leaks through the atmosphere and is recorded by the detector, allowing for the reconstruction of the shower parameters.

The available information consists of pulses left by the incoming Cherenkov photons on the photomultiplier tubes arranged in a plane, known as the camera. Depending on the primary gamma energy, the total number of Cherenkov photons collected ranges from a few hundred to over 10,000. The characteristic pattern, called the "shower image," enables statistical discrimination between primary gamma-induced showers (signal) and hadronic showers caused by cosmic rays (background).

Typically, after preprocessing, a shower image forms an elongated cluster. A principal component analysis (PCA) performed in the camera plane results in a correlation axis, defining an ellipse. This characteristic ellipse (Hillas parameters) can be used for discrimination, along with other features like cluster extent and total depositions. Asymmetry along the major axis is also useful for discrimination.

The data were generated by a Monte Carlo program, Corsika, as described in:

D. Heck et al., CORSIKA, A Monte Carlo code to simulate extensive air showers, Forschungszentrum Karlsruhe FZKA 6019 (1998). `http://rexa.info/paper?id=ac6e674e9af20979b23d3ed4521f1570765e8d68`

Variable Information:

1. `fLength:` continuous — major axis of ellipse [mm]
2. `fWidth:` continuous — minor axis of ellipse [mm]
3. `fSize:` continuous — 10-log of the sum of content of all pixels [in phot]
4. `fConc:` continuous — ratio of sum of two highest pixels over `fSize` [ratio]
5. `fConc1:` continuous — ratio of highest pixel over `fSize` [ratio]
6. `fAsym:` continuous — distance from highest pixel to center, projected onto major axis [mm]
7. `fM3Long:` continuous — 3rd root of third moment along major axis [mm]
8. `fM3Trans:` continuous — 3rd root of third moment along minor axis [mm]
9. `fAlpha:` continuous — angle of major axis with vector to origin [deg]
10. `fDist:` continuous — distance from origin to center of ellipse [mm]
11. `class:` g (gamma, signal), h (hadron, background)

Class Distribution:

- **Gamma (signal):** 12,332
- **Hadron (background):** 6,688

For comparison of different classifiers, an ROC curve should be used, as the simple classification accuracy is not meaningful for this data. Specific thresholds for accepting background events as signals are relevant for different experiments.

Q2.2)

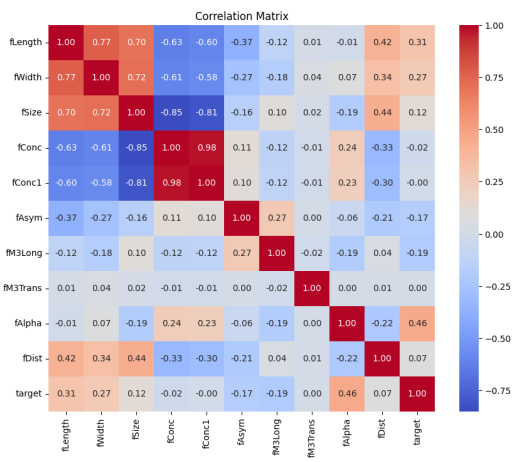


Figure 1: Correlation Matrix of the dataset

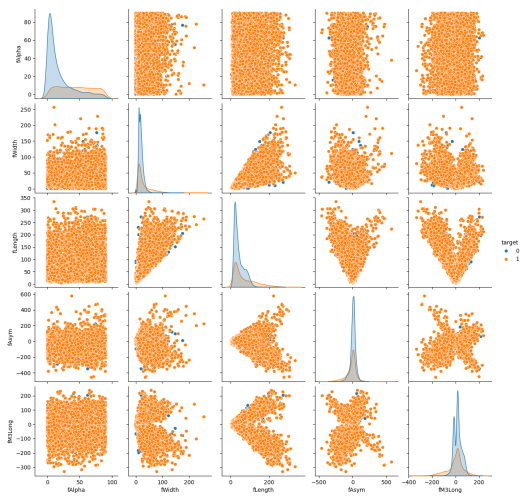


Figure 2: SNS Pairplot of chosen features

Comments Using the Correlation Matrix, I chose the `'fAlpha', 'fWidth', 'fLength', 'fAsym', 'fM3Long'` as the features because they have the highest correlation with the target value. Using

those features I plot the pair plots. In 2 dimensions, there's no apparent boundary to classify in each of feature. Therefore, we have to consider multiple features to train a model to classify into the two target classes.

Q2.3) Evaluation of the trained Logistic Regression Model

```python
# Import necessary libraries
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import pandas as pd
from ucimlrepo import fetch_ucirepo

# Fetch dataset
magic_gamma_telescope = fetch_ucirepo(id=159)

# Features and target
X = magic_gamma_telescope.data.features
y = magic_gamma_telescope.data.targets

# Label encoding the target variable
le = LabelEncoder()
y_encoded = le.fit_transform(y.values.ravel())

# Split the data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

# Initialize logistic regression model
logreg = LogisticRegression(solver='liblinear')  # Using liblinear solver for binary classification

# Train the model on the training data
logreg.fit(X_train, y_train)

# Predict the target variable on the test set
y_pred = logreg.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")

# Confusion matrix and classification report
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Detailed classification metrics (precision, recall, F1-score)
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)
```
✓ 20.9s

Accuracy: 0.7892

Confusion Matrix:

Actual / Predicted	0	1
0	2207	253
1	549	795

Classification Report:

```
              precision    recall  f1-score   support

           0       0.80      0.90      0.85      2460
           1       0.76      0.59      0.66      1344

    accuracy                           0.79      3804
   macro avg       0.78      0.74      0.76      3804
weighted avg       0.79      0.79      0.78      3804
```

Q2.4) Interpreting the p-values for the predictors and determine if any features can be discarded

```python
import statsmodels.api as sm
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from ucimlrepo import fetch_ucirepo

# Fetch dataset from UCI
magic_gamma_telescope = fetch_ucirepo(id=159)

# Features and target
X = magic_gamma_telescope.data.features
y = magic_gamma_telescope.data.targets

# Label encoding the target variable
le = LabelEncoder()
y_encoded = le.fit_transform(y.values.ravel())

# Add an intercept column to the dataset (statsmodels does not add it by default)
X = sm.add_constant(X)

# Fit the Logistic Regression model using statsmodels
logit_model = sm.Logit(y_encoded, X)
result = logit_model.fit()

# Print the summary which includes p-values, coefficients, and other statistics
print(result.summary())
```

Figure 3: Stat Model Code

```
Optimization terminated successfully.
         Current function value: 0.457329
         Iterations 7
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                19020
Model:                          Logit   Df Residuals:                    19009
Method:                           MLE   Df Model:                           10
Date:                Wed, 02 Oct 2024   Pseudo R-squ.:                  0.2947
Time:                        15:58:32   Log-Likelihood:                -8698.4
converged:                       True   LL-Null:                       -12334.
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.5854      0.311    -21.200      0.000      -7.194      -5.977
fLength        0.0296      0.001     28.010      0.000       0.028       0.032
fWidth         0.0055      0.002      2.226      0.026       0.001       0.010
fSize          0.6430      0.096      6.696      0.000       0.455       0.831
fConc         -0.0529      0.521     -0.102      0.919      -1.073       0.968
fConc1         5.4400      0.755      7.201      0.000       3.959       6.921
fAsym       1.254e-05      0.000      0.029      0.977      -0.001       0.001
fM3Long       -0.0072      0.001    -13.493      0.000      -0.008      -0.006
fM3Trans      -0.0006      0.001     -0.553      0.581      -0.003       0.002
fAlpha         0.0451      0.001     52.965      0.000       0.043       0.047
fDist          0.0006      0.000      1.857      0.063    -3.1e-05       0.001
==============================================================================
```

Figure 4: Stat Model Summary

Comments P-values for the predictors: • fAlpha 0.000 • fWidth 0.026 • fLength 0.000 • fAsym 0.977 • fM3Long 0.000 Features to discard (p-value 0.05): according to the threshold, fAsym is discarded and after removing that p-values for other classes also can be reduced.

0.3 Logistic regression First/Second-Order Methods

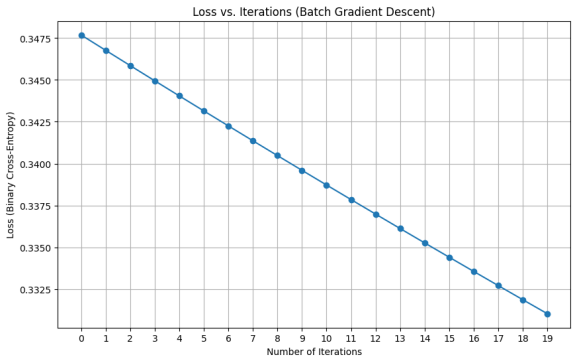
0.3.1 Batch Gradient Descent

```
# Batch Gradient Descent
for i in range(iterations):
    # Compute predictions
    z = np.dot(X, weights)
    predictions = sigmoid(z)

    # Compute the gradient
    gradient = np.dot(X.T, (predictions - y)) / y.size

    # Update weights
    weights -= learning_rate * gradient
```

Q3.2) the method used to initialize the weights and reason for your selection

I used random values in between in 0 1. If we choose w as zeros, we can get initial boundaries with very low gradients but also low accuracy, which can cause the model to get stuck in those incorrect values.

Q3.3) BGD: Loss vs Iterations



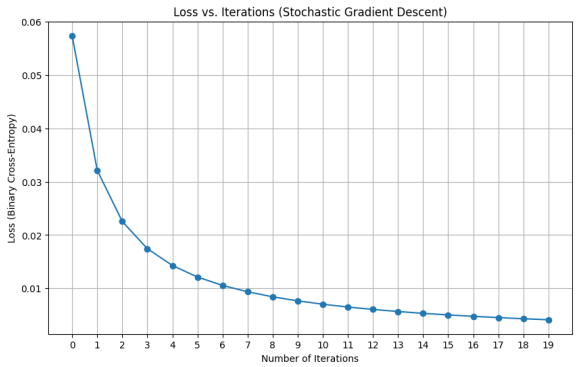
Q3.3) Loss function I used

I used **Binary Cross Entropy Loss**. Binary cross-entropy is suitable because logistic regression outputs a probability between 0 and 1, and cross-entropy captures the difference between these probabilities and the actual class labels (0 or 1).It measures how well the logistic regression model predicts the true class labels.

0.3.2 Stochastic Gradient Descent

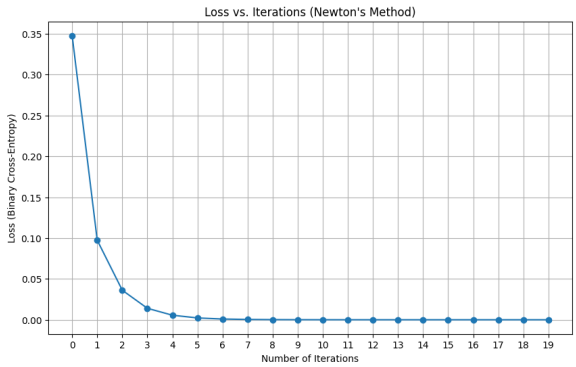
```
# Stochastic Gradient Descent
for i in range(iterations):
    for j in range(X.shape[0]):  # Iterate over each sample
        # Compute prediction for the j-th sample
        z = np.dot(X[j], weights)
```

```
        prediction = sigmoid(z)

        # Compute the gradient for the j-th sample
        gradient = (prediction - y[j]) * X[j]

        # Update weights
        weights -= learning_rate * gradient

    # Compute and store the cost for the entire dataset after each epoch
    z_all = np.dot(X, weights)
    predictions_all = sigmoid(z_all)
    cost = -np.mean(y * np.log(predictions_all + 1e-15) + (1 - y) * np.log(1 - predictions_all + 1e-15))
    sgd_loss_values.append(cost)
```



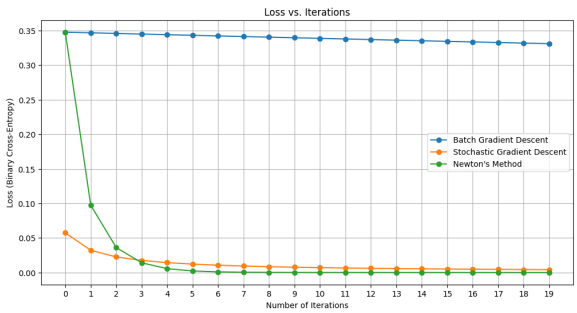
0.3.3 Newton's Method

```
# Newton's Method
for i in range(iterations):
    # Compute predictions
    z = np.dot(X, weights)
    predictions = sigmoid(z)

    # Compute the gradient
    gradient = np.dot(X.T, (predictions - y)) / y.size

    # Compute the Hessian matrix
    W = np.diag(predictions * (1 - predictions))  # Diagonal matrix of the second derivatives
    H = np.dot(X.T, np.dot(W, X)) / y.size  # Hessian

    # Update weights using the Newton's method formula
    weights -= np.linalg.inv(H).dot(gradient)

    # Compute and store the cost for the entire dataset after each iteration
    cost = -np.mean(y * np.log(predictions + 1e-15) + (1 - y) * np.log(1 - predictions + 1e-15))
    newtons_method_loss_values.append(cost)
```



0.3.4 Comparison

Q3.8) Comparison



Q3.9) Approaches to decide number of iterations for Gradient descent and Newton's method

Gradient Descent

- **Early Stopping**: Monitor loss and stop training if it doesn't improve after a set number of iterations. Prevents overfitting and saves resources.
- **Fixed Iterations with Validation**: Set a predefined iteration count using cross-validation to optimize performance and balance training time.

Newton's Method

- **Convergence Tolerance**: Stop when changes in weights or cost are below a set threshold, ensuring negligible improvements aren't pursued.
- **Max Iterations with Backtracking**: Limit iterations while adjusting learning rate dynamically. This balances efficiency and optimal convergence.

Q3.10) Suppose the centers in in listing 4 are changed to centers = [[3, 0], [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 provide an explanation for convergence behavior. The data shows that with the new centers [[3, 0], [5, 1.5]], convergence is significantly slower compared to the original configuration. In the previous scenario, the cost rapidly decreased from 0.6931 to 0.3321 by iteration 15, indicating quicker progress toward finding the decision boundary. With the new centers, the cost only drops from 0.6931 to

0.6571 by iteration 15, highlighting a much slower reduction in loss. This slower convergence can be attributed to the closer proximity of the new centers, which causes increased overlap between the classes, making it harder for the model to distinguish between them. As a result, the gradient descent algorithm takes longer to find an optimal solution. The decision boundary is more complex, leading to a gradual decrease in the cost function.