Assignment 02

Learning from data and related challenges and classification

Thuvaragan S. 210657G

02 October 2024

Submitted in partial fullfillment of the requirements for the module **EN3510** : **Pattern Recognition**
from *Electronics and Telecommunication Department, University of Moratuwa*

1 Logistic regression

1.1. Use the code given in `listing 1` to load data

```
# listing 1
df = sns.load_dataset("penguins")
df.dropna(inplace=True)

selected_classes = ["Adelie", "Chinstrap"]
df_filtered = df[df["species"].isin(selected_classes)].copy()

le = LabelEncoder()

y_encoded = le.fit_transform(df_filtered["species"])

df_filtered["class_encoded"] = y_encoded

print(df_filtered[["species", "class_encoded"]])

y = df_filtered["class_encoded"]
X = df_filtered.drop(["species", "island", "sex", "class_encoded"], axis=1)
```

1.2. What is the purpose of `y_encoded = le.fit_transform(df_filtered['species'])`

It encodes the categorical target variable `species` into numeric values using the `LabelEncoder`. The `fit_transform` method assigns a unique integer to each category ("Adelie" might be encoded as 0 and "Chinstrap" as 1). Since logistic regression requires the target variable `y` to be numeric this step is necessary.

1.3. What is the purpose of `X = df.drop(['species', 'island', 'sex'], axis=1)`

This line of code drops the columns `species`, `island`, and `sex` from the dataframe to create the feature matrix `X` with a simplified feature space leaving only the numerical and continuous variables. The `species` column is dropped because it is the target variable (label) for prediction.

1.4. Why we cannot use `island` and `sex` features

`island` and `sex` are categorical variables. Without proper encoding (such as one-hot encoding), logistic regression cannot handle non-numeric features directly. Categorical variables lead to poor performance or errors in Logistic Regression. Therefore, these features should either be transformed into numeric representations (one-hot encoding) or excluded as they are not relevant.

1.5. Train a logistic regression model using code given in `listing 2`

```
# listing 2
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

logreg = LogisticRegression(solver="saga")
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("accuracy:", accuracy)
print(logreg.coef_, logreg.intercept_)
```

Output

```
ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
warnings.warn(
accuracy: 0.5813953488372093
[[ 2.76307837e-03 -8.24275882e-05  4.68334550e-04 -2.86859667e-04]] [-8.43695981e-06]
```

The model is trained very poorly and sega solver cannot even converge within the default iteration count.

1.6. What is the usage of `random_state=42`

Setting `random_state=42` ensures reproducibility. When using a fixed `random_state`, the train-test split is always the same. This allows to consistently obtain the same results, making debugging and comparing models easier.

1.7. Why is accuracy low. why does the saga solver perform poorly.

The saga solver is more suited for large datasets and sparse data and may perform more slowly or less optimally on smaller datasets because it is based on `SAG` optimization which is an extension of `SGD` algorithm. Stochastic nature is not very effective in smaller datasets and will complicate the converging process. Hence the capability to handle complex and large datasets becomes a bottleneck when it comes to smaller datasets like `penguin`. And that is why we observe poor performance with saga solver here.

1.8. What is the classification accuracy with `liblinear` configuration

Output

```
liblinear accuracy: 1.0
[[ 1.61343628 -1.46657604 -0.15152343 -0.00398477]] [-0.08866944]
```

1.9. Why does the `liblinear` solver perform better than `saga` solver

The `liblinear` solver is specifically **designed for smaller datasets**. It uses a coordinate descent algorithm that can converge faster for small or medium-sized datasets like the penguins dataset. Compared to the SGD based algorithm used in `saga` solver, `liblinear` is more efficient and stable for smaller datasets and can perform better.

1.10. Compare the performance of the `liblinear` and `saga` solvers with feature scaling. If there is a significant difference in the accuracy with and without feature scaling, what is the reason for that.

Output

```
saga accuracy: 0.5813953488372093
[[ 2.76307837e-03 -8.24275882e-05  4.68334550e-04 -2.86859667e-04]] [-8.43695981e-06]
liblinear accuracy: 1.0
[[ 1.61343628 -1.46657604 -0.15152343 -0.00398477]] [-0.08866944]
liblinear accuracy with scaling: 0.9767441860465116
[[ 3.84019948 -0.76794126  0.18337305 -0.71426409]] [-0.08866944]
saga accuracy with scaling: 0.9767441860465116
[[ 3.94091253 -0.82720771  0.1955952  -0.73538893]] [-0.08866944]
```

Using `StandardScaler` (selected feature scaling technique) **significantly impacts the performance** of logistic regression, **especially for solvers like saga**, which are sensitive to the magnitude of feature values. Here, we can see the accuracy shot up from `0.5813953488372093` to `0.9767441860465116` when standard scaler is used.

Scaling brings all features to the same range, allowing the optimization algorithm to converge more efficiently. The liblinear solver performs well without scaling but seems to be negatively affected by standard scaling as we can observe the accuracy dropped from `1.0` to `0.9767441860465116`, this might be because **saga benefits more from scaled features** since it can handle larger and more complex datasets while liblinear cannot.

1.11 What is the problem with `listing 3` and how to solve it

```python
# listing 3
# ... same as listing 1 ...

# >>>
# categorical variables are undesiarable in logistic regression
X = df_filtered.drop(["species", "class_encoded"], axis=1)
# >>>

y = df_filtered["class_encoded"]  # target variable
X.head()
# ... same as listing 2 ...
```

`listing 3` drops only the `species` and `class_encoded` columns, leaving the `island` and `sex` columns in `X`. For the same reason mentioned in part 4 above, keeping the `island` and `sex` columns without encoding them will cause issues during model training.

Solution: Dropping the `island` and `sex` columns from `X` or use one-hot encoding to convert them into numerical features if they are relevant.

```python
X = df_filtered.drop(["species", "island", "sex", "class_encoded"], axis=1)
```

1.12 Suppose you have a categorical feature with the categories `red`, `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.

Applying label encoding followed by feature scaling to categorical variables is incorrect because label encoding assigns arbitrary numeric values to categories. Scaling these values would mean that there is an ordinal/numeric relationship between the categories (e.g., “red” < “blue” < “green”), which doesn’t exist. Hence, this approach is **not correct**.

Proposed Solution: Use one-hot encoding for categorical variables instead of label encoding. One-hot encoding creates binary columns for each category, eliminating any notion of ordinal relationships between them. After one-hot encoding, you can apply scaling to any numeric features if needed, but categorical features shouldn’t be scaled.

Question 2

Consider a dataset collected from a statistics class that includes information on students. The dataset includes variables x_1 representing the number of hours studied, x_2 representing the undergraduate GPA, and y indicating whether the student received an $A+$ in the class. After conducting a logistic regression analysis, we obtained the following estimated coefficients: $w_0 = -5.9$, $w_1 = 0.06$, and $w_2 = 1.5$.

1. What is the estimated probability that a student, who has studied for 50 hours and has an undergraduate GPA of 3.6, will receive an $A+$ in the class? [12.5 marks]

The logistic regression model for the probability of receiving an $A+$ is

$P(y = 1|x_1, x_2) = \frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2)}}$

where the logistic regression coefficients are given as $w_0 = -5.9$, $w_1 = 0.06$ and $w_2 = 1.5$

$x_1 = 50$ is the number of hours studied and $x_2 = 3.6$ is the undergraduate GPA for which the probability should be estimated.

By substituting and simplifying $P(y = 1) = \frac{1}{1+e^{-(-5.9+0.06\cdot 50+1.5\cdot 3.6)}} = \frac{1}{1+e^{-2.5}} \approx 0.924$

Hence, the estimated probability is 92.4%

2. To achieve a 60% chance of receiving an $A+$ in the class, how many hours of study does a student like the one in part 1 need to complete? [12.5 marks]

Substituting the suitable values leaves us with $0.6 = \frac{1}{1+e^{-(-5.9+0.06\cdot x_1+1.5\cdot 3.6)}}$ and rearranging the equation gives us $e^{-(-5.9+0.06\cdot x_1+5.4)} = \frac{2}{3}$ and by simplifying further, $x_1 = \frac{0.5-\ln(\frac{2}{3})}{0.06} \approx 15.09$

So the student will need around 15.09 hours of study to have 60% chance of receiving an $A+$, which is insane.

2 Logistic regression on real world data

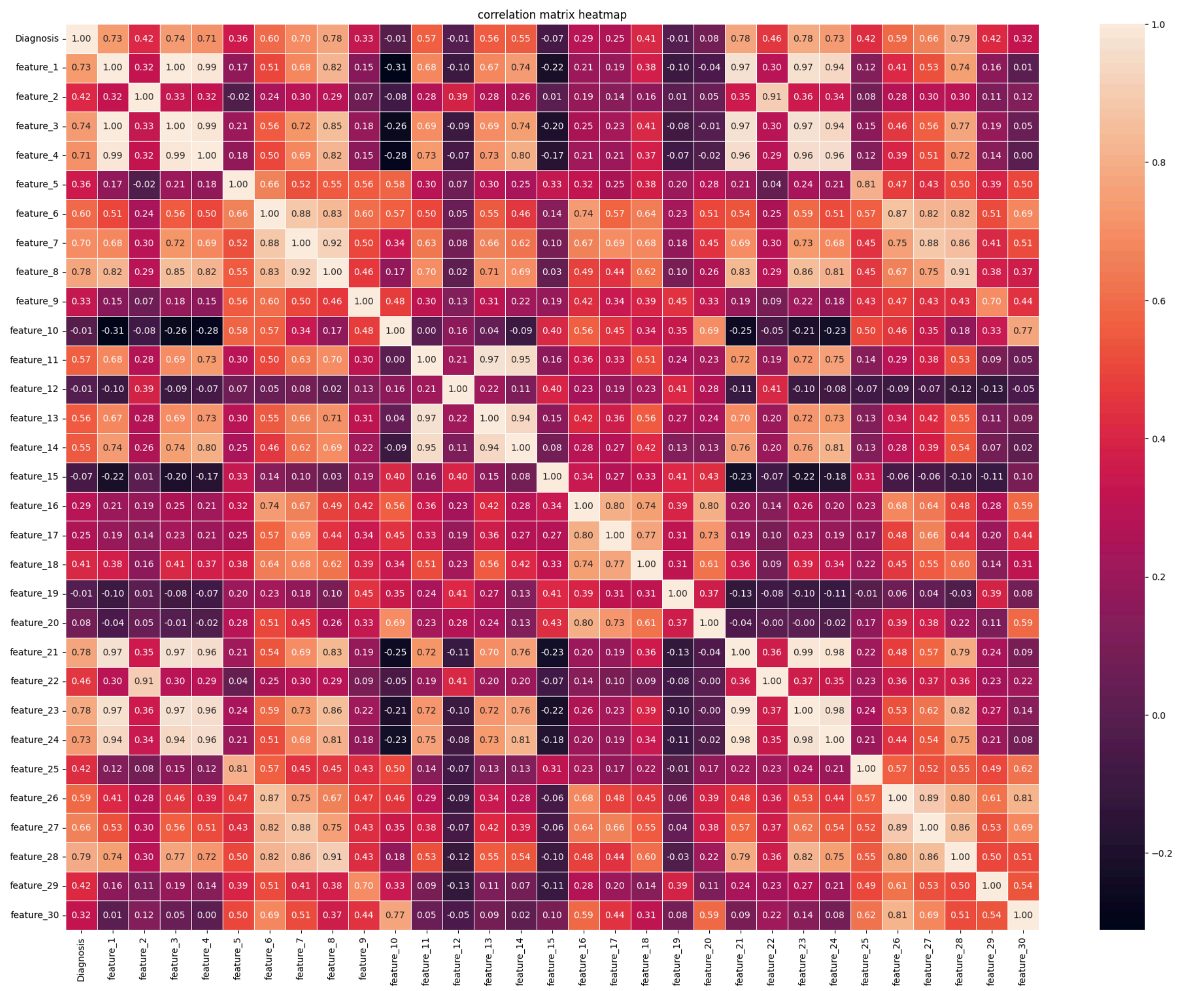
2.1. Choose a data set from UCI Machine Learning Repository that is appropriate for logistic regression

Selected Diagnostic Wisconsin Breast Cancer Database

Goal is to predict whether a tumor is malignant (1) or benign (0) from several features. Since the target variable `Diagnosis` is binary and the feaatures are 30 numerical, this is well suited for logistic regression.

2.2. Obtain the correlation matrix for the dataset you have chosen. Further, obtain pair plots using sns.pairplot. If your dataset contains many features, select up to 5 features for analysis. Comment on your results

1. ***Correlation Matrix***



Selected `feature_1`, `feature_4`, `feature_8`, `feature_10`, and `feature_15` to observe the different correlations.

It can be observed that the features 1, 3, and 4 are highly correlated and redundant, meanwhile 14, 15 are have little to no correlation between them.

Negative correlation with Diagnosis is hard to observe in this dataset, yet there are some features like 1, 10, and 15 which are negatively correlated within themselves.

We can observe how different correlation affects different features in the graph below too.

2. ***Pair Plots***

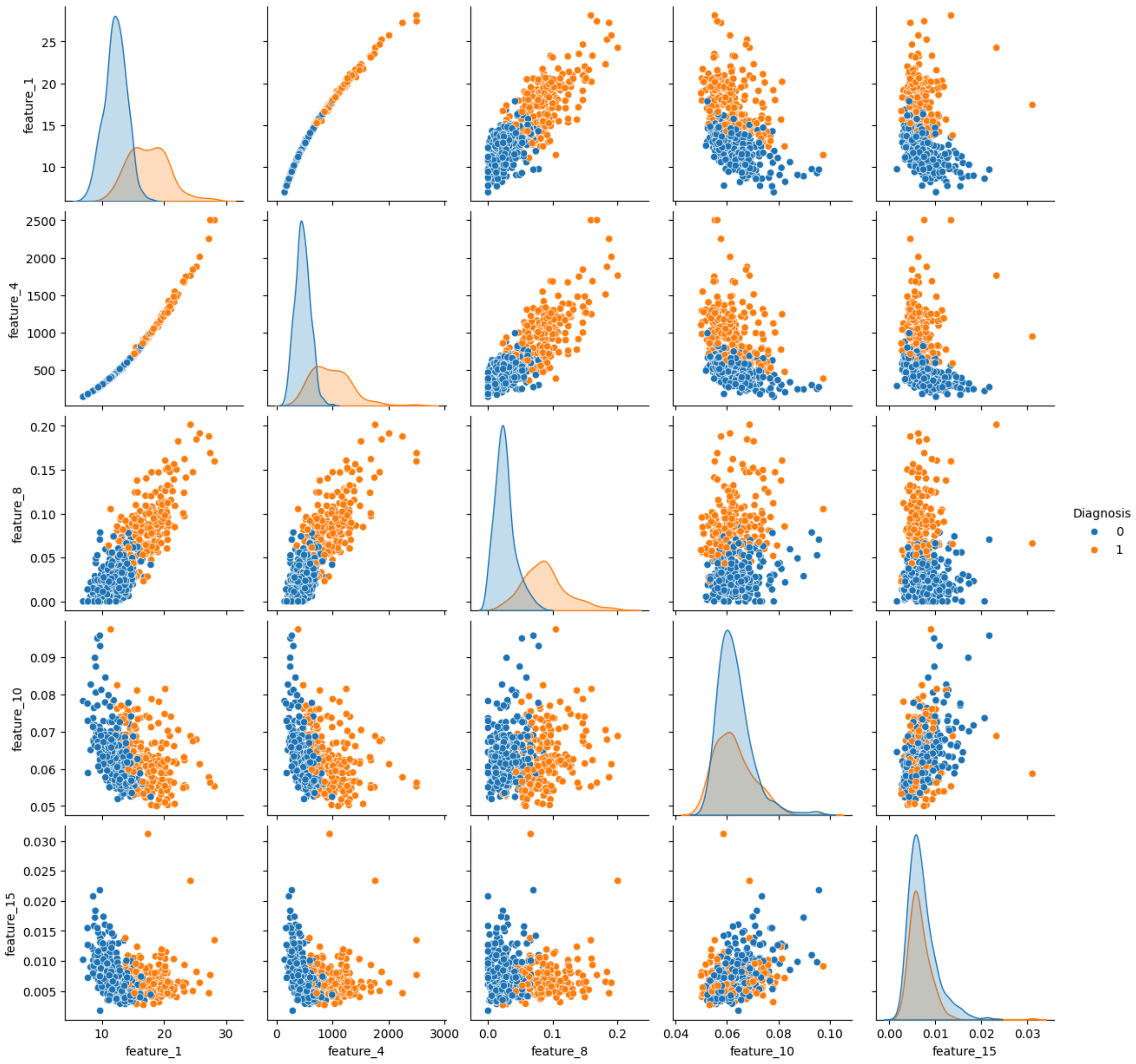


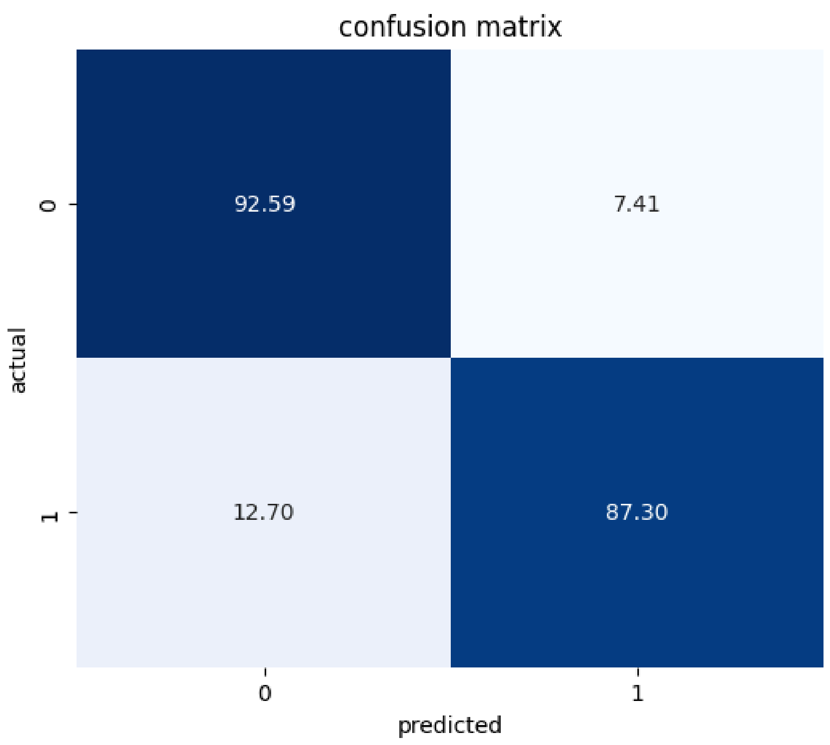
Figure 2: Pair plot of tumor diagnosis w.r.t. 5 different features

2.3. Fit a logistic regression model to predict the dependent variable. Evaluate the model's performance and determine how often it correctly predicts dependent variable (class).

Output

```
accuracy: 0.9064327485380117
classification report:
              precision    recall  f1-score   support

           0       0.93      0.93      0.93       108
           1       0.87      0.87      0.87        63

    accuracy                           0.91       171
   macro avg       0.90      0.90      0.90       171
weighted avg       0.91      0.91      0.91       171
```

Using StandardScaler and `saga` solver results in the performance metrics mentioned in the above classification report and the below confusion matrix. It should be noted that after scaling there is no noticeable difference between `liblinear` and `saga`.



2.4. Use statsmodels.Logit to obtain and interpret the p-values for the predictors and determine if any features can be discarded.

statsmodels.Logit Output →

```
                           Logit Regression Results
==============================================================================
Dep. Variable:              Diagnosis   No. Observations:                  398
Model:                          Logit   Df Residuals:                      392
Method:                           MLE   Df Model:                            5
Date:                Wed, 02 Oct 2024   Pseudo R-squ.:                  0.7299
Time:                        15:21:43   Log-Likelihood:                -71.093
converged:                       True   LL-Null:                       -263.17
Covariance Type:            nonrobust   LLR p-value:                 7.676e-81
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         18.5071     10.027      1.846      0.065      -1.146      38.160
feature_1     -2.9623      1.250     -2.369      0.018      -5.413      -0.512
feature_4      0.0365      0.014      2.557      0.011       0.009       0.065
feature_8    119.6588     19.489      6.140      0.000      81.461     157.856
feature_10   -78.5012     47.676     -1.647      0.100    -171.945      14.942
feature_15  -137.8380     70.149     -1.965      0.049    -275.327      -0.349
==============================================================================

Possibly complete quasi-separation: A fraction 0.11 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.
```

From the selected features, `feature_10` has a p_value of 0.100 > 0.05 and hence can be removed.

3 Logistic regression First/Second-Order Methods

3.1. Use the code given in `listing 4` to generate data. Here, variable `y` and `X` are class labels and corresponding feature values, respectively.

```
# listing 4
import numpy as np
import matplotlib.pyplot as plt
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)
```

3.2. Implement batch gradient descent to update the weights for the given dataset over 20 iterations. State the method used to initialize the weights and reason for your selection. [15 marks]

```
# batch gradient descent
def batch_gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    losses = []

    for i in range(iterations):
        z = np.dot(X, weights)
        predictions = sigmoid(z)

        # gradient calculation
        gradient = np.dot(X.T, (predictions - y)) / m
        weights -= learning_rate * gradient

        # compute and store the loss
        loss = compute_loss(X, y, weights)
        losses.append(loss)

    return weights, losses
```

Initializing the weights with small random values since random initialization avoids symmetry issues. The weights are initialized randomly using np.random.randn(). This will be effective for learning different features rather than being stuck in symmetry.

3.3. Specify the loss function you have used and state reason for your selection. [5 marks]

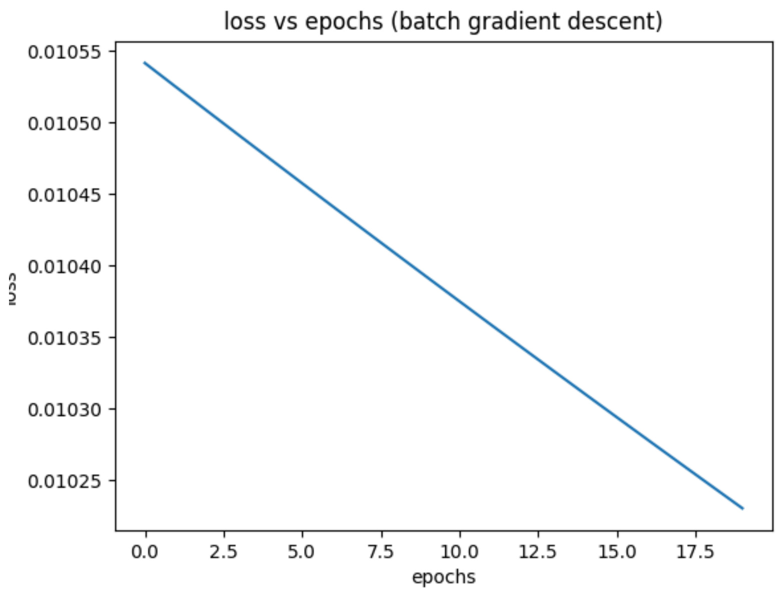
The binary cross-entropy (BCE/log-loss) is used here, as it is a standard and optimized loss function for logistic regression/binary classification tasks.

$$L = -\frac{1}{m}\sum_{i=1}^{m}[(y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

```
def compute_loss(X, y, weights):
    predictions = 1 / (1 + np.exp(-X @ weights))  # sigmoid function
    # compute loss using clipped predictions
    # to avoid inf errors during center shifting
    epsilon = 1e-15
    predictions = np.clip(predictions, epsilon, 1 - epsilon)
    loss = -np.mean(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    return loss
```

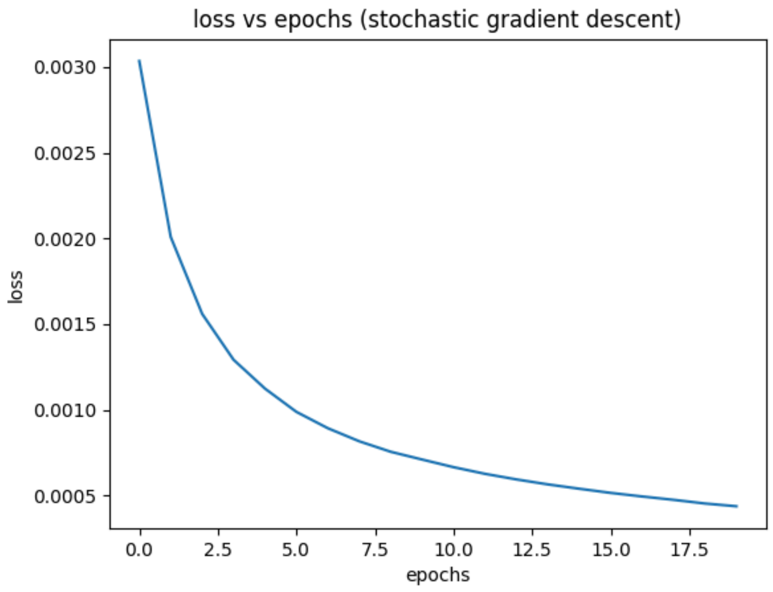
A more simpler and straightforward MSE can also be used but BCE is more aligned with the sigmoid function used in the logistic regression, making gradient updates efficient. Penalizing confident wrong predictions heavily is advantageous when it comes to logistic regression.

3.4. Plot the loss with respect to number of iterations.



3.5. Repeat step 2 for Stochastic Gradient descent. [5 marks]

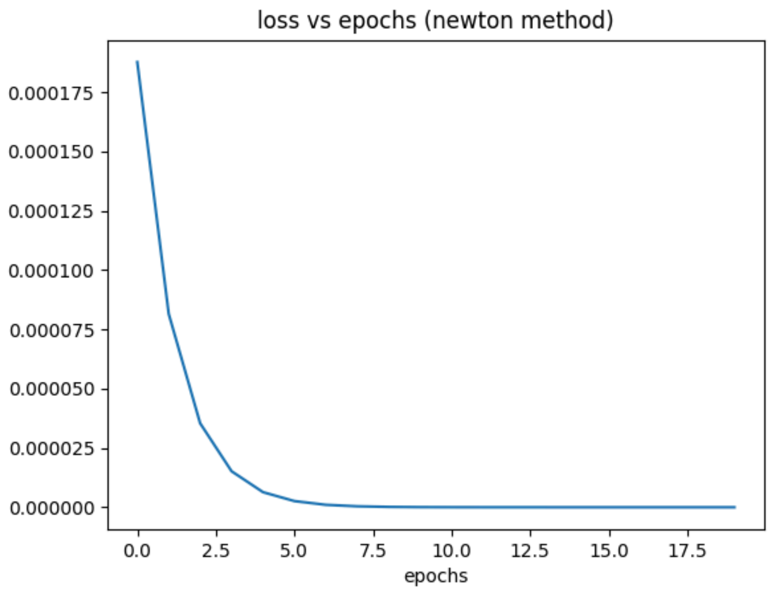
```
# stochastic gradient descent
def stochastic_gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    losses = []

    for i in range(iterations):
        for j in range(m):
            rand_index = np.random.randint(0, m)
            x_i = X[rand_index : rand_index + 1]
            y_i = y[rand_index : rand_index + 1]
            z = np.dot(x_i, weights)
            predictions = sigmoid(z)

            gradient = np.dot(x_i.T, (predictions - y_i))
            weights -= learning_rate * gradient

        # Compute and store the loss
        loss = compute_loss(X, y, weights)
        losses.append(loss)

    return weights, losses
```



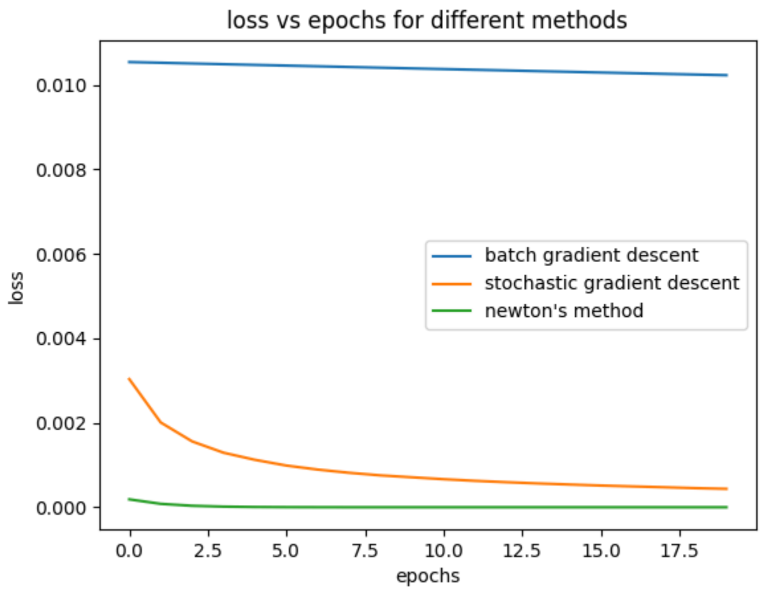
3.6. Implement Newton's method to update the weights for the given dataset over 20 iterations.

```python
# newton's method
def newton_method(X, y, weights, iterations):
    m = len(y)
    losses = []

    for i in range(iterations):
        z = np.dot(X, weights)
        predictions = sigmoid(z)

        gradient = np.dot(X.T, (predictions - y)) / m

        # hessian matrix
        diag = predictions * (1 - predictions)
        H = np.dot(X.T, diag[:, None] * X) / m

        # newton's update
        weights -= np.linalg.inv(H).dot(gradient)

        # compute and store the loss
        loss = compute_loss(X, y, weights)
        losses.append(loss)

    return weights, losses
```

3.7. Plot the loss with respect to number of iterations.



3.8. Plot the loss with respect to number of iterations for Gradient descent, Stochastic Gradient descent and Newton method in a single plot. Comment on your results.



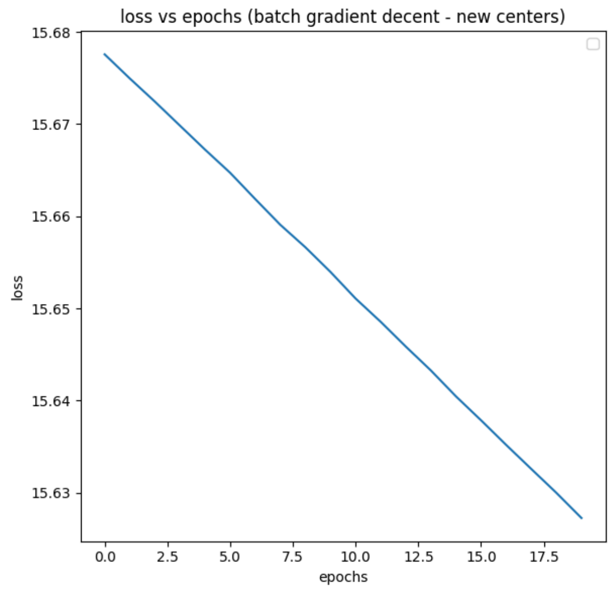
Observations

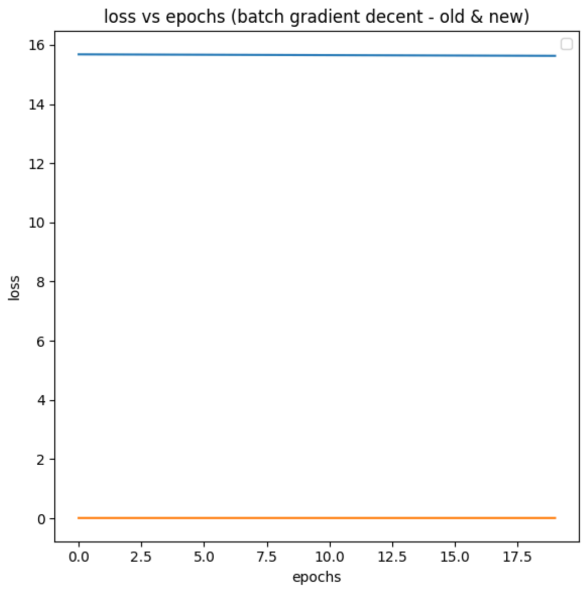
- Batch gradient descent is the most stable but slowest to converge. Thus it requires more iterations and not practical to use on large datasets.
- Stochastic gradient descent has more noise in the updates but converges faster initially. This is problamatic in small datasets but can be useful in large dataset due to efficiency in computational requirements.
- Newton's method converges much faster than either batch gradient descent or stochastic gradient descent. This is expected due to the application of second-order information (Hessian matrix) to find the optimal weights, which allows for more precise updates.
- Newton's method is both fast and relatively stable but from the algorithm and running time we can infer that it is computationally intensive for large datasets.

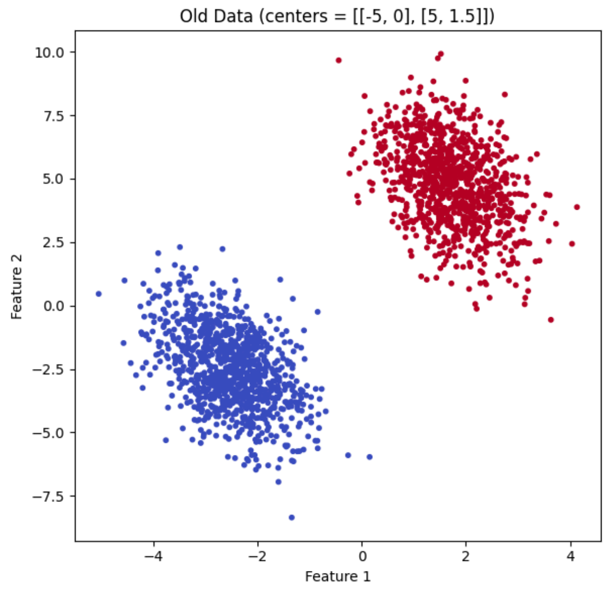
3.9. Propose two approaches to decide number of iterations for Gradient descent and Newton's method

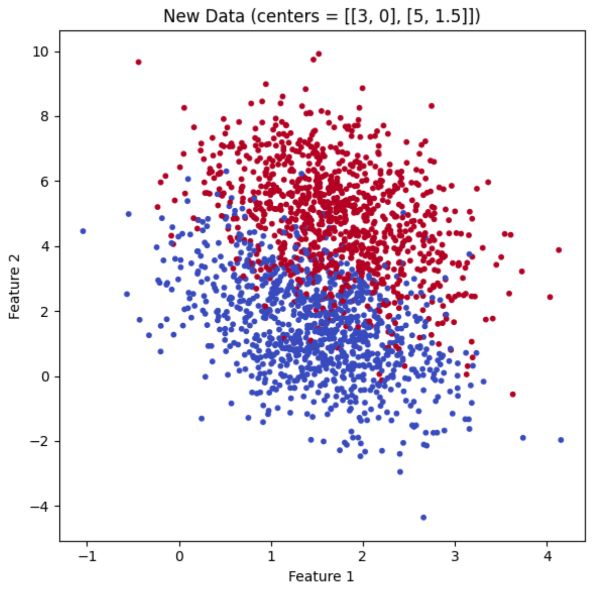
1. Early Stopping based on threshold: Monitor the loss and stop once it stops decreasing significantly (i.e., when changes are below a certain threshold). Useful for gradient descent functions as they produce noise after overfitting.
2. Cross-validation: Split the dataset into training and validation sets, and stop training when performance on the validation set degrades, indicating overfitting. Useful for methods that converge quickly like Newton's method.

3.10. 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.









Changing the centers to [[3, 0], [5, 1.5]] results in data distribution changes. We can observe that the classes have become less separable now and this leads to a significant increase in loss values for the

Batch Gradient Descent. In our case this allows prediction values that are exactly 0 or 1 in the predictions array and makes the convergence fail on a usual BCE loss function due to divide by zero error. This lead to an implementation of clipping in the loss function to avoid going below a certain very small value to make the fitting process work. This slow and suboptimal convergence is due to the complicated distribution of data caused by the center change. We need to move towards more optimized methods of preprocessing the data in order to have optimal performance.

Appendix A - Python code for Assignment 2

```python
"""
en3150 assignment 02: learning from data and related challenges and classification
instructor: sampath k. perera
author: thuvaragan s.
index no.: 210657G
"""

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler


def listing1n2():
    df = sns.load_dataset("penguins")
    df.dropna(inplace=True)

    selected_classes = ["Adelie", "Chinstrap"]
    df_filtered = df[df["species"].isin(selected_classes)].copy()

    le = LabelEncoder()

    y_encoded = le.fit_transform(df_filtered["species"])

    df_filtered["class_encoded"] = y_encoded

    print(df_filtered[["species", "class_encoded"]])

    y = df_filtered["class_encoded"]
    X = df_filtered.drop(["species", "island", "sex", "class_encoded"], axis=1)

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

    logreg = LogisticRegression(solver="saga")
    logreg.fit(X_train, y_train)
    y_pred = logreg.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    print("saga accuracy:", accuracy)
    print(logreg.coef_, logreg.intercept_)

    logreg2 = LogisticRegression(solver="liblinear")
    logreg2.fit(X_train, y_train)
    y_pred2 = logreg2.predict(X_test)

    accuracy2 = accuracy_score(y_test, y_pred2)
    print("liblinear accuracy:", accuracy2)
    print(logreg2.coef_, logreg2.intercept_)

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

    X_train_scaled, X_test_scaled, y_train, y_test = train_test_split(
        X_scaled, y, test_size=0.2, random_state=42
    )

    logreg_liblinear = LogisticRegression(solver="liblinear")
    logreg_liblinear.fit(X_train_scaled, y_train)

    accuracy_liblinear = accuracy_score(y_test,
logreg_liblinear.predict(X_test_scaled))
    print("liblinear accuracy with scaling:", accuracy_liblinear)
    print(logreg_liblinear.coef_, logreg2.intercept_)

    logreg_saga = LogisticRegression(solver="saga")
    logreg_saga.fit(X_train_scaled, y_train)

    accuracy_saga = accuracy_score(y_test, logreg_saga.predict(X_test_scaled))
    print("saga accuracy with scaling:", accuracy_saga)
    print(logreg_saga.coef_, logreg2.intercept_)


def listing3fixed():
    df = sns.load_dataset("penguins")
    df.dropna(inplace=True)

    selected_classes = ["Adelie", "Chinstrap"]
    df_filtered = df[
        df["species"].isin(selected_classes)
    ].copy()  # make a copy to avoid the warning

    le = LabelEncoder()
    y_encoded = le.fit_transform(df_filtered["species"])
    df_filtered["class_encoded"] = y_encoded

    print(df_filtered.head())

    # BUG:
    # X = df_filtered.drop(["species", "class_encoded"], axis=1)
    # fixed by dropping excess categorical data as in listing1
    X = df_filtered.drop(["species", "island", "sex", "class_encoded"], axis=1)
    y = df_filtered["class_encoded"]  # target variable
    # alternate fix - using one-hot encoding to encode other categorical features

    X.head()

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

    logreg = LogisticRegression(solver="saga")
    logreg.fit(X_train, y_train)
```

```python
    y_pred = logreg.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    print("accuracy:", accuracy)

    print(logreg.coef_, logreg.intercept_)


def lr_on_real_world_data():
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-
wisconsin/wdbc.data"
    columns = ["ID", "Diagnosis"] + [f"feature_{i}" for i in range(1, 31)]
    data = pd.read_csv(url, header=None, names=columns)

    # map diagnosis to binary
    data["Diagnosis"] = data["Diagnosis"].map({"M": 1, "B": 0})

    unionset = data[["Diagnosis"] + [f"feature_{i}" for i in range(1, 31)]]

    # correlation matrix
    correlation_matrix = unionset.corr()
    fig1 = plt.figure(figsize=(24, 18))
    sns.heatmap(correlation_matrix, annot=True, fmt=".2f", linewidths=0.5)
    plt.title("correlation matrix heatmap")
    fig1.savefig("./assignments/assets/a2q2fig1.png", bbox_inches="tight")
    # plt.show()

    # select up to 5 features for analysis
    features = ["feature_1", "feature_4", "feature_8", "feature_10", "feature_15"]
    subset = data[["Diagnosis"] + features]

    # pair plot using seaborn
    sns.pairplot(subset, hue="Diagnosis").savefig("./assignments/assets/
a2q2fig2.png")
    # sns.pairplot(subset, hue="Diagnosis")
    # plt.show()

    X = subset[features]
    y = subset["Diagnosis"]

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42
    )

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

    # fit logistic regression on scaled data
    model = LogisticRegression(solver="saga")
    # model = LogisticRegression(solver="liblinear")
    model.fit(X_train_scaled, y_train)
    # model.fit(X_train, y_train)

    # evaluate the model
    y_pred = model.predict(X_test_scaled)
    # y_pred = model.predict(X_test)

    # evaluate the model's performance
    accuracy = accuracy_score(y_test, y_pred)
    conf_matrix = confusion_matrix(y_test, y_pred)
    classification_rep = classification_report(y_test, y_pred)
    cm_percentage = (
        conf_matrix.astype("float") / conf_matrix.sum(axis=1)[:, np.newaxis] * 100
    )

    # Plot heatmap
    fig3 = plt.figure(figsize=(6, 5))
    sns.heatmap(cm_percentage, annot=True, fmt=".2f", cmap="Blues", cbar=False)
    # sns.heatmap(
    #     conf_matrix_percent,
    #     annot=True,
    #     fmt="d",
    #     # cmap="Blues",
    #     xticklabels=[1, 0],
    #     yticklabels=[1, 0],
    # )
    plt.xlabel("predicted")
    plt.ylabel("actual")
    plt.title("confusion matrix")
    fig3.savefig("./assignments/assets/a2q2fig3.png", bbox_inches="tight")
    plt.show()

    print(f"accuracy: {accuracy}")
    # print("confusion matrix:")
    # print(conf_matrix)
    print("classification report:")
    print(classification_rep)

    # fit logistic regression using statsmodels
    X_train_const = sm.add_constant(X_train)
    logit_model = sm.Logit(y_train, X_train_const)
    result = logit_model.fit()

    print(result.summary())


def listing4():
    # 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)

    weights = np.random.randn(X.shape[1])
    learning_rate = 0.01
    iterations = 20

    def sigmoid(z):
        return 1 / (1 + np.exp(-z))
```

```python
def compute_loss(X, y, weights):
    z = np.dot(X, weights)
    predictions = sigmoid(z)
    epsilon = 1e-15
    predictions = np.clip(predictions, epsilon, 1 - epsilon)
    # compute loss using clipped predictions
    loss = -np.mean(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    return loss

def batch_gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    losses = []

    for i in range(iterations):
        z = np.dot(X, weights)
        predictions = sigmoid(z)

        # gradient calculation
        gradient = np.dot(X.T, (predictions - y)) / m
        weights -= learning_rate * gradient

        # compute and store the loss
        loss = compute_loss(X, y, weights)
        losses.append(loss)

    return weights, losses

# run batch gradient descent
weights_bgd, losses_bgd = batch_gradient_descent(
    X, y, weights, learning_rate, iterations
)

def stochastic_gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    losses = []

    for i in range(iterations):
        for j in range(m):
            rand_index = np.random.randint(0, m)
            x_i = X[rand_index : rand_index + 1]
            y_i = y[rand_index : rand_index + 1]
            z = np.dot(x_i, weights)
            predictions = sigmoid(z)

            gradient = np.dot(x_i.T, (predictions - y_i))
            weights -= learning_rate * gradient

        # Compute and store the loss
        loss = compute_loss(X, y, weights)
        losses.append(loss)

    return weights, losses

# run sgd
weights_sgd, losses_sgd = stochastic_gradient_descent(
    X, y, weights, learning_rate, iterations
)

def newton_method(X, y, weights, iterations):
    m = len(y)
    losses = []

    for i in range(iterations):
        z = np.dot(X, weights)
        predictions = sigmoid(z)

        gradient = np.dot(X.T, (predictions - y)) / m

        # hessian matrix
        diag = predictions * (1 - predictions)
        H = np.dot(X.T, diag[:, None] * X) / m

        # newton's update
        weights -= np.linalg.inv(H).dot(gradient)

        # compute and store the loss
        loss = compute_loss(X, y, weights)
        losses.append(loss)

    return weights, losses

# run newton's method
weights_newton, losses_newton = newton_method(X, y, weights, iterations)

f1 = plt.figure(1)
plt.plot(range(iterations), losses_bgd)
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss vs epochs (batch gradient descent)")
f1.savefig("./assignments/assets/a2q3fig1.png")
# f1.show()

f2 = plt.figure(2)
plt.plot(range(iterations), losses_sgd)
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss vs epochs (stochastic gradient descent)")
f2.savefig("./assignments/assets/a2q3fig2.png")
# f2.show()

f3 = plt.figure(3)
plt.plot(range(iterations), losses_newton)
plt.xlabel("epochs")
plt.ylabel("loss")
plt.title("loss vs epochs (newton method)")
f3.savefig("./assignments/assets/a2q3fig3.png")
# f3.show()

f4 = plt.figure(4)
plt.plot(range(iterations), losses_bgd, label="batch gradient descent")
plt.plot(range(iterations), losses_sgd, label="stochastic gradient descent")
plt.plot(range(iterations), losses_newton, label="newton's method")
```

```
    plt.xlabel("epochs")
    plt.ylabel("loss")
    plt.legend()
    plt.title("loss vs epochs for different methods")
    f4.savefig("./assignments/assets/a2q3fig4.png")
    # f4.show()

    # change the centers
    print("changing centers")
    centers_new = [[3, 0], [5, 1.5]]
    X_new, y_new = make_blobs(n_samples=2000, centers=centers_new, random_state=5)
    X_new = np.dot(X_new, transformation)

    # apply batch gradient descent again
    weights_new, losses_new = batch_gradient_descent(
        X_new, y_new, weights, learning_rate, iterations
    )

    f5, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
    ax1.set_title("loss vs epochs (batch gradient decent - new centers)")
    ax1.set_xlabel("epochs")
    ax1.set_ylabel("loss")
    ax1.legend()
    ax1.plot(range(iterations), losses_new, label="bgd (new center)")
    ax2.set_title("loss vs epochs (batch gradient decent - old & new)")
    ax2.set_xlabel("epochs")
    ax2.set_ylabel("loss")
    ax2.legend()
    ax2.plot(range(iterations), losses_new, label="bgd (new center)")
    ax2.plot(range(iterations), losses_bgd, label="bgd (initial)")
    plt.tight_layout()
    f5.savefig("./assignments/assets/a2q3fig5.png")
    plt.show()

    f6, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
    ax1.scatter(X[:, 0], X[:, 1], c=y, cmap="coolwarm", s=10)
    ax1.set_title("Old Data (centers = [[-5, 0], [5, 1.5]])")
    ax1.set_xlabel("Feature 1")
    ax1.set_ylabel("Feature 2")
    ax2.scatter(X_new[:, 0], X_new[:, 1], c=y_new, cmap="coolwarm", s=10)
    ax2.set_title("New Data (centers = [[3, 0], [5, 1.5]])")
    ax2.set_xlabel("Feature 1")
    ax2.set_ylabel("Feature 2")
    plt.tight_layout()
    f6.savefig("./assignments/assets/a2q3fig6.png")
    plt.show()


if __name__ == "__main__":
    # uncomment the section of the assignment you want to test and run the file
    # of course all can be tested at the same time
    # listing1n2()
    # listing3()
    lr_on_real_world_data()
    # listing4()
```