

Assignment 02 - Learning from data and related challenges and classification

EN3150 Pattern Recognition



Name:	Gunawardane E.R.N.H.
Index No:	210200C
Date	02.10.2024

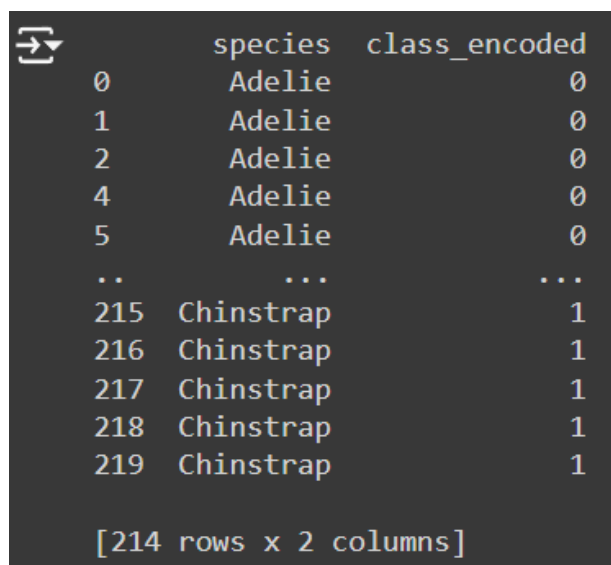
Contents

1	Logistic regression	3
1.1	Loading the data	3
1.2	Purpose of <code>y_encoded = le.fit_transform(df_filtered['species'])</code>	3
1.3	Purpose of <code>X = df.drop(['species', 'island', 'sex'], axis=1)</code>	4
1.4	Why we cannot use "island" and "sex" features?	4
1.5	Training of the Logistic Regression Model	4
1.6	<code>random_state=42</code>	4
1.7	Why is accuracy low? Why does the saga solver perform poorly?	5
1.7.1	Why is accuracy low?	5
1.7.2	Why does the saga solver perform poorly?	5
1.8	Changing the solver to liblinear	5
1.9	Why does the "liblinear" solver perform better than "saga" solver?	5
1.10	Comparison with the feature scaling with the Standard Scalar	6
1.11	Running the code in Listing 3 and solve the issue	7
1.12	Scaling and Encoding in categorical features	8
1.13	Logistic Regression Model	9
1.14	Given Values	9
2	Logistic regression on real world data	10
2.1	Loading the Wine quality data set from the UCI Machine Learning Repository	10
2.2	Correlation matrix and the pair plot	10
2.3	Logistic Regression model and Evaluation	11
2.4	P value comparison	13
3	Logistic regression First/Second-Order Methods	13
3.1	Generte data	13
3.2	Implementing Batch Gradient Descent	13
3.3	Utilized loss function	15
3.3.1	Selected Loss Function	15
3.3.2	Reason for Selection	15
3.3.3	Loss Function	15
3.4	Plot the loss with respect to number of iterations	16
3.5	Stochastic Gradient Descent (SGD)	16
3.6	Newton's Method	18
3.7	Newton's Method Loss w.r.t. number of iterations	19
3.8	Combined Loss Plot	19
3.9	Approach to Decide Number of Iterations	20
3.10	Changing Centers for Data	20

1 Logistic regression

1.1 Loading the data

```
1 import seaborn as sns
2 import pandas as pd
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import LabelEncoder
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.metrics import accuracy_score
7 # Load the penguins dataset
8 df = sns.load_dataset("penguins")
9 df.dropna(inplace=True)
10 # Filter rows for 'Adelie' and 'Chinstrap' classes
11 selected_classes = ['Adelie', 'Chinstrap']
12 df_filtered = df[df['species'].isin(selected_classes)].copy() # Make a
13 # copy to avoid the warning
14 # Initialize the LabelEncoder
15 le = LabelEncoder()
16 # Encode the species column
17 y_encoded = le.fit_transform(df_filtered['species'])
18 df_filtered['class_encoded'] = y_encoded
19 # Display the filtered and encoded DataFrame
20 print(df_filtered[['species', 'class_encoded']])
21 # Split the data into features (X) and target variable (y)
22 y = df_filtered['class_encoded'] # Target variable
23 X = df_filtered.drop(['species', 'island', 'sex', 'class_encoded'], axis=1)
```



	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 1: After loading the data

1.2 Purpose of `y_encoded = le.fit_transform(df_filtered['species'])`

The line `y_encoded = le.fit_transform(df_filtered['species'])` is used to convert the categorical target variable 'species' into numerical values, such as assigning 0 to 'Adelie' and 1 to

'Chinstrap'. This encoding is necessary because logistic regression models require numerical input, so each species is represented by an integer.

1.3 Purpose of `X = df.drop(['species', 'island', 'sex'], axis=1)`

The 'species', 'island', and 'sex' columns are eliminated from the dataset by this line since the logistic regression model cannot utilize them as features.

species: The original labels for the species, which have already been swapped out for their numeric values that have been encoded.

island and **sex:** These are unconverted categorical variables that do not have a number representation. These columns are eliminated unless correctly encoded since logistic regression needs numerical input.

1.4 Why we cannot use "island" and "sex" features?

The attributes 'island' and 'sex' are categorical variables that lack a simple numerical representation. Given that logistic regression requires numerical inputs, including them without using the appropriate encoding, such one-hot encoding, may result in unreliable findings. Although these variables may be encoded, doing so may make the dataset more sparse and complicated. We can lessen the possibility of overfitting, streamline the model, and maybe enhance its accuracy and generalization by eliminating these variables.

1.5 Training of the Logistic Regression 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,
3     random_state=42)
4 #Train the logistic regression model. Here we are using saga solver to
5     learn weights.
6 logreg = LogisticRegression(solver='saga')
7 logreg.fit(X_train, y_train)
8 # Predict on the testing data
9 y_pred = logreg.predict(X_test)
10 # Evaluate the model
11 accuracy = accuracy_score(y_test, y_pred)
12 print("Accuracy:", accuracy)
13 print(logreg.coef_, logreg.intercept_)
```

1.6 `random_state=42`

By managing the random number generator during the split process, the `random_state=42` setting guarantees the reproducibility of the train-test split. This ensures that each time the code is executed, the identical training and testing sets are generated. Randomization is used to divide the data into training and testing sets. We make sure the split is constant each time we execute the function by setting `random_state=42`. Inconsistent results would arise from splitting the data differently for every iteration if a random state wasn't specified. Since 42 is an arbitrary number, any value (such as 42, 0, 1) will suffice as long as it stays constant to guarantee that the split is the same between runs.

1.7 Why is accuracy low? Why does the saga solver perform poorly?

1.7.1 Why is accuracy low?

The two target classes—the Adelie and Chinstrap penguin species—are disproportionately large in number, with one class being far larger than the other. This imbalance may make it difficult for the model to anticipate the minority class, which would result in subpar performance. Furthermore, if there is insufficient training data, the model might not be able to identify important patterns, which might lead to underfitting—a situation in which the model performs badly on both training and testing data. Moreover, a linear relationship between the characteristics and the target variable's log chances is assumed by logistic regression. The model's performance can be negatively impacted if this connection is not linear.

1.7.2 Why does the saga solver perform poorly?

The choice of the saga solver in logistic regression might also explain the model's poor performance in this scenario. The saga solver is specifically designed for handling large datasets and works efficiently with both L1 and L2 regularization. However, with a smaller dataset like the penguins dataset, the saga solver may not be the optimal choice. It could introduce unnecessary complexity, overcomplicating the learning process and potentially affecting convergence. The slower convergence and suboptimal weight learning associated with the saga solver in small datasets can negatively impact the model's predictive performance.

1.8 Changing the solver to liblinear

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

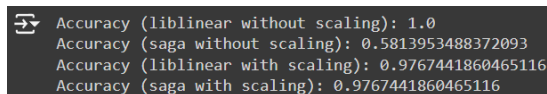
The accuracy is 100%

1.9 Why does the "liblinear" solver perform better than "saga" solver?

Because the liblinear solver is optimized for short datasets and excels at binary classification tasks, using it increased accuracy. It makes use of a more straightforward optimization method that prevents overfitting and is effective with less data points. Due to the limited size of the penguin dataset, liblinear is able to train the model parameters more efficiently, resulting in nearly flawless accuracy.

1.10 Comparision with the feature scaling with the Standard Scalar

```
1 from sklearn.preprocessing import StandardScaler
2 from sklearn.linear_model import LogisticRegression
3 from sklearn.metrics import accuracy_score
4 from sklearn.model_selection import train_test_split
5
6 # Split the data into training and testing sets
7 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
8     random_state=42)
9
10 # Apply Standard Scaler to the features
11 scaler = StandardScaler()
12 X_train_scaled = scaler.fit_transform(X_train)
13 X_test_scaled = scaler.transform(X_test)
14
15 # Train and evaluate logistic regression using liblinear without scaling
16 logreg_liblinear = LogisticRegression(solver='liblinear')
17 logreg_liblinear.fit(X_train, y_train)
18 y_pred_liblinear = logreg_liblinear.predict(X_test)
19 accuracy_liblinear = accuracy_score(y_test, y_pred_liblinear)
20
21 # Train and evaluate logistic regression using saga without scaling
22 logreg_saga = LogisticRegression(solver='saga')
23 logreg_saga.fit(X_train, y_train)
24 y_pred_saga = logreg_saga.predict(X_test)
25 accuracy_saga = accuracy_score(y_test, y_pred_saga)
26
27 # Train and evaluate logistic regression using liblinear with scaling
28 logreg_liblinear_scaled = LogisticRegression(solver='liblinear')
29 logreg_liblinear_scaled.fit(X_train_scaled, y_train)
30 y_pred_liblinear_scaled = logreg_liblinear_scaled.predict(X_test_scaled)
31 accuracy_liblinear_scaled = accuracy_score(y_test, y_pred_liblinear_scaled)
32
33 # Train and evaluate logistic regression using saga with scaling
34 logreg_saga_scaled = LogisticRegression(solver='saga')
35 logreg_saga_scaled.fit(X_train_scaled, y_train)
36 y_pred_saga_scaled = logreg_saga_scaled.predict(X_test_scaled)
37 accuracy_saga_scaled = accuracy_score(y_test, y_pred_saga_scaled)
38
39 # Print out the accuracy values for comparison
40 print(f"Accuracy (liblinear without scaling): {accuracy_liblinear}")
41 print(f"Accuracy (saga without scaling): {accuracy_saga}")
42 print(f"Accuracy (liblinear with scaling): {accuracy_liblinear_scaled}")
43 print(f"Accuracy (saga with scaling): {accuracy_saga_scaled}")
```



```
Accuracy (liblinear without scaling): 1.0
Accuracy (saga without scaling): 0.5813953488372093
Accuracy (liblinear with scaling): 0.9767441860465116
Accuracy (saga with scaling): 0.9767441860465116
```

Figure 2: Accuracy of the solvers with and without feature scaling

In the absence of feature scaling

- On smaller datasets, `liblinear` works well, however, variations in feature sizes have an impact on it.
- Because it has trouble handling features on various sizes, `Saga` performs badly while scaling.

Using Feature Scaling

- Scaling guarantees that all features have comparable ranges, which improves the performance of solvers like `Saga`. With standard scaling, all features have means of 0 and standard deviation of 1.
- Although there is less accuracy loss with scaling for `liblinear`, it might still be advantageous.

Why Scaling Matters

- `liblinear`: Performs quite well in the absence of scaling but may somewhat degrade in its presence.
- `Saga`: Uses gradient algorithms, thus scaling improves performance and allows for faster convergence.

1.11 Running the code in Listing 3 and solve the issue

A `ValueError` was encountered during the model fitting stage when running the code supplied in Listing 3. Categorical variables like `island` and `sex` were still in string format, according to the warning message "could not convert string to float," while the Logistic Regression model in `sklearn` required numerical inputs. The island known as "Dream" specifically could not be made to float.

We have to convert all category columns into numerical format in order to fix issue. The `LabelEncoder` from the `sklearn.preprocessing` module was used to do this. Additionally, feature scaling was applied to standardize the data using `StandardScaler` to improve model performance, especially for solvers like `saga`.

```
1 import seaborn as sns
2 import pandas as pd
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import LabelEncoder, StandardScaler
5 from sklearn.linear_model import LogisticRegression
6 from sklearn.metrics import accuracy_score
7
8 # Load the penguins dataset and remove rows with missing values
9 df = sns.load_dataset("penguins")
10 df.dropna(inplace=True)
11
12 # Filter for only 'Adelie' and 'Chinstrap' species
13 selected_classes = ['Adelie', 'Chinstrap']
14 df_filtered = df[df['species'].isin(selected_classes)].copy() # Copy to
15 # avoid potential SettingWithCopyWarning
16
17 # Initialize the LabelEncoder to convert categorical values to numerical
18 le = LabelEncoder()
```

```

19 # Encode the 'species' column (target variable)
20 df_filtered['class_encoded'] = le.fit_transform(df_filtered['species'])
21
22 # Encode other categorical columns ('island', 'sex') for feature use
23 df_filtered['island_encoded'] = le.fit_transform(df_filtered['island'])
24 df_filtered['sex_encoded'] = le.fit_transform(df_filtered['sex'])
25
26 # Remove the original categorical columns and any unnecessary ones
27 X = df_filtered.drop(['species', 'class_encoded', 'island', 'sex'], axis
28                      =1) # Features
29 y = df_filtered['class_encoded'] # Target
30
31 # Split the dataset into training and testing sets (80% training, 20%
32   testing)
33 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
34                                                    random_state=42)
35
36 # Standardize the feature values for better model performance
37 scaler = StandardScaler()
38 X_train = scaler.fit_transform(X_train)
39 X_test = scaler.transform(X_test)
40
41 # Train the logistic regression model using the saga solver
42 logreg = LogisticRegression(solver='saga')
43 logreg.fit(X_train, y_train)
44
45 # Make predictions on the testing data
46 y_pred = logreg.predict(X_test)
47
48 # Evaluate the model's performance by calculating accuracy
49 accuracy = accuracy_score(y_test, y_pred)
50 print("Accuracy:", accuracy)
51
52 # Display the model's coefficients and intercept
53 print(logreg.coef_, logreg.intercept_)

```

1.12 Scaling and Encoding in categorical features

It is not suitable to use label encoding followed by feature scaling techniques like Standard Scaling or Min-Max Scaling when working with categorical features like "red," "blue," and "green." An arbitrary numerical value ('red' = 0, 'blue' = 1, 'green' = 2) is assigned via label encoding, but there is no clear ordinal link between these values. Scaling these figures might establish erroneous assumptions about their relationships. One-hot encoding ensures that the model treats each color independently, without implying any order or relationship.

Question 2

1.13 Logistic Regression Model

The logistic regression model is defined as:

$$P(y = 1|x_1, x_2) = \frac{1}{1 + e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)}}$$

where:

- $P(y = 1|x_1, x_2)$ is the probability of receiving an A+,
- w_0, w_1, w_2 are the coefficients,
- x_1 is the number of hours studied,
- x_2 is the undergraduate GPA.

1.14 Given Values

- $w_0 = -5.9$
- $w_1 = 0.06$
- $w_2 = 1.5$
- $x_1 = 50$ hours
- $x_2 = 3.6$ GPA

1. Estimated Probability Calculation

$$\begin{aligned} z &= w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 \\ z &= -5.9 + (0.06 \cdot 50) + (1.5 \cdot 3.6) \\ z &= -5.9 + 3 + 5.4 = 2.5 \end{aligned}$$

$$P(y = 1|x_1, x_2) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-2.5}} \approx \frac{1}{1 + 0.0821} \approx \frac{1}{1.0821} \approx 0.923$$

Estimated probability is approximately 0.923.

2. Hours of Study for 60% Probability

We want to find the number of hours x_1 required to achieve a 60% chance of receiving an A+:

$$P(y = 1|x_1, x_2) = 0.6$$

Using the logistic model:

$$\begin{aligned} 0.6 &= \frac{1}{1 + e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)}} \\ 1 + e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)} &= \frac{1}{0.6} \\ e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)} &= \frac{1}{0.6} - 1 = \frac{0.4}{0.6} = \frac{2}{3} \\ -(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2) &= \ln\left(\frac{2}{3}\right) \end{aligned}$$

Thus:

$$\begin{aligned}w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 &= -\ln\left(\frac{2}{3}\right) \\-5.9 + 0.06 \cdot x_1 + 1.5 \cdot 3.6 &= -\ln\left(\frac{2}{3}\right) \\-5.9 + 0.06 \cdot x_1 + 5.4 &= -\ln\left(\frac{2}{3}\right) \\0.06 \cdot x_1 - 0.5 &= -\ln\left(\frac{2}{3}\right) \\0.06 \cdot x_1 &= 0.405 + 0.5 = 0.905 \\x_1 &= \frac{0.905}{0.06} \approx 15.083\end{aligned}$$

Need to study 15.1 hours approximately.

2 Logistic regression on real world data

2.1 Loading the Wine quality data set from the UCI Machine Learning Repository

```
1 from ucimlrepo import fetch_ucirepo
2
3 # fetch dataset
4 wine_quality = fetch_ucirepo(id=186)
5
6 # data (as pandas dataframes)
7 X = wine_quality.data.features
8 y = wine_quality.data.targets
9
10 # metadata
11 print(wine_quality.metadata)
12
13 # variable information
14 print(wine_quality.variables)
```

2.2 Correlation matrix and the pair plot

Select the features 'alcohol', 'sulphates', 'fixed acidity', 'density', 'chlorides'.

```
1 from ucimlrepo import fetch_ucirepo
2 import pandas as pd
3 import seaborn as sns
4 import matplotlib.pyplot as plt
5
6 # Convert features and target to a single dataframe
7 data = pd.concat([X, y], axis=1)
8
9 # Create binary target variable (1 for quality >= 6, 0 for quality < 6)
10 data['quality_binary'] = (data['quality'] >= 6).astype(int)
11
```

```

12 # Select up to 5 features for the analysis
13 selected_features = ['alcohol', 'sulphates', 'fixed_acidity', '
    total_sulfur_dioxide', 'chlorides']
14
15 # Correlation matrix
16 correlation_matrix = data[selected_features].corr()
17 print(correlation_matrix)
18
19 # Pairplot using seaborn
20 sns.pairplot(data[selected_features + ['quality_binary']], hue='
    quality_binary')
21 plt.show()

```

	alcohol	sulphates	fixed_acidity	\
alcohol	1.000000	-0.003029	-0.095452	
sulphates	-0.003029	1.000000	0.299568	
fixed_acidity	-0.095452	0.299568	1.000000	
total_sulfur_dioxide	-0.265740	-0.275727	-0.329054	
chlorides	-0.256916	0.395593	0.298195	
	total_sulfur_dioxide	chlorides		
alcohol	-0.265740	-0.256916		
sulphates	-0.275727	0.395593		
fixed_acidity	-0.329054	0.298195		
total_sulfur_dioxide	1.000000	-0.279630		
chlorides	-0.279630	1.000000		

Figure 3: Correlation Matrix

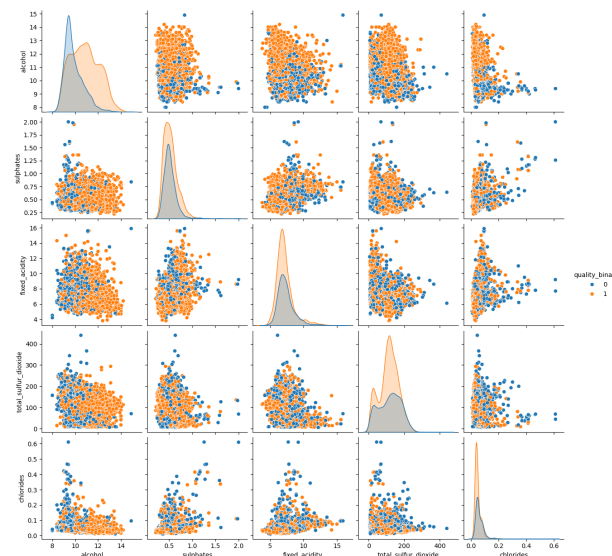


Figure 4: Pair Plots

Comments: All the correlation values are less than 0.5, indicating that all features are relatively independent. Therefore, we can conclude that our feature selection is effective.

2.3 Logistic Regression model and Evaluation

```

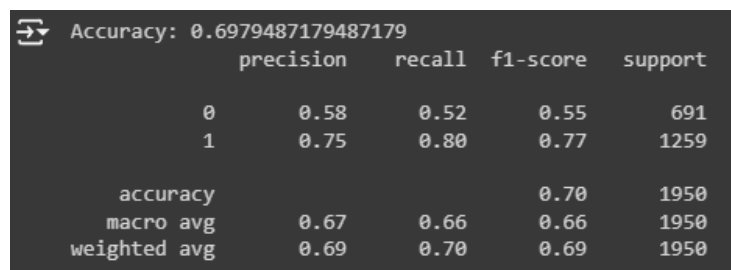
1 from sklearn.model_selection import train_test_split
2 from sklearn.preprocessing import StandardScaler

```

```

3 from sklearn.linear_model import LogisticRegression
4 from sklearn.metrics import accuracy_score, classification_report
5
6 # Assign the selected features to X (input data) and the target variable (
   quality_binary) to y
7 X = data[selected_features]
8 y = data['quality_binary']
9
10 # Split the dataset into training and testing sets, reserving 30% for
   testing and using a random seed for reproducibility
11 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
   random_state=42)
12
13 # Standardize the training and testing feature sets to have zero mean and
   unit variance
14 scaler = StandardScaler()
15 X_train_scaled = scaler.fit_transform(X_train)
16 X_test_scaled = scaler.transform(X_test)
17
18 # Initialize the logistic regression model
19 model = LogisticRegression()
20
21 # Train the model using the scaled training data
22 model.fit(X_train_scaled, y_train)
23
24 # Predict the target variable using the model on the scaled testing data
25 y_pred = model.predict(X_test_scaled)
26
27 # Calculate the accuracy and display it
28 accuracy = accuracy_score(y_test, y_pred)
29 print(f'Accuracy: {accuracy}')
30
31 # Display the classification report, which includes precision, recall, and
   F1-score for the test predictions
32 print(classification_report(y_test, y_pred))

```



Accuracy: 0.6979487179487179				
	precision	recall	f1-score	support
0	0.58	0.52	0.55	691
1	0.75	0.80	0.77	1259
accuracy			0.70	1950
macro avg	0.67	0.66	0.66	1950
weighted avg	0.69	0.70	0.69	1950

Figure 5: Pair Plot Table

The model performs better at identifying wines with higher quality (class 1) with higher precision and recall compared to identifying lower-quality wines (class 0). However, the accuracy of 69.8% indicates there's still room for improvement, possibly by tuning hyperparameters or exploring different algorithms to better handle the class imbalance.

2.4 P value comparison

```
1 import statsmodels.api as sm
2
3 # Add a constant to the features for the intercept term in the logistic
  regression model
4 X_train_const = sm.add_constant(X_train_scaled)
5
6 # Fit logistic regression model using statsmodels
7 logit_model = sm.Logit(y_train, X_train_const)
8 result = logit_model.fit()
9
10 # Print the summary of the logistic regression model
11 print(result.summary())
```

Logit Regression Results						
Dep. Variable:	quality_binary	No. Observations:	4547			
Model:	Logit	Df Residuals:	4541			
Method:	MLE	Df Model:	5			
Date:	Wed, 02 Oct 2024	Pseudo R-squ.:	0.1500			
Time:	16:06:10	Log-Likelihood:	-2551.7			
converged:	True	LL-Null:	-3001.9			
Covariance Type:	nonrobust	LLR p-value:	2.329e-192			
	coef	std err	z	P> z	[0.025	0.975]
const	0.6887	0.036	19.021	0.000	0.618	0.760
x1	1.0504	0.047	22.513	0.000	0.959	1.142
x2	0.2262	0.041	5.584	0.000	0.147	0.306
x3	-0.0403	0.038	-1.068	0.285	-0.114	0.034
x4	0.1389	0.039	3.517	0.000	0.061	0.216
x5	-0.2037	0.043	-4.707	0.000	-0.288	-0.119

Figure 6: P values

Since the P value of the feature x3 is greater than 0.05 this variable (fixed_acidity) is less significant. So, it can be ignored.

3 Logistic regression First/Second-Order Methods

3.1 Generte data

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from sklearn.datasets import make_blobs
5 # Generate synthetic data
6 np.random.seed(0)
7 centers = [[-5, 0], [5, 1.5]]
8 X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
9 transformation = [[0.5, 0.5], [-0.5, 1.5]]
10 X = np.dot(X, transformation)
```

3.2 Implementing Batch Gradient Descent

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Sigmoid function
5 def sigmoid(z):
6     # Sigmoid function returns the probability prediction
7     return 1 / (1 + np.exp(-z))
8
9 # Loss function (Logistic loss)
10 def compute_loss(y, y_pred):
11     m = len(y) # Number of samples
12     # Logistic loss (binary cross-entropy) formula
13     loss = (-1 / m) * np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 -
14         y_pred))
15     return loss
16
17 # Batch Gradient Descent
18 def batch_gradient_descent(X, y, learning_rate=0.01, iterations=20):
19     # Initialize weights with zeros (X.shape[1] = number of features)
20     weights = np.zeros(X.shape[1])
21     bias = 0 # Initialize bias term
22     m = len(y) # Number of training examples
23
24     # Store loss for each iteration to plot later
25     loss_history = []
26
27     for i in range(iterations):
28         # Linear model
29         linear_model = np.dot(X, weights) + bias
30
31         # Apply sigmoid function to linear model to get predictions
32         y_pred = sigmoid(linear_model)
33
34         # Compute gradients for weights (dw) and bias (db)
35         dw = (1 / m) * np.dot(X.T, (y_pred - y)) # Gradient of weights
36         db = (1 / m) * np.sum(y_pred - y) # Gradient of bias
37
38         # Update weights and bias using the gradients and learning rate
39         weights -= learning_rate * dw
40         bias -= learning_rate * db
41
42         # Compute the loss and store it for plotting
43         loss = compute_loss(y, y_pred)
44         loss_history.append(loss)
45
46         # Print progress (optional)
47         print(f"Iteration {i}: Loss = {loss:.4f}")
48
49     return weights, bias, loss_history
50
51 # Adding bias term to X by inserting a column of ones for the intercept
52 X_bias = np.c_[np.ones(X.shape[0]), X] # Adding bias term (X0 = 1)
53
54 # Run Batch Gradient Descent

```

```

54 weights, bias, loss_history = batch_gradient_descent(X_bias, y,
55     learning_rate=0.01, iterations=20)
56
57 # Plot the loss over iterations
58 plt.plot(range(1, 21), loss_history, label='Batch Gradient Descent')
59 plt.xlabel('Iteration')
60 plt.ylabel('Loss')
61 plt.title('Loss over Iterations for Batch Gradient Descent')
62 plt.legend()
63 plt.show()

```

The weights have been initialized to zeros. This approach has several benefits:

- **Simplicity:** In binary classification tasks in particular, starting with zeros guarantees that the first predictions are not skewed towards any certain class.
- **Convergence:** While initializing to zeros ensures that the model converges without adding needless randomness, alternative initialization strategies (such as random values) may also be effective.

3.3 Utilized loss function

3.3.1 Selected Loss Function

In binary classification issues, where the goal is to minimize the error in predicting the proper class labels, the logistic loss, also known as the cross-entropy loss, is applied.

3.3.2 Reason for Selection

Logistic loss is well-suited for classification problems since it penalizes inaccurate predictions and provides a smooth gradient for optimization using gradient-based approaches.

3.3.3 Loss Function

The logistic loss is defined as follows:

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

Where:

- y_i is the true label for sample i ,
- p_i is the predicted probability for the label being 1.

3.4 Plot the loss with respect to number of iterations

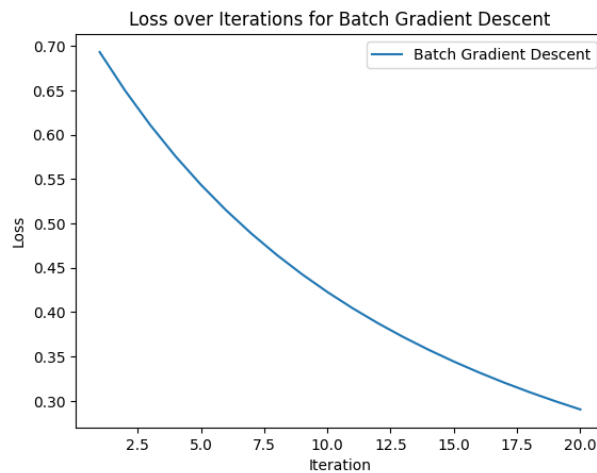


Figure 7: BGD loss plot

3.5 Stochastic Gradient Descent (SGD)

```
1 # Stochastic Gradient Descent
2 def stochastic_gradient_descent(X, y, learning_rate=0.01, iterations=20):
3     # Initialize weights with zeros (X.shape[1] = number of features)
4     weights = np.zeros(X.shape[1])
5     bias = 0 # Initialize bias term
6     m = len(y) # Number of training examples
7
8     # Store loss for each iteration to plot later
9     loss_history = []
10
11     for i in range(iterations):
12         # Shuffle the data to ensure stochasticity in SGD
13         indices = np.random.permutation(m) # Generate shuffled indices
14         X_shuffled = X[indices] # Shuffle features
15         y_shuffled = y[indices] # Shuffle target values
16
17         # Update weights using each training example one at a time
18         for j in range(m):
19             # Select a single training example
20             X_i = X_shuffled[j, :].reshape(1, -1) # Reshape to match
21                                     dimensions
22             y_i = y_shuffled[j] # Target value for the selected example
23
24             # Linear model (dot product of weights and input, plus bias)
25             linear_model = np.dot(X_i, weights) + bias
26
27             # Apply sigmoid function to get the predicted probability
28             y_pred = sigmoid(linear_model)
29
30             # Compute gradients for weights (dw) and bias (db) for the
31                 selected example
32             dw = X_i.T * (y_pred - y_i) # Gradient of weights
```



```

31         db = (y_pred - y_i)                # Gradient of bias
32
33         # Update weights and bias using the computed gradients and
           learning rate
34         weights -= learning_rate * dw.flatten() # Update weights
35         bias -= learning_rate * db             # Update bias
36
37         # After each full pass (iteration) over the dataset, compute the
           loss
38         linear_model_full = np.dot(X, weights) + bias # Linear
           combination for all data
39         y_pred_full = sigmoid(linear_model_full) # Predictions for all
           data
40         loss = compute_loss(y, y_pred_full) # Compute the loss (logistic
           loss)
41         loss_history.append(loss) # Store the loss for later plotting
42
43         # Print progress (optional)
44         print(f"Iteration {i}: Loss = {loss:.4f}")
45
46     return weights, bias, loss_history
47
48 # Run Stochastic Gradient Descent
49 weights_sgd, bias_sgd, loss_history_sgd = stochastic_gradient_descent(
           X_bias, y, learning_rate=0.01, iterations=20)
50
51 # Plot the loss over iterations for Stochastic Gradient Descent
52 plt.plot(range(1, 21), loss_history_sgd, label='Stochastic Gradient
           Descent', color='orange')
53 plt.xlabel('Iteration')
54 plt.ylabel('Loss')
55 plt.title('Loss over Iterations for Stochastic Gradient Descent')
56 plt.legend()
57 plt.show()

```

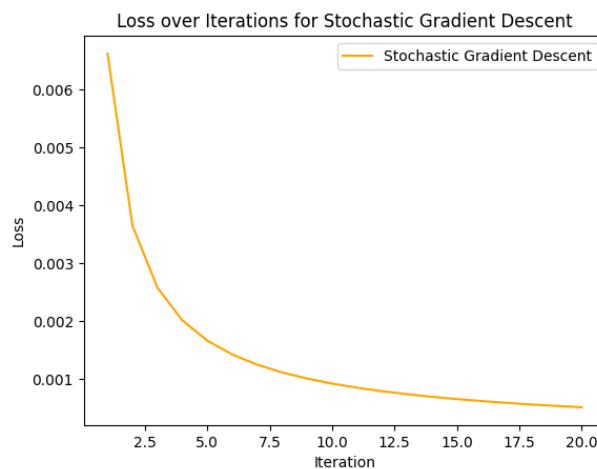


Figure 8: SGD loss plot

3.6 Newton's Method

```
1 # Define the sigmoid function
2 def sigmoid(z):
3     # Sigmoid activation function
4     return 1 / (1 + np.exp(-z))
5
6 # Define the compute_loss function
7 def compute_loss(y, y_pred):
8     # Calculate binary cross-entropy loss (logistic loss)
9     # A small epsilon (1e-15) is added to avoid log(0)
10    return -np.mean(y * np.log(y_pred + 1e-15) + (1 - y) * np.log(1 -
        y_pred + 1e-15))
11
12 # Implement Newton's Method
13 def newtons_method(X, y, iterations=20):
14     # Initialize weights with zeros (X.shape[1] = number of features)
15     weights = np.zeros(X.shape[1])
16     bias = 0 # Initialize bias term
17     m = len(y) # Number of training examples
18
19     # Store loss for each iteration to plot later
20     loss_history = []
21
22     for i in range(iterations):
23         # Linear model: Z = X * weights + bias
24         linear_model = np.dot(X, weights) + bias
25         # Predictions using sigmoid function
26         y_pred = sigmoid(linear_model)
27
28         # Gradient (first derivative of the loss function with respect to
            weights)
29         gradient = np.dot(X.T, (y_pred - y)) / m # Mean of the gradients
30
31         # Hessian matrix (second derivative of the loss function with
            respect to weights)
32         S = np.diag(y_pred * (1 - y_pred)) # Diagonal matrix of the
            second derivatives (variance of predictions)
33         hessian = np.dot(X.T, np.dot(S, X)) / m # Hessian matrix (second
            derivative)
34
35         # Update weights using Newton's method: weights = weights - (
            Hessian^-1 * Gradient)
36         weights -= np.linalg.inv(hessian).dot(gradient)
37
38         # Update bias term (bias update uses simple gradient descent)
39         bias -= np.mean(y_pred - y) # Mean of the errors
40
41         # Calculate the loss (binary cross-entropy)
42         loss = compute_loss(y, y_pred)
43         loss_history.append(loss)
44
45         # Print loss for each iteration
46         print(f"Iteration {i}: Loss = {loss:.4f}")
47
```

```

48     return weights, bias, loss_history
49
50 # Run Newton's Method
51 weights_newton, bias_newton, loss_history_newton = newtons_method(X_bias,
52     y, iterations=20)
53
54 # Plot the loss over iterations
55 plt.plot(range(1, 21), loss_history_newton, label='Newton\'s Method',
56     color='green')
57 plt.xlabel('Iteration')
58 plt.ylabel('Loss')
59 plt.title('Loss over Iterations for Newton\'s Method')
60 plt.legend()
61 plt.show()

```

3.7 Newton's Method Loss w.r.t. number of iterations

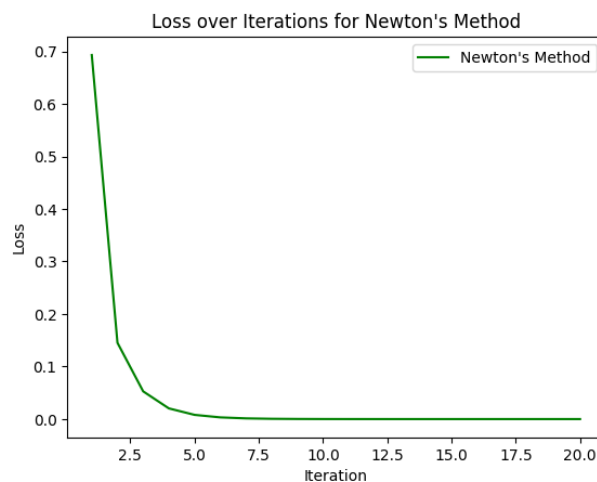


Figure 9: Loss for each Iteration

3.8 Combined Loss Plot

```

1 import matplotlib.pyplot as plt
2
3 # Plotting the losses
4 plt.figure(figsize=(10, 6))
5 plt.plot(range(1, 21), loss_history, label='Batch Gradient Descent', color
6     ='blue', marker='o')
7 plt.plot(range(1, 21), loss_history_sgd, label='Stochastic Gradient
8     Descent', color='orange', marker='x')
9 plt.plot(range(1, 21), loss_history_newton, label='Newton\'s Method',
10     color='green', marker='s')
11 plt.xlabel('Iteration')
12 plt.ylabel('Loss')
13 plt.title('Loss over Iterations for Different Optimization Methods')
14 plt.legend()
15 plt.grid()
16 plt.show()

```

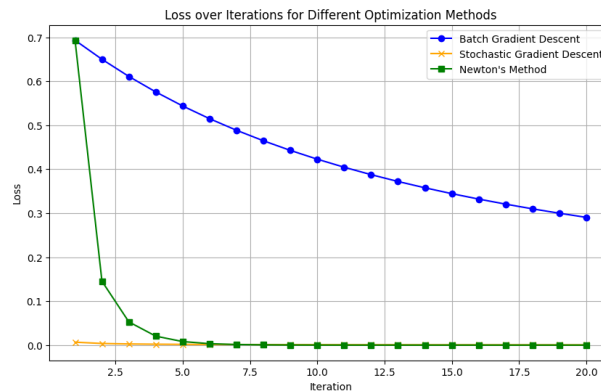


Figure 10: Loss Plots

Although batch gradient descent converges more slowly, it is more stable. Newton's Method has a faster rate of convergence but less stability. The convergence speed and stability are both higher in stochastic gradient descent.

3.9 Approach to Decide Number of Iterations

Gradient Descent: Early Stopping: Based on a predetermined improvement threshold, watch the loss function and terminate the procedure when the loss ceases to decrease noticeably. Reason: By doing this, overfitting is avoided and time-wasting declining returns are avoided by the algorithm.

Newton's Approach: The Hessian threshold and gradient Keep an eye on the characteristics of the Hessian and the norm (magnitude) of the gradient. Stop if the Hessian approaches singularity, which suggests a possible problem with convergence, or if the gradient's norm is less than a specific threshold, suggesting a possible local minimum.

3.10 Changing Centers for Data

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from sklearn.datasets import make_blobs
4
5  # Generate synthetic data
6  np.random.seed(0)
7  centers = [[3, 0], [5, 1.5]]
8  X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
9  transformation = [[0.5, 0.5], [-0.5, 1.5]]
10 X = np.dot(X, transformation)
11
12 # Convert y to binary (0 and 1) for binary classification
13 y_binary = np.where(y == 0, 0, 1) # Assuming two classes in the data
14
15 # Adding bias term to X
16 X_bias = np.c_[np.ones(X.shape[0]), X] # Adding bias term (X0 = 1)
17
18 # Run Batch Gradient Descent
19 weights, bias, loss_history = batch_gradient_descent(X_bias, y_binary,
20               learning_rate=0.01, iterations=20)

```

```

21 # Plot the loss over iterations
22 plt.figure(figsize=(10, 5))
23 plt.plot(range(1, 21), loss_history, label='Batch Gradient Descent')
24 plt.xlabel('Iteration')
25 plt.ylabel('Loss')
26 plt.title('Loss over Iterations for Batch Gradient Descent')
27 plt.legend()
28 plt.show()
29
30 # Scatter plot of the data
31 plt.figure(figsize=(10, 6))
32 plt.scatter(X[:, 0], X[:, 1], c=y_binary, cmap='viridis', alpha=0.5)
33 plt.title('Scatter Plot of Synthetic Data')
34
35 # Plot decision boundary
36 x_values = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
37 y_values = -(weights[1] * x_values + bias) / weights[2]
38 plt.plot(x_values, y_values, color='red', label='Decision Boundary')
39
40 plt.xlabel('X1')
41 plt.ylabel('X2')
42 plt.grid(True)
43 plt.legend()
44 plt.show()

```

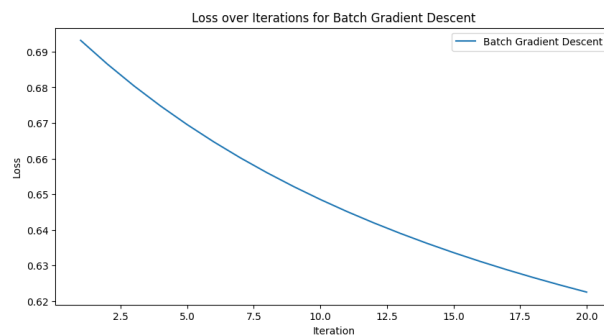


Figure 11: Loss Plots

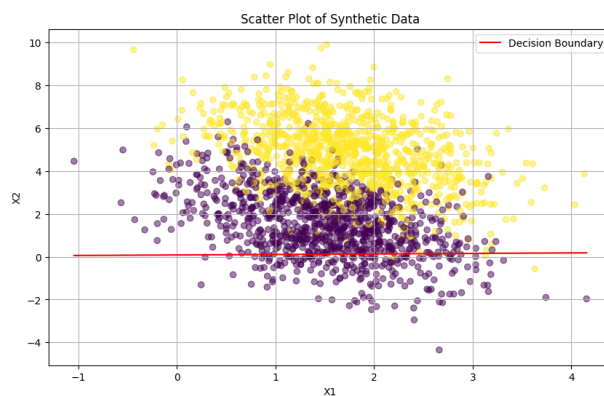


Figure 12: Scatter plot when the center is [3,0] and [5,1.5]

Slower Loss Reduction: As a result of the model's difficulty classifying points close to the border, loss decreases more gradually.

Complex Error Surface: The model must balance misclassifications due to the difficult error surface created by overlapping classes.

More Iterations Needed: Convergence requires more iterations due to its complexity.

In general, optimization takes longer when classes overlap since it might be challenging to define the decision border.