

Department of Electronic & Telecommunication
Engineering
University of Moratuwa

EN3150 Pattern Recognition



Learning from data and related challenges and
classification

Individual Assignment 02

Name: Dilshan N.L
Index No: 210129P

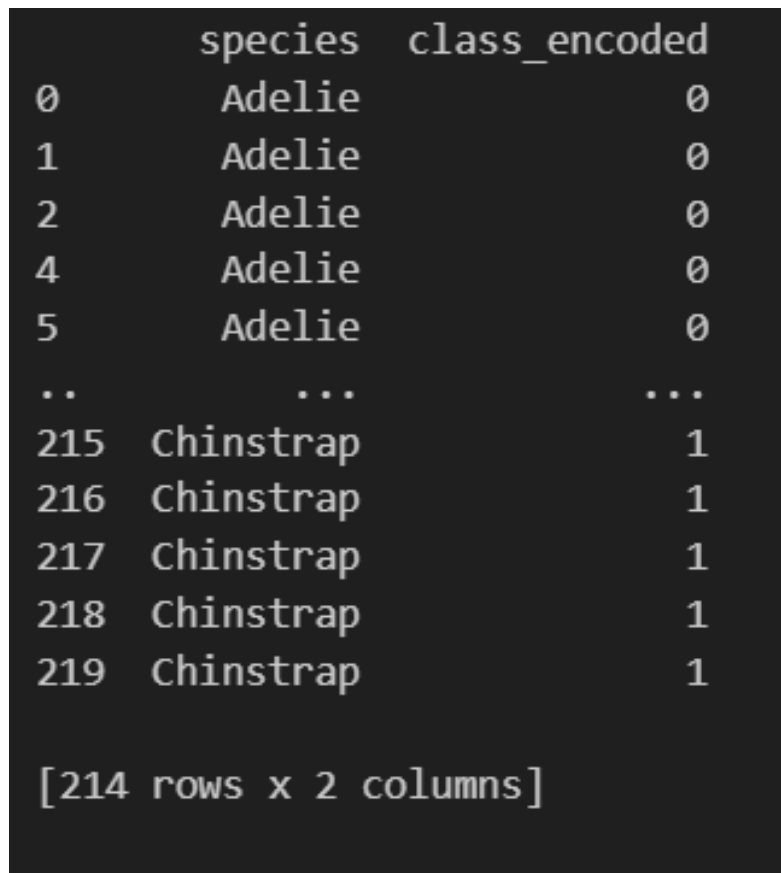
Date - 2024.09.28

Contents

1	Logistic Regression	2
1.1	Loading data	2
1.2	Purpose of <code>y_encoded = le.fit_transform(df_filtered['species'])</code>	2
1.3	Purpose of <code>X = df.drop(['species', 'island', 'sex'], axis=1)</code>	2
1.4	Why can't we use 'island' and 'sex' features?	2
1.5	Training the Logistic Regression Model	3
1.6	Usage of <code>random_state=42</code>	3
1.7	Why is the Accuracy Low? why does the <code>saga</code> solver perform poorly?	3
1.7.1	Why is accuracy low?	3
1.7.2	Why does the <code>saga</code> solver perform poorly?	4
1.8	Accuracy with <code>liblinear</code> Solver	4
1.9	Why does the <code>liblinear</code> solver perform better than <code>saga</code> solver ?	4
1.10	Feature Scaling Comparison	4
1.11	Run the code given in listing 3. What is the problem of this code and how to solve this?	6
1.12	Incorrect Use of Scaling on Encoded Categorical Features	7
2	Logistic Regression on Real-World Data	10
2.1	Choose a data set from UCI Machine Learning Repository that is appropriate for logistic regression.	10
2.2	Correlation Matrix and Pair Plot	10
2.3	Logistic Regression Model and Evaluation	12
2.4	P-values and Feature Selection	13
3	Gradient Descent Methods	15
3.1	Data Generation	15
3.2	Batch Gradient Descent	15
3.3	Loss Function	17
3.4	Loss Plot (Batch Gradient Descent)	17
3.5	Stochastic Gradient Descent (SGD)	17
3.6	Newton's Method	20
3.7	Loss Plot for Newton's Method	21
3.8	Combined Loss Plot	22
3.9	Approaches to Decide Number of Iterations	23
3.10	Changing Centers for Data	23

1 Logistic Regression

1.1 Loading data



	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: Loaded Data

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

The purpose of `y_encoded = le.fit_transform(df_filtered['species'])` is to convert the categorical target variable 'species' into a numerical format (e.g., 'Adelie' = 0, 'Chinstrap' = 1) for use in the logistic regression model. Logistic regression requires numerical inputs, so we encode each species as an integer.

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

This line drops the columns `species`, `island`, and `sex` from the dataset, which are not suitable for inclusion as features in the logistic regression model.

- **species:** This is the original species label, which we replaced with the encoded version.
- **island** and **sex:** These are categorical variables that have not been encoded or converted into numerical form. Therefore, they are dropped. Logistic regression models require numerical input, so categorical features must be encoded if they are to be used.
- **class_encoded:** This is the target variable and is thus excluded from the feature set.

1.4 Why can't we use 'island' and 'sex' features?

The `island` and `sex` features are categorical variables that don't have a clear numeric representation. Additionally, including them without proper encoding (such as one-hot encoding) would lead to inaccurate results, as logistic regression operates on numerical data. Although we properly encode them

this transformation can add complexity and increase sparsity within the dataset. By omitting these variables, we can reduce the risk of overfitting, streamline the model, and potentially improve its ability to generalize and accuracy.

1.5 Training the Logistic Regression Model

To train the logistic regression model, the following code was used, and the accuracy is less than 60%.

```
1 logreg = LogisticRegression(solver='saga')
2 logreg.fit(X_train, y_train)
```

```
# 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 saga solver to learn weights
logreg = LogisticRegression(solver='saga')
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_)
✓ 0.0s Python
```

```
Accuracy : 0.5813953488372093
[[ 2.75862851e-03 -8.28939890e-05  4.55637098e-04 -2.86188999e-04]] [-8.51183128e-06]
c:\Users\HP\AppData\Local\Programs\Python\Python311\lib\site-packages\sklearn\linear_model\sag.py:350: ConvergenceWarning: The max_iter was reac
warnings.warn(
```

Figure 2: Logistic Regression Model Training

1.6 Usage of random_state=42

Setting `random_state=42` ensures the reproducibility of the train-test split. It controls the random number generator for the splitting process so that the same training and testing sets are generated every time the code is run.

- When splitting the data into training and testing sets, the data is split randomly. By setting `random_state=42`, we're ensuring that the data split is reproducible each time we run the code.
- If we don't specify a random state, the training and testing sets will be different on each run, making our results inconsistent.
- 42 is just an arbitrary number. We could use any value (e.g., 42, 0, 1, etc.), and as long as we keep using the same number, our data split will remain the same across different runs.

1.7 Why is the Accuracy Low? why does the saga solver perform poorly?

1.7.1 Why is accuracy low?

Since the target classes (Adelie and Chinstrap penguin species) are imbalanced (one class significantly outnumbers the other), the model might struggle to predict the minority class, leading to poor performance.

```
> <
# Count the occurrences of each class in the target variable
class_counts = y.value_counts()

# Display the counts
print(class_counts)
```

```
[5] ✓ 0.0s
```

```
... class_encoded
0    146
1     68
Name: count, dtype: int64
```

Figure 3: Imbalanced Dataset

Since there is not enough data for training, the model may fail to learn the necessary patterns. This could lead to underfitting, where the model performs poorly on both training and testing data. Also

the logistic regression assumes a linear relationship between the features and the log odds of the target variable. If the relationship between the features and the target is not linear, the model may not perform well.

1.7.2 Why does the saga solver perform poorly?

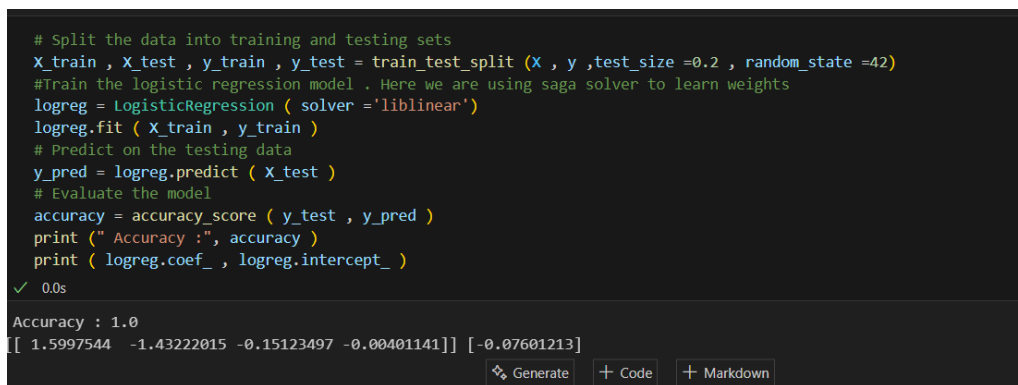
The choice of the `saga` solver in logistic regression might also explain the model's poor performance in this scenario. The `saga` solver is specifically designed for handling large datasets and works efficiently with both L1 and L2 regularization. However, with a smaller dataset like the penguins dataset, 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 Accuracy with liblinear Solver

After changing the solver to `liblinear`, the classification accuracy improved:

```
1 logreg = LogisticRegression(solver='liblinear')
2 logreg.fit(X_train, y_train)
3 accuracy = accuracy_score(y_test, y_pred)
```

The accuracy with `liblinear` solver was found to be higher(almost 100%) because this solver is more suited for smaller datasets and works better with binary classification problems.



```
# 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 saga solver 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_)
✓ 0.0s

Accuracy : 1.0
[[ 1.5997544 -1.43222015 -0.15123497 -0.00401141]] [-0.07601213]
```

Figure 4: 'liblinear' Solver

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

Using the `liblinear` solver improved the accuracy because it is designed for small datasets and works well for binary classification problems. It uses a simpler optimization technique that is efficient with fewer data points and helps avoid overfitting. Since the penguins dataset is small, `liblinear` can learn the model parameters more effectively, leading to almost perfect accuracy.

1.10 Feature Scaling Comparison

```
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, random_state=42)
8
9 # Apply Standard Scaler to the features
10 scaler = StandardScaler()
11 X_train_scaled = scaler.fit_transform(X_train)
12 X_test_scaled = scaler.transform(X_test)
```

```

13
14 # Train and evaluate logistic regression using liblinear without scaling
15 logreg_liblinear = LogisticRegression(solver='liblinear')
16 logreg_liblinear.fit(X_train, y_train)
17 y_pred_liblinear = logreg_liblinear.predict(X_test)
18 accuracy_liblinear = accuracy_score(y_test, y_pred_liblinear)
19
20 # Train and evaluate logistic regression using saga without scaling
21 logreg_saga = LogisticRegression(solver='saga')
22 logreg_saga.fit(X_train, y_train)
23 y_pred_saga = logreg_saga.predict(X_test)
24 accuracy_saga = accuracy_score(y_test, y_pred_saga)
25
26 # Train and evaluate logistic regression using liblinear with scaling
27 logreg_liblinear_scaled = LogisticRegression(solver='liblinear')
28 logreg_liblinear_scaled.fit(X_train_scaled, y_train)
29 y_pred_liblinear_scaled = logreg_liblinear_scaled.predict(X_test_scaled)
30 accuracy_liblinear_scaled = accuracy_score(y_test, y_pred_liblinear_scaled)
31
32 # Train and evaluate logistic regression using saga with scaling
33 logreg_saga_scaled = LogisticRegression(solver='saga')
34 logreg_saga_scaled.fit(X_train_scaled, y_train)
35 y_pred_saga_scaled = logreg_saga_scaled.predict(X_test_scaled)
36 accuracy_saga_scaled = accuracy_score(y_test, y_pred_saga_scaled)
37
38 # Print out the accuracy values for comparison
39 print(f"Accuracy (liblinear without scaling): {accuracy_liblinear}")
40 print(f"Accuracy (saga without scaling): {accuracy_saga}")
41 print(f"Accuracy (liblinear with scaling): {accuracy_liblinear_scaled}")
42 print(f"Accuracy (saga with scaling): {accuracy_saga_scaled}")

```

Table 1: Model Accuracy for Different Configurations

Configuration	Accuracy
Liblinear (without scaling)	1.00
SAGA (without scaling)	0.5813953488372093
Liblinear (with scaling)	0.9767441860465116
SAGA (with scaling)	0.0.9767441860465116

Without Feature Scaling

- **liblinear** performs well on smaller datasets but it is affected by differences in feature scales.
- **saga** performs poorly without scaling since it struggles with features on different scales.

With Feature Scaling

- Scaling helps solvers like **saga** perform better by ensuring all features have similar ranges (In standard scaling all the features have similar mean=0 and similar standard deviation=1).
- **liblinear** sees a smaller reduction of accuracy with scaling but can still benefit.

Why Scaling Matters

- **liblinear**: Works decently without scaling but may improve or deteriorate slightly with it.
- **saga**: Relies on gradient methods, so scaling helps it converge faster and perform better.

```

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 5: 'liblinear' Solver

1.11 Run the code given in listing 3. What is the problem of this code and how to solve this?

While running the code provided in Listing 3, a `ValueError` was encountered during the model fitting step. The error message, "could not convert string to float," indicated that categorical variables such as `island` and `sex` were still in string format, and the `LogisticRegression` model in `sklearn` expects numerical inputs. Specifically, the island name 'Dream' could not be converted to a float.

To resolve this, we need to convert all categorical columns into numerical format. This was done using the `LabelEncoder` from the `sklearn.preprocessing` library. 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
9 df = sns.load_dataset("penguins")
10 df.dropna(inplace=True)
11
12 # Filter rows for 'Adelie' and 'Chinstrap' classes
13 selected_classes = ['Adelie', 'Chinstrap']
14 df_filtered = df[df['species'].isin(selected_classes)].copy() # Make a copy to avoid
15 # the warning
16
17 # Initialize the LabelEncoder
18 le = LabelEncoder()
19
20 # Encode the species column
21 df_filtered['class_encoded'] = le.fit_transform(df_filtered['species'])
22
23 # Encode categorical columns ('island', 'sex')
24 df_filtered['island_encoded'] = le.fit_transform(df_filtered['island'])
25 df_filtered['sex_encoded'] = le.fit_transform(df_filtered['sex'])
26
27 # Drop original categorical columns and unnecessary ones
28 X = df_filtered.drop(['species', 'class_encoded', 'island', 'sex'], axis=1)
29 y = df_filtered['class_encoded'] # Target variable
30
31 # Split the data into training and testing sets
32 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
33 =42)
34
35 # Standardize the features (optional but recommended)
36 scaler = StandardScaler()
37 X_train = scaler.fit_transform(X_train)
38 X_test = scaler.transform(X_test)
39
40 # Train the logistic regression model using saga solver
41 logreg = LogisticRegression(solver='saga')
42 logreg.fit(X_train, y_train)
43
44 # Predict on the testing data
45 y_pred = logreg.predict(X_test)
46
47 # Evaluate the model
48 accuracy = accuracy_score(y_test, y_pred)
49 print("Accuracy:", accuracy)

```

```
48 print(logreg.coef_, logreg.intercept_)
```

1.12 Incorrect Use of Scaling on Encoded Categorical Features

When dealing with categorical features such as 'red', 'blue', and 'green', applying label encoding followed by feature scaling methods like Standard Scaling or Min-Max Scaling is not appropriate. Label encoding assigns arbitrary numerical values (e.g., 'red' = 0, 'blue' = 1, 'green' = 2), but these values do not have a meaningful ordinal relationship. Scaling these numbers could introduce false assumptions about their relationships.

Proposed Solution: One-Hot Encoding

The correct approach is to use one-hot encoding, which converts each category into a separate binary feature. This ensures that no ordinal relationships are inferred between the categories. After one-hot encoding, feature scaling should only be applied to continuous numerical variables, not categorical ones. For example, the one-hot encoding of the feature would be:

red	blue	green
1	0	0
0	1	0
0	0	1
0	1	0
0	0	1

One-hot encoding ensures that the model treats each color independently, without implying any order or relationship.

Question 2

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.

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

$$z = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2$$

$$z = -5.9 + (0.06 \cdot 50) + (1.5 \cdot 3.6)$$

$$z = -5.9 + 3 + 5.4 = 2.5$$

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

So, the 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:

$$0.6 = \frac{1}{1 + e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)}}$$

$$1 + e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)} = \frac{1}{0.6}$$

$$e^{-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)} = \frac{1}{0.6} - 1 = \frac{0.4}{0.6} = \frac{2}{3}$$

$$-(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2) = \ln\left(\frac{2}{3}\right)$$

Thus:

$$w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = -\ln\left(\frac{2}{3}\right)$$

$$-5.9 + 0.06 \cdot x_1 + 1.5 \cdot 3.6 = -\ln\left(\frac{2}{3}\right)$$

$$-5.9 + 0.06 \cdot x_1 + 5.4 = -\ln\left(\frac{2}{3}\right)$$

$$0.06 \cdot x_1 - 0.5 = -\ln\left(\frac{2}{3}\right)$$

$$0.06 \cdot x_1 = 0.405 + 0.5 = 0.905$$

$$x_1 = \frac{0.905}{0.06} \approx 15.083$$

Therefore, a student needs to study approximately 15.1 hours to achieve a 60% chance of receiving an A+.

2 Logistic Regression on Real-World Data

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

Wine Quality Dataset

The Wine Quality dataset is available at the UCI Machine Learning Repository. It consists of features based on chemical properties, and the target variable is the quality of wine.

Features include:

- Fixed acidity
- Volatile acidity
- Citric acid
- Residual sugar
- Chlorides
- Free sulfur dioxide
- Total sulfur dioxide
- Density
- pH
- Sulphates
- Alcohol

Target variable: Wine quality (scaled between 0 and 10). For logistic regression, we will convert this into a binary classification (e.g., quality ≥ 6 : good wine, quality < 6 : not good wine).

```

1 import pandas as pd
2
3 # Load the dataset
4 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
      winequality-red.csv"
5 data = pd.read_csv(url, delimiter=';')
6
7 # Create binary target variable (1 for quality >= 6, 0 for quality < 6)
8 data['quality_binary'] = (data['quality'] >= 6).astype(int)

```

2.2 Correlation Matrix and Pair Plot

We first analyze the correlation between the selected features: *fixed acidity*, *volatile acidity*, *citric acid*, *residual sugar*, and *alcohol*. The correlation matrix shows the relationships between these variables. Features with a high correlation may provide redundant information, while low correlations suggest that the features are more independent.

```

1 import seaborn as sns
2 import matplotlib.pyplot as plt
3
4 # Select up to 5 features for the analysis
5 selected_features = ['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
6                     'alcohol']
7
8 # Correlation matrix
9 correlation_matrix = data[selected_features].corr()
10 print(correlation_matrix)
11
12 # Pairplot using seaborn
13 sns.pairplot(data[selected_features + ['quality_binary']], hue='quality_binary')

```

	fixed acidity	volatile acidity	citric acid \
fixed acidity	1.000000	-0.256131	0.671703
volatile acidity	-0.256131	1.000000	-0.552496
citric acid	0.671703	-0.552496	1.000000
residual sugar	0.114777	0.001918	0.143577
alcohol	-0.061668	-0.202288	0.109903

	residual sugar	alcohol
fixed acidity	0.114777	-0.061668
volatile acidity	0.001918	-0.202288
citric acid	0.143577	0.109903
residual sugar	1.000000	0.042075
alcohol	0.042075	1.000000

Figure 6: Correlation Matrix

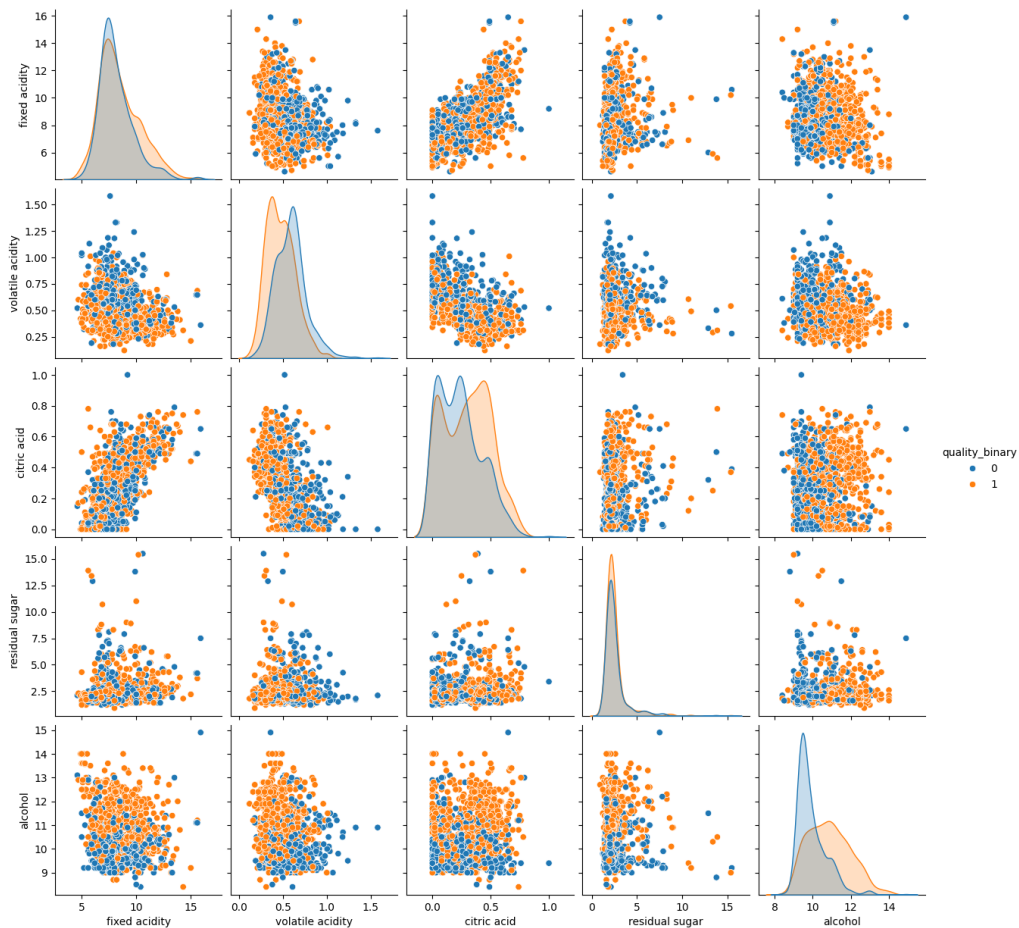


Figure 7: Pair Plots

Comments on Results

Most of the correlation values are less than 0.5, indicating that most features are relatively independent. However, the correlation between fixed acidity and citric acid, as well as between volatile acidity and citric acid, is above 0.5, suggesting some degree of dependency. Nevertheless, these correlation values are not particularly high. 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 # Features and target variable
7 X = data[selected_features]
8 y = data['quality_binary']
9
10 # Split the dataset into training and testing sets
11 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state
12     =42)
13
14 # Standardize the features
15 scaler = StandardScaler()
16 X_train_scaled = scaler.fit_transform(X_train)
17 X_test_scaled = scaler.transform(X_test)
18
19 # Logistic regression model
20 model = LogisticRegression()
21 model.fit(X_train_scaled, y_train)
22
23 # Predictions
24 y_pred = model.predict(X_test_scaled)
25
26 # Evaluate the model
27 accuracy = accuracy_score(y_test, y_pred)
28 print(f'Accuracy: {accuracy}')
29 print(classification_report(y_test, y_pred))

```

Accuracy: 0.7208333333333333					
	precision	recall	f1-score	support	
0	0.67	0.75	0.70	213	
1	0.78	0.70	0.74	267	
accuracy			0.72	480	
macro avg	0.72	0.72	0.72	480	
weighted avg	0.73	0.72	0.72	480	

Figure 8: Pair Plots

Model Evaluation

- **Accuracy:** The model achieves an accuracy of approximately **72%**, meaning that it correctly predicts the binary wine quality in 72% of the cases. While this is a reasonable result, it indicates that there's room for improvement, possibly through more advanced techniques or feature engineering.
- **Precision and Recall:** The precision for class 0 (not good wine) is 0.67, while for class 1 (good wine), it is 0.78. This suggests that the model is slightly better at correctly identifying "good" wines than "not good" wines. However, the recall is higher for class 0 (0.75) compared to class 1 (0.70), indicating that the model tends to predict "not good" wines more accurately, but with some trade-off in precision for class 0.
- **F1-Score:** The F1-scores (a balance between precision and recall) are 0.70 for class 0 and 0.74 for class 1. This shows a relatively balanced performance across both classes but with a slight emphasis on correctly predicting "good" wines.

2.4 P-values and Feature Selection

```

1 import statsmodels.api as sm
2
3 # Add a constant to the model
4 X_train_const = sm.add_constant(X_train_scaled)
5
6 # Logistic regression using statsmodels
7 logit_model = sm.Logit(y_train, X_train_const)
8 result = logit_model.fit()
9
10 # Summary of the model
11 print(result.summary())

```

```

Optimization terminated successfully.
      Current function value: 0.532404
      Iterations 6

Logit Regression Results
=====
Dep. Variable:      quality_binary    No. Observations:      1119
Model:              Logit            Df Residuals:          1113
Method:             MLE              Df Model:              5
Date:               Sun, 29 Sep 2024  Pseudo R-squ.:         0.2305
Time:               17:02:42          Log-Likelihood:        -595.76
converged:          True              LL-Null:              -774.18
Covariance Type:    nonrobust         LLR p-value:          5.897e-75
=====

```

	coef	std err	z	P> z	[0.025	0.975]
const	0.2097	0.073	2.873	0.004	0.067	0.353
x1	0.2807	0.099	2.848	0.004	0.088	0.474
x2	-0.7813	0.100	-7.823	0.000	-0.977	-0.586
x3	-0.3953	0.120	-3.301	0.001	-0.630	-0.161
x4	-0.0839	0.078	-1.080	0.280	-0.236	0.068
x5	1.1972	0.093	12.927	0.000	1.016	1.379

```

=====

```

Figure 9: Pair Plots

Comment on Results

- **Intercept (const):** The intercept (const) has a p-value of **0.004**, which is statistically significant (p less than 0.05). This suggests that the baseline level of the model, when all features are at zero, has a meaningful impact on the prediction.
- **Feature x1 (fixed acidity):**
 - **Coefficient:** 0.2807
 - **p-value:** **0.004** (significant)
 - This feature has a positive relationship with the wine being classified as "good." A higher value of this feature increases the likelihood of predicting "good" wine.
- **Feature x2 (volatile acidity):**
 - **Coefficient:** -0.7813
 - **p-value:** **0.000** (highly significant)
 - This feature has a strong negative effect on the prediction of wine quality. Higher values decrease the likelihood of the wine being classified as "good."
- **Feature x3 (citric acid):**

- **Coefficient:** -0.3953
- **p-value:** **0.001** (significant)
- Like x2, this feature has a negative impact, but its effect is less pronounced compared to x2. This suggests that higher levels of this feature also reduce the probability of "good" wine classification.
- **Feature x4** (residual sugar):
 - **Coefficient:** -0.0839
 - **p-value:** **0.280** (not significant)
 - This feature has a negligible impact on wine quality prediction. Given its high p-value, it is not statistically significant and could potentially be discarded from the model.
- **Feature x5** (alcohol):
 - **Coefficient:** 1.1972
 - **p-value:** **0.000** (highly significant)
 - This is the most impactful feature with a large positive coefficient, indicating that an increase in this feature strongly increases the probability of predicting "good" wine.

Feature that can be discarded: Residual sugar.

3 Gradient Descent Methods

3.1 Data Generation

```

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 Batch Gradient Descent

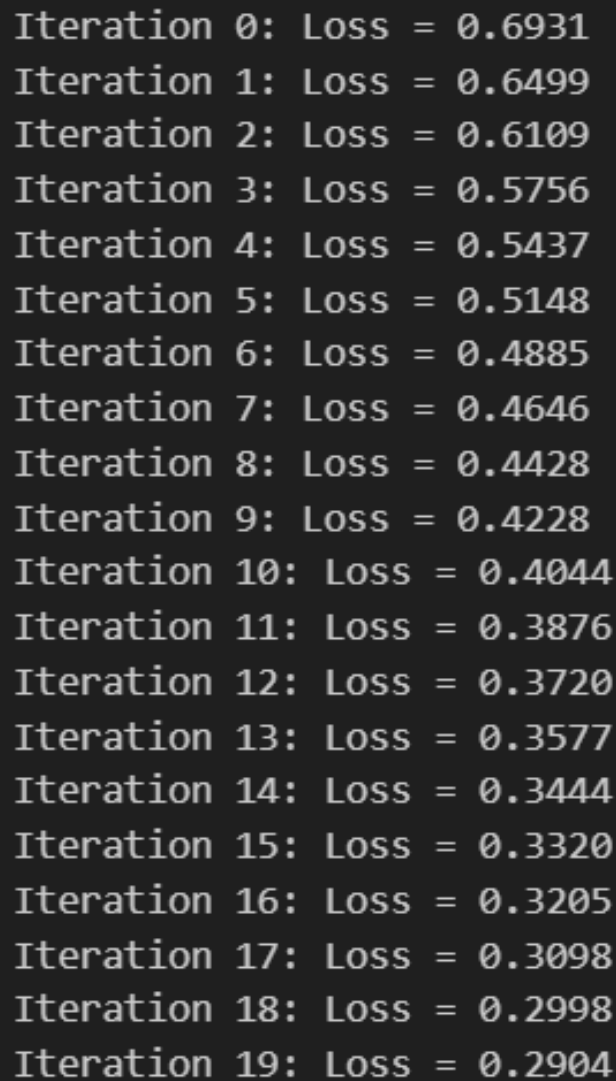
```

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

```



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



Iteration	Loss
0	0.6931
1	0.6499
2	0.6109
3	0.5756
4	0.5437
5	0.5148
6	0.4885
7	0.4646
8	0.4428
9	0.4228
10	0.4044
11	0.3876
12	0.3720
13	0.3577
14	0.3444
15	0.3320
16	0.3205
17	0.3098
18	0.2998
19	0.2904

Figure 10: Loss Values for each Iteration

In our code, the weights has been initialized to zeros. This approach has several benefits:

- **Simplicity:** Starting with zeros ensures that the initial predictions are not biased towards any particular class, especially in binary classification tasks.
- **Convergence:** While other initialization methods (like random values) could also work, initializing to zeros helps ensure that the model converges without adding unnecessary randomness.

3.3 Loss Function

Selected Loss Function

The logistic loss (also called cross-entropy loss) is used for binary classification problems, where we aim to minimize the error in predicting the correct class labels.

Reason for Selection

Logistic loss is well-suited for classification tasks because it penalizes incorrect predictions and provides a smooth gradient for optimization using gradient-based methods.

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 Loss Plot (Batch Gradient Descent)

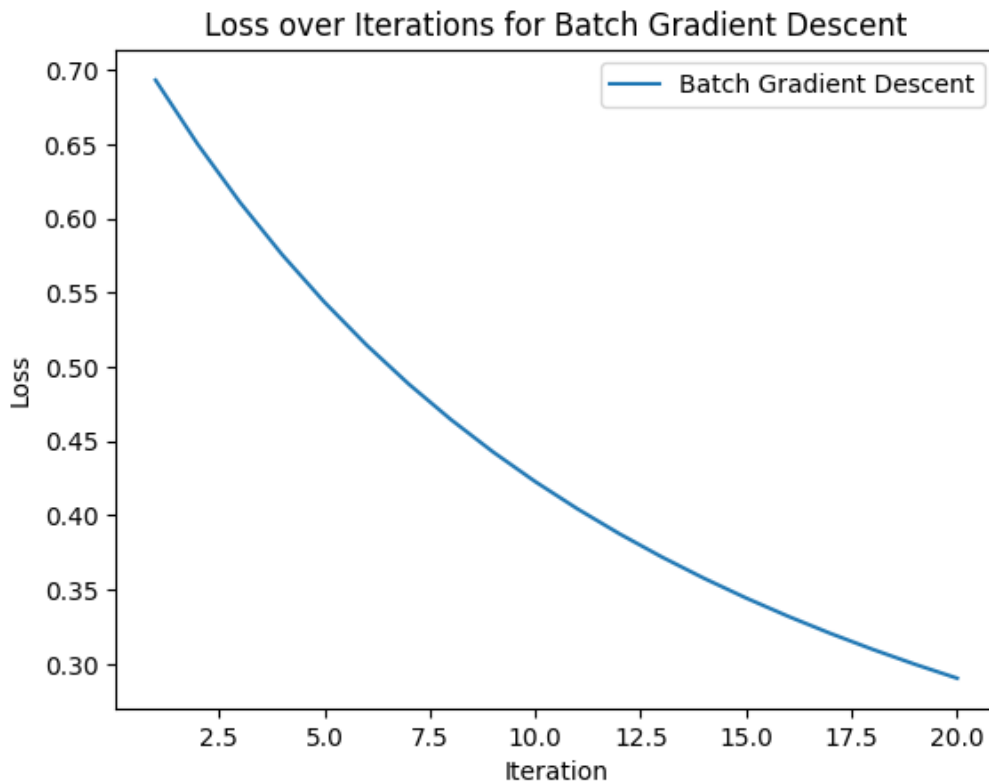


Figure 11: Loss Plot (Batch Gradient Descent)

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
4     weights = np.zeros(X.shape[1])
5     bias = 0
6     m = len(y)
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)
14         X_shuffled = X[indices]
15         y_shuffled = y[indices]
16
17         for j in range(m):
18             # Select a single training example
19             X_i = X_shuffled[j, :].reshape(1, -1)
20             y_i = y_shuffled[j]
21
22             # Linear model
23             linear_model = np.dot(X_i, weights) + bias
24
25             # Apply sigmoid to get prediction
26             y_pred = sigmoid(linear_model)
27
28             # Compute gradients for this single example
29             dw = X_i.T * (y_pred - y_i) # Gradient of weights
30             db = (y_pred - y_i)          # Gradient of bias
31
32             # Update weights and bias
33             weights -= learning_rate * dw.flatten()
34             bias -= learning_rate * db
35
36             # After each full pass over the dataset (one iteration), compute the loss
37             linear_model_full = np.dot(X, weights) + bias
38             y_pred_full = sigmoid(linear_model_full)
39             loss = compute_loss(y, y_pred_full)
40             loss_history.append(loss)
41
42             # Print progress
43             if i % 5 == 0:
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,
50                               learning_rate=0.01, iterations=20)
51
52 # Plot the loss over iterations
53 plt.plot(range(1, 21), loss_history_sgd, label='Stochastic Gradient Descent', color='
54         orange')
55 plt.xlabel('Iteration')
56 plt.ylabel('Loss')
57 plt.title('Loss over Iterations for Stochastic Gradient Descent')
58 plt.legend()
59 plt.show()

```

```
Iteration 0: Loss = 0.0067
Iteration 1: Loss = 0.0037
Iteration 2: Loss = 0.0026
Iteration 3: Loss = 0.0020
Iteration 4: Loss = 0.0017
Iteration 5: Loss = 0.0014
Iteration 6: Loss = 0.0013
Iteration 7: Loss = 0.0011
Iteration 8: Loss = 0.0010
Iteration 9: Loss = 0.0009
Iteration 10: Loss = 0.0009
Iteration 11: Loss = 0.0008
Iteration 12: Loss = 0.0007
Iteration 13: Loss = 0.0007
Iteration 14: Loss = 0.0007
Iteration 15: Loss = 0.0006
Iteration 16: Loss = 0.0006
Iteration 17: Loss = 0.0006
Iteration 18: Loss = 0.0005
Iteration 19: Loss = 0.0005
```

Figure 12: Loss values for each Iteration (SGD)

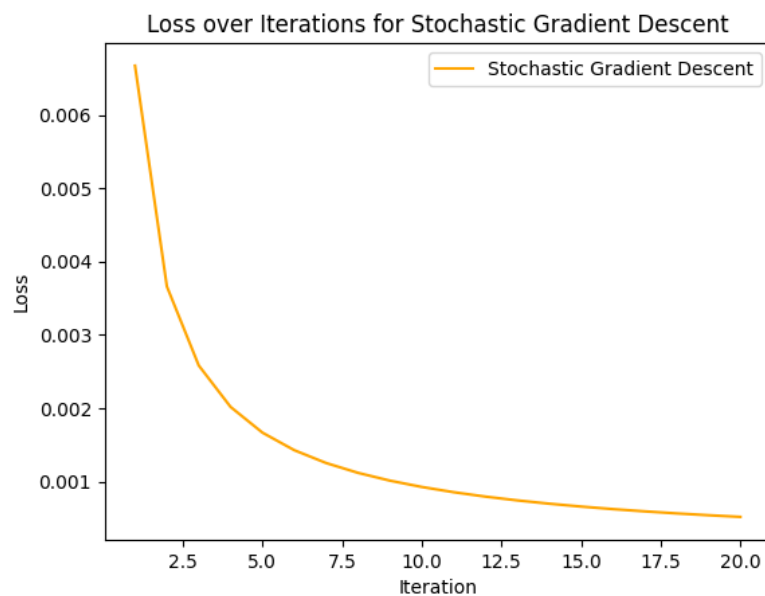


Figure 13: Loss Plot (SGD)

Difference from Batch Gradient Descent: The weight updates are noisier in SGD, but it can reach a solution faster in terms of the number of updates.

3.6 Newton's Method

The update rule for Newton's Method is given by:

$$w_{t+1} = w_t - H^{-1} \nabla L(w_t)$$

Where:

- H is the Hessian matrix,
- $\nabla L(w_t)$ is the gradient of the loss function.

```

1 # Define the sigmoid function
2 def sigmoid(z):
3     return 1 / (1 + np.exp(-z))
4
5 # Define the compute_loss function
6 def compute_loss(y, y_pred):
7     # Using binary cross-entropy loss
8     return -np.mean(y * np.log(y_pred + 1e-15) + (1 - y) * np.log(1 - y_pred + 1e-15))
9
10 # Implement Newton's Method
11 def newtons_method(X, y, iterations=20):
12     # Initialize weights with zeros
13     weights = np.zeros(X.shape[1])
14     bias = 0
15     m = len(y)
16
17     # Store loss for each iteration to plot later
18     loss_history = []
19
20     for i in range(iterations):
21         # Linear model
22         linear_model = np.dot(X, weights) + bias
23         y_pred = sigmoid(linear_model)
24
25         # Gradient
26         gradient = np.dot(X.T, (y_pred - y)) / m
27         # Compute Hessian matrix
28         S = np.diag(y_pred * (1 - y_pred)) # Diagonal matrix
29         hessian = np.dot(X.T, np.dot(S, X)) / m
30
31         # Update weights and bias using Newton's method
32         weights -= np.linalg.inv(hessian).dot(gradient)
33         # We will update bias separately
34         bias -= np.mean(y_pred - y)
35
36         # Calculate the loss
37         loss = compute_loss(y, y_pred)
38         loss_history.append(loss)
39
40         # Print progress
41         if i % 5 == 0:
42             print(f"Iteration {i}: Loss = {loss:.4f}")
43
44     return weights, bias, loss_history
45
46 # Run Newton's Method
47 weights_newton, bias_newton, loss_history_newton = newtons_method(X_bias, y, iterations
48                             =20)
49
50 # Plot the loss over iterations
51 plt.plot(range(1, 21), loss_history_newton, label='Newton\'s Method', color='green')
52 plt.xlabel('Iteration')
53 plt.ylabel('Loss')
54 plt.title('Loss over Iterations for Newton\'s Method')
55 plt.legend()
56 plt.show()

```

```
Iteration 0: Loss = 0.6931
Iteration 1: Loss = 0.1452
Iteration 2: Loss = 0.0528
Iteration 3: Loss = 0.0203
Iteration 4: Loss = 0.0080
Iteration 5: Loss = 0.0032
Iteration 6: Loss = 0.0013
Iteration 7: Loss = 0.0005
Iteration 8: Loss = 0.0002
Iteration 9: Loss = 0.0001
Iteration 10: Loss = 0.0000
Iteration 11: Loss = 0.0000
Iteration 12: Loss = 0.0000
Iteration 13: Loss = 0.0000
Iteration 14: Loss = 0.0000
Iteration 15: Loss = 0.0000
Iteration 16: Loss = 0.0000
Iteration 17: Loss = 0.0000
Iteration 18: Loss = 0.0000
Iteration 19: Loss = 0.0000
```

Figure 14: Loss for each Iteration

3.7 Loss Plot for Newton's Method

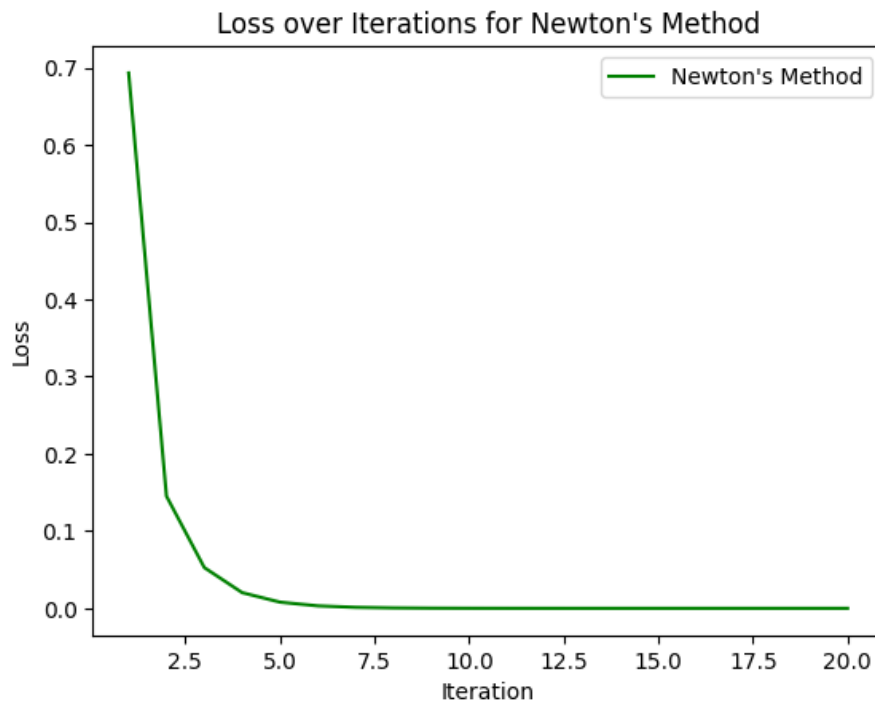


Figure 15: Loss for each Iteration

3.8 Combined Loss Plot

```

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

```

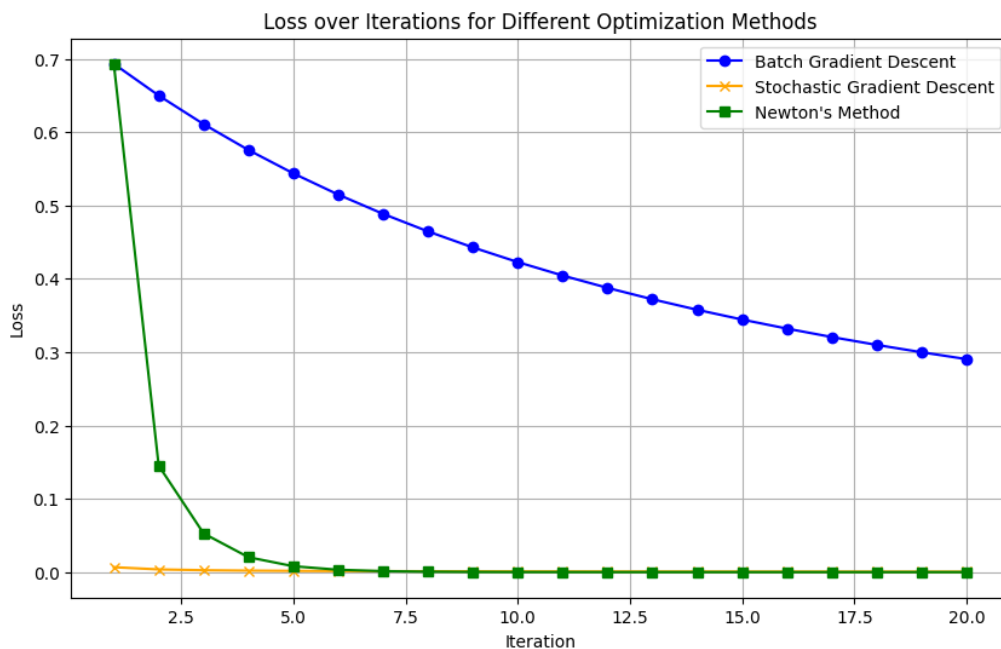


Figure 16: Loss Plots

Convergence Behavior

Batch Gradient Descent

This method shows a gradual decrease in loss values, indicating stable convergence over iterations. However, it takes longer to reach a minimum compared to other two.

Stochastic Gradient Descent

This method shows more fluctuation in loss values, as updates are made based on individual samples. While it can converge faster in terms of iterations, the path to convergence is less stable.

Newton's Method

This method shows rapid convergence due to its utilization of the second derivative (Hessian matrix) for updates. It reaches lower loss values in fewer iterations compared to the other two.

Comparison of Effectiveness

Batch Gradient Descent has higher stability but slower convergence speed. Newton's Method has lower stability but higher convergence speed. Stochastic Gradient Descent exhibits both higher stability and higher convergence speed.

3.9 Approaches to Decide Number of Iterations

Gradient Descent: Early Stopping: Monitor the loss function and stop the algorithm when the loss stops decreasing significantly (based on a predefined threshold for improvement). Reason: This prevents overfitting and ensures that the algorithm doesn't waste time on diminishing returns.

Newton's Method: Gradient and Hessian Threshold: Monitor the gradient's norm (magnitude) and the Hessian's properties. Stop when the gradient's norm is less than a certain threshold (indicating close to a local minimum) or if the Hessian becomes close to singular (indicating a potential issue with convergence).

3.10 Changing Centers for Data

```

1 # Generate synthetic data
2 np . random . seed (0)
3 centers = [[ 3 , 0] , [5 , 1.5]]
4 X , y = make_blobs ( n_samples =2000 , centers = centers , random_state =5)
5 transformation = [[0.5 , 0.5] , [ -0.5 , 1.5]]
6 X = np . dot (X , transformation )
7
8 # Adding bias term to X by inserting a column of ones for intercept
9 X_bias = np.c_[np.ones(X.shape[0]), X] # Adding bias term (X0 = 1)
10
11 # Run Batch Gradient Descent
12 weights, bias, loss_history = batch_gradient_descent(X_bias, y, learning_rate=0.01,
13     iterations=20)
14
15 # Plot the loss over iterations
16 plt.plot(range(1, 21), loss_history, label='Batch Gradient Descent')
17 plt.xlabel('Iteration')
18 plt.ylabel('Loss')
19 plt.title('Loss over Iterations for Batch Gradient Descent')
20 plt.legend()
21 plt.show()

```

```

Iteration 0: Loss = 0.6931
Iteration 1: Loss = 0.6865
Iteration 2: Loss = 0.6804
Iteration 3: Loss = 0.6747
Iteration 4: Loss = 0.6695
Iteration 5: Loss = 0.6647
Iteration 6: Loss = 0.6602
Iteration 7: Loss = 0.6560
Iteration 8: Loss = 0.6521
Iteration 9: Loss = 0.6485
Iteration 10: Loss = 0.6451
Iteration 11: Loss = 0.6420
Iteration 12: Loss = 0.6390
Iteration 13: Loss = 0.6362
Iteration 14: Loss = 0.6336
Iteration 15: Loss = 0.6312
Iteration 16: Loss = 0.6288
Iteration 17: Loss = 0.6266
Iteration 18: Loss = 0.6245
Iteration 19: Loss = 0.6226

```

Figure 17: Loss values for each Iterations

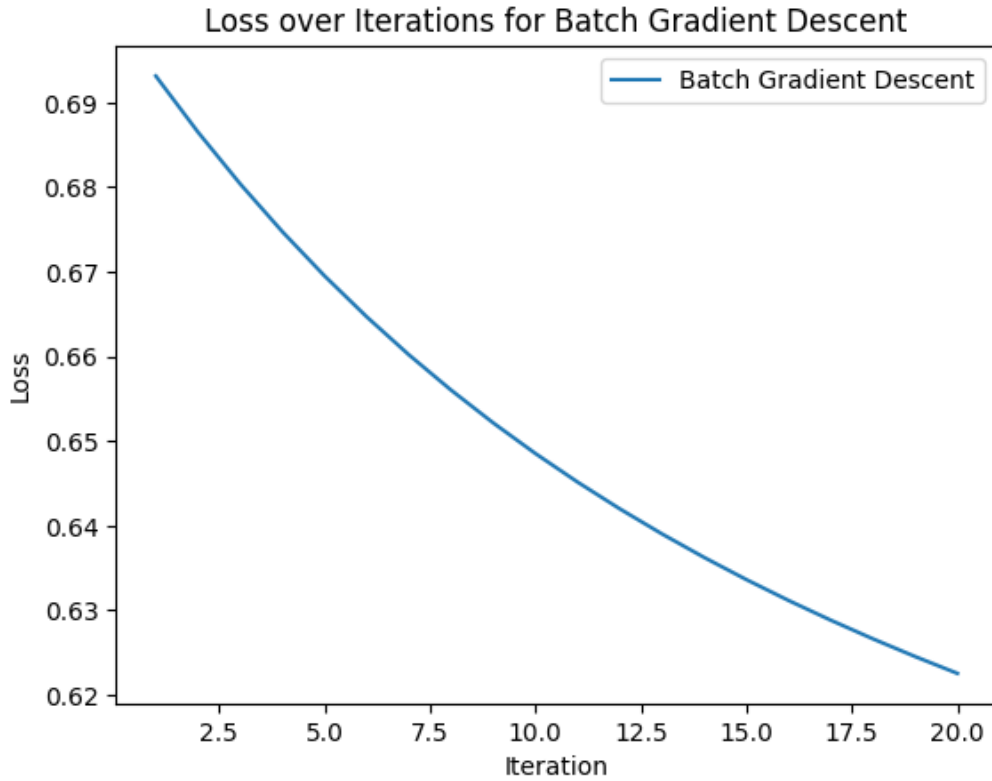


Figure 18: Loss Plots

With centers at $[3, 0]$ and $[5, 1.5]$, the increased overlap between classes leads to a less distinct decision boundary, making it harder for optimization algorithms to converge quickly.

- **Slower Loss Reduction:** The model struggles to classify points near the boundary, leading to a gradual decrease in loss.
- **Complex Error Surface:** Overlapping classes create a challenging error surface, requiring the model to balance misclassifications.
- **More Iterations Required:** This complexity necessitates additional iterations for convergence.

Overall, when classes overlap, optimization takes longer due to difficulties in defining the decision boundary.

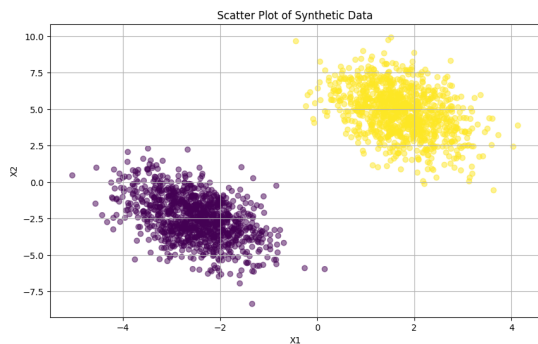
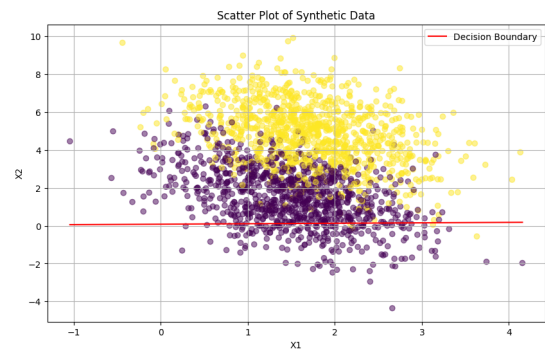
(a) Scatter plot when the center is $[-5, 0]$ and $[5, 1.5]$ (b) Scatter plot when the center is $[3, 0]$ and $[5, 1.5]$

Figure 19: Comparison of scatter plots with different centers