Факультет программной инженерии и компьютерной техники

Системы искусственного интеллекта

Лабораторная работа №7

Логистическая регрессия

Преподаватель: Кугаевских Александр Владимирович
Выполнил: Ле Чонг Дат
Группа: P33302

2023 г.

Лабораторная 7. Логистическая регрессия

логистическая регрессия.docx

1. Выбор датасета:

- Датасет о пассажирах Титаника: Titanic Dataset
- Датасет о диабете: Diabetes Dataset
- Загрузите выбранный датасет и выполните предварительную обработку данных.
- Получите и визуализируйте (графически) статистику по датасету (включая количество, среднее значение, стандартное отклонение, минимум, максимум и различные квантили).
- Разделите данные на обучающий и тестовый наборы в соотношении, которое вы считаете подходящим.
- Реализуйте логистическую регрессию "с нуля" без использования сторонних библиотек, кроме NumPy и Pandas. Ваша реализация логистической регрессии должна включать в себя:
  - Функцию для вычисления гипотезы (sigmoid function).
  - Функцию для вычисления функции потерь (log loss).
  - Метод обучения, который включает в себя градиентный спуск.
  - Возможность варьировать гиперпараметры, такие как коэффициент обучения (learning rate) и количество итераций.

1. Исследование гиперпараметров:
   - Проведите исследование влияния гиперпараметров на производительность модели. Варьируйте следующие гиперпараметры:
     - Коэффициент обучения (learning rate).
     - Количество итераций обучения.
     - Метод оптимизации (например, градиентный спуск или оптимизация Ньютона).
2. Оценка модели:
   - Для каждой комбинации гиперпараметров оцените производительность модели на тестовом наборе данных, используя метрики, такие как accuracy, precision, recall и F1-Score.

Сделайте выводы о том, какие значения гиперпараметров наилучшим образом работают для данного набора данных и задачи классификации. Обратите внимание на изменение производительности модели при варьировании гиперпараметров.

Step 1: Installing libraries

This step is usually done once. If you already have these libraries installed, you can skip this step

In [1]:
```
# !pip install numpy seaborn scipy
```

Step 2: Importing libraries and defining helper functions

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.model_selection import train_test_split
import warnings

warnings.filterwarnings('ignore')
```

In [3]:
```
def print_bold(text):
    BOLD = '\033[1m'
    END = '\033[0m'
    print('-' * 100 + f"\n{BOLD}{text}{END}")
```

In [4]:
```
def fill_missing_with_mean(column):
    mean_val = column.mean()
    return column.fillna(mean_val)
```

In [5]:
```
def min_max_scaler(column):
    # Calculate the minimum and maximum values of the column
    min_val = column.min()
    max_val = column.max()

    # Apply the Min-Max scaling formula
    scaled_column = (column - min_val) / (max_val - min_val)

    return scaled_column
```

Step 3: Performing tasks

Task 1: Получите и визуализируйте (графически) статистику по датасету (включая количество, среднее значение, стандартное отклонение, минимум, максимум и различные квантили).

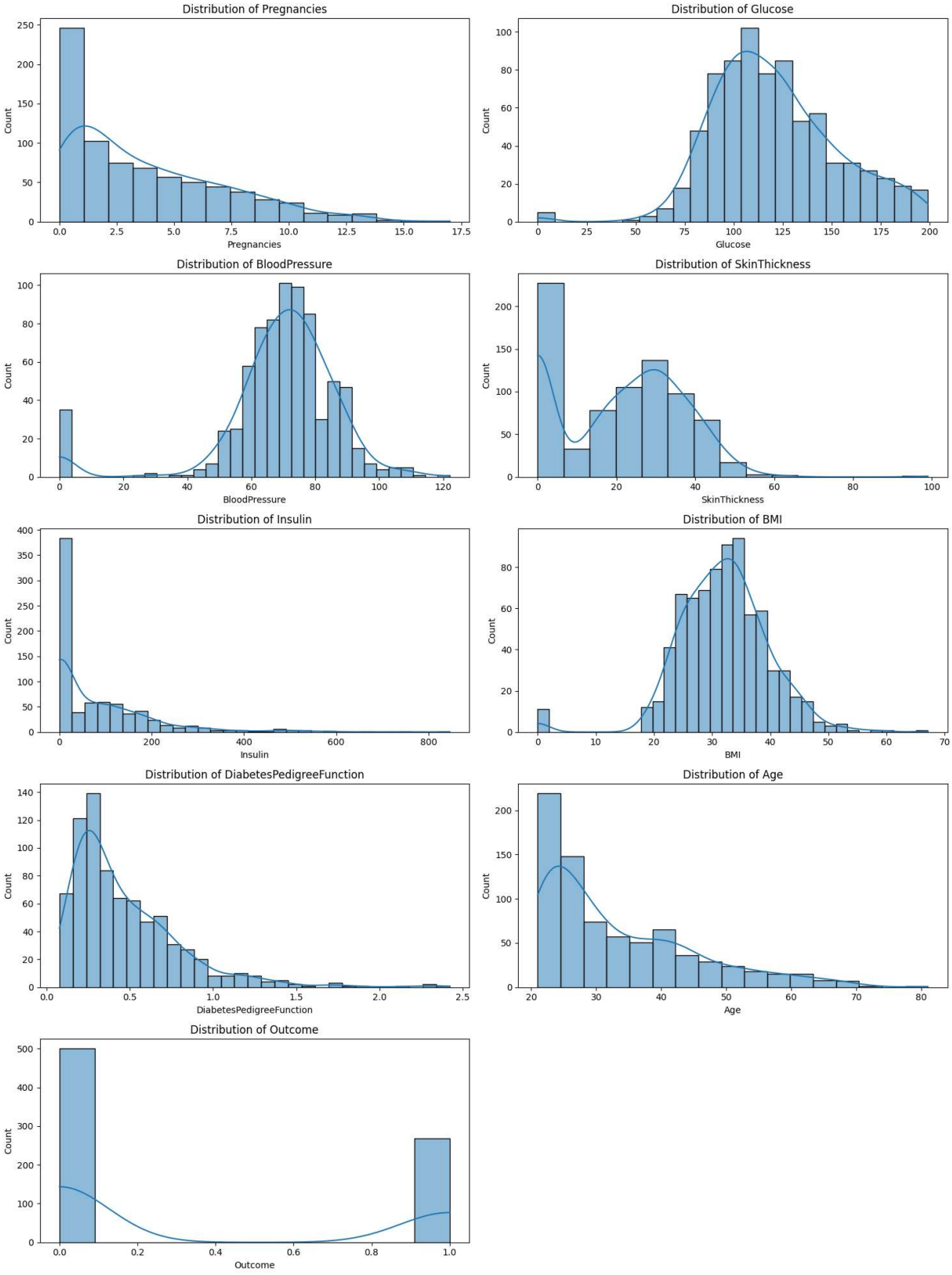
In [6]:
```
print_bold("Load the dataset")
df = pd.read_csv('diabetes.csv')

print_bold("Display the size of the dataset")
print(df.shape)
```

```
print_bold("Display basic statistics")
stats = df.describe()
print(stats)
```

```
--------------------------------------------------------------------------------
---------------
Load the dataset
--------------------------------------------------------------------------------
---------------
Display the size of the dataset
(768, 9)
--------------------------------------------------------------------------------
---------------
Display basic statistics
       Pregnancies     Glucose  BloodPressure  SkinThickness     Insulin  \
count   768.000000  768.000000     768.000000     768.000000  768.000000
mean      3.845052  120.894531      69.105469      20.536458   79.799479
std       3.369578   31.972618      19.355807      15.952218  115.244002
min       0.000000    0.000000       0.000000       0.000000    0.000000
25%       1.000000   99.000000      62.000000       0.000000    0.000000
50%       3.000000  117.000000      72.000000      23.000000   30.500000
75%       6.000000  140.250000      80.000000      32.000000  127.250000
max      17.000000  199.000000     122.000000      99.000000  846.000000

              BMI  DiabetesPedigreeFunction         Age     Outcome
count  768.000000                768.000000  768.000000  768.000000
mean    31.992578                  0.471876   33.240885    0.348958
std      7.884160                  0.331329   11.760232    0.476951
min      0.000000                  0.078000   21.000000    0.000000
25%     27.300000                  0.243750   24.000000    0.000000
50%     32.000000                  0.372500   29.000000    0.000000
75%     36.600000                  0.626250   41.000000    1.000000
max     67.100000                  2.420000   81.000000    1.000000
```

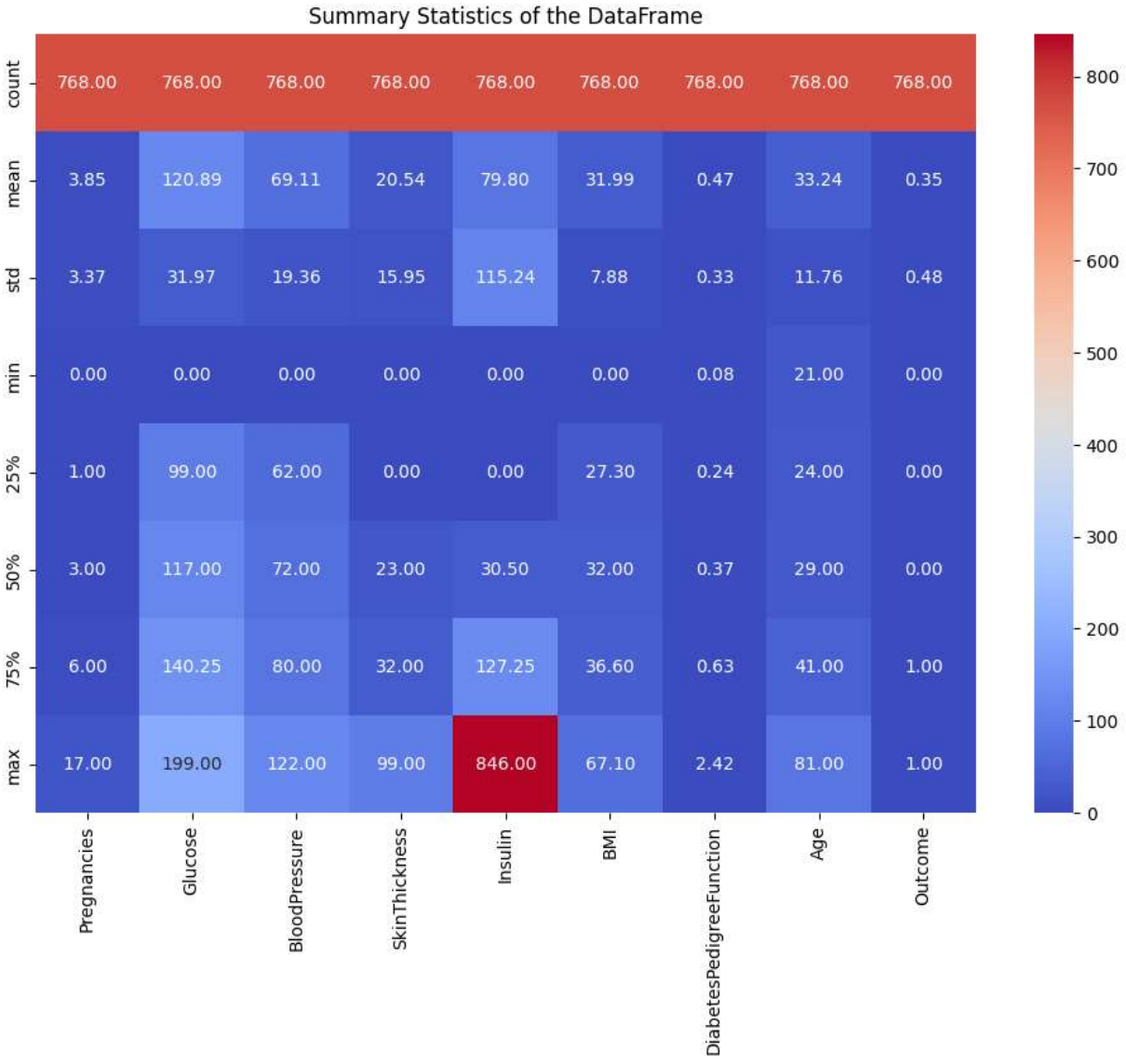
In [7]:

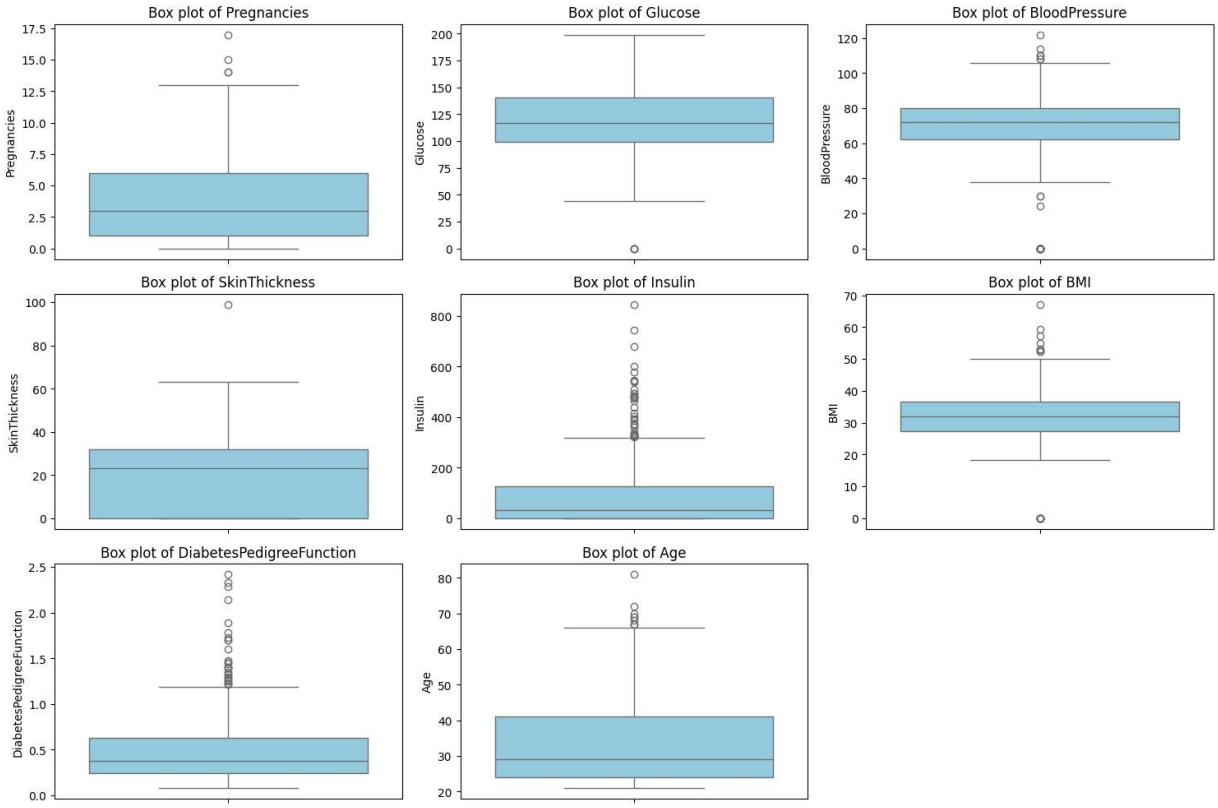
```
# Define the number of columns for visualization
num_cols = df.select_dtypes(include=['float64', 'int64']).columns

# Set the number of columns in each row of plots
num_plots_per_row = 2

# Calculate the number of rows needed for subplots
num_rows = (len(num_cols) + 1) // num_plots_per_row

plt.figure(figsize=(15, 4 * num_rows))

# Loop through all numerical columns
for i, col in enumerate(num_cols, 1):
    # Distribution plot
    plt.subplot(num_rows, num_plots_per_row, i)
    sns.histplot(df[col], kde=True)
    plt.title('Distribution of ' + col)

plt.tight_layout()
plt.show()
```



In [8]:
```
# Summary statistics of the DataFrame
summary_statistics = df.describe()

# Plotting
plt.figure(figsize=(12, 8))

# Heatmap for the summary statistics
```

```
sns.heatmap(summary_statistics, annot=True, fmt=".2f", cmap="coolwarm")
plt.title("Summary Statistics of the DataFrame")
plt.show()

plt.figure(figsize=(15, 10))
boxplot_color = 'skyblue'

# Loop through all numerical columns
for i, col in enumerate(num_cols.drop('Outcome')):
    plt.subplot(3, 3, i + 1)
    sns.boxplot(y=df[col], color=boxplot_color)
    plt.title(f'Box plot of {col}')
    plt.ylabel(col)

plt.tight_layout()
plt.show()
```





In [9]:
```
# Apply the function to each numerical column
numerical_cols = df.select_dtypes(include=['float64', 'int64']).columns
df[num_cols] = df[num_cols].apply(fill_missing_with_mean)

# Print the first few rows to check the result
print(df.head())
```

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

In [10]:
```
# Exclude the 'Outcome' column and apply Min-Max scaling to the rest of the columns
features = df.columns.drop('Outcome')
df[features] = df[features].apply(min_max_scaler)

# Print the first few rows of the scaled dataframe to check the result
print(df.head())
```

```
   Pregnancies   Glucose  BloodPressure  SkinThickness   Insulin       BMI  \
0     0.352941  0.743719       0.590164       0.353535  0.000000  0.500745
1     0.058824  0.427136       0.540984       0.292929  0.000000  0.396423
2     0.470588  0.919598       0.524590       0.000000  0.000000  0.347243
3     0.058824  0.447236       0.540984       0.232323  0.111111  0.418778
4     0.000000  0.688442       0.327869       0.353535  0.198582  0.642325

   DiabetesPedigreeFunction       Age  Outcome
0                  0.234415  0.483333        1
1                  0.116567  0.166667        0
2                  0.253629  0.183333        1
3                  0.038002  0.000000        0
4                  0.943638  0.200000        1
```

Task 2: Разделите данные на обучающий и тестовый наборы в соотношении, которое вы считаете подходящим.

In [11]:
```
X = df.drop('Outcome', axis=1)  # Features
y = df['Outcome']  # Target variable

# Splitting the dataset into the Training set and Test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta
```

Task 3:

- Реализуйте логистическую регрессию "с нуля" без использования сторонних библиотек, кроме NumPy и Pandas. Ваша реализация логистической регрессии должна включать в себя:
  - Функцию для вычисления гипотезы (sigmoid function).
  - Функцию для вычисления функции потерь (log loss).
  - Метод обучения, который включает в себя градиентный спуск.
  - Возможность варьировать гиперпараметры, такие как коэффициент обучения (learning rate) и количество итераций.

In [12]:
```
import numpy as np

class LogisticRegression:
    def __init__(self, learning_rate=0.01, max_iter=100, optimization_method='gradi
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.optimization_method = optimization_method
        self.weights = None

    def _sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def _cost_function(self, X, y):
        m = len(y)
        h = self._sigmoid(X @ self.weights)
        return -1/m * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))

    def _gradient_descent(self, X, y):
```

```
        m = len(y)
        for _ in range(self.max_iter):
            h = self._sigmoid(X @ self.weights)
            gradient = np.dot(X.T, (h - y)) / m
            self.weights -= self.learning_rate * gradient

    def _newton_method(self, X, y):
        m = len(y)
        for _ in range(self.max_iter):
            h = self._sigmoid(X @ self.weights)
            gradient = np.dot(X.T, (h - y)) / m
            # Create a diagonal matrix for element-wise multiplication
            S = np.diag(h * (1 - h))
            H = X.T @ S @ X / m
            self.weights -= np.linalg.inv(H) @ gradient

    def fit(self, X, y):
        X = np.insert(X, 0, 1, axis=1)  # Add intercept
        self.weights = np.zeros(X.shape[1])
        if self.optimization_method == 'gradient_descent':
            self._gradient_descent(X, y)
        elif self.optimization_method == 'newton':
            self._newton_method(X, y)

    def predict(self, X):
        X = np.insert(X, 0, 1, axis=1)  # Add intercept
        probabilities = self._sigmoid(X @ self.weights)
        return np.where(probabilities >= 0.5, 1, 0)
```

In [13]:

```
def calculate_accuracy(y_true, y_pred):
    correct = sum([1 for true, pred in zip(y_true, y_pred) if true == pred])
    return correct / len(y_true)

def calculate_precision(y_true, y_pred):
    true_positives = sum([1 for true, pred in zip(y_true, y_pred) if true == pred =
    predicted_positives = sum([1 for pred in y_pred if pred == 1])
    return true_positives / predicted_positives if predicted_positives != 0 else 0

def calculate_recall(y_true, y_pred):
    true_positives = sum([1 for true, pred in zip(y_true, y_pred) if true == pred =
    actual_positives = sum([1 for true in y_true if true == 1])
    return true_positives / actual_positives if actual_positives != 0 else 0

def calculate_f1_score(y_true, y_pred):
    precision = calculate_precision(y_true, y_pred)
    recall = calculate_recall(y_true, y_pred)
    return 2 * (precision * recall) / (precision + recall) if (precision + recall)

def evaluate_model(model, X_test, y_test):
    predictions = model.predict(X_test)
    accuracy = calculate_accuracy(y_test, predictions)
    precision = calculate_precision(y_test, predictions)
    recall = calculate_recall(y_test, predictions)
    f1 = calculate_f1_score(y_test, predictions)
    return accuracy, precision, recall, f1
```

```python
# Hyperparameters to test
learning_rates = [0.01, 0.1, 0.5]
iterations = [100, 500, 1000]
optimization_methods = ['gradient_descent', 'newton']

# Store the results
results = []

# from sklearn.linear_model import LogisticRegression

for lr in learning_rates:
    for itr in iterations:
        for om in optimization_methods:
            model = LogisticRegression(learning_rate=lr, max_iter=itr, optimization
            # model = LogisticRegression(C=lr, max_iter=itr, solver='lbfgs')
            model.fit(X_train, y_train)

            # Evaluate the model
            accuracy, precision, recall, f1 = evaluate_model(model, X_test, y_test)
            results.append((lr, itr, accuracy, precision, recall, f1, om))


# Convert the results into a pandas DataFrame
results_df = pd.DataFrame(results, columns=['Learning Rate', 'Iterations', 'Accurac

# Sorting the DataFrame by the 'Accuracy' column in descending order
sorted_results_df = results_df.sort_values(by='Accuracy', ascending=False)

# Set display options to avoid line breaks in columns
pd.set_option('display.max_colwidth', None)
pd.set_option('display.expand_frame_repr', False)

# Display the sorted DataFrame
print(sorted_results_df)
```

	Learning Rate	Iterations	Accuracy	Precision	Recall	F1-Score	Optimization Method
16	0.50	1000	0.748918	0.657143	0.5750	0.613333	gradient_descent
9	0.10	500	0.735931	0.617284	0.6250	0.621118	newton
7	0.10	100	0.735931	0.617284	0.6250	0.621118	newton
15	0.50	500	0.735931	0.617284	0.6250	0.621118	newton
13	0.50	100	0.735931	0.617284	0.6250	0.621118	newton
11	0.10	1000	0.735931	0.617284	0.6250	0.621118	newton
1	0.01	100	0.735931	0.617284	0.6250	0.621118	newton
17	0.50	1000	0.735931	0.617284	0.6250	0.621118	newton
5	0.01	1000	0.735931	0.617284	0.6250	0.621118	newton
3	0.01	500	0.735931	0.617284	0.6250	0.621118	newton
14	0.50	500	0.727273	0.644068	0.4750	0.546763	gradient_descent
10	0.10	1000	0.705628	0.657895	0.3125	0.423729	gradient_descent
8	0.10	500	0.666667	0.666667	0.0750	0.134831	gradient_descent
12	0.50	100	0.666667	0.666667	0.0750	0.134831	gradient_descent
6	0.10	100	0.653680	0.000000	0.0000	0.000000	gradient_descent
4	0.01	1000	0.653680	0.000000	0.0000	0.000000	gradient_descent
2	0.01	500	0.653680	0.000000	0.0000	0.000000	gradient_descent
0	0.01	100	0.653680	0.000000	0.0000	0.000000	gradient_descent

Conclusion

1. **Best Performing Hyperparameters**:
    - The highest accuracy is achieved with a learning rate of 0.50 and 1000 iterations using the gradient descent optimization method. This combination also shows good balance across precision, recall, and F1-score. This suggests that a higher learning rate and sufficient iterations are beneficial for this dataset.
2. **Effect of Optimization Method**:
    - The Newton optimization method generally performs comparably to gradient descent in terms of accuracy, precision, recall, and F1-score for most combinations.

However, it doesn't reach the peak performance of gradient descent with the highest learning rate and iteration count.

3. **Influence of Learning Rate and Iterations**:
    - With a learning rate of 0.10 and iterations varying from 100 to 1000, the performance metrics are quite consistent across both optimization methods. This indicates stability at this learning rate.
    - For a learning rate of 0.01, the model consistently predicts a single class (precision, recall, and F1-score are 0), regardless of the number of iterations and optimization method. This suggests that the learning rate is too low to converge effectively for this dataset.
4. **Performance at Different Iterations**:
    - At 500 and 1000 iterations, the models generally perform better than at 100 iterations for a higher learning rate (0.50), indicating that more iterations allow the model to learn more effectively.
    - However, for a lower learning rate (0.10 and 0.01), increasing iterations doesn't seem to significantly impact the model's performance, possibly due to a too-slow learning process.
5. **Gradient Descent vs. Newton Method**:
    - Gradient descent seems to outperform the Newton method at higher learning rates and iterations, suggesting it might be a more suitable optimization method for this particular dataset.
6. **Class Imbalance Indication**:
    - The zero precision, recall, and F1-score for some hyperparameter settings might indicate class imbalance in the dataset, where the model tends to predict only them to model tuning.