Handwritten-Digit-Recognition-with-Logistic-Regression

Silent0Wings

August 2, 2025

Contents

1 Introduction and Data Preparation

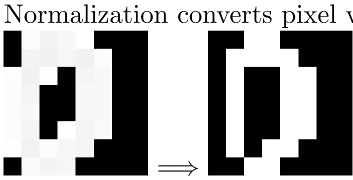
This report implements and evaluates a digit classifier using scikit-learn's logistic regression on two different datasets:

- The **"load-digits"** dataset from scikit-learn [1], consisting of 1,797 grayscale images sized 8×8 pixels. Pixel values range from 0 to 16 and are normalized by dividing by 16 to scale between 0 and 1 for better performance.
- The **MNIST** dataset in CSV format [2] containing 70,000 grayscale images sized 28×28 pixels, normalized by dividing pixel values by 255.

These two datasets are distinct: `load-digits` is a smaller, low-resolution dataset often used for quick prototyping, while MNIST is a larger, widely used benchmark for handwritten digit classification.

```
from sklearn.datasets import load_digits
digits = load_digits()
x = digits.data / 16.0   # Normalize pixels to 0-1
y = digits.target
```

Normalization converts pixel values where 16 = white, 0 = black, and 8 = gray:



We export the dataset to CSV for easy inspection and manipulation:

```
import pandas as pd
data = pd.DataFrame(digits.data)
data['label'] = digits.target
data.to_csv('digits_dataset.csv', index=False)
```

This visualizes pixel intensities as digits from 0 to 16:

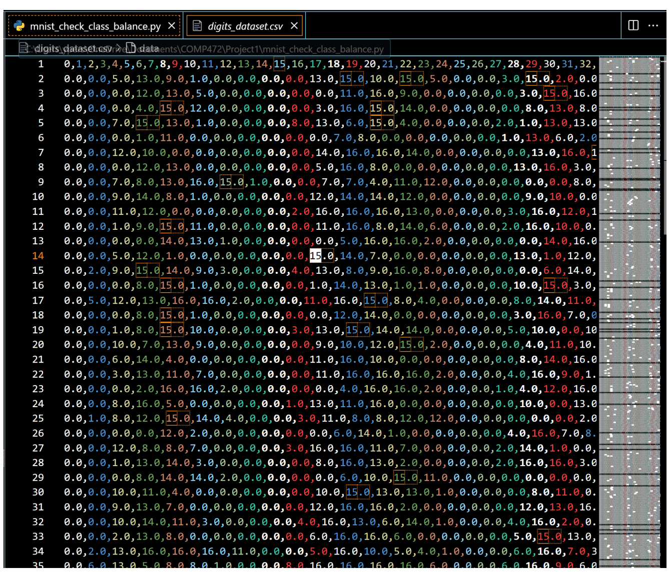


Figure 1: image of scikit-learn load-digits dataset (8×8)

Figure 2: image of MNIST dataset (28×28)

To convert CSV rows back into images saved by label folder:

```python
import numpy as np
import os
import cv2

df = pd.read_csv('digits_dataset.csv')
output_dir = 'processed_images'
os.makedirs(output_dir, exist_ok=True)

for idx, row in df.iterrows():
    label = row['label']
    pixels = np.array(row[1:]).reshape(8, 8).astype(np.uint8) * 255
    label_folder = os.path.join(output_dir, str(label))
    os.makedirs(label_folder, exist_ok=True)
    filename = os.path.join(label_folder, f'image_{idx}.png')
    cv2.imwrite(filename, pixels)
```

All images are saved in folders by digit, easy to navigate:

```
PS C:\Users\ypers\OneDrive\Documents\COMP472\Project1> cd .\processed_images\
PS C:\Users\ypers\OneDrive\Documents\COMP472\Project1\processed_images> tree
Folder PATH listing for volume Windows-SSD
Volume serial number is FE20-4359
C:.
├───0.0
├───1.0
├───2.0
├───3.0
├───4.0
├───5.0
├───6.0
├───7.0
├───8.0
└───9.0
```

Here are four sample images per digit (0–9), showing balance and handwriting variation:

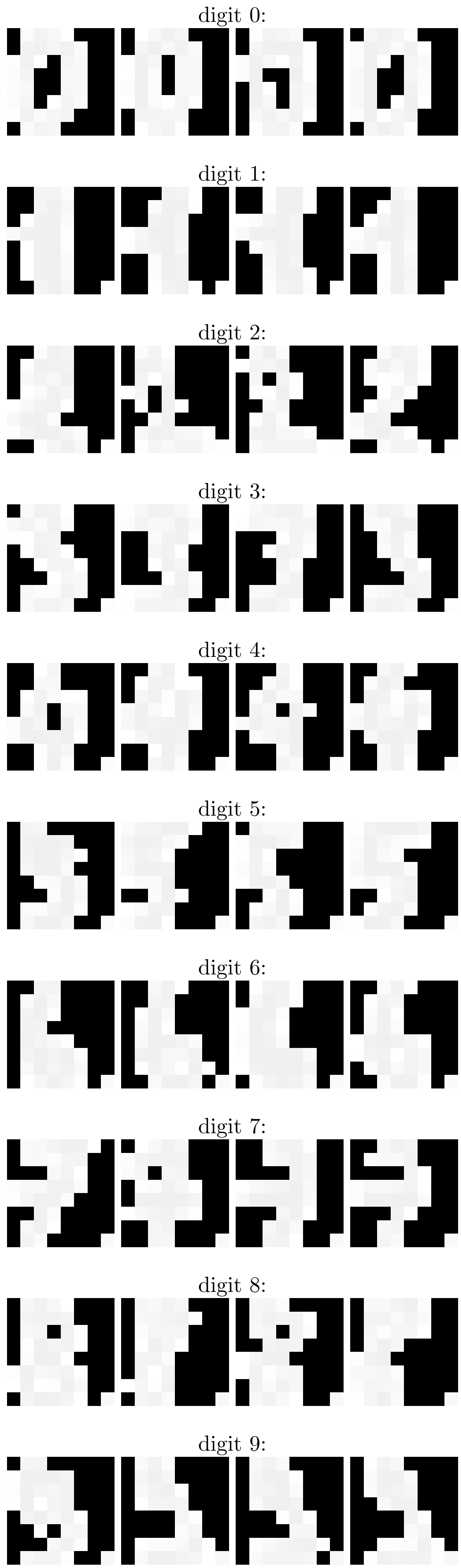


Figure 3: Four sample images per digit (0–9); shows dataset balance and handwriting variation.

We confirm dataset balance with this class distribution plot:

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('digits_dataset.csv')
label_counts = df['label'].value_counts().sort_index()

ax = label_counts.plot(kind='bar', color='black', edgecolor='gray', title='Class Distribution')
plt.xlabel('Digit')
plt.ylabel('Count')

for idx, val in enumerate(label_counts):
    ax.text(idx, val + 0.5, str(val), ha='center', va='bottom')

plt.show()
```

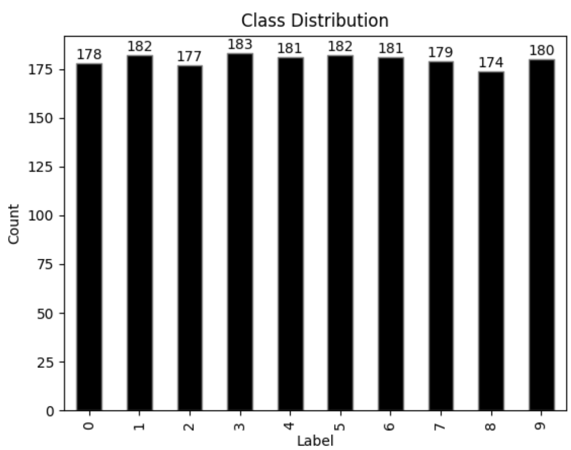


Figure 4: Digit distribution for scikit-learn dataset

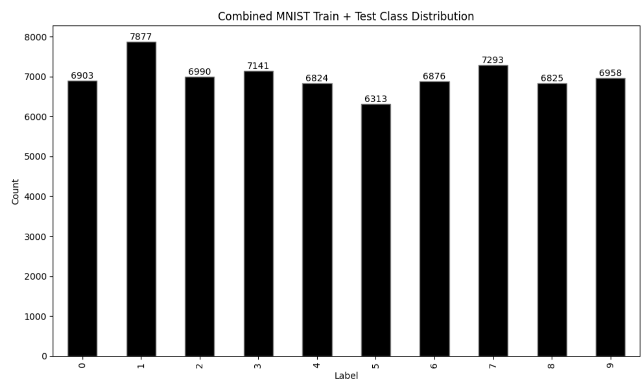


Figure 5: Digit distribution for MNIST dataset

1.1 Training Setup and Evaluation Approach

We apply an 80/20 train/test split and use logistic regression with tuned parameters to train a handwritten digit classifier. The setup is designed to ensure efficient convergence and balanced performance.

The model's performance will be evaluated using key metrics:

- Overall accuracy
- Precision, recall, and F1 scores (via classification report)
- Error patterns (via confusion matrix)

The implementation relies on established machine learning libraries and standard evaluation workflows to ensure reliable and reproducible results.

1.2 What is Logistic Regression?

Logistic regression is a simple but powerfull model for clasification. In stats, it is often used to tell if something belongs to a set or not — like yes (1) or no (0).

In our project, we are not doing just yes or no, we have 10 classes (digits 0 to 9). Because we use the `'lbfgs'` solver, scikit-learn handles multiclass directly using softmax [3]. This means the model looks at all the classes together and picks the one with the highest score.

After we train the model on the train data, we use the left over data to check how good it works.

2 Parameter Tuning & Testing

For parameter tuning, you can explore various options available in the LogisticRegression documentation [3].

In this project, I iteratively test multiple values for each key hyperparameter separately, aiming to find the value that yields the best model accuracy. This approach is a form of manual hyperparameter tuning where each parameter is varied individually rather than exhaustively searching all combinations.

The parameters tuned include:

- **max_iter**: Maximum iterations allowed for the solver to converge.
- **solver**: The optimization algorithm used (e.g., `lbfgs`, `saga`, `newton-cg`, `liblinear`).
- **class_weight**: Strategy to handle imbalanced classes (e.g., `None` or `balanced`).
- **C**: Inverse of regularization strength; smaller values specify stronger regularization.
- **normalization value**: Pixel value scaling factor used to normalize input data (e.g., dividing by 16).

This method is simpler than full grid search or random search, focusing on tuning each parameter individually to identify its impact on model performance. For more details on hyperparameter tuning techniques including grid and random search, see [5] and the scikit-learn documentation [6, 7].

Normalization Value vs Accuracy

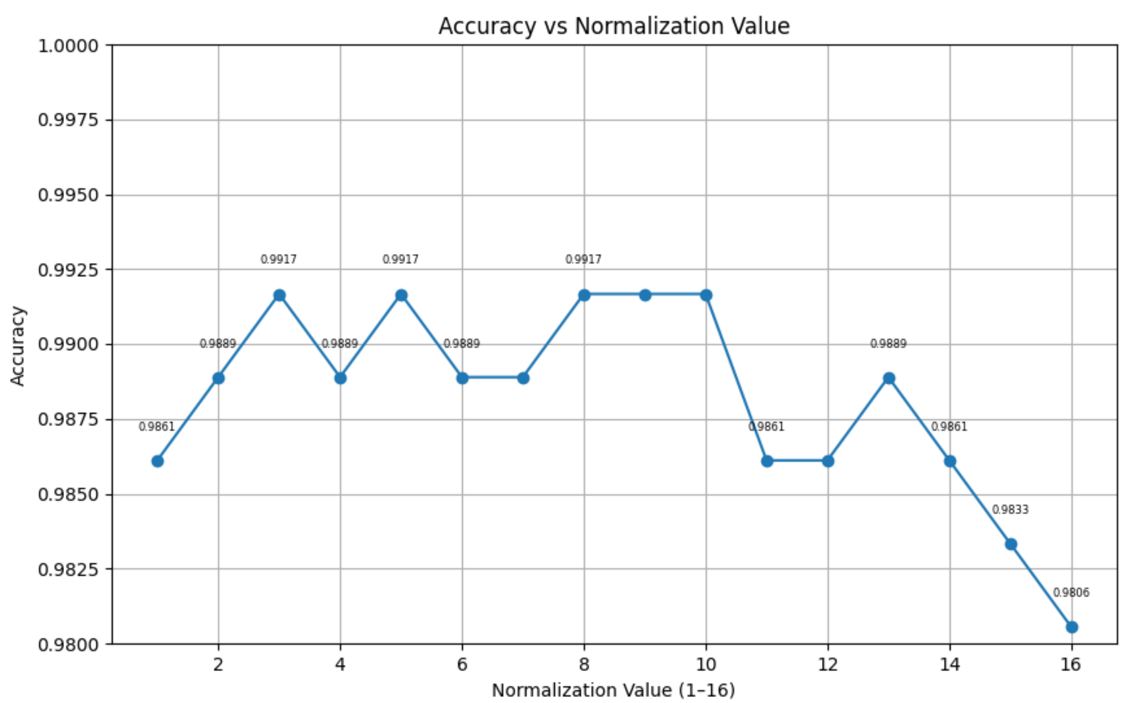


Figure 6: Accuracy stays high (98.0%–99.2%) across scaling factors 1 to 16. The best accuracy (99.2%) occurs at 3,5,8,9,10. Note: Only dividing by 16 produces true normalization (0–1 range); other divisors apply partial scaling, not full normalization.

Test Size vs Accuracy

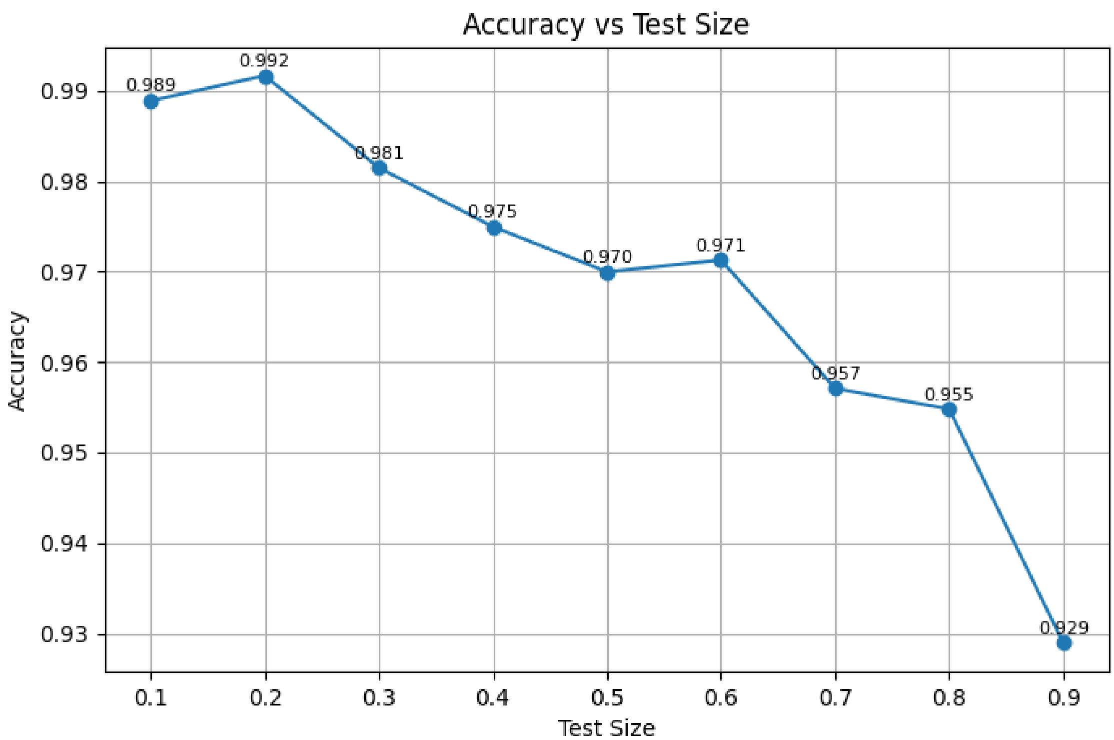


Figure 7: Accuracy vs Test Size; accuracy drops sharply below 80% training share

Accuracy vs Random Seed

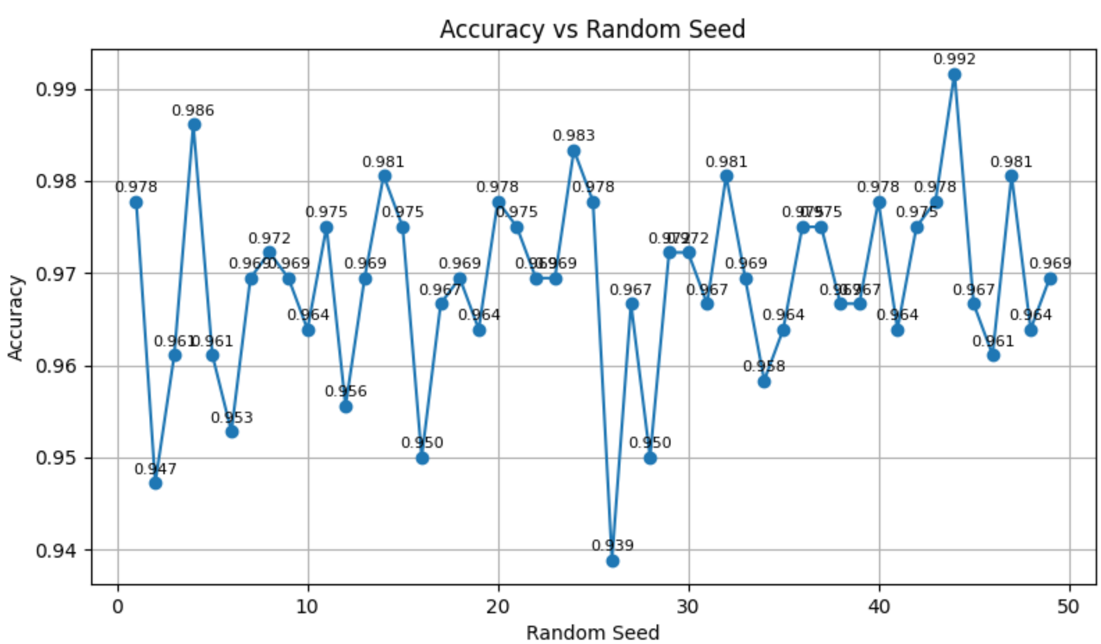


Figure 8: Accuracy vs Random Seed; accuracy remains stable regardless of seed choice max at 44

Accuracy vs Max Iteration

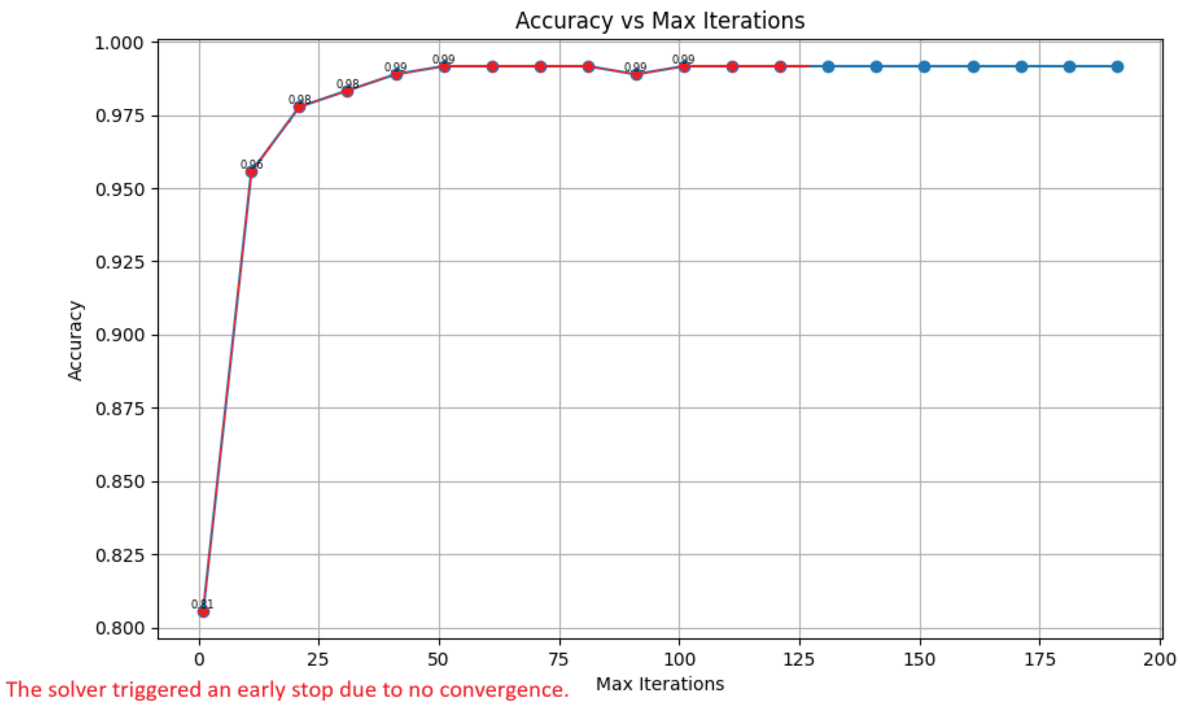


Figure 9: Accuracy vs. Max Iterations: The model's accuracy levels off after about 126 iterations. Setting `max_iter` to 128 gives good results without wasting time. Raising `max_iter` higher does not add much benefit and only makes training slower. "However, if the number is set too high, the training process might become unnecessarily long without significant gains in performance." [4]

Accuracy Comparison Across Solvers

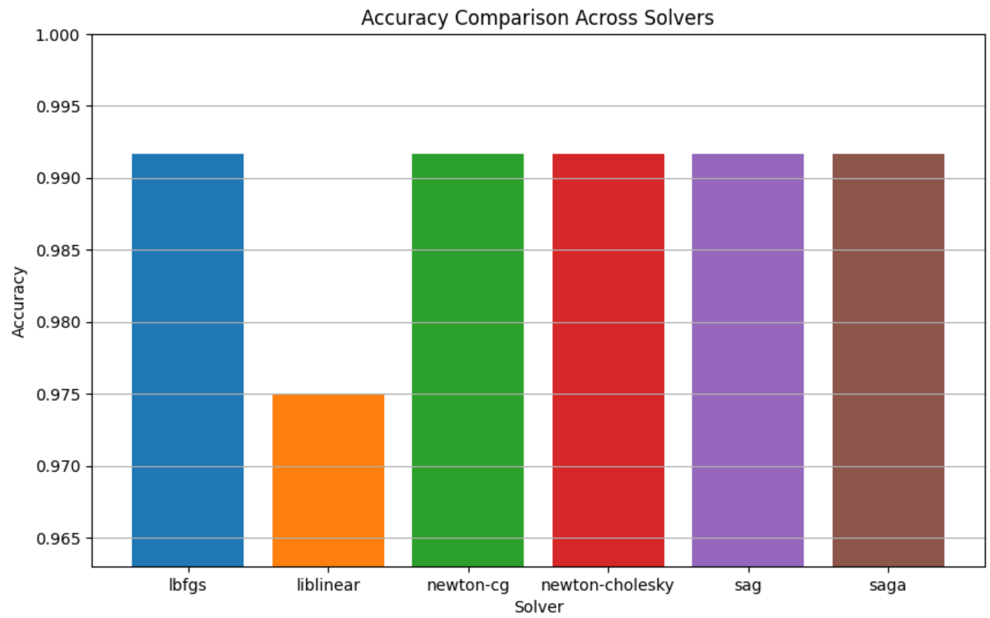


Figure 10: Accuracy comparison across solvers. Multiclass solvers perform similarly; only `liblinear` differs. We use the default `lbfgs`.

Accuracy vs Regularization

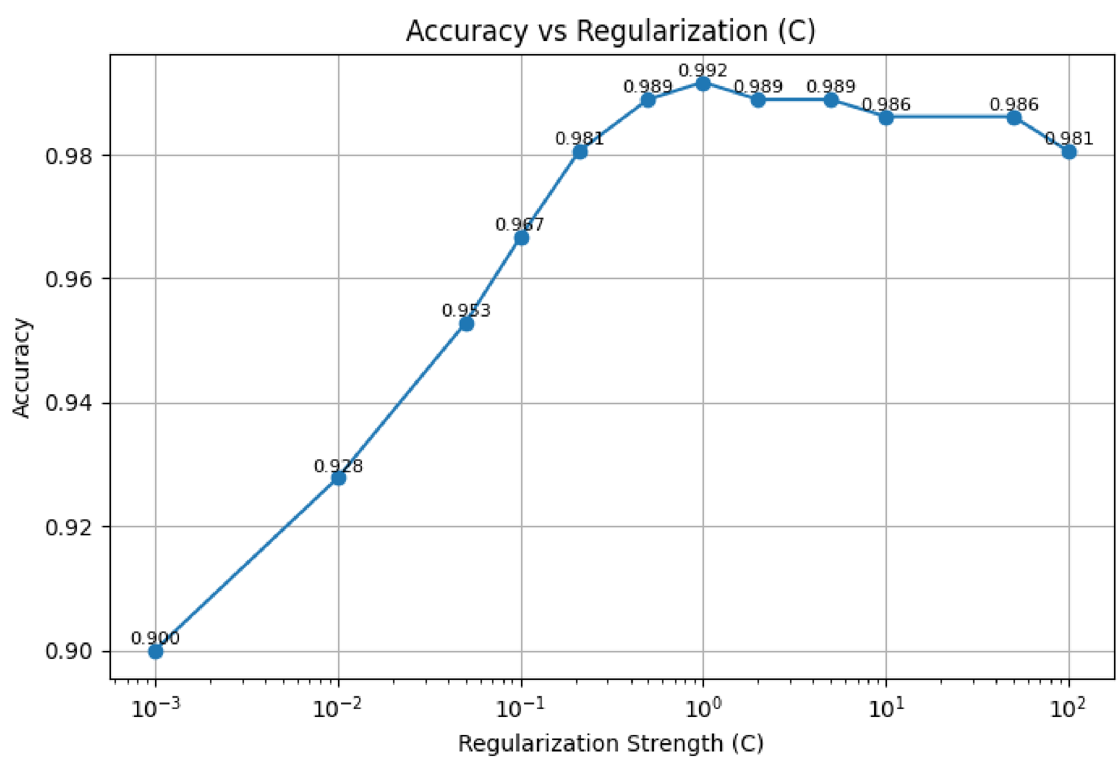


Figure 11: Accuracy vs Regularization; best performance at medium `C` values with `C`=1

This means `C`=1 works best because it avoids being too strict or too loose, keeping the model balanced.

Accuracy vs Class Weight

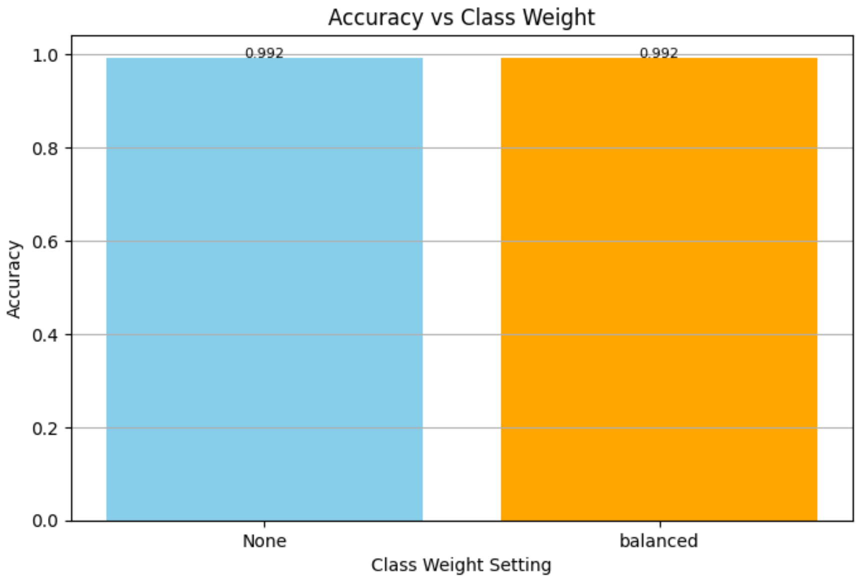


Figure 12: Accuracy vs Class Weight: This plot compares model accuracy using `class_weight=None` and `class_weight='balanced'`. Since the dataset is highly balanced (each digit averages 176 samples, range 174–183), applying class weights provides no meaningful performance gain. The classifier already handles slight class size differences, resulting in nearly identical accuracy for both settings.

Final Recommendations

We set `max_iter = 128` since accuracy levels off beyond this, avoiding long training [3, 4]. We scaled pixels by `9.0` to keep feature values manageable. We used `lbfgs` as it works well for multiclass tasks [3].

- Solver: `lbfgs`
- Regularization: `C = 1.0`
- Pixel scaling: divide by `9.0`
- Iterations: `max_iter = 128`
- Random seed: 44
- Split: 80/20 (`test_size = 0.2`)
- Class weight: `None`

Best Parameter Configuration

Listing 1: Best Parameters Applied

```
# Train logistic regression
model = LogisticRegression(
    max_iter= 128,          # Ensure enough iterations for convergence
    solver='lbfgs',                             # Best multiclass solver
    class_weight=None,                         # Balanced dataset, no weighting needed
    C=1.0                               # Regularization strength
)  # Set up logistic regression
model.fit(X_train, y_train)  # Train the model
```

Achieved accuracy:

Accuracy: 0.9916666666666667

Note: Increasing `max_iter` to 128 improved accuracy compared to lower values, as 126 appears to be the minimum needed for convergence without triggering early stopping.

3 Results

3.1 Model and Evaluation

We used an 80/20 train/test split and logistic regression with `max_iter=128` for convergence.

Note: Setting `random_state=44` ensures consistent train/test splits for reproducible results. Initially, `max_iter=10000` guaranteed convergence, but later experiments show that `max_iter=128` suffices, improving efficiency without sacrificing accuracy.

Model performance was evaluated using accuracy, the classification report (precision, recall, F1), and the confusion matrix.

Imports required to run the project code:

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

Set up the parameters:

```
# parameter setup for logistic regression
iterations_max = 128 # maximum amount of iterations
number_to_divide_pixels_for_normal = 9.0 # divisor for partial normalisation
size_of_test_data = 0.2  # 20% testing, 80% training
random_seed_number = 44  # random seed for reproducibility
solver = 'lbfgs'  # solver type
regularization_c = 1.0  # regularization strength
```

Load and split the data:

```
digits = load_digits()  # load built-in digit data from sklearn
X = digits.data / number_to_divide_pixels_for_normal  # image data
y = digits.target  # label data
all_the_indexs = np.arange(len(X))

# split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test, idx_train, idx_test = train_test_split(
    X, y, all_the_indexs, test_size=size_of_test_data, random_state=random_seed_number
)
```

Set up and train the logistic regression model:

```
model = LogisticRegression(
    max_iter=max(iterations_max, 128),  # ensure enough iterations
    solver=solver,                      # best multiclass solver
    class_weight=None,                  # balanced dataset, no weighting needed
    C=regularization_c                  # regularization strength
)
model.fit(X_train, y_train)  # train the model
```

Evaluate and print accuracy:

```
y_pred = model.predict(X_test)  # make predictions
acc = accuracy_score(y_test, y_pred)  # compute accuracy
print("Accuracy:", acc)
```

Output:

```
Accuracy: 0.9916666666666667
```

full code : Section 6.1

3.2 Classification Report

We refer to the `plot_classification_report` function (Listing 3) inside **plot_utils.py**, which visualizes the classification report as a heatmap.

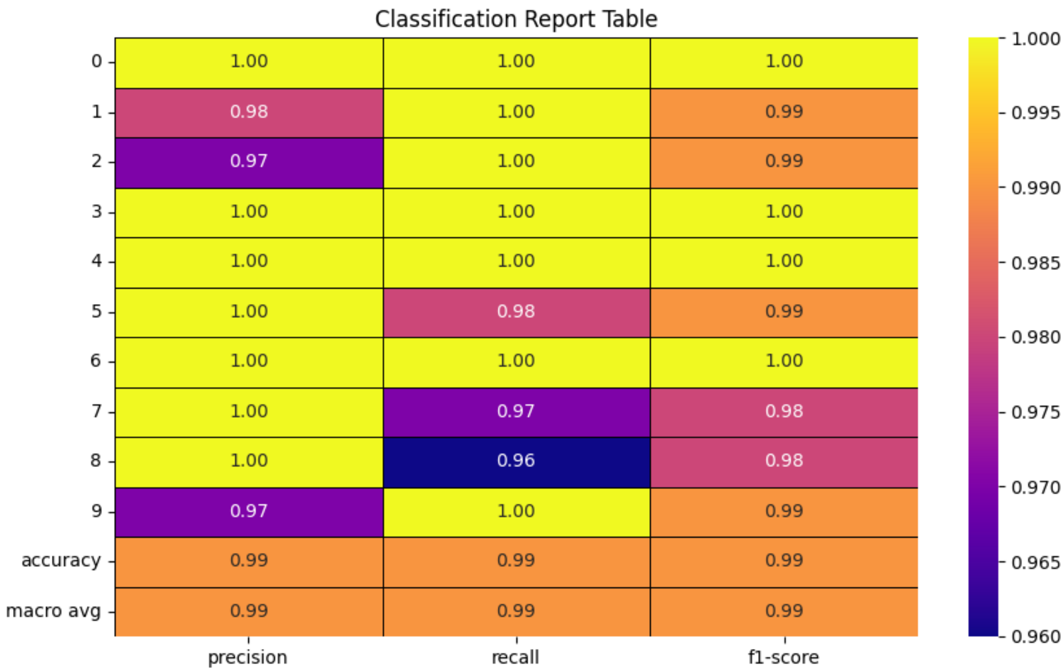


Figure 13: Classification Report Table; overall ∼99% accuracy, minor challenges on digits 5 and 9

The model shows very strong performance with precision, recall, and F1 scores between 0.99 and 1.00. The macro and weighted averages are approximately 0.99, indicating consistent, near-perfect accuracy across all digits.

3.3 Confusion Matrix

We refer to the `plot_confusion_matrix` function (Listing 4) inside **plot_utils.py**, which visualizes the confusion matrix as a heatmap.

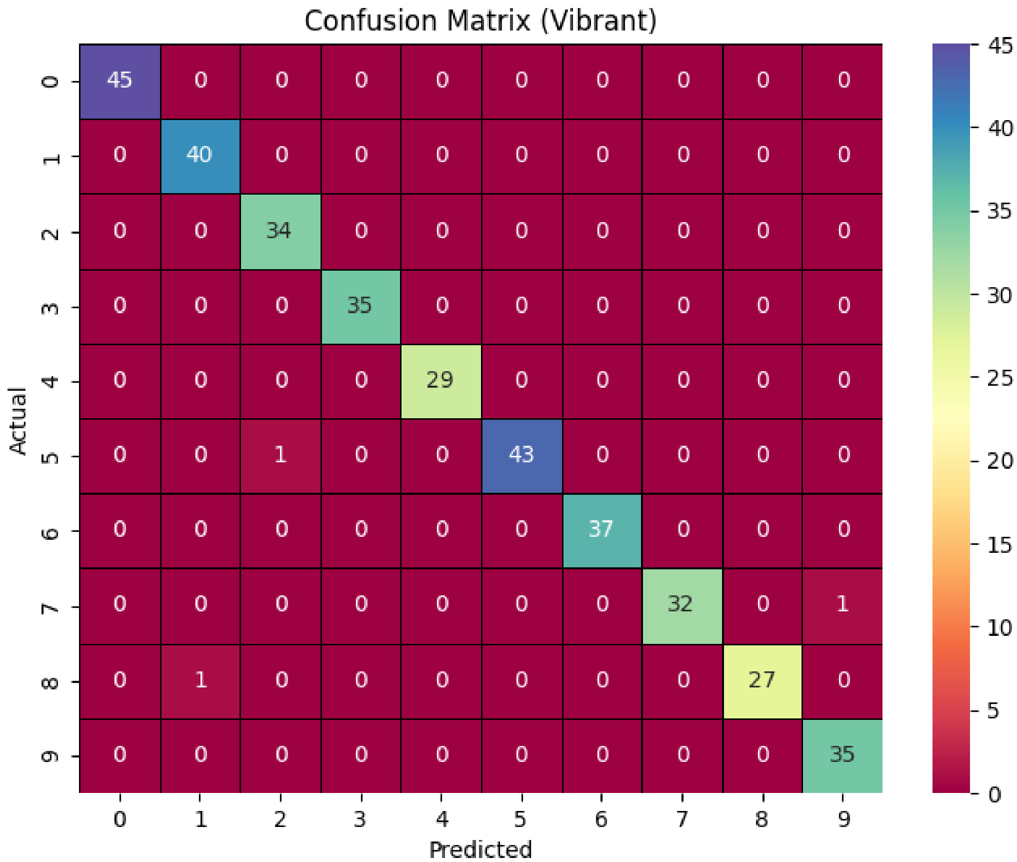


Figure 14: Confusion Matrix showing near-perfect predictions with only minor errors on digits 5, 7, and 8.

The confusion matrix shows almost all predictions are correct, with just a few small errors mainly on digits 5, 7, and 8 — reflecting the overall high accuracy of 99.2

4 Sample Prediction

Below is an example of a test image with its predicted and actual label (Listing 2) inside **plot_utils.py**, including the accuracy status:

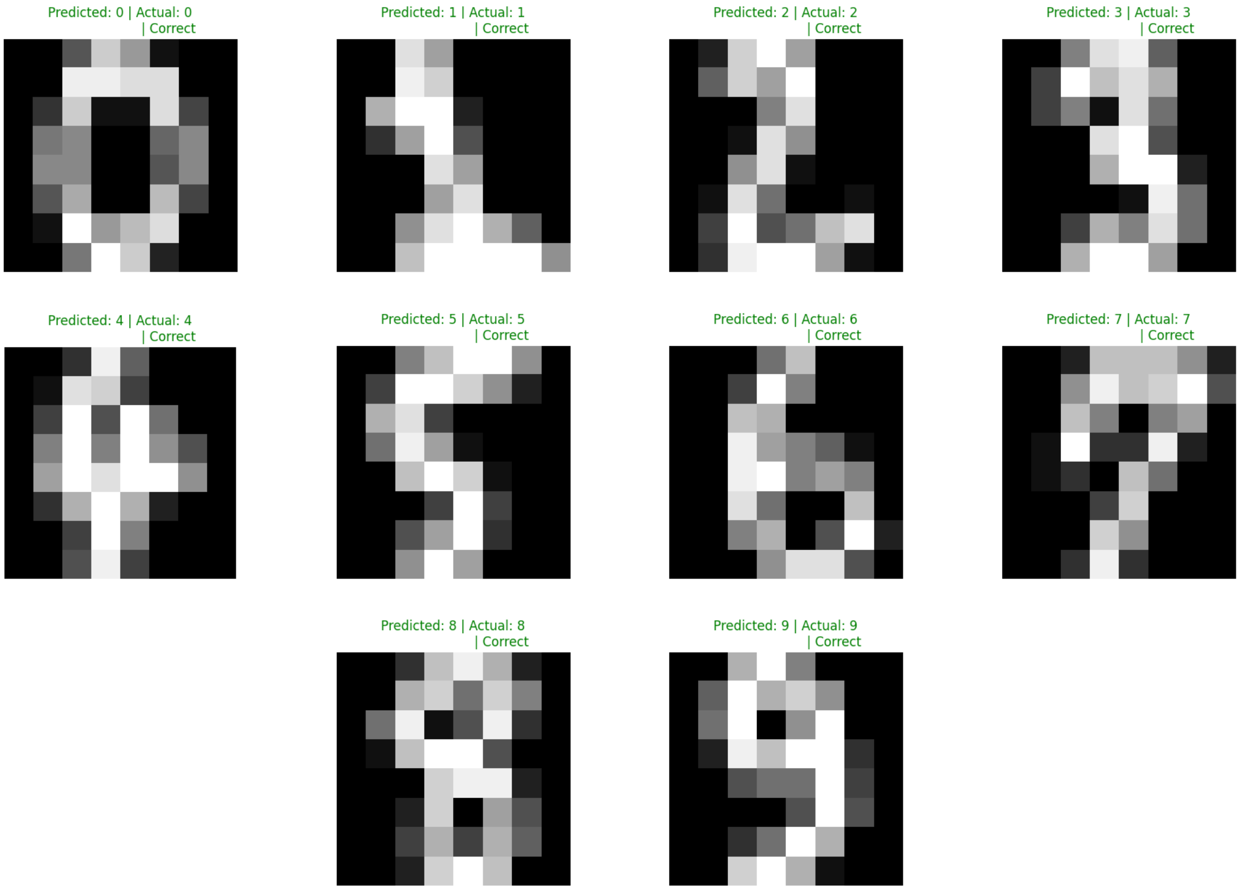


Figure 15: Sample predictions and key visual results

5 MNIST Logistic Regression: Small vs Large Dataset

MNIST Dataset on Kaggle [2]

Parameters

Same as Section 2, except for:

- **max_iter = 1000**: Maximum iterations allowed for solver convergence (since its a bigger dataset).
- **Normalization divisor = 255.0**: Pixel values scaled to the [0, 1] range.

The code for this is in Section 6.2.

Overview

We compare logistic regression performance on:

- Small dataset (scikit-learn dataset): 1,438 train / 359 test (80/20 split)
- Full MNIST: 60,000 train / 10,000 test (80/20 split)

This isolates dataset size effects on accuracy and generalization.

Accuracy Comparison

- **Small dataset:** 99.2%
- **Full dataset:** 92.2%

The 4.5% drop shows larger data size challenges model generalization despite proportional splits.

Confusion Matrices

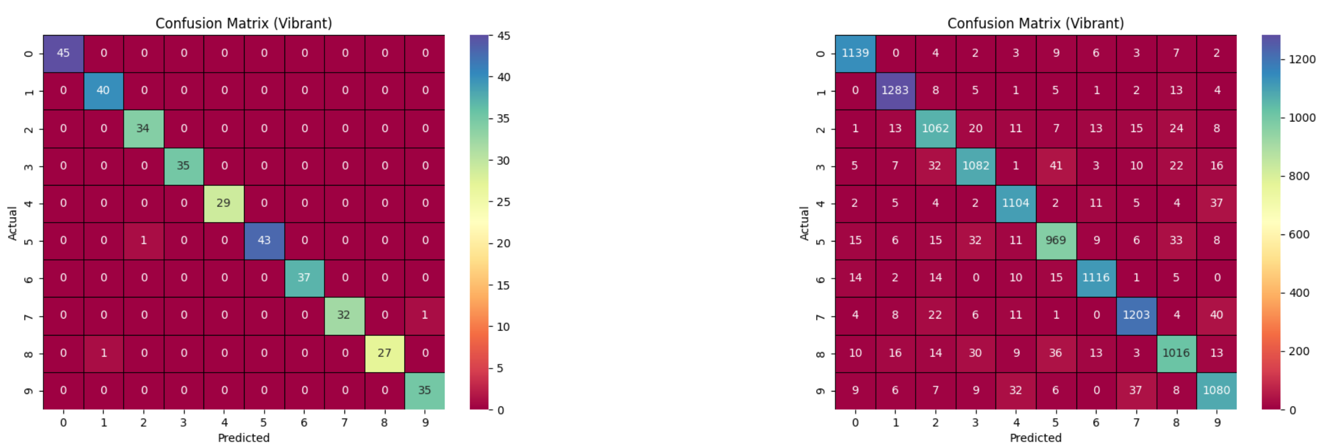


Figure 16: Small dataset: near-perfect classification; Full dataset: more misclassifications due to complexity.

Small data shows near-perfect results; full data reveals errors mainly among digits 2, 3, 5, 8, and 9.

Classification Reports

Test Set Size Impact

- Both use 80/20 splits.
- Small test set: 360 samples.
- Full test set: 10,000 samples.

Larger test sets provide more reliable, robust evaluation, uncovering weaknesses hidden by small samples.

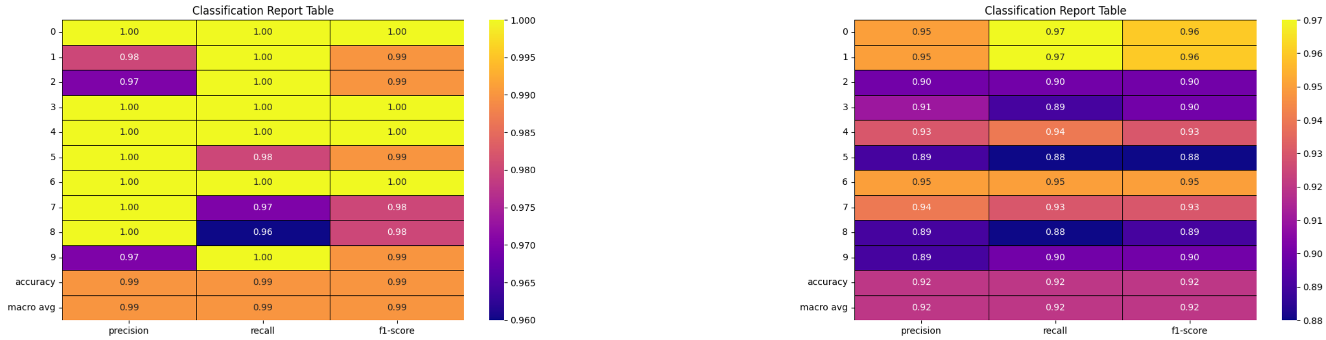


Figure 17: Small dataset: near-perfect F1 scores; Full dataset: average F1 0.92 reflecting greater diversity.

Scaling Effects and Recommendations

The full dataset is 33× larger in train and test size. Scaling improves generalization but requires:

- Stronger regularization.
- Careful hyperparameter tuning.
- Possibly advanced models (e.g., neural networks) for top accuracy.

Conclusion

Small datasets can mask true model weaknesses. Large datasets reveal realistic performance, ensuring better real-world readiness despite minor accuracy drops.

Final Recommendations

- Set `max_iter = 128` to ensure convergence without unnecessary computation.
- Use `solver = 'lbfgs'` for efficient multiclass logistic regression.
- Keep regularization parameter `C = 1.0` for balanced fitting.
- Normalize pixel values by dividing by 9 for best observed accuracy.
- Use `random_state = 44` for consistent train/test splits.
- Maintain an 80/20 train/test split.

This setup achieves a top accuracy of **99.2%** on the `sklearn` dataset and **92.2%** on the MNIST dataset, reflecting excellent model performance.

6 Project Code

This section will contain all the relevant code used threw out the project

6.1 Sklearn Dataset Code

File: sklearn_digits_logreg_tuned.py code for the sklearn data set

```
# tuned_mnist_logistic_regression_parameter.py

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import numpy as np

# very simple version
def run_the_logistic_regression_model():
    # parameter setup for logistic regression
    iterations_max = 128
    number_to_divide_pixels_for_normal = 9.0
    size_of_test_data = 0.2  # 20% testing, 80% training
    random_seed_number = 44  # random seed
    solver = 'lbfgs'  # solver type
    regularization_c = 1.0  # regularization strength

    digits = load_digits()  # load built-in digit data from sklearn
    X = digits.data / number_to_divide_pixels_for_normal  # image data
    y = digits.target  # label data
    all_the_indexs = np.arange(len(X))

    # split into training and testing sets
    X_train, X_test, y_train, y_test, idx_train, idx_test = train_test_split(
        X, y, all_the_indexs, test_size=size_of_test_data, random_state=random_seed_number
    )

    model = LogisticRegression(
        max_iter=max(iterations_max, 128),  # ensure enough iterations
        solver=solver,
        class_weight=None,
        C=regularization_c
    )
    model.fit(X_train, y_train)  # train the model

    y_pred = model.predict(X_test)  # make predictions
    acc = accuracy_score(y_test, y_pred)
    print("Accuracy:", acc)
    print(classification_report(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))

    return acc

if __name__ == "__main__":
    final_accuracy = run_the_logistic_regression_model()
    print(f"Returned accuracy: {final_accuracy}")
```

6.2 MNIST Dataset Code

code for the mnsit datase **File: mnist_logistic_regression.py**

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plot_utils
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.model_selection import train_test_split

# Load both MNIST train and test CSVs
train_data = pd.read_csv("mnist/mnist_train.csv")
test_data = pd.read_csv("mnist/mnist_test.csv")

# Combine datasets
full_data = pd.concat([train_data, test_data], ignore_index=True)


# Separate features and labels, normalize
X = full_data.drop('label', axis=1) / 255.0
y = full_data['label']

# Train-test split 80/20
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train logistic regression
model = LogisticRegression(
    max_iter=1000,
    solver='lbfgs',
    class_weight=None,
    C=1.0
)
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
acc = accuracy_score(y_test, y_pred)
print("Accuracy:", acc)
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))

# Plot confusion matrix and classification report
plot_utils.plot_confusion_matrix(y_test, y_pred, labels=list(range(10)))
plot_utils.plot_classification_report(y_test, y_pred)

# Display one test image with prediction
index = 1
y_test_array = np.array(y_test)
is_correct = y_pred[index] == y_test_array[index]

image_data = X_test.iloc[index].values.reshape(28, 28)

plot_utils.display_prediction_image(
    image=image_data,
    predicted=y_pred[index],
    actual=y_test_array[index],
    accuracy=acc,
    is_correct=is_correct,
    y_test=y_test_array,
    y_pred=y_pred,
    reshape_size=28
)
```

6.3 Utility Functions

File: plot_utils.py

This code allows the main script to display graphical interpretations, including the confusion matrix, the classification report, or a single predicted image.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, classification_report
```

6.3.1 display_prediction_image Function

Listing 2: display_prediction_image function

```
# display a single prediction image with details
def display_prediction_image(image, predicted, actual, accuracy, is_correct, y_test, y_pred,
    reshape_size):
    indices = np.where(y_test == actual)[0]
    total = len(indices)
    correct = np.sum(y_pred[indices] == y_test[indices])
    digit_acc = correct / total if total > 0 else 0

    plt.figure(figsize=(4, 4))
    plt.imshow(image.reshape(reshape_size, reshape_size), cmap='gray')
    result_text = "Correct" if is_correct else "Wrong"
    color = 'green' if is_correct else 'red'
    plt.title(f"\nPredicted: {predicted} | Actual: {actual}\n"
              f"Overall Acc: {accuracy:.2f} | Digit Acc: {digit_acc:.2f} | {result_text}",
              fontsize=10, color=color)
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

6.3.2 plot_classification_report Function

Listing 3: plot_classification_report function

```
def plot_classification_report(y_true, y_pred):
    report_dict = classification_report(y_true, y_pred, output_dict=True)
    df = pd.DataFrame(report_dict).transpose().round(2)
    plt.figure(figsize=(10, 6))
    sns.heatmap(df.iloc[:-1, :-1], annot=True, fmt='.2f', cmap='plasma',
                cbar=True, linewidths=0.5, linecolor='black')
    plt.title('Classification Report Table')
    plt.yticks(rotation=0)
    plt.show()
```

6.3.3 plot_confusion_matrix Function

Listing 4: plot_confusion_matrix function

```
# plot confusion matrix as heatmap
def plot_confusion_matrix(y_true, y_pred, labels):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Spectral',
                cbar=True, linewidths=0.5, linecolor='black',
                xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix (Vibrant)')
    plt.show()
```

References

[1] Scikit-learn documentation,
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html

[2] MNIST dataset on Kaggle,
https://www.kaggle.com/datasets/oddrationale/mnist-in-csv

[3] Scikit-learn LogisticRegression documentation,
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[4] GeeksforGeeks, *How to Optimize Logistic Regression Performance*,
https://www.geeksforgeeks.org/how-to-optimize-logistic-regression-performance/?utm_source=chatgpt.com

[5] Jason Brownlee, *Hyperparameter Optimization With Random Search and Grid Search*, MachineLearningMastery.com, 2020.
https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/

[6] Scikit-learn GridSearchCV documentation,
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[7] Scikit-learn RandomizedSearchCV documentation,
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html