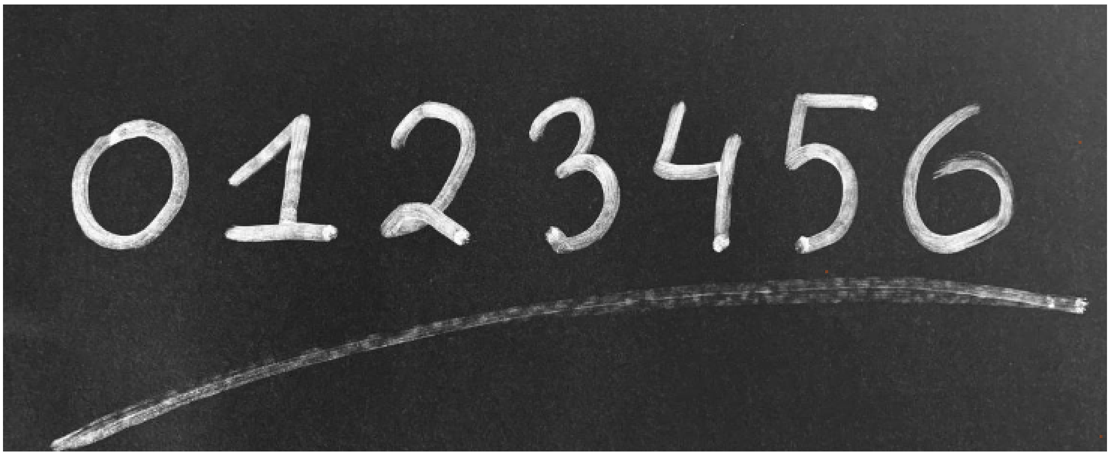
Digit Recognizer :

Decoding Handwritten Digits with Machine Learning



By: Praveen Choudhary

Project Description:

The Digit Recognizer project aims to develop a machine learning model that can accurately recognize handwritten digits. Handwritten digit recognition is a fundamental problem in the field of computer vision and pattern recognition. The project utilizes a dataset of handwritten digits, where each digit is represented as an image.

Objective

The primary objective of the Digit Recognizer project is to develop a robust machine learning model capable of accurately recognizing handwritten digits. Through data exploration, preprocessing, and model development, the project aims to achieve a high level of accuracy in digit classification. Additionally, the project seeks to provide clear documentation and visualizations to enhance understanding and showcase the model's effectiveness in practical applications

In [189…

```
# Import some library
from sklearn.datasets import load_digits
import pandas as pd
import matplotlib.pyplot as plt
```

In [190…

```
# Load Data
dataset = load_digits()

# Keys feature in datasets
dataset.keys()
```

Out[190…

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

In [191…

```
print('Size of DataSet is :',dataset.data.shape)
```

```
Size of DataSet is : (1797, 64)
```

In [192…

```
# Display the data variable
dataset.data[0]
```

Out[192…

```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
       12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
        0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
       10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

In [193…

```
# Display the target variable
dataset.target
```

Out[193…

```
array([0, 1, 2, ..., 8, 9, 8])
```

Transformation

Create an 8x8 array to enhance the visualization of a single image since the total values in the data for each image amount to 64. This transformation facilitates a clearer representation of the image structure in an 8 by 8 grid.

In [194…

```
# Make a array of 8X8 because total value in data for single image is 64 so we convert in into 8 by 8. for clear visulization
dataset.data[0].reshape(8,8)
```

Out[194…

```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

Visualization

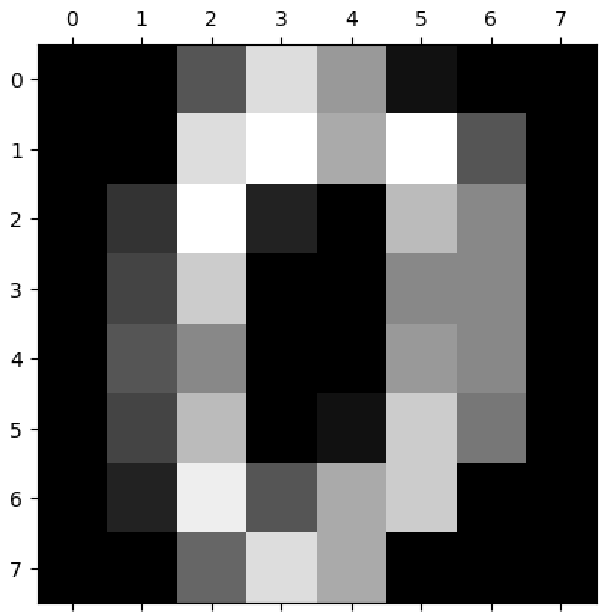
In this visual exploration of our dataset, where each image represents a unique handwritten digit. This collection of digit images provides an insightful glimpse into the diversity and characteristics of handwritten numerical symbols. Join us on this visual journey as we showcase the beauty and variability encapsulated in the world of handwritten digits.

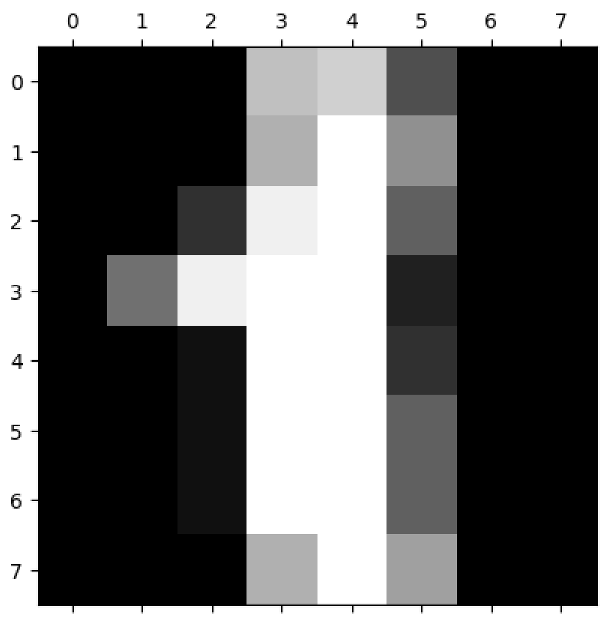
In [195…

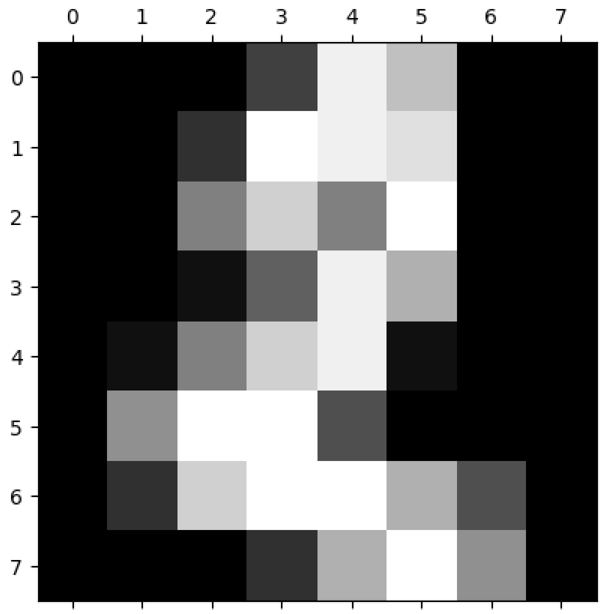
```
from matplotlib import pyplot as plt
%matplotlib inline
```

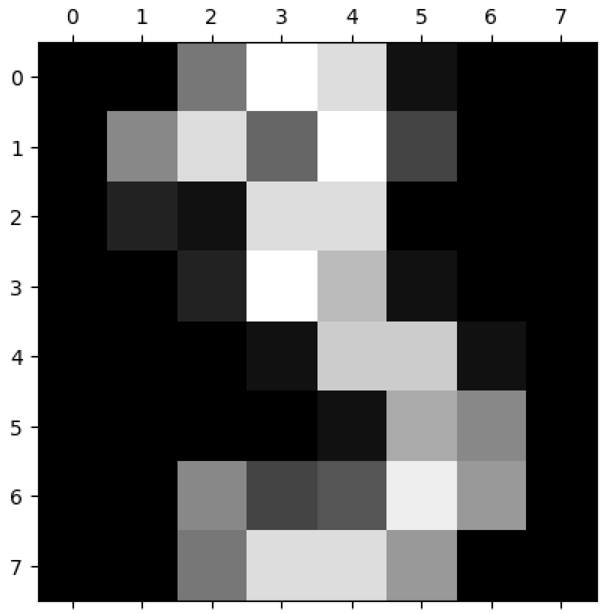
In [221…

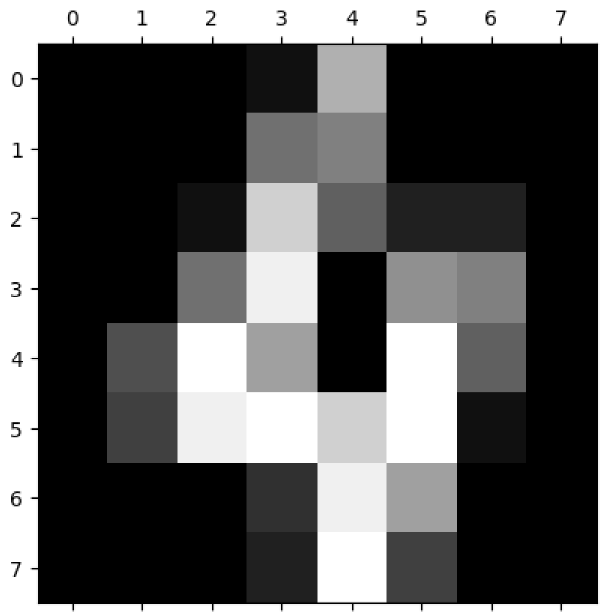
```
# Display the first 10 images of handwritten digits
for i in range(10):
    dis=dataset.data[i].reshape(8,8)
    plt.matshow(dis)
plt.show()
```

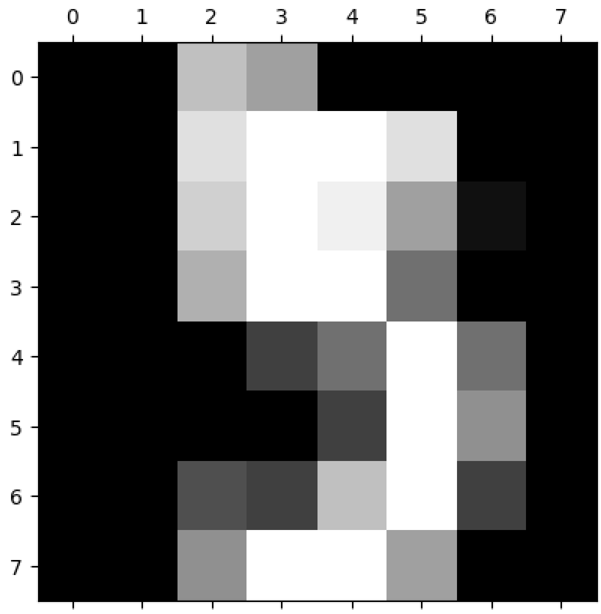


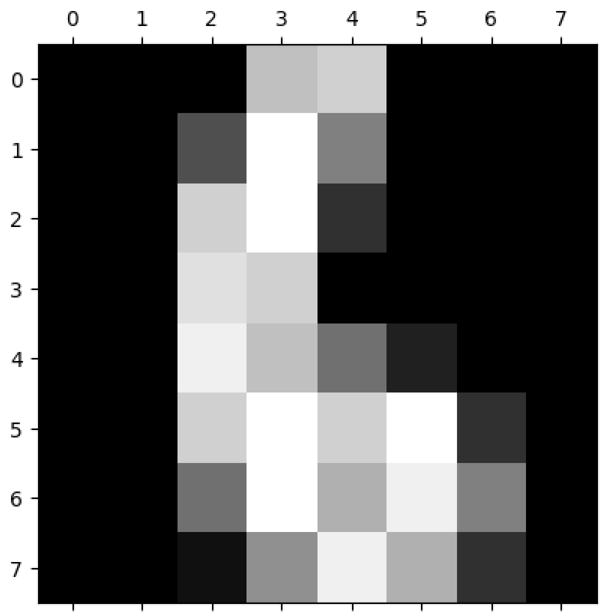


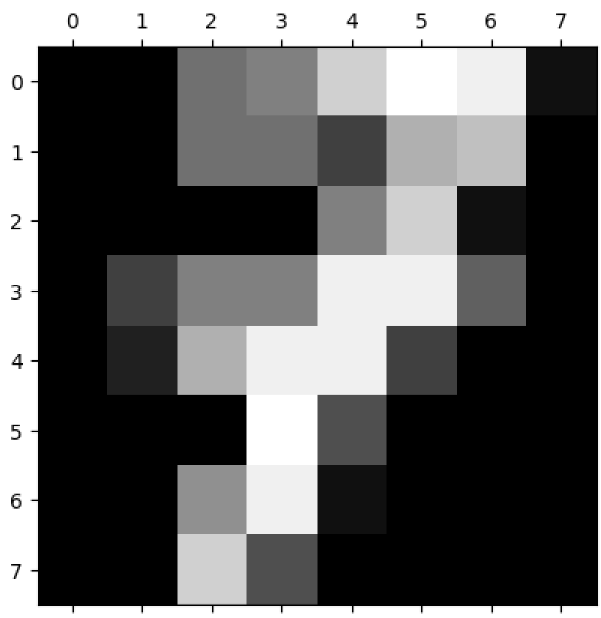

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7

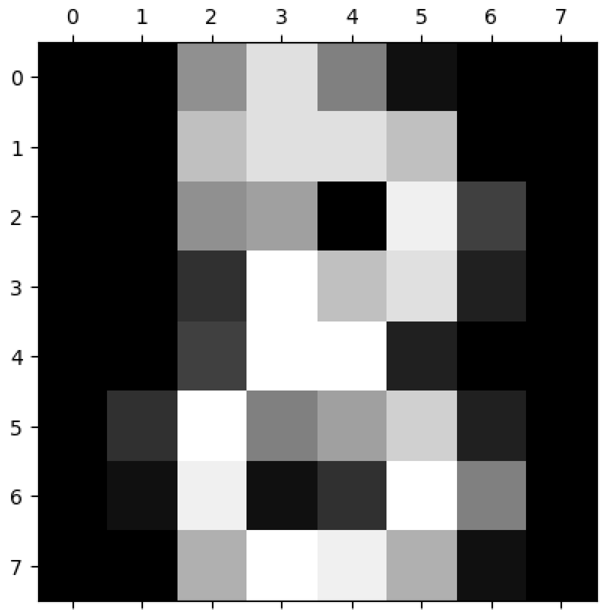

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7

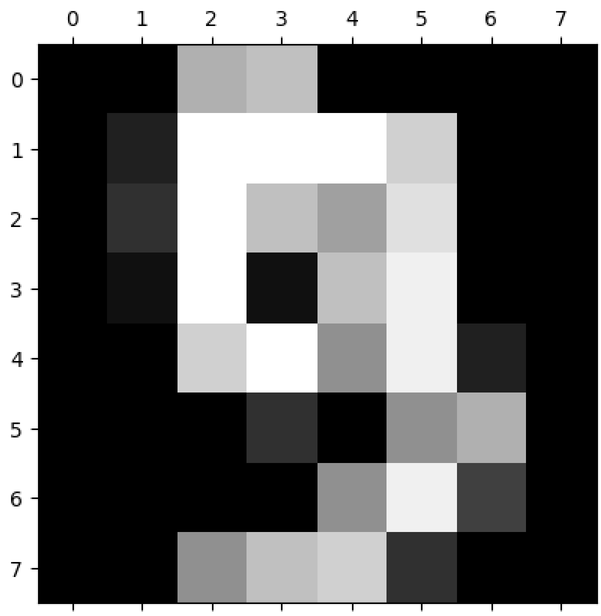

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7


0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7


0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7


0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7





In [197…
```
# Create the DataFrame for the Input Variable :
df = pd.DataFrame(dataset.data, columns=dataset.feature_names)
```

In [198…
```
# Top 5 Rows
df.head(5)
```

Out[198…

	pixel_0_0	pixel_0_1	pixel_0_2	pixel_0_3	pixel_0_4	pixel_0_5	pixel_0_6	pixel_0_7	pixel_1_0	pixel_1_1	...	pixel_6_6	pixel_6_7	pixel_7_0
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	0.0	8.0	...	9.0	0.0	0.0
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0

5 rows × 64 columns

In [199…
```
# Top 5 Cloumn
df.tail(5)
```

Out[199…

	pixel_0_0	pixel_0_1	pixel_0_2	pixel_0_3	pixel_0_4	pixel_0_5	pixel_0_6	pixel_0_7	pixel_1_0	pixel_1_1	...	pixel_6_6	pixel_6_7	pixel_7_
1792	0.0	0.0	4.0	10.0	13.0	6.0	0.0	0.0	0.0	1.0	...	4.0	0.0	0
1793	0.0	0.0	6.0	16.0	13.0	11.0	1.0	0.0	0.0	0.0	...	1.0	0.0	0
1794	0.0	0.0	1.0	11.0	15.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0
1795	0.0	0.0	2.0	10.0	7.0	0.0	0.0	0.0	0.0	0.0	...	2.0	0.0	0
1796	0.0	0.0	10.0	14.0	8.0	1.0	0.0	0.0	0.0	2.0	...	8.0	0.0	0

5 rows × 64 columns

Descriptive Statistics

Explore the descriptive statistics of the dataframe containing image pixel values. Gain insights into the central tendencies, variations, and distributions of pixel intensities across the dataset. This analysis provides a comprehensive overview of the numerical features, offering a deeper understanding of the image data's characteristics.

In [200…

```
df.describe()
```

Out[200…

	pixel_0_0	pixel_0_1	pixel_0_2	pixel_0_3	pixel_0_4	pixel_0_5	pixel_0_6	pixel_0_7	pixel_1_0	pixel_1_1	...	
count	1797.0	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000	1797.000000	...	179
mean	0.0	0.303840	5.204786	11.835838	11.848080	5.781859	1.362270	0.129661	0.005565	1.993879	...	
std	0.0	0.907192	4.754826	4.248842	4.287388	5.666418	3.325775	1.037383	0.094222	3.196160	...	
min	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	
25%	0.0	0.000000	1.000000	10.000000	10.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	
50%	0.0	0.000000	4.000000	13.000000	13.000000	4.000000	0.000000	0.000000	0.000000	0.000000	...	
75%	0.0	0.000000	9.000000	15.000000	15.000000	11.000000	0.000000	0.000000	0.000000	3.000000	...	
max	0.0	8.000000	16.000000	16.000000	16.000000	16.000000	16.000000	15.000000	2.000000	16.000000	...	1

8 rows × 64 columns

In [201…

```
# Information About The Data Frame
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1797 entries, 0 to 1796
Data columns (total 64 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   pixel_0_0  1797 non-null   float64
 1   pixel_0_1  1797 non-null   float64
 2   pixel_0_2  1797 non-null   float64
 3   pixel_0_3  1797 non-null   float64
 4   pixel_0_4  1797 non-null   float64
 5   pixel_0_5  1797 non-null   float64
 6   pixel_0_6  1797 non-null   float64
 7   pixel_0_7  1797 non-null   float64
 8   pixel_1_0  1797 non-null   float64
 9   pixel_1_1  1797 non-null   float64
 10  pixel_1_2  1797 non-null   float64
 11  pixel_1_3  1797 non-null   float64
 12  pixel_1_4  1797 non-null   float64
 13  pixel_1_5  1797 non-null   float64
 14  pixel_1_6  1797 non-null   float64
 15  pixel_1_7  1797 non-null   float64
 16  pixel_2_0  1797 non-null   float64
 17  pixel_2_1  1797 non-null   float64
 18  pixel_2_2  1797 non-null   float64
 19  pixel_2_3  1797 non-null   float64
 20  pixel_2_4  1797 non-null   float64
 21  pixel_2_5  1797 non-null   float64
 22  pixel_2_6  1797 non-null   float64
 23  pixel_2_7  1797 non-null   float64
 24  pixel_3_0  1797 non-null   float64
 25  pixel_3_1  1797 non-null   float64
 26  pixel_3_2  1797 non-null   float64
 27  pixel_3_3  1797 non-null   float64
 28  pixel_3_4  1797 non-null   float64
 29  pixel_3_5  1797 non-null   float64
 30  pixel_3_6  1797 non-null   float64
 31  pixel_3_7  1797 non-null   float64
 32  pixel_4_0  1797 non-null   float64
 33  pixel_4_1  1797 non-null   float64
 34  pixel_4_2  1797 non-null   float64
 35  pixel_4_3  1797 non-null   float64
 36  pixel_4_4  1797 non-null   float64
 37  pixel_4_5  1797 non-null   float64
 38  pixel_4_6  1797 non-null   float64
 39  pixel_4_7  1797 non-null   float64
 40  pixel_5_0  1797 non-null   float64
 41  pixel_5_1  1797 non-null   float64
 42  pixel_5_2  1797 non-null   float64
 43  pixel_5_3  1797 non-null   float64
 44  pixel_5_4  1797 non-null   float64
 45  pixel_5_5  1797 non-null   float64
 46  pixel_5_6  1797 non-null   float64
 47  pixel_5_7  1797 non-null   float64
 48  pixel_6_0  1797 non-null   float64
 49  pixel_6_1  1797 non-null   float64
 50  pixel_6_2  1797 non-null   float64
 51  pixel_6_3  1797 non-null   float64
 52  pixel_6_4  1797 non-null   float64
 53  pixel_6_5  1797 non-null   float64
 54  pixel_6_6  1797 non-null   float64
 55  pixel_6_7  1797 non-null   float64
 56  pixel_7_0  1797 non-null   float64
 57  pixel_7_1  1797 non-null   float64
 58  pixel_7_2  1797 non-null   float64
 59  pixel_7_3  1797 non-null   float64
 60  pixel_7_4  1797 non-null   float64
 61  pixel_7_5  1797 non-null   float64
 62  pixel_7_6  1797 non-null   float64
 63  pixel_7_7  1797 non-null   float64
dtypes: float64(64)
memory usage: 898.6 KB
```

In [202…

```python
# Total number of Rows in dataset
len(df)
```

Out[202…

```
1797
```

Training and Testing Sets For Machine Learning

Split the data into two parts: independent variables, used as input values, and dependent variables, providing our predicted values.

In [203…

```python
X=df # Independent
```

In [204…
```
y=dataset['target'] # Dependent
```

Standardization

In the process of preparing our data for digit recognition, we employ the StandardScaler from scikit-learn for a crucial step known as standardization. This technique is applied to ensure that the pixel values of our images are on a consistent scale throughout the dataset.

In [205…
```
from sklearn.preprocessing import StandardScaler

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

Out[205…
```
array([[ 0.        , -0.33501649, -0.04308102, ..., -1.14664746,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -1.09493684, ...,  0.54856067,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -1.09493684, ...,  1.56568555,
         1.6951369 , -0.19600752],
       ...,
       [ 0.        , -0.33501649, -0.88456568, ..., -0.12952258,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -0.67419451, ...,  0.8876023 ,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649,  1.00877481, ...,  0.8876023 ,
        -0.26113572, -0.19600752]])
```

In [206…
```
# Split into Testing and Training Set
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=30)
```

In [207…
```
X_train
```

Out[207…
```
array([[ 0.        ,  1.87020193,  0.79840364, ...,  0.20951905,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649, -0.88456568, ...,  1.56568555,
         3.40687545,  4.10598346],
       [ 0.        , -0.33501649,  0.79840364, ...,  1.56568555,
         3.40687545,  1.4172391 ],
       ...,
       [ 0.        , -0.33501649, -0.46382335, ...,  1.56568555,
         1.6951369 , -0.19600752],
       [ 0.        , -0.33501649,  0.16729015, ..., -0.46856421,
        -0.5056698 , -0.19600752],
       [ 0.        , -0.33501649,  1.21914597, ..., -0.80760583,
        -0.5056698 , -0.19600752]])
```

In [208…
```
y_train
```

Out[208…
```
array([5, 3, 2, ..., 8, 0, 5])
```

Model Execution

LogisticsRegression: I employed this classification algorithm to predict the likelihood of each digit's presence based on the pixel values. Logistic Regression, known for its simplicity and interpretability, provided valuable insights into the relationships between features and the digit classes.

In [209…
```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
```

In [210…
```
# Fitting the model
model.fit(X_train, y_train)
```

Out[210…
```
▾ LogisticRegression
LogisticRegression()
```

In [211…
```
model.score(X_test, y_test)
```

Out[211…
```
0.9722222222222222
```

In [214…
```
from sklearn.metrics import mean_squared_error

#Predicted On The Test Set
y_pred= model.predict(X_test)

# Evaluate The Model
```

```
Training_score  = model.score(X_train,y_train)
Testing_score  = model.score(X_test,y_test)

mse = mean_squared_error(y_test,y_pred)
rmse = np.sqrt(mse)

print('Training Score :',Training_score,'%')
print('Testing Score :',Testing_score,'%')
print('Mean Squared Error (MSE) :',mse)
print('Root Mean Squared Error (MSE) :',rmse)
```

```
Training Score : 0.9993041057759221 %
Testing Score : 0.9722222222222222 %
Mean Squared Error (MSE) : 0.85
Root Mean Squared Error (MSE) : 0.9219544457292888
```

Result :

The model evaluation results indicate high performance on the training set, with a training score of 99.93%. However, on the testing set, the accuracy slightly decreases to 97.22%, suggesting the model generalizes well to unseen data.

The Mean Squared Error (MSE) is calculated to be 0.85, representing the average squared difference between the predicted and actual values. The Root Mean Squared Error (RMSE) is 0.92, providing an interpretation of the average magnitude of these differences.

These results collectively suggest that while the model performs exceptionally well on the training data, there is a slight drop in accuracy on the testing data, and the MSE and RMSE metrics offer insights into the model's predictive accuracy and the magnitude of prediction errors. Further analysis and fine-tuning may be considered to improve generalization performance on unseen data.

Confusion Matrix :

A confusion matrix is a visual representation that helps evaluate the performance of a classification model. It provides a detailed breakdown of the model's predictions, comparing them with the actual ground truth. The confusion matrix is particularly useful for understanding where the model excels and where it makes errors.

In [215...

```
# confustion matrix
from sklearn.metrics import confusion_matrix
```

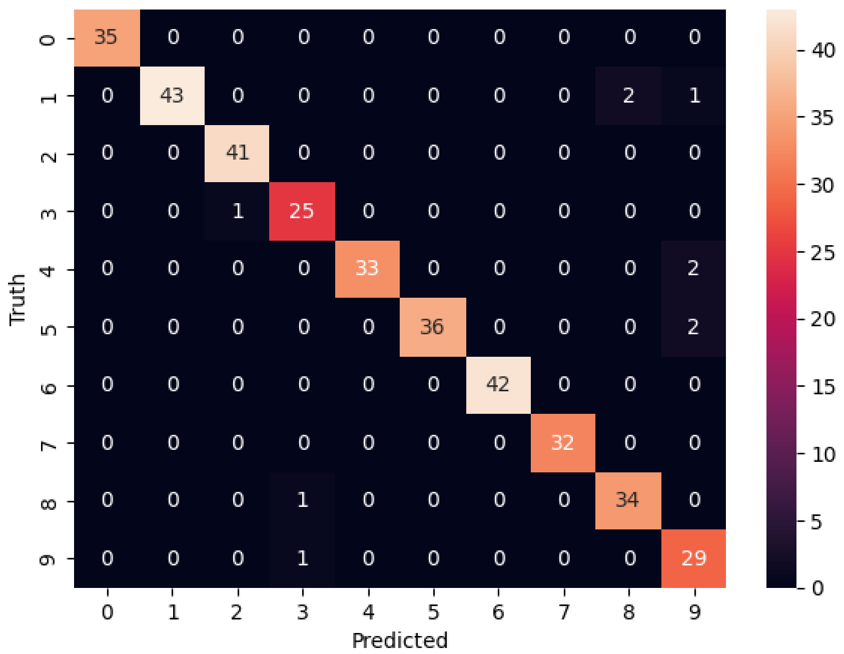
In [120...

```
cm = confusion_matrix(y_test,y_pred)
```

In [127...

```
# confusion metric visual
import seaborn as sn
plt.figure(figsize=(7,5))

sn.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
plt.show()
```



- digit recognition model is performing well on most numbers, but it encounters more errors, particularly with digit 1. The confusion matrix and heatmap highlight misclassifications, with instances where the predicted digit differs from the true digit.

- To address the higher error rate for digit 1, further investigation and potential model refinement may be necessary. Strategies could include collecting more training data for digit 1

Conclusion :

In summary, the digit recognition model exhibits outstanding performance on the training data, achieving an impressive 99.93% accuracy. However, when evaluated on the testing set, the accuracy slightly decreases to 97.22%, indicating a robust but not perfect generalization to unseen data. The calculated Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) further shed light on prediction accuracy and error magnitudes.

Despite the model's overall proficiency, it faces challenges, particularly with digit 1, as highlighted by the confusion matrix and heatmap. The higher error rate for this specific digit suggests a need for further investigation and model refinement. Possible strategies include augmenting the training dataset with more examples of digit 1 or exploring adjustments to hyperparameters to enhance the model's capability to accurately recognize this particular digit. Continued analysis and fine-tuning are essential to achieving optimal generalization and improving the model's performance on diverse digit representations

Thank You