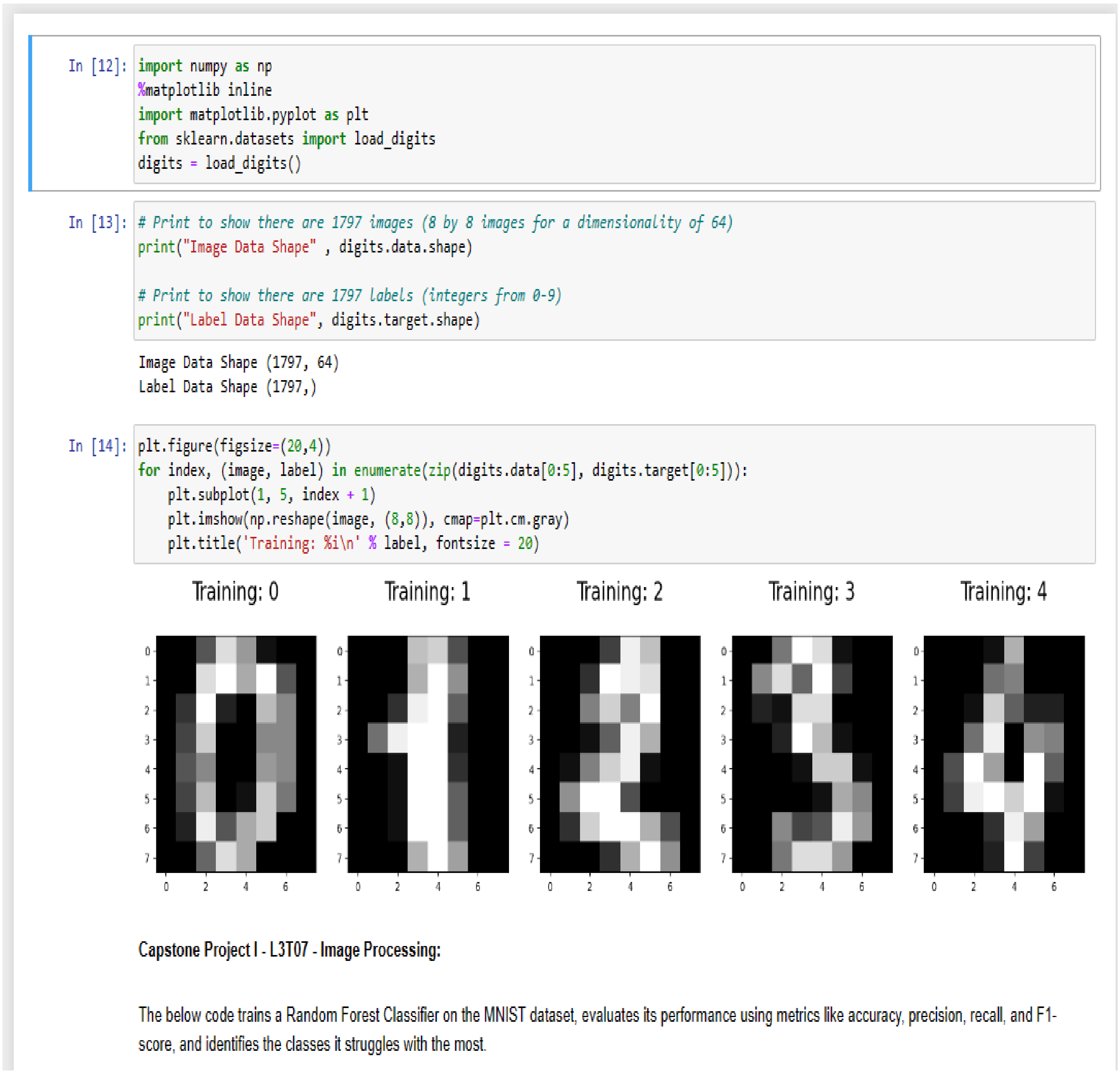
L3T07 – Capstone – MNIST Task – Jupyter Notebook snippet



In [12]:
```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
digits = load_digits()
```

In [13]:
```
# Print to show there are 1797 images (8 by 8 images for a dimensionality of 64)
print("Image Data Shape" , digits.data.shape)

# Print to show there are 1797 labels (integers from 0-9)
print("Label Data Shape", digits.target.shape)
```

```
Image Data Shape (1797, 64)
Label Data Shape (1797,)
```

In [14]:
```
plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(digits.data[0:5], digits.target[0:5])):
    plt.subplot(1, 5, index + 1)
    plt.imshow(np.reshape(image, (8,8)), cmap=plt.cm.gray)
    plt.title('Training: %i\n' % label, fontsize = 20)
```

Capstone Project I - L3T07 - Image Processing:

The below code trains a Random Forest Classifier on the MNIST dataset, evaluates its performance using metrics like accuracy, precision, recall, and F1-score, and identifies the classes it struggles with the most.

In [15]:
```
# Importing necessary libraries (for handling data, plotting, and machine learning tasks)
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
```

In [16]:
```
# Load the MNIST dataset
digits = load_digits()
```

In [17]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)
```

- The purpose of the train and test sets is to evaluate the model's performance.
- The model is trained on the training set and evaluated on the test set.
- The data is split into training and testing sets:
- The train_test_split() function is used to split the dataset into a training set (80% of the data)
- and a test set (20% of the data).

In [18]:
```
# Create a Random Forest Classifier

# Chose 'n_estimators' as the parameter to tune, which is the number of trees in the forest.
# Increasing the number of trees can increase the model's performance, but it may also increase computation time.

n_estimators = 100
clf = RandomForestClassifier(n_estimators=n_estimators, random_state=42)
clf.fit(X_train, y_train)
```

Out[18]:
```
RandomForestClassifier(random_state=42)
```

In [19]:
```
# Predict on the test data
y_pred = clf.predict(X_test)
```

In the above, The trained classifier is used to predict the class labels for the test data using the predict() function.

In [20]:
```
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
```

```
Confusion Matrix:
 [[32  0  0  0  1  0  0  0  0  0]
 [ 0 28  0  0  0  0  0  0  0  0]
 [ 0  0 33  0  0  0  0  0  0  0]
 [ 0  0  0 32  0  1  0  0  1  0]
 [ 0  0  0  0 46  0  0  0  0  0]
 [ 0  0  0  0  0 45  1  0  0  1]
 [ 0  0  0  0  0  1 34  0  0  0]
 [ 0  0  0  0  0  0  0 33  0  1]
 [ 0  1  0  0  0  0  0  0 29  0]
 [ 0  0  0  0  0  1  0  1  0 38]]
```

The confusion_matrix() function is used to compute the confusion matrix for the classifier's predictions on the test data. The confusion matrix helps in understanding the performance of the classifier on each class and where the classifier is making mistakes.

In [24]:
```
# Report which classes the model struggles with the most

# (Added a check to see if there are any elements in the result of np.argwhere
# (before trying to index it to avoid getting an IndexError).

class_difficulties = np.argwhere(cm - np.diag(np.diag(cm)) == cm.max())
if len(class_difficulties) > 0:
    class_difficulty = class_difficulties[0]
    print(f"The model struggles the most with class {class_difficulty[0]} being predicted as class {class_difficulty[1]}")
else:
    print("No class difficulties found.")
```

```
No class difficulties found.
```

In [22]:
```
# Calculate accuracy, precision, recall, and f1-score
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="macro")
recall = recall_score(y_test, y_pred, average="macro")
f1 = f1_score(y_test, y_pred, average="macro")
```

The average="macro" parameter is used above to compute the average precision, recall, and F1-score over all classes.

In [23]:
```
# Print / Report the results
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

```
Accuracy: 0.9722222222222222
Precision: 0.9740424119023985
Recall: 0.9727003722185199
F1-score: 0.9732067700933176
```