CS 440: Intro to Artificial Intelligence
Final Report

Dylan Farrell, Awad Shawl
199002491, 200003451

May 1st, 2024

1 Algorithm Descriptions

Perceptron

The perceptron is among the earliest algorithms used for supervised learning and functions as a linear classifier. This algorithm establishes a dividing line that optimally separates different labeled data categories for classification purposes.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

In this context, w represents the weight vector and x is the input vector. The term b denotes the bias, which adjusts the decision boundary, shifting it from the origin.

Implementation Details

Our Perceptron class is initialized with several parameters:

- **learning_rate**: This parameter controls the step size during the learning process, helping to converge to a minimum of the loss function.
- **n_iterations**: Defines the number of times the algorithm will iterate over the entire dataset, allowing for the refinement of weights with each pass.
- **n_classes**: Specifies the number of distinct output classes. If greater than 2, the Perceptron is configured for multiclass classification.

Training Process

During training:

1. Training data is augmented with a bias term to handle the bias weight.
2. Weights are initialized as zero vectors (or matrices, in the case of multiclass classification).
3. For each iteration and each example:
   - Compute the dot product of weights and inputs to make predictions.
   - In multiclass settings, use a one-vs-all strategy by updating the weights of the predicted and actual classes differently.
   - For binary classification, update weights based on the error between the predicted and actual labels.
4. Validation is performed at the end of each iteration to monitor performance and possibly update the model with the best observed weights.

Prediction Process

For making predictions:

- Input features are also augmented with a bias term.
- The trained weights are used to compute the class scores.
- The final prediction is determined by the sign of the dot product (in binary classification) or by selecting the class with the highest score (in multiclass classification).

Time Complexity

The implementation includes timing the training process, providing insights into the computational cost associated with the training for different classification tasks (digit vs. face recognition).

Neural Network

Our neural network takes in the input layer of 784 or 4,200 depending on if you are inputting the digit or face. We put in 128 nodes in our hidden layer for both digit and face as we found that to be a sweet spot. It starts with **forward propagation**, where an input layer is a flattened array of 0's and 1's. Here is the math behind forward propagation:

$$Z[1] = w[1]A[0] + b[1]$$

where $A[0]$ is the input layer and $Z[1]$ is the unactivated first layer. After this, we apply an activation function, in which we chose ReLU, which stands for rectified linear unit.

$$A[1] = \text{ReLU}(Z[1])$$

$$Z[2] = w[2]A[1] + b[2]$$

$$A[2] = \text{softmax}(Z[2])$$

Softmax converts the output logits into probabilities by applying the function:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

To prevent overfitting, we incorporate L2 regularization, which penalizes the magnitude of the weights, into our loss function. This is done by adding a regularization term to the loss, calculated as $\lambda \sum_i w_i^2$, where λ is the regularization strength. In our case, we used 0.001 for digits and 0.01 for faces as we found it fit best.
Now we perform **backward propagation** to better adjust weights and biases.

$$DZ[2] = A[2] - Y$$

Where Y is the one-hot encoded output and DZ[2] represents the error at the output layer.

$$DW[2] = \frac{1}{m} DZ[2]A[1]^T$$

$$DB[2] = \frac{1}{m} \sum DZ[2]$$

DW[2] and DB[2] These are the gradients of the loss function with respect to weights and biases at the second layer.

$$DZ[1] = w[2]^T DZ[2] * g'(Z[1])$$

DZ[1]: The gradient of the loss.

$$DW[1] = \frac{1}{m} DZ[1]x^T$$

$$DB[1] = \frac{1}{m} \sum DZ[1]$$

DW[1] and DB[1] are the gradients for the first layer.
Update Step:

$$w[1] = w[1] - \alpha DW[1]$$

$$b[1] = b[1] - \alpha DB[1]$$

$$w[2] = w[2] - \alpha DW[2]$$

$$b[2] = b[2] - \alpha DB[2]$$

Where α is the learning rate

2 Features

We extracted the data by extracting all pixels, 28x28 equals 784 for digits and 60x70 equals 4,200 for faces, and counted as a 1 if the character was a # or a +. We then flattened the array and used that as our input layer. For digits, we found that adding the amount of pixels in the 4 different quadrants gave positive feedback regarding perceptron and the neural network only for digits, and negatively affected faces. So we only applied this feature to digits.

3 Evaluation

Training Perceptrons:
Digit

Training Percentage	Time(s)	Accuracy(%)
10	4.073	74.0
20	6.276	75.0
30	9.710	76.8
40	10.831	79.1
50	13.100	80.6
60	15.210	82.2
70	17.696	82.6
80	19.342	81.9
90	21.708	82.1
100	24.291	82.3

Face

Training Percentage	Time(s)	Accuracy(%)
10	2.451	72.7
20	2.927	78.8
30	2.991	82.0
40	3.230	83.3
50	3.543	85.3
60	3.855	86.7
70	3.671	84.7
80	3.698	86.7
90	4.095	85.3
100	4.365	88.0

Training Neural Networks:

Digit

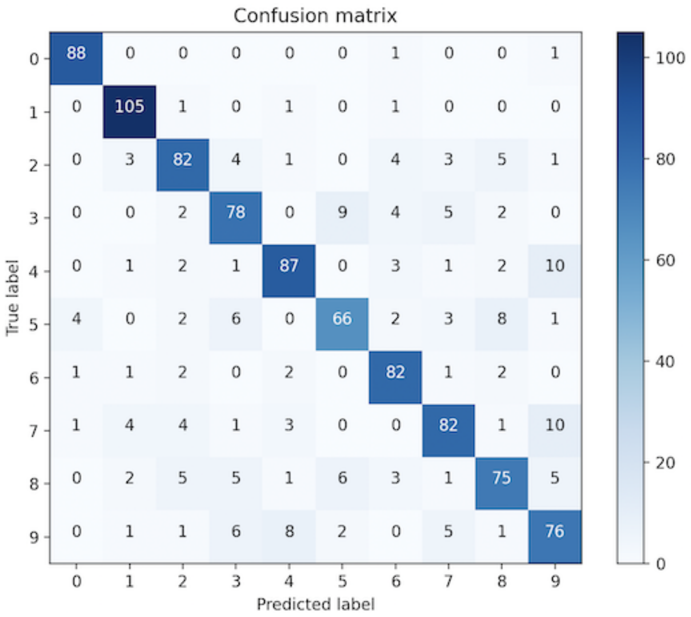
Training Percentage	Time(s)	Accuracy(%)
10	4.131	70.2
20	10.478	79.3
30	10.923	82
40	15.964	83
50	20.479	84.1
60	24.770	85.1
70	28.031	85.8
80	32.536	86.0
90	37.010	86.2
100	42.408	86.6

Face

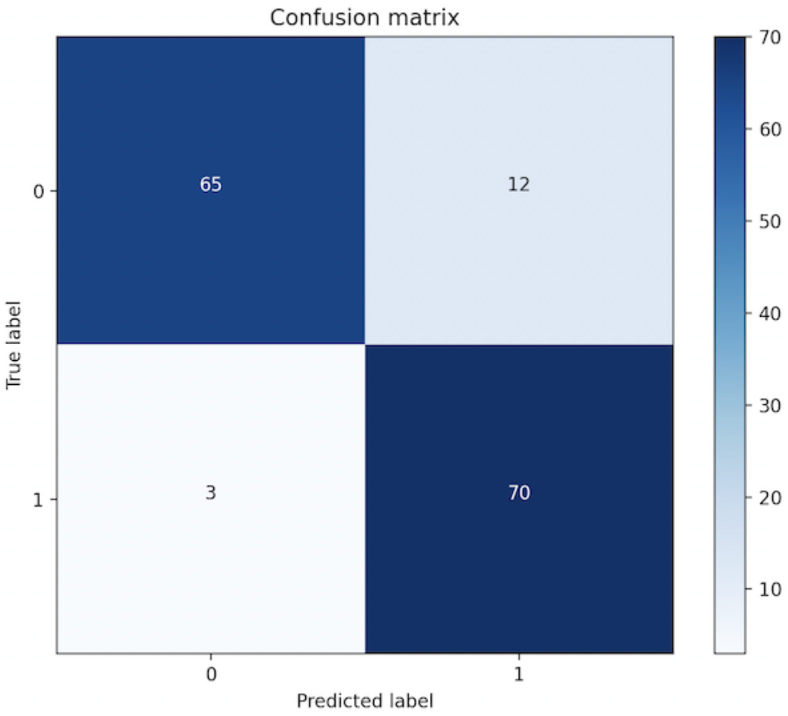
Training Percentage	Time(s)	Accuracy(%)
10	7.106	61.3
20	9.446	81.3
30	11.388	85.3
40	15.115	88.7
50	14.900	88.7
60	17.785	90.0
70	20.669	88.0
80	21.270	92.0
90	24.359	92.7
100	26.019	91.3

4 Performance of each algorithm

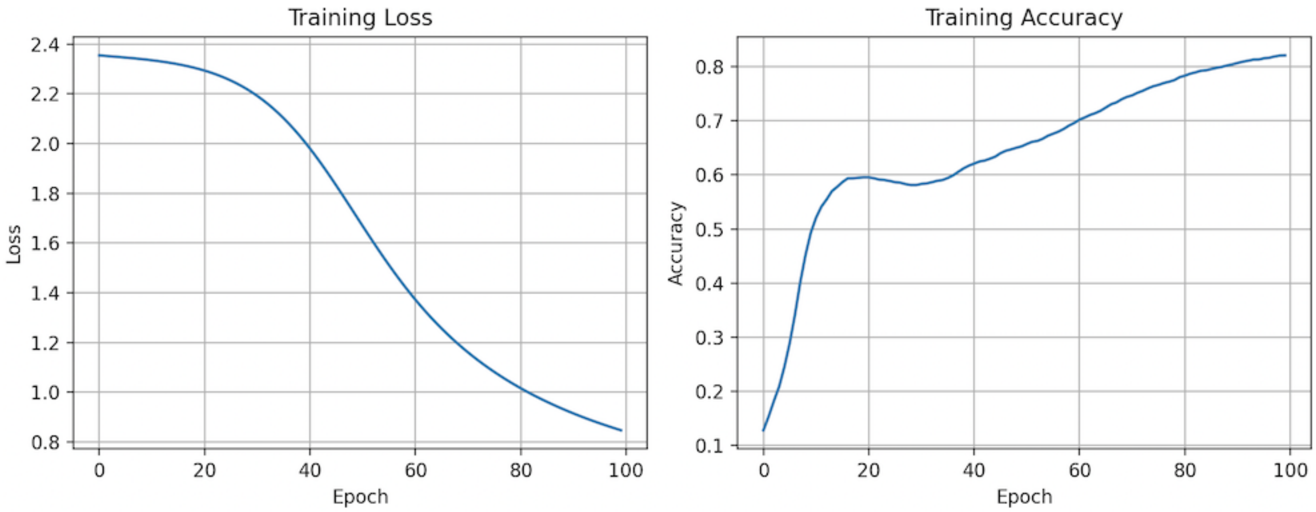
Confusion Matrix of Digits:



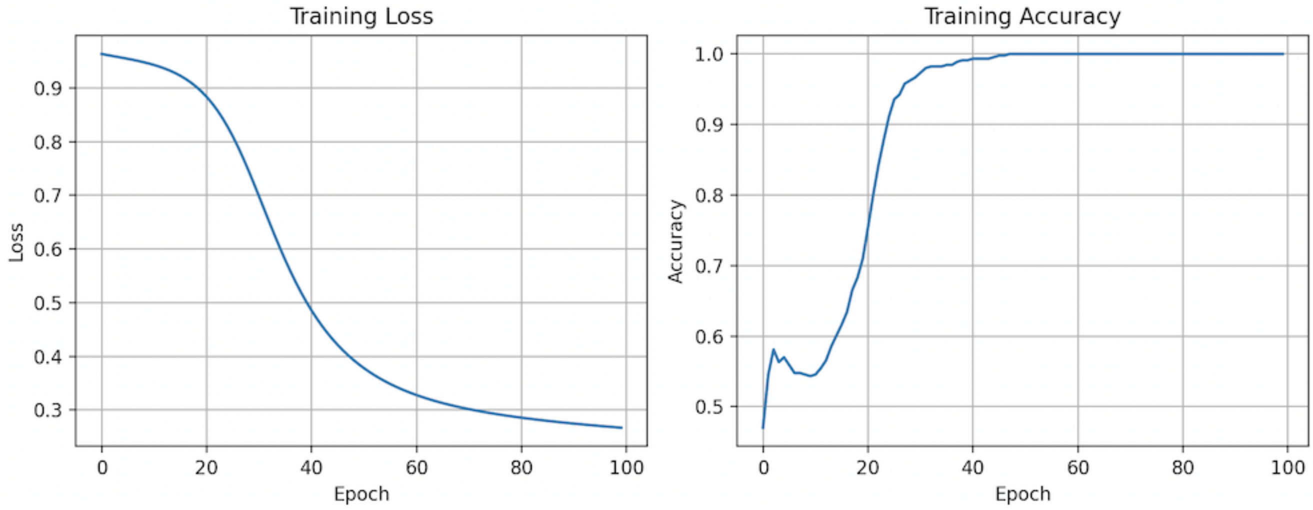
Confusion Matrix of Faces:



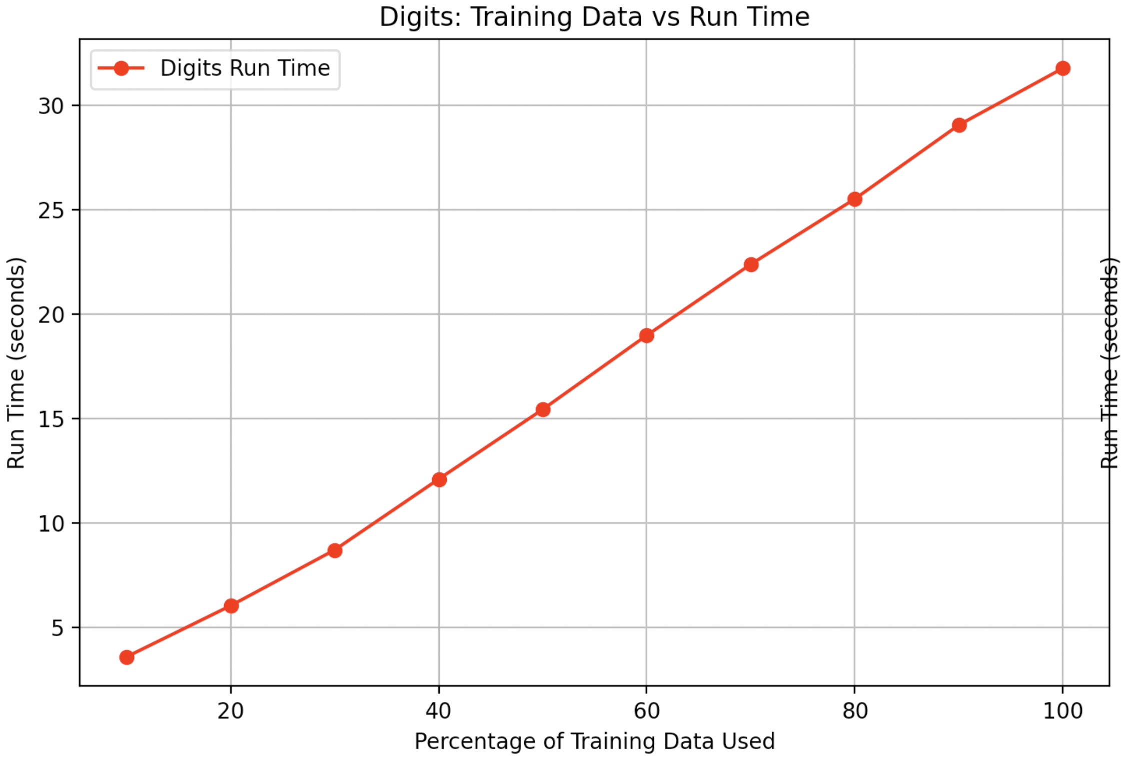
Training Loss and Accuracy for Digit Neural Network over Epochs:



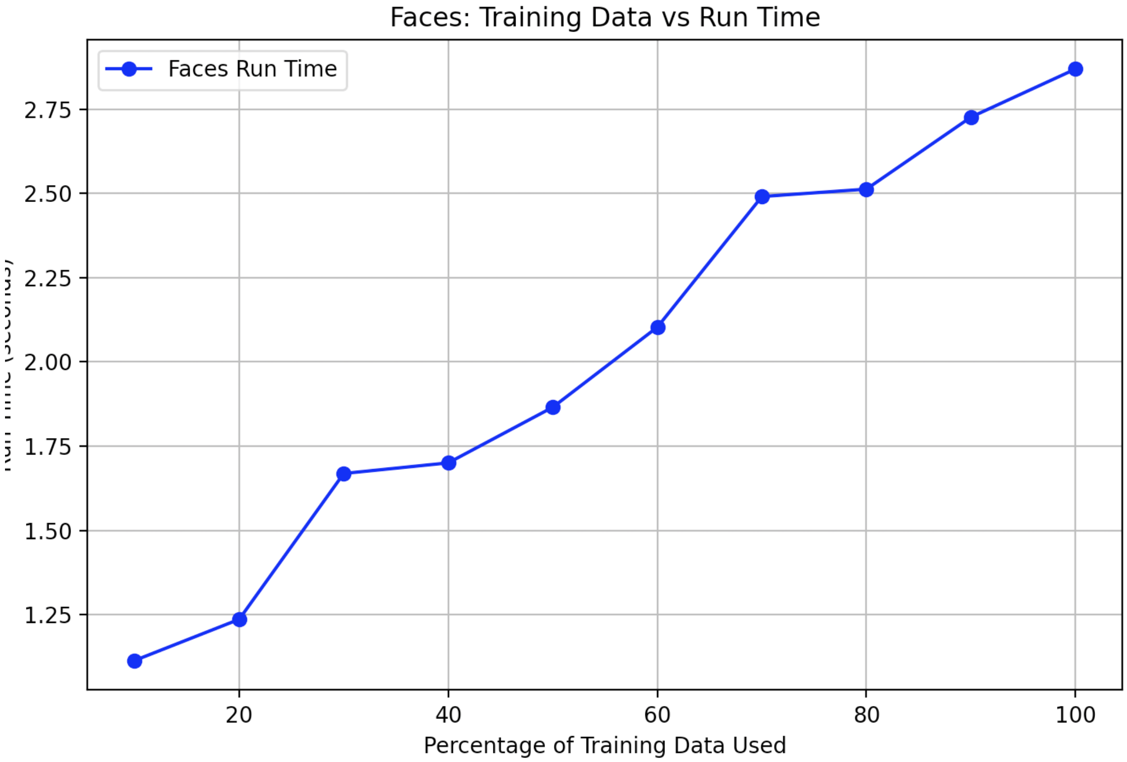
Training Loss and Accuracy for Faces Neural Network over Epochs:



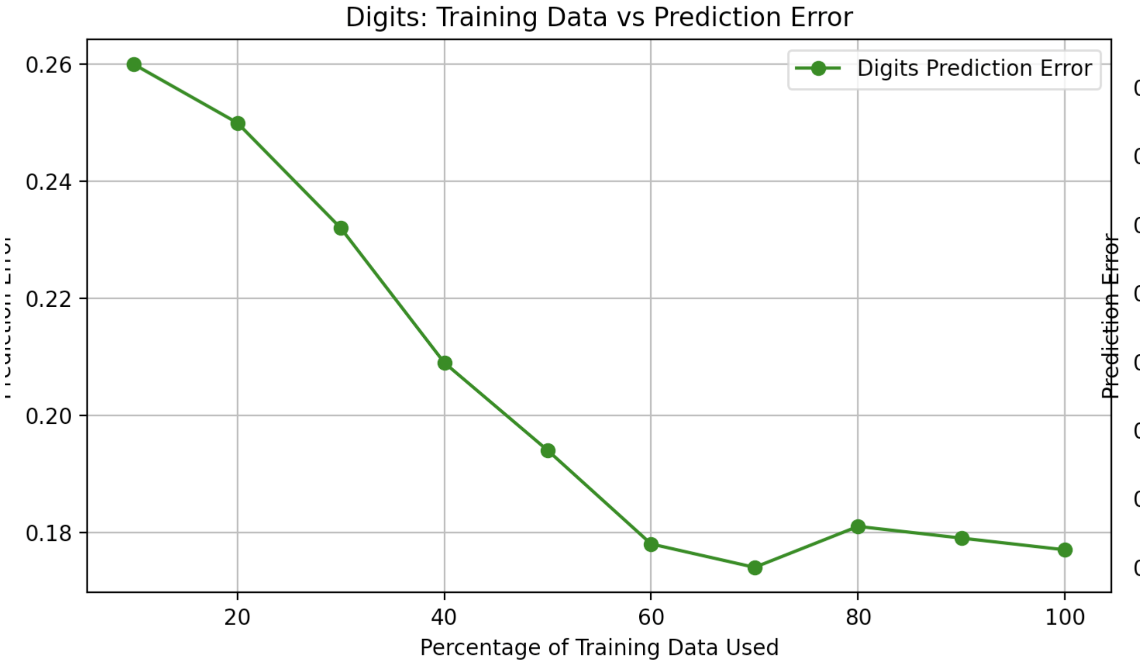
1. **Digits: Training Data vs Run Time -** This graph illustrates how the training time required for the digit recognition model increases as the percentage of the training data used increases, providing insights into the computational cost associated with training on larger datasets.



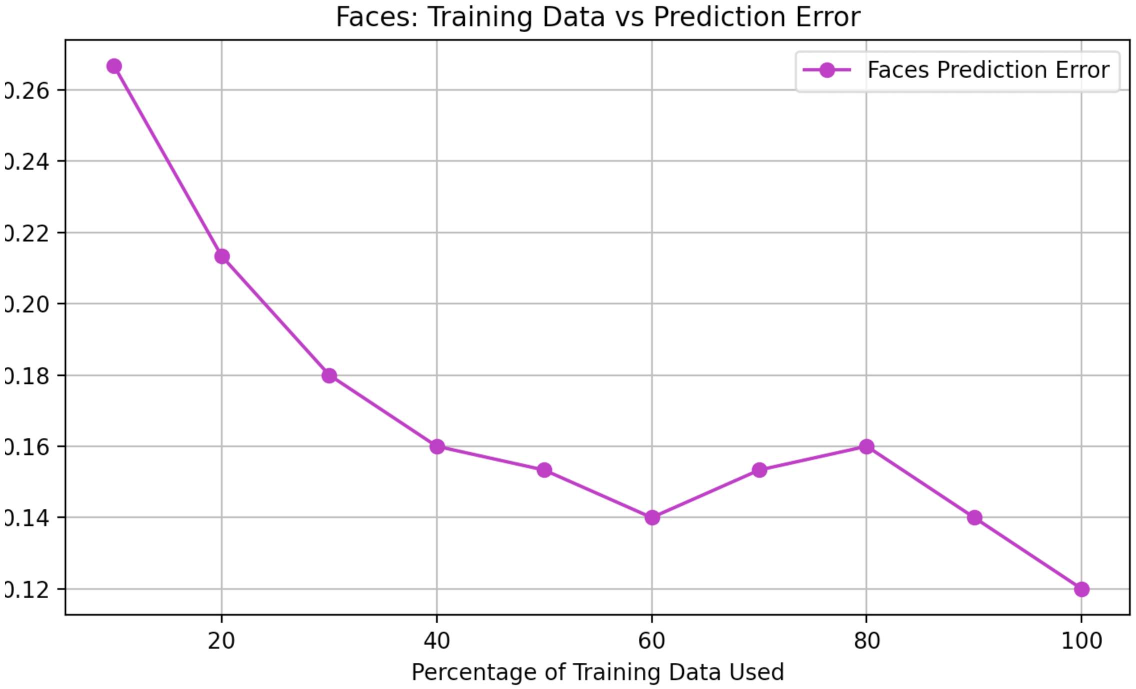
2. **Faces: Training Data vs Run Time -** This plot shows the relationship between the amount of face data used for training and the time it takes to train the model, highlighting the scalability and efficiency of the face recognition model under different data volumes.



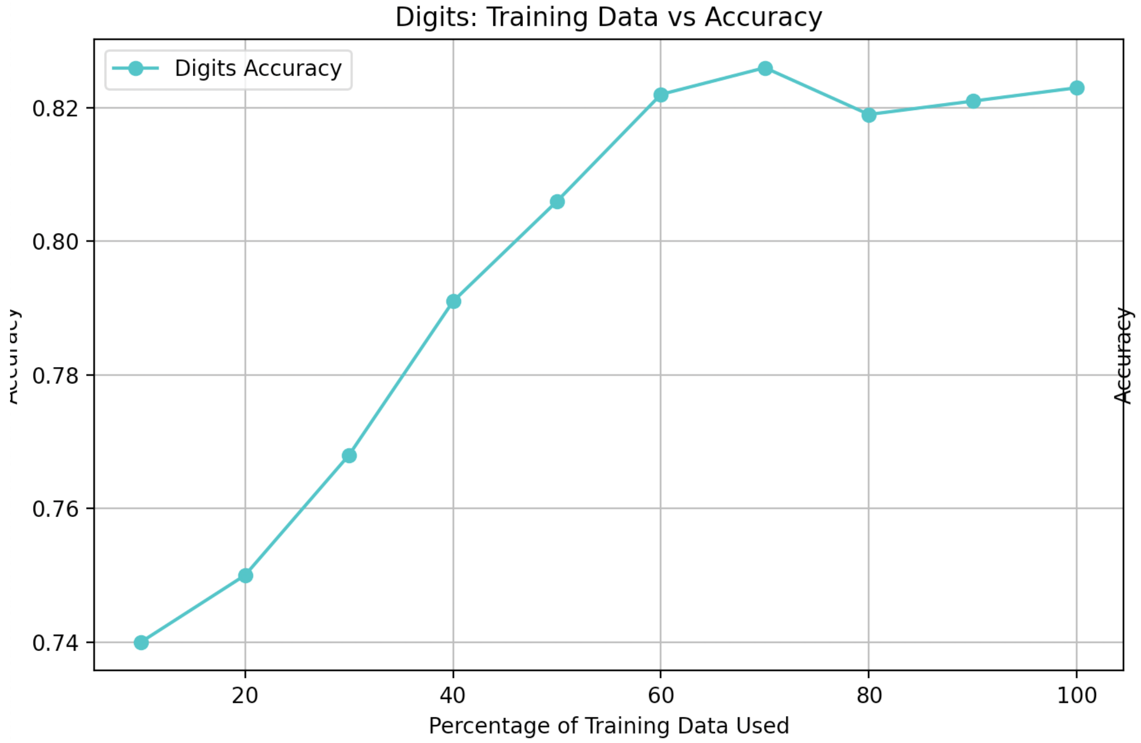
3. **Digits: Training Data vs Prediction Error -** The graph depicts the prediction error of the digit recognition model as a function of the training data size, showing how model accuracy improves as more data is utilized during training.



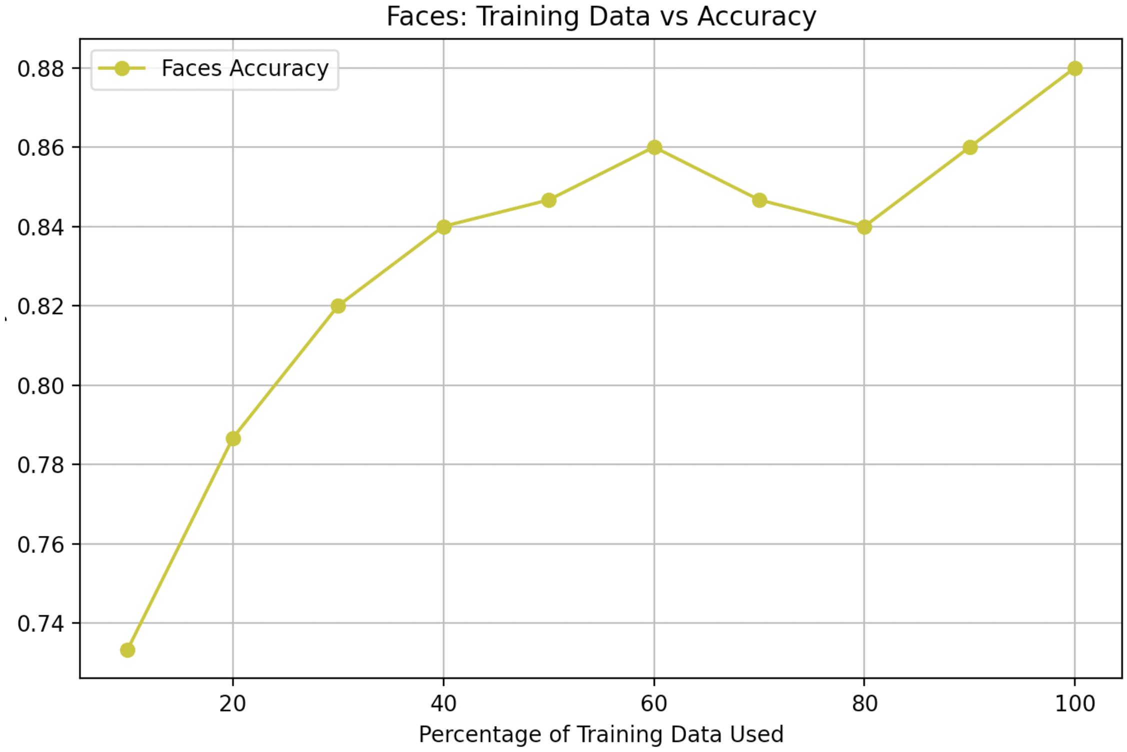
4. **Faces: Training Data vs Prediction Error -** This plot tracks the prediction error for the face recognition model across varying sizes of training data, illustrating how errors decrease and performance improves with more comprehensive training.



5. **Digits: Training Data vs Accuracy** - The graph displays the accuracy of the digit recognition model as training data size increases, emphasizing the positive impact of more extensive training datasets on model performance.



6. **Faces: Training Data vs Accuracy** - This plot demonstrates the accuracy of the face recognition model as a function of the amount of data used for training, revealing how the model's ability to correctly identify faces enhances with greater data input.


Faces: Training Data vs Accuracy
Faces Accuracy
Percentage of Training Data Used