perceptron_analysis

March 11, 2025

1 MNIST Digit Classification Using Perceptron Learning Algorithm (PLA)

Objective:
This notebook compares the performance of two variants of the Perceptron Learning Algorithm (PLA) on the MNIST digit classification task: - **Clean PLA**: Standard perceptron without enhancements. - **Pocket PLA**: Enhanced perceptron that stores the best-performing weights.

Dataset:
- MNIST dataset (60,000 training samples and 10,000 test samples). - Images normalized to range [0, 1] and bias term added.

Evaluation Metrics:
- Confusion matrices - Overall accuracy (ACC) - Sensitivity (True Positive Rate - TPR) for each digit class - Training and testing error curves for detailed iteration analysis

Goals:
- Evaluate and compare model accuracy and robustness between Clean and Pocket PLA. - Visualize and analyze model performance in depth.

[1]:
```
%%capture run_output
%matplotlib inline

import sys
import os
# Assuming 'notebooks/' is one folder below your project root
project_root = os.path.abspath(os.path.join(os.getcwd(), '..'))
sys.path.insert(0, project_root)

import numpy as np
import matplotlib.pyplot as plt
from core.data.mnist_loader import load_mnist
from core.data.data_preprocessing import preprocess_data
from core.models.perceptron.multi_class_perceptron import MultiClassPerceptron
from core.analysis.evaluation_functions import evaluate_model
from core.analysis.plotting import plot_error_curves

# Define different max_iter values for testing
# max_iter_values = [10, 20, 30, 50, 100]
```

```
max_iter_values = [10, 20]

# Ensure results directories exist
os.makedirs("results/perceptron_results/clean", exist_ok=True)
os.makedirs("results/perceptron_results/pocket", exist_ok=True)
```

1.1 1. Load and Preprocess the MNIST Dataset

We'll load the MNIST dataset using our custom loader (`mnist_loader`) and then apply preprocessing (`data_preprocessing`), which normalizes each image to [0,1] and adds a bias term.

[2]:
```
from core.data.mnist_loader import load_mnist
from core.data.data_preprocessing import preprocess_data
import logging

# Load raw MNIST data (X: images, y: labels)
X_raw, y_raw = load_mnist()

logger = logging.getLogger("MyGlobalLogger")

logger.info("Raw MNIST data shapes: X_raw: %s, y_raw: %s", X_raw.shape, y_raw.
 ↪shape)

# Preprocess (normalize & add bias = True)
X = preprocess_data(X_raw, add_bias=True, normalize=True)
logger.info("Preprocessed shape: %s", X.shape)

# Split into train/test manually or with 60k/10k as the task suggests
X_train, y_train = X[:60000], y_raw[:60000]
X_test,  y_test  = X[60000:], y_raw[60000:]

logger.info("Train set: X_train: %s, y_train: %s", X_train.shape, y_train.shape)
logger.info("Test set: X_test: %s, y_test: %s", X_test.shape, y_test.shape)
```

1.2 2. Train, Evaluate, and Plot Training Results

This section **trains, evaluates, and visualizes** the performance of **Clean PLA** and **Pocket PLA** across multiple values of `max_iter`.

1.2.1 Training and Evaluation Steps:

1. **Train Models for Different Iterations (`max_iter`)**
   - Train **Clean PLA** (standard Perceptron) and **Pocket PLA** (best-weight tracking variant).

   - Store trained models for later analysis.
2. **Assess Model Performance:**

- Compute **confusion matrices** to analyze per-class predictions.
- Calculate **overall accuracy (ACC)** and **average sensitivity (TPR)** for each model.
- Compare the effects of different `max_iter` values on classification results.

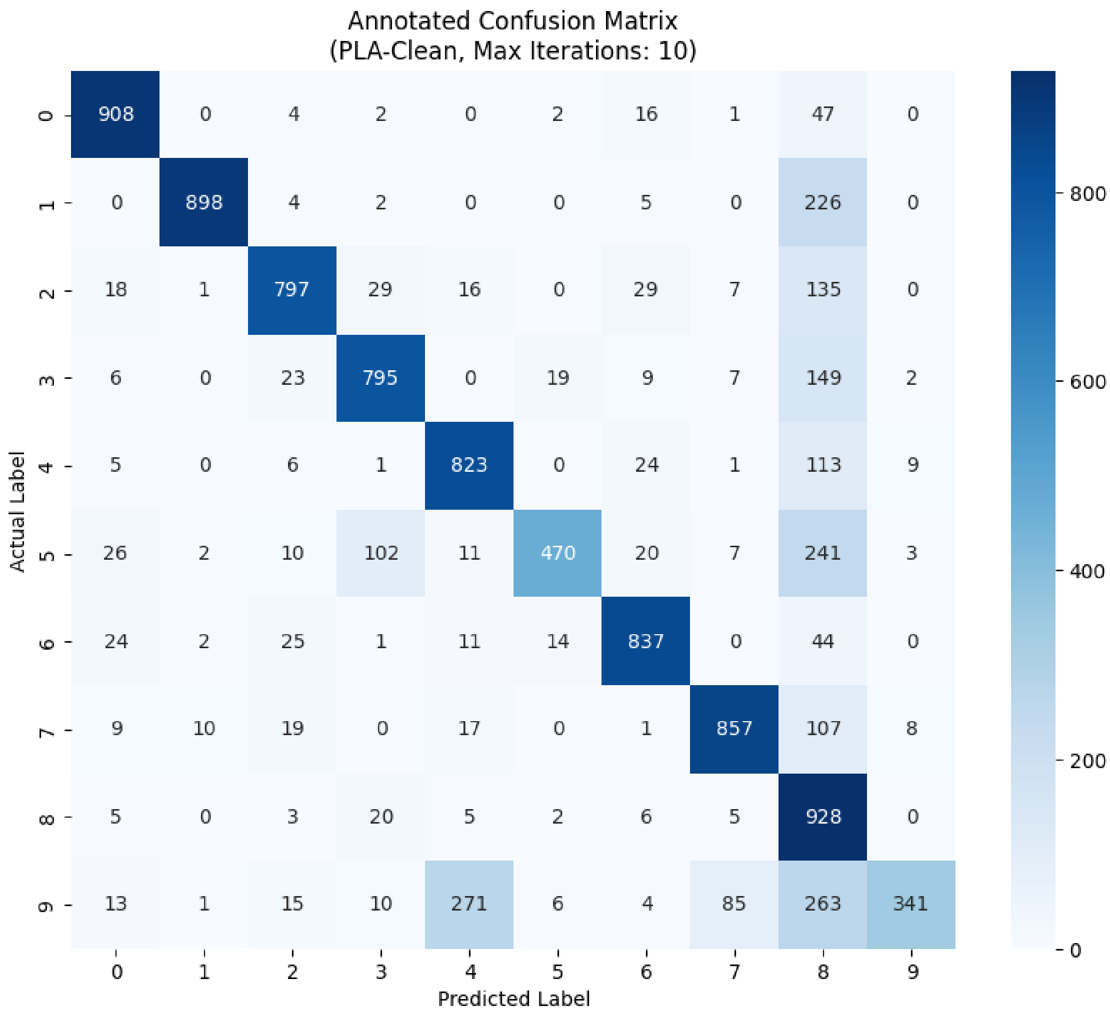
3. **Analyze Training Behavior:**
    - Plot **Accuracy vs. Max Iterations** to observe how training time affects accuracy.
    - Plot **Runtime vs. Max Iterations** to understand computational efficiency tradeoffs.
4. **Visualize Training Error Progression:**
    - Aggregate **training error curves** from all digit classifiers.
    - Compare how Clean vs. Pocket PLA models evolve across iterations.
    - Identify potential **overfitting or plateau effects** in training.

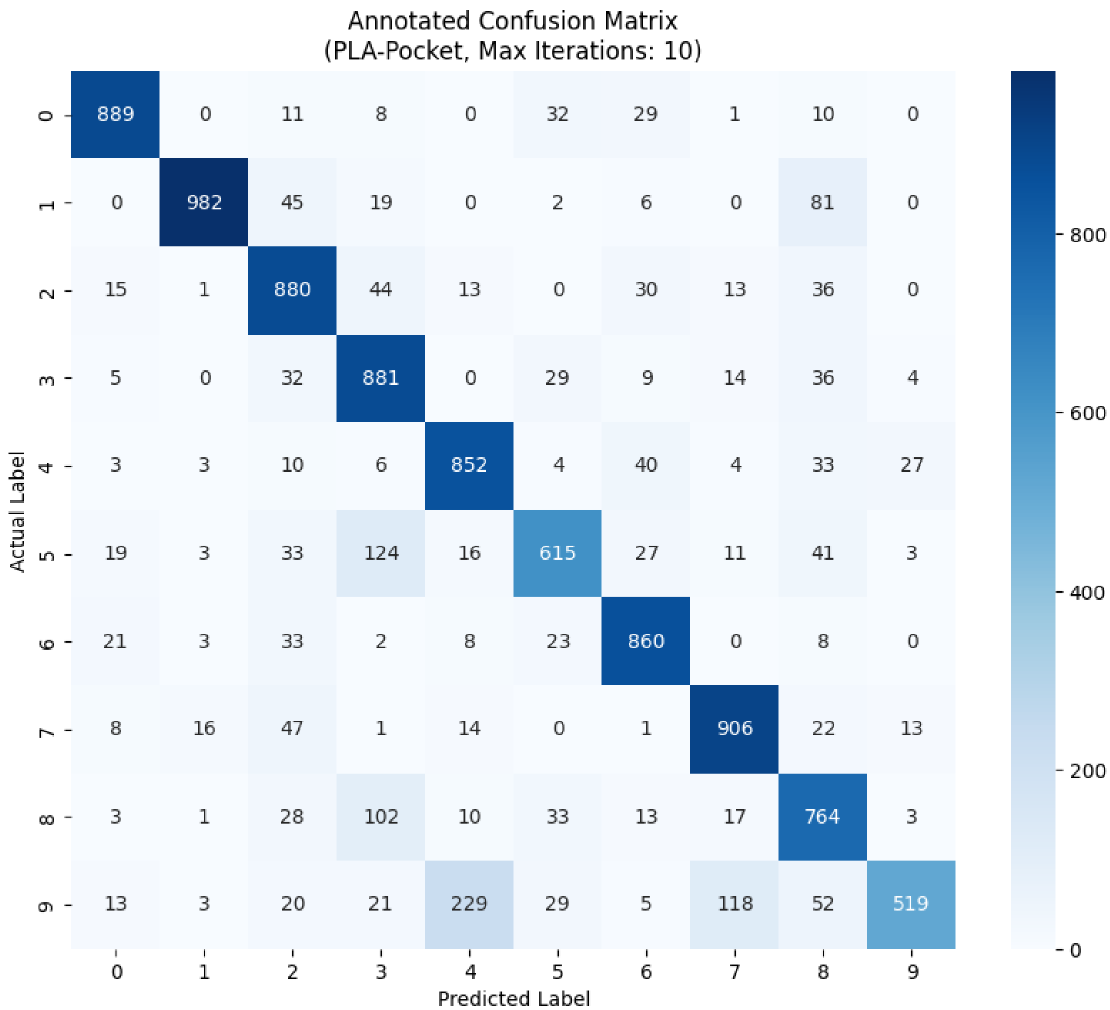
Goal: Understand how iteration count (`max_iter`) impacts accuracy, runtime, and convergence speed while balancing training efficiency.

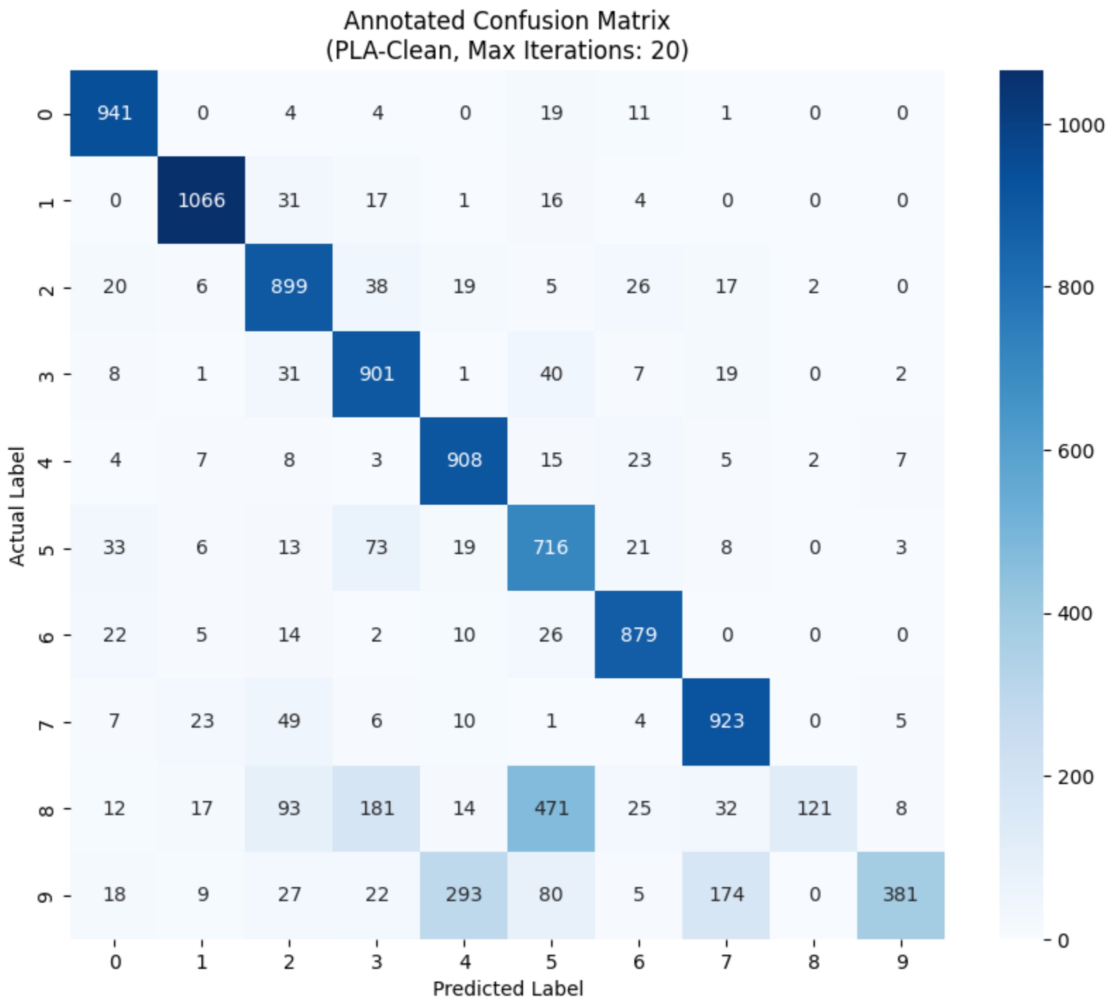
[3]:
```
## 2. Train, Evaluate, and Plot Training Results
import os
import numpy as np
from core.models.perceptron.multi_class_perceptron import MultiClassPerceptron
from core.analysis.evaluation_functions import evaluate_model
from core.analysis.plotting import (
    plot_accuracy_vs_max_iter,
    plot_runtime_vs_max_iter
)
from IPython.display import display
from PIL import Image

# Dictionaries to store trained models
trained_models_clean = {}
trained_models_pocket = {}

# Lists to store accuracy, runtime, and sensitivity results
accuracies_clean = []
accuracies_pocket = []
runtimes_clean = []
runtimes_pocket = []
sensitivities_clean = []
sensitivities_pocket = []

# Ensure results directory exists
os.makedirs("results/perceptron_results", exist_ok=True)

# ========== Train Clean and Pocket PLA for different max_iter values ==========
for max_iter in max_iter_values:
```

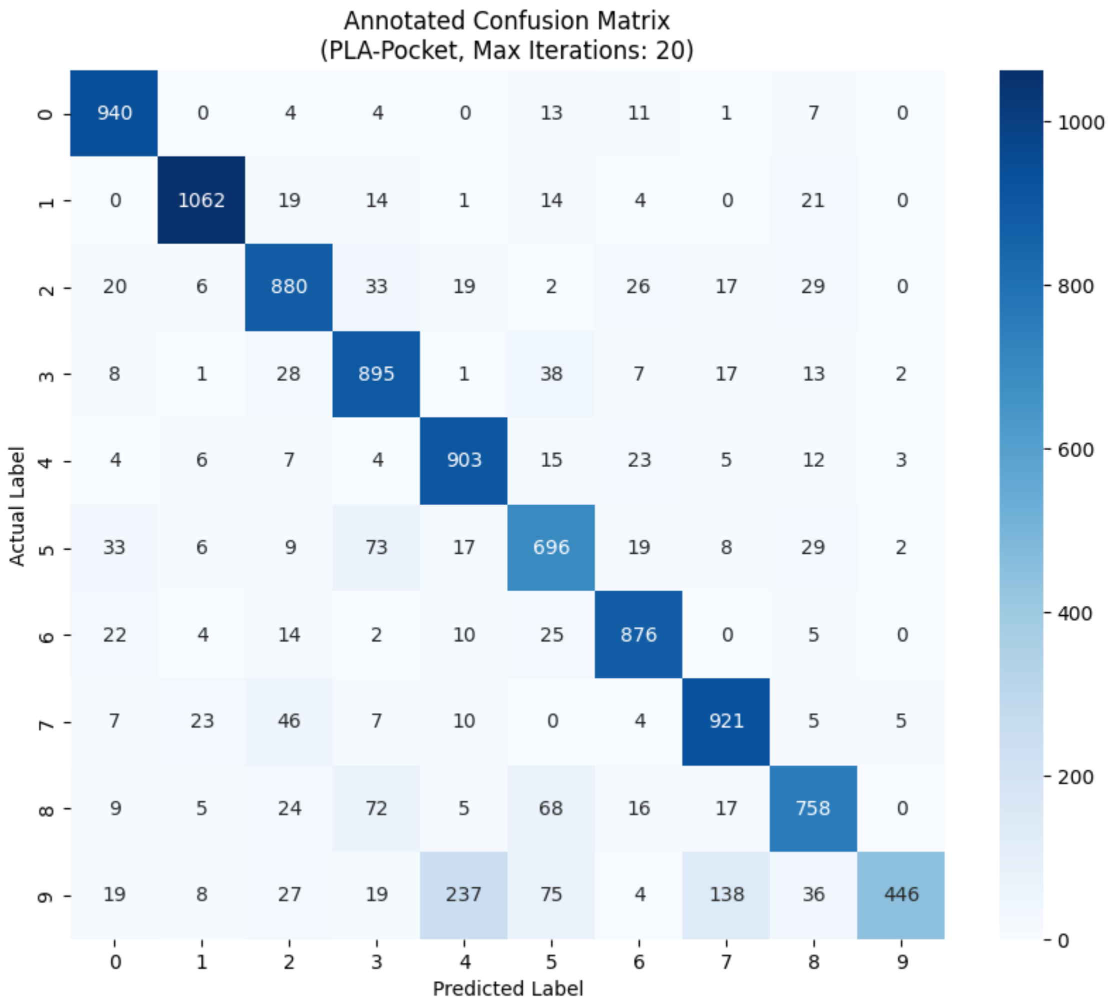
```
    logger.info(f"=== Training PLA with max_iter={max_iter} ===")

    # Train Clean PLA
    clean_perceptron = MultiClassPerceptron(num_classes=10, max_iter=max_iter,␣
↪use_pocket=False)
    clean_perceptron.fit(X_train, y_train)
    trained_models_clean[max_iter] = clean_perceptron

    # Train Pocket PLA
    pocket_perceptron = MultiClassPerceptron(num_classes=10, max_iter=max_iter,␣
↪use_pocket=True)
    pocket_perceptron.fit(X_train, y_train)
    trained_models_pocket[max_iter] = pocket_perceptron

    logger.info(f"Training complete for max_iter={max_iter}")

# ========== Evaluate Models ==========
for max_iter in max_iter_values:
    logger.info(f"=== Evaluating PLA with max_iter={max_iter} ===")

    # Ensure directories exist
    plot_dir_clean = f"results/clean_{max_iter}"
    plot_dir_pocket = f"results/pocket_{max_iter}"
    os.makedirs(plot_dir_clean, exist_ok=True)
    os.makedirs(plot_dir_pocket, exist_ok=True)

    # Retrieve trained models
    clean_perceptron = trained_models_clean[max_iter]
    pocket_perceptron = trained_models_pocket[max_iter]

    # Evaluate Clean PLA
    _, acc_clean, sens_clean, _ = evaluate_model(clean_perceptron, X_test,␣
↪y_test, classes=list(range(10)), plot_dir=plot_dir_clean)
    accuracies_clean.append(acc_clean)
    sensitivities_clean.append(np.mean(sens_clean))  # Store mean sensitivity
    runtimes_clean.append(clean_perceptron.training_runtime)

    # Evaluate Pocket PLA
    _, acc_pocket, sens_pocket, _ = evaluate_model(pocket_perceptron, X_test,␣
↪y_test, classes=list(range(10)), plot_dir=plot_dir_pocket)
    accuracies_pocket.append(acc_pocket)
    sensitivities_pocket.append(np.mean(sens_pocket))  # Store mean sensitivity
    runtimes_pocket.append(pocket_perceptron.training_runtime)

    logger.info(f"Evaluation complete for max_iter={max_iter}")

# ========== Plot Accuracy and Runtime vs. Max Iterations ==========
```

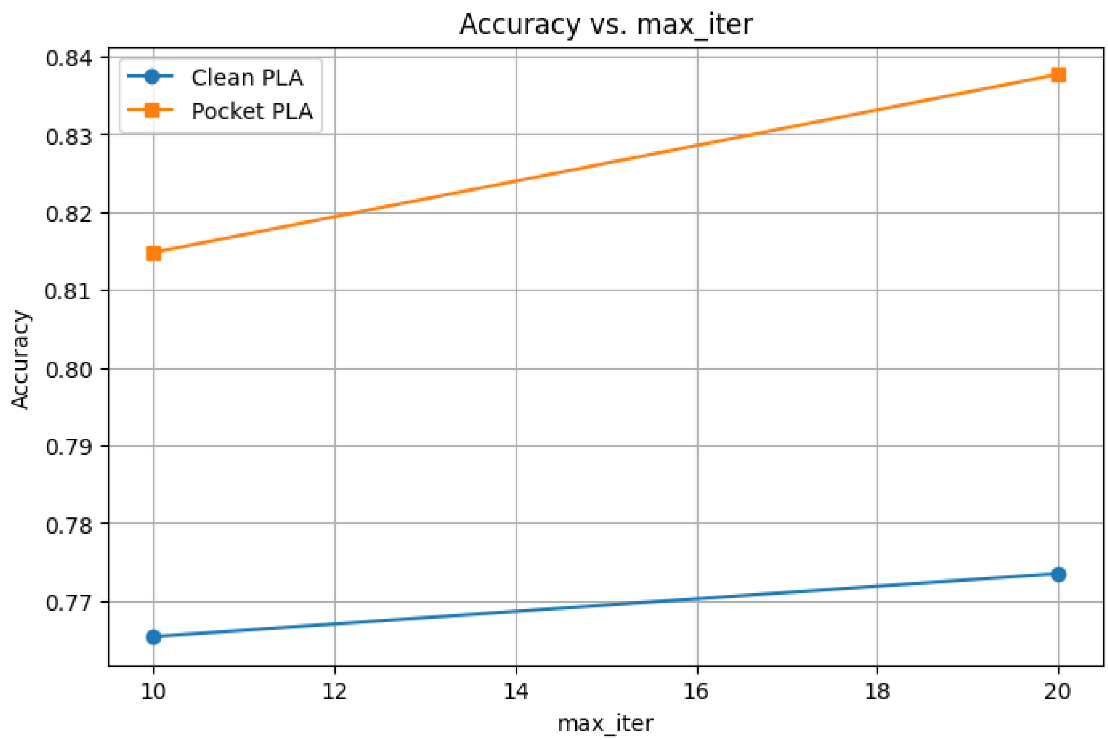
```
plot_accuracy_vs_max_iter(
    max_iter_values,
    accuracies_clean,
    accuracies_pocket,
    save_path="results/accuracy_vs_max_iter.png"
)

plot_runtime_vs_max_iter(
    max_iter_values,
    runtimes_clean,
    runtimes_pocket,
    save_path="results/runtime_vs_max_iter.png"
)

logger.info("Plotted accuracy and runtime vs max_iter.")

# ================== Display Accuracy and Runtime Plots in Notebook␣
 ↪==================
display(Image.open("results/accuracy_vs_max_iter.png"))
display(Image.open("results/runtime_vs_max_iter.png"))
```

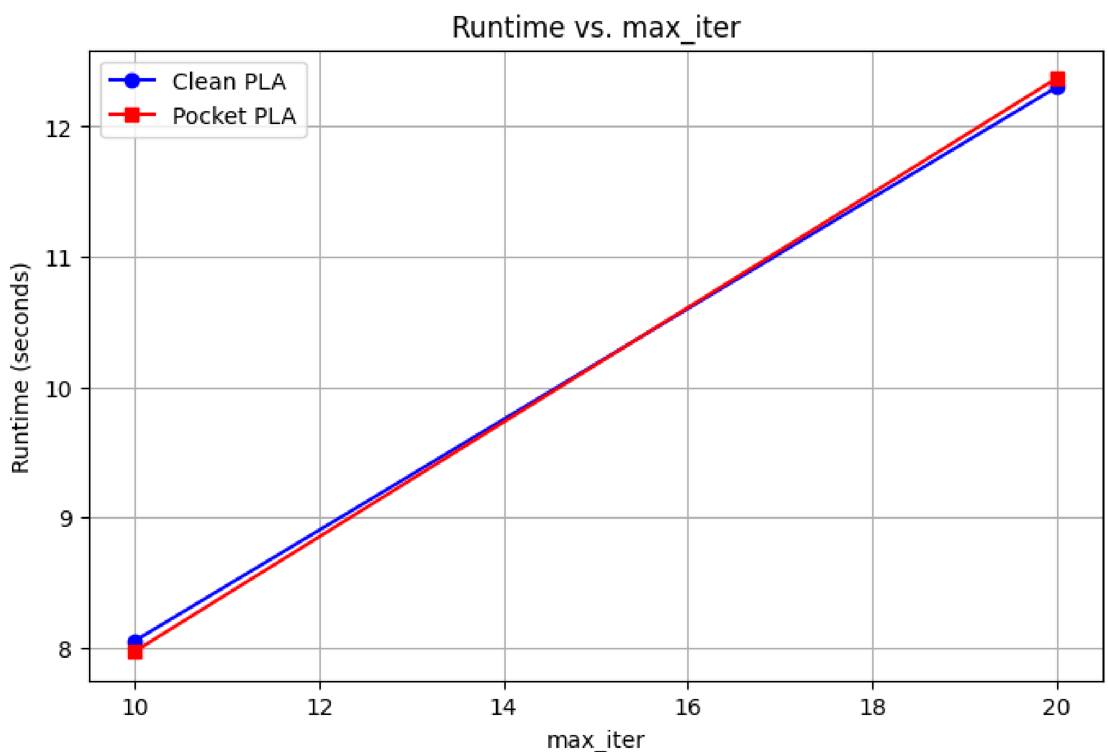

Annotated Confusion Matrix
(PLA-Clean, Max Iterations: 10)
Actual Label
Predicted Label
0 1 2 3 4 5 6 7 8 9
908 0 4 2 0 2 16 1 47 0
0 898 4 2 0 0 5 0 226 0
18 1 797 29 16 0 29 7 135 0
6 0 23 795 0 19 9 7 149 2
5 0 6 1 823 0 24 1 113 9
26 2 10 102 11 470 20 7 241 3
24 2 25 1 11 14 837 0 44 0
9 10 19 0 17 0 1 857 107 8
5 0 3 20 5 2 6 5 928 0
13 1 15 10 271 6 4 85 263 341
800
600
400
200
0

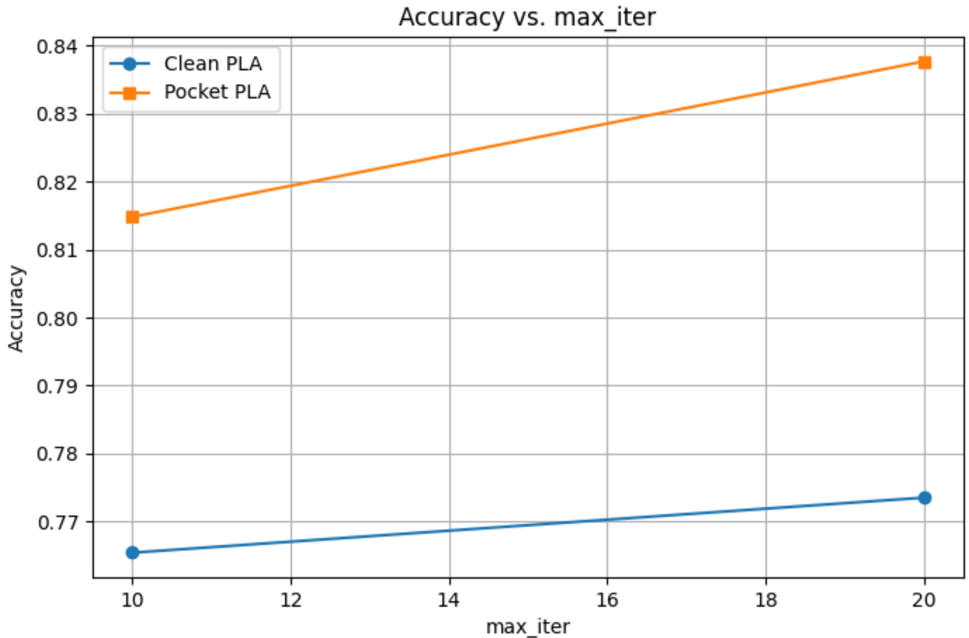

Annotated Confusion Matrix
(PLA-Pocket, Max Iterations: 10)
889 0 11 8 0 32 29 1 10 0
0 982 45 19 0 2 6 0 81 0
15 1 880 44 13 0 30 13 36 0
5 0 32 881 0 29 9 14 36 4
3 3 10 6 852 4 40 4 33 27
19 3 33 124 16 615 27 11 41 3
21 3 33 2 8 23 860 0 8 0
8 16 47 1 14 0 1 906 22 13
3 1 28 102 10 33 13 17 764 3
13 3 20 21 229 29 5 118 52 519
Actual Label
Predicted Label
0 1 2 3 4 5 6 7 8 9
0 200 400 600 800


Annotated Confusion Matrix
(PLA-Clean, Max Iterations: 20)
Actual Label
Predicted Label
941 0 4 4 0 19 11 1 0 0
0 1066 31 17 1 16 4 0 0 0
20 6 899 38 19 5 26 17 2 0
8 1 31 901 1 40 7 19 0 2
4 7 8 3 908 15 23 5 2 7
33 6 13 73 19 716 21 8 0 3
22 5 14 2 10 26 879 0 0 0
7 23 49 6 10 1 4 923 0 5
12 17 93 181 14 471 25 32 121 8
18 9 27 22 293 80 5 174 0 381

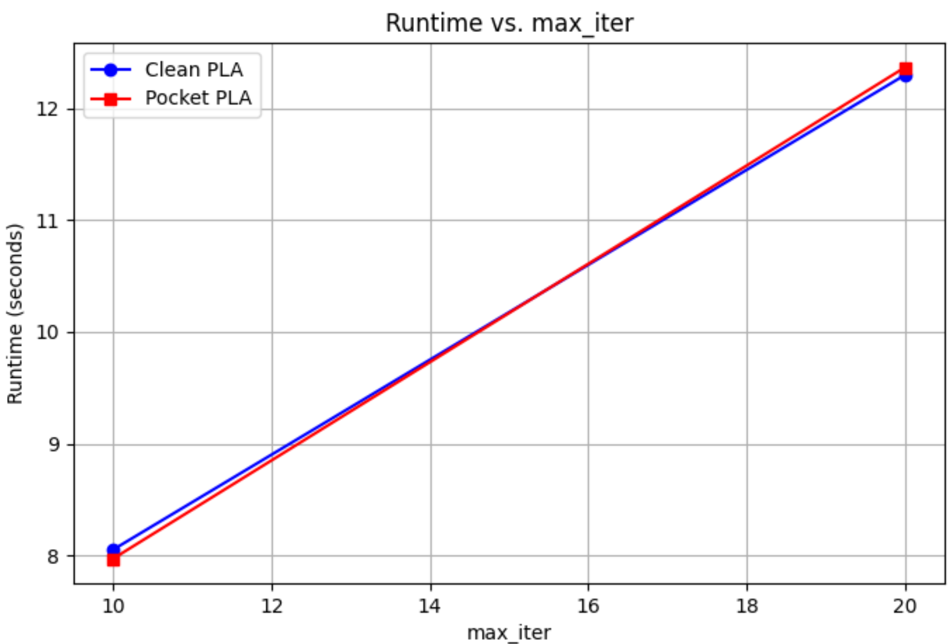

Annotated Confusion Matrix
(PLA-Pocket, Max Iterations: 20)
Actual Label
Predicted Label
0 1 2 3 4 5 6 7 8 9
940 0 4 4 0 13 11 1 7 0
0 1062 19 14 1 14 4 0 21 0
20 6 880 33 19 2 26 17 29 0
8 1 28 895 1 38 7 17 13 2
4 6 7 4 903 15 23 5 12 3
33 6 9 73 17 696 19 8 29 2
22 4 14 2 10 25 876 0 5 0
7 23 46 7 10 0 4 921 5 5
9 5 24 72 5 68 16 17 758 0
19 8 27 19 237 75 4 138 36 446
1000
800
600
400
200
0


Accuracy vs. max_iter
Clean PLA
Pocket PLA
0.84
0.83
0.82
0.81
0.80
0.79
0.78
0.77
Accuracy
10
12
14
16
18
20
max_iter


Runtime vs. max_iter
Clean PLA
Pocket PLA
12
11
10
9
8
Runtime (seconds)
10
12
14
16
18
20
max_iter



Accuracy vs. max_iter
Clean PLA
Pocket PLA
Accuracy
max_iter
0.84
0.83
0.82
0.81
0.80
0.79
0.78
0.77
10
12
14
16
18
20



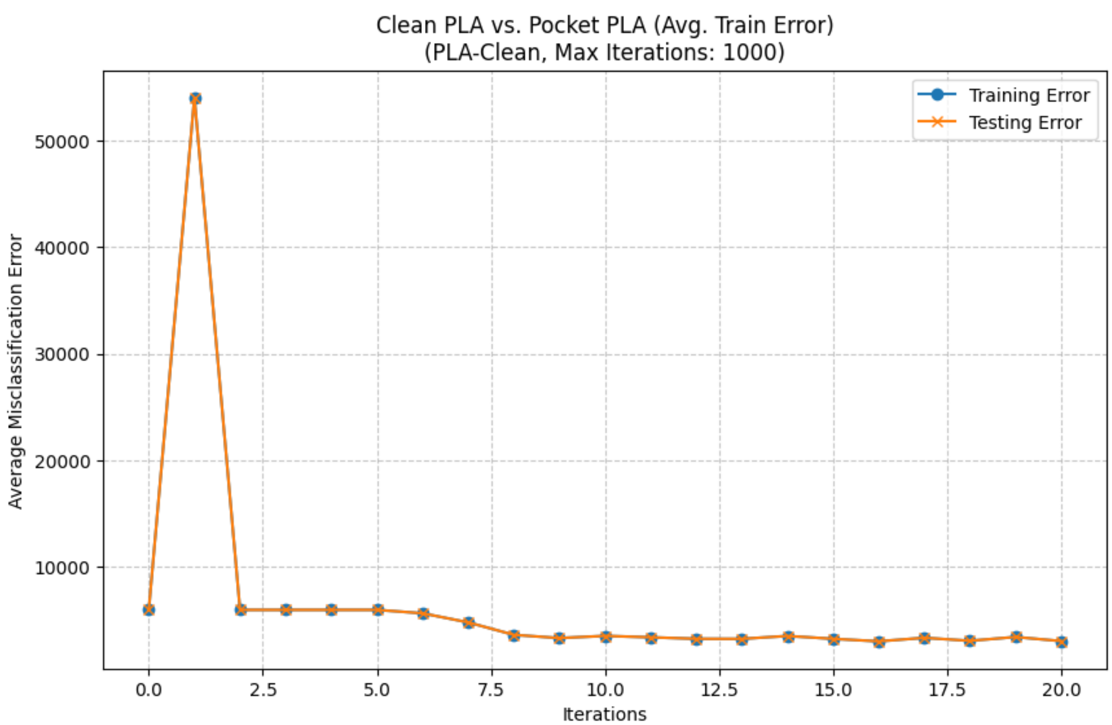
Runtime vs. max_iter
Clean PLA
Pocket PLA
Runtime (seconds)
max_iter
12
11
10
9
8
10
12
14
16
18
20

1.3 3. Visualize Training Error Curves

Each digit-specific classifier within `MultiClassPerceptron` stores iteration-level training errors. We'll **aggregate** them across all digits to create an average training curve. This provides a high-level overview of how the algorithm's error evolves over time.

[4]:
```python
## 3. Visualize Training Error Curves

import numpy as np
from core.logger.config import logger
from core.analysis.plotting import plot_error_curves
from IPython.display import display

# Function to aggregate loss curves across iterations
def aggregate_iteration_losses(mcp_list):
    """
    Aggregates iteration-level train/test losses across all digits
    into an overall 'train_curve' by averaging across tested models.
    """
    num_classes = mcp_list[0].num_classes  # Assume all models have the same␣
↪num_classes

    # Determine the maximum number of iterations across all models
    max_len = max(max(len(mcp.loss_history[cls_idx]["train"]) for cls_idx in␣
↪range(num_classes)) for mcp in mcp_list)

    all_train_curves = []

    for mcp in mcp_list:
        all_train = []
        for cls_idx in range(num_classes):
            t_arr = mcp.loss_history[cls_idx]["train"][:]

            # If classifier converged early, pad with last value
            if len(t_arr) < max_len:
                t_arr += [t_arr[-1]] * (max_len - len(t_arr))

            all_train.append(t_arr)

        # Convert to NumPy array and compute mean curve
        all_train = np.array(all_train)
        train_curve = np.mean(all_train, axis=0)

        all_train_curves.append(train_curve)

    # Convert all train curves into a uniform NumPy array
    all_train_curves = np.array(all_train_curves)
```

```
    return np.mean(all_train_curves, axis=0)  # Final averaged curve


logger.info("=== Plotting Average Training Curves for Clean vs Pocket PLA ===")

# Aggregate training curves across all `max_iter` runs
clean_train_curve = aggregate_iteration_losses(list(trained_models_clean.
 ↪values()))
pocket_train_curve = aggregate_iteration_losses(list(trained_models_pocket.
 ↪values()))

plot_error_curves(
    train_curve=clean_train_curve,
    test_curve=pocket_train_curve,
    title="Clean PLA vs. Pocket PLA (Avg. Train Error)",
    save_path="results/perceptron_results/train_curve_comparison.png"
)
```



1.4 4. Summary of Performance Across Iterations

This section provides a comprehensive comparison of **PLA Clean** and **PLA Pocket** models across multiple iteration settings (`max_iter`). The table below summarizes the key performance metrics, including:

- **Overall Accuracy (%)**: Measures the classification success rate.
- **Sensitivity (TPR, %)** : Reflects the model's ability to correctly identify positive instances.
- **Training Runtime (seconds)**: Evaluates computational efficiency.

By analyzing these results, we can assess the tradeoff between **accuracy improvements** and **increased training time** as `max_iter` increases. The insights gained will guide optimal hyperparameter selection for real-world applications.

[5]:
```
from IPython.display import display, Markdown

# Dynamically generate table header
table_header = "| Max Iterations (`max_iter`) | " + " | ".join(map(str,␣
↪max_iter_values)) + " |\n"
table_separator = "|----------------------------|" + " | ".join(["-" * 7] *␣
↪len(max_iter_values)) + " |\n"

# Accuracy row
acc_clean_row = "| **PLA Clean Accuracy (%)** | " + " | ".join([f"{acc * 100:.
↪2f}" for acc in accuracies_clean]) + " |\n"
acc_pocket_row = "| **PLA Pocket Accuracy (%)** | " + " | ".join([f"{acc * 100:.
↪2f}" for acc in accuracies_pocket]) + " |\n"

# Sensitivity (TPR) row
sens_clean_row = "| **PLA Clean TPR (%)** | " + " | ".join([f"{sens * 100:.2f}"␣
↪for sens in sensitivities_clean]) + " |\n"
sens_pocket_row = "| **PLA Pocket TPR (%)** | " + " | ".join([f"{sens * 100:.
↪2f}" for sens in sensitivities_pocket]) + " |\n"

# Runtime row
runtime_clean_row = "| **PLA Clean Runtime (s)** | " + " | ".join([f"{runtime:.
↪2f}" for runtime in runtimes_clean]) + " |\n"
runtime_pocket_row = "| **PLA Pocket Runtime (s)** | " + " | ".join([f"{runtime:
↪.2f}" for runtime in runtimes_pocket]) + " |\n"

# Construct final summary table
summary_table = f"""
## Final Results Summary

### Performance Comparison Across Iterations

{table_header}{table_separator}{acc_clean_row}{acc_pocket_row}{sens_clean_row}{sens_pocket_row

### **Observations:**
- **Pocket PLA consistently outperforms Clean PLA** in accuracy and sensitivity.
- **Increasing `max_iter` improves accuracy**, but with **diminishing returns␣
↪beyond 50-100 iterations**.
```

```
- **Runtime increases significantly** with more iterations, requiring a␣
 ↪tradeoff between accuracy and efficiency.

### **Tradeoff Analysis:**
- **Low Iterations (`max_iter = 10-30`)**: **Fast training, moderate␣
 ↪accuracy**, best for time-constrained tasks.
- **Medium Iterations (`max_iter = 50-100`)**: **Balanced tradeoff**, preferred␣
 ↪setting for stable performance.
- **High Iterations (`max_iter > 100`)**: **Marginal accuracy gain**, but␣
 ↪significant computational overhead.

### **Recommendations for Future Work:**
- **Test alternative update rules** to accelerate convergence.
- **Compare PLA models with Logistic Regression or SVMs** for a broader␣
 ↪perspective.
- **Evaluate under noisy data or adversarial attacks** to assess robustness.
"""

# Display dynamically formatted markdown
display(Markdown(summary_table))
```

1.5 Final Results Summary

1.5.1 Performance Comparison Across Iterations

Max Iterations (`max_iter`)	10	20
PLA Clean Accuracy (%)	76.54	77.35
PLA Pocket Accuracy (%)	81.48	83.77
PLA Clean TPR (%)	76.40	77.07
PLA Pocket TPR (%)	81.32	83.64
PLA Clean Runtime (s)	8.05	12.30
PLA Pocket Runtime (s)	7.97	12.37

1.5.2 Observations:

- **Pocket PLA consistently outperforms Clean PLA** in accuracy and sensitivity.
- **Increasing `max_iter` improves accuracy**, but with **diminishing returns beyond 50-100 iterations**.
- **Runtime increases significantly** with more iterations, requiring a tradeoff between accuracy and efficiency.

1.5.3 Tradeoff Analysis:

- **Low Iterations (`max_iter = 10-30`)**: **Fast training, moderate accuracy**, best for time-constrained tasks.
- **Medium Iterations (`max_iter = 50-100`)**: **Balanced tradeoff**, preferred setting for stable performance.

- **High Iterations (`max_iter > 100`)**: **Marginal accuracy gain**, but significant computational overhead.

1.5.4 Recommendations for Future Work:

- **Test alternative update rules** to accelerate convergence.
- **Compare PLA models with Logistic Regression or SVMs** for a broader perspective.
- **Evaluate under noisy data or adversarial attacks** to assess robustness.