End-Semester Project Report: MNIST Digit Classifier

M SHIVAARCHITHA VUDUTHA (DA24B052)

(a) A Summary of Models Used & System Architecture

Final Submitted System Architecture

My final submitted system (`main.py`) is a **Tuned K-Nearest Neighbors (k=5) Classifier** combined with a **Principal Component Analysis (PCA)** pre-processing step.

This architecture was chosen after an extensive experimental process because it provided the highest Macro F1 score on the validation set, while also being exceptionally fast to train and evaluate.

The system's data flow is as follows:

1. **Load Data:** The 784-pixel `MNIST_train.csv` data is loaded.
2. **Normalize:** Pixel values are normalized by dividing by 255.0.
3. **PCA:** Our from-scratch `PCA` class is fit on the training data to find the 64 most important "principal components."
4. **Transform:** The data is transformed from 784 features to just 64, drastically reducing dimensionality and noise.
5. **Train:** Our from-scratch `KNNClassifier` is "trained" by memorizing the 64-dimensional PCA-transformed data points and their labels.
6. **Predict:** For a new validation point, the model finds the 5 closest points in its memory (using PCA-feature distance) and predicts the majority-voted class.

Experimental Models

To arrive at my final model, I first implemented and tested a wide range of algorithms from the four categories discussed in class. All algorithms were built from scratch.

1. **Classification:** `SoftmaxRegression` (Multi-class Logistic Regression)
2. **Regression-based:** `OvRLinearClassifier` (A One-vs-Rest model using 10 `LinearRegression` models)
3. **Clustering:** `KMeansClassifier` (A K-Means model with K=10, where each centroid was labeled by its majority class)
4. **Ensemble (Bagging):** `RandomForest` (From-scratch, optimized `DecisionTree` model)

5. **Ensemble (Stacking):** I built two "stacked" ensembles (Path A and Path B) in an attempt to combine my best models.

Ultimately, my analysis (detailed in section 'd') proved that my single `KNNClassifier` outperformed even our most complex stacked ensembles.

(b) Summary of Hyper-Parameter Tuning and Results

I conducted a rigorous tuning process for my three most promising "champion" models.

Model	Hyper-parameter	Values Tested	Best Value	Resulting F1 Score
K-Nearest Neighbors (KNN)	`k` (neighbors)	`[3, 4, 5, 6, 7]`	`5`	**0.9568**
Softmax Regression	`reg_strength`	`[0.1, 0.01, 0.001]`	`0.001`	0.8917
Random Forest	`(depth, n_trees, thresholds)`	`depths [8, 10, 12], n_trees [20, 30, 40], thresholds [10]`	`(10, 30, 10)`	0.8796

Tuning Notes:

- **KNN:** I selected `k=5` over `k=6` (which had a similar 0.9568 score) to prevent ties in the majority vote, ensuring model stability.
- **Random Forest:** Tuning this model required the most optimization (see section c). We found that `depth=10` was the optimal balance of accuracy and speed.

(c) Steps Taken to Optimize System Performance and Limit Run Time

My final `main.py` (which trains and evaluates our champion KNN model) runs in **5.95 seconds**, well within the time limit. This speed was achieved through several key optimization steps:

1. **PCA (Primary Optimization):** My single most important optimization. By using PCA to reduce the feature space from 784 to 64, I made all model training and prediction *dramatically* faster. A `KNNClassifier` on 784 features would have taken hours; on 64 features, it takes seconds.
2. **Random Forest `max_thresholds` (Algorithm-level Optimization):** My initial `RandomForest` was too slow, as its `DecisionTree` checked every unique pixel value as a potential split. I modified the `_find_best_split` function to **sample only 10 random thresholds** (`max_thresholds=10`) per feature. This reduced its training time by over 90% (from 10+ minutes to ~99 seconds) while maintaining high accuracy (0.8796 F1).
3. **Random Forest `max_depth` (Hyper-parameter Optimization):** I proved that `max_depth > 10` (e.g., 12 or 14) caused an exponential increase in training time, failing the time limit. We deliberately chose `max_depth=10` to balance bias and variance *while respecting the time constraint*.

Evaluation Results on Training and Validation Datasets

Model / System	F1 Score (Validation)	Total Runtime
Final Model: Tuned KNN (k=5)	**0.9568**	**5.95 seconds**
Tuned Softmax Regression	0.8917	~20 seconds
Tuned Random Forest	0.8796	~99 seconds
Simple Stack (Ensemble A)	0.9489	~6.39 minutes
Smarter Stack (Ensemble B)	0.9289	~6.44 minutes

(d) Detailed Summary of Your Thoughts and Observations

This exercise was an excellent demonstration of the classic "complexity vs. performance" trade-off.

My most important observation was that **model complexity does not guarantee better performance.**

We began by building a suite of individual models and found a clear champion: the **Tuned KNN (F1: 0.9568)**. My hypothesis was that a "Stacked Ensemble" would combine the strengths of our KNN, Softmax (0.89), and RF (0.87) models to correct for individual errors and achieve an even higher score (e.g., 0.98).

The experimental results proved this hypothesis wrong:

- **Simple Stack (Path A) F1:** `0.9489` (Worse than KNN alone)
- **Smarter Stack (Path B) F1:** `0.9289` (Even worse than Path A)

This is a fascinating result. It demonstrates that my "weaker" expert models (Softmax and RF) were not correcting the KNN's few mistakes. Instead, their own incorrect predictions were just **adding noise** to the meta-model. The meta-model, in trying to listen to all three "experts," got confused and performed worse than if it had just listened to the single best expert, the KNN.

My final conclusion is that for this specific, PCA-transformed dataset, the simple, instance-based `KNNClassifier` was the optimal solution. The data-driven decision was to *reject* the more complex ensembles and submit the single, well-tuned model that demonstrably performed the best.