

# MNIST Classification System Report

## (a) Summary of Models Used & System Architecture

This is implemented by using simple libraries without external ML libraries. These are the models defined in `algorithms.py` :

- **LinearRegressionOVR**: Uses a one-vs-rest setup. The code computes a closed-form weight matrix  $W = (X^T X + I)^{-1} X^T Y$ . Prediction simply multiplies input  $X$  with  $W$  and picks the class with the highest score.
- **LogisticRegressionOVR**: Also one-vs-rest. The code initializes weights to zero and performs a fixed number of gradient descent updates using the sigmoid function from SciPy. Each iteration updates all 10 class-specific weight vectors simultaneously.
- **KMeansClassifier**: The code randomly selects initial centroids, iteratively updates them using Euclidean distance, and assigns each centroid the majority class label based on the training data points that fall into that cluster.
- **GaussianClassifier**: The code computes per-class means and variances for each pixel. During prediction, for each class, it computes a vectorized log-likelihood and selects the class with the highest value.
- **EnsembleClassifier**: The ensemble takes predictions from all previously trained models and performs weighted voting using manually specified weights [1.2, 2.0, 0.6, 0.8].

We read CSV files, normalize pixel values, train each model, measure runtime, compute macro-F1, and finally evaluate the ensemble. All components communicate using `fit()` and `predict()` methods.

## (b) Hyperparameter Tuning and Model Results

\* All hyperparameters used come directly from the code:

- Linear Regression: `lambda_reg = 1e-2`
- Logistic Regression: learning rate `lr = 0.8`, iterations 150
- K-Means: `k = 60` clusters, `iters = 15`

- Gaussian Classifier: fixed variance smoothing  $1e-5$
- Ensemble Weights: [1.2, 2.0, 0.6, 0.8]

The main script stores results in a dictionary that maps model names to:

(macro F1 score, runtime in seconds)

These are printed after training. The ensemble result prints separately using measured `ens_time`. The exact scores depend on the user's machine and dataset, since the code computes them dynamically.

## (c) Steps Taken to Optimize Runtime and System Performance

- Input data is normalized immediately after loading (`X / 255.0`), reducing numerical instability.
- All models are implemented in vectorized NumPy form without Python loops except where unavoidable (e.g., K-Means cluster updates).
- Linear regression uses the closed-form solver `np.linalg.solve` instead of slower gradient descent.
- Logistic regression performs a fixed number of iterations, ensuring predictable runtime.
- K-Means uses only 15 iterations and 60 clusters to control computation cost.
- Gaussian classifier uses elementwise operations, making prediction extremely fast.
- The ensemble does not re-train any model; it only aggregates their predictions.

Evaluation uses macro-F1 computed using a vectorized custom function `f1_macro()` defined in the script.

## (d) Thoughts and Observations From the Exercise

- Each model implements a minimal and clean interface (`fit()`, `predict()`), making them easy to combine.
- The models differ significantly in training cost: linear and logistic regression compute large matrix operations, while Gaussian is the fastest to train.
- K-Means is the slowest due to repeated distance calculations, but it still integrates cleanly into the ensemble.
- The ensemble performance depends heavily on the weights specified in code; logistic regression is weighted highest because it tends to perform best.

- The macro-F1 metric implemented by hand provides consistent evaluation across models and is easy to modify if needed.
- The system demonstrates how different classical ML models can be unified under one architecture without using libraries like scikit-learn.