Report

Sai Jagadeesh

November 2025

1 Summary of Models Used and System Architecture

This project implements four classical machine learning models from scratch (using only `NumPy`) to perform multi-class digit classification on the MNIST dataset. A point-wise summary of the models and the overall system architecture is provided below.

1.1 Models Used

1) Logistic Regression (One-vs-Rest)

- Ten binary logistic classifiers are trained (one per digit).
- Uses the sigmoid activation function and mini-batch gradient descent.
- Features are standardized using training-set mean and variance.
- During prediction, probabilities from the ten classifiers are computed and the class with the highest value is selected.

2) Support Vector Machine (Hinge Loss with SGD)

- Implements a linear SVM trained using stochastic gradient descent on hinge loss.
- One-vs-rest strategy: Ten separate binary classifiers.
- Includes L2 regularization to maximize margin and prevent overfitting.
- Predicts the class with highest linear score ($w^\top x - b$).

3) K-Nearest Neighbors (Cosine Distance)

- Instance-based method: no training phase required.
- Pixel intensities are scaled to $[0, 1]$ and L2-normalized.

- Cosine similarity is computed efficiently using matrix multiplication.
- Weighted voting among the k nearest neighbors determines the predicted class.

4) XGBoost Multiclass (Custom Implementation)

- Implements gradient boosting using decision stumps or shallow trees.
- Uses first- and second-order gradients (gradient + Hessian).
- Trained in a one-vs-rest manner for multi-class classification.
- Final logits from all ten binary boosters are combined using softmax for calibrated probabilities.

1.2 System Architecture

1) Data Loading

- MNIST training and validation datasets are loaded from CSV files.
- Each sample is a flattened 28×28 grayscale image with its label.

2) Preprocessing

- Logistic Regression and SVM: Standardization (zero mean, unit variance).
- KNN: Pixel scaling to $[0, 1]$ followed by L2 normalization.
- XGBoost: Raw pixel intensities used (trees do not require scaling).

3) Model Training

- Each model is trained independently using its tuned hyperparameters.
- One-vs-rest leads to ten classifiers per model (except KNN which is non-parametric).

4) Prediction Pipeline

- For each model, predictions from the ten binary classifiers are computed.
- Final class label is obtained using the $\arg\max$ rule over scores or probabilities.

5) Evaluation Metrics

- Accuracy and macro F1-score are computed on the validation split.
- Macro F1 ensures balanced evaluation across all digit classes.

6) System Constraints and Design Choices

- Entire training pipeline completes within the allowed limit of five minutes.
- All models are implemented from scratch, ensuring transparency and full control.
- Architecture is modular, allowing easy experimentation and debugging.

2 Hyperparameter Tuning

This section presents a detailed description of hyperparameter tuning conducted individually for each model. For every algorithm, the most influential hyperparameters were varied systematically, and their effects on validation accuracy were analyzed. The goal was to determine configurations that yield the best accuracy while respecting the computational limit of five minutes. All experiments were executed using the MNIST dataset on models implemented solely with `NumPy`.

2.1 Logistic Regression (One-vs-Rest)

Logistic Regression depends primarily on the learning rate, number of epochs, and mini-batch size. Each hyperparameter was evaluated independently while keeping the others fixed.

Learning Rate

A very small learning rate slows convergence, whereas a large value causes oscillations. The results of tuning the learning rate are shown in Table 1.

Table 1: Learning Rate Tuning for Logistic Regression

Learning Rate	Accuracy
0.10	92.17%
0.20	92.98%
0.20	**93.12%**
0.30	92.87%

Epochs

Increasing epochs beyond a certain point does not lead to further gains. Performance plateaued around 40 epochs, as shown in Table 2.

Table 2: Epoch Tuning for Logistic Regression

Epochs	Accuracy
20	90.8%
30	92.4%
40	**93.12%**

Batch Size

Mini-batch size affects both runtime and gradient stability. Batch size 256 provided the best overall trade-off (Table 3).

Table 3: Batch Size Tuning for Logistic Regression

Batch Size	Accuracy
128	91.4%
256	**93.12%**
512	92.9%

Final Setting: lr = 0.2, epochs = 40, batch size = 256.
The Final Macro F1 score is 93.12% **and the total run time is** $1.2min$

2.2 Support Vector Machine (Hinge Loss SGD)

The Multiclass SVM implementation performs per-sample SGD updates, making it sensitive to the learning rate, regularization strength, and number of iterations.

Learning Rate

Table 4 shows that the best performance was achieved at a relatively small learning rate of 0.0015.

Table 4: Learning Rate Tuning for SVM

Learning rate	Macro F1 score
0.0003	87.443%
0.0005	88.323%
0.0007	88.661%
0.0010	88.853%
0.0015	**89.069%**

Regularization λ

Moderate regularization gave the best results (Table 5).

Table 5: Regularization Tuning for SVM

Lambda	Macro F1 score
0.01	**89.069%**
0.02	88.553%
0.03	87.127%
0.04	86.460%

Iterations

Since each iteration scans all samples, increasing iterations significantly increases runtime. Table 6 shows that 5 iterations offer an ideal balance.

Table 6: Iteration Tuning for SVM

Iterations	Macro F1 score	Notes
3	88.611%	Fast but underfits
5	**89.069%**	Best
7	89.974%	No significant gain

Final Setting: lr $= 0.0015$, $\lambda = 0.01$, iterations $= 5$.
The Final Macro F1 score is 89.069% **and the total run time is** 5 **seconds**

2.3 K-Nearest Neighbors (Cosine Distance)

KNN performance depends on the number of neighbors k .

Tuning the Value of k

Small k values introduce noise, while large k values oversmooth class boundaries. Table 7 shows that $k = 3$ achieves the best performance.

Table 7: Tuning of k for KNN

k	Macro F1 score
1	95.4896%
3	**95.7066%**
5	95.4155%
7	95.255%

Final Setting: $k = 3$.
The Final Macro F1 score is 95.7066% **and the total run time is** 0.11 **seconds**

2.4 XGBoost Multiclass (Custom Implementation)

The XGBoost implementation is highly sensitive to hyperparameters such as learning rate, number of estimators, max depth.

Learning Rate

A moderate learning rate of 0.40 yielded the best stability and accuracy (Table 8).

Table 8: Learning Rate Tuning for XGBoost

Learning Rate	Macro F1 score
0.10	92.98%
0.20	93.43%
0.4	**94.54%**

Number of Estimators

Increasing the number of boosting rounds improves accuracy until saturation at 40 estimators (Table 9).

Table 9: Estimator Tuning for XGBoost

Estimators	Macro F1 score
20	92.23%
30	93.78%
40	**94.54%**

Max Depth

Depth 3 was found to be the most effective (Table 10).

Table 10: Tree Depth Tuning for XGBoost

Depth	Macro F1 score
2	93.1%
3	**94.54%**
4	92.23

Final Setting: 40 estimators, lr = 0.4, depth = 3.

The Final Macro F1 score is 94.54% **and the total run time is** $5.26min$

3 comparion of algoritms

Table 11: Final Evaluation Results (Macro F1 and Training Time)

Model	**Training Time**	**Macro F1-Score**
Logistic Regression	1.2 minutes	0.9312
SVM (Hinge Loss SGD)	5 seconds	0.89069
KNN (Cosine + Weighted)	0.11 seconds	0.957066
XGBoost (Custom)	5.26 minutes	0.9454

4 Thoughts and Observations

This exercise provided a practical understanding of how classical machine learning models behave when implemented from scratch. Building Logistic Regression, SVM, KNN, and XGBoost manually (using only `NumPy`) helped me appreciate both the mathematical ideas and the computational challenges behind each algorithm.

A major learning outcome was the importance of **data preprocessing**. Standardization significantly improved the stability of Logistic Regression and SVM, while L2-normalization made cosine-based KNN both accurate and computationally efficient. This demonstrated that good preprocessing is often as important as the choice of model.

I also observed clear **trade-offs between model complexity, runtime, and accuracy**.

- Logistic Regression provided stable mid-range accuracy but required more training time.
- SVM trained extremely fast but performed lower due to its linear boundaries.
- KNN required no training and achieved the highest macro F1-score, though prediction is more expensive.
- XGBoost captured non-linear patterns well but needed careful tuning and longer runtime.

Implementing the algorithms manually also highlighted the necessity of **vectorized computation**. Replacing loops with matrix operations dramatically reduced execution time and made it possible to meet the 5-minute runtime requirement. This reinforced the value of writing efficient numerical code.

Overall, this project deepened my intuition about how different ML models learn, make decisions, and trade off flexibility for speed. It also strengthened my understanding of hyperparameter tuning, regularization, and computational optimization—skills that are essential when building real machine learning systems.