Endsem Report

- Eshika Nahata DA24B004

Models Used and System Architecture

MNIST Dataset: We use the standard MNIST dataset of handwritten digits (0–9). Each image is 28×28 pixels (grayscale), flattened into a 784-dimensional vector. Pixel values are normalized to [0,1] by dividing by 255. This produces a balanced 10-class problem (labels 0–9) with roughly equal samples per class.

Preprocessing: All training and validation data are read from CSV files and preprocessed by dropping any extra columns (the “even” flag), normalizing pixel intensities, and flattening each image into a 1×784 vector. The data are also grouped by class for pairwise training (one-vs-one).

Note: My main motivation behind picking 1 model with linear decision boundary (SVM), one model with axis aligned splits (XGBoost) and one model with non linear boundary (KNN) was to introduce diversity for ensembling. Although ensembling over multiple high performance tree based models can be done, there is rarely much difference between each individual model and also the ensemble model.

Support Vector Machine (SVM): I implemented a linear SVM with hinge loss trained by mini-batch gradient descent. Since SVMs are inherently binary, I use a one-vs-one strategy. Concretely, for each pair of classes (example: 0 vs. 1, 0 vs. 2, … 8 vs. 9), I train a binary SVM that separates the two classes. During prediction, each pairwise SVM votes, and the class with the most votes is chosen. This approach trains 45 classifiers, each with hyperparameter C=1.0 and learning rate tuned. The SVM code updates the weight vector Wand bias b per batch using hinge-loss gradients.

K-Nearest Neighbors (KNN): The KNN model stores all training samples and classifies a test image by majority vote of its k nearest neighbors in Euclidean distance. I implemented distance computation via vectorized NumPy operations for efficiency, rather than Python loops. In inference, I split the validation set into batches and for each batch compute distances to all training points. I used k=5 (chosen by experimentation) so each test image’s predicted label is the most frequent label among its 5 nearest neighbors.

XGBoost (Gradient Boosted Trees): I implement a custom multi-class XGBoost classifier. This ensemble builds decision trees sequentially using gradient boosting. At each boosting round, for each class k we grow a regression tree to fit the gradient and hessian of the multiclass log-loss (via a softmax-based formulation). My final chosen hyperparameters include: n_estimators=10 trees, learning rate = 0.1, max depth = 6, L2 regularization lambda=1.0, no gamma pruning, and feature subsampling = 0.8. These relatively modest settings were chosen to limit runtime while

retaining predictive power. By combining 10 “weak” trees iteratively, XGBoost learns a strong multi-class predictor.

Ensemble Methods: I combine the three models using hard voting. I did not use aggregation since KNN and SVM are not interpretable in a probabilistic manner and using distance as a proxy does not necessarily give good results. I explored weighted voting as well but decided not to keep it in the final solution since it is very easy to overfit the weights to validation dataset and hard to make a robust choice for unseen data. It decided it is safest to not assign external weights to the models. Coming to hard voting, each model casts a class vote for each instance, and the final prediction is the majority label. This is implemented by stacking the three prediction arrays and taking the mode.

Hyperparameter Tuning and Results

XGBoost

Accuracy	Max Depth	Learning Rate	Max Features
0.849	4	0.1	25
0.858	4	0.1	50
0.879	4	0.1	100
0.847	5	0.1	25
0.887	5	0.1	50
0.896	5	0.1	100
0.859	6	0.1	25
0.904	6	0.1	50
0.916	6	0.1	100
0.873	4	0.3	25
0.893	4	0.3	50
0.897	4	0.3	100
0.871	5	0.3	25

0.903	5	0.3	50
0.918	5	0.3	100
0.875	6	0.3	25
0.917	6	0.3	50
0.916	6	0.3	100
0.875	4	0.5	25
0.889	4	0.5	50
0.898	4	0.5	100
0.866	5	0.5	24
0.902	5	0.5	50
0.909	5	0.5	100
0.884	6	0.5	25
0.912	6	0.5	50
0.913	6	0.5	100

KNN

Accuracy	K
0.949	1
0.949	2
0.947	3
0.947	4
0.952	5
0.950	6
0.948	7
0.947	8
0.945	9

0.947	10

The chosen hyperparameter values and validation performance are summarized below:

SVM : batch_size=32, learning_rate=0.001, max_iter=100, c= 1.0

KNN : k=5, batch_size = 32

XGBoost : n_estimators = 10, learning_rate = 0.3, max_depth = 5, max_features = 100, lambda_reg = 1, gamma = 0, subsample = 0.8

Model	Accuracy	F1 Score	Training Time (s)
SVM	0.928	0.927	230.088
KNN	0.952	0.952	7.82
XGBoost	0.913	0.912	1382
Hard Voting Ensemble	0.94	0.94	0

Since XGBoost is taking >5mins to train, I will use just the KNN as my primary model as is the best performing model.

System Optimization and Runtime Control

To meet the requirement of training under 5 minutes, the optimization strategies I used are:

- Controlled Complexity: I limited iterative work. For example, SVM uses only 100 total iterations (batch updates) per pair, and XGBoost uses only 10 trees. Keeping these parameters modest reduces training time.

- Batch Processing: Both SVM and KNN are implemented with batching. In SVM training, weights are updated in mini-batches of size 32 (rather than full-batch) to leverage vectorized operations. In KNN prediction, we process validation data in chunks (32 samples at a time) to avoid memory spikes and allow progress reporting.

- Vectorized Operations: Heavy computations are done in NumPy. The KNN distance computation uses matrix dot-products (no Python loops) for speed. In XGBoost, data splits and gradient/hessian sums are computed with efficient array operations. This

minimizes Python overhead.

- Algorithm Choices: Using a linear SVM (rather than a kernel SVM) and a fixed small number of boosting rounds greatly cuts cost. The XGBoost implementation also subsamples features when finding splits to speed up tree growth.

Observations and Insights

- **Model Trade-offs:** The three models show unexpected results. KNN is simple to implement and also the best performer. It is not affected by the curse of dimensionality in the case due to the pixel values lying in a low dimensional manifold. It can capture complex decision boundaries and in the case of MNIST there is good separation between classes as well (it is a low-bias, high-variance learner). The linear SVM is fast at inference (just a dot-product per classifier) and performs well on this dataset as well. It is able to find a high dimensional decision boundary for each one-vs-one classifier. XGBoost has boosted trees that can model nonlinearity (lower bias) but we control variance via regularization and few trees. In practice, XGBoost did not give the best accuracy due to limitations on training time and computation. If parallelized and trained for longer, this model would prove to be the best performing one.

- **Ensemble Effectiveness:** The voting ensemble slightly improved stability but still ended up performing poorer than the KNN. By taking the majority vote of SVM, KNN, and XGBoost, errors unique to one model are often outvoted by the others which improved the accuracy compared to just SVM or just XGBoost. As theory suggests, stacking can achieve accuracy at least as good as the best base model and often slightly better by combining strengths.

- **Bias-Variance Considerations:** Qualitatively, KNN (k=5) has a good bias variance tradeoff. The linear SVM has relatively higher bias (it imposes linear decision boundaries in high-D space) but lower variance. XGBoost, with deep trees and many rounds, can have very low bias but risks overfitting (high variance); we mitigated that through regularization and limiting tree depth. The ensemble methods reduced variance: voting averages out idiosyncratic errors, and stacking further reduces bias by learning to correct systematic mistakes. Overall, the bias-variance trade-off was managed by choosing moderate hyperparameters and by ensembling.

- **Common Confusions:** Examination of errors (via confusion matrices) revealed that the models often confuse similar-looking digits. For example, 3’s vs 5’s and 4’s vs 9’s were frequent confusions. Digit '8' is also often misclassified (its loops can resemble 3 or 6). These patterns are typical for handwritten digits. The confusion matrix highlights that certain classes are harder for all models, indicating intrinsic ambiguity in those digits.