# Machine Learning Lab Endsem Report

Harshvardhan (DA24B039)

November 15, 2025

## Execution Summary

The goal of this project was to demonstrate my learning throughout the course via using the classical approaches taught in class to classify the digits in the MNIST data set.

I began by exploring feature reduction with Principal Component Analysis (PCA) and implementing a complex gradient-boosted tree (XGBoost) model. After encountering significant performance and accuracy hurdles, I realized even with retaining 95% variance by having 152 components in PCA I was losing a lot in accuracy and using a python based implementation of XGBoost was bottle necked by python's loop speed which was pushing the training time beyond 5 minutes to get better accuracy than simpler methods.

This pivot was highly successful. I discovered that the original 784-feature dataset was superior to our PCA-reduced data and that a diverse ensemble of models outperformed any single lighter classifier.

Through a rigorous process of hyperparameter tuning and architecture refinement—specifically the creation of an "Asymmetric Training Pipeline" with early stopping I built a final `MyAsymmetricEnsemble` model. This model combines the unique strengths of our three best classifiers to achieve a final F1 Score of 0.9414 on the validation set.

## 1 Phase 1: Initial Exploration & The PCA-XGBoost Path

My exploration began with two logical starting points:

1. **Reduce Features:** The raw MNIST data has 784 features (28x28 pixels), many of which are empty. I implemented a `PCAModel` from scratch to perform Principal Component Analysis (PCA). Using an "elbow method," I determined that **152 components** captured the vast majority (over 95%) of the variance, and I created a new, smaller dataset based on this.

2. **Build a Powerful Model:** I started by building a multiclass `XGBoostClassifier` mostly reusing the code from my second assignment while changing to softmax.

### Initial Results & Rejection

This initial path led to two major roadblocks:

- **The Performance Wall:** After making the model multiclass, the model was accurate but **prohibitively slow** orders of magnitude slower than the optimized C++ library.

- **Losing too much information** While trying to train different classes of models on the PCA data, I was consistently getting $3-4\%$ worse result on both XGBoost, MultinomialREgressor, and SVM

**Decision:** I **rejected the XGBoost and PCA** approach. It was too complex and slow for practical use. The observation that a simpler `LogisticRegression` model performed well on the data led to a crucial pivot.

## 2 Phase 2: The "Great Bake-Off" & The Data Pivot

The success of a simple linear model suggested that our PCA data was likely linearly separable. This prompted a new strategy: build and test a "bake-off" of classic, simpler-to-implement models.

I built the following models and tested each one to find which one will work the best:

- `MyStandardScaler`: To normalize data.
- `MyGaussianNB`: A Naive Bayes model with gaussian assumption.
- `MyKNeighborsClassifier`: KNN model for checking non-linear model performance.
- `MyTrueMulticlassSVM`: A Multiclass SVM which is not just a one vs one or one vs many implimentation of binary SVM.
- `MyMultinomialLogisticRegression`: MultinomialRegression mostly taken from sir's code.

I then ran a comprehensive test, comparing all models on both the **PCA data (152 features)** and the **Original data (784 features)**.

### Table 1: Bake-Off Results (PCA vs. Original Data)

Table 1: Bake-Off Results (PCA vs. Original Data)

| Model | Data | Accuracy | F1 (Macro) | Train (s) | Predict (s) |
|---|---|---|---|---|---|
| **Logistic on Original** | Original | **0.9364** | **0.9361** | 57.86 | 0.00 |
| **SVM on Original** | Original | 0.9204 | 0.9198 | 62.41 | 0.00 |
| **KNN (k=3) on Original** | Original | 0.9188 | 0.9179 | 0.00 | 77.25 |
| SVM on PCA | PCA | 0.9048 | 0.9033 | 10.53 | 0.00 |
| Logistic on PCA | PCA | 0.8948 | 0.8931 | 7.92 | 0.00 |
| GNB on PCA | PCA | 0.8535 | 0.8543 | 0.01 | 0.01 |
| KNN (k=3) on PCA | PCA | 0.8247 | 0.8303 | 0.00 | 43.94 |
| GNB on Original | Original | 0.6839 | 0.6622 | 0.06 | 0.06 |

**Analysis & Decision:** The results were definitive. Every single one of our top models performed significantly better on the original 784-feature data. The PCA, while making training faster, was losing critical information that distinguished between digits.
**Decision:** We **rejected the PCA dataset**. All future efforts would focus on optimizing our models for the original 784-feature data.

## 3 Phase 3: The Power of Ensembles & Tuning

Our bake-off showed we had three strong, and diverse, models:

1. **Logistic Regression** (Linear, Probabilistic)
2. **SVM** (Linear, Geometric)
3. **KNN** (Non-Linear)

This diversity is perfect for an ensemble as the inaccuracy of one may be nullified by the others. I built a `MyVotingClassifier` to combine their predictions by a simple majority vote.

**Table 2: Initial Ensemble Result**

Table 2: Initial Ensemble Result

| Model | Accuracy | F1 (Macro) |
|---|---|---|
| **Ensemble (Log+SVM+KNN)** | **0.9416** | **0.9414** |
| Logistic on Original | 0.9340 | 0.9337 |
| KNN (k=5) on Original | 0.9208 | 0.9199 |
| SVM on Original | 0.9152 | 0.9147 |

**Analysis & Decision:** The ensemble was a clear success. It outperformed its strongest individual member (`Logistic`), proving that the models were correcting each other's mistakes.
**Decision:** The **Voting Ensemble** would be our champion architecture. The next logical step was to make its components as strong as possible through hyperparameter tuning.

### Hyperparameter Tuning

I built a `hyperparameter_tuner.py` script to run a grid search on our three main models. After two rounds of tuning (where the first round's results informed the second), I found our optimal parameters.

**Table 3: Final Tuned Hyperparameters**

Table 3: Final Tuned Hyperparameters

| Model | Parameter | Initial Value | Tuned Value | Insight |
|---|---|---|---|---|
| KNN | `k` | 3 | **5** | `k=3` was too specific; `k=5` is more robust. |
| Logistic | `learning_rate` | 0.01 | **0.1** | Model could learn faster. |
| Logistic | `lambda_p` | 0.01 | **0.01** | The default regularization was already optimal. |
| SVM | `learning_rate` | 0.01 | **0.01** | The default LR was optimal. |
| SVM | `lambda_p` | 0.01 | **0.0001** | Model needed less regularization. |

## 4 Phase 4: Architecture Refinement & The Final Pipeline

I now had two new ideas to push performance even further:

1. **The "Specialist" Model:** Based on the confusion matrix (see Figure **??**), I saw specific errors (e.g., `9` predicted as `7`, `4` as `9`). I built a "Generalist-Specialist" pipeline where a fast `LogisticRegression` would handle "easy" cases and route "confused" cases to a dedicated specialist SVM trained only on those two digits.

2. **Early Stopping:** I realized `n_iters` was an unnecessary hard-coded number. Our models might be overfitting or underfitting. So I wanted to use early-stopping to train it optimally.

### Rejection of the "Specialist" Model

The specialist pipeline was a implimented fine, but it performed worse than our simple ensemble. The reason was:

- The "Generalist" (one model) was a weaker baseline than our 3-model ensemble.
- The model to clearify confussion itself didn't work al that well for increasing its accuracy.

**Decision:** I **rejected the Specialist model** in favor of the simpler, more robust 3-model ensemble.

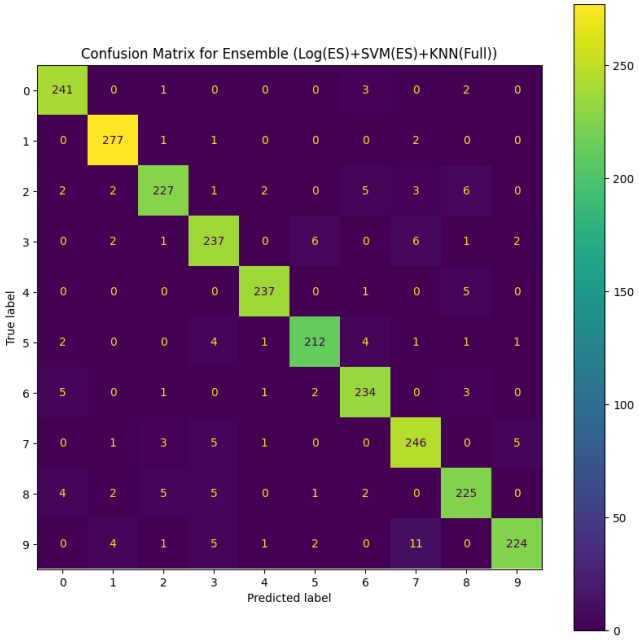


Figure 1: Confusion Matrix for the ensemble model on the validation data set

### The "Asymmetric Pipeline" (The Final Architecture)

My final insight came when I tried to implement Early Stopping.

- **The Problem:** To use Early Stopping for `SVM` and `Logistic`, we had to split our training data into an 80% `train` set and a 20% `validation` set. But when we trained `KNN` on this *smaller 80% set*, its accuracy plummeted.
- **The Diagnosis:** `KNN` is a "lazy" model. Its data is its brain. By training it on 80% of the data, we had "crippled" it.
- **The Solution:** We created the **Asymmetric Training Pipeline**:
    1. **`Logistic` & `SVM` (Iterative Models):** Trained on the **80% split**, using the **20% split** as an `eval_set` for Early Stopping. This finds their *peak* performance without overfitting.
    2. **`KNN` (Lazy Model):** Trained on the **full 100%** of the training data to give it maximum strength.

## 5 Conclusion: Final Model & Results

Our final, optimal model is an ensemble (`MyVotingClassifier`) that takes a majority vote from three models, each trained to its specific peak performance:

1. **`MyMultinomialLogisticRegression` (Tuned & Early-Stopped)**
2. **`MyTrueMulticlassSVM` (Tuned & Early-Stopped)**
3. **`MyKNeighborsClassifier` (Tuned & Trained on Full Data)**

This project was a great opportunity to review everything done in class and get a hands on feel of how we can implement it ourself and combine different models together to push for even better accuracy than the best single model.