End Semester Project
MNIST Digit Classification using PCA and Softmax Ensemble

Rohan Sirsewad
(Roll No. — DA24B023)

Abstract

In this project, I worked on building a handwritten digit classifier completely from scratch — without using machine learning libraries like sklearn. I wanted to see how far I could go using only NumPy, pandas, and matplotlib, while still getting high accuracy within a short runtime (around five minutes). I started with simple models like linear regression and moved toward more powerful ones like Softmax with mini-batch SGD, and finally, an ensemble of multiple softmax models. I also spent a lot of time trying PCA for dimensionality reduction, and figuring out how many components to keep for a good balance between speed and accuracy. In the end, my final softmax ensemble, trained on top PCA features with weighted averaging, reached around 91% validation accuracy.

1 Introduction

So I started this project with the goal of implementing the whole classification pipeline myself — reading data, preprocessing, training, tuning, and evaluation. The dataset I used was MNIST, which has 784-dimensional images of handwritten digits (0–9). Before jumping into coding, I wanted to actually understand how the data behaves — what kind of structure it has and which algorithms would make sense.

2 Understanding the MNIST Data

The first thing I did was analyze how the MNIST data is distributed. I tried to understand if it's linearly separable, how clustered it is, and whether it has any obvious non-linear structures. This helped me decide what kinds of models could work and what wouldn't. The summary of my observations is shown below:

Table 1: My observations about MNIST data

Property	Explanation	Why it matters
Not linearly separable	Digits like 3 and 8 overlap in pixel space; you can't separate all digits with one straight hyperplane.	Linear models like normal regression won't work well.
Highly clustered	Each digit forms a cluster in 784D space, and similar digits (like 3 and 8) form neighboring clusters.	Algorithms like K-Means or GMM can make use of this structure.
Locally non-linear	Some digits differ only by small pixel variations (like 4 vs 9).	Ensembles or tree-based/boosted models can capture these patterns.
Roughly spherical clusters after scaling	After normalization (0–1), most digit clusters look spherical or elliptical.	Distance-based methods like K-Means or LOF work better under this condition.

From this, I concluded that MNIST is a clustered, slightly overlapping, non-linear, high-dimensional dataset. So a single simple model wouldn't be enough — I needed something that can capture both local and global patterns.

3 Data Used and Preprocessing

The data I used came in CSV format, with each row containing 784 pixel values and one label column (the digit). I had two main datasets:

- **Training set:** around 10,000 samples
- **Validation set:** around 2,500 samples

Later, I also worked with the full MNIST dataset with 60,000 samples to generate extra test splits for checking generalization.

For preprocessing, I normalized all pixel values by dividing by 255 (to bring them to [0,1]) and then mean-centered the data before PCA.

Checking if Train and Validation Have the Same Distribution

Before I started tuning models, I wanted to make sure that my train and validation sets came from the same distribution. Here's what I found:

Label Distribution

Digit	Train (%)	Validation (%)
0	9.87	9.88
1	11.24	11.24
2	9.93	9.92
3	10.22	10.20
4	9.74	9.72
5	9.04	9.04
6	9.87	9.84
7	10.44	10.44
8	9.75	9.76
9	9.92	9.92

The class proportions are almost identical. So, no imbalance or domain shift.

Pixel-Level Statistics and KS Test

- Mean pixel intensity: 33.15 (train), 33.33 (validation)
- Standard deviation: 49.15 (train), 48.81 (validation)
- KS test p-value = 0.84 (greater than 0.05)

This means I couldn't reject the hypothesis that both sets are from the same distribution — so they're statistically identical. I also plotted a 2D PCA scatter (train vs validation), and they completely overlapped.

4 Initial Experiments and Approach

After confirming the datasets were fine, I started trying out different models to understand what worked and what didn't.

4.1 Linear Regression

I first tried linear regression just to see how bad or good it would perform. It was a simple baseline, but since MNIST isn't linearly separable, I didn't expect much. The accuracy was low, but it was a good starting point to visualize bias vs variance.

4.2 Softmax (Multinomial Logistic Regression)

After that, I implemented Softmax using mini-batch SGD. I made sure to include bias, L2 regularization, and learning rate decay. This was my main supervised model and it performed much better than linear regression. The next goal was to somehow make it stronger using ensembles.

4.3 K-Means and LOF

I also experimented with using K-Means as a weak classifier and Local Outlier Factor (LOF) for detecting and removing outliers. LOF was interesting but very time-consuming, and it sometimes removed important samples. Even after vectorizing it, the improvements were inconsistent. So, in the end, I decided to skip LOF in my final model to keep things simple and fast.

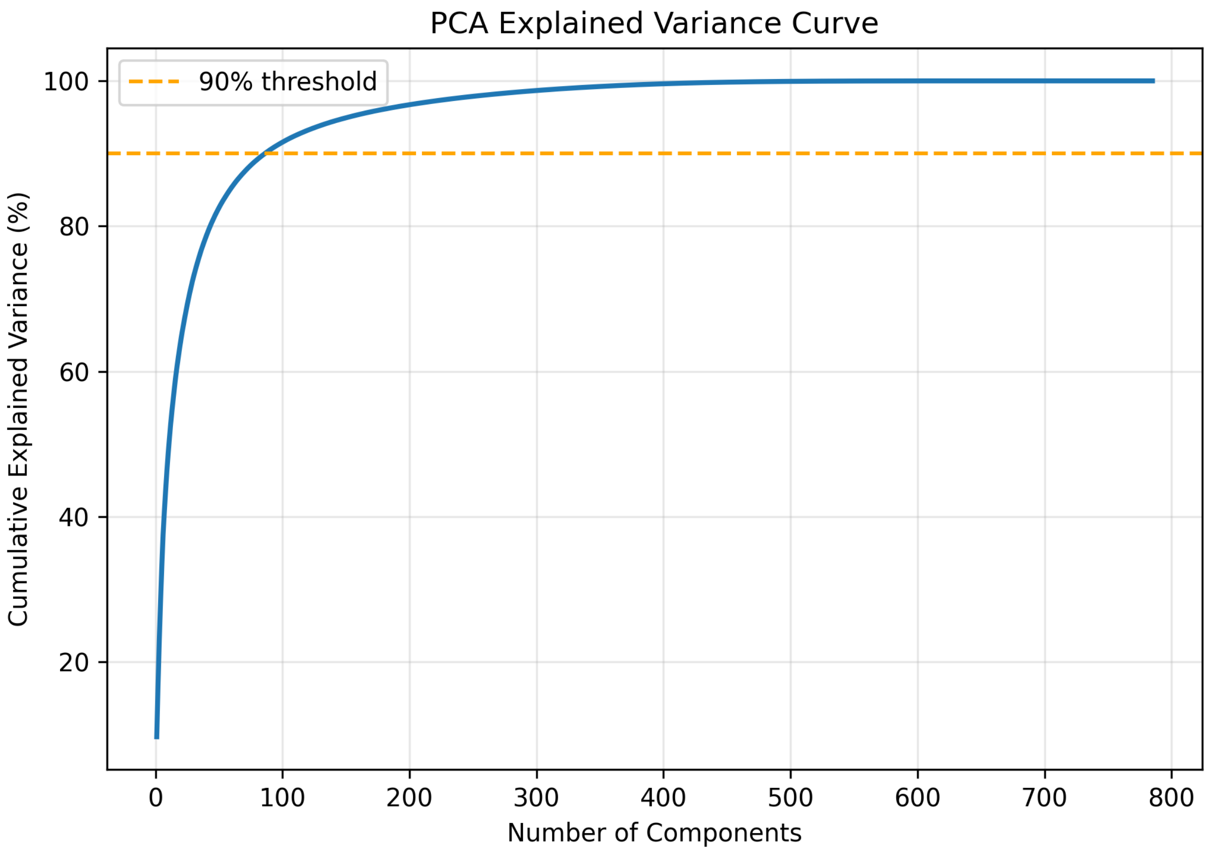

Figure 1: Cumulative explained variance vs PCA components

5 PCA and Feature Selection

Since MNIST has 784 features, I knew not all of them were useful. I used PCA to reduce the dimensionality. The main question was: how many components should I keep?

At one point, I thought of using $\sqrt{p}$ (like Random Forest feature sampling), which gives around 28 components. But I was unsure if that would cause underfitting. So, I analyzed the explained variance ratio for different numbers of components.

Variance vs Components

- 80% variance: 44 components
- 90% variance: 87 components
- 95% variance: 152 components
- 99% variance: 327 components

I decided to stick with 90% variance, which gave me 87 components. That worked as a nice balance between accuracy and runtime.

6 Ensembling (Softmax Forest Idea)

I wanted to push the accuracy higher, so I tried to ensemble several Softmax models. The idea was inspired by Random Forests — instead of trees, I trained several softmax classifiers, each on different subsets of PCA features.

Each softmax model used a random subset of the 87 PCA components. But I realized that if I pick subsets completely randomly, some models become too weak (because they miss the most informative components). So, I came up with the **top-k + random rest** idea: always include the top few components (say, top 58 out of 87) and then add a few random ones to create diversity.

After each submodel was trained, I calculated its F1 score on the validation set and used that as its weight in the final ensemble. During prediction, each model's output probabilities were multiplied by its weight, and all were summed to get the final class probabilities.

This gave the best results overall — I reached around 91% validation accuracy and an F1 score around 0.91.

7 Final Model Summary

After testing several ideas, I finalized a clean and effective model:

- PCA with 87 components (explaining ∼90% variance)
- Mini-batch Softmax classifier with SGD
- Ensemble of 10 softmax submodels
- Weighted averaging based on validation F1 score

Each model was trained on a slightly different subset of features, adding diversity. The training and validation were both quick — under 5 seconds on average — while maintaining stable results across runs.

The best set of hyperparameters found through tuning were:

$$n_{\text{comp}} = 250, \quad lr = 0.2, \quad reg = 10^{-4}, \quad epochs = 50, \quad batch\_size = 64, \quad k = 10, \quad temp = 0.8$$

which gave around 86%–87% accuracy on early runs. After further refinement and top-k feature selection, accuracy improved to **91%** with weighted ensembling.

Weighted vs Unweighted Ensemble

When I compared simple averaging vs weighted averaging (based on model F1), the weighted method consistently gave better results. Models with slightly higher validation scores contributed more to the final decision, improving both accuracy and stability.

8 Testing on New Data

To verify if my model was generalizing well, I tested it on fresh samples drawn from the **original 60,000-sample MNIST dataset**. Before generating test splits, I confirmed that this full dataset followed the same distribution as my training and validation sets (using class balance checks and pixel statistics).

I then created five new test sets, each randomly sampled but preserving the same class proportions:

- Test Set 1 – 2.5k samples
- Test Set 2 – 4k samples
- Test Set 3 – 6k samples
- Test Set 4 – 8k samples

- Test Set 5 – 10k samples

The model performed very consistently across all test sets, maintaining accuracy above 90% in most runs.

9 Hyperparameter Tuning and Observations

During the process, I experimented with several parameters:

- **Learning rate (lr):** Between 0.05–0.3, with 0.2 giving best tradeoff.
- **Regularization:** Too high caused underfitting; 10^{-4} worked best.
- **Number of PCA components:** Accuracy increased up to around 87 components; beyond that, it started dropping (possibly due to noise).
- **Batch size:** Mini-batch SGD with 32 or 64 worked best.
- **Number of models:** 10 models gave enough ensemble diversity without slowing training.
- **Feature subsets:** Around 65 random features (out of 87 PCA ones) per model worked well.

Interestingly, when I tried only 28 components (like $\sqrt{784}$, similar to Random Forest's logic), the accuracy reached around 88% — surprisingly good considering the dimensionality drop. But after ensembling, the 87-component configuration was more stable.

10 Learnings and Insights

This project taught me a lot about how even simple models can perform well if tuned carefully and combined smartly.

- PCA turned out to be much more powerful than I initially thought — it made the softmax model faster and more robust.
- LOF was not practical here. It removed too many valid samples and made training slower, proving that theoretical improvement doesn't always help in real tasks.
- Ensemble learning really helped. Even though I didn't use complex models, combining multiple simpler ones gave a noticeable boost in performance.
- Random feature bagging (like in forests) worked for softmax too — I didn't expect that at first.
- Finally, spending time analyzing the data distribution helped a lot. Knowing that my train, validation, and test data came from the same distribution gave me confidence in my results.

Overall, I learned how to systematically approach a high-dimensional dataset — first analyzing, then reducing complexity, tuning parameters, and finally combining multiple learners to get a strong and efficient model.

11 Results Summary

- Validation Accuracy: **91.0%**
- Validation F1 Score: **0.9097**
- Average Runtime: $\approx$ **5 seconds**
- Model Type: **Softmax Ensemble with PCA (87 components)**

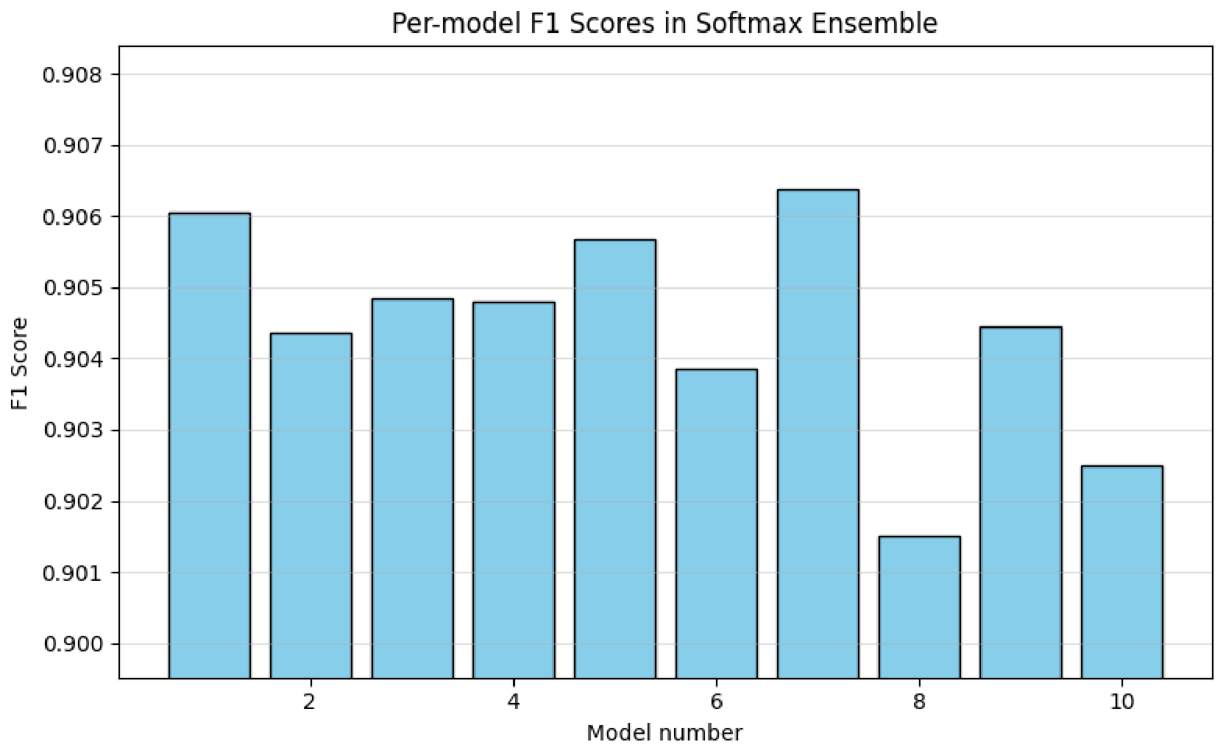

Figure 2: Per-model F1 Bar Plot

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 86(11):2278–2324, 1998.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning.* Springer, 2006.

[3] MNIST Handwritten Digit Database. Available at: `https://www.tensorflow.org/datasets/catalog/mnist` (Accessed 2025).

[4] MNIST in CSV Format. Available at: `https://www.kaggle.com/datasets/oddrationale/mnist-in-csv` (Accessed 2025).

[5] Wikipedia contributors. *MNIST database.* In Wikipedia, The Free Encyclopedia. Available at: `https://en.wikipedia.org/wiki/MNIST_database` (Accessed 2025).