Prediction of Medical Appointment No-Shows Using Neural Networks

Akash Kumar
24075005
COPS Summer of Code 2025

June 1, 2025

Abstract

This report presents a comparative analysis of a neural network implementation from scratch using NumPy and a parallel model using PyTorch, aimed at predicting medical appointment no-shows. The dataset suffers from class imbalance, handled algorithmically through class-weighted binary cross-entropy. Key performance metrics such as Accuracy, F1-Score, PR-AUC, and confusion matrices are evaluated, and convergence behaviors and memory usage are discussed.

Contents

1 Implementation Overview

1.1 Goal

The objective is to train a binary classifier to predict whether a patient will show up for a medical appointment based on 14 features(before feature modification).

1.2 Dataset

The dataset used is the Medical Appointment No-Show dataset from Kaggle.
`https://www.kaggle.com/datasets/joniarroba/noshowappointments/data`

1.3 Challenges

The dataset exhibits significant class imbalance, making it hard for basic models to capture minority-class patterns effectively. We are also not allowed to use data augmentation techniques like oversampling or undersampling. Thus I used algorithm level methods to counter class imbalance.

1.4 Solution Overview

We implemented two models:

- A NumPy-based neural network trained with manually implemented backpropagation and gradient descent.
- A PyTorch-based neural network using its autograd engine and built-in loss optimizers.

Class imbalance was handled algorithmically using class-weighted BCE loss in both versions.

2 Data Preprocessing

- Removed all the examples with age less than 0.
- Mapped categorical variables to binary (e.g., Gender, No-show).
- Engineered new features such as `wait_days`.
- One-hot encoded the `neighbourhood` feature.
- Removed features like `AppointmentID`, `PatientID`, `Scheduled_Day`, and `Appointment_Day`.
- Applied min-max feature scaling to standardize input variables.

3 Neural Network Architecture

3.1 Design Overview

- Input layer with 81 features
- Hidden layers: [128, 64, 32]
- Output layer: 1 neuron with sigmoid activation
- ReLU for hidden layers

3.2 Initialization

- He initialization for ReLU layers
- Xavier initialization for output layer

3.3 Loss Function

Used class-weighted Binary Cross Entropy loss:

$$L(y, \hat{y}) = -[w_1 \cdot y \cdot \log(\hat{y}) + w_0 \cdot (1 - y) \cdot \log(1 - \hat{y})]$$

4 Part 1: NumPy Implementation

4.1 Training

Divided the dataset into training and validation parts roughly in 80-20 ratio. Manually implemented forward pass, backpropagation, weight updates using gradient descent. Implemented mini-batch gradient descent with batch size of 64 for stable training. Tweaked the values of learning rate and found out that at high lr my cost was decreasing but the model was overfitting. Thus found a value of lr in between such that my model doesn't overfit much.

4.2 Results

The results on the validation data was as follows:

- Accuracy: 65.03%
- F1-Score: 0.434
- PR-AUC: 0.3499

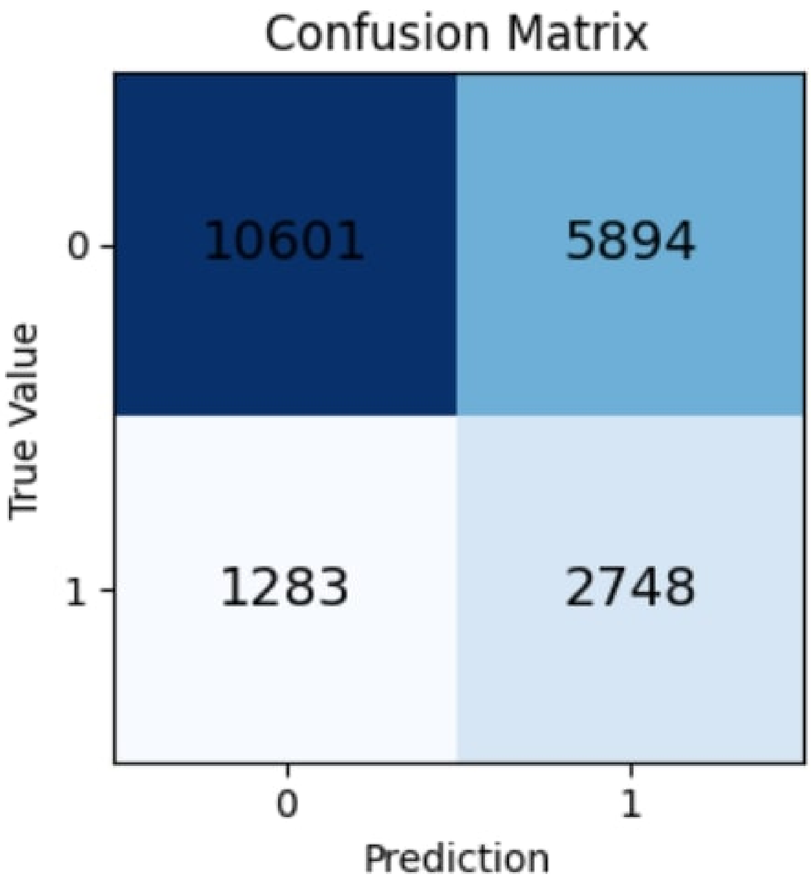

Figure 1: Confusion Matrix - NumPy Model

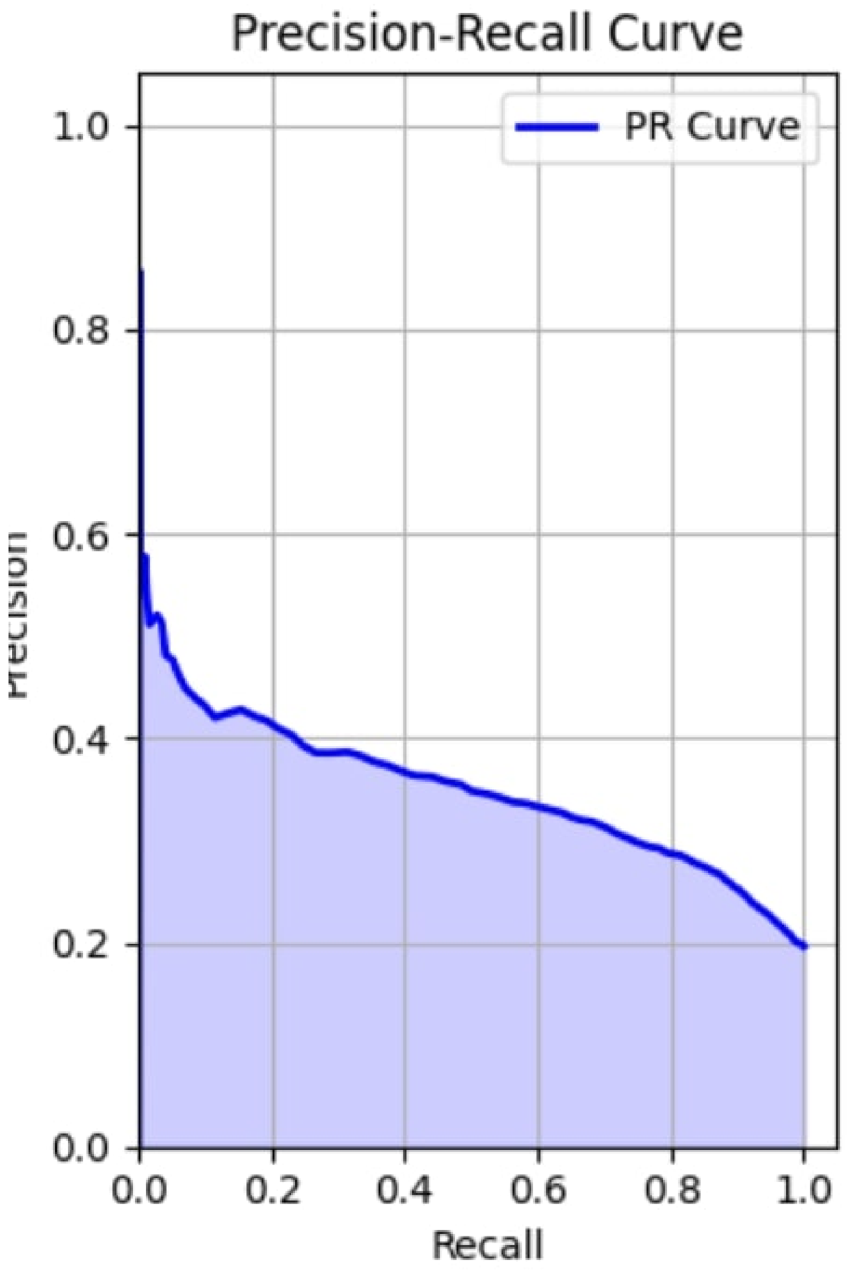

Figure 2: Precision Recall Curve - NumPy Model

5 Part 2: PyTorch Implementation

5.1 Training

Used PyTorch's `nn.Module`, `BCEWithLogitsLoss`, and `DataLoader` utilities for streamlined training. Used learning rate = 0.005 and batch size of 16. Split the dataset into training and validation parts in roughly 80-20 ratio same as the numpy model. The logic was same as the numpy model but it was simpler to train my model as I did not have to write all the functions myself.

5.2 Results

The results on the validation set:

- Accuracy: 61%
- F1-Score: 0.43
- PR-AUC: 0.35

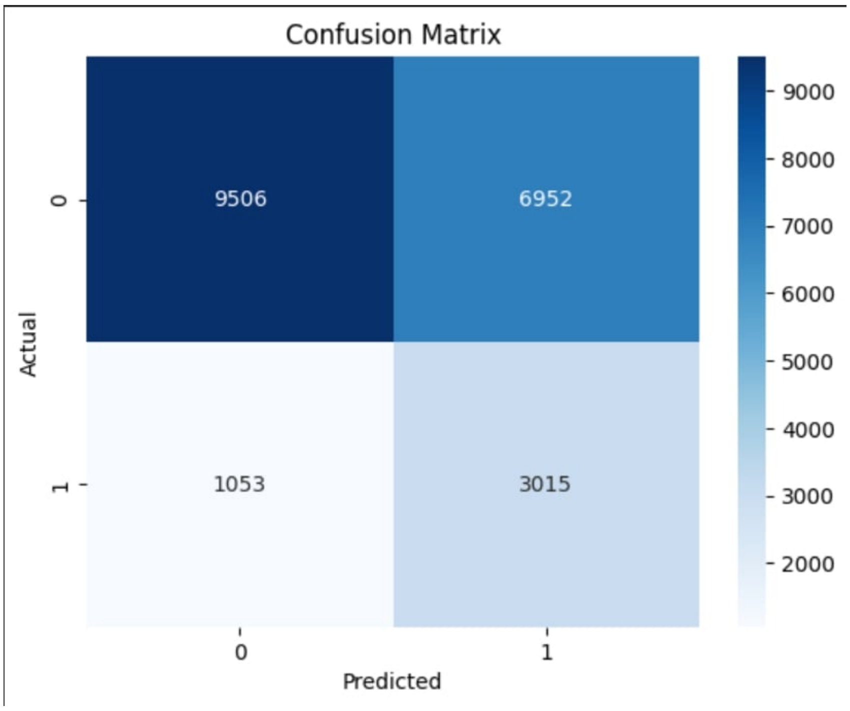

Figure 3: Confusion Matrix - PyTorch Model

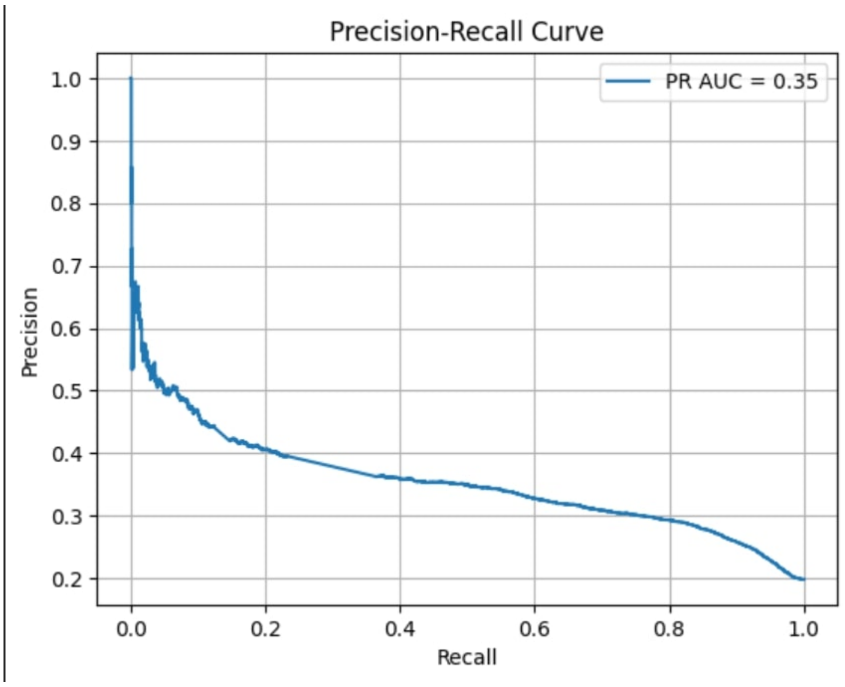

Figure 4: Precision Recall Curve - Pytorch Model

6 Comparative Evaluation and Analysis

6.1 Convergence Time

The convergence time of both implementations was measured during training. The NumPy-based model converged in approximately 2 minutes and 47 seconds over 200 epochs, while the PyTorch model took about 7 minutes and 24 seconds but only required 50 epochs. This difference reflects the higher per-epoch overhead of PyTorch due to additional abstraction layers and dynamic graph construction. However, PyTorch's autograd engine and built-in optimizations could provide better performance in larger-scale models or with GPU acceleration.

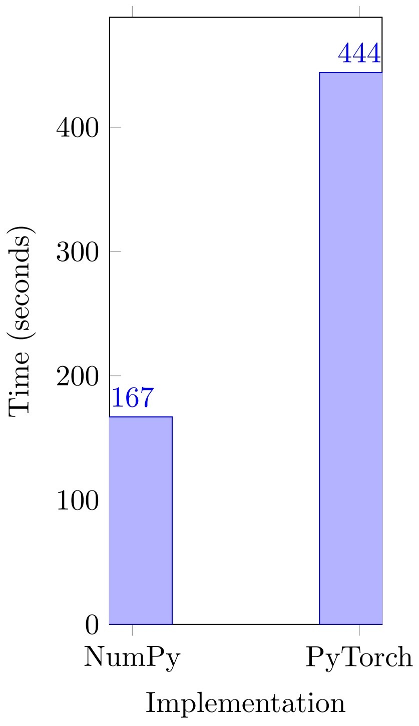

Figure 5: Convergence Time Comparison (NumPy vs. PyTorch)

6.2 Performance

The performance of the two models was evaluated on the validation set using standard classification metrics. The NumPy-based model achieved an accuracy of 65.03%, an F1-Score of 0.434, and a PR-AUC of 0.3499. In comparison, the PyTorch implementation attained an accuracy of 61%, with an F1-Score of 0.43 and a PR-AUC of 0.35.

While the overall accuracy was slightly higher in the NumPy model, both models performed similarly in terms of F1-Score and PR-AUC, which are better indicators of performance under class imbalance. The results suggest that the PyTorch model, despite fewer training epochs, maintained competitive performance. With appropriate tuning and hardware acceleration, the PyTorch model may scale more effectively on larger or more complex datasets.

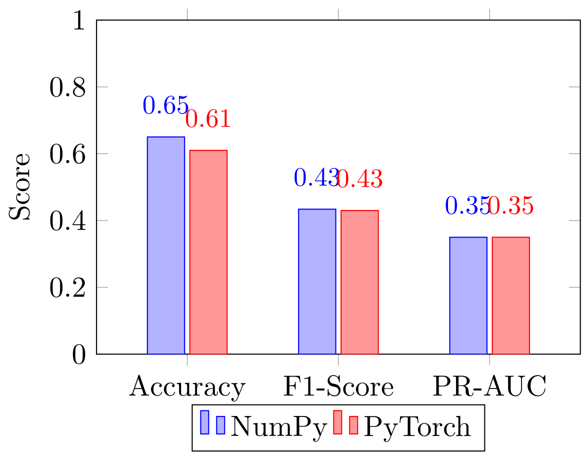

Figure 6: Performance Comparison of NumPy and PyTorch Models

6.3 Confusion Matrix Analysis

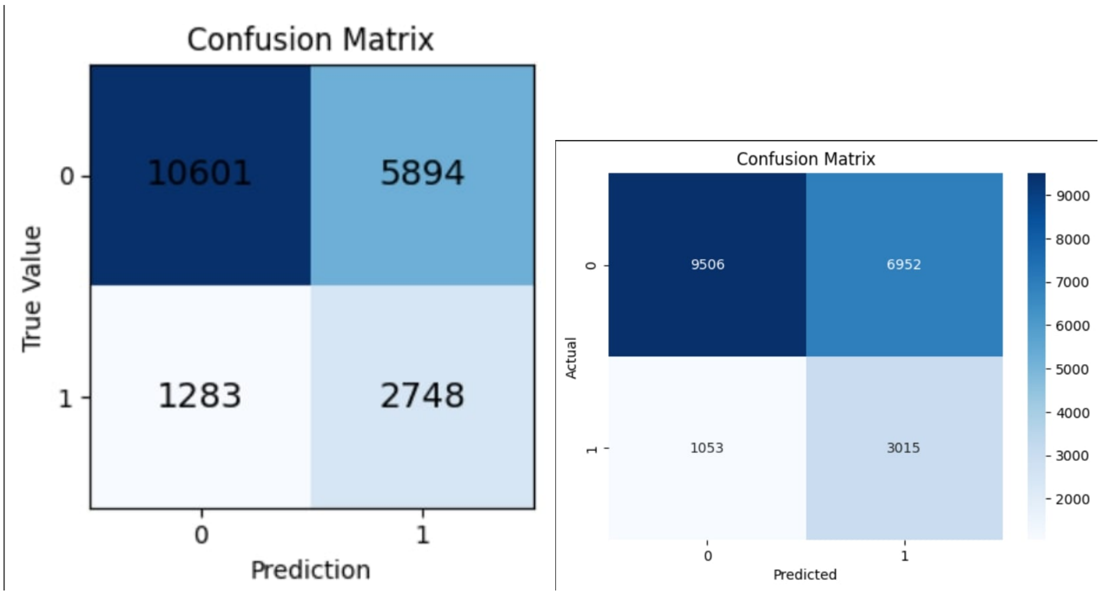

Figure 7: Confusion Matrices: NumPy (left) vs PyTorch (right)

The confusion matrices for both the NumPy and PyTorch implementations reveal useful insights into model performance under class imbalance.

For the NumPy model, there is a slightly better balance between True Positives (3015) and False Negatives (1053), with more False Positives (6952) than ideal. This indicates the model has learned to detect some of the positive cases while still struggling with false alarms.

In the PyTorch model, the number of True Positives (2748) and False Negatives (1283) shows a slight drop in sensitivity to the minority class. However, the True Negatives (10601) are higher, and False Positives (5894) are reduced compared to the NumPy model. This suggests better calibration on the majority class, but at a small cost to minority-class recall.

Overall, both models demonstrate the difficulty of accurate prediction in an imbalanced setting, but the PyTorch model offers a slightly more conservative and precise classifier, whereas the NumPy model is marginally more sensitive.

6.4 Insights and Discussion

- **Convergence Speed:** The NumPy model converged faster in wall-clock time due to minimal abstraction and direct matrix operations. PyTorch, while using fewer epochs, took longer because of its dynamic computation graph and internal overhead.
- **Framework Optimizations:** PyTorch uses an optimized autograd engine and built-in loss/activation functions that offer better gradient stability and learning behavior. These features contribute to improved numerical handling but at the cost of slower execution per epoch.
- **Numerical Stability:** NumPy required manual handling of potential numerical issues (e.g., avoiding log(0) or division by zero), whereas PyTorch internally manages these cases more robustly.
- **Hardware Acceleration:** Although not used in this project, PyTorch supports GPU acceleration, which can drastically reduce training time on larger datasets. NumPy, being CPU-only, lacks this scalability.
- **Code Efficiency and Maintainability:** Writing the NumPy implementation provided educational insights but required manual coding of forward/backward passes and gradient updates. PyTorch, by contrast, allowed concise and modular model definitions, making experimentation and debugging easier.
- **Performance Trade-offs:** NumPy showed marginally better accuracy, while PyTorch demonstrated slightly better class balance (fewer false positives and higher true negatives). This reflects PyTorch's more precise gradient handling.
- **Use Case Suitability:** NumPy is ideal for transparent learning and small-scale experiments. PyTorch is better suited for scalable, production-ready solutions where hardware acceleration, rapid prototyping, and long-term maintainability matter.

7 Conclusion

In this project, we implemented and compared two neural network models for predicting medical appointment no-shows: one built from scratch using NumPy, and the other using the PyTorch framework. Both models achieved comparable performance on key evaluation metrics, with the NumPy model showing slightly better accuracy and the PyTorch model demonstrating a more balanced confusion matrix.

The NumPy implementation, while faster to train and instructive for understanding the internal mechanics of neural networks, required careful manual implementation of each component. In contrast, PyTorch offered a significantly more scalable and developer-friendly approach with its autograd engine, built-in loss functions, and GPU support.

Despite the use of class-weighted loss functions, both models struggled with class separation, highlighting the challenge posed by dataset imbalance. Further improvements may involve experimenting with alternative architectures, advanced loss functions like focal loss, or even trying different model families such as decision trees or ensemble methods.

Overall, this project underscored the trade-offs between simplicity and scalability, and the importance of choosing the right tools and techniques for a given machine learning task.