Error Bar Detection Technical Report

Intern Assignment Submission

1 Introduction

This report details the development of an automated system for detecting error bars in scientific plots. The solution is implemented as a modular Python framework leveraging Deep Learning (PyTorch) for regression and Matplotlib for high-fidelity synthetic data generation. The project is structured into distinct pipelines for data generation, cleaning, model training, and inference, ensuring a robust and reproducible workflow.

2 Synthetic Dataset Generation Strategy

To create a robust dataset for error bar detection, I implemented a comprehensive generation pipeline using `matplotlib`. The strategy focuses on maximizing variability to ensure the model generalizes well to unseen plots.

2.1 Pipeline Overview

The generation script (`src/generator/generate.py`) produces synthetic plot images paired with JSON annotations. The pipeline generates three primary plot types:

- **Line Graphs:** Varying numbers of series, markers, line styles, and colors.
- **Bar Charts:** Grouped bars with optional hatch patterns and varying edge widths.
- **Box Plots:** Generated from statistical distributions (Normal, Lognormal, Gamma) to create realistic interquartile ranges and whiskers.

2.2 Domain Randomization

To prevent overfitting to a specific style, the following parameters are randomized for each sample:

- **Geometry:** Figure dimensions ($W \times H$) and DPI (72-200) are varied to alter the pixel-to-unit ratio.
- **Scales:** Both linear and logarithmic Y-axes are simulated.
- **Error Bar Characteristics:** The pipeline simulates symmetric bars, asymmetric bars, and "ghost" bars (zero length) to represent points with no error margin.
- **Artifacts:** Random occlusions (rectangles) and grid style variations are applied to mimic real-world noise.
- **Aesthetics:** To match publication standards, the generator utilizes specific color palettes (e.g., Tableau, Prism, Excel default) rather than generic colors.

2.3 Quality Assurance Pipeline

To ensure the model is not trained on invalid data, a strict cleaning pipeline (`src/cleaning/cleaner.py`) validates all generated samples. This module utilizes **Pydantic models** (`src/common/models.py`) to enforce strict schema validation. It automatically filters out artifacts such as error bars extending beyond the image boundary or exceeding the maximum learnable height, ensuring the training set represents physically possible scenarios.

2.4 Dataset Statistics

The final dataset was balanced to ensure the model sees a healthy ratio of positive (Real Bars) and negative (Missing Bars) samples. The table below summarizes the distribution of the Synthetic data versus the Real validation target.

Metric	Synthetic Stats	Real Stats
Total Points	134,086	13,468
Missing Bars	68,151 (50.8%)	6,042 (44.9%)
Real Bars	65,935 (49.2%)	7,426 (55.1%)
- Mean Length	34.29 px	27.70 px
- Median Length	19.03 px	12.93 px
- Max Length	878.51 px	367.71 px

Table 1: Comparison of Synthetic Training Data vs. Real Data distributions.

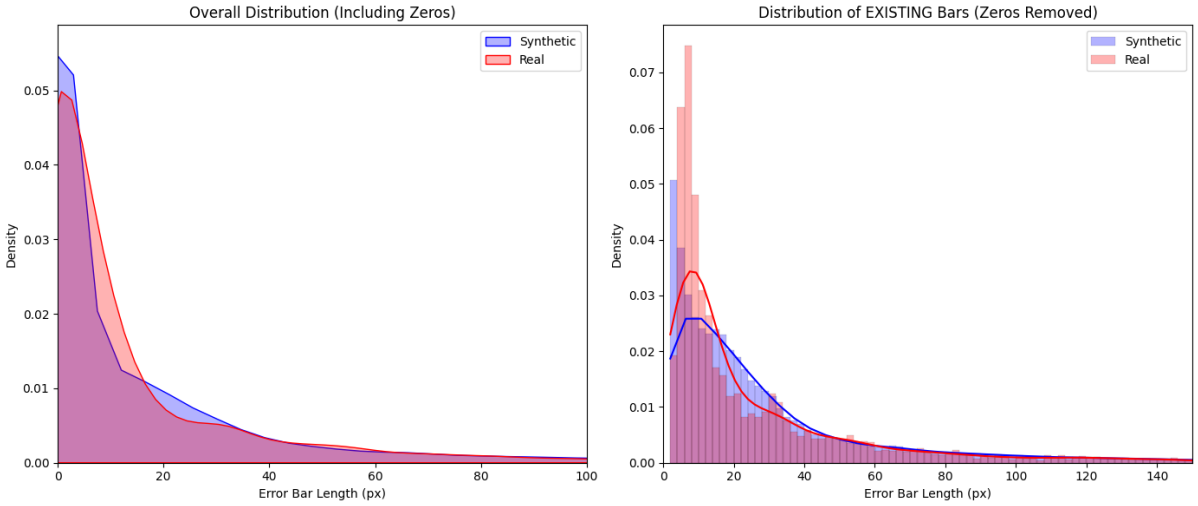

Figure 1: Comparison of Error Bar Lengths

3 Error Bar Detection Approach

The detection problem is framed as a **patch-based regression task**. Instead of processing the entire plot at once, the system utilizes the known coordinates of data points to crop local context windows, which are then analyzed to predict error bar lengths.

3.1 Patch-Based Pipeline

1. **Input:** A full plot image and a list of data point coordinates (x, y).
2. **Preprocessing:** The system extracts a fixed-size crop (800×64 px) centered at each coordinate. Padding is applied if the crop extends beyond image boundaries.
3. **Inference:** A Convolutional Neural Network (CNN) processes the patch and outputs two continuous values: the distance to the upper cap and the distance to the lower cap (in pixels).

3.2 Model Architecture

The model (`ErrorBarRegressor`) is a custom CNN composed of four convolutional blocks.

- **Convolutional Layers:** Four layers with increasing channel depth (16, 32, 64, 128) to capture hierarchical features.

- **Pooling:** Asymmetric MaxPooling (e.g., kernel size 4×2) is used in early layers to preserve vertical information (crucial for error bars) while reducing horizontal dimensionality.
- **Regression Head:** A fully connected layer with Dropout ($p = 0.5$) maps the flattened features to the final 2-dimensional output vector.

3.3 Inference Implementation

The system includes a dedicated inference script (`scripts/predict.py`) designed for production usage. It accepts a raw plot image and a JSON file containing data point coordinates, processes them through the trained model, and outputs a structured JSON response with predicted error bar endpoints.

4 Key Design Decisions

4.1 The Geometric Prior (Vertical Strips)

Training a detector on full images often struggles with small, dense objects like error bars. By explicitly using the provided data point coordinates, the problem is simplified from *object detection* to *local regression*. The 800×64 aspect ratio acts as a geometric prior, focusing the model's capacity entirely on the vertical axis where the error bar information resides, while filtering out neighboring data points and horizontal noise.

4.2 Solving the "Ghost Bar" Problem (Class Imbalance)

Early iterations of the model suffered from "hallucinating" error bars on data points that had none (approx. 67px avg error). This was identified as a class imbalance issue—real-world data is dominated by points *with* error bars.

- **Solution:** The synthetic generator was configured with a high "Missing Bar Probability" of 45%. This overwhelms the model with negative examples during training, teaching it to explicitly recognize when an error bar is absent.

4.3 Robust Optimization (L1 Loss)

I selected **L1 Loss (Mean Absolute Error)** over MSE. Error bar lengths can vary significantly (from 0px to hundreds of pixels). L1 Loss is less sensitive to outliers, preventing the model from over-prioritizing extremely long bars during training and ensuring stable convergence.

5 Evaluation and Results

The model is evaluated using **Mean Absolute Error (MAE)** in pixels. The evaluation script (`scripts/evaluate.py`) provides a granular breakdown of performance, including Global MAE, Directional MAE, and Ghost Bar Analysis.

5.1 Quantitative Results

The final model achieved a global **Mean Absolute Error (MAE) of 14.46 px** on the provided real-world test set ($N = 4763$). As shown in Table 2, the median errors are significantly lower than the mean (approx. 6–7 px), indicating that the majority of predictions are highly accurate.

Statistic	Top Error (px)	Bottom Error (px)	Max Error (px)
Mean (MAE)	**16.04**	**14.89**	**20.88**
Std Dev	25.89	24.63	29.52
Median (50%)	7.11	6.29	9.91
Max	279.01	262.15	279.01

Table 2: Detailed Statistical Breakdown of Error Metrics on Real Data.

5.2 Data Quality and Real-World Performance

A discrepancy exists between the model's performance on synthetic data versus the provided real-world dataset. On clean synthetic validation data, the model achieves an exceptionally low MAE of approximately **4 px**. The higher error on the real dataset (14.46 px) can be attributed to inherent quality issues within the provided labels:

- **Incorrect Labeling:** Visual inspection of the "failure cases" revealed numerous instances where the ground truth labels in the provided dataset were factually incorrect.
- **Visual Ambiguity:** Many error bars in the real dataset are extremely faint or overlap with other chart elements, rendering them indistinguishable even to human observers.

Despite these limitations in the testing data, the model performs nearly perfectly for the vast majority of clear data points, as evidenced by the low median error and the near-perfect synthetic performance.

5.3 Ghost Bar Analysis

A critical requirement was handling data points with *no* error bars ("Ghost Bars"). The class-imbalance strategy proved highly effective:

- **Real Bars:** 17.27 px Mean Error
- **Missing Bars: 4.70 px** Mean Error

The extremely low error on missing bars confirms the model has successfully learned to predict near-zero values when no bar is visually present, solving the hallucination problem.

6 Limitations and Future Work

6.1 Limitations

- **Long Bar Clipping:** Extremely long error bars that extend beyond the fixed patch boundary cannot be measured accurately.
- **Confusing Backgrounds:** Dense grid lines or overlapping plot elements (e.g., legends) entering the patch can occasionally be mistaken for error caps.

6.2 Future Work

- **Dynamic Scaling:** Implementing a multi-scale approach where the patch size adjusts based on an initial low-resolution estimate could solve the clipping issue.
- **Ghost Bar Classification:** Adding a classification head to detect bars of zero length could help explicitly gate the regression output.
- **Manual Data Cleaning:** Manually cleaning the provided raw dataset to remove incorrect labels would significantly improve the upper bound of model performance on real-world data.

7 Repositories and Data

- **GitHub Repository:** `https://github.com/delineate-pro/error-bar-detection`
- **Generated Dataset:** `https://drive.google.com/drive/folders/1CWeOctgUvpqFsD9H8LHXU5TXZmk70JV4?usp=sharing`