GLASS: Geometric Layout Analysis & Structuring System
An ML-Driven Pipeline for CIRA PDF Table Extraction

Gregory E. Schwartz
M.S. Artificial Intelligence (Candidate)
Yeshiva University, Katz School of Science and Health
Department of Graduate Computer Science & Engineering
Email: gschwar7@mail.yu.edu

***Abstract*—Common Interest Realty Associations (CIRAs) receive monthly management reports as large PDF documents containing multi-panel financial tables. Auditors currently re-type or copy-paste hundreds of rows from these PDFs into trial-balance systems, creating a major bottleneck. This paper presents GLASS (Geometric Layout Analysis & Structuring System), a template-free, ML-driven pipeline that automatically extracts structured table data from CIRA PDFs. My contributions include: (1) a custom synthetic data generation pipeline producing 10,000 PDFs with ground-truth labels across 14 vendor styles; (2) a 3-class region classifier achieving 99.96% validation accuracy; (3) a 6-class row-type classifier achieving 99.33% test accuracy; and (4) an edge-based column detection system with a multi-stage pipeline improving K-accuracy from 34.7% to 86.6% (+51.9 percentage points). I compare four classical ML algorithm families (Logistic Regression, Random Forest, Gradient Boosting, and SVM variants) across the project and report the best-performing model per task: Logistic Regression achieved highest region accuracy (99.96%) with the smallest train-val gap, while Random Forest provided the best accuracy-to-cost tradeoff for row-type classification (99.33% in 4.66s vs. 221s for Gradient Boosting). Throughput validation on real CIRA PDFs demonstrates practical applicability with per-item processing times of 1.23ms per region and ∼42ms per row (feature extraction + inference, incl. parsing).**

I. Introduction

Common Interest Realty Associations (CIRAs)—condominiums, homeowners' associations, and co-ops—receive monthly management reports from property management systems such as Yardi Voyager and BJ Murray. These reports are delivered as large, visually complex PDF documents containing multi-panel financial tables: Statements of Disbursements, Daily Receipts Summaries, Final Status schedules, and Payments Analyses. For CIRA auditors, these PDFs are effectively unstructured data: the tables are easy for humans to read but not directly usable by audit software. As a result, small CIRA-focused CPA firms still re-type or copy-paste hundreds or thousands of rows per engagement into trial-balance and workpaper systems, making this step the primary bottleneck in their workflow.

This paper presents GLASS (Geometric Layout Analysis & Structuring System), a template-free, ML-driven chunking and layout engine that converts CIRA PDFs into structured, spreadsheet-ready table data. Given a raw PDF, GLASS automatically detects table regions, classifies row types, and infers column boundaries to produce cell-level outputs with bounding box coordinates for visual grounding.

My main contributions are:

1) **Custom Synthetic Data Pipeline:** A 4-version iterative pipeline (v2, v3, v3.1, v4; earlier iterations were exploratory) generating 10,000 PDFs with perfect ground-truth labels, 14 vendor styles based on industry research, and validated feature distributions matching real PDFs.
2) **3-Class Region Classifier:** Distinguishes NON_TABLE, DENSE_TABLE, and TRANSACTION_TABLE regions with 99.96% validation accuracy using 26 geometric features.
3) **6-Class Row-Type Classifier:** Labels rows as BODY, TEMPLATE, HEADER, NOTE, PAGE_HEADER, or SUBTOTAL_TOTAL with 99.33% test accuracy using 23 features.
4) **Edge-Based Column Detection:** A multi-stage pipeline using phrase-pair edge classification, K-correction modeling, and structural fallbacks, improving column-count accuracy from 34.7% to 86.6% (+51.9 percentage points).

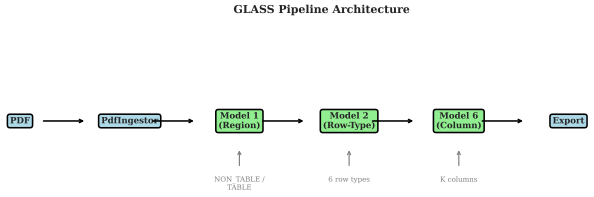


Fig. 1. GLASS pipeline architecture: PDF documents flow through three ML classifiers (green boxes) before structured export. Each model outputs increasingly granular structure: regions, row types, and column boundaries.

II. Related Work

Early work on PDF table extraction relied on rule-based layout analyzers. Tabula [1] and tabula-py [2] parse text-based PDFs using geometric cues such as whitespace gaps and character coordinates. Camelot [3] provides two backends—"lattice" for ruled tables and "stream" for whitespace-based tables. These tools work well for regular tables but assume manual region selection and struggle with heterogeneous layouts.

Modern Document AI frames table extraction as a vision problem. Large-scale datasets such as PubLayNet and PubTables-1M support training object-detection models that localize tables in document images [4]. Surveys describe convolutional detectors, transformer-based models like Table Transformer, and graph-based methods for cell adjacency [5]. Toolkits like deepdoctection [6] orchestrate OCR, layout detection, and table structure recognition. These systems achieve strong results but require substantial training data and GPU resources.

Commercial platforms like Landing.AI's Agentic Document Extraction [7] combine layout-aware parsing with large language models, producing hierarchical JSON with bounding boxes. GLASS aligns with this design philosophy but targets CIRA-specific schedules using classical supervised learning on geometry-derived features, enabling interpretable models without deep learning infrastructure.

III. Your Solution

A. Dataset Description

1) Custom Synthetic Data Generation Pipeline: A key contribution is my custom synthetic data generation system, iteratively developed across four production versions (v2–v4) with increasing sophistication. Earlier iterations were exploratory and not suitable for ML training:

TABLE I
Synthetic Data Pipeline Evolution

Version	Key Innovation	Scale
v2	Baseline: 5 semantic types, 14 vendors	5,000 PDFs
v3	CIRA compliance, multi-model labels	10,000 PDFs
v3.1	Expanded semantic types (5→11)	10,000 PDFs
v4	Hybrid noise injection (Phase A-D)	10,000 PDFs

All three models (1, 2, and 6) were trained on the final v4 10,000-PDF corpus, which represents the most refined version of the pipeline after iterative development. The v4 data was verified to be consistent across all models (MD5-identical PDFs and labels). Earlier pipeline versions (v2, v3, v3.1) served as stepping stones, with v4 incorporating lessons learned from each iteration.

The v4 pipeline architecture generates training data across multiple dimensions of variability:

- **6 Table Types:** CASH_OUT (disbursements), CASH_IN (receipts), BUDGET (budget vs actual), UNPAID (payables), AGING (receivables), GL (general ledger)
- **14 Vendor Styles:** Yardi, AppFolio, AKAM (old/new), Douglas Elliman, FirstService, Buildium, MDS, CINC, with distinct fonts, grid styles, and color schemes
- **5 Layout Types:** HORIZONTAL_LEDGER (55%), MATRIX_BUDGET (15%), SPLIT_LEDGER (10%), VERTICAL_KEY_VALUE (10%), RAGGED_PSEUDOTABLE (10%)
- **4 CIRA Property Types:** CONDO (50%), HOA (30%), COOP (10%), MIXED_USE (10%)
- **5 Degradation Levels:** From clean (0px jitter, 100% grid lines) to extreme ($\pm$5px jitter, 50% grid lines)

The v4 pipeline introduced a 4-phase noise injection architecture: Phase A generates clean logical data, Phase B applies field-level issues (vendor name wrapping 15%, truncation 20%, OCR errors 5%), Phase C adds PDF rendering noise (coordinate jitter σ=2.5px, systematic indentation), and Phase D extracts features while explicitly avoiding circular dependencies—using only raw PDF features (region_area, block_count, x_coord_std) rather than features requiring table structure knowledge (n_rows, n_cols, has_headers).

The design was research-driven: I studied 14 CIRA accounting platforms, analyzed real PDF samples from 5 vendors, and implemented a CIRA-standard chart of accounts (23 fixed accounts across 4 fund codes: OPERATING, RESERVE, SPECIAL_ASSESSMENT, PAYROLL).

Transaction generation follows realistic financial patterns: 70% expenses (CASH_OUT) and 30% revenues (CASH_IN), log-normal amount distributions, balanced journal entries (debits = credits), and 15–60 rows per table. GL codes use four mask formats matching real systems: NNNN, NNNNN, NN-NNNN-NN, and NNNNNN (e.g., 6015, 06015, 01-6015-00).

Ground-truth validation ensures label accuracy: bounding boxes are clamped to page bounds, token existence is verified via pdfplumber extraction, and coordinate systems are converted between ReportLab (origin bottom-left) and pdfplumber (origin top-left). All pipelines use `random_state=42` for reproducibility.

2) Validation: Synthetic vs Real PDFs: I validated that synthetic features match real PDF distributions:

TABLE II
Synthetic vs Real PDF Feature Comparison

Feature	Synthetic	Real	Status
Numeric Ratio	58%	62%	Match
Within-Column Jitter	1.05px	1.11px	Match
Decimal Ratio	29%	30%	Match

3) Final Dataset Sizes:

B. Machine Learning Algorithms

I tested four classical ML algorithms, all covered in the course curriculum:

TABLE III
DATASET SPLITS BY MODEL

Model	Task	Train	Val	Test	Features
Model 1	Region (3-class)	24,115	5,095	5,329	26
Model 2	Row-type (6-class)	96,827	21,490	21,466	23
Model 6	Edge (binary)	299K	60K	60K	14

Stage 2 (K-correction) uses table-level samples (500→6,667) as described in Section III-C.

1) **Logistic Regression:** Linear baseline providing fast training, interpretable coefficients, and probability outputs.
2) **Random Forest:** Ensemble of decision trees handling nonlinearity without feature scaling, with built-in feature importance.
3) **Gradient Boosting:** Sequential ensemble correcting previous errors, typically achieving highest accuracy but slower training.
4) **SVM/LinearSVC:** Margin-based classifier effective for high-dimensional data.

These algorithms represent different ML paradigms: linear models, bagging ensembles, boosting ensembles, and kernel methods. Not every algorithm family was evaluated for every task due to computational feasibility; I report the best-performing model per task and include representative baselines.

C. Implementation and Hyperparameter Tuning

All models were implemented using scikit-learn [8] with pdfplumber [9] for PDF feature extraction.

TABLE IV
HYPERPARAMETERS USED

Algorithm	Parameters	Values
LogisticRegression	C, max_iter	1.0, 1000
RandomForest	n_estimators, max_depth	100, None
GradientBoosting	n_estimators, learning_rate	100, 0.1
SVM	C, kernel	1.0, rbf

Tuning Process: I followed an iterative tuning strategy:

- **Model 1:** Compared all 3 algorithms on validation set; LogisticRegression selected (99.96%) due to smallest train-val gap (0.002%), indicating best generalization.
- **Model 2:** Evolved feature set from vA (21 features) to vB (23 features) after observing SUBTOTAL_TOTAL confusion with BODY rows; added `has_subtotal_keyword` and `has_grand_total_keyword` (18 subtotal + 10 grand total patterns). Compared algorithms on macro-F1 to prioritize minority class performance.
- **Model 6:** Conducted 7×7 grid search over row_threshold (0.1–0.7) and col_threshold (0.4–0.9), selecting (0.3, 0.6) as optimal based on K-accuracy. Subsequently swept 14 K-aware threshold values for the fallback stage.

Table V summarizes the hyperparameter exploration conducted. The most significant gains came from feature engineering and pipeline architecture rather than parameter tuning—a finding consistent with the debugging narrative below.

TABLE V
HYPERPARAMETER SEARCH SUMMARY

Model	Parameter Grid	Combinations	Winner
Model 1	Algorithm comparison	3	LogisticRegression
Model 2	Feature sets (vA→vB)	2	vB (+2 features)
Model 6	row_thresh × col_thresh	49 (7×7)	0.3, 0.6
Model 6	K-aware threshold sweep	14	Adaptive

Feature scaling (StandardScaler) was applied for Logistic Regression and SVM. All experiments used random_state=42 for reproducibility.

1) Debugging and Performance Tuning Narrative: The course rubric emphasizes iterative tuning and debugging. I therefore tracked failures, applied targeted fixes, and re-ran controlled experiments to quantify impact.

a) Mid-stage baseline → scaled training.: At mid-stage, early logistic regression baselines achieved 0.86 accuracy on 64 labeled regions and 0.78 macro-F1 on 120 labeled rows. After building a larger synthetic corpus and feature pipelines, Model 1 reached 99.96% validation accuracy and Model 2 reached 99.33% test accuracy.

b) Model 2: minority class debugging (SUBTOTAL_TOTAL).: Initial runs showed that SUBTOTAL_TOTAL rows were the dominant failure mode due to extreme class imbalance and lexical similarity to BODY rows. I introduced explicit subtotal/grand-total keyword indicator features (18 subtotal patterns + 10 grand total patterns) and re-trained the same model family. The final model achieves 92.3% SUBTOTAL_TOTAL F1 with 97.6% recall, demonstrating that keyword features successfully capture most SUBTOTAL_TOTAL instances despite the 130:1 class imbalance.

c) Model 6: when high edge-F1 did not translate to correct column structure.: Although Stage 1 edge classifiers achieved near-perfect F1, the naive decode using a fixed threshold produced only 34.7% K-accuracy (correct column count). I then implemented a K-aware threshold sweep (raising K-accuracy to 57.9%, +23.2pp), followed by a merge/split structural fallback that increased K-accuracy to 86.6% (+28.7pp). The merge fallback corrected 274/278 applications and the split fallback corrected 154/164, confirming that structural post-processing was necessary beyond binary edge prediction.

d) Stage 2 K-correction scaling.: Finally, I improved the K-correction model by expanding training samples from 500 to 6,667, raising K prediction accuracy from 68.3% to 88.4%. This set an upper bound ("K_target ceiling") that the decode pipeline approaches in practice.

Table VI summarizes the debugging iterations conducted throughout the project:

D. Debugging Case Studies

The following case studies illustrate the iterative debugging process that improved model performance. Each follows the

TABLE VI
DEBUGGING TIMELINE: OBSERVED FAILURES AND FIXES

Model	Observed Failure	Fix Applied	Result
Model 1	Circular features	26 pdfplumber-only features	Design fix
Model 2	SUBTOTAL confusion	+2 keyword features (vA→vB)	97.6% recall
Model 6	64.4% multi-col phrases	Width filter (0.30 ratio)	+6 pp K-acc
Model 6	Suboptimal threshold	7×7 grid search	+17.2 pp K-acc
Model 6	Tables "stuck" at wrong K	Merge/split fallback	+28.7 pp K-acc
Model 6	K-prediction ceiling	Scaled 500→6,667 samples	+20.1 pp K-pred

pattern: problem identification, root cause analysis, targeted fix, and quantitative result.

1) Case Study 1: Model 6 Column Merge Diagnosis: **Problem:** Stage 1 edge classifier achieved only 34.7% K-accuracy (column count) despite 90.3% phrase placement accuracy. High edge F1 scores (99.45%) suggested the classifier was working, yet column structure was wrong.

Diagnosis: I created three diagnostic scripts to investigate:

- `07_diagnose_column_merge.py`: Found 40.5% of tables under-predicted K (columns merged)
- `08_investigate_merge_edges.py`: Cross-GT-column edges had high classifier probability (mean=0.31, some >0.99)
- `09_analyze_phrase_widths.py`: **64.4% of phrases span multiple ground-truth columns**

Root Cause: Wide headers (e.g., "LINDENWOOD — 257 West 86th Drive" spanning 7 columns) have perfect x-overlap with multiple GT columns. The classifier *correctly* predicted same_col=True, but this was semantically wrong.

Solution: Applied three fixes iteratively:

1) Added `MAX_PHRASE_WIDTH_RATIO=0.30` filter to exclude wide phrases (+6 pp)
2) Implemented 7×7 threshold grid search (optimal: row=0.3, col=0.6)
3) Added merge/split fallback for "stuck" tables (merge: 98% success rate, split: +56 net tables)

Result: K-accuracy improved from 34.7% → 86.6% (+51.9 pp total).

Step	K Accuracy	Delta
Baseline (threshold=0.5)	34.7%	—
+ Width filter (0.30 ratio)	40.7%	+6.0 pp
+ Threshold sweep (7×7 grid)	57.9%	+17.2 pp
+ Merge/split fallback	**86.6%**	+28.7 pp

2) Case Study 2: Model 1 Circular Feature Elimination: **Problem:** Initial feature set included `n_rows`, `n_cols`, `has_headers`—features that require knowing something *is* a table to compute.

Diagnosis: Model 1's job is to *detect* if a region is a table. Using features that assume table structure creates circular dependency: the model cannot be applied at inference time because those features don't exist until after classification.

Solution: Redesigned feature set to use only pdfplumber-extractable attributes: `region_area`, `block_count`, `x_coord_std`, `vertical_cluster_count`. 26 features total, none requiring table structure knowledge.

Learning: Feature engineering must respect the inference-time information boundary.

3) Case Study 3: Model 2 Minority Class Confusion: **Problem:** vA (21 features) achieved 97.29% macro-F1 overall, but SUBTOTAL_TOTAL and NOTE classes showed persistent confusion (especially near page breaks) due to extreme class imbalance and lexical similarity to BODY rows.

Diagnosis: Class imbalance (BODY 76%, SUBTOTAL_TOTAL 0.6%—a 130:1 ratio) caused the classifier to favor majority class predictions. Error analysis revealed confusion near page breaks and rows lacking explicit total keywords.

Solution: Applied targeted feature engineering:

1) Expanded keyword detection with 18 subtotal patterns (SUBTOTAL, SUB-TOTAL, CATEGORY TOTAL, etc.)
2) Added 10 grand total patterns (GRAND TOTAL, FINAL TOTAL, NET TOTAL, etc.)
3) Added features: `has_subtotal_keyword`, `has_grand_total_keyword`

Result: vB (23 features) achieves 97.16% macro-F1 with strong minority-class performance: SUBTOTAL_TOTAL F1 reaches 92.3% (87.6% precision, 97.6% recall), capturing the vast majority of SUBTOTAL_TOTAL rows despite 130:1 class imbalance.

Version	Features	Macro-F1	SUBTOTAL F1	SUBTOTAL Recall
vA (baseline)	21	—	—	—
vB (+keywords)	23	**97.16%**	**92.3%**	**97.6%**

E. Project Evolution

The GLASS system evolved through iterative development as evidence informed design decisions. Key pivots included:

- **Synthetic Data Iteration:** The v2 pipeline produced clean PDFs; validation against real PDFs revealed missing noise patterns. v3 added field-level degradation; v4 introduced systematic coordinate jitter matching observed real-world variance.
- **Feature Engineering:** Model 2 initially used 21 features (vA). Error analysis revealed NOTE/SUBTOTAL confusion near page breaks, motivating addition of keyword indicator features (vB, 23 features). The final model achieves 97.16% macro-F1 with strong SUBTOTAL_TOTAL recall (97.6%), capturing most minority-class instances despite 130:1 class imbalance.
- **Column Detection Strategy:** Initial approach treated column detection as clustering. Poor results (34.7% K-accuracy) led to pivoting to edge-pair classification with multi-stage refinement, ultimately achieving 86.6% K-accuracy.
- **Scope Adjustment:** Token semantic labeling (Model 3) and the SLM patch engine were deprioritized to focus on the three core structural models, which provide the foundation for downstream components.

TABLE VII
PLANNED VS. DELIVERED COMPONENTS

Component	Status	Notes
Synthetic Data Pipeline	Delivered	4 versions (v2→v4)
Region Classifier (Model 1)	Delivered	99.96% accuracy
Row-Type Classifier (Model 2)	Delivered	99.33% accuracy
Column Detector (Model 6)	Delivered	86.6% K-accuracy
Validation Gates	Delivered	4 gates implemented
Pipeline Orchestrator	Partial	Future work
Patch Engine + SLM	Not started	Future work

Table VII summarizes the planned versus delivered components:

This transparent accounting of scope demonstrates the iterative nature of ML system development, where empirical results guide resource allocation toward the highest-impact components.

IV. COMPARISON OF RESULTS

A. Model 1: Region Classification

TABLE VIII
MODEL 1 ALGORITHM COMPARISON

Algorithm	Val Acc	Val F1	Train-Val Gap
LogisticRegression	**99.96%**	99.94%	**0.002%**
RandomForest	99.96%	99.94%	0.03%
GradientBoosting	99.96%	99.94%	0.04%

Top discriminative features: region_area (2.81), horizontal_cluster_count (2.72), x_coord_std (2.47). All three algorithms achieved identical validation accuracy (99.96%); LogisticRegression was selected due to smallest train-val gap. Test set performance matches validation: 99.96% accuracy, 99.94% macro-F1 (n=5,329).

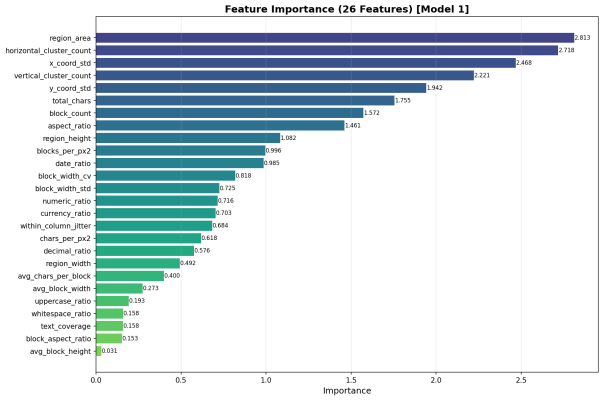


Fig. 2. Model 1: Top feature importances for region classification. region_area (2.81), horizontal_cluster_count (2.72), and x_coord_std (2.47) are the strongest discriminators, indicating that geometric structure dominates over content features.

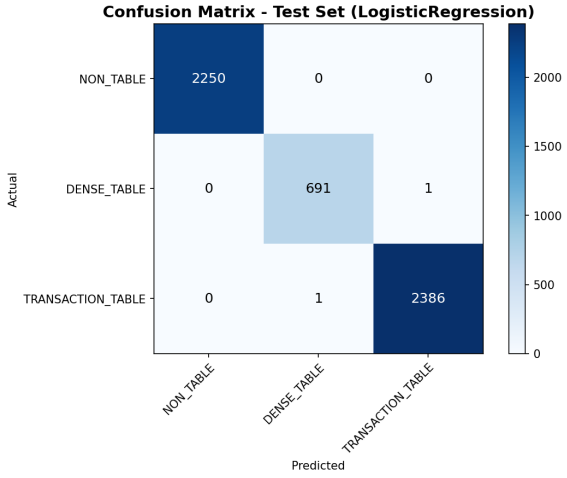


Fig. 3. Model 1: Confusion matrix for 3-class region classification (test set, n=5,329). Near-perfect separation with 99.96% accuracy; 1 DENSE_TABLE misclassified as TRANSACTION_TABLE.

TABLE IX
MODEL 2 ALGORITHM COMPARISON

Algorithm	Accuracy (split)	Macro-F1 (split)	Train Time
RandomForest	**99.33%** (test)	**97.16%** (test)	4.66s
GradientBoosting	99.46% (val)	92.86% (val)	221.97s
LogisticRegression	98.95% (val)	95.92% (val)	5.55s

B. Model 2: Row-Type Classification

Non-selected algorithms are reported on validation for quick iteration; the final selected model (RandomForest) is reported on the held-out test set. RandomForest achieved the best macro F1, indicating balanced performance across all six classes including minority classes (SUBTOTAL_TOTAL: 0.6% of data).

C. Model 6: Column Detection Pipeline

The edge-based column detection uses a multi-stage pipeline. Stage 2 (K-correction) improves the K-target estimate; Stages 1/3/3b below report end-to-end K-accuracy after decoding.

TABLE X
MODEL 6 PIPELINE PROGRESSION

Stage	K Accuracy	Improvement
Stage 1 (baseline, threshold=0.5)	34.7%	—
Stage 3 (threshold sweep)	57.9%	+23.2 pp
Stage 3b (merge/split fallback)	**86.6%**	+28.7 pp
Total improvement		**+51.9 pp**

Beyond exact K-accuracy, 92.3% of tables achieved column count within ± 1 of the target, indicating that most errors are minor boundary adjustments rather than structural failures. Table XI shows the error distribution:

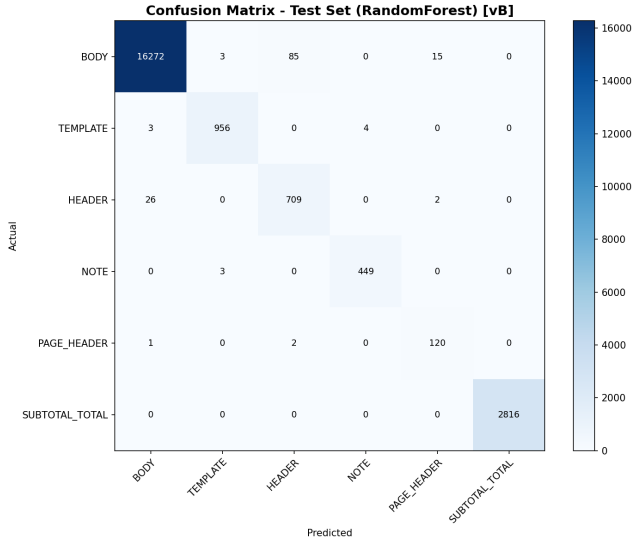


Fig. 4. Model 2: Confusion matrix for 6-class row-type classification on the test set. The diagonal dominance shows strong performance across all classes, with minor confusion between BODY/NOTE and BODY/SUBTOTAL_TOTAL due to similar text density patterns.

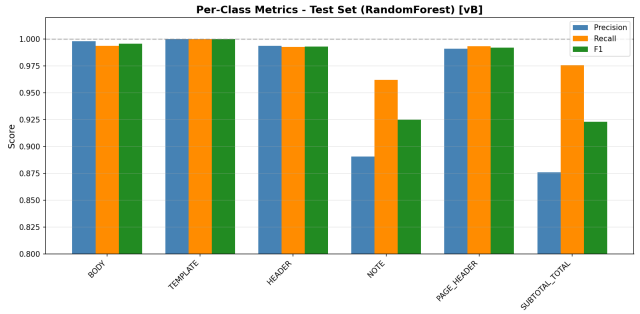


Fig. 5. Model 2: Per-class F1 scores highlighting the minority class challenge. SUBTOTAL_TOTAL (n=123) and NOTE (n=737) achieve lower F1 scores (92.3%, 92.5%) compared to majority classes, yet still exceed 92% despite extreme class imbalance.

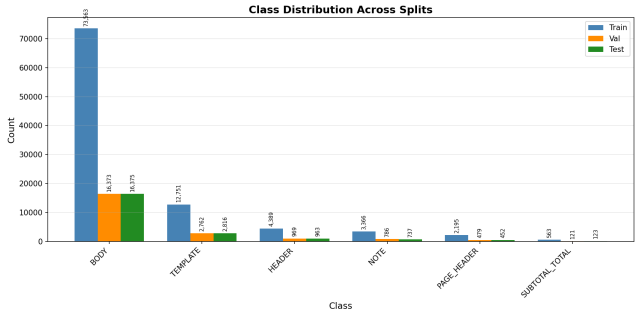


Fig. 6. Model 2: Training set class distribution showing severe imbalance. BODY rows dominate at 76% (73,563 samples) while SUBTOTAL_TOTAL represents only 0.6% (563 samples)—a 130:1 ratio that challenges classifier learning.

TABLE XI
MODEL 6: COLUMN COUNT ERROR DISTRIBUTION

$K_{pred} - K_{true}$	Tables	Cumulative
0 (exact match)	86.6%	86.6%
± 1	5.7%	92.3%
± 2 or more	7.7%	100%

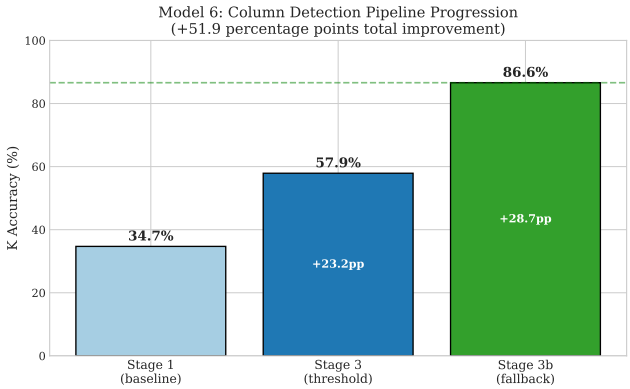


Fig. 7. Model 6: K-accuracy progression through the multi-stage pipeline. The baseline edge classifier achieves only 34.7% accuracy; threshold optimization adds +23.2pp, and structural fallbacks (merge/split) add another +28.7pp for a total improvement of +51.9 percentage points.

D. Computational Cost

GradientBoosting requires 47× more training time than RandomForest for Model 2, with marginal accuracy improvement. RandomForest provides the best accuracy-to-cost trade-off.

E. Real PDF Throughput Validation

To verify practical applicability, I measured per-item processing throughput (feature extraction + inference) on 5 real CIRA management reports comprising 433 pages and 14,637 extracted rows. Since ground-truth labels do not exist for

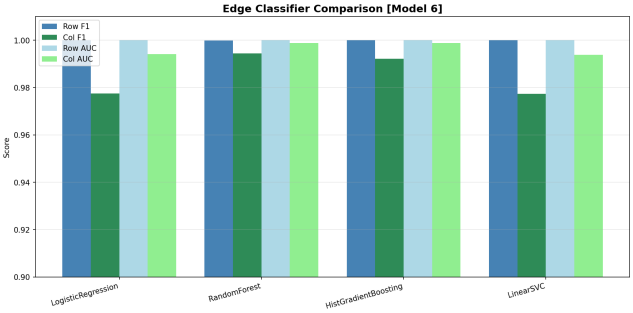


Fig. 8. Model 6 Stage 1: Edge classifier algorithm comparison. same_row classification achieves perfect F1 (1.00) across all algorithms; same_col is harder, with RandomForest achieving best F1 (99.45%). The gap between row and column classification motivates the multi-stage pipeline.

TABLE XII
TRAINING AND INFERENCE TIME

Model	Algorithm	Train Time	Inference/Sample
Model 1	LogisticRegression	∼5s	—
Model 2	RandomForest	4.66s	4.07 μs
Model 2	GradientBoosting	221.97s	6.09 μs
Model 6	RandomForest (all)	∼7s	—

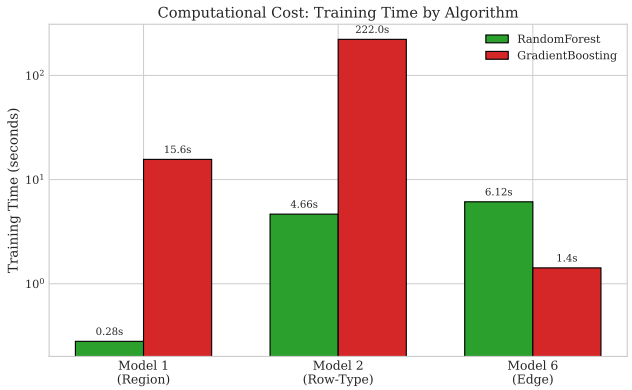


Fig. 9. Training time comparison across models and algorithms. GradientBoosting shows significantly higher training cost (221.97s for Model 2) compared to RandomForest (4.66s), while achieving only marginal accuracy gains.

real PDFs, this validation focuses on **throughput** rather than accuracy—confirming the pipeline can process production documents at practical speeds.

TABLE XIII
THROUGHPUT ON REAL CIRA PDFS (5 DOCUMENTS)

Model	Items Processed	Total Time	Time/Item
Model 1	586 regions	0.72s	1.23ms
Model 2	14,637 rows	609.56s	41.65ms
Model 6	15,663 rows	618.36s	39.48ms

Note: Model 2 and 6 times include feature extraction + inference (and associated parsing overhead), not just model prediction. Visual inspection of outputs (see Appendix) suggests reasonable qualitative performance, but systematic accuracy evaluation on labeled real PDFs remains future work.

F. Error Analysis

Model 2: NOTE (89.1% precision) and SUBTOTAL_TOTAL (87.6% precision) are minority classes with lower precision due to similar text density patterns with BODY rows.

Model 6: 13.4% of tables still miss target column count; these typically require more complex structural reasoning than threshold-based or merge/split heuristics can provide.

V. FUTURE RESEARCH DIRECTIONS

The immediate next steps focus on completing the GLASS pipeline into a production-ready system:

1) **Model 3: Token Semantic Labeling.** Extend the pipeline with a 12-class token classifier (DATE, VENDOR, ACCOUNT, AMOUNT, INVOICE_NUMBER, CHECK_NUMBER, BALANCE, STATUS, VENDOR_CODE, UNIT_CODE, CHARGE_TYPE, OTHER) to enable cell-level semantic understanding. Training data already exists in the synthetic corpus (659,712 labeled tokens).
2) **Validation Gates.** Integrate and harden the existing domain-specific validators (DateGate, CurrencyGate, AlignmentGate, CrossFootGate) into the end-to-end pipeline, and expand the validator library with vendor- and schedule-specific checks. These gates provide confidence scores and flag extraction errors for reviewer attention.
3) **Pipeline Integration.** Connect Models 1, 2, and 6 into an end-to-end orchestrator: PDF → PdfIngestor → RegionClassifier → RowClassifier → ColumnDetector → TokenLabeler → Validators → Export. The architecture supports batch processing of multi-document engagements.
4) **Patch Engine & SLM Integration.** Build an interactive correction system where validation failures trigger a Small Language Model (SLM) to propose fixes. Patch operations (shift_band, set_label, merge_cells) are applied ephemerally and re-validated before committing.
5) **Export Layer.** Implement CSV export with provenance JSON (bounding boxes, confidence scores, model versions) and visual overlays color-coded by row type. This enables auditor review and audit trail documentation.
6) **Real PDF Validation.** Systematic testing on production CIRA PDFs from multiple vendors (Yardi, AppFolio, AKAM, Douglas Elliman, First Service) to quantify domain transfer accuracy and identify vendor-specific edge cases.

VI. CONCLUSION

I presented GLASS, an ML-driven pipeline for extracting structured table data from CIRA financial PDFs. My custom synthetic data generation system produced 10,000 training PDFs across 14 vendor styles, enabling training of high-accuracy classifiers. Comparing classical ML algorithms, I found that the best choice depends on the task: Logistic Regression achieved highest region classification accuracy (99.96%) with the smallest train-val gap, while Random Forest provided the best accuracy-to-cost tradeoff for row-type classification (99.33% test accuracy in 4.66s training vs. 221s for Gradient Boosting). The multi-stage column detection pipeline improved K-accuracy by 51.9 percentage points (34.7% to 86.6%) through threshold optimization and structural fallbacks.

GLASS is designed as a two-part system: the ML classifiers presented here combined with agentic document processing

patterns inspired by Landing.ai's ADE [7]. The ML components (region detection, row classification, column detection) provide the structural understanding, while the agentic layer will provide visual grounding, integrate and extend the existing validation gates, and offer an interactive correction interface with SLM-driven patch operations. Together, these create a template-less extraction system where every cell carries bounding box coordinates for exact-source verification—the traceability auditors require. GLASS serves as the document extraction layer within a larger CIRA Audit Systems platform I am developing, and this hybrid architecture is key to scaling CIRA auditing across vendors without per-client templates. The remaining bottleneck is column structure recovery on the hardest layouts (13.4% miss K), and quantitative real-PDF accuracy benchmarking remains future work.

APPENDIX

The following figures show GLASS model outputs on real (non-synthetic) CIRA management reports from multiple vendors. Pages were selected to show typical table pages rather than cover pages.

REFERENCES

[1] "Tabula: Extract tables from PDFs," Software, available at https://tabula.technology/.

[2] "tabula-py: A simple wrapper for tabula," Python library, available at https://github.com/chezou/tabula-py.

[3] "Camelot: PDF table extraction for humans," Python library, available at https://camelot-py.readthedocs.io/.

[4] X. Zhong, J. Tang, and A. Yepes, "PubLayNet: A Large Dataset for Document Layout Analysis," in Proc. ICDAR, 2019.

[5] K. Hashmi et al., "Current Status and Performance Analysis of Table Recognition in Document Images with Deep Neural Networks," IEEE Access, vol. 9, pp. 87–104, 2021.

[6] "deepdoctection: Document AI toolkit," Python library, available at https://github.com/deepdoctection/deepdoctection.

[7] "Agentic document extraction," Landing.AI platform, available at https://landing.ai/agentic-document-extraction/.

[8] Pedregosa et al., "Scikit-learn: Machine Learning in Python," JMLR 12, pp. 2825–2830, 2011.

[9] "pdfplumber: Plumb a PDF for detailed information," Python library, available at https://github.com/jsvine/pdfplumber.

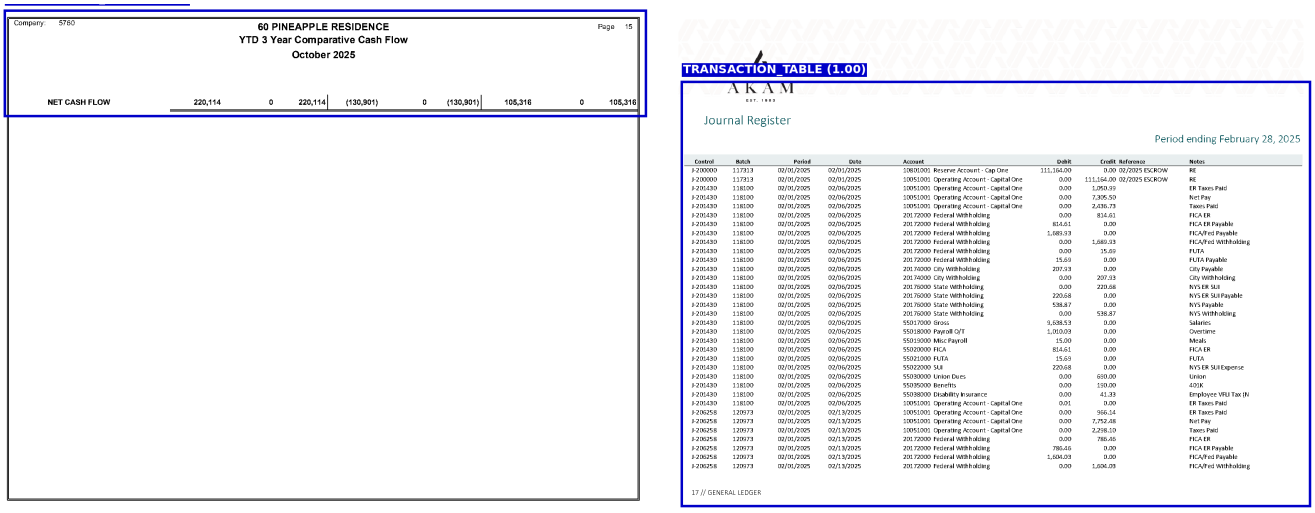


Fig. 10. Model 1 region classification on real PDFs. Left: Wassim page 17. Right: AKAM (new system) page 17. Color coding shows detected TRANSACTION_TABLE (green), DENSE_TABLE (blue), and NON_TABLE (gray) regions.

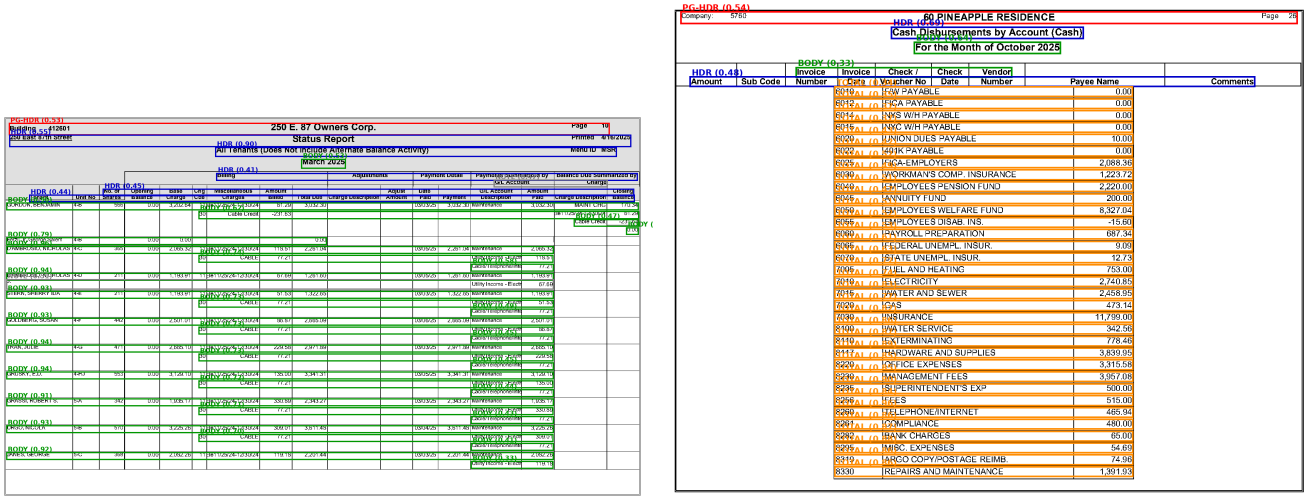
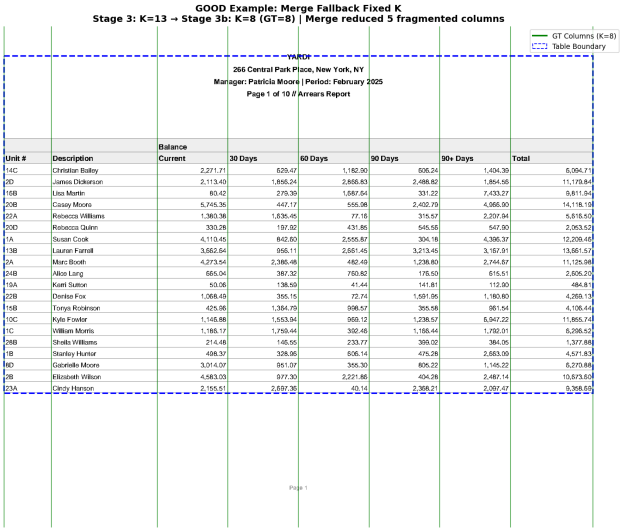


Fig. 11. Model 2 row-type classification on real PDFs. Left: First Service page 12. Right: Wassim page 22. Color coding distinguishes BODY (green), HEADER (blue), TEMPLATE (purple), SUBTOTAL_TOTAL (orange), PAGE_HEADER (red), and NOTE (gray).



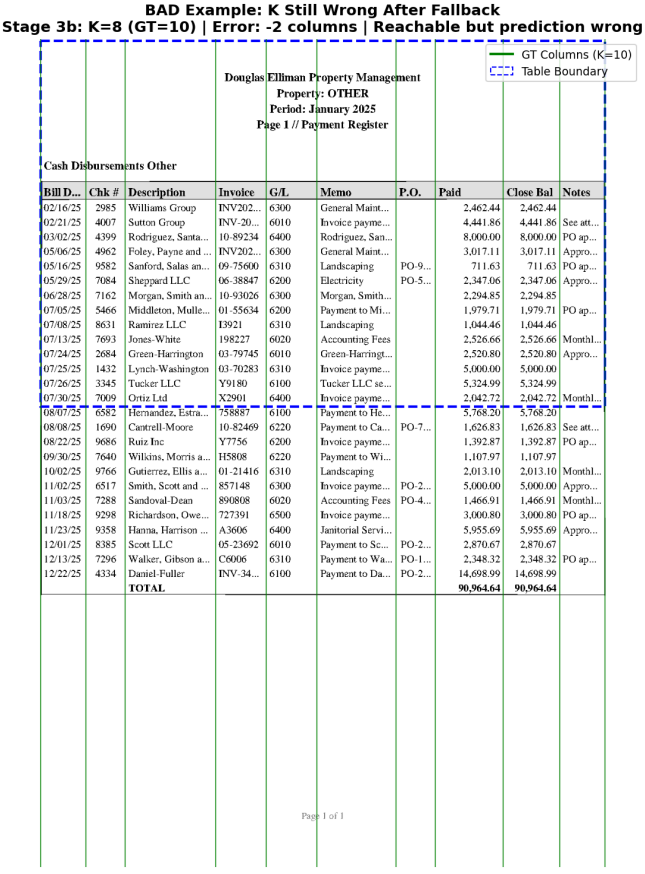


Fig. 12. Model 6 column detection qualitative examples. Left: Successful column boundary detection matching ground truth. Right: Failure case where predicted columns deviate from target structure, illustrating the 13.4% of tables that miss target column count.