International Journal of Technology and Applied Science (IJTAS)
E-ISSN: 2230-9004 ● Website: www.ijtas.com ● Email: editor@ijtas.com

AutoSynth AI: Automated Data Intelligence and Model Generation System

Akshatha R[1], Ms. Vedashree M R[2], Chitra M[3], Neha R[4], Anup Santosh Patil[5]

Department of Artificial Intelligence and Machine Learning
SJB Institute of Technology, Bengaluru, India

Abstract:
Most automated machine learning (AutoML) tools expect a clean table and a user who already knows whether the problem is classification or regression. Real files rarely arrive that way. We built AutoSynth AI to accept whatever the user has: a CSV, JSON, TXT, PDF, or DOCX file. The system recovers a table from the file (including tables inside PDF and Word documents and data pasted as delimited text), decides whether the task is classification or regression and whether the signal lives in a free-text column or in structured features, and sends the data down one of two pipelines. Each pipeline tries a few fixed hyperparameter settings per algorithm, picks the winner by 5-fold cross-validation on the training split, and reports held-out scores. Shapley-value (SHAP) and local surrogate (LIME) explanations accompany the winner, and the trained model can be downloaded as a self-contained package. Behind the interface, the system runs as separate local services (a REST API, a Model Context Protocol (MCP) server, and a Streamlit client) over one core library and an SQLite registry, so it needs no cloud account or API key. On four public structured benchmarks with 6% of the feature values removed, held-out weighted F1 ranges from 0.933 to 0.982 for classification, and R^2 is 0.420 on the Diabetes regression task. On a 1,083-message SMS spam subset, the text pipeline reaches a weighted F1 of 0.976.

Index Terms: AutoML, explainable AI, LIME, Model Context Protocol, multi-format data ingestion, SHAP, text classification.

I. INTRODUCTION

Training a useful model involves a string of small decisions. What should happen to the missing values? How should a column of city names be encoded? Which algorithm fits the problem, and how would anyone know the result is any good? A practitioner answers these almost without thinking. Everyone else gets stuck. Automated machine learning (AutoML) tools such as Auto-sklearn [1], TPOT [2], and PyCaret [3] take over much of this work, yet they all start from a clean table. A support-ticket export, a product-review dump, or a report saved as PDF seldom looks like that, and a non-specialist has no easy way to reshape it.

We built AutoSynth AI around a simpler promise: give it the file you have, and it works out the rest. The main contributions are as follows.

1) Table recovery from five formats. CSV, JSON, TXT, PDF, and DOCX files are accepted. For PDF and DOCX we look for a real table first, and if there is none, we test whether the text itself is a delimited table that someone pasted in as prose.
2) Automatic routing with a quality gate. The system infers the problem type and any free-text column from simple statistics, shows both decisions to the user, and drops feature columns that are too empty to be worth imputing.
3) Preprocessing that never sees the test split. Data is split before imputation and encoding, so every fitted statistic comes from training rows.
4) Cross-validated light tuning. Two or three fixed settings are tried per algorithm, and the winner is chosen by 5-fold cross-validation (CV), not by the test score.
5) A service-oriented design. A shared core library sits behind a REST API, an MCP server, and a Streamlit client, each on its own local port, with an SQLite registry for datasets and models. Nothing needs a cloud account or an API key.
The rest of the paper covers related work (Section II), the system (Section III), experiments (Section IV), a feature comparison (Section V), and a discussion with conclusions (Section VI).

II. RELATED WORK

A. AutoML Systems

In Auto-WEKA [4] and Auto-sklearn [1], picking an algorithm and tuning its hyperparameters are folded into one search problem, which Bayesian optimization then explores. TPOT [2] evolves whole pipelines with genetic programming. PyCaret [3] offers a low-code wrapper for comparing and tuning models. Two surveys [5], [6] describe the field and the usual trade-off between how long a search runs and how good a model it finds. All of these tools expect a DataFrame. AutoSynth AI starts one step earlier, with the raw file, and then chooses between a tabular and a text pipeline.

B. Data Preparation

Mumuni and Mumuni [7] review automated cleaning, imputation, encoding, and feature construction, the work that happens before any model search begins. Our contribution here is modest and practical. We recover tables from documents that were never meant to hold one, and we keep every preprocessing statistic inside the training partition.

C. Explainability

Local Interpretable Model-agnostic Explanations (LIME) [8] explain a single prediction by fitting a simple surrogate model around it, and SHapley Additive exPlanations (SHAP) [9] give each feature a Shapley-value share of a prediction. We run both on the winning model, on the same screen that shows its accuracy.

III. METHODOLOGY

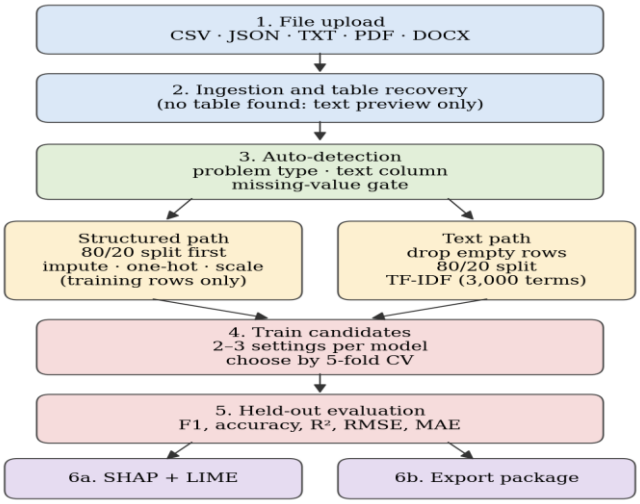


Fig. 1. Overall pipeline, from file upload to model export [10].

A. Service Architecture

The code is split into a core library and three thin front ends (Fig. 2). The library holds ingestion, detection, training, explanation, and prediction, and it is the only place where machine learning happens. A FastAPI service exposes it as a representational state transfer (REST) API on port 8000. A server on port 8001 offers the same operations as tools to any client that speaks the Model Context Protocol (MCP) [11], so an assistant or a code editor can upload a file, train a model, and score records without a custom integration. The Streamlit user interface (UI) on port 8501 is a client of the REST API and contains no modeling code. A small SQLite registry records every dataset and model, and the files themselves live in a local folder. Because the services share one library and one registry, a model trained in the web interface can be scored from an MCP client, and the other way around.

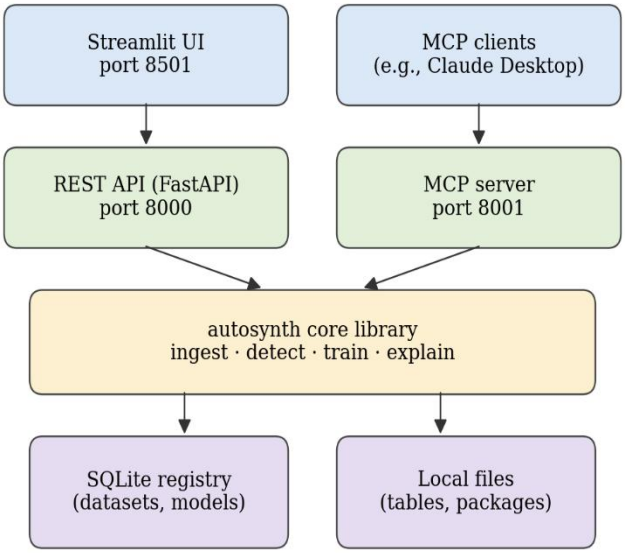


Fig. 2. Service architecture. Every front end calls the same core library.

B. Ingestion and Table Recovery

The loader picks a reader by file extension. CSV and JSON are read directly, with JSON flattened by json_normalize. A TXT file becomes a table when a comma, tab, or semicolon appears a consistent number of times per line, and otherwise each line is one free-text row. For PDF files, pdfplumber pulls tables out page by page. A table that runs onto the next page is joined when the column counts agree, and repeated header rows are removed. If no table is found, the extracted text goes through the delimiter check. DOCX files follow the same order, with native Word tables first and pasted delimited paragraphs second. The delimiter check counts each separator per line and keeps the one that is most consistent. Columns in which at least 80% of the values parse as numbers are converted. When nothing table-like turns up, the user sees a text preview and training stops.

C. Detection and the Missing-Value Gate

A numeric target counts as regression if it has more than 20 distinct values and the distinct-to-row ratio is above 0.05, and as classification otherwise. A column is free text when its average length exceeds 25 characters and more than half of its values are distinct. The interface shows both decisions as badges, so a wrong guess is easy to spot. A user-set missing-value gate (default 40%, range 10 to 90%) then runs. It removes rows with no target, stops training if the target is too sparse or fewer than 20 labeled rows remain, and drops feature columns above the threshold. We drop them because imputing a column that is mostly empty fills it with one repeated value and adds little else.

D. Structured Pipeline

The raw features are split 80/20 first, with stratification when every class has enough rows. Means for numeric columns and modes for categorical columns are then computed on the training rows alone and applied to both partitions. Categorical columns are one-hot encoded with the first level dropped. The test partition is encoded with all of its levels and then aligned to the training columns. This keeps the baseline level from training intact, and a level that appears only at test time becomes an all-zero row. Tree models do not need scaling, so standardization is applied only for Logistic Regression, support vector machines (SVM), support vector regression (SVR), and Linear Regression, and it is fitted on the training partition. For classification the candidates are Logistic Regression, Decision Tree, Random Forest [12], and SVM [13]; for regression they are Linear Regression, Decision Tree, Random Forest, and SVR.

E. Text Pipeline

When a dominant text column is found and the task is classification, empty-text rows are dropped, the labels are encoded, and the data is split 80/20. Features come from a term frequency–inverse document frequency (TF-IDF) vectorizer [14] that keeps 3,000 terms, drops English stop-words, and is fitted on the training text alone. We train Logistic Regression, Linear SVM, and Multinomial Naive Bayes, since all three cope well with sparse, high-dimensional input.

F. Tuning and Selection

Each algorithm is tried with two or three fixed settings (Table I). Every setting is scored by mean 5-fold cross-validation on the training partition, using weighted F1 for classification and R^2 for regression. The best setting is refit on the whole training partition and scored once on the held-out 20%. The overall winner is the algorithm with the highest cross-validated score, so the test split plays no part in choosing a model. This search is much lighter than grid or randomized search, and that is deliberate: it keeps training interactive on a laptop without a GPU.

TABLE I- MODELS AND SETTINGS TRIED

MODEL	TASK	SETTINGS TRIED
Logistic Regression	Classification / text	C in {0.1, 1, 10}
Decision Tree	Both	max_depth in {None, 5, 10}
Random Forest	Both	n_estimators in {100, 200}
SVM / SVR	Both	C in {0.1, 1, 10}
Linear Regression	Regression	none
Linear SVM	Text	C in {0.1, 1, 10}
Multinomial NB	Text	alpha in {0.5, 1.0, 2.0}

G. Explainability and Export

SHAP uses TreeExplainer for tree models and LinearExplainer for linear ones. Kernel SHAP is too slow for ordinary hardware, so for SVM and SVR the system reports permutation importance on a stored sample of training rows. LIME explains one record at a time, using predict_proba for classification and predict for regression. For text, SHAP uses its Text masker and LIME uses LimeTextExplainer, and both are skipped if the winner has no predict_proba. If the shap or lime packages are not installed, the system falls back to the model's own importances. The winning model is saved with joblib as a self-contained package. For structured data it holds the estimator, the fitted scaler, the label encoder, the fill values, and the training column order. For text it holds the estimator, the vectorizer, and the label encoder. New raw records can then be scored outside the application.

IV. EXPERIMENTS AND RESULTS

A. Setup

We ran the pipeline on Iris [15] (150 rows, 3 classes), Wine (178 rows, 3 classes), Breast Cancer Wisconsin Diagnostic (569 rows, 2 classes), and Diabetes (442 rows, continuous target). To make the data look more like a raw file, we deleted 6% of the feature values at random before training. Model selection follows Section III-F, and every score below comes from the untouched 20% test partition. The experiments run through the same core library that serves the API.

B. Structured Benchmarks

TABLE II- CROSS-VALIDATED SELECTION AND HELD-OUT RESULTS (6% MISSING VALUES)

DATASET	SELECTED MODEL	CV SCORE	TEST SCORE
Iris	Decision Tree	F1 = 0.975	F1 = 0.933
Wine	SVM	F1 = 0.993	F1 = 0.972
Breast Cancer Wisconsin	SVM	F1 = 0.974	F1 = 0.982
Diabetes	Linear Regression	R^2 = 0.437	R^2 = 0.420

Classification scores are high on all three datasets, and no single model wins everywhere: a decision tree for Iris, SVM for Wine and Breast Cancer, and linear regression for Diabetes. The R^2 of 0.420 on Diabetes

is typical for classical models on that noisy benchmark. On the unmodified Iris file, cross-validation selects SVM for species (CV F1 0.967, held-out F1 0.967) and Linear Regression for sepal length (CV R^2 0.828, held-out R^2 0.867). Logistic Regression comes within 0.001 of SVM on CV F1 (0.966) and scores a held-out F1 of 1.000 on the 30 test rows. It is the model explained in Section IV-E.

C. Text Benchmark

For the text pipeline we used a 1,083-message subset of the SMS Spam Collection [16], with 920 ham and 163 spam messages, under the same protocol. Multinomial Naive Bayes was selected (CV F1 0.962) and reached a held-out weighted F1 of 0.976 and an accuracy of 0.977.

D. Missing-Value Threshold

To check the default 40% gate, we rebuilt each structured dataset with uneven missingness: most columns 5 to 10% empty and about a quarter 45 to 75% empty. Thresholds of 20% and 40% dropped the same damaged columns on every dataset (1 of 4 on Iris, 3 of 13 on Wine, 7 of 30 on Breast Cancer, and 2 of 10 on Diabetes). At 60%, some of those columns slipped through (0, 1, 3, and 1 columns dropped). So 40% removes the damaged columns and still leaves room for columns with moderate gaps. Users can change it.

E. Explanations

Figure 3 shows the SHAP ranking for the Logistic Regression model on Iris. Petal width and petal length contribute most, and the two sepal measurements contribute less. Figure 4 shows the LIME explanation of the class versicolor for one test row. A low sepal length is the strongest negative contribution, a low petal length also pushes against the class, and sepal width and petal width add small positive amounts.

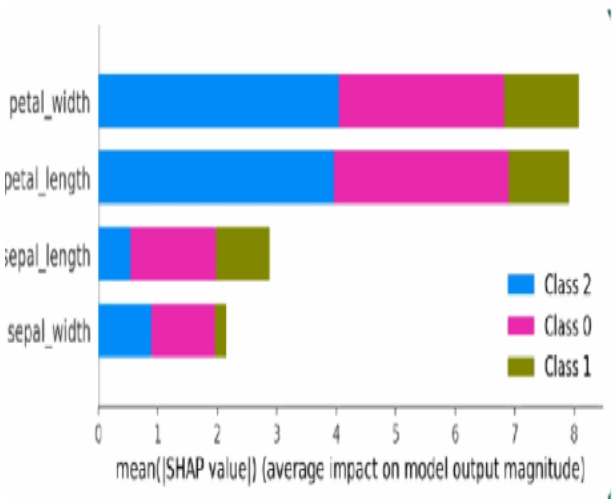


Fig. 3. SHAP global feature importance (Iris, Logistic Regression).

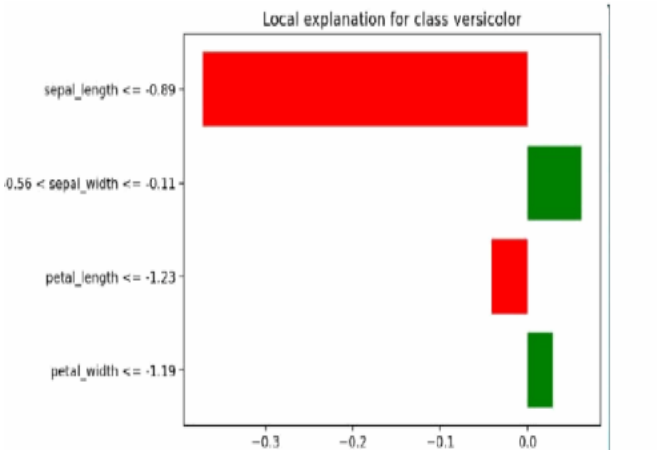


Fig. 4. LIME local explanation for class versicolor.

F. Automated Tests

The core library comes with 15 automated tests, and all of them pass. They cover reading CSV, JSON, TXT, DOCX, and PDF files (including a table pasted as plain text into a .txt or .docx file), problem-type and text-column detection, the quality gate, all three pipelines, and a 25-row dataset in which one class has only one or two members. They also check categorical alignment at prediction time (a baseline level, a level never seen in training, and a missing numeric field) and a complete upload, train, predict, and explain round trip through the service layer.

V. COMPARISON WITH EXISTING TOOLS

TABLE III- DESIGN COMPARISON WITH EXISTING AUTOML TOOLS

FEATURE	AUTO-SKLEARN	TPOT	PYCARET	AUTOSYNTH AI
Interface	Python API	Python API	Low-code Python	Browser GUI, REST API, MCP tools
Input	Tabular DataFrame	Tabular DataFrame	Tabular DataFrame	CSV, JSON, TXT, PDF, DOCX
Search	Bayesian + meta-learning	Genetic programming	Model comparison + tuning	Small fixed grid + 5-fold CV
Explain-ability	Not built in	Not built in	interpret_model (SHAP)	SHAP + LIME in the UI

FEATURE	AUTO-SKLEARN	TPOT	PYCARET	AUTOSYNTH AI
Main user	Practitioner	Practitioner	Analyst / practitioner	Non-expert

We make no claim that AutoSynth AI beats these libraries at raw optimization. What it offers is different: it takes whatever file a non-expert already has, routes it automatically, recovers tables from documents, and explains the result, and it exposes the same functions to other software through a REST API and MCP tools.

VI. CONCLUSION

The fixed grid gives up the wider search of Bayesian or genetic AutoML. In return, training stays fast on a laptop, and a user can follow what the system tried. Table II suggests the small grid is enough to make sensible, data-dependent choices. SHAP covers tree and linear winners, permutation importance covers kernel models, and LIME gives a local view, so every winning model comes with some explanation.

AutoSynth AI shows that one pipeline can carry a user from whatever file they happen to have to a trained, explained, downloadable model. Table recovery fills a gap that purely tabular tools leave open. The structured pipeline keeps a clean boundary between training and test data, and model selection never touches the test split. Splitting the system into a core library with an API, an MCP server, and a web client also means it can be embedded in other software without changes to the modeling code. Future work includes larger and more varied tables and text corpora, a controlled comparison with Auto-sklearn, TPOT, and PyCaret on identical data, a sweep of the routing thresholds like the one in Section IV-D, and a transformer-based text option.

REFERENCES:

[1] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, “Efficient and robust automated machine learning,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 28, 2015.

[2] R. S. Olson and J. H. Moore, “TPOT: A tree-based pipeline optimization tool for automating machine learning,” in *Automated Machine Learning*. Cham, Switzerland: Springer, 2019, pp. 151–160.

[3] M. Ali, “PyCaret: An open source, low-code machine learning library in Python,” 2020. [Online]. Available: https://pycaret.org

[4] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms,” in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 847–855.

[5] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning: Methods, Systems, Challenges*. Cham, Switzerland: Springer, 2019.

[6] X. He, K. Zhao, and X. Chu, “AutoML: A survey of the state-of-the-art,” *Knowl.-Based Syst.*, vol. 212, 2021.

[7] A. Mumuni and F. Mumuni, “Automated data processing and feature engineering for deep learning and big data applications: A survey,” *J. Inf. Intell.*, 2024.

[8] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?': Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 1135–1144.
[9] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017.
[10] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[11] Anthropic, "Model Context Protocol specification," 2024. [Online]. Available: https://modelcontextprotocol.io
[12] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[13] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.
[14] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, no. 5, pp. 513–523, 1988.
[15] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Ann. Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
[16] T. A. Almeida, J. M. G. Hidalgo, and A. Yamakami, "Contributions to the study of SMS spam filtering: New collection and results," in *Proc. 11th ACM Symp. Document Eng.*, 2011, pp. 259–262.
[17] Anthropic, "Claude" (large language model), 2026. [Online]. Available: https://www.anthropic.com