OMEGA: OPTIMIZING MACHINE LEARNING BY EVALUATING GENERATED ALGORITHMS

Jeremy Nixon
Infinity Artificial Intelligence Institute
San Francisco, CA, USA
jeremy@infinity.inc

Annika Singh
Infinity Artificial Intelligence Institute &
Computer Science, Stanford University
Palo Alto, CA, USA
annikaks@stanford.edu
annikas@infinity.inc

ABSTRACT

In order to automate AI research we introduce a full, end-to-end framework, OMEGA: Optimizing Machine learning by Evaluating Generated Algorithms, that starts at idea generation and ends with executable code. Our system combines structured meta-prompt engineering with executable code generation to create new ML classifiers. The OMEGA framework has been utilized to generate several novel algorithms that outperform scikit-learn baselines across a robust selection of 20 benchmark datasets (`infinity-bench`). You can access models discussed in this paper and more in the python package: `pip install omega-models`.

1 INTRODUCTION

The evolution of machine learning (ML) models has historically been driven by the manual derivation and implementation of novel algorithms. However, the transition from a theoretical hypothesis to a production-ready, validated implementation remains a high-friction process. Translating a novel, non-intuitive idea into executable code often requires extensive debugging for both coding and integration within existing pipelines. While existing automation techniques, such as Neural Architecture Search (NAS) and AutoML, have succeeded in optimizing hyperparemeters and layer choice within fixed layer types, the discovery of entirely new algorithmic logic remains largely a manual endeavor (Zoph & Le, 2016; Real et al., 2020).

In this paper, we propose a shift in how we utilize Large Language Models (LLMs) for machine learning research. Instead of treating LLM outputs as static text artifacts, we treat them as executable learning systems. We investigate whether LLMs can reason about and generate novel algorithms that are competitive with established baselines without human intervention. To facilitate this, we introduce **OMEGA**, a framework that specializes LLM synthesis for the creation of standardized, API-compliant machine learning estimators. OMEGA bridges the gap between raw code generation and systematic algorithmic evaluation, enabling a closed-loop system for algorithmic discovery.

1.1 DETAILED CONTRIBUTIONS

Our contributions are fivefold

1. We propose OMEGA: an automated end to end framework that allows developers to enter a simple prompt for a novel classification model and return compile-error free, scikit-learn compatible code that has been evaluated on our benchmarking dataset.
2. We propose `infinity-bench`: A benchmark to evaluate classifcation models on robustness and accuracy.
3. We analyze 2 (of many) novel classification models generated using OMEGA, that out perform scikit-learn baselines.
4. We present an analysis comparing 4 popular LLMs' coding ability.

5. We present results regarding recursive self prompting and code improvement across 4 LLM's used in OMEGA.

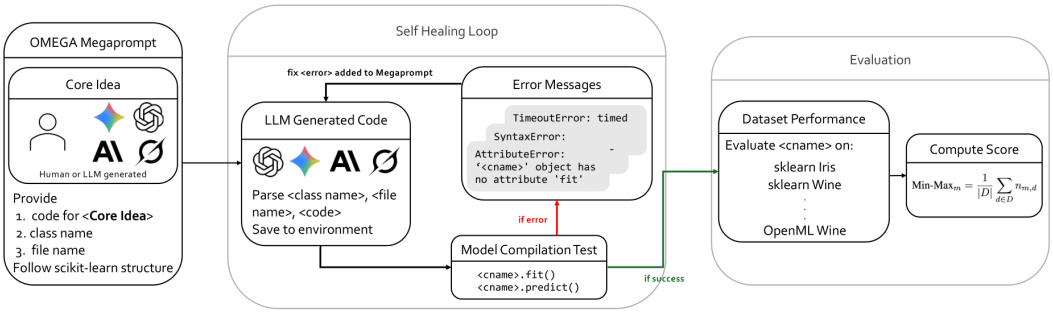


Figure 1: Core OMEGA Framework

1.2 Brief Literature Review

OMEGA is at the intersection of automated discovery, meta-learning, and neural program synthesis.

AutoML and Architectural Discovery: Traditional AutoML has primarily focused on structured search over fixed sets of algorithms and hyperparameters to optimize performance for specific data manifolds (Hutter et al., 2019). Early efforts in algorithmic generation utilized genetic programming and evolutionary algorithms to evolve programs via natural selection, though these methods often struggled with the complexity of modern ML architectures (Koza, 1992). This led to the rise of Meta-learning and Neural Architecture Search (NAS), where systems optimize learning strategies using reinforcement learning and Bayesian optimization (Zoph & Le, 2016; Snoek et al., 2012; Finn et al., 2017; Elsken et al., 2019).

Autonomous Algorithmic Discovery: Recent work has pushed discovery beyond architectural tuning toward the creation of new mathematical logic. AlphaEvolve demonstrated that machine learning algorithms could be evolved from basic primitives (Real et al., 2020). In parallel, systems like AlphaTensor and FunSearch have discovered non-intuitive and provably correct algorithms for fundamental tasks by combining deep learning with automated evaluators (Fawzi et al., 2022; Romera-Paredes et al., 2024). Furthermore, OMEGA is inspired by the emergence of "AI Scientists", which have suggested a future where the entire research pipeline, from hypothesis generation to paper writing, is fully automated (Lu et al., 2024; Akiba et al., 2024).

Program Synthesis via LLMs: Current advances in neural program synthesis have transitioned from simple code completion to functional, natural-language-driven logic generation. Benchmarks such as HumanEval have established the baseline for functional correctness in LLM-generated code (Chen et al., 2021). Modern execution environments now enable iterative debugging and closed-loop execution within the model's workflow (OpenAI, 2023b;a). OMEGA builds on this shift by directing this generative capacity toward the synthesis of industry-standard, scikit-learn-compatible machine learning models.

2 Methods

The OMEGA framework mirrors a code generation framework many engineers manually utilize. We ingest a prompt that is either entered by a user or LLM generated using some initial model inspiration. This is used to generate code that will go through a self-healing pipeline to ensure execution capability before being evaluated on our benchmark. We dive into each of these steps below.

2.1 IDEA GENERATION

To generate executable algorithms, we adopt two complementary approaches: autonomous hypothesis generation by language models and human-submitted algorithmic ideas.

LLM Ontology Search: To trigger an LLM generating ideas itself, we provided a list of Models and Research Principles. For each of the models provided, we prompted the LLM to use the principles to generate 10 unique, novel ways to modify the base models. We parse through the list of ideas and use them as individual prompts for actual code generation. This mirrors prior work on meta-learning and computational creativity, where systems explore structured spaces of solutions (Vilalta & Drissi, 2002; Colton & Wiggins, 2008).

Human Prompted Ideas: Simultaneously, we allowed humans to submit prompts to OMEGA regarding novel ideas they had for new models. These ideas were entered in the framework at the same point as the parsed ideas from LLM generation.

2.2 CODE GENERATION

Once the LLMs returned the responses for the prompts, the responses were parsed to extract the executable code. During this stage, we prioritized two things (1) easy integration with existing pipelines and (2) eliminating the need to revise the algorithm.

Scikit-Learn Design Patters: For the former goal, we enforce scikit-learn design patterns by requiring generated models to inherit from `BaseEstimator` and implement `.fit()` and `.predict()`. This aligns with established API conventions that prioritize composability, evaluation consistency, and reproducibility (Pedregosa et al., 2011; Buitinck et al., 2013). We treat scikit-learn as a domain-specific language (DSL) for algorithm generation. Thus, when using the algorithms from the package, it is easy to integrate into existing scikit-learn workflows.

Self-Healing: For the latter goal, the framework includes a self-healing mechanism in which the error stacktrace is captured and fed back into the generation loop. This is inspired by similar execution-based validation strategies that have proven critical for reliable code synthesis and reproducible research (Gundersen & Kjensmo, 2018; Aho et al., 2006; Chen et al., 2021). After a fixed set of retries, if the code fails to self heal, we don't return any code and instead ask the user to try again. This process is required to ensure that all classifiers that we publish are error-free, as unexamined code is often useless (Wang et al., 2025).

2.3 EVALUATION AND INFINITY-BENCH

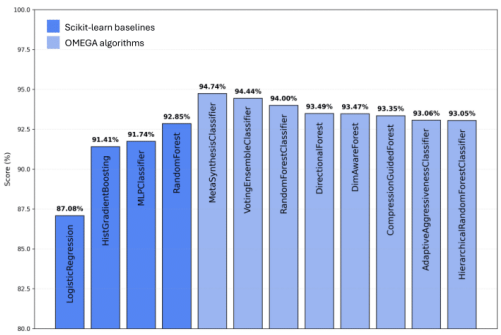
Each valid model is evaluated on 20 classification datasets sourced from scikit-learn and OpenML (Vanschoren et al., 2014). To account for the variation in the datasets, we use min-max normalized accuracy and aggregate performance between datasets, following best practices for multi-data set evaluation (Demšar, 2006).

Performance Score: Performance is purely evaluated on the basis of classification accuracy. However, we quickly found that certain included datasets were easier to perform well (many models were able to achieve near 100% accuracy). Since our datasets have varying difficulty levels, we rank ranking algorithms relative to each other on each dataset, then aggregating these rankings (Brazdil et al., 2009). This allows us to compare relative performance rather than absolute performance to account for dataset difficulty. The accuracy for each model on each is calculated as follows:

$$n_{m,d} = \frac{s_{m,d} - \min_d}{\max_d - \min_d}$$

Where $s_{m,d}$ is the accuracy of the model on dataset d, and min_d and max_d are the scores of the worst and the best models, respectively. We then take the average score of the model across all 20 datasets to compute the min-max score per model:

$$\text{Min-Max}_m = \frac{1}{|D|} \sum_{d \in D} n_{m,d}$$



Rank	Model	MinMax	Generator
1	MetaSynthesisClassifier	0.9474	User
2	VotingEnsembleClassifier	0.9445	User
3	RandomForestClassifier	0.9401	User
4	DirectionalForest	0.9350	System
5	DimAwareForest	0.9347	System
6	CompressionGuidedForest	0.9336	System
7	AdaptiveAggressivenessClassifier	0.9307	System
8	HierarchicalRandomForestClassifier	0.9305	User
...	...	...	...
11	RandomForest	0.9285	Scikit-Learn
19	MLPClassifier	0.9174	Scikit-Learn
21	HistGradientBoosting	0.9141	Scikit-Learn
59	LogisticRegression	0.8708	Scikit-Learn

Figure 2: Best Models vs Scikit-Learn Baselines (Min-Max Score)

Dataset Diversity: By only focusing on classification, we were able to ensure

1. models were properly formatted (important for the self healing property)
2. models were easily comparable

However, within classification, we ensured that we had dataset diversity via numerical and categorical features, varying dataset size, binary and multiclass data that spanned various fields including bio, medicine, education, etc.

While our models haven't been tested on differing types of datasets (i.e., image/video), the core OMEGA framework can virtually be abstracted to any use case and is only constrained by the LLM's ability to handle complexity. While we recommend using the models presented in this paper and in our python package for classification tasks, we would encourage users to utilize the OMEGA framework for any other use cases.

Benchmark Dataset With recent agentic developments, particularly with coding agents, algorithmic development within existing environments has become far easier. In order to establish a universal way to test classification model creation, we propose the `infinity-bench`, a repository containing our these datasets that can be used to test newly generated classification models on and evaluate and compare their performance.

2.4 Library Creation

OMEGA democratizes the models that are created by including the top models in a python package (`omega-models`) that can be easily imported and utilized on other datasets. By ensuring that models take the format of scikit-learn packages, these imports can be seamlessly integrated with existing workflows.

3 OMEGA-Generated Model Results

3.1 Top Generated Models

Above (Figure 2) we include the top performing algorithms as well as their aggregate score compared to the scikit-learn baseline algorithms (shown in blue).

These figures demonstrate the viability of the OMEGA framework for the ideation and creation of classification modules. Below we do a deeper dive into two of the generated algorithms, one that is human prompted (#1. `MetaSynthesisClassifier` Section 3.2) and one that was autonomously LLM prompted (#4. `Directional Forest` Section 3.3) to better understand the novelty of algorithms generated by OMEGA). Models generated by Anthropic's Claude Sonnet 4.5.

3.2 METASYNTHESISCLASSIFIER

The `MetaSynthesisClassifier` represents a meta-learning approach to ensemble synthesis, specifically utilizing a stacked generalization architecture. Stacking is an ensemble technique in which a meta-learner is trained to optimally combine the predictions of several base models to improve generalization (Wolpert, 1992). The core objective is to have base learners and then treat the combination of these learners as a *secondary* supervised learning task. The code can be found in Appendix A.1.

3.2.1 ALGORITHM

We define the system as a hierarchical set $\mathcal{H} = \{\mathcal{E}, M_\psi\}$, where $\mathcal{E} = \{E_1, E_2, \ldots, E_m\}$ is a set of heterogeneous base estimators (e.g., Logistic Regression, Random Forest, and Decision Trees) and M_ψ is the meta-estimator responsible for optimal synthesis.

Meta-Feature Generation: For a training sample $(\mathbf{x}_i, y_i)$, the meta-feature vector $\mathbf{z}_i$ is constructed by concatenating the probability outputs of the base estimators, essentially storing the opinion of all the base estimations:

$$\mathbf{z}_i = [P(y \mid \mathbf{x}_i; E_1^{(-k)}), \ldots, P(y \mid \mathbf{x}_i; E_m^{(-k)})] \tag{1}$$

where $E_j^{(-k)}$ is trained on a subset of data that excludes sample i. This is mainly done to prevent data leakage and ensure the meta-classifier will not inherit the training-set bias of base learners and *actually* learn the relative reliability and error correlations of the generated algorithms based on their performance on previously unseen instances.

Meta-Level Synthesis: Once the meta-features $\mathbf{Z}$ are generated for the entire training set, the meta-estimator M_ψ is trained on the mapping from the base learners' collective predictions to the true labels. This optimization problem is defined as:

$$\min_{\psi} \sum_{i=1}^{n} \mathcal{L}(y_i, f(\mathbf{z}_i; \psi)) \tag{2}$$

This allows M_ψ to learn which base learners are most reliable for specific patterns in the prediction space, effectively weighting their influence based on their historical accuracy during the cross-validation stage

Final Inference: During inference, a test sample $\mathbf{x}^*$ is passed through the fully-trained base ensemble $\mathcal{E}$ to produce the latent vector $\mathbf{z}^*$. The final prediction $\hat{y}$ is then derived from the meta-classifier's evaluation of that synthesized vector:

$$\hat{y} = \text{argmax}_{c \in C} P(y = c \mid \mathbf{z}^*; M_\psi) \tag{3}$$

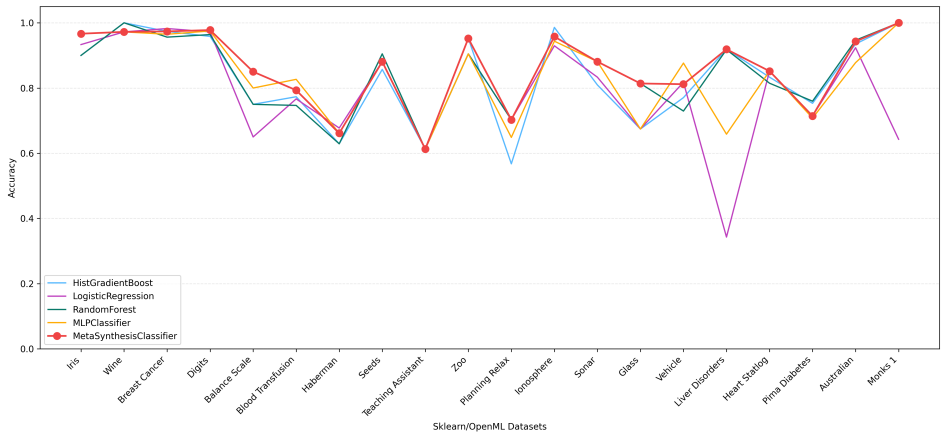


Figure 3: `MetaSynthesisClassifier` vs Scikit-Learn Individual Dataset Scores

By learning to synthesize the diverse biases of the base learners, the `MetaSynthesisClassifier` achieves a lower generalization error than individual learners by finding the optimal manifold in the prediction space.

3.2.2 RESULTS

As seen in Figure 3 `MetaSynthesisClassifier` demonstrates exceptional robustness. By adaptively learning to combine the strengths of linear models and tree-based ensembles, it successfully navigates different manifold complexities across the 20 benchmark datasets.

3.3 DIRECTIONALFOREST

The `DirectionalForest` incorporates feature directionality into an ensemble of decision trees to align the input space with class-specific statistical deviations. This algorithm was generated by the OMEGA framework to investigate whether pre-calculating feature orientations can improve the split efficiency of standard ensemble methods (Masoomi et al., 2023). The code can be found in Appendix A.2.

3.3.1 ALGORITHM

We define the forest as an ensemble $\mathcal{E} = \{T_1, T_2, \ldots, T_n\}$, where each T_i is a decision tree trained on a modified feature space. The core mechanism is the computation of a directionality vector $\mathbf{d} \in \{-1, 0, 1\}^f$ that rescales features based on their relationship with class means.

Feature Directionality Calculation: For a training set (X, y), we first compute the mean vector for each class $c \in C$, denoted as $\boldsymbol{\mu}_c$, and the global mean of the dataset $\boldsymbol{\mu}_g$. The directionality vector $\mathbf{d}$ is calculated as the sign of the sum of deviations:

$$\mathbf{d} = \text{sgn}\left(\sum_{c \in C} (\boldsymbol{\mu}_c - \boldsymbol{\mu}_g)\right) \tag{4}$$

Directional Transformation: Before training each tree T_i and during inference, the input features $\mathbf{x}$ are transformed into a directional space $\mathbf{x}_{dir}$ via an element-wise product with the directionality vector:

$$\mathbf{x}_{dir} = \mathbf{x} \odot \mathbf{d} \tag{5}$$

This transformation serves as a primitive form of feature engineering that attempts to orient the features such that splits in the decision trees are more discriminative relative to the global mean.

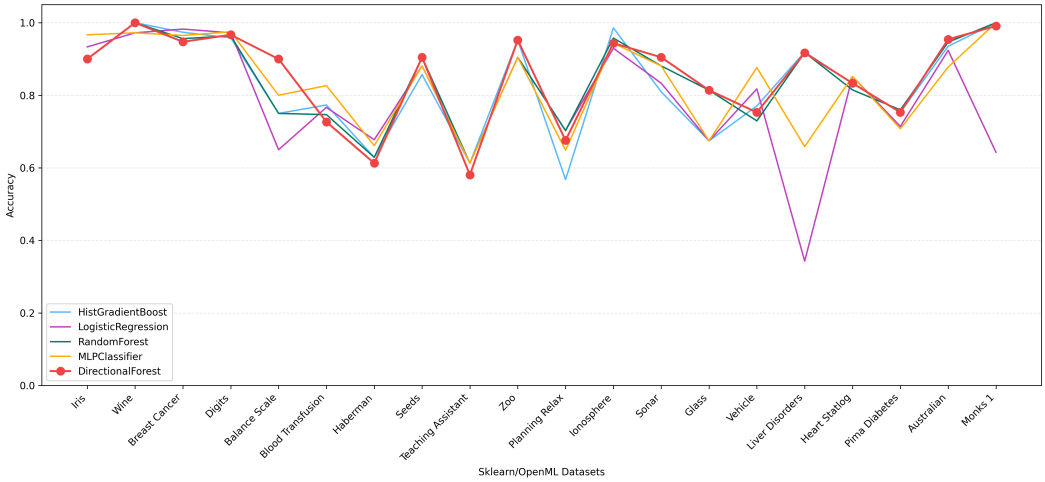


Figure 4: `DirectionalForest` vs Scikit-Learn Individual Dataset Scores

Final Inference: The forest utilizes a plurality voting mechanism to determine the final class label. For a set of predictions from the ensemble $\{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n\}$, the final prediction $\hat{y}$ is:

$$\hat{y} = \text{mode}(\{T_i(\mathbf{x} \odot \mathbf{d})\}_{i=1}^{n}) \tag{6}$$

The algorithm essentially enforces a consistent orientation of the feature space across all learners. By calculating the aggregate sign of class-wise deviations, it attempts to normalize the direction of feature importance, allowing the decision trees to focus on capturing non-linear interactions rather than discovering feature polarity.

3.3.2 Results

A closer look at the evaluation metrics shows that the `DirectionalForest` demonstrates significant stability across high-dimensional datasets. While its linear directionality assumption is simpler than standard Random Forests, it effectively reduces variance in specific manifolds, outperforming Scikit-learn baselines on a majority of the 20 benchmark datasets used for evaluation.

4 Comparing LLMs within OMEGA

Once confirming that this framework could be utilized to generate novel algorithms, we conducted a secondary experiment to determine if the LLM utilized to generate the idea for the algorithms and then subsequently write the code made an impact on how well the model performed.

In this experiment, we tested four of the most popular code-generation LLMs: **Anthropic's Claude Sonnet 4.5 model, OpenAI's GPT-4.1 mini, Google's Gemini 2.5 Flash, and xAI's grok-code-fast-1**. (The paper up until this point has used Anthropic's Claude Sonnet 4.5 model exclusively). Table 5 shows four of the LLM's performances on the top performing (by average of all LLMs) prompts.

Across the top ten prompts, Gemini performs the best in 6, GPT in 2, and Claude and Grok both in 1. Notably the performance across the four LLMs is quite comparable across all these prompts perhaps alluding to the idea that prompt quality is more important than the actual LLM being used. We do a deeper comparison on whether asking the LLM to iterate on the code vs the prompt performs better in Section 5

While Gemini 2.5 Flash was by far the best model at generating executable classification models, this doesn't necessarily correlate with the novelty of code or the ability for these LLM's to "long-horizon" reason and is solely evaluated on how well their generated classification models perform.

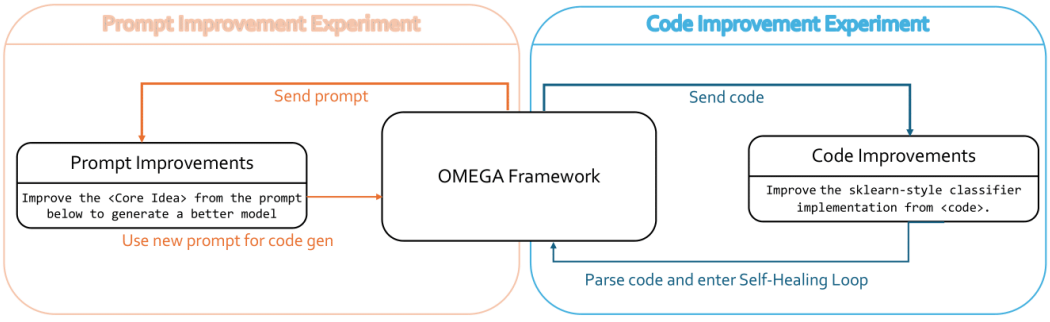


Figure 5: OMEGA Prompt & Code Improvement Experiments

5 Self-Improving Prompts vs Code

To further push the capabilities of the OMEGA framework, we implemented a self-improving loop; see Figure 5. We iterated on the base architecture, so all the initially generated algorithms would:

Table 1: Scores for Model Generation Prompts Across LLMs
(bolded by best performing LLM per prompt)

PID	Prompt Summary	Gemini	OpenAI	Claude	xAI
P01	Design a framework for incorporating domain knowledge into the Random Forests classifier through the use of abstraction layers and feature engineering.	**0.9307**	0.9295	0.9202	0.9123
P02	Create a Random Forests classifier that can handle both biased and unbiased data by incorporating a bias-variance decomposition step into the training process.	0.9236	0.9179	0.9202	**0.9290**
P03	Dynamically adjust the degree of feature sub-sampling randomness for each tree based on the predictability or inherent noise level of its bootstrap sample.	**0.9281**	0.9186	0.8456	0.9202
P04	Control the degree of randomness (e.g., bootstrap sample size, feature subspace size) for each tree in the ensemble to explicitly target different bias-variance tradeoffs, creating a heterogeneous mix of learners.	0.9208	**0.9268**	0.8315	0.9229
P05	Design multi-directional split criteria that consider both forward and backward feature dependencies.	**0.9113**	0.8849	0.8900	0.8826
P06	Create bags with controlled bias-variance profiles by mixing shallow high-bias and deep high-variance trees in learned proportions.	**0.9177**	0.8756	0.8991	0.9141
P07	Create adaptive random subspace selection where feature subset size varies per tree based on estimated intrinsic dimensionality of the bootstrap sample.	**0.9322**	0.8816	0.8456	0.9134
P08	Apply entropy-guided feature selection to prioritize features with high information gain.	0.8901	0.8898	**0.8914**	0.8887
P09	Use hierarchical abstraction layers where shallow trees capture coarse patterns and deep trees capture fine details.	**0.9161**	0.9109	0.8653	0.8578
P10	Integrate randomness control by varying the degree of noise injected into feature values during tree construction.	0.8913	**0.9036**	0.8468	0.8758
	Average Performance	**0.9162**	**0.9039**	**0.8736**	**0.9017**

1. Generate a new prompt using the initial generation prompt as context

2. Generate new code using the old code as context

In the table below, we present these results of this experiment (conducted independently of the one shown in Figure 4). The **Average Score at Each Step** contains the average score of models generated from the base generation, models generated from prompt improvements, and models generated from code improvements. The **% Improvement** contains the percentage improvement from base performance to prompt improvements and base performance to code improvements.

Table 2: Avg Scores and Percentage Improvement (bolded by best strategy for improvement)

Model	Average Score			% Improvement	
	Base	Prompt	Code	Prompt	Code
Gemini	0.885	0.894	0.890	**0.90%**	0.53%
GPT	0.734	0.780	0.769	**4.59%**	3.50%
Claude	0.770	0.859	0.818	**8.88%**	4.85%
Grok	0.778	0.763	0.798	-1.54%	**1.95%**

The results reveal that prompt tuning and optimization outperform pure code optimization for nearly all LLMs, to varying degrees (note that part of the variance in percentage performance will be related to the initial performance between models) (Gong et al., 2025). This aligns with the vast amount of prompt-tuning research conducted to prove that fine-tuning prompts can result in significant improvements (Shin et al., 2023; Liu et al., 2025; Peng et al., 2024)

6 CONCLUSION

In this work, we introduce OMEGA, a framework for end-to-end classification model generation and evaluation. We demonstrate the efficacy of OMEGA by presenting models generated by human and LLM prompting in this framework that are novel and outperform scikit-learn baselines on our evaluation benchmark. We further analyze the best models to use for model generation and the best strategies for self-improvement. We believe that this framework, and similar frameworks that will follow, will open up the door for LLM-generated ideation and code at scale and by technical and non technical engineers.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2nd edition, 2006.

Takuya Akiba, Makoto Shing, Yasuhiro Majima, and Shinya Kaneko. Evolutionary optimization of model merging recipes. *arXiv preprint arXiv:2403.13187*, 2024.

Pavel Brazdil, Christophe Giraud-Carrier, Carlos Soares, and Ricardo Vilalta. *Metalearning: Applications to Data Mining*. Cognitive Technologies. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-540-73262-4. doi: 10.1007/978-3-540-73263-1.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. Api design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Simon Colton and Geraint A. Wiggins. Computational creativity: The final frontier? In *Proceedings of the 20th European Conference on Artificial Intelligence*, pp. 21–26, 2008.

Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Alhussein Fawzi et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pp. 1126–1135. PMLR, 2017.

Jingzhi Gong, Rafail Giavrimis, Paul Brookes, Vardan Voskanyan, Fan Wu, Mari Ashiga, Matthew Truscott, Mike Basios, Leslie Kanthan, Jie Xu, and Zheng Wang. Tuning llm-based code optimization via meta-prompting: An industrial perspective, 2025. URL https://arxiv.org/abs/2508.01443.

Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pp. 1644–1651, 2018.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2019.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

Yuanye Liu, Jiahang Xu, Li Lyna Zhang, Qi Chen, Xuan Feng, Yang Chen, Zhongxin Guo, Yuqing Yang, and Cheng Peng. Beyond prompt content: Enhancing llm performance via content-format integrated prompt optimization. *arXiv preprint arXiv:2502.04295*, 2025.

Chris Lu et al. The ai scientist: Towards fully automated machine learning scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Aria Masoomi, Davin Hill, Zhonghui Xu, Craig P Hersh, Edwin K. Silverman, Peter J. Castaldi, Stratis Ioannidis, and Jennifer Dy. Explanations of black-box models based on directional feature interactions, 2023. URL https://arxiv.org/abs/2304.07670.

OpenAI. Assistants api. OpenAI Documentation, 2023a.

OpenAI. Chatgpt code interpreter. OpenAI Blog, 2023b.

Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Cheng Peng, Xi Yang, Kaleb E. Smith, Zehao Yu, Aokun Chen, Jiang Bian, and Yonghui Wu. Model tuning or prompt tuning? a study of large language models for clinical concept and relation extraction. *Journal of Biomedical Informatics*, 153:104630, 2024.

Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *PMLR*, pp. 8007–8017, 2020.

Bernardino Romera-Paredes et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine-tuning: An empirical assessment of llms for code. *arXiv preprint arXiv:2310.10508*, 2023.

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pp. 2951–2959, 2012.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2014.

Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

Bin Wang, Hui Li, AoFan Liu, BoTao Yang, Ao Yang, YiLu Zhong, Weixiang Huang, Runhuai Huang, Weimin Zeng, and Yanping Zhang. Reflexgen:the unexamined code is not worth using. In *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1–5. IEEE, April 2025. doi: 10.1109/icassp49660.2025.10890824. URL http://dx.doi.org/10.1109/ICASSP49660.2025.10890824.

David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

A Appendix

A.1 Source Code: MetaSynthesisClassifier

```
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin, clone
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
from sklearn.utils.multiclass import unique_labels
from sklearn.model_selection import cross_val_predict
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

class MetaSynthesisClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, base_estimators=None, meta_estimator=None, cv=5,
                 use_probas=True, use_original_features=False):
        self.base_estimators = base_estimators
        self.meta_estimator = meta_estimator
        self.cv = cv
        self.use_probas = use_probas
        self.use_original_features = use_original_features
```

```
    def fit(self, X, y):
        X, y = check_X_y(X, y)
        self.classes_ = unique_labels(y)
        self.n_features_in_ = X.shape[1]

        if self.base_estimators is None:
            self.base_estimators_ = [
                LogisticRegression(max_iter=1000, random_state=42),
                RandomForestClassifier(n_estimators=100, random_state=42),
                DecisionTreeClassifier(random_state=42)
            ]
        else:
            self.base_estimators_ = [clone(est) for est in self.base_estimators]

        if self.meta_estimator is None:
            self.meta_estimator_ = LogisticRegression(max_iter=1000, random_state=42
        else:
            self.meta_estimator_ = clone(self.meta_estimator)

        meta_features = self._generate_meta_features(X, y)

        for estimator in self.base_estimators_:
            estimator.fit(X, y)

        if self.use_original_features:
            meta_features = np.hstack([X, meta_features])

        self.meta_estimator_.fit(meta_features, y)
        return self

    def predict(self, X):
        check_is_fitted(self)
        X = check_array(X)
        meta_features = self._generate_meta_features_predict(X)

        if self.use_original_features:
            meta_features = np.hstack([X, meta_features])

        return self.meta_estimator_.predict(meta_features)

    def predict_proba(self, X):
        check_is_fitted(self)
        X = check_array(X)
        meta_features = self._generate_meta_features_predict(X)

        if self.use_original_features:
            meta_features = np.hstack([X, meta_features])

        if hasattr(self.meta_estimator_, 'predict_proba'):
            return self.meta_estimator_.predict_proba(meta_features)
        else:
            raise AttributeError("Meta■estimator■does■not■support■predict_proba")

    def _generate_meta_features(self, X, y):
        meta_features_list = []
        for estimator in self.base_estimators_:
            if self.use_probas and hasattr(estimator, 'predict_proba'):
                cv_preds = cross_val_predict(
```

```
                    estimator, X, y, cv=self.cv, method='predict_proba'
                )
                meta_features_list.append(cv_preds)
            else:
                cv_preds = cross_val_predict(estimator, X, y, cv=self.cv)
                meta_features_list.append(cv_preds.reshape(-1, 1))

        return np.hstack(meta_features_list)

    def _generate_meta_features_predict(self, X):
        meta_features_list = []
        for estimator in self.base_estimators_:
            if self.use_probas and hasattr(estimator, 'predict_proba'):
                preds = estimator.predict_proba(X)
                meta_features_list.append(preds)
            else:
                preds = estimator.predict(X)
                meta_features_list.append(preds.reshape(-1, 1))

        return np.hstack(meta_features_list)
```

A.2 IMPLEMENTATION OF DIRECTIONALFOREST

```
    import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
from sklearn.utils.multiclass import unique_labels
from sklearn.tree import DecisionTreeClassifier

class DirectionalForest(BaseEstimator, ClassifierMixin):
    def __init__(self, n_estimators=100, max_depth=None, min_samples_split=2,
                 min_samples_leaf=1, max_features='sqrt', random_state=None):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.max_features = max_features
        self.random_state = random_state

    def fit(self, X, y):
        X, y = check_X_y(X, y)

        self.classes_ = unique_labels(y)

        self.feature_directions_ = self._calculate_feature_directions(X, y)

        self.estimators_ = []

        for _ in range(self.n_estimators):
            tree = self._grow_tree(X, y)
            self.estimators_.append(tree)

        return self

    def predict(self, X):
        check_is_fitted(self)

        X = check_array(X)

        X_directional = X * self.feature_directions_

        predictions = np.array([tree.predict(X_directional) for tree in self.estimat

        return np.apply_along_axis(lambda x: np.argmax(np.bincount(x)), axis=0, arr=

    def _grow_tree(self, X, y):
        tree = DecisionTreeClassifier(
            max_depth=self.max_depth,
            min_samples_split=self.min_samples_split,
            min_samples_leaf=self.min_samples_leaf,
            max_features=self.max_features,
            random_state=self.random_state
        )

        X_directional = X * self.feature_directions_

        tree.fit(X_directional, y)

        return tree
```

```
def _calculate_feature_directions(self, X, y):
    class_means = [np.mean(X[y == c], axis=0) for c in self.classes_]

    overall_mean = np.mean(X, axis=0)

    directions = np.sign(np.sum([cm - overall_mean for cm in class_means], axis=

    return directions
```