AutoPAC- Automated Plan Synthesis and Code Generation for Machine Learning Ideation

Aryan Sahu[1], Harshvardhan Mestha[1], Sarvesh B[1], and Dhruv Shah[1]

[1] BITS Pilani,Goa,India
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

Abstract. Many machine learning practitioners find the rapid pace of research overwhelming, with numerous new papers being published every day at various venues. There are approaches in the existing literature to retrieve papers potentially relevant to an ongoing ML project. However, most of these approaches do not explicitly articulate the exact nature of how a particular idea in a paper may be of interest. In this paper, we assume the availability of a high-level description of the project, including the documentation of its current implementation and data used, the techniques formerly or currently employed, the challenges associated with the project, and the code or pseudo-code (or both) of the current implementation, provided by the practitioner. We have designed AutoPAC, an LLM (Large Language Model) based pipeline with the aim to assist the practitioner in producing a relevant list of ideas or techniques from the available set of related or relevant papers that could possibly address some of the defined challenges of the project or indicate if the idea(s) are not applicable to address any of the challenges. The pipeline also identifies if an idea is applicable in silos or a novel technique is elicited by composing ideas from multiple papers which together may resolve a challenge. AutoPAC further aims at enhancing the existing implementation of the project provided by the practitioner by generating the code for the devised techniques produced by our pipeline, in an automated manner. Code generation helps in accelerating the process of incorporating the new idea or technique with minimal intervention by a machine learning specialist. Our work demonstrates its effectiveness by enabling the generation of high-quality code implementations in a continual setting by taking two datasets and four papers into consideration, using two LLMs, and manually evaluating the outcome, thereby facilitating their seamless integration and application.

Keywords: Large Language Models · Machine Learning· Code Generation· Continual Learning.

1 Introduction

2 Background

2.1 LLM-based Code Generation

Recent advancements in Large Language Models have demonstrated strong capabilities in code generation and modification. These models, trained on vast corpora of natural language and programming code, have shown remarkable ability to understand and generate complex software implementations.

CodeCoT [3] introduced a chain-of-thought prompting approach to enhance code generation, enabling LLMs to break down complex programming tasks into manageable steps. This method significantly improved the quality and correctness of generated code, particularly for challenging problems requiring multi-step reasoning.

AgentCoder [4] expanded on this concept by implementing a multi-agent framework for code generation. By simulating different roles in the software development process (e.g., architect, programmer, tester), AgentCoder demonstrated improved code quality and bug detection capabilities compared to single-agent approaches.

AceCoder [5] focused on the task of code modification and refactoring. By leveraging existing codebases as context, AceCoder showed proficiency in adapting and improving code to meet new requirements or optimize performance.

While these approaches have significantly advanced the field of automated code generation, they primarily focus on generating code from scratch or modifying existing code based on specific instructions. They do not address the challenge of rapidly integrating new research ideas into existing codebases, which is the primary focus of AutoPAC.

2.2 Continual Learning in Machine Learning

Continual learning, also known as lifelong learning or incremental learning, aims to develop systems that can adapt to new information without forgetting previously learned knowledge. This paradigm is crucial for creating AI systems that can evolve and improve over time, similar to human learning [6]. Our work draws inspiration from these continual learning methods to create a system capable of implementing and incorporating new papers without the need for model retraining or fine-tuning. By leveraging the generalization capabilities of large language models and structuring the input appropriately, AutoPAC aims to achieve a form of continual learning in the domain of code generation and research implementation.

2.3 Time Series Datasets

Time series data involves updating models regularly as and when technology improves, making it difficult to keep a pipeline updated with the latest techniques. In this paper we closely look at two datasets containing time series data - Numerai, and a sample dataset used by us interally which we refer to as TSD (Template Stock Data).

- **Numerai** - This is a data science competition where the task is to predict the stock market. It is a highly documented datasets with many tutorials and guides available. It has thousands of features and 5 classes to predict. The dataset changes continually with new datapoints added every week and new features being added with the various versions of the datasets being released. This is an optimal dataset for AutoPAC due to the requirement to adapt to frequent dataset changes
- **TSD (Template Stock Data)** - This is our internal dataset that we use to test the robustness of our archtechture. It is also time series data, with features and classes different from that of Numerai. It is has low documentation, making it a more difficult dataset to use for an LLM.

3 Methodology

AutoPAC operates through a two-phase process: Plan Generation and Code Generation. These phases work in tandem to translate research ideas into implementable code, leveraging the power of Large Language Models (LLMs) at each step It addresses the challenge of automatically integrating new machine learning techniques into an existing continual learning pipeline, whether it involves updating models or datasets. Importantly, these phases can be iterated multiple times, allowing for the incorporation of ideas from multiple papers sequentially. This iterative approach enables continuous refinement and expansion of the implementation based on new research insights. While our experiments focused on two iterations, the process can theoretically be repeated indefinitely. In the following sections, we provide a comprehensive explanation of our Plan Generator and Code Generator, detailing each component thoroughly.

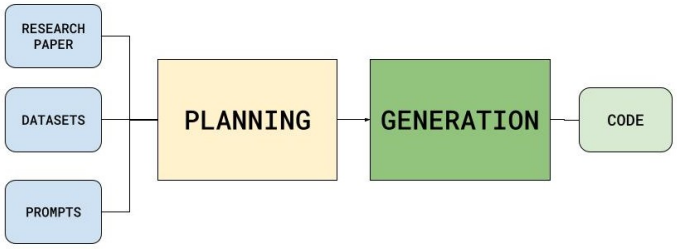

Fig. 1. Overview of AutoPAC pipeline

3.1 Plan Generation

The Plan Generation phase forms the conceptual foundation of AutoPAC, transforming research papers and high-level ideas into structured plans for implementation. This process involves several key steps, each designed to distill and organize the essential information from the input sources. The plan generation process involves a multi step process, similar to how one reads a paper and analyses it, inspired from [1]. The first step involves providing the paper as context to an LLM. We then use a prompt to summarise the paper. We provide some high level idea as context, this includes information about the dataset and model and ask the LLM to produce a methodology using the paper summary and the high level as context, and ultimately ask it to refine its methodology which gives us the plan as the output.

3.2 Providing Context

The user needs to provide two inputs to AutoPAC. The Paper and the Idea

- **Paper** - The paper should be relevant to the problem, otherwise AutoPAC may produce logically incorrect code due to hallucination.
- **Idea** - This is the Idea or the approach to be used by the user, this involves specifying the details of the dataset, such as the presence of missing values, pre processing, choosing subsets etc. The context of the model also needs to be provided, for example the abstract of the paper along with certain limitations in using the model with a dataset, such as handling missing values, size limitations, data type handling etc. The model is then instructed to incorporate the dataset and model with the given paper keeping in mind the specified limitations.

3.3 Analysing the Paper

Once the context is established, AutoPAC employs a multi-stage analysis process to extract and synthesize key information from the provided research paper. This systematic approach ensures that critical methodological insights are captured and integrated into the final plan. The paper is analysed in various stages. As seen in figure 2 the Analyse section instructs the LLM to identify the important steps of the paper, and extracts the methodology of the paper. The create methodology combines the Idea provided by the user and combines the Idea and the paper's methodology by asking the LLM to find connections, and build a new methodology that leverages the strengths of the paper while accommodating the dataset and model. Finally the methodology is refined which verifies if the limitations in the method are addressed, comments on its feasibility, and most importantly provides psuedocode for the final method.

Data: L : an LLM, P :Paper, D : Dataset, M :Model, H :High Level Idea, C : Context/Memory of the LLM, *out* :Output produced by the LLM
Result: PLN : Plan used for Code Generation
$C \leftarrow \phi$;
append P to C;
$out = summarise(L, P)$;
append *out* to C;
$out = analyse(L, P, C)$;
append *out* to C;
$out = generate\_methodology(L, P, C, H(D, M))$;
append *out* to C;
$PLN = refine\_methodology(L, P, C, H(D, M))$;
return PLN;

Algorithm 1: Plan Generation

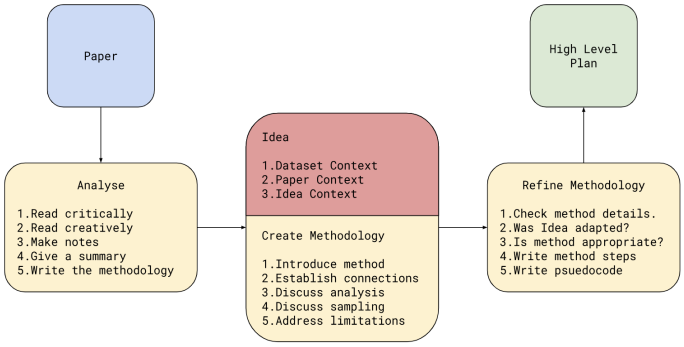

Fig. 2. Overview of plan generation pipeline

3.4 Code Generation

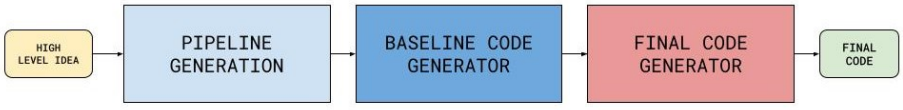

Fig. 3. Overview of the Code Generator

In this section, we describe the structure of the Code Generator along with its components. Figure 3 presents an overview of the Code Generator, our proposed LLM-facilitated pipeline for code generation. The system comprises three core components:

1. **Pipeline Generation:** This module generates a pipeline to be followed for generating the code as shown in figure 4
   - **Input:** High-level specification of the task or problem generated from the *Plan Generator*.
   - **Output:** Detailed steps and structure of the code generation pipeline.
2. **Baseline Code Generation:** This module generates code from the provided base high-level idea,as shown in figure 5.
   - **Input:** High-level idea and pipeline steps from the Pipeline Generation module.
   - **Output:** Baseline code that serves as an initial implementation.

Step/Prompt	Description
summarise	Giving LLM the paper, either PDF or raw text, without any additional prompt
analyse	Telling the LLM that it is a researcher and you are doing your literature review, you have have been provided with a systematic approach on how to read a paper. Uses Chain of Thought to analyse the methodology of the paper
generate_methodology	This prompt contains the context of the Dataset and Model that we want to apply the paper on. It also contains the instructions to produce an effective methodology given all the context of the paper, dataset and model.
refine_methodology	This prompt is to refine the methodology produced with generate_methodology, and produces a a brief methodology, and its psuedocode which is given as input to the Code generator

Table 1. Details of various stages of Plan Generation.

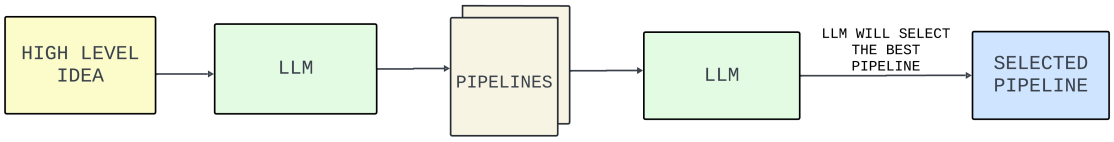

Fig. 4. Pipeline Generator

3. **Final Code Generation:** This module generates code from the modified high-level idea and uses the baseline code as sample code,as shown in figure 6.
   - **Input:** Modified high-level idea and the Baseline code.
   - **Output:** Final, refined code that meets the specific requirements.

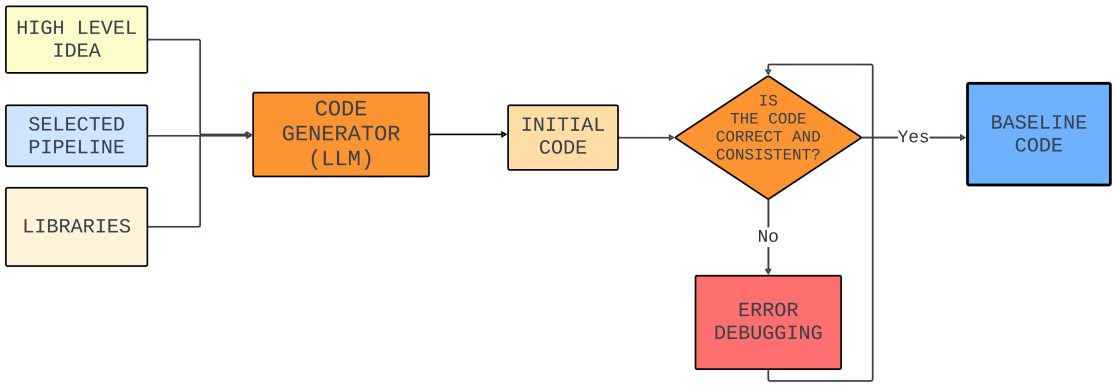

Fig. 5. Overview of the Baseline Code Generator

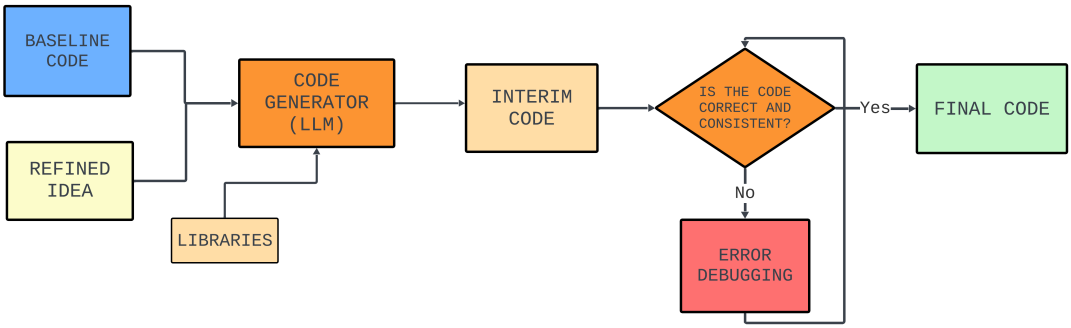

Fig. 6. Overview of the Final Code Generator

The following sections introduce the details of the pipeline.

Require: $L \leftarrow \{\text{gpt-4}, \text{gemini}\}$, PLN: High-Level Idea, $Libs$: Libraries, out: Output produced by the LLM

Ensure: $debugged\_code$

0: **/* Initialize High-Level Idea and Libraries** */
0: $PLN \leftarrow$ "Plan used for Code Generation"
0: **/* Generate Pipelines from High-Level Idea** */
0: $pipelines \leftarrow L.generate\_pipelines(PLN)$
0: **/* Select the Optimal Pipeline** */
0: $best\_pipeline \leftarrow L.select\_best\_pipeline(pipelines)$
0: **/* Generate Initial Code from the Selected Pipeline** */
0: $initial\_code \leftarrow CodeGenerator.generate\_code(best\_pipeline, PLN, Libs)$
0: **/* Debug the Initial Code** */
0: $debugged\_code \leftarrow Debugger.debug\_code(initial\_code)$
0: **/* Return the Debugged Code** */
0: **return** $debugged\_code$ =0

Algorithm 2: Automated Code Generation and Debugging

1.Pipeline Generation

This module generates a structured pipeline for code implementation based on the high-level plan produced by the Plan Generation phase. It breaks down the implementation task into logical steps, creating a road map for the code generation process ,as shown in Figure 4.

The process commences with a high-level conceptualization, comprising a *methodology* and a *refined methodology*, as elucidated previously. The refined methodology provides the logical underpinnings for the generated methodology by addressing specific queries. This high-level idea is supplied as input to a Large Language Model (LLM), specifically the Gemini 1.0 Pro variant in this instance. The LLM then generates a set of five distinct pipelines. The prompts employed for the pipeline generation are furnished in the appended section. Subsequently, the LLM is tasked with selecting the most optimal pipeline that exhibits the highest degree of consistency and coherence with the provided high-level idea. This comprehensive procedure is denominated as Self-Consistency in LLMs.

2.Baseline Code Generation

Code Generator: Using the generated pipeline and the high-level plan, this module produces initial code implementations. The prompts for the Code Generator are appended at the conclusion. These prompts are composed with due consideration given to mitigating hallucinations, circumventing excessive brevity, and adhering stringently to the generated pipeline. This baseline code serves as a starting point, capturing the core logic and structure of the proposed methodology, which may not be entirely free from errors or may exhibit inconsistencies with the high-level idea. This phenomenon is attributable to the tendency of LLMs to deviate at times, as elucidated in (...).The baseline code generator pipeline is shown in 5.

Error Debugger: The generated code may have certain inconsistencies, owing to the inherent behavior of LLMs. Hence, to mitigate such occurrences, we have created an error-debugging module. This debugger essentially comprises try-catch-exception blocks. The error, in tandem with the code and pipeline, is iteratively furnished as input to the LLM until an error-free code is generated. Furthermore, we have kept an upper limit to preclude an infinite loop of error correction. The prompts for the error-debugger are also appended at the end in the Appendix.

3.Final Code Generation

This module utilizes the baseline code generated in the preceding step. This baseline code serves as a source of inspiration to effectuate changes in accordance with the requisite model. This encapsulates the core principle of Continual Learning, wherein the final code is built upon the baseline code.

For generating the final code, a textual description, which is the new high-level concept, is employed. This new high-level concept not only takes into account the context of the baseline paper and dataset but also incorporates the model that is to be utilized. The procedure for generating this new high-level concept is delineated in (...).The Final code generation pipeline is shown in 6.

Code Generator: The LLM is furnished with the baseline code generated, along with the new high-level idea (comprising the context of the model to be used). Once again, the prompts for the Code Generator are provided in the Appendix.

Error Debugger: This component is same the Baseline Debugger, but it also incorporates a limited aspect of Human feedback for certain intricate ideas.

4 Experiments

We conducted a comprehensive evaluation of AutoPAC's performance using two distinct datasets and four diverse research papers which we found "interesting" and speculated on their relevance to the experiment. This section details our experimental setup, evaluation metrics, and results. The Final model for which the code is to be generated for all the experiments is the TabPFN transformer architecture.This choice allows us to evaluate AutoPAC's ability to generate code for a specific, advanced machine learning model.

The chosen papers were:

- **P1**: *Nabar, Omkar, and Gautam Shroff. "Conservative Predictions on Noisy Financial Data.".*
- **P2**: *McElfresh, Duncan, et al. "When do neural nets outperform boosted trees on tabular data?.*
- **P3**: *Chen, S. A., et al. "Tsmixer: An all-MLP architecture for time series forecasting.".*
- **P4**: *Letteri, Ivan. "VolTS: A Volatility-based Trading System to forecast Stock Markets Trend using Statistics and Machine Learning." .*

We assessed the relevance of each paper based on the following criteria:

- **Applicability to tabular data**: Papers focusing on techniques directly applicable to tabular datasets were considered highly relevant.
- **Compatibility with time series analysis**: Given our focus on financial datasets (Numerai and TSD), papers addressing time series aspects were deemed more relevant.
- **Novelty of approach**: Papers introducing novel methodologies or improvements to existing techniques for tabular data analysis were prioritized.
- **Potential for integration**: We favored papers whose methods could be feasibly integrated into our existing pipeline and combined with TabPFN.

Based on these criteria, we manually evaluated each paper and marked it as either relevant relevant(✓) or not relevant (✗) for our specific experimental setup.

- The Model (M) used for the Continual experiment is TABPFN[**?**].We selected TabPFN (Tabular Prior-Free Network) as our final model for these experiments due to its unique characteristics and suitability for our task:
  - **Versatility**: TabPFN excels in handling tabular data without requiring extensive hyperparameter tuning, making it ideal for our diverse datasets.
  - **Sample efficiency**: It performs well with limited training data, which is crucial when integrating new ideas that may not have extensive datasets available.
  - **Adaptability**: TabPFN's architecture allows for easy incorporation of new features and methodologies, aligning with our goal of continual learning and integration of novel research ideas.
- The datasets chosen are **Numerai** and another tabular dataset similar to Numerai, referred to as **Template Stock Data (TSD)**.

The Numerai and TSD datasets have been chosen to evaluate how the architectures handle differences in documentation. The TSD dataset has minimal documentation and code examples, whereas Numerai is a popular dataset with extensive documentation.

4.1 Experimental settings

The following are the details of the experiments:
Plan Generator:

- Claude 3 Sonnet was used at all stages.

Code Generator:

- x was used at y.
- x was used at y.
- x was used at y.

The Paper,Dataset and Model configuration is given to the Plan Generator as described in section x. The resulting plan is implemented using the code generator, as described in section y. This process is repeated 10 times and we measure how many times the Final Code runs correctly and verify its logic manually, reported in the table.

Code Generator:

- The LLM used for Pipeline Generator: **`gpt-4-turbo or gemini-1.5-pro-latest`**
- The LLM used for Code Generator and Debugger: **`gpt-4-turbo`**

The outputs were manually classified in the following 4 categories:

- **Running & Correct**: The code executes without errors and the desired logic is fully implemented.
- **Running & Incorrect**: The code executes without errors, but the desired logic is either incomplete or missing.
- **Not Running & Small Errors**: The logic is correctly implemented, but the code has minor issues (e.g., incorrect dataset version, incorrect variable initialization) preventing it from running.
- **Not Running & Incorrect**: The code neither runs without errors nor implements the desired logic correctly.
- **Relevant**: Additionally, we manually assessed the relevance of each paper to the given dataset.

Evaluation: Regarding the evaluation, the numbers in the cells of Tables represent the proportion of codes that were (Running Correct, Running Incorrect, Not Running with Small Errors, Not Running Incorrect) out of the total y codes generated overall.

5 Results

5.1 Dataset:Numerai

Research Paper	Relevant	Running & Correct	Running & Incorrect	Not Running Small Errors	Not Running & Incorrect
P1	✓	8/10	1/10	0/10	1/10
P2	✓	6/10	1/10	2/10	1/10
P3	✗	2/10	2/10	0/10	6/10
P4	✗	0/10	1/10	0/10	9/10

Table 2. Performance of the Baseline Code Generator on different papers.

Table 2 shows the performance of the Baseline Code Generator on four different papers (P1-P4). The table includes columns for relevance, running correctness, running incorrectness, and non-running errors. Key findings include:

- 'P1' and 'P2' are marked as relevant, while 'P3' and 'P4' are not.
- 'P1' shows the best performance, with **8/10** running correctly.
- 'P4' has the poorest performance, with **9/10** not running and incorrect.

Research Paper	Running & Correct	Running & Incorrect	Not Running Small Errors	Not Running & Incorrect
P1 + M	7/10	1/10	1/10	1/10
P2 + M	5/10	1/10	3/10	1/10
P3 + M	2/10	1/10	0/10	7/10
P4 + M	0/10	1/10	0/10	9/10

Table 3. This Table shows the Performance of the Final Code Generator on different papers along with the models.

- 'P3' shows mixed results, with equal parts running correctly and incorrectly (**2/10** each).

Table 3 presents the performance of the Final Code Generator on the same papers, with the addition of a model (denoted as "+ M"). Notable results include:

- Performance on 'P1' and 'P2' slightly decreased compared to the baseline (**7/10** and **5/10** running correctly, respectively).
- 'P3' and 'P4' show similar patterns to the baseline, with P4 still performing poorly.
- The "Not Running Small Errors" category shows some variation, particularly for 'P2+M' (**3/10**).

5.2 Dataset:TSD

Research Paper	Relevant	Running & Correct	Running & Incorrect	Not Running Small Errors	Not Running & Incorrect
P1	✓	6/10	2/10	1/10	1/10
P2	✓	4/10	0/10	5/10	1/10
P3	✗	5/10	0/10	4/10	1/10
P4	✓	5/10	1/10	3/10	1/10

Table 4. This Table shows the Performance of the Baseline Code Generator on different papers utilizing TSD.

Research Paper	Running & Correct	Running & Incorrect	Not Running Small Errors	Not Running & Incorrect
P1 + M	4/10	1/10	2/10	3/10
P2 + M	4/10	1/10	3/10	2/10
P3 + M	2/10	3/10	4/10	1/10
P4 + M	4/10	1/10	3/10	2/10

Table 5. This Table shows the Performance of the Final Code Generator on different papers along with the models and utilizing TSD.

5.3 Key Findings

1. **Performance on relevant papers**: AutoPAC demonstrated high success rates (up to 80%) in generating running and correct code for papers directly relevant to the given dataset and model.

2. **Dataset impact**: The system consistently performed better with the Numerai dataset compared to TSD, highlighting the importance of comprehensive documentation in facilitating automated code generation
3. **Relevance vs. Performance**: There was a clear correlation between the relevance of a paper and the quality of generated code, with relevant papers (P1 and P2) consistently outperforming less relevant ones (P3 and P4).
4. **Baseline vs. Final Code**: The final code generation phase, which incorporates model-specific requirements, generally maintained or slightly improved upon the performance of the baseline code generation.
5. **Error types**: For less relevant papers or the less documented dataset (TSD), the system was more prone to generating code that either didn't run or had incorrect implementations.

6 Discussion

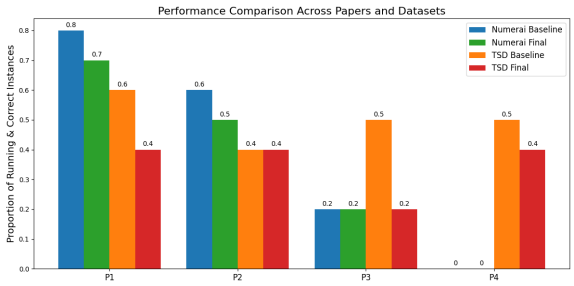

Fig. 7. Overview of the Final Code Generator

The AutoPAC presents a novel approach to automated code generation from research papers using a large language model (LLM) and a baseline code generator. The results demonstrate the potential of this approach in translating high-level ideas and natural language descriptions into executable code. While the performance varies across different research papers, the generator showcases its ability to generate running and correct code in several instances.It also demonstrates the accuracy and precision of the Plan Generator, which serves as the basis for generating the code.

Table 2 depicts the baseline code generation done using Numerai as dataset.The Baseline Code Generator performed reasonably well in generating running and correct code for some papers (P1 and P2), with 8/10 and 6/10 instances respectively. However, it also generated a significant number of instances where the code was either running but incorrect (P1: 1/10, P2: 1/10, P3: 2/10, P4: 1/10) or not running with small errors (P2: 2/10, P3: 0/10, P4: 0/10).This occurs either because the dataset lacks relevance to the paper or requires additional input that is not provided within the paper.

As evident from the experiments and figure 7, the AutoPAC pipeline demonstrates superior performance with the Numerai dataset compared to the TSD dataset, attributed to the dataset's comprehensive documentation and descriptions.

The entire process can

7 Conclusion

We have built a pipeline using LLMs to continually incorporate new papers into an existing pipeline, by generating code. Our work performs well when combining the relevant papers, with papers that are not relevant our work integrates the paper in a logical manner, albeit with small errors. AutoPAC can also reliably handle various datasets. The idea generation allows the creation of strong plans that

accounts for finer nuances in the problem, and this plan is reliably converted to code with the code generator. We hope that our work helps the research community to build working implementations of new ideas, at a rapid pace, making the barriers to implementing new ideas lower than ever before.

References

1. Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., Jiao, W.: Self-planning code generation with large language models (2024)

A High Level Ideas Generated

B Prompts used for Code Generation

Prompt for Baseline Code Generation

```
<role>:"system":
<content>:"You are a Machine Learning engineer.
Your task is to generate comprehensive and detailed
Python code for a complete end-to-end
Machine Learning pipeline. Ensure that no steps
are omitted due to brevity and that all aspects
of the implementation are thoroughly addressed.
Include all necessary data preprocessing, model
training, evaluation, and deployment steps.
Provide detailed explanations. Ensure the code is
robust, efficient,and follows best practices in
software development and machine learning.
Avoid using placeholders or simulating parts of
the code; provide fully functional implementations".
</content>

<role>:"user"
<content>:"idea:{idea} pipeline:{Pipeline}
miscellaneous:{miscellaneous}.Assume the data
is already loaded in train variable. Follow the
instructions clearly. Do all the complicated
implementations. Follow the \textit{Instruction};
\textit{Pipeline} strictly,
Ensure all steps are implemented comprehensively
and efficiently",
```

Prompt for Final Code Generation

```
<role>"user"</role>
<content>"Your task is to modify the existing
Python code provided in \n\n{Code}\n\naccording
to {Model_Idea}, ensuring alignment with
{model_refined_methodology}. Maintain the current
code structure while adapting it to meet the
requirements of {Model_Idea}. Avoid altering the
fundamental code layout. Aim for full
implementation without assuming any details.
"</content>

<role>"user"</role>
<content>"Ensure no assumptions are made; employ
various ML techniques creatively to align with
{Model_Idea}. Avoid brevity, placeholders, or
assumptions.Provide a comprehensive implementation."
```

Prompt for Pipeline Generation

```
<content>
You will be provided with an idea and the necessary
libraries. Your task is to outline a detailed
pipeline without giving actual code.

Review the provided context and idea carefully.
Do not include the code; focus on describing the
pipeline.
Explain each step thoroughly, considering the dataset
characteristics and the proposed approach.
Utilize the provided libraries as necessary.
Aim for clarity and coherence in your response.
Approach this as a Machine Learning Researcher,
providing a step-by-step plan for the analysis.

[Idea]:{idea}
</content>
```

C Some Code Snippets from Generated Codes

P1+M

```
. . . . . .
thresholds = np.percentile(base_impurities, [75])
for level in range(1, num_levels):
    keep_indices = base_impurities < thresholds[-1]
    if len(keep_indices) != len(X_train):
        keep_indices = np.repeat(keep_indices, len(X_train)
          // len(keep_indices) + 1)[:len(X_train)]
    X_train_pruned = X_train[keep_indices]
    y_train_pruned = y_train[keep_indices]
    model, accuracy, impurities, probabilities, predictions = train_and_evaluate(
    X_train_pruned, X_val, y_train_pruned, y_val)
    print(f"Level {level} Model Accuracy: {accuracy}")
    thresholds = np.concatenate((thresholds,
    np.percentile(impurities, [75])))
    models.append(model)
    accuracies.append(accuracy)

return models, accuracies
    . . . . . .
model = TabPFNClassifier(device='cpu',
N_ensemble_configurations=32)
model.fit(X_train, y_train)

# Predict probabilities and classes
val_predictions, probabilities = model.predict(X_val,return_winning_probability=True)

# Gini impurity of predictions
def compute_gini_impurity(probabilities):
    return 1 - np.sum(np.square(probabilities), axis=1)

impurities = compute_gini_impurity(probabilities)
accuracy = accuracy_score(y_val, val_predictions)

return model, accuracy, impurities, probabilities, val_predictions

    . . . . . .
    . . . . . .
```

P2+M

```
. . . . . .
. . . . . .

def create_tabpfn_datasets(X, y, n_datasets=10, n_samples=1000, n_features=100):
datasets = []
for _ in range(n_datasets):
sample_idx = np.random.choice(X.shape[0], n_samples, replace=False)
      feature_idx = np.random.choice(X.shape[1], n_features,replace=False)
      X_sample = X.iloc[sample_idx, feature_idx]
      y_sample = y.iloc[sample_idx]
      datasets.append((X_sample, y_sample))
return datasets
. . . . . .

predictions = []

for model in best_models:
    if isinstance(model, TabPFNClassifier):
        tabpfn_datasets = create_tabpfn_datasets(test_preprocessed,

        np.zeros(len(test_preprocessed)))
        model_preds = []
        for X_tabpfn, _ in tabpfn_datasets:
            model_preds.append(model.predict_proba(X_tabpfn)[:, 1])
        predictions.append(np.mean(model_preds, axis=0))
    else:
        predictions.append(model.predict_proba(test_preprocessed)[:, 1])

ensemble_predictions = np.mean(predictions, axis=0)
. . . . . .
. . . . . .
```