Paper CT01

Generative AI: A Clinical Programmer's Friend!

Karl Kennedy, GSK, London, UK
Daniel Ghali, ex-GSK IP Student, London, UK

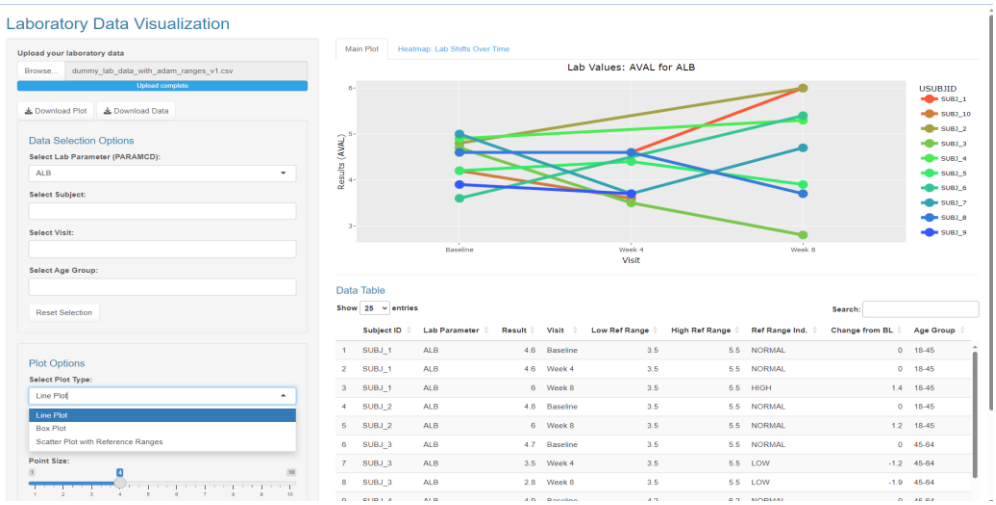
ABSTRACT

This paper discusses the benefits of incorporating Generative AI into a clinical programmer's daily routine. It demonstrates these benefits through an R-based Shiny app that showcases various coding features that programmers can utilize by using Generative AI. The app illustrates how to use Generative AI natural language prompts to comment, test, optimize, debug, explain, and refactor R code. The use of these features can lead to increased productivity, efficiency, and quality in programming tasks while also reducing the learning curve for programmers new to R.

INTRODUCTION

The growing complexity of clinical programming, especially in regulated industries like pharmaceuticals, presents significant challenges for programmers. From managing large datasets to ensuring compliance with regulatory standards, clinical programmers must navigate a variety of complex tasks. Generative AI, particularly tools like ChatGPT, GSK's Jules, are emerging as a valuable aid in this domain. By automating code generation, debugging, and optimization, AI can significantly reduce development time and minimize errors. This paper explores the practical application of generative AI in clinical programming, focusing on the development of a Shiny app for laboratory data analysis. Through real-world examples, it demonstrates how AI can streamline workflows, enhance code quality, and support faster decision-making.

This is a screenshot of the final Shiny app using an iterative development approach. It was developed using ChatGPT 4o. The dummy data displayed in the app was generated by a prompt that described the structure of the data required. This dummy data was used throughout the Shiny App development process.



Shiny App Development Process

Initial Setup

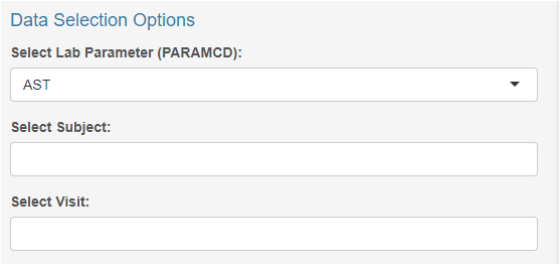
Developing a Shiny app to analyse laboratory data is a multi-step process, and Generative AI played a crucial role at each stage. Initially, the task involved setting up an interface for uploading laboratory data in different formats (CSV, Excel, SAS). Generative AI was instrumental in guiding the choice of functions like `fileInput()` and explaining the differences between file formats, ensuring the app would correctly handle multiple file types.

Generative AI helped generate dummy data based on CDISC ADaM standards, providing sample data in the required structure. The tool generated clear and concise R code to create dummy laboratory data using variables such as USUBJID, PARAMCD, AVAL, AVISIT, and reference ranges (ANRLO, ANRHI).

Interactive Features

As the app developed, Generative AI successfully incorporated the app's requirements using natural language prompts.

For instance, it added interactive features like dynamic filtering by subject and visit. This led to the integration of dropdowns and reactive UI components. For example, in one version of the app, Generative AI helped identify the best ways to bind the filtered table selections with the graphical output, ensuring the plots updated based on user selections.



Example Interactions

An instance of Generative AI's problem-solving capabilities was demonstrated during the development of a table viewer. Initially, the table viewer was limited to displaying only the first 10 rows. To enhance user experience, I issued the following prompt:

Prompt Example: `"In the UI.r code, the table viewer is set to only display the first 10 rows. I would like to have the ability to view all the data using a scrollable bar. Could you regenerate the code to incorporate a scrollable bar for data viewing?"`

In response, Generative AI suggested a scrollable data table using `DT::renderDataTable()` and provided code that facilitated a more adaptable, scrollable view of the data, including a search function.

Optimizing R Code with Generative AI

Code Suggestions and Refactoring

Generative AI significantly improved the quality of the Shiny app by providing optimized versions of the initial code. For instance, when working on plotting the data, Generative AI helped optimize plot rendering by suggesting reactive expressions to handle large datasets more efficiently. It also recommended the use of `req()` to ensure the UI didn't attempt to render plots or tables before data was available.

Another optimization example was in managing the download functionality. Generative AI suggested creating dynamic download handlers, ensuring that both filtered datasets and graphical outputs could be downloaded in formats such as CSV and PNG, respectively.

```
output$download_plot <- downloadHandler(
  filename = function() {
    paste("lab_plot", Sys.Date(), ".png", sep = "")
  },
  content = function(file) {
    ggsave(file, plot = last_plot(), device = "png")
  }
)
```

Code Explanation: `downloadHandler`: This function creates a downloadable file from the Shiny app. It consists of two key components:

- filename: A function that specifies the file name. In this case, it uses `paste()` to create a name like "lab_plot_2023-09-15.png", combining "lab_plot", the current date (Sys.Date()), and the .png extension.
- content: This function defines the content of the file. It uses `ggsave()` to save the most recently generated `plot (last_plot())` as a PNG file. The file argument refers to the file location where the PNG will be saved.

This simple code suggestion from Generative AI streamlined the process of enabling users to download their customized plots from the app interface.

Generative AI also contributed to improving the plotting performance and customization of the Shiny app, ensuring smooth updates and responsiveness when user selections were applied.

Prompt Example: `"The graph didn't update when I changed the data selection in the table. Please review and correct the code."`

Result: Generative AI analyzed the issue and suggested optimizing the reactive expressions to ensure that the data and plot were properly linked. By improving the reactivity of the plot generation process, the AI helped optimize the app's performance, especially when users selected specific data rows from the table.

```
# Reactive expression to filter data based on user selections
filtered_data <- reactive({
  req(input$file_upload)  # Ensure file is uploaded
  data <- read.csv(input$file_upload$datapath, stringsAsFactors = FALSE)

  # Apply subject and visit filters from the data table
  data <- data[data$USUBJID %in% input$subject_filter & data$AVISIT %in%
input$visit_filter, ]
  data
})

# Plot updated based on the filtered data
output$lab_plot <- renderPlotly({
  req(filtered_data())  # Ensure filtered data is available
  data <- filtered_data()

  p <- ggplot(data, aes(x = AVISIT, y = AVAL, color = USUBJID)) +
    geom_line(size = 1.5) + geom_point(size = 3)

  ggplotly(p)
})
```

Code Explanation: This block uses a reactive expression in Shiny to dynamically filter data based on user inputs. The req() function ensures that the dataset is available before proceeding. Data is filtered by Subject ID (USUBJID) and visit (AVISIT) to ensure the user sees only the relevant rows in both the table and plot. This enhances performance and improves the user experience by updating the data in real-time.

By ensuring that the reactive data filtering and plot rendering were properly linked, the AI helped optimize the Shiny app's performance, enabling real-time updates to the plot when selections in the data table were modified. This improved both the app's efficiency and the user experience.

Debugging and Error Resolution with Generative AI

Handling Common Errors

Throughout the development process, various errors and roadblocks were encountered, particularly when handling different types of data. One example was the error encountered when plotting a heatmap of the data: `"Warning: Error in unit: 'x' and 'units' must have length > 0."` Generative AI played a crucial role in diagnosing this issue by suggesting that the problem might lie in how the data was being passed to `ggplot2` and offering solutions to filter or correct missing values.

Another key contribution was error handling in the dynamic UI elements, where Generative AI helped implement checks to ensure that the app did not break when no data was available.

Case Study: Heatmap Error

The heatmap functionality presented significant challenges, with the app throwing errors related to missing values and empty datasets. Generative AI provided a solution by suggesting that the axis values (x, y) be converted into factors and that the dataset be filtered to remove any missing or invalid data.

```
# Ensure AVISIT and PARAMCD are factors
data$AVISIT <- as.factor(data$AVISIT)
data$PARAMCD <- as.factor(data$PARAMCD)
```

This simple fix resolved a critical issue that had been stalling the app's development.

Enhancing the User Interface

UI Improvements

In addition to back-end code optimization, Generative AI also assisted in enhancing the visual elements of the app. Initially, the app's layout was basic, with minimal styling. Generative AI suggested leveraging the *'shinythemes'* package to give the app a more polished look. By using a theme like "cerulean," the overall user experience was enhanced.

```
fluidPage(
  theme = shinytheme("cerulean"),
  ...
)
```

Moreover, Generative AI suggested ways to improve the table layout, making it scrollable and more user-friendly. This was crucial when dealing with large datasets, as the table could otherwise become difficult to navigate.

Customizing Plots

Plot customization was another area where Generative AI provided invaluable support. From adjusting the thickness of lines in the line plot to dynamically generating colour palettes based on the number of subjects, Generative AI offered tailored solutions that improved the readability and aesthetics of the plots.

For instance, to ensure that subjects were easily distinguishable in the line plot, Generative AI suggested generating enough colours dynamically based on the dataset:

```
color_palette <- colorRampPalette(c("#FF5733", "#33FF57",
"#3357FF"))(length(unique(data$USUBJID)))
```

Natural Language Prompts in Shiny App Development

One of the most powerful aspects of using generative AI in the context of Shiny app development is its ability to interpret natural language prompts and generate precise code based on user input. Throughout the development process, a variety of prompts were used to create and enhance the app's functionality, optimize performance, and resolve technical challenges.

During the development process, no prompt engineering techniques were employed. However, it's clear that the more precise and detailed your prompt, the greater the likelihood of success. This precision also reduces the need for additional follow-up prompts to achieve your desired result.

1. Initial Setup and Data Uploads

The first step was to allow users to upload laboratory data in Excel, CSV, or SAS formats. Generative AI helped set up the file upload interface and logic for handling multiple formats.

Prompt Example: *"The first requirement of the app is to be able to upload laboratory data in Excel, CSV, or in SAS format."*

Result: The AI generated the `fileInput()` widget and server-side logic for handling various file formats, ensuring the app could accept data in the required formats.

2. Data Filtering and Visualization

Generative AI provided solutions to add subject and visit filters to ensure users could dynamically select data and view updated visualizations.

Prompt Example: *"I would like to link the table data selection with the graph, for example only display the selected subject data from the table in the graph."*

Result: The AI generated code to create filters using `selectInput()` and linked them to the `ggplot2` visualizations, allowing real-time updates to plots based on user selections.

3. Plot Customization and Debugging

Several prompts were used to address issues with the plot rendering and ensure proper customization of plots like line plots, box plots, and scatter plots.

Prompt Example: *"The graph didn't update when I changed the data selection in the table. Please review and correct code."*

Result: Generative AI identified issues in the reactive expressions and provided suggestions to correct the data linking between the table and the plot, ensuring dynamic updates.

4. Error Handling

Errors in plot rendering were a common challenge. The following prompt helped address an issue with missing values when rendering a heatmap.

Prompt Example: *"I get this error message when I tried to load the alluvial plot - 'Warning: Error in geom_alluvium: Problem while computing aesthetics...'"*

Result: The AI model suggested converting variables to factors and filtering out missing values to resolve the error, allowing proper heatmap rendering.

5. Download Functionality

Generative AI also assisted in setting up download functionality for both the plots and the filtered datasets.

Prompt Example: *"Could you add functionality to allow users to download the plots as images or export the underlying data?"*

Result: The AI provided code using `downloadHandler()` to enable users to download both the current plot as a PNG file and the filtered data as CSV.

6. Code Optimization: Using a Configuration File for Customization

One of the key examples of code optimization during the Shiny app development process involved making the app more customizable. This was achieved by using a configuration file to manage style attributes, reducing the need for hard-coded values in the app. The process was initiated with the following prompt:

Prompt Example: *"I would like to make the app customisable by using a config file for style attributes. Would you be able to implement this requirement?"*

The generative AI model responded by suggesting the use of a JSON configuration file to store and manage the app's style attributes. This allowed for a cleaner, more modular code structure, where changes to the appearance **of** the app could be made by simply editing the configuration file rather than modifying the core code**.**

7. Enhancing Code Maintenance: Using Generative AI for Code Commenting and Structuring

Effective code maintenance is crucial for ensuring that applications remain scalable, understandable, and easy to update. Generative AI can assist developers by providing professional code comments and structured headers, which improve readability and make it easier for future developers to work with the code. This was demonstrated using two key prompts:

Prompt Example: *"Could you provide professional code commenting to the code, please?"*
The AI responded by analyzing the existing code and adding detailed comments, explaining the functionality of each section and the purpose of specific lines. The AI-generated comments were clear, concise, and aligned with best practices, ensuring that any future developer could understand the code's logic and functionality without needing extensive explanations from the original programmer.

Prompt Example: *"Please add a headerbox to the code too."*
The AI then added a header box to the top of the script, providing a high-level overview of the app's purpose, its key features, and the date of the last modification. This header box serves as documentation for developers, ensuring that the purpose of the code is immediately clear and that the code is well-organized.

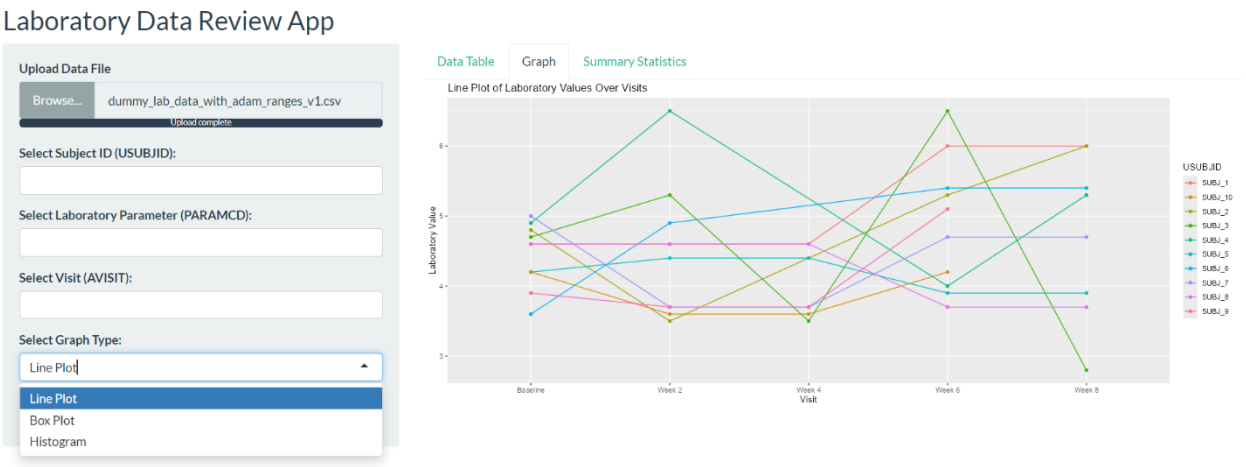
The Usage of ChatGPT o1- preview

To demonstrate the improving capability of generative AI models, I tested ChatGPT o1-preview by assigning it the task of producing the Shiny app with a single prompt. This prompt aligns with the key Shiny app functionality that had been generated using earlier versions of the model:

"Act as a Shiny and R programmer, please create a Shiny app that has the following functionality. This app will help the end user to review laboratory data. The laboratory data will follow CDISC ADaM standards.

1. *The ability to upload CSV, Excel, and SAS datasets.*
2. *The functionality to filter the data by Subject ID (USUBJID), Laboratory parameter (PARAMCD), and visit (AVISIT). When the user uses these filters, the data table and graphs should update dynamically.*
3. *The data should be displayed in tabular and graphical forms.*
4. *I would like an option where the user can select between different types of graphs. Please select the best 3 graphs for visualizing laboratory data.*
5. *Please set up a configuration file that can be used to change the appearance of the app.*
6. *Please include any other features that you think may improve the user experience."*

The model generated the app without any errors and incorporated the key functionality as requested. Below is an image of the Shiny app produced by ChatGPT o1-preview.



For point 6, it included several enhancements that were not explicitly asked for:

- Data Download: It enabled users to download the filtered dataset as a CSV file, functionality that existed in the original app.
- User Experience Enhancements:
  - Responsive UI with dynamic updates.
  - Error handling for invalid file types.
  - Intuitive layout with tabs for the data table, graphs, and summary statistics.

Additionally, the model recommended the following extra features to further improve the app:

- Additional Graphs: Implement scatter plots or trend analyses for better data visualization.
- Statistical Tests: Provide options to perform statistical analyses on the data.
- User Authentication: Secure the app for handling sensitive data.
- Interactive Plots: Suggested using *Plotly* for interactive graph capabilities (functionality already implemented in the original app).

This result demonstrates the model's advanced capabilities, managing to generate a complete and functional Shiny app based on a single prompt while also suggesting valuable enhancements that improve both the user experience and app functionality.

How Generative AI Improved the Paper Writing Process

Generative AI played a pivotal role not only in drafting this paper but also in refining it to meet high standards of quality and clarity. Beyond simply writing sections, the AI contributed by suggesting improvements, structuring content, and facilitating technical accuracy. Here's how generative AI made the writing process smoother and more efficient:

1. Accelerating the Initial Draft

To initiate the process of writing the paper, I uploaded my abstract and the PHUSE paper template. This assisted the

Generative AI model in crafting the initial draft by offering suggestions for each section in line with the paper's specifications. By feeding in key topics and themes, the AI swiftly generated content that laid the groundwork for the paper. This efficiency freed up more time to concentrate on honing the arguments and incorporating technical details, as opposed to spending hours constructing each section from the ground up.

2. Enhancing Structure and Organization

A key challenge in writing a technical paper is maintaining a clear and logical structure. Generative AI provided an outline early in the process, ensuring that the content flowed seamlessly. It grouped related concepts together and suggested section headings that made the content easier to follow. This step ensured that the paper was coherent and readable, guiding the audience through the discussion of Shiny app development and debugging workflows.

3. Providing Tailored Suggestions for Improvement

Generative AI offered several suggestions to improve clarity and depth. For instance, when generating technical explanations for debugging workflows or describing Shiny app functions, the AI flagged certain areas for more detailed explanation. It also suggested transitions between sections and offered ways to make complex technical ideas more accessible, ensuring that the content was both detailed and understandable to a technical audience.

4. Supporting Technical Precision

Throughout the drafting process, the AI ensured technical accuracy by providing correct syntax and detailed explanations for specific R code examples. When the need arose for highly technical content—such as discussions on reactive programming or dynamic UI elements—AI helped validate the content by ensuring that the syntax was correct, and the examples aligned with the best practices in clinical programming. Moreover, when technical concepts needed clarification, generative AI offered concise, jargon-free explanations that helped maintain the balance between accessibility and technical rigor.

5. Improving Readability and Flow

Generative AI also acted as an editor, suggesting improvements to sentence structure, transitions, and the overall readability of the paper. It identified instances of overly complex language and simplified the writing, making it easier for readers to grasp the key ideas. The AI maintained the technical depth needed for an expert audience while improving the flow between concepts, which was essential in a paper focused on technical workflows and debugging strategies.

6. Supporting Visuals and Diagrams

In addition to text generation, AI provided guidance on which visual elements would complement the technical discussion. By suggesting the inclusion of system architecture diagrams, workflow illustrations, and sample screenshots of the Shiny app, the AI enhanced the paper’s overall presentation. The suggested visuals helped to clarify complex processes, making the paper more engaging and comprehensive for the reader.

7. Refining the Paper with Feedback

As the paper evolved, generative AI continuously provided feedback, recommending improvements based on structure, depth of explanation, and flow. Each time the paper was reviewed, the AI highlighted areas that could be expanded or restructured for better comprehension. This feedback loop allowed for a cycle of iteration, ensuring that the paper improved with each pass, ultimately leading to a more polished and technically sound final product.

Discussion

Beyond the immediate technical improvements, integrating Generative AI into the development process offers broader implications for the field of clinical programming:

Training and Onboarding

New programmers can benefit significantly from Generative AI's explanations and guidance, shortening the learning curve and reducing the time required to become proficient in R and Shiny. Generative AI can serve as an always-available resource for answering questions and clarifying concepts.

Generative AI, such as Generative AI, has proven to be a valuable tool for onboarding new programmers. It serves as a personalized tutor, capable of answering questions and explaining complex programming concepts in real time. This is especially useful in clinical programming, where new programmers often face a steep learning curve.

Daniel's Experience with Generative AI

Daniel, a co-author of this paper, started his clinical programming journey with minimal programming knowledge. Through his use of generative AI tools like Generative AI, he was able to quickly advance from basic troubleshooting to more complex coding tasks. In Daniel's own words:

“What started out as simple debugging help, later became a centre for learning and

experimentation, from which my skills proliferated. By asking how I could improve my code or if it was possible to do a certain task, I rapidly expanded my knowledge and understanding, taking it question by question."

This experience highlights the key advantages of using generative AI for learning:

- **Incremental Learning**: The AI provides answers tailored to the user's current level of knowledge, allowing them to build on existing skills without being overwhelmed.
- **Immediate Feedback**: Questions are answered instantly, helping new programmers quickly resolve issues and move forward.
- **Encouraging Experimentation**: By having an always-available assistant, Daniel was encouraged to try new approaches and experiment with different methods, which accelerated his learning.

However, Daniel also noted potential risks: *"A potential risk can occur when used for tasks well beyond the user's current level. This can place the user into a situation with semi-working code but with no understanding of the problems or how to fix them."*

This reinforces the idea that while AI is a powerful learning tool, human oversight is critical for ensuring that the code generated or modified is fully understood by the user.

Daniel's Experience with Generative AI for Code Readability

Daniel, a co-author, shared his experience in improving the readability of his R code through generative AI. By removing all indentations and comments from a table QC program, he prompted the AI with the following:

```
"Make this code more readable."
```

The generative AI provided a response that restructured the code, added necessary comments, and improved the overall readability. Here's an excerpt of the code after AI suggestions:

This result demonstrates how AI can assist in making code cleaner, easier to read, and aligned with best practices. Daniel noted, however, that while AI-generated comments were useful, domain-specific knowledge was sometimes missing, particularly around industry-specific functions like the `add_label()` function. This underlines the importance of pairing AI suggestions with human oversight.

Daniel's insights highlight the balance between AI-driven code generation and the need for human review to ensure both accuracy and alignment with organizational coding standards.

Daniel's Experience with Generative AI for Code Efficiency

Daniel, a co-author of this paper, experimented with an earlier language model to improve R code efficiency. He used the model to rewrite his code, aiming for enhanced performance while maintaining the accuracy of his results. Initially, Daniel provided the following prompt:

"I will give you a list of steps to address one by one as if they are individual prompts but all connected.

1. *Give me a list of ways to improve efficiency of R code.*
2. *Apply steps 3-7 to this input code.*
3. *Keep a record of input datasets and their file types.*
4. *Keep a record of datasets created or changed as far as you can deduce.*
5. *Keep a record of variables loaded, created or changed and their types as far as you can deduce.*
6. *Keep a record of values loaded, created or changed and their types as far as you can deduce.*
7. *Given steps 3, 4 and 5 keep a record of the final output this code produces as far as you can deduce, and call it output 1.*
8. *Describe known parts of output 1 that you have deduced from the code.*
9. *Go through the list from step 1 and assess if any can be applied to the input code without creating errors downstream.*
10. *If they can be applied, rewrite the input code, implementing the changes (I know you cannot test it, but don't worry, I will test it), but do your best to keep the output the same as output 1"*

The language model responded with suggestions for optimizing the code by reducing redundancy, using efficient data structures, and improving function calls. However, while the AI suggestions improved code efficiency, some challenges arose when working with specific datasets.

For example, the AI attempted to apply the `fread()` function, which is typically used for reading CSV files, to a SAS dataset. This led to errors as `fread()` is not compatible with the SAS data format. Daniel had to intervene, correcting the AI's mistake and guiding the process by offering additional context. Despite these challenges, the result of the optimized code proved beneficial.

Daniel's experience highlights the potential and limitations of using generative AI for optimizing code. While it significantly improved efficiency, human oversight was essential to ensure accuracy, especially when dealing with specialized data formats like SAS.

Another member of the team, used an earlier version of a generative AI model, GSK's Jules, to assist with his R programming training. He initially used Jules to rewrite some of his R code for better efficiency. He describes his experience as follows:

"I asked Jules to rewrite my code so that it would be more efficient. It returned a rewrite of my code, but it contained errors. I spent time debugging it as a learning exercise. The code had about four errors, and after I fixed them, it ran cleanly. However, some issues that my original code had resolved, such as floating point and trailing zero problems, were reintroduced by Jules' solution."

This experience emphasizes the role of generative AI as a learning tool. Although AI-generated solutions may not always be perfect, the process of troubleshooting and refining AI-suggested code provided him with valuable learning opportunities in R programming. This highlights the need for human oversight when working with AI-generated code, ensuring that the programmer fully understands the solutions being implemented. On a positive note, it also taught him some key R programming concepts:

"*But Jules's code did teach me a few things in R:*

1. *creation/use of functions,*
2. *across – to perform an action on several variables within a single statement,*
3. *bind rows to call a function multiple times passing different values each time and have the results stacked together."*

The Evolution of Generative AI Models

During the development of this paper, multiple versions of generative AI models were utilized, including ChatGPT 3.5, ChatGPT 4.o, and GPT-4-turbo (code-named o-preview). Over time, the coding abilities of these models have significantly improved, with a noticeable reduction in hallucinations and the need for follow-up prompts.

For example, when using the ChatGPT 3.5 model, debugging and issue resolution required more interaction. The model often generated incomplete or error-prone code, resulting in a basic Shiny app that required extensive manual intervention. The process was iterative, with each bug or error prompting multiple rounds of corrections before a functional solution was reached.

With ChatGPT 4o, the development process became more streamlined and interactive. The model was capable of handling functionality piece by piece, allowing for more efficient code generation. Each component of the Shiny app was built sequentially, with fewer errors and a smoother overall process.

The latest model, GPT-4-turbo (0-preview), demonstrated an impressive leap in capability. It was able to generate a complete Shiny app in a single prompt, outlining all required functionalities with no errors. This evolution highlights the model's ability to handle complex prompts with a much lower error rate, showcasing how the newer versions can deliver higher-quality outputs with far less interaction.

Ethical Considerations

As generative AI becomes more integrated into clinical programming, it is crucial to address potential ethical concerns and biases. While AI-driven tools like ChatGPT improve efficiency and accuracy, they are not immune to generating biased outputs or recommendations. The training data used by these models can carry inherent biases, which may influence code suggestions or dataset handling. For example, certain assumptions in the data could lead to biased recommendations or inappropriate handling of edge cases.

Moreover, in clinical settings where patient data is involved, it's critical to ensure that AI-generated solutions comply with data privacy regulations (e.g., HIPAA, GDPR). Transparency is needed regarding how AI models handle sensitive information, and programmers must remain vigilant in reviewing and validating AI-generated outputs to avoid unintended consequences.

Another consideration is the tendency of earlier AI models to produce “hallucinations,” generating code that appears plausible but contains errors. While this has improved in newer models, such as GPT-4-turbo, it emphasizes the need for human oversight to ensure code correctness, particularly in regulated environments like healthcare and clinical trials.

To mitigate these issues, generative AI should always be used in conjunction with human judgment, particularly when handling sensitive clinical data or complex programming tasks. Programmers should validate the outputs rigorously, ensuring alignment with industry standards and ethical guidelines.

Impact on Productivity

Integrating Generative AI into the clinical programming workflow, particularly for Shiny app development, has demonstrated significant improvements in productivity. By assisting with mundane coding tasks, such as writing reactive expressions and handling errors, Generative AI allowed the programmer to focus on more complex tasks. The tool’s ability to suggest optimizations and fix errors quickly resulted in a more efficient development process.

Limitations

While Generative AI provides strong support in R and Shiny development, its capabilities are limited in highly specialized areas of clinical programming. For instance, domain-specific statistical modelling or regulatory-compliant workflows may require additional human oversight. Generative AI's recommendations are helpful but not infallible, and all outputs should be reviewed carefully for accuracy and relevance.

Future Directions

The success of Generative AI in this project opens the door for further exploration of AI tools in clinical programming. Future developments could include deeper integration of AI for real-time code review, automated regulatory compliance checks, or the creation of entirely autonomous workflows for certain programming tasks. Generative AI could also be combined with other AI-driven tools to enhance clinical trial data analysis further.

Conclusion

Generative AI has demonstrated its potential as a valuable tool for clinical programmers, offering time-saving solutions and improving overall code quality. As seen in the development of the Shiny app, AI-assisted programming can streamline processes, from data visualization to dynamic filtering, while significantly reducing the manual effort involved. The evolution of AI models from ChatGPT 3.5 to ChatGPT o1-preview reflects a rapid advancement in coding capabilities, where newer models are not only more efficient but also more capable of handling complex tasks with minimal errors. As AI continues to evolve, its role in clinical programming will likely expand, providing even more sophisticated solutions and enhancing the ability of programmers to manage the increasing demands of data-driven industries.

REFERENCES

OpenAI’s ChatGPT – https://openai.com/chatgpt
GSK’s Generative AI Model - Jules

ACKNOWLEDGMENTS

I extend my heartfelt appreciation to Salaja Sirsalewala and Scott Worrell for their insightful guidance and contributions to this paper. Their support is greatly appreciated and respected.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Karl Kennedy
Company: GSK
Address:
Email: karl.j.kennedy@gsk.com