SASBuddy: Enhancing SAS Programming with Large Language Model Integration

Karma Tarap, Derek Morgan, Pooja Ghangare

ABSTRACT

SASBuddy, dubbed "your friendly SAS programming assistant," is a pioneering SAS macro designed to facilitate SAS programming through Large Language Models (LLMs). This tool empowers users, particularly those with limited SAS coding expertise, to generate precise SAS code efficiently by interpreting natural language inputs (Wei et al., 2023). Beyond code generation, SASBuddy now integrates ***iterative error resolution, session tracking, macro utilization, retrieval-augmented generation (RAG), and an API-driven architecture*** *that separates execution from intelligence. This architecture enables advanced session management and structured debugging while maintaining context across interactions. The system implements* ***react-style code fixing*** *for targeted error resolution and incorporates* ***Search-O1*** *(Li et al., 2025) with a "Reason-in-Documents" module for efficient knowledge retrieval. Powered by the latest* ***o3-Mini*** *reasoning model, SASBuddy delivers optimized performance with faster inference times and enhanced debugging capabilities, representing a significant advance in AI-assisted clinical SAS programming.*

INTRODUCTION

In a realm where the intricacy of human language meets the computational prowess of machines, we find ourselves on the brink of significant advancements. Large Language Models (LLMs) like ChatGPT are making waves, promising a transformative impact across various sectors including the pharmaceutical industry. These models offer a glimpse into a future where tasks like code generation and data analysis could become notably streamlined. This paper delves into the world of LLMs, their journey from conception to the current state, and the prospective applications.

We introduce the SASBuddy, a SAS macro envisaged to tap into the potential of LLMs, aiming to provide a helping hand to Clinical SAS Programmers through a macro interface. Specifically, it addresses a critical nexus of challenges in clinical programming: the generation of relevant code without disclosing patient data to LLMs, the integration of standard macro usage where applicable, and the implementation of automated feedback loops for continual code refinement.

Through this discourse, we seek to offer a realistic insight into how LLMs could not only enhance clinical programming but also significantly alter our interaction with this emerging technology.

BACKGROUND

What are Large Language Models?

Language models, a subset of machine learning models, excel at understanding and generating human language. They're trained on vast amounts of text, leveraging statistical methods to predict subsequent words in a sequence based on the preceding ones. The journey from simple n-gram models and probabilistic context-free grammars to today's sophisticated architectures underscores the remarkable strides made in this field. The landmark paper "Attention Is All You Need" (Vaswani et al., 2017), marked a pivotal point by introducing the Transformer architecture, a significant shift from traditional RNNs and LSTMs. This novel approach to handling sequences through self-attention mechanisms spurred the development of advanced language models like BERT, GPT-2, and eventually GPT-3. The innovations brought about by the Transformer architecture, such as eliminating recurrent layers, enabled parallel processing and effective management of long-term dependencies, paving the way for the creation of models with billions of parameters – the cornerstone of contemporary LLMs.

Rise of the Parameters

The advent of the Transformer architecture ignited the race for developing increasingly expansive language models. Parameter count became a crucial metric, with new models regularly showcasing billions of parameters (*Fig.1*). An augmented parameter count not only enhances a model's contextual understanding but also its capability to generate more nuanced and accurate responses (Chernyavskiy et al., 2021). However, this surge in parameters requires significant computational resources and extends training durations, presenting challenges in model training and deployment (A*nthaswamy, 2023*).

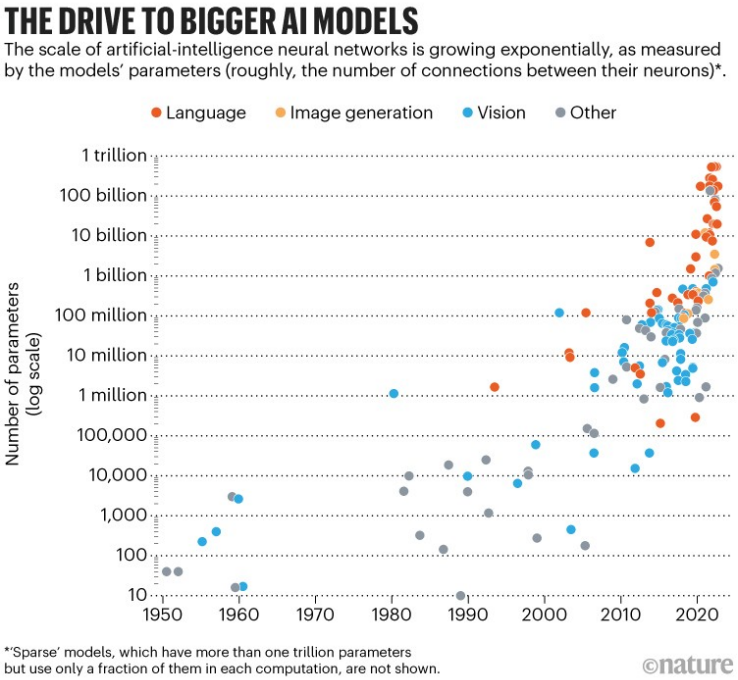

Figure 1. The Drive to Bigger AI Models (Source: Ananthaswamy, 2023).

Public's Imagination: The ChatGPT Phenomenon

The graph below showcases the swift user adoption across various platforms; notably, ChatGPT amassed 100 million users within two months post-release (*Fig.2*). But what fueled this swift adoption?

Among LLMs, ChatGPT uniquely resonated with the public owing to its conversational proficiency, diverse knowledge base, and human-like reasoning. Although built on GPT-3's architecture, ChatGPT underwent a distinct fine-tuning process incorporating conversational data and user feedback. This resulted in a versatile model capable of managing a myriad of tasks and dialogues, thereby appealing to a broader user base.

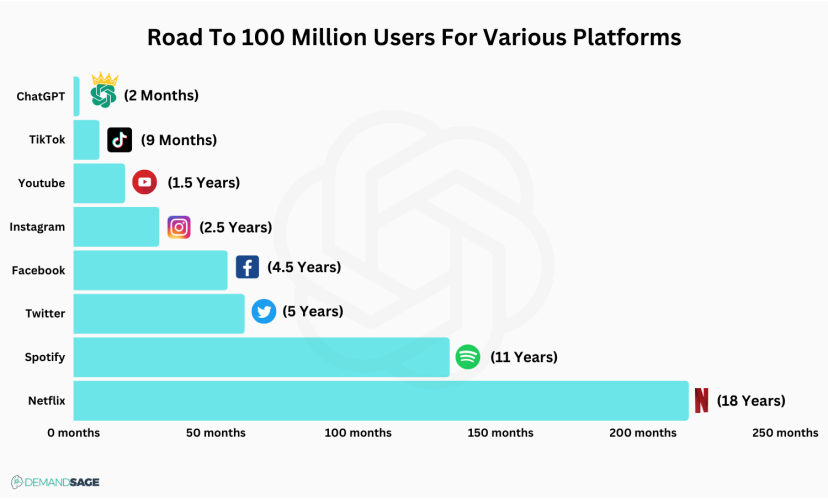

Figure 2. Road to 100 Million Users for Various Platforms (Source: Demand Sage, n.d.)

CHALLENGES OF LLM'S IN CLINICAL SAS PROGRAMMING

The integration of LLMs in clinical SAS programming presents significant challenges, particularly in error resolution, context retention, and augmenting knowledge retrieval (Brown et al., 2020). Despite the potential benefits of LLMs, a significant hurdle remains in the proprietary nature of SAS, which results in less representation within LLM training datasets compared to open-source languages such as Python or Java (Zhao et al., 2023). This discrepancy impacts the LLMs' capability to directly generate SAS code that is both relevant and precise. Moreover, the sensitive and voluminous nature of clinical data limits its availability for both LLM training and context providing.

SASBuddy addresses these challenges through several integrated approaches:

Data-Relevant Code Generation

SASBuddy's ability to generate data-relevant SAS code is intricately linked to its initial step of incorporating the metadata of datasets into the query process. This approach ensures that the generated code is not only syntactically correct but also contextually aligned with the specific data structures it is meant to analyze or manipulate (Wei et al., 2023). By passing this rich metadata into the query, SASBuddy arms the LLM with the necessary context to understand where and how to look for specific data points within the datasets.

Session Management and Context Retention

Building on recent advances in language agents (Shinn et al., 2023), SASBuddy implements session-aware processing that enables users to iteratively refine their queries without losing context. Each interaction is tracked with a session ID, ensuring that fixes are applied progressively and maintain alignment with previous exchanges. This approach significantly improves the system's ability to understand and build upon prior interactions.

Standard Macro Utilization and Error Resolution

The utilization of standard macros within SASBuddy is facilitated by injecting a JSON function schema of the macros into the process, parsed from the Doxygen macro headers. This schema provides the LLM with a detailed blueprint of available macros, including their functions, parameters, and usage conventions. Additionally, the system incorporates react-style code fixing, where only faulty lines of code are modified, reducing redundancy and improving debugging efficiency.

Knowledge Augmentation and Retrieval

To further enhance accuracy, SASBuddy integrates retrieval-augmented generation (RAG) and external search mechanisms. If the system detects uncertainty in its responses, it performs Search-O1 (Li et al., 2025), allowing it to extract authoritative information efficiently rather than relying solely on model-generated completions. This augmentation strategy makes SASBuddy significantly more reliable than standalone LLM-based coding assistants by iteratively retrieving, refining, and integrating documents into the reasoning chains.

Automated Feedback and Versioning

SASBuddy's feedback loop mechanism captures execution logs and errors, feeding them back to the LLM for analysis and code refinement. This iterative improvement process is complemented by a versioning system that records successful prompts and responses, creating a valuable repository for fine-tuning future interactions (Yao et al., 2023). This combination of immediate feedback and historical learning ensures continuous improvement in code generation accuracy.

SASBUDDY: OPERATIONALIZING THE SOLUTION

SASBuddy implements a sophisticated architecture that decouples execution from intelligence while maintaining robust data handling and error correction capabilities. The system operates through distinct phases that ensure secure, efficient, and accurate code generation.

Setup Phase:

The process begins with two critical initialization steps:

1. **Metadata Extraction**: Users run the extract_metadata macro, which scans the datasets of interest and creates metadata.json. This file details the structure and attributes of the data while abstracting away any confidential or patient-identifiable information to maintain privacy.

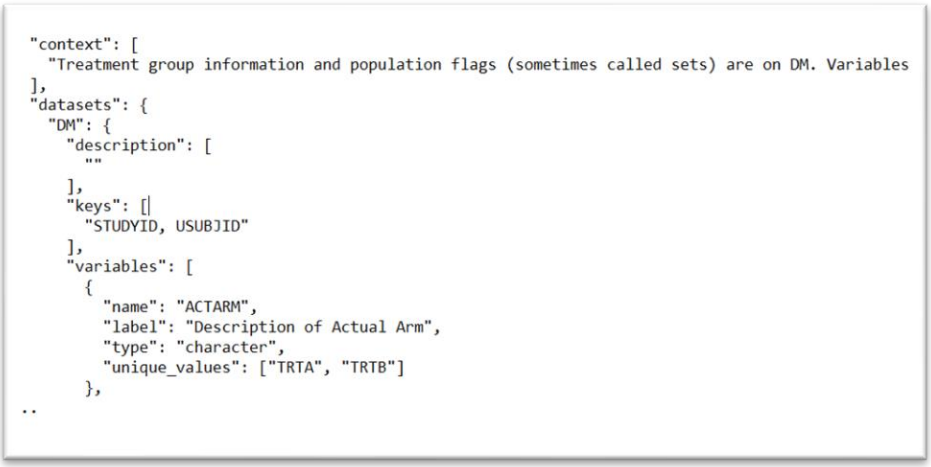

Figure 3. Excerpt of metadata.json, used to pass in data context for code generation.

The metadata includes:

- SDTM/ADaM specific rules and metadata from company specifications
- Variable labels for natural language mapping
- Key variables for unique dataset identification
- Codelists (unique_values) for parameter-specific subsetting

2. **Macro Documentation**: Concurrently, the extract_macros macro analyzes Doxygen headers of standard SAS macros, compiling functionality details into macros.json. This catalog provides a comprehensive reference of available macros, their parameters, and usage patterns.

3. **Intelligence Layer Architecture**

Decoupling Execution from Intelligence

A key architectural innovation in SASBuddy is the **separation of SAS execution (client-side) from intelligence (API-side)**. The Python-powered API side handles:

- Session persistence and state management
- Query refinement and optimization
- Document retrieval and knowledge augmentation
- Algorithmic code correction

This separation enables sophisticated enhancements that would be impractical in a purely SAS-based solution while maintaining robust security boundaries.

Session-Aware Processing

Building on principles established in recent work on language agents, SASBuddy implements comprehensive session tracking where:

- Each interaction is assigned a unique session ID
- Execution history and error patterns are preserved
- Contextual metadata informs subsequent code generation
- Progressive refinements maintain consistency across interactions

React-Style Code Fixing

The system implements a reactive error-correction framework inspired by chain-of-thought prompting techniques (Wei et al., 2023):

- Only erroneous code segments are modified
- Changes are surgically precise to minimize side effects
- Refinements incorporate execution feedback
- Pattern recognition guides correction strategies

Knowledge Integration and Search-O1

SASBuddy enhances its responses through multiple knowledge integration mechanisms (Li et al., 2025):

- Dataset metadata validation
- Standard macro repository integration
- Search-O1 for efficient document retrieval
- Reason-in-Documents for iterative knowledge refinement

Execution and Feedback Loop

With these architectural components in place, the operational flow proceeds as follows:

1. **Query Formation**: Users pose natural language questions to SASBuddy, referencing the metadata.json and macros.json locations. The system crafts queries using reflexion and few-shot learning techniques (Yao et al., 2023).

2. **Code Generation**: The intelligence layer generates SAS code with embedded explanations and guidance, leveraging the full context of the session history and available metadata.

3. **Execution Flow**: When execute="Y" is specified:
   - Code executes in the client environment (e.g., SAS Studio)
   - Error messages are captured and analyzed
   - The intelligence layer refines code based on feedback
   - This loop continues until success or iteration limit

4. **Learning Integration**: Successful interactions are logged to a Training Data Repository, building a knowledge base for future refinements.

```
/* Set the graphics environment */
goptions reset=all cback=white border htitle=12pt htext=10pt;

/* Create a boxplot */

proc sgplot data=DM;
    title "Boxplot of Age by Actual Arm"; /* Set the title of the boxplot */
    footnote "Created using the SAS System"; /* Add a footnote */
    /* Set the X and Y variables and the color of the line */
    hbox AGE / group=ACTARM lineattrs=(color=blue)
        /* Set the color of the box and the transparency level */
        fillattrs=(color=lightblue) transparency=0.5;
    xaxis label="Actual Arm"; /* Set the label for the X-axis */
    yaxis label="Age"; /* Set the label for the Y-axis */
run;
```

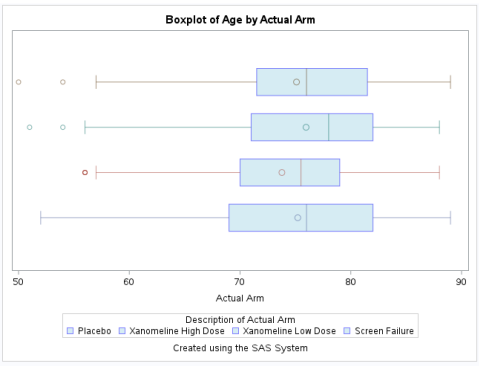

Figure 4 The generated code and the executed output for the query "create boxplot of age by actual arm. Add a relevant title and footnote and aesthetically pleasing colors" to SASBuddy.

The system maintains this iterative improvement cycle while preserving security boundaries and ensuring that sensitive data never leaves the client environment. This architecture enables SASBuddy to provide increasingly accurate and context-aware assistance while adhering to strict clinical programming standards.

CONCLUSION

The evolution of SASBuddy represents a shift in AI-assisted SAS programming—from basic LLM-based code generation to an intelligent, session-aware, retrieval-augmented, and execution-aware coding assistant (Chernyavskiy et al., 2021). By decoupling intelligence from execution, SASBuddy enables powerful Python-based enhancements, such as react-style debugging, structured error correction, and Search-O1 knowledge retrieval (Li et al., 2025). The integration of Reason-in-Documents mechanisms further refines its ability to dynamically retrieve and apply authoritative knowledge. These innovations ensure that SAS programmers benefit from a continuously improving, context-aware, and domain-augmented AI assistant that significantly enhances efficiency in clinical programming workflows.

REFERENCES

OpenAI, Chatgpt https://chat.openai.com, (2023), Accessed: 2023-07-30.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention Is All You Need. arXiv preprint arXiv:1706.03762. Available at: https://doi.org/10.48550/arXiv.1706.03762

Chernyavskiy, A., Ilvovsky, D. and Nakov, P., 2021. Transformers: "The End of History" for NLP? arXiv preprint arXiv:2105.00813. Available at: https://arxiv.org/abs/2105.00813.

Ananthaswamy, A. (2023). In AI, is bigger always better? Nature, 615, 202-205. https://doi.org/10.1038/d41586-023-00641-w

Demand Sage. (n.d.). Road to 100 Million Users For Various Platforms. [online] Available at: https://www.demandsage.com/chatgpt-statistics/ [Accessed on: 10 October 2023].

X. Wei, X. Cui, N. Cheng, X. Wang, X. Zhang, S. Huang, P. Xie, J. Xu, Y. Chen, M. Zhang et al., Zero-shot information extraction via chatting with chatgpt, arXiv preprint arXiv:2302.10205 (2023).

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D., 2020. Language Models are Few-Shot Learners. arXiv. Available at: https://arxiv.org/abs/2005.14165

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K. and Yao, S., 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv preprint arXiv:2303.11366. Available at: https://arxiv.org/abs/2303.11366.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q. and Zhou, D., 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv preprint arXiv:2201.11903. Available at: https://arxiv.org/abs/2201.11903.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y. and Narasimhan, K., 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv preprint arXiv:2305.10601. Available at: https://arxiv.org/abs/2305.10601.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T.,

2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288. Available at: https://arxiv.org/abs/2307.09288.

H. Face, Transformers language modeling https://github.com/huggingface/transformers/tree/main/examples/pytorch/language-modeling, (2023).

L. Team, Llama.cpp https://github.com/ggerganov/llama.cpp, (2023).

Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. (2023). A Survey of Large Language Models. arXiv. Available at: https://arxiv.org/abs/2303.18223.

Li, X., Dong, G., Jin, J., Zhang, Y., Zhou, Y., Zhu, Y., Zhang, P. and Dou, Z., 2025. Search-O1: Agentic Search-Enhanced Large Reasoning Models. arXiv preprint arXiv:2501.05366. Available at: https://arxiv.org/abs/2501.05366

Scialom, T., Duval, F., Lambert, P., von Platen, P., Kennedy, R., Fan, A., Chang, Y-L., Schmid, C., Lachaux, M-A., Labeau, M. and Li, X., 2024. Building Blocks for Language Model Agents: Tools, Skills and Workflows. arXiv preprint arXiv:2401.17370. Available at: https://arxiv.org/abs/2401.17370

Zhang, Y., Yang, C., Li, X., Dong, G., Jin, J., Zhou, Y., Zhang, P. and Dou, Z., 2024. Agentic Frameworks for Large Language Models: A Systematic Survey. arXiv preprint arXiv:2402.02691. Available at: https://arxiv.org/abs/2402.02691

Zhu, Y., Zhang, Y., Li, X., Jin, J., Zhou, Y., Zhang, P. and Dou, Z., 2024. Knowledge Management in Large Language Model Agents: A Survey. arXiv preprint arXiv:2401.10602. Available at: https://arxiv.org/abs/2401.10602

APPENDIX A: INCORPORATING STANDARD MACROS INTO SASBUDDY

To demonstrate SASBuddy's of standard macros within its code generation process, we present a simplified example that outlines how these macros are selected and implemented based on the user's query and the contextual metadata provided.

Context: Let's consider a scenario where a user wants to calculate the mean age, but wants to use standard macros instead of raw SAS code. Let's also assume we have the following standard macros in our library.

```
/**
 * @brief Calculates the mean of a numeric variable in a SAS dataset.
 * @param ds_name Name of the SAS dataset.
 * @param var_name Name of the variable to calculate the mean for.
 */
%macro calc_mean(ds_name=, var_name=);
    proc means data=&ds_name noprint;
        var &var_name;
        output out=mean_result mean(&var_name)=mean_var;
    run;
%mend calc_mean;

/**
 * @brief Filters a SAS dataset based on a condition and creates a new dataset.
 * @param input_ds Name of the input SAS dataset.
 * @param output_ds Name of the output SAS dataset.
 * @param condition The condition to filter the dataset on (e.g., age > 18).
 */
%macro filter_dataset(input_ds=, output_ds=, condition=);
    data &output_ds;
        set &input_ds;
        if &condition;
    run;
%mend filter_dataset;
```

Through running *extract_macro* we get the following JSON in our *macros.json* file which describes the macro arguments available in our macro library.

```
{ "role": "assistant", "function_call": { "name": "calc_mean", "arguments": "{ds_name,var_name}" } }
{ "role": "assistant", "function_call": { "name": "filter_dataset", "arguments": "{input_ds,output_ds,condition}" } }
```

Now, when the user submits a query to SASBuddy and provides a macro location, SASBuddy has enough context to first understand which variables and domains to use as this is already provided in the metadata.json file.

```
%SASBuddy(question=filter the dm dataset for patients older than 20 then calculate the mean age,
          output_file = &srcdir/gptx.sas,
          metadata_path=&srcdir/metadata.json,
          macros_path = &srcdir/macros.json);
```

It also has sufficient context to identify if a standard macro(s) exist that can answer the user's question and what the necessary parameters should be.

SASBuddy's Generated Code:

```
/* Filter the dataset for occurences where Age is more than 20 years */
%filter_dataset(input_ds=DM, output_ds=DM_filtered, condition=AGE > 20);

/* Now on the filtered dataset, calculate mean Age */
%calc_mean(ds_name=DM_filtered, var_name=AGE);
```

CONTACT INFORMATION

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

Karma Tarap
Bristol Myers Squibb
Boudry / 2017
Email: karma.tarap@bms.com