TEXT SUMMARIZER USING JULIA

Amey Thakur[1] and Mega Satish[2]

[1-2]Department of Computer Engineering, University of Mumbai, Mumbai, Maharashtra, India

[1]ameythakur20@gmail.com

[2]mega.modha@gmail.com

\

***Abstract*— The purpose of this paper is to introduce the Julia programming language with a concentration on Text Summarization. An extractive summarization algorithm is used for summarizing. Julia's evolution and features, as well as comparisons to other programming languages, are briefly discussed. The system's operation is depicted in a flow diagram, which illustrates the processes of sentence selection.**

***Keywords*— Text Summarizer, Extractive Summarization, Sentence Score, Topic Representation, Julia Programming Language.**

I. Introduction

People are being overloaded by the massive number of digital content and papers as the Internet has grown dramatically. The need for material that has been summarized rather than read has also grown. We can now summarize text thanks to advances in machine learning and deep learning. Furthermore, the amount of data available online is massive, making it easier to train the models. The challenge of generating a short and fluent summary yet keeping vital information material and general context is known as automatic text summarization. Throughout the past few years, multiple techniques for text summarization have been devised and widely used in a variety of applications. For instance, search engine crawlers create snippets as document previews. Other examples are news websites that offer abbreviated summaries of news subjects, typically in the form of headlines, to aid browsing or knowledge extraction methodologies. Automatic text summarizing is difficult since, when we summarize a piece of text, we normally read it completely to build our knowledge, and then create a summary stressing its important points. Because computers and technology lack basic human understanding and linguistic aptitude, automated text summarization is a tough and time-consuming operation.

There are two techniques to automated summarization in general. Extractive summarizing approaches function by detecting key chunks of the text and reproducing them exactly; consequently, they rely only on phrase extraction from the original text. Abstractive summarization approaches, on the other hand, strive to generate important content in a novel way. In other words, they evaluate and analyze the content using powerful natural language techniques to develop a new shorter text that provides the most important information from the original text. Whereas most human-created summaries are not extractive, the majority of summarizing research today is focused on extractive summarization. When compared to automatic abstractive summaries, pure extractive summaries frequently produce better outcomes. That’s because abstractive summarization approaches deal with difficulties like semantics, reasoning, and natural language production, which are more difficult than data-driven methods.

II. Related Work

A. Extractive Summarization

Extractive summarization [1][3] algorithms generate summaries by selecting a subset of the sentences in the original text. The most essential sentences from the input are included in these summaries. A single document or a collection of documents can be used as input. To get a better understanding of how summarizing systems function, we define three somewhat distinct activities that all summarizers do: First, they create an intermediate representation of the input text that communicates the text's essential points. Then give each sentence a score depending on its representation. Finally, they choose a summary made up of a few phrases.

While summarizing a text, every summarizing system develops an intermediate representation of it. It then applies this representation to find the most important material. Topic representation and indicator representation are two forms of representation-based techniques. Topic representation methods convert the text into an intermediate representation and understand the text's topic(s). The sophistication and representation model of topic representation-based summarization algorithms varies, and they are separated into frequency-driven approaches, topic word approaches, latent semantic analysis, and Bayesian topic models. Every sentence is described as a set of important qualities (indicators) such as sentence length, position in the text, and the presence of certain words in indicator representation techniques [2].

B. Sentence Score

When working with textual data, sentence score [4] is one of the most commonly used procedures in the field of Natural Language Processing (NLP). It's a method of associating a numerical number with a statement dependent on the priority of the algorithm being employed. This method is widely used, particularly for text summaries. Many prominent sentence scoring systems exist, including TF-IDF, TextRank, and others [4].

We give a significance score to each sentence when the intermediate representation is created. In topic representation techniques, a sentence's score measures how effectively it communicates some of the text's most essential themes. The score is derived in most indicator representation systems by pooling data from several indicators.

C. Topic Words Representations

One of the most prominent topic representation techniques [3] is the topic words procedure, which seeks to find words that characterize the source text's topic. In the news domain, using topic signature terms as topic representation was also quite beneficial and boosted the accuracy of complex abstract summarization. See this page for further information on the log-likelihood ratio test. The relevance of a sentence may be calculated in two ways: as a function of the number of subject signatures in the sentence or as a proportion of the topic signatures in the sentence. All sentence scoring systems use the same subject representation, but they may give sentences wildly different results. Lengthier phrases may receive better marks under the first technique since they include more information. The frequency of the topic terms is measured using the second method.

III. Model Architecture

A. Flow Diagram

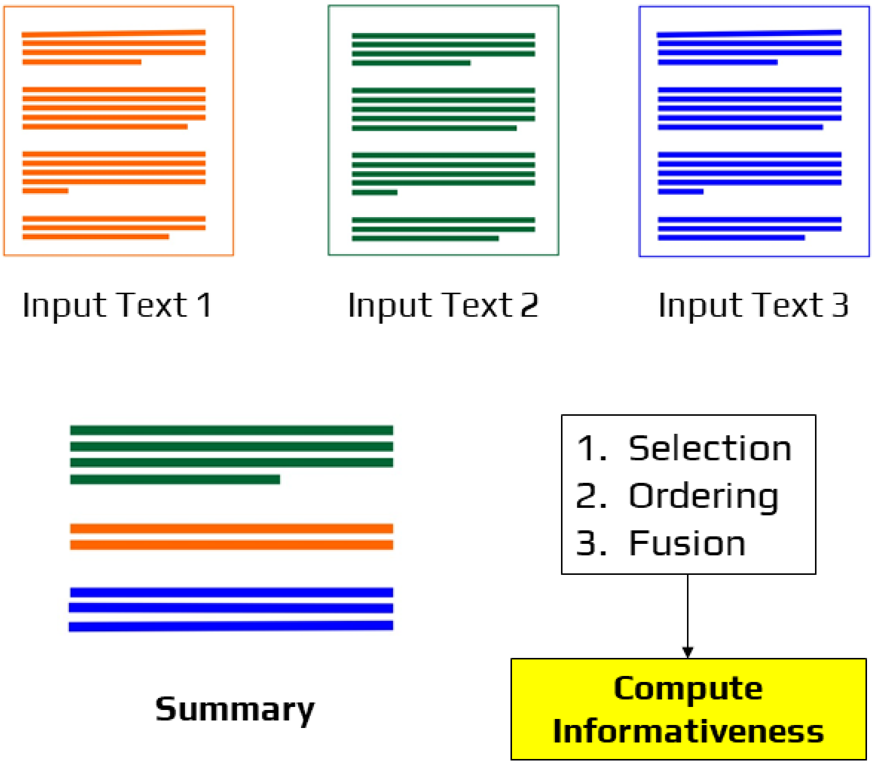


Figure 1: Flow Diagram

B. *Working of the system*

The working of the text summarizer system is very simple and easy to understand and implement in Julia [5]. As explained above, Julia is a high computational language that has been recently developed. It's faster and easier than python which has been by far the easiest programming language to learn.

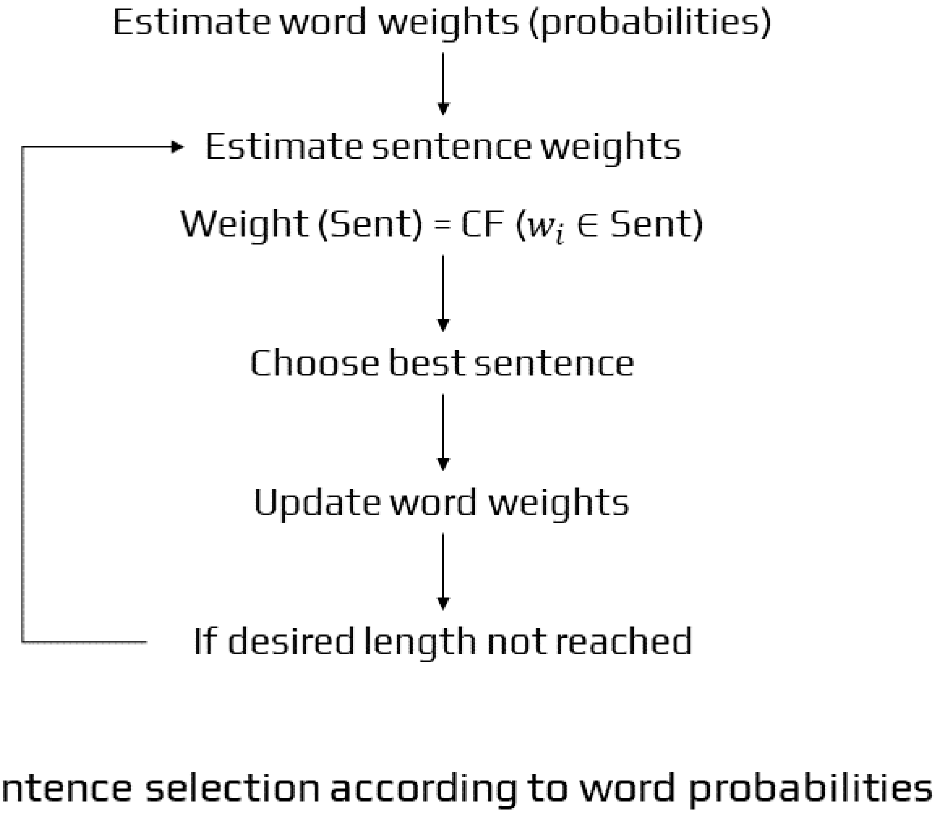
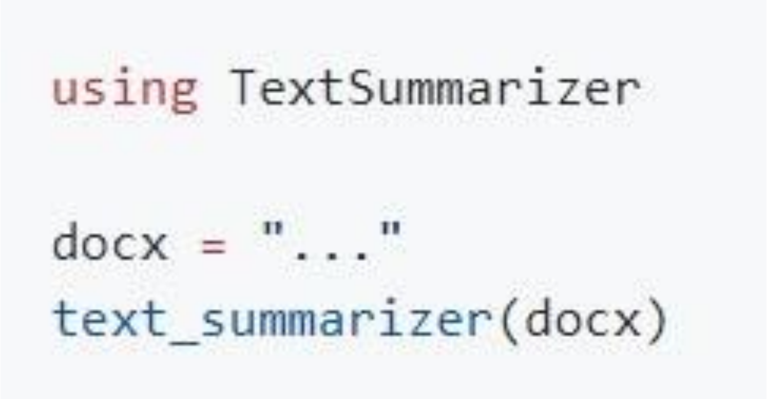

Estimate word weights (probabilities)
Estimate sentence weights
Weight (Sent) = CF (w_i ∈ Sent)
Choose best sentence
Update word weights
If desired length not reached
Sentence selection according to word probabilities

Figure 2: Flowchart of Extractive Summarization

For our system, we first specify the stop words, which are terms that are not required and are removed to simplify the content. Then the sentence is split and each word is tokenized; if the length of the sentence is greater than 30, then that sentence is not considered. Furthermore, we determine the frequency of each word in the sentences by counting the number of times that particular word appears. We then calculate the sentence score after obtaining the word frequency. Based on the sentence score, we get the extracted summary from the original text. We have basically created a function capable of summarizing any text. We, later on, call the library in which the function is written in our driver code. The below code shows how the function can be called and summarizes a doc file.



```
using TextSummarizer

docx = "..."
text_summarizer(docx)
```

Figure 3: Code Snippet

C. Why Julia

Julia is a relatively new programming language that was originally released in 2012 and aspires to be both simple and quick. "It runs like C but reads like Python,"[8]. It was designed for scientific computing, with the ability to handle enormous quantities of data and processing while remaining relatively simple to manage, generate, and prototype code. The developers of the languages stated the reasons why they created this language: “We seek an open-source language with a permissive licence […] Ruby's dynamism combined with C's speed. […] a homoiconic language with genuine macros, similar to Lisp, but with obvious, familiar mathematical notation, similar to Matlab. […] as excellent at glueing programmes together as the shell, as good at general programming as Python, as easy to use for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, and as natural for string processing as Perl. Something that is really simple to understand while also satisfying the most ardent hackers”.

In pursuit of that goal, people from all over the world have been constantly updating and improving Julia, resulting in a vibrant and robust network. Over 700 people have contributed to Julia, and countless others have produced amazing open-source Julia packages [5].

Summarization is essential in today’s world. It is becoming more necessary to summarize News Articles, Scientific Articles, Books, Websites, Social Media Streams and Emails as daily tonnes of new data are released, making it a must to keep relevant material accessible in order to save time. This topic was quite interesting to implement in Julia [6] as the language is very much flexible to use.

D. Julia Features

1. Dynamic: Julia is dynamically typed, has a scripting aspect to it, and supports interactive use quite well.
2. Faster: Julia was designed from the start-up to be a greater system. On a range of computers, the Low-Level Virtual Machine (LLVM) translates Julia programs into efficient native code.
3. General: It does numerous relays as a model, which makes it easier to implement object-oriented programming design. The standard library contains asynchronous I/O, process control, logging, profiling, package management, and other features.
4. Easy Use Syntax: Julia is a very rich language in terms of its syntax. The declarations of the data type can be used to define and strengthen computations in Julia.
5. Complexity: Julia language is great when it comes to handling complex computational tasks [7] as it is easy to write mathematical formulas because of Julia’s syntax. Also, it supports a wide range of predefined arithmetic types of data, and established parallelism.

E. Comparison with other languages

When it comes to speed, languages like C and Fortran are very fast as they are compiled before the execution. However, in terms of complexity and learning, it takes more time to get used to and implement it. In this case, Julia is very fast and very easy to write as well. Julia, like C or Fortran, is compiled, so it runs quickly. Julia, on the other hand, is compiled at runtime. So it appears to be an interpreted language: you can create a script, click 'run,' and it works exactly like Python [7].

Most generated languages utilize static typing since the compiler can only generate machine code for specified kinds — integers, floats, and so on. As a result, the programmer must define the type of each variable, making the code somewhat fragile. In comparison to dynamically typed languages such as Python, where every variable can be any type at any moment, this makes coding a little... hard. So, how does Julia handle dynamic typing while also compiling it before execution?

Here’s how: Julia reads the instructions and compiles them for the types it finds (a process known as type inference), then generates and stores the bytecode [8]. When you run the same commands with a different type, Julia recompiles for that type and caches the resulting bytecode in a separate place. Later runs recompile and utilize the proper bytecode.

Additionally, Unicode characters are supported as variables and arguments in Julia. This means that you should no longer use sigma or sigma i, but rather σ or σi in mathematical notation. When you look at code for an algorithm or a mathematical problem, you'll see that the notation and idioms are nearly identical. This is a remarkable feature that we call "One-To-One Code and Math Relationship." This is a feature that is only possible in Julia.

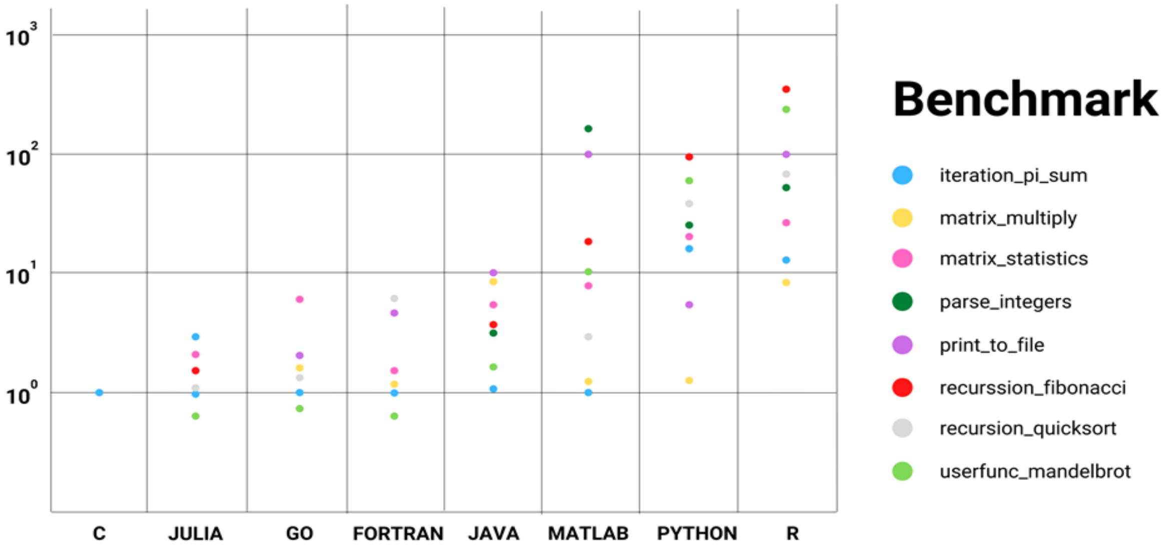
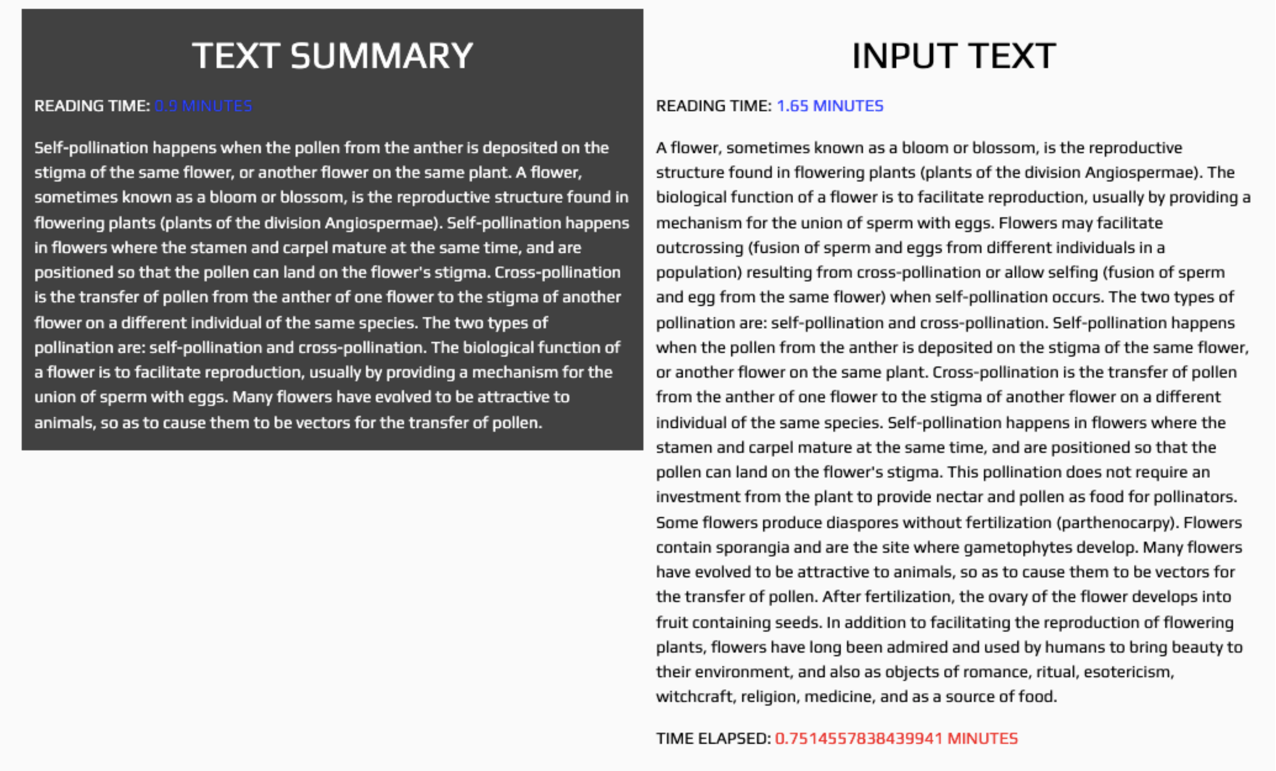


Figure 4: Comparison of Computational Resource for different operations in programming languages

F. Results

As the method used is extractive summarization, the summarized text would be sentences that are extracted based on the sentence score. The system is less efficient as it requires additional and better techniques to enhance its performance. It is in the initial stage, so further work is needed.



TEXT SUMMARY

READING TIME: 0.9 MINUTES

Self-pollination happens when the pollen from the anther is deposited on the stigma of the same flower, or another flower on the same plant. A flower, sometimes known as a bloom or blossom, is the reproductive structure found in flowering plants (plants of the division Angiospermae). Self-pollination happens in flowers where the stamen and carpel mature at the same time, and are positioned so that the pollen can land on the flower's stigma. Cross-pollination is the transfer of pollen from the anther of one flower to the stigma of another flower on a different individual of the same species. The two types of pollination are: self-pollination and cross-pollination. The biological function of a flower is to facilitate reproduction, usually by providing a mechanism for the union of sperm with eggs. Many flowers have evolved to be attractive to animals, so as to cause them to be vectors for the transfer of pollen.

INPUT TEXT

READING TIME: 1.65 MINUTES

A flower, sometimes known as a bloom or blossom, is the reproductive structure found in flowering plants (plants of the division Angiospermae). The biological function of a flower is to facilitate reproduction, usually by providing a mechanism for the union of sperm with eggs. Flowers may facilitate outcrossing (fusion of sperm and eggs from different individuals in a population) resulting from cross-pollination or allow selfing (fusion of sperm and egg from the same flower) when self-pollination occurs. The two types of pollination are: self-pollination and cross-pollination. Self-pollination happens when the pollen from the anther is deposited on the stigma of the same flower, or another flower on the same plant. Cross-pollination is the transfer of pollen from the anther of one flower to the stigma of another flower on a different individual of the same species. Self-pollination happens in flowers where the stamen and carpel mature at the same time, and are positioned so that the pollen can land on the flower's stigma. This pollination does not require an investment from the plant to provide nectar and pollen as food for pollinators. Some flowers produce diaspores without fertilization (parthenocarpy). Flowers contain sporangia and are the site where gametophytes develop. Many flowers have evolved to be attractive to animals, so as to cause them to be vectors for the transfer of pollen. After fertilization, the ovary of the flower develops into fruit containing seeds. In addition to facilitating the reproduction of flowering plants, flowers have long been admired and used by humans to bring beauty to their environment, and also as objects of romance, ritual, esotericism, witchcraft, religion, medicine, and as a source of food.

TIME ELAPSED: 0.7514557838439941 MINUTES

Figure 5: Sample of Result

IV. CONCLUSION

Because of the Internet's rapid growth, a large amount of data is now available. Humans find it difficult to summarize large amounts of text. As a result, there is a high demand for automatic summarizing solutions in this age of information overload. In this research, we suggested a text summarizing system using Julia that is based on the extractive approach. As a paradigm, this new language employs multiple dispatches, making it simple to express a wide range of object-oriented and functional programming patterns. The standard library includes asynchronous input and output, operational processes, logging, profiling, a package manager, and other features. Until now, no text summarizing has been implemented in this language. The algorithm we developed computes the word frequency and sentence score and then provides a summary based on the highest sentence score. The system is currently in its first stages of development. If we raise the sentence's complexity, more problems may appear. It's still a pretty simple summarizer. However, more work on the system, particularly the implementation of a machine learning model with Julia, would greatly improve the project because Julia is a high computing language.

REFERENCES

[1] Nenkova, Ani, and Kathleen McKeown. "A survey of text summarization techniques." In Mining text data, pp. 43-76. Springer, Boston, MA, 2012.

[2] Allahyari, Mehdi, Seyedamin Pouriyeh, Mehdi Assefi, Saeid Safaei, Elizabeth D. Trippe, Juan B. Gutierrez, and Krys Kochut. "Text summarization techniques: a brief survey." arXiv preprint arXiv:1707.02268 (2017).

[3] Moratanch, N., and S. Chitrakala. "A survey on extractive text summarization." In 2017 international conference on computer, communication and signal processing (ICCCSP), pp. 1-6. IEEE, 2017.

[4] Ferreira, Rafael, Luciano de Souza Cabral, Rafael Dueire Lins, Gabriel Pereira e Silva, Fred Freitas, George DC Cavalcanti, Rinaldo Lima, Steven J. Simske, and Luciano Favaro. "Assessing sentence scoring techniques for extractive text summarization." Expert systems with applications 40, no. 14 (2013): 5755-5764.

[5] Bezanson, Jeff, et al. "Julia Language Documentation." The Julia Manual. http://docs. julialang. org/en/release-0.2/manual (2014): 1-261.

[6] Gao, Kaifeng, Gang Mei, Francesco Piccialli, Salvatore Cuomo, Jingzhi Tu, and Zenan Huo. "Julia language in machine learning: Algorithms, applications, and open issues." Computer Science Review 37 (2020): 100254.

[7] Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A fresh approach to numerical computing." SIAM review 59, no. 1 (2017): 65-98.

[8] Jeffrey M. Perkel, “Julia: Come for the syntax, Stay for the speed.” Volume 572, August 2019, Nature. DOI: https://doi.org/10.1038/d41586-019-02310-3