

Alumnos:

Cotarelo, Rodrigo 98577

Longo, Nicolás 98271

Reffle, Lucía 97788

Materia: Teoría del lenguaje

Fecha: 1° Cuatrimestre 2018

Origen e historia:

El lenguaje Julia comienza a gestarse en 2009 como un proyecto liderado por Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman con el objetivo de crear un lenguaje que tuviera una gran capacidad computacionalmente y fuera veloz.

En Febrero de 2012 se lanzó y en 2015 se creó Julia Computing para proveer soporte, entrenamiento y servicios de consultoría a clientes (aquellos que ya utilizaban el lenguaje y potenciales interesados). Desde entonces ha resultado un lenguaje sumamente atractivo para resolver problemáticas de alto contenido matemático y análisis numérico en disciplinas como, sólo para dar los ejemplos más importantes, programación concurrente y distribuida y *data science*.

Para enero del 2018, la comunidad de Julia ascendía a 1.800.000 descargas.

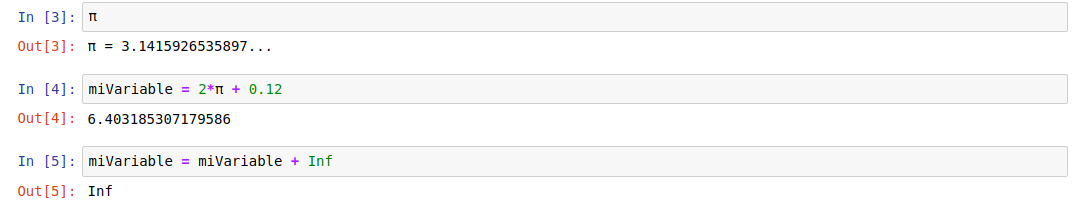
Carecterísticas básicas

**Manejo de variables y constantes matemáticas:**

Julia es un lenguaje que busca cooptar la atención de la comunidad científica y especializarse en la facilidad y el poder de manejo matemático. Es por esto que cuenta con características muy interesantes. Para empezar cualquier variable puede adoptar cualquier valor (si en principio no se declara su tipo, más sobre esto más adelante).



Julia cuenta con una serie de constantes matemáticas predefinidas.

También permite el manejo de números irracionales.

Y números racionales (minimizando la pérdida de dígitos por redondeo).

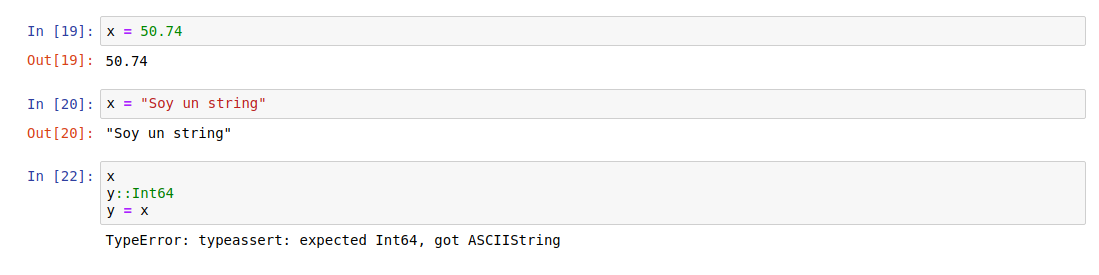


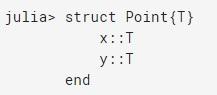
**Paradigmas:**

Julia es un lenguaje multiparadigma y como tal permite abordar programación tanto imperativa como funcional y orientada a objetos. Para ello se pueden definir funciones convencionales, funciones lambda, clases e interfaces, solo para mencionar algunas características de estos paradigmas, y combinarlos.

**Tipado**

En Julia el tipado es dinámico y los tipos son, por defecto, omitidos de manera tal que las variables admitan valores de cualquier tipo. Y esto se da así: son los valores los que tienen tipo mientras que las variables son simples notaciones que hacen referencia a entidades. De todas formas, el compilador también permite la especificación del tipo esperado para una cierta variable. Esto suele mejorar no solamente la robustez de los programas, sino también su performance.

Una característica interesante de Julia es que la declaración de nuevos tipos puede llevar parámetros y esto introduce una versatilidad importante.



Así, a partir de Point {T} podemos tener una variable que sea de tipo Point {Float64} o Point {Abstract String}, con lo cual tenemos toda una familia de tipos a partir de la definición de una con un parámetro T. Esto es extensible a varios parámetros.

En Julia, todos los tipos son objetos, pero sus métodos no "pertenecen" a la clase, sino que son parte de un listado que el compilador utiliza a la hora de interpretar el código. Ésta es una consideración del diseño del lenguaje que favorece el Multiple Dispatch.

**Compilacion e interpretación**

Julia posee un compilador denominado JIT (“Just in Time”) que aunqe se utiliza en tiempo de ejecución, el compilador infiere los tipos a través de una primera compilación o “precompilación”. Mediante este mecanismo el lenguaje aparenta ser interpretado en términos de velocidad y así es comparable a lenguajes como C o Fortran, pero con un código muchísimo más simple.

**Manejo de errores:**

Ante un evento inesperado, Julia tiene un manejo de errores comparable al de la mayoría de los lenguajes. Si la función no puede devolver una salida razonable, entonces levanta una excepción y termina el programa indicando un código de error. En algunos casos, el programador puede especificar hasta el mensaje de error a mostrar.

**Funciones y argumentos**

Ejemplo de función en Julia que devuelve dos parámetros.

En Julia se pueden utilizar expresiones lambda

Los argumentos en Julia se pasan siguiendo una convención que suele llamarse "pass-by-sharing". Esto significa que no se pasan por copia, sino que los argumentos funcionan en sí mismos como una nueva variable que tiene idéntico valor al que ha sido pasado.

También se puede definir valores de argumentos por default

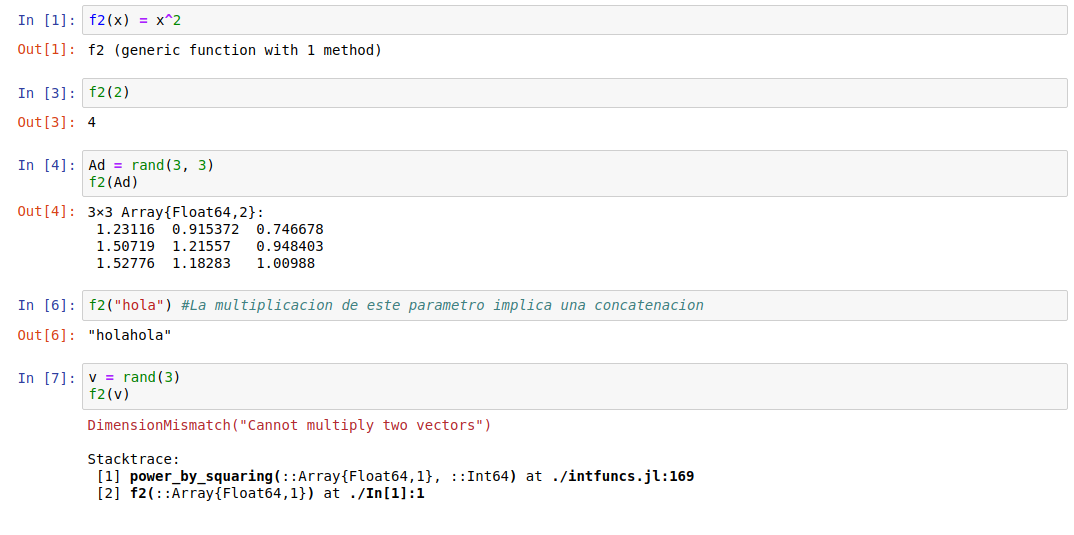


**Duck Typing**

Duck typing es el estilo de tipificación dinámica de datos en que el conjunto actual de métodos y propiedades determina la validez semántica, en vez de que lo hagan la herencia de una clase en particular o la implementación de una interfaz específica.

En duck typing, el programador solo se ocupa de los aspectos del objeto que van a usarse, y no del tipo de objeto que se trata. Por ejemplo en un lenguaje sin duck-typing uno puede crear una función que toma un objeto de tipo Pato y llama los métodos "caminar" y "parpar" de ese objeto. En un lenguaje con duck-typing, la función equivalente tomaría un objeto de cualquier tipo e invocaría los métodos caminar y parpar. Si el objeto tratado no tiene los métodos pedidos, la función enviará una señal de error en tiempo de ejecución.

En los siguientes ejemplos se puede ver cómo Julia intentará aplicarle a mis datos la función pedida, sean del tipo que sean.

Como vemos, en el último ejemplo el compilador levantó una excepción porque la operacion definida en f2 es ambigua ya que hay diferentes manera de multiplicar un vector.

**Funciones mutantes y no mutantes:**

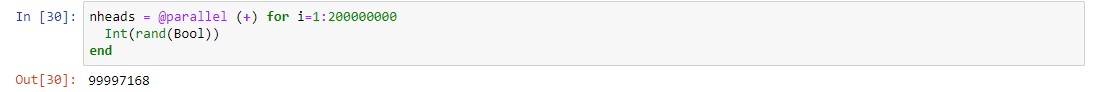
Por convención las funciones seguidas por un “!” alteran, o bien mutan, el contenido de sus argumentos y las que carecen de un “!”, no lo hacen.



**Paralelismo:**

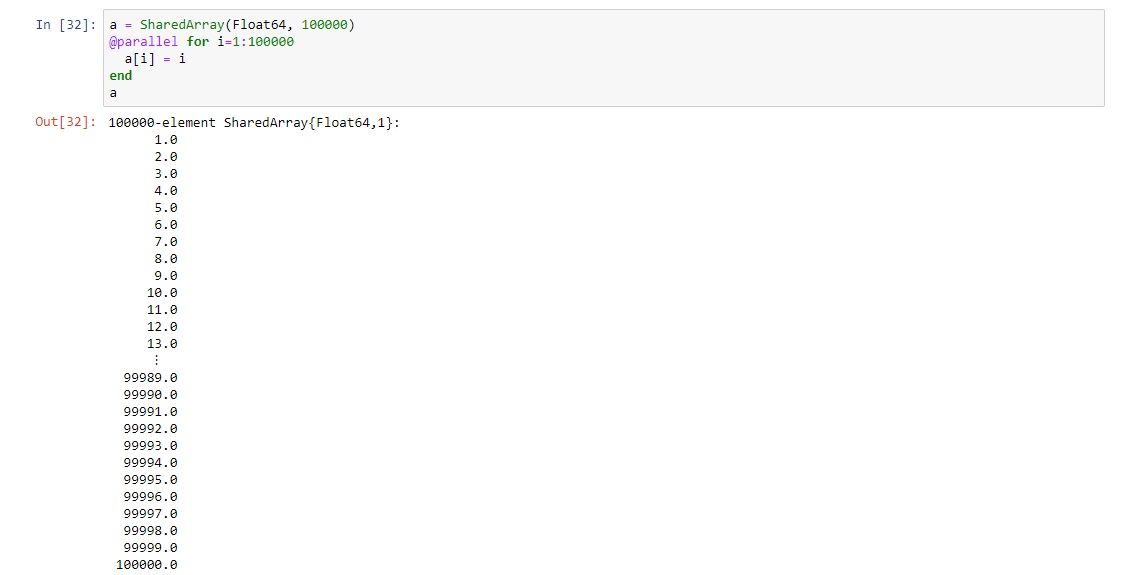
Como Julia está pensado para ser útil para la computación científica busca favorecerla permitiendo utilizar todos los recursos disponibles en una máquina, como los procesadores múltiples. Por esto Julia permite correr operaciones en paralelo, este se basa en el paso de mensajes entre programas corriendo en varios procesadores con memoria compartida, donde cada proceso tiene un id (siendo el 1 el principal y el resto “workers”).

Este paso de mensajes es unilateral y está construido sobre dos recursos: remote references y remote calls. Al hacerse un remote call se obtiene una remote reference *Future* y el valor de esta se puede recuperar usando la función fetch ().

Ciclos

Este es un ciclo for que se realiza de manera paralela entre los workers disponibles y la última operación suele ser conocida como reducción, la cual hace parte de un patrón de computación paralela o distribuida bastante usual.

Al usar un for paralelo hay que tener en cuenta que las instrucciones no se ejecutan en un orden específico y que no se está manejando al mismo worker, por lo que las escrituras a variables y arrays no serán visibles globalmente ya que las iteraciones corren en distintos procesos y cada uno tendrá su propia copia de la variable o array. Por esto se necesitan utilizar variables como Shared Arrays.

Se puede reemplazar el @Parallel utilizando la función pmap para que ejecute una función en paralelo dividendo el trabajo entre los workers donde cada llamada a función realiza una gran cantidad de trabajo.

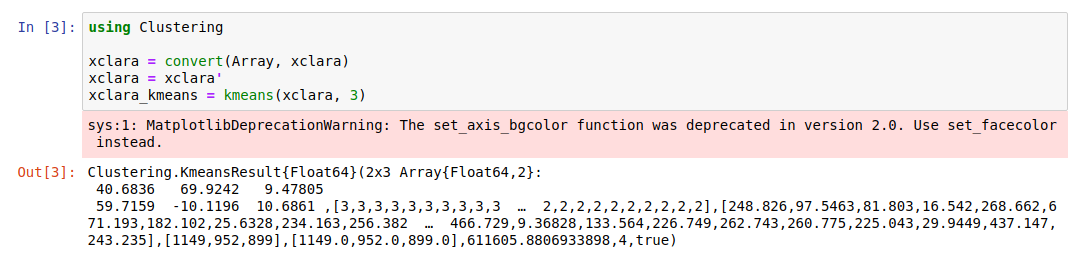
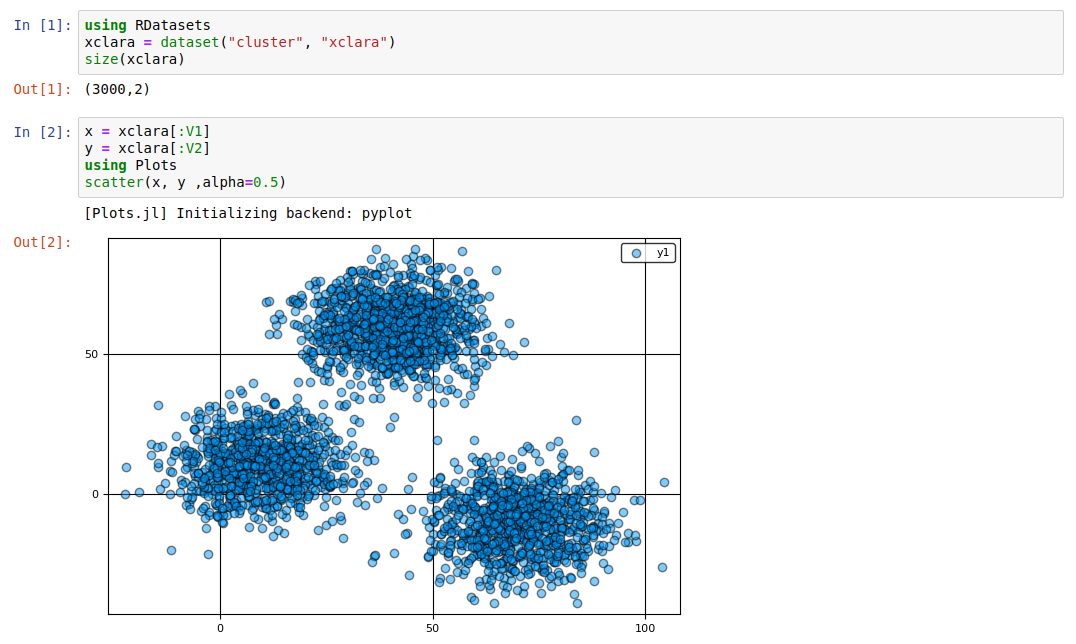
Variables Globales

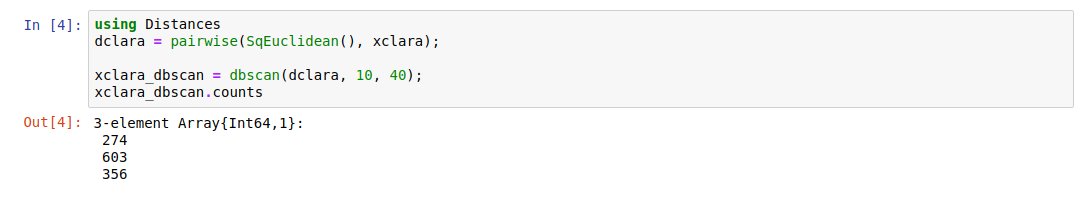
Las llamadas remotas con referencias globales integradas (solo en el módulo principal) administran las variables globales generando bindings de estas variables en los procesos a los que se llama, volviendolo a enviar a un trabajador de destino sólo si su valor ha cambiado.

**Ejemplo de Machine Learning:**

Este punto es muy importante a la hora de estudiar el lenguaje, ya que éste ha sido creado para encarar problemáticas computacionales como las que el programador se topa clásicamente en Machine Learning. Esto es, una gran cantidad de datos a los cuales hay que aplicarles una serie de funciones o transformaciones matemáticas con el objetivo de manipular los mismos y obtener algún tipo de conclusión sobre ellos: ya sea graficarlos para encontrar su distribución o bien predecir un valor asociado al dato.

En el ejemplo que adjuntamos a continuación aplicamos el algoritmo de KMeans++ a un set de datos que sacamos de R (utilizando un package de Julia llamado Rdatasets). Éste algoritmo resuelve el problema de Clustering, recibiendo como hiper-parámetro la cantidad de clusters que hay en los datos. Para ver eso, primero graficamos la distribución de los mismos en R2 (utilizamos PyPlot para realizar el gráfico).

Por último, incluiremos un ejemplo similar. Ésta vez resolveremos el problema de clustering a través del algoritmo DBSCAN el cual es un poco más avanzado que KMeans++, en general tiene un orden de tiempo mayor, y posee la ventaja por encima del primero que no requiere la cantidad de clusters como hiper-parámetro, sino que este número lo encuentra por sí mismo.



Estadísticas

**Index TIOBE Mayo 2018**

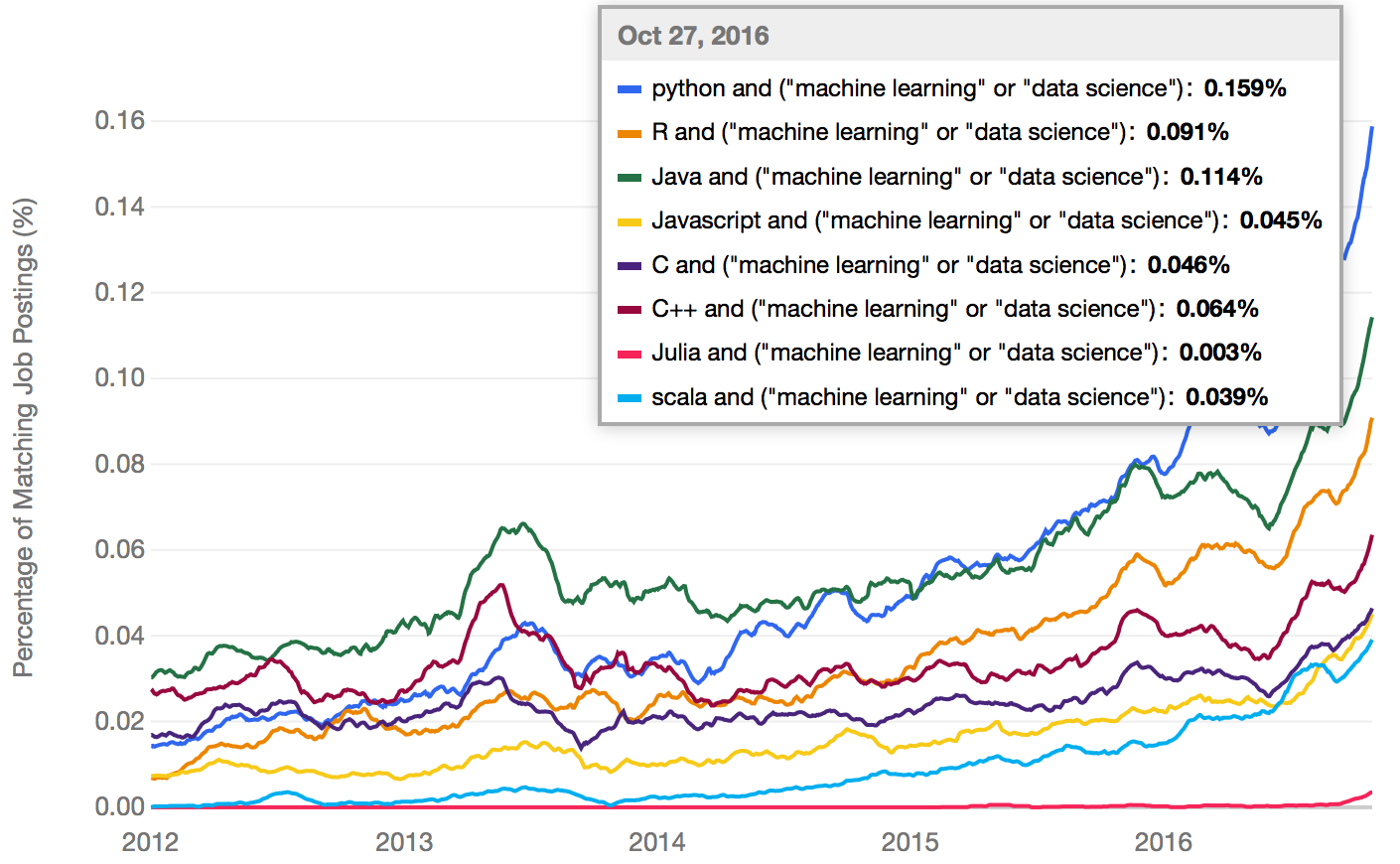


**Populridad en Githut en 2018**

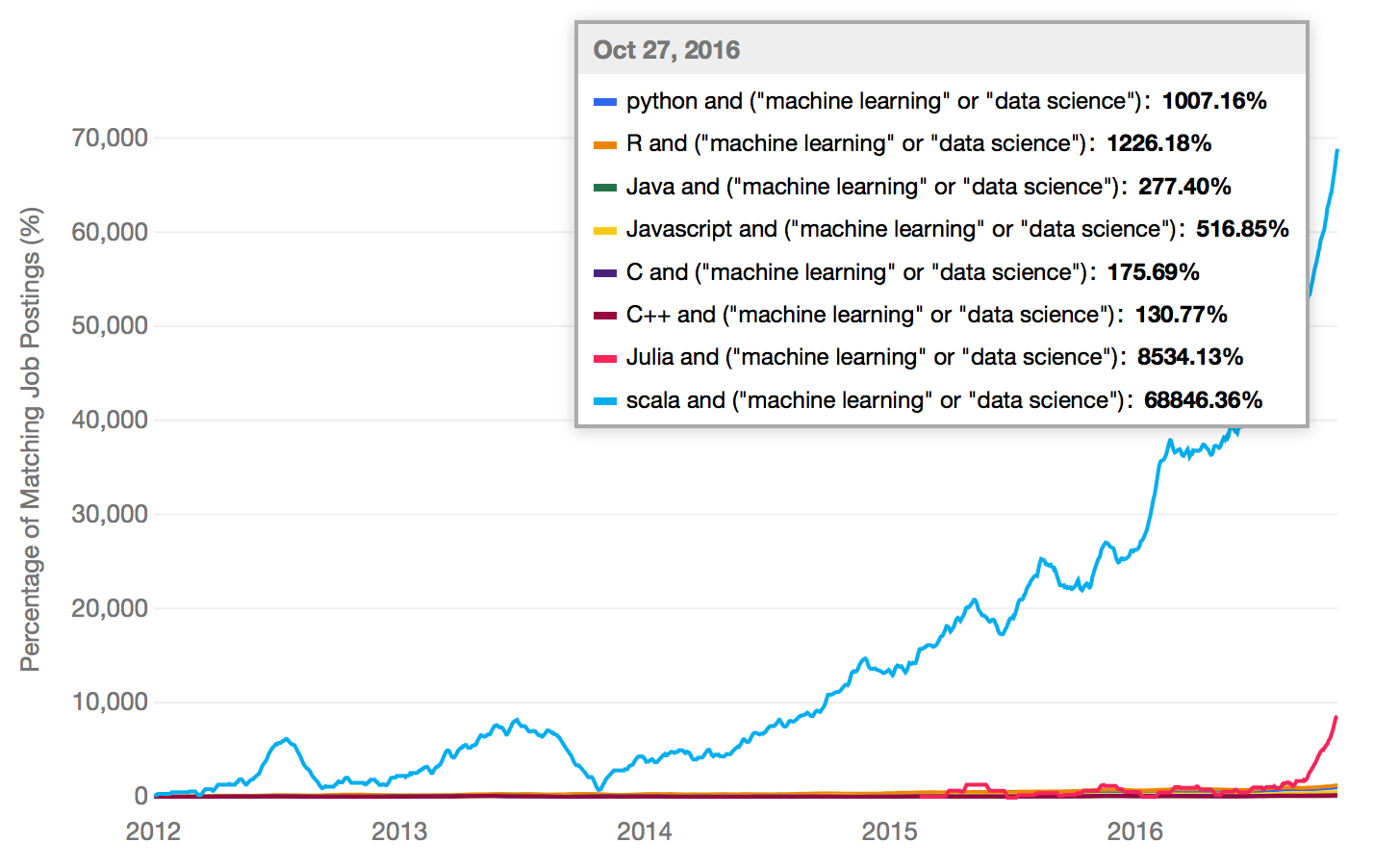


A fines del 2017 Julia subió al puesto 27 de popularidad en Githut, en la atualidad decayó al puesto 35, lo que muestra que en Github se está usando menos el lenguaje para proyectos.

**Popularidad de lenguajes en ofertas laborales en cuanto a machine learning o data science en 2016**



Julia parece no ser muy considerado para machine learning o data science pero cuando cambiamos a la popularidad relativa vemos que últimamente su uso está aumentando mucho.



Referencias

https://julialang.org/

https://juliacomputing.com

https://github.com/DataWookie/MonthOfJulia

https://docs.julialang.org

https://juliabox.com/#