



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

**Vulnerabilidad de redes de
paquetes software II**



Presentado por Daniel Alonso Báscones
en Universidad de Burgos — 2 de julio de 2023
Tutor: Carlos López Nozal



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. Carlos López Nozal, profesor del departamento de Ingeniería Informática, area de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Daniel Alonso Báscones, con DNI 71298886J, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado Vulnerabilidad de redes de paquetes software II.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 2 de julio de 2023

Vº. Bº. del Tutor:

D. Carlos López Nozal

Resumen

En prácticamente todos los proyectos software, la utilización de bibliotecas de repositorios centralizados de paquetes es universal. El motivo fundamental es disminuir los tiempos de desarrollo y coste. Debido a la transitividad de las dependencias entre paquetes, la aparición de un único defecto en el repositorio puede tener efectos extensos y difíciles de predecir en el repositorio de paquetes. Estos defectos provocan errores funcionales o problemas de rendimiento o de seguridad. El riesgo es difícil de apreciar por los desarrolladores, que solo importan explícitamente una pequeña parte de las dependencias.

En un trabajo previo, bajo la teoría de la ciencia de redes, se definió un modelo de dependencias de paquetes software a partir de un conjunto de datos público en libraries.io y generado entre 2020 y 2021. Libraries.io indexa datos de 7,352,165 paquetes de 32 gestores de paquetes. El modelo generado en la primera versión del TFG permite analizar las redes de paquetes PyPI, Maven y npm desde el punto de vista de la vulnerabilidad y de la inmunización. El objetivo de este proyecto es estudiar e implementar mediante técnicas de Webscraping y de acceso a Web API que permitan generar conjuntos de datos actualizados de redes de paquetes software: PyPI, npm, CRAN y Bioconductor. Los resultados de este trabajo son la biblioteca olivia.finder de Python y notebooks de uso, los conjuntos de datos actualizados y los nuevos (Bioconductor) y el análisis de la evolución de los sistemas de paquetes desde su publicación en libraries.io. Además, a modo de aplicación se proporciona un análisis de las dependencias transitivas de un proyecto de Github.

A partir del análisis de evolución temporal de las redes de dependencias de PyPI observamos que la red ha crecido sustancialmente. Como conclusión se recomienda que todos los trabajos basados en libraries.io repliquen sus análisis con los nuevos conjunto de datos para confirmar sus resultados.

Descriptores

Ciencia de redes, Vulnerabilidad de red, Grafo de dependencias, Repositorio de paquetes, npm, PyPI, CRAN, Bioconductor, Web scraping.

Abstract

In virtually all software projects, the use of centralized package repository libraries is universal. The fundamental reason is to reduce development time and cost. Due to the transitivity of dependencies between packages, the occurrence of a single defect in the repository can have extensive and unpredictable effects on the package repository. These defects can lead to functional errors or performance and security issues. The risk is difficult for developers to appreciate, as they only explicitly import a small portion of the dependencies.

In a previous work, based on network science theory, a software package dependency model was defined using publicly available data from libraries.io, generated between 2020 and 2021. Libraries.io indexes data from 7,352,165 packages from 32 package managers. The model generated in the initial version of the Bachelor's Thesis allows for the analysis of package networks such as PyPI, Maven, and npm from the perspective of vulnerability and immunization.

The objective of this project is to study and implement techniques such as web scraping and access to web APIs to generate updated datasets of software package networks: PyPI, npm, CRAN, and Bioconductor. The outcomes of this work include the olivia.finder library for Python and usage notebooks, the updated and new (Bioconductor) datasets, and the analysis of the evolution of package systems since their publication on libraries.io. Additionally, as an application, an analysis of the transitive dependencies of a GitHub project is provided.

Through the analysis of the temporal evolution of PyPI's dependency networks, we observe substantial growth in the network. As a conclusion, it is recommended that all studies based on libraries.io replicate their analyses with the new datasets to confirm their results.

Keywords

Network science, Network vulnerability, Dependency graph, Package repository, npm, PyPI, CRAN, Bioconductor, Web scraping.

Índice general

Índice general	iii
Índice de figuras	v
Índice de tablas	viii
1. Introducción	1
2. Objetivos del proyecto	7
2.1. Objetivo general	7
2.2. Objetivos técnicos	8
3. Conceptos teóricos	9
3.1. Repositorios de paquetes software	9
3.2. Dependencias entre paquetes	11
3.3. Gestores de paquetes	12
3.4. Ciencia de redes	13
3.5. Métricas en la ciencia de redes	14
4. Técnicas y herramientas	17
4.1. Análisis de los datos disponibles	17
4.2. Entorno de desarrollo	21
4.3. Computación en la nube: <i>Kaggle</i> y <i>Deepnote</i>	23
4.4. Sistema de control de versiones	24
4.5. Integración continua y el control de calidad	24
4.6. Persistencia de datos	25
4.7. Gestión y organización del proyecto	26

5. Aspectos relevantes del desarrollo del proyecto	29
5.1. Análisis estático de las redes: Bowtie	29
5.2. La red de dependencias de CRAN	37
5.3. La red de dependencias de Bioconductor	59
5.4. La red de dependencias de PyPI	67
6. Trabajos relacionados	93
7. Conclusiones y Líneas de trabajo futuras	97
7.1. Conclusiones	97
7.2. Líneas de trabajo futuras	98
Bibliografía	101

Índice de figuras

5.1. Comparacion de los paquetes comunes entre los dos conjuntos de datos	38
5.2. Comparacion del numero de paquetes.	38
5.3. Distribucion de grado <i>libraries.io</i>	40
5.4. Distribucion de grado <i>scraped</i>	40
5.5. Distribucion de Out degree de <i>libraries.io</i>	41
5.6. Distribucion de Out degree de <i>scraped</i>	42
5.7. Top paquetes con mayor grado de salida en <i>libraries.io</i>	42
5.8. Top paquetes con mayor grado de salida en <i>scraped</i>	43
5.9. Distribucion de dependientes.	44
5.10. Distribucion de In degree de <i>libraries.io</i>	44
5.11. Distribucion de In degree de <i>scraped</i>	45
5.12. Top paquetes con mayor grado de entrada en <i>libraries.io</i>	46
5.13. Top paquetes con mayor grado de entrada en <i>scraped</i>	46
5.14. Incremento de dependencias.	47
5.15. Distribucion de <i>PageRank</i> en <i>libraries.io</i>	48
5.16. Distribucion de <i>PageRank</i> en <i>scraped</i>	49
5.17. Top <i>PageRank</i> en <i>libraries.io</i>	49
5.18. Top <i>PageRank</i> en <i>scraped</i>	50
5.19. Top 20 <i>PageRank</i> numero de dependencias transitivas en <i>libraries.io</i>	51
5.20. Top 20 <i>PageRank</i> numero de dependencias transitivas en <i>scraped</i>	51
5.21. Top 20 <i>PageRank</i> paquetes en <i>scraped</i>	52
5.22. Top 20 <i>PageRank</i> paquetes numero de dependencias transitivas en <i>scraped</i>	52
5.23. Distribución de <i>Impact</i> en <i>libraries.io</i>	53
5.24. Distribución de <i>Impact</i> en <i>scraped</i>	53
5.25. Top paquetes con mayor <i>Impact</i> en <i>libraries.io</i>	54

5.26. Top paquetes con mayor <i>Impact</i> en scraped	54
5.27. Incremento de <i>Impact</i>	56
5.28. Top paquetes con mayor <i>Reach</i> en libraries.io	57
5.29. Incremento de <i>Reach</i>	57
5.30. Incremento de <i>Reach</i>	59
5.31. Distribucion de grado de la red de dependencias de Bioconductor	61
5.32. Distribucion de grado de la red de dependencias de Bioconductor	61
5.33. Distribucion de grado de salida de la red de dependencias de Bioconductor	62
5.34. Top 10 de los paquetes con mayor grado de salida en la red de dependencias de Bioconductor	62
5.35. Distribucion de grado de entrada de la red de dependencias de Bioconductor	63
5.36. Top 10 de los paquetes con mayor grado de entrada en la red de dependencias de Bioconductor	64
5.37. Distribucion de PageRank de la red de dependencias de Bioconductor	65
5.38. Top dependencias con mayor PageRank en la red de dependencias de Bioconductor	65
5.39. Top paquetes con mayor PageRank en la red de dependencias de Bioconductor	66
5.40. Comparacion de la cantidad de paquetes en PyPI entre los datos de libraries.io (2020) y los recolectados en este trabajo (2023).	68
5.41. Paquetes comunes y no comunes entre los datos de libraries.io (2020) y los recolectados en este trabajo (2023).	68
5.42. Popularidad de Python a lo largo del tiempo.	69
5.43. Distribucion de grado de PyPI para libraries.io.	70
5.44. Distribucion de grado de PyPI para los datos recolectados en este trabajo (2023).	70
5.45. Top de paquetes con mayor grado de salida en PyPI para libraries.io (2020).	72
5.46. Top de paquetes con mayor grado de salida en PyPI para scraped (2023).	72
5.47. Distribución de <i>Out degree</i> para libraries.io (2020)	73
5.48. Distribución de <i>Out degree</i> para scraped (2023)	73
5.49. Incremento del <i>Out degree</i> en PyPI (2010-2023)	74
5.50. Top paquetes con mayor <i>In degree</i> en libraries.io (2020)	75
5.51. Top paquetes con mayor <i>In degree</i> en scraped (2023)	75
5.52. Distribucion del <i>In degree</i> en libraries.io (2020)	76
5.53. Distribucion del <i>In degree</i> en scraped (2023)	77
5.54. Top 20 <i>PageRank</i> en libraries.io (2020)	78

5.55. Dependencias transitivas del top 20 <i>PageRank</i> en libraries.io (2020)	79
5.56. Top <i>PageRank</i> en scraped (2023)	80
5.57. Top <i>PageRank</i> paquetes en scraped (2023)	81
5.58. Distribución del impacto en la red de libraries.io (2020)	82
5.59. Distribución del impacto en la red scraped (2023)	83
5.60. Top Reach en libraries.io (2020)	86
5.61. Top Reach en scraped (2023)	87
5.62. Distribución del Reach en libraries.io (2020)	87
5.63. Distribución del Reach en scraped (2023)	88
5.64. Incremento del Reach (2020-2023)	88
5.65. Distribucion de grado del mayor componente fuertemente conexo (283 nodos) en scraped (2023)	90
5.66. Mayor componente fuertemente conexo (283 nodos) en scraped (2023)	91

Índice de tablas

5.1.	Tabla de datos para Bioconductor y CRAN	31
5.2.	Tabla de datos para PyPI	33
5.3.	Tabla de datos para NPM	35
5.4.	Comparación de paquetes en CRAN entre los datos de libraries.io y los recolectados en este trabajo	37
5.5.	Comparación de medidas de tamaño entre los dos conjuntos de datos	39
5.6.	Top 10 paquetes con mayor incremento en <i>Impact</i>	55
5.7.	Top 10 paquetes con mayor decremento en <i>Impact</i>	55
5.8.	Top 10 paquetes con mayor <i>Reach</i>	56
5.9.	Top 10 paquetes con mayor incremento en <i>Reach</i>	58
5.10.	Medidas de la red de dependencias de Bioconductor	60
5.11.	Comparación de paquetes en PyPI entre los datos de libraries.io (2020) y los recolectados en este trabajo	69
5.12.	Comparación entre paquetes obtenidos de libraries.io (2020) y scraped (2023) para la métrica <i>Impact</i>	82
5.13.	Comparación del incremento del impacto para de libraries.io (2020) y scraped (2023)	84
5.14.	Disminución del impacto en libraries.io (2020) y scraped (2023)	85
5.15.	Componentes fuertemente conexos más grandes en scraped (2023)	89

1. Introducción

Uno de los principios más importantes de la ingeniería del software es aprovechar al máximo los componentes existentes [19]. El empleo de librerías externas para disminuir los tiempos de desarrollo y costes es prácticamente universal, en todos los lenguajes y tipos de proyecto software. En una encuesta realizada por Sonartype en 2014, más de once mil arquitectos y desarrolladores revelaron que el 90 % del código de una aplicación típica en la actualidad está compuesto por componentes externos. Además, más del 80 % de los proyectos utilizan repositorios centralizados de componentes. Estos repositorios están configurados por defecto en las herramientas que usamos para instalar y administrar los paquetes de cada lenguaje. Son como almacenes gigantes que contienen componentes los cuales aportan alguna utilidad. Algunos ejemplos famosos son PyPI (Python Package Index), npm (Node Package Manager) para JavaScript (Node.js), CRAN y Bioconductor para R, CPAN (Comprehensive Perl Archive Network) y Maven Central para Java.

El problema reside en que hay veces que utilizar componentes externos tiene sus riesgos. Esos riesgos pueden ser difíciles de notar para los desarrolladores, ya que solo importan explícitamente una pequeña parte de las librerías que se incluyen en cada proyecto.

Ilustraremos la afirmación anterior con un suceso que hizo temblar npm. En marzo de 2016, un desarrollador llamado Azer Koçulu, de California, recibió un correo de unos abogados de Kik Interactive. Le pedían amablemente que cambiara el nombre de su librería llamada Kik, que había publicado en npm. Resulta que los abogados decían que la marca *Kik Messenger* les pertenecía a sus representados [7] por derechos de propiedad intelectual. Koçulu no se quedó de brazos cruzados, se negó rotundamente a cambiar el nombre. Los abogados, entonces, se pusieron en contacto con Isaac Schule-

ter, el director de npm, quien accedió a transferir el nombre. Koçulu muy decepcionado con la decisión tomó medidas, y eliminó más de 250 paquetes del repositorio que habían sido desarrollados por él.

Entre los paquetes eliminados por Koçulu había uno llamado *left-pad*, que era un simple script para justificar cadenas de texto. Pero... Muchos de los paquetes más populares de npm dependían directa o indirectamente de *left-pad*. Casi un millón de sitios web comenzaron a tener problemas para actualizar sus dependencias o lanzar nuevas versiones, e incluso gigantes como Facebook o Netflix se vieron afectados. Fue un verdadero caos.

Finalmente, npm tuvo que intervenir y, aunque generó mucha polémica, decidieron restaurar el paquete *left-pad*. Laurie Voss, el director de tecnología de npm, explicó que tomaron esta decisión sin precedentes debido a la gravedad y la amplia repercusión del problema. No lo hicieron a la ligera, la decisión se tomó pensando en la comunidad de usuarios de npm y en sus necesidades.

Después del incidente, se hizo evidente lo frágiles que son los sistemas de paquetes en los que depende una gran parte del desarrollo de software en todo el mundo. Surgieron muchas preguntas sobre el uso indiscriminado de librerías externas [3] y el modelo de gestión centralizada. Es como darse cuenta de lo mucho que dependemos de ciertas cosas y preguntarnos si estamos siendo prudentes al hacerlo. La problemática no se limita solo a la retirada voluntaria de un paquete en particular, como sucedió con *left-pad*. También puede ser causada por todo tipo de defectos en el software o ataques maliciosos que se propagan a través de la estructura de dependencias. Según un informe llamado *The state of open source security report*, las vulnerabilidades de seguridad en el software de código abierto prácticamente se duplican cada dos años, y aproximadamente el 80 % de estas vulnerabilidades se introducen a través de dependencias indirectas. Las conclusiones de este informe son preocupantes, e hicieron que el estudio de la seguridad y estabilidad de estos repositorios, que antes no había recibido mucha atención se pusiese en el punto de mira de la comunidad [1] [10] [6].

Es aquí donde entra la ciencia de redes[5] a jugar un papel importante en el mundo del software. Dado que las dependencias entre los proyectos de software, especialmente entre los paquetes de un repositorio, se pueden representar como una red o un grafo dirigido y analizar desde una perspectiva estructural, la ciencia de las redes se ha convertido en un enfoque reciente para comprender mejor la estructura y evolución [13] de los repositorios de paquetes de software. Es como mirar la forma en que todas estas piezas se conectan y se influyen mutuamente. La ciencia de redes es una disciplina que

se dedica a estudiar las redes complejas que podemos encontrar en muchos campos como la tecnología, la biología y las ciencias sociales. Cuando decimos *complejas* nos referimos al tamaño de las estructuras que se investigan y a la falta de un patrón obvio de relaciones entre sus elementos. En los modelos de ciencia de redes, representamos estos elementos como nodos o vértices, mientras que las conexiones entre ellos se simbolizan con enlaces o aristas. No es de sorprender que la ciencia de redes se base fundamentalmente en la teoría matemática de grafos, pero no es la única ciencia que interviene en este análisis. Es una materia interdisciplinaria que incorpora conocimientos de estadística, teorías de control e información, algoritmos, minería de datos y muchas otras áreas de la informática. Los principales logros de la ciencia de redes se centran en la construcción de modelos que explican la síntesis y evolución de las redes reales. También se estudia la centralidad o importancia de los nodos, es decir, qué nodos son los más cruciales dentro de una red. Además, se investiga la detección de agrupaciones significativas, como clústeres y comunidades, que revelan patrones interesantes dentro de una red. Y por si fuera poco, se exploran fenómenos de difusión, es decir, cómo se propaga la información a través de las redes.

Alexandre Decan junto a otros investigadores [8], realizaron un estudio en el que compararon el número de dependencias directas y transitivas, así como el número de componentes débilmente conectados en los repositorios CRAN, PyPI y npm. En otros estudios posteriores [14], sugieren que las estrategias actuales de versionado de paquetes pueden causar problemas de falta de robustez en los repositorios. Esta conclusión fue demostrada experimentalmente en un análisis de CRAN [9], apareciendo un patrón de comportamiento que a priori se mantiene común en los repositorios de software. Por si eso no fuera suficiente, otro estudio [26] analiza la evolución del número de dependencias y la importancia relativa de cada componente en npm. Utiliza PageRank como métrica para evaluar la centralidad de los paquetes y tienen en cuenta cómo se utilizan en proyectos de GitHub. Otro aspecto que se estudia en la ciencia de redes es la vulnerabilidad de la red. La vulnerabilidad se refiere a la incapacidad de la red para funcionar correctamente cuando se eliminan algunos nodos o enlaces importantes [17]. Para medir la vulnerabilidad, necesitamos conocer el tipo de red en particular. Por lo general, distinguimos entre casos en los que los problemas ocurren de manera aleatoria o cuando son ataques intencionales [4]. En el último caso, se supone que los atacantes utilizan información de la red para planificar estrategias que provoquen el mayor daño posible o maximicen algún objetivo que generalmente va en contra de los intereses de los actores de la red. En cuanto a los repositorios de paquetes, un estudio [26] revela

que la eliminación de un solo paquete puede afectar al 30 % de los paquetes y aplicaciones en los ecosistemas de JavaScript, Rust y Ruby. Otro estudio [10] concluye que el número de vulnerabilidades encontradas en el código fuente de los paquetes de npm ha estado aumentando constantemente desde 2012, propagándose a través de la red de dependencias.

OLIVIA [20], desarrollada por el alumno Daniel Setó Rey como parte de su Trabajo de Fin de Grado en la Universidad de Burgos y tutorizada por los profesores Carlos López Nozal y Jose Ignacio Santos Martín en 2021, es una herramienta de código abierto que se centra en la identificación y análisis de defectos en bibliotecas de software desde la perspectiva de la ciencia de redes. Estos defectos pueden provocar errores funcionales, problemas de rendimiento e incluso problemas de seguridad. Para los desarrolladores, comprender completamente el riesgo es complicado, ya que solo importan explícitamente una pequeña parte de las dependencias utilizadas en sus proyectos.

OLIVIA utiliza un enfoque basado en la *vulnerabilidad de la red de dependencias* de los paquetes de software para medir la sensibilidad del repositorio a la introducción aleatoria de defectos. Su objetivo es contribuir a la comprensión de los mecanismos de propagación de defectos en el software y estudiar estrategias factibles de protección.

OLIVIA está publicado a nivel académico[21] Despues de su desarrollo como proyecto de fin de carrera, se realizaron los esfuerzos necesarios para compartir sus resultados y contribuciones con la comunidad científica. Esta publicación permite que otros investigadores y profesionales del campo accedan a esta herramienta y se beneficien de su enfoque innovador en la identificación y análisis de vulnerabilidades en las bibliotecas de software. Además, sienta las bases para futuros avances en la comprensión de los mecanismos de propagación de defectos y la implementación de estrategias efectivas de protección en el desarrollo de software.

Una vez que tenemos el modelo de análisis definido, lo que se necesitan son los datos a analizar mediante este. La obtención de un conjunto de datos actualizado y el análisis a bajo nivel de los mismos, han aportado una actualización de los resultados relativamente obsoletos de *Libraries.io*.

La recopilación de datos sobre las dependencias de los paquetes de software es un desafío complejo que puede resultar difícil de abordar. Uno de los principales problemas radica en la falta de una única fuente confiable de información sobre estas dependencias. En muchos casos, no existe un repositorio centralizado o una base de datos completa que contenga todos los detalles necesarios. Debido a esta falta de una fuente única, recopilar

datos sobre las dependencias de los paquetes de software a menudo requiere un esfuerzo manual y exhaustivo. Los desarrolladores y analistas deben investigar y rastrear las dependencias de cada paquete individualmente, lo que puede llevar una cantidad considerable de tiempo y recursos. Además, la información sobre las dependencias de los paquetes de software puede dispersarse en diferentes fuentes, como documentación oficial, repositorios de código, foros de desarrolladores y otras fuentes en línea.

Esta dispersión puede dificultar aún más la recopilación de datos y aumentar la posibilidad de omitir o malinterpretar información relevante. El análisis de las redes de dependencias de paquetes de software es un proceso complejo debido a la naturaleza de estas redes, que pueden ser enormes y altamente interconectadas. Estas redes pueden contener miles o incluso millones de nodos y conexiones, lo que hace que el análisis manual sea prácticamente imposible.

Otro desafío asociado con la recopilación de datos es mantener la información actualizada. Las dependencias de los paquetes de software pueden cambiar con el tiempo debido a actualizaciones, nuevas versiones o cambios en los requisitos del sistema. Por lo tanto, es crucial realizar un seguimiento constante de los cambios y actualizar la información de las dependencias de manera regular para garantizar la precisión de los datos recopilados.

Para abordar estos desafíos, y como uno de los principales resultados de este TFG es ilustrar al lector de cómo poder enfrentarse a esta tarea, identificando las distintas vías de obtención de datos, y concluyendo con el desarrollo de una herramienta que facilita el proceso de obtención de los mismos desde los repositorios oficiales de los gestores de paquetes que hemos elegido como caso de estudio. En concreto nos referimos a la recopilación de datos de las dependencias en los paquetes de R (*CRAN* y *Bioconductor*), de Python *PyPI* y de Node.js *npm*.

2. Objetivos del proyecto

2.1. Objetivo general

Este estudio analiza estrategias de extracción de datos de repositorios de paquetes, considerando restricciones y características específicas, y busca publicar los datos obtenidos como referencia para futuras investigaciones.

Podemos dividir los objetivos del proyecto en estos cuatro puntos:

- Realizar un análisis de la viabilidad de las diferentes estrategias de extracción de datos de los repositorios *CRAN*, *Bioconductor*, *PyPI* y *npm* teniendo en cuenta las restricciones y particularidades de cada plataforma. Consideramos aspectos como la disponibilidad de datos, las políticas de acceso y las limitaciones técnicas para determinar la mejor manera de obtener y procesar la información requerida.
- Publicar los datos obtenidos con el propósito de que sirvan como referencia y recurso para futuras investigaciones en el campo, fomentando la colaboración en la comunidad científica, permitiendo a otros investigadores utilizarlos como base para nuevos análisis y descubrimientos.
- Realizar un análisis de las principales métricas de teoría de grafos utilizando el conjunto de datos disponible para comparar la evolución de los paquetes y evaluar el estado de los repositorios a lo largo del tiempo.
- Obtener las métricas propuestas por OLIVIA para el nuevo conjunto de datos y su comparación con los datos expuestos en el Trabajo de Fin de Grado anterior.

2.2. Objetivos técnicos

Para la consecución de los objetivos generales, se han establecido los siguientes objetivos técnicos:

- Aplicación de técnicas de *web scraping* para la extracción de datos.
- Aplicación de técnicas de acceso a datos a través de una *Web API*.
- Comprensión y aplicación de formatos de datos para almacenar persistentemente una red de dependencias.
- Medición y análisis de redes de dependencias mediante la aplicación de conceptos teóricos de la biblioteca *networkx*.
- Acceso a las dependencias de un repositorio en GitHub utilizando la funcionalidad de *grafo de dependencias* de GitHub.

3. Conceptos teóricos

3.1. Repositorios de paquetes software

En el desarrollo de software, los *repositorios de paquetes* juegan un papel crucial al proporcionar un entorno centralizado donde los desarrolladores pueden acceder, compartir y distribuir bibliotecas de código predefinidas. Estos repositorios están diseñados específicamente para diferentes plataformas y lenguajes de programación, brindando a los desarrolladores un acceso conveniente a una amplia gama de recursos.

A lo largo del tiempo, han surgido numerosos repositorios de paquetes de software para diversas plataformas y lenguajes. Por ejemplo, en el campo de la bioinformática, destaca *Bioconductor* como un importante repositorio que se enfoca en paquetes y herramientas para el análisis de datos genómicos. En el ecosistema de Python, *PyPI* (Python Package Index) es un repositorio central que alberga una gran cantidad de paquetes para una amplia variedad de aplicaciones y bibliotecas.

En el panorama actual, los repositorios de paquetes de software siguen siendo vitales para la comunidad de desarrollo. Brindan a los desarrolladores un acceso rápido y sencillo a una amplia gama de funcionalidades y bibliotecas de código predefinidas, lo que les permite acelerar el desarrollo de aplicaciones y proyectos. Además, estos repositorios fomentan la colaboración y el intercambio de código entre los desarrolladores, promoviendo un entorno de desarrollo más dinámico y eficiente.

Un *repositorio de paquetes* es básicamente un almacén de software que tiene como objetivo distribuir librerías de componentes reutilizables. El término paquete se refiere al artefacto adecuado para facilitar esta distribución, incluyendo la descarga, verificación e instalación de los componentes

por parte de los usuarios del repositorio. Normalmente, un paquete está compuesto por código fuente y/o archivos binarios, junto con metadatos que incluyen directrices de compilación, listas de requisitos, parámetros de configuración, referencias a la documentación y detalles de la licencia, todo ello contenido en un único archivo comprimido.

Existen dos tipos principales de repositorios de paquetes: los *repositorios de nivel de sistema*, integrados en los sistemas operativos, y los *repositorios de nivel de aplicación*, integrados en los entornos de desarrollo de diferentes lenguajes de programación. En ambos casos, suele haber un sistema oficial o muy popular desde donde se realizan la mayoría de las descargas, lo que se conoce como *repositorios centralizados*.

En los últimos años, los repositorios de nivel de aplicación han experimentado un crecimiento exponencial en términos de número de paquetes y cantidad de descargas. Por ejemplo, el repositorio oficial de Node.js, llamado *npm*, es actualmente el más grande en cuanto al número de componentes publicados.

Para facilitar la interacción con los repositorios, existen herramientas llamadas *gestores de paquetes*, que permiten a los desarrolladores automatizar la descarga e instalación recursiva de componentes.

El software utilizado para desplegar y administrar los repositorios de paquetes se conoce como *gestor de repositorios*.

En el ámbito del software, existen diferentes tipos de dependencias que juegan un papel diferenciado. Por ejemplo, en el repositorio CRAN, utilizado para paquetes de R, encontramos distintas categorías de dependencias, como *depends*, *imports*, *enhances* y *suggests*. Cada una de ellas indica el nivel de interdependencia entre los paquetes, desde las dependencias necesarias para que un paquete funcione correctamente (*depends*), hasta las recomendadas para mejorar su funcionalidad (*suggests*).

Además, en lenguajes como Python, se distinguen las dependencias de *runtime* y las dependencias de *desarrollo*. Las primeras se refieren a los componentes necesarios para ejecutar el programa, como bibliotecas y módulos externos, mientras que las segundas son herramientas y bibliotecas utilizadas durante el desarrollo del software, como entornos de prueba y sistemas de construcción.

También se encuentran las *dependencias foráneas*, las cuales pueden no estar ubicadas en el mismo repositorio que el paquete en cuestión. Estas dependencias pueden incluir componentes binarios o estar escritas en

otros lenguajes de programación. Su presencia implica la necesidad de gestionar y resolver estas dependencias externas para garantizar el correcto funcionamiento y compatibilidad del software.

3.2. Dependencias entre paquetes

En el contexto de un proyecto de software, las *dependencias externas* se refieren a las conexiones con productos provenientes de actividades realizadas por agentes externos al equipo del proyecto. Estos productos pueden ser requisitos, recursos o controles, como requisitos de clientes, personal, hardware, software, procesos y estructuras de gestión. Las dependencias externas están fuera del control directo del equipo del proyecto y, si no funcionan según lo esperado, pueden comprometer el éxito del proyecto.

En el desarrollo de paquetes de software, al igual que en cualquier otro proyecto de software, los desarrolladores reutilizan componentes distribuidos en otros paquetes, lo que genera dependencias externas de software. Estas dependencias pueden ser *directas*, cuando el paquete A invoca directamente funciones de componentes del paquete B, o *transitivas*, cuando el paquete A invoca al paquete B, que a su vez utiliza el paquete C. Las dependencias transitivas pueden dar lugar a largas cadenas de requisitos encadenados, que se aplican a distintas fases del ciclo de desarrollo de software, como la implementación/construcción, la prueba y el despliegue. Los gestores de paquetes generalmente se encargan de resolver automáticamente todas estas dependencias según sea necesario, a través de las herramientas de los entornos de desarrollo, integración continua y despliegue.

Por lo tanto, un defecto en un paquete alojado en un repositorio puede propagarse a otros paquetes dependientes, tanto directa como transitivamente, y en última instancia, afectar a todos los proyectos que utilicen dicho paquete. La propagación de este defecto está vinculada a la transferencia de la versión dañada del paquete desde el repositorio. En otras palabras, el software que utiliza los componentes almacenados en los entornos de ejecución locales no se verá afectado hasta que se actualice la dependencia. Esto puede ocurrir en diferentes momentos, por lo que la propagación a cada proyecto dependiente puede tener un retraso variable:

- En el caso de software que incluye componentes compilados, el impacto se producirá cuando se realice una compilación después de descargar alguno de los paquetes comprometidos desde el repositorio.

- En el caso de software interpretado, con compilación JIT o con enlace dinámico de librerías, el impacto se producirá después de descargar alguno de los paquetes comprometidos desde el repositorio.

Estas descargas suelen ser realizadas por el gestor de paquetes y pueden deberse a diferentes motivos, como la creación de un nuevo entorno de desarrollo, ejecución o pruebas (incluyendo entornos de integración continua), o la decisión de incluir versiones actualizadas de las dependencias en los requisitos del proyecto, ya sea por motivos funcionales o de seguridad.

3.3. Gestores de paquetes

Los *gestores de paquetes de software* desempeñan un papel fundamental al proporcionarnos herramientas y funcionalidades para administrar la instalación, actualización y eliminación de bibliotecas y dependencias en nuestros proyectos. Estos gestores están diseñados específicamente para diferentes plataformas y lenguajes de programación, brindando a los desarrolladores una forma eficiente de gestionar y distribuir el código.

Entre los gestores de paquetes más destacados, encontramos *pip* en el ecosistema de Python, el cual nos permite instalar y administrar de manera sencilla las bibliotecas necesarias para nuestros proyectos en Python. Por otro lado, *mvn* (Maven) es ampliamente utilizado en el ámbito de Java para gestionar las dependencias y configuraciones de proyectos. Cada gestor de paquetes tiene su propia sintaxis y funcionalidades específicas, pero todos comparten el objetivo común de simplificar la gestión de bibliotecas y garantizar la resolución de dependencias.

En el panorama actual, los gestores de paquetes de software continúan desempeñando un papel crucial en el desarrollo de software. Proporcionan a los desarrolladores una forma conveniente y eficiente de administrar las bibliotecas y dependencias necesarias para sus proyectos, permitiéndoles enfocarse en la implementación de funcionalidades sin preocuparse por la instalación manual y la gestión de dependencias.

Además, estos gestores fomentan la reutilización de código y la colaboración dentro de la comunidad de desarrolladores, al facilitar el intercambio y la distribución de bibliotecas y proyectos. También simplifican la tarea de mantener actualizadas las dependencias, garantizando que los proyectos estén siempre al día y protegidos contra vulnerabilidades conocidas.

3.4. Ciencia de redes

La teoría de grafos es un campo fundamental en las matemáticas discretas y la ciencia de la computación, que estudia las propiedades y relaciones de los grafos. Un *grafo* es una estructura compuesta por un conjunto de *nodos* o *nodos*, conectados entre sí por *enlaces* llamados *enlaces*. Estos grafos pueden representar una amplia variedad de situaciones y fenómenos, desde redes de comunicación y relaciones sociales hasta sistemas de transporte y circuitos eléctricos.

La teoría de grafos se enfoca en el análisis y estudio de las propiedades estructurales y combinatorias de los grafos, así como en el desarrollo de algoritmos eficientes para resolver problemas relacionados. Se exploran conceptos fundamentales como la conectividad, los ciclos, los caminos más cortos, los flujos y cortes mínimos, entre otros.

Los grafos encuentran aplicaciones en numerosos campos, incluyendo la optimización de rutas en logística, la modelización de redes sociales, la planificación de proyectos, la programación lineal y la criptografía, por mencionar solo algunos. La teoría de grafos proporciona herramientas conceptuales y metodológicas para analizar y resolver problemas complejos en estos y otros dominios.

Grafos dirigidos y no dirigidos

En la teoría de grafos, existen dos tipos principales de grafos: los *grafos dirigidos* y los *grafos no dirigidos*. Estas diferencias se refieren a la forma en que se establecen las conexiones entre los nodos del grafo.

Un *grafo no dirigido* es aquel en el que las enlaces, que representan las conexiones entre los nodos, no tienen una dirección asociada. En otras palabras, la relación entre dos nodos es simétrica, lo que significa que si el nodo A está conectado con el nodo B, entonces el nodo B también está conectado con el nodo A. En este tipo de grafo, la comunicación o la relación entre los nodos se considera *bidireccional*. Un ejemplo común de un grafo no dirigido es una red social, donde los nodos representan personas y las enlaces representan amistades. La conexión entre dos personas en una red social no depende de la dirección.

Por otro lado, un *grafo dirigido* es aquel en el que las enlaces tienen una dirección asociada. Esto significa que la relación entre dos nodos puede ser asimétrica, es decir, el nodo A puede estar conectado con el nodo B, pero no necesariamente el nodo B está conectado con el nodo A. En este tipo de grafo,

la comunicación o la relación entre los nodos se considera *unidireccional*. Un ejemplo común de un grafo dirigido es una red de transporte, donde los nodos representan ubicaciones y las enlaces representan las rutas o los caminos que van en una dirección específica.

3.5. Métricas en la ciencia de redes

En general, las *métricas* son medidas o indicadores cuantitativos que se utilizan para evaluar y cuantificar diferentes aspectos de un objeto, sistema o fenómeno. Proporcionan una forma objetiva de medir y comparar características específicas, permitiendo realizar análisis, tomar decisiones y realizar mejoras basadas en datos concretos.

En el contexto de la teoría de grafos, las métricas se utilizan para analizar y caracterizar diferentes propiedades y aspectos de los grafos. Estas métricas ofrecen medidas cuantitativas que describen la estructura, la conectividad, la eficiencia y otros atributos relevantes de un grafo.

Grado

El *grado* de un nodo se refiere al número de enlaces que están conectados a ese nodo en particular. El grado de un nodo es una métrica fundamental para comprender la conectividad y la importancia relativa de los nodos dentro de un grafo. En un grafo no dirigido, el grado de un nodo se calcula contando el número total de enlaces incidentes a ese nodo. En un grafo dirigido, se distingue entre el grado de entrada o *In degree* (número de enlaces entrantes) y el grado de salida o *Out degree* (número de enlaces salientes) de un nodo.

El grado de un nodo puede proporcionar información valiosa sobre la estructura y las propiedades del grafo en general. Por ejemplo, los nodos con un grado alto suelen desempeñar un papel central en la comunicación y la transferencia de información dentro de un grafo. También puede revelar la existencia de nodos aislados (grado cero) o nodos de grado bajo, que pueden tener implicaciones en la conectividad y la eficiencia de un grafo.

Además, el análisis de la distribución de los grados en un grafo puede revelar patrones interesantes, como la presencia de hubs o nodos altamente conectados que desempeñan un papel clave en la red. Estudiar el grado de los nodos y su distribución puede ser útil para comprender la robustez, la centralidad y otras propiedades estructurales de un grafo.

Centralidad

La *centralidad* es una medida utilizada en la teoría de grafos para identificar los nodos más importantes o influyentes dentro de un grafo. Se basa en la idea de que algunos nodos tienen un papel más destacado en la transmisión de información o la propagación de influencias en una red.

Existen diferentes métricas de centralidad que se utilizan para evaluar la importancia de los nodos en función de diferentes criterios. Dos de las métricas de centralidad más comunes son la centralidad de intermediación y la centralidad de cercanía.

La *centralidad de intermediación* (*betweenness centrality*) mide la importancia de un nodo en función de la cantidad de veces que ese nodo se encuentra en el camino más corto entre otros pares de nodos en el grafo. Un nodo con una alta centralidad de intermediación actúa como un puente o intermediario entre otros nodos, desempeñando un papel crucial en la comunicación y el flujo de información dentro de la red.

Por otro lado, la *centralidad de cercanía* (*closeness centrality*) se refiere a la proximidad de un nodo con respecto a todos los demás nodos en el grafo. Un nodo con una alta centralidad de cercanía está más cerca en términos de distancia geodésica de todos los demás nodos, lo que implica que puede acceder rápidamente a la información y transmitirla eficientemente a otros nodos en la red.

Hacemos una mención especial a *PageRank* por su importancia en estos últimos años. *PageRank* es un algoritmo utilizado como medida de centralidad en la teoría de grafos. Fue desarrollado por Google, como parte de su motor de búsqueda original.

PageRank se basa en el concepto de que una página web es importante si es enlazada por otras páginas importantes. Considera los enlaces como votos de confianza, donde los enlaces provenientes de páginas con mayor autoridad tienen un peso mayor. Además, el algoritmo tiene en cuenta la cantidad total de enlaces que apuntan a una página y la importancia de esas páginas de origen.

El nombre "*PageRank*" se deriva del apellido de su inventor, pero también hace referencia a la noción de *ranking* o clasificación de páginas web. Page y Brin implementaron el algoritmo *PageRank* en el motor de búsqueda de Google, que se lanzó en 1998. Fue una de las innovaciones clave que distinguió a Google de otros motores de búsqueda de la época, ya que permitía generar resultados más relevantes y de mayor calidad[15].

Con el tiempo, PageRank se convirtió en una medida ampliamente utilizada para evaluar la importancia de los nodos en diversos tipos de redes, no solo en la web. Ha encontrado aplicaciones en áreas como las redes sociales, la biología de sistemas, la epidemiología y la física de redes, entre otras.

4. Técnicas y herramientas

4.1. Análisis de los datos disponibles

Para llevar a cabo el análisis de las redes de dependencias de paquetes de software, contamos con datos proporcionados por *libraries.io*[12] en formato CSV. Estos datos nos permiten extraer la red de dependencias de los principales repositorios de paquetes que vamos a estudiar en este trabajo (*CRAN*, *PyPI*, *npm*)¹. Sin embargo, es importante destacar que estos datos no están actualizados a la fecha actual debido a la elevada evolución de los cambios en los proyectos de software. Por lo tanto, los análisis que realicemos con estos datos no reflejarán la situación actual y actualizada de la red de dependencias.

A pesar de esta limitación, contamos con un conjunto de datos que abarca un gran número de repositorios y un histórico de las dependencias de los paquetes para sus distintas versiones. Esto nos permite realizar análisis retrospectivos y estudiar la evolución de las dependencias a lo largo del tiempo.

En conclusión, es evidente la necesidad de obtener un nuevo conjunto de datos que nos brinde un estudio más preciso y actualizado de las relaciones de paquetes en los repositorios. Dado que tanto el conjunto de datos de *libraries.io* como su API tiene ciertas limitaciones, se ha considerado como una opción secundaria dando preferencia a otras técnicas de extracción de datos como el *web scraping*.

¹Algunos repositorios incluidos en los conjuntos de datos son: npm, Maven, PyPI, CRAN, Rubygems, Packagist, NuGet

Web Scraping

Para obtener un conjunto de datos actualizado, se plantea la posibilidad de realizar un *web scraping* de los repositorios de paquetes de software. El *web scraping* es una técnica que consiste en extraer información de sitios web de manera automatizada. Esta técnica nos permite obtener datos de los repositorios de paquetes de software y construir un conjunto de datos actualizado.

Sin embargo, el *web scraping* presenta ciertas limitaciones que dificultan su uso para la extracción de datos. A continuación, se presentan las principales limitaciones del *web scraping*:

- **Estructura de los sitios web:** Los sitios web no están diseñados para ser analizados por máquinas, sino para ser visualizados por humanos. Por lo tanto, la estructura de los sitios web puede cambiar con frecuencia, lo que dificulta el proceso de extracción de datos.
- **Detección de bots:** Algunos sitios web pueden detectar el uso de bots y bloquear las solicitudes. Esto puede ocurrir si se realizan demasiadas solicitudes en un período de tiempo corto.
- **Limitaciones de velocidad:** Algunos sitios web pueden limitar la velocidad de las solicitudes para evitar una carga excesiva en los servidores. Esto puede provocar que el proceso de extracción de datos sea más lento y menos eficiente.

Es importante tener en cuenta estas limitaciones al utilizar el *web scraping* para asegurar un uso adecuado y obtener los datos necesarios.

El web scraping es una técnica que puede aparecer combinada con otras técnicas de extracción de datos, como la API de *libraries.io* o las APIs propias de los gestores de paquetes.

Exploración de los repositorios de paquetes

La exploración de los repositorios de paquetes desempeña un papel fundamental en nuestro estudio. A continuación, se presentan los puntos básicos a tener en cuenta en esta temática y en los que se centra el análisis:

- **Diversidad de repositorios:** Existe una amplia gama de repositorios de paquetes, cada uno especializado en un lenguaje de programación o

plataforma específica. Es esencial conocer la variedad de repositorios relevantes para nuestra área de interés, como npm para JavaScript, PyPI para Python o Maven para Java. Cada repositorio tiene su propia comunidad, conjunto de reglas y mejores prácticas.

- **Estructura y metadatos de los paquetes:** Cada paquete de software en un repositorio está acompañado de metadatos que proporcionan información importante. Estos metadatos incluyen el nombre del paquete, la versión, la descripción, el autor, la licencia, las dependencias y más. Estos son los datos relevantes para nuestra investigación y en los que nos centraremos durante la extracción².
- **Versionado y mantenimiento:** Los repositorios de paquetes suelen gestionar diferentes versiones de un paquete, cada una con sus propias mejoras, correcciones de errores y posibles cambios en las dependencias. El seguimiento y análisis de los patrones de versionado y las prácticas de mantenimiento son aspectos esenciales para comprender la evolución de los paquetes a lo largo del tiempo³.

API de libraries.io

Libraries.io proporciona una API para acceder a los datos de los repositorios de paquetes de software. Esta API nos permite obtener información sobre los paquetes y sus dependencias, así como también información sobre los repositorios y sus características.

La API de *libraries.io* es una herramienta muy útil para obtener información sobre los paquetes y repositorios de software. Sin embargo, presenta algunas limitaciones que dificultan su uso para la extracción de datos. A continuación, se presentan las principales limitaciones de la API:

- **Autenticación:** Para realizar cualquier solicitud a la API, es necesario incluir el parámetro *api_key* que se obtiene desde la página de tu cuenta. Esto implica un registro en el servicio.
- **Distintas versiones del servicio:** Existen diferentes versiones del servicio, siendo la versión gratuita la que ofrece características básicas, mientras que las características adicionales están disponibles a través de planes de pago.

²Los metadatos varían entre los diferentes repositorios y pueden contener información adicional específica del repositorio en cuestión.

³El versionado de paquetes sigue diferentes convenciones dependiendo del repositorio y las prácticas adoptadas por los desarrolladores del software.

- **Límite de velocidad:** Todas las solicitudes están sujetas a un límite de velocidad (de 60 solicitudes por minuto en la versión gratuita) basado en tu *api_key*. Si se realizan más solicitudes dentro de ese período de tiempo, se recibirá una respuesta de error 429. Este límite está diseñado para garantizar un uso equitativo de los recursos y evitar una carga excesiva en los servidores.
- **Paginación:** Algunas solicitudes pueden devolver múltiples resultados, y para manejar estos casos, se puede utilizar la paginación. Se pueden utilizar los parámetros de consulta *page* y *per_page* para controlar la cantidad de resultados por página. El valor predeterminado de *page* es 1 y el valor predeterminado de *per_page* es 30 resultados por página. Esto podría implicar resultados truncados, lo cual no es deseable.

NetworkX

La librería *NetworkX*, implementada en el lenguaje de programación *Python*, se presenta como una poderosa herramienta para el análisis de redes. Su notable versatilidad y amplio conjunto de funcionalidades la han consolidado como una elección destacada en diversos ámbitos, que abarcan desde la ciencia de datos y la investigación científica hasta la ingeniería y la sociología.

La librería *NetworkX* proporciona una interfaz flexible e intuitiva que permite la creación, manipulación y visualización de redes. Estas redes, que pueden ser representadas como grafos dirigidos o no dirigidos, contemplan nodos como entidades y enlaces o aristas que denotan las relaciones existentes entre ellas.

Además de las operaciones fundamentales para la construcción y manipulación de grafos, *NetworkX* se destaca por ofrecer una amplia gama de algoritmos y métricas específicamente diseñados para el análisis de redes. Estos algoritmos permiten realizar tareas tales como la detección de comunidades, la evaluación de la centralidad de los nodos, la búsqueda de caminos más cortos y la identificación de estructuras relevantes dentro de la red.

La capacidad de visualización constituye otro aspecto sobresaliente de *NetworkX*, ya que proporciona diversas alternativas para representar gráficamente las redes. Estas opciones visuales favorecen la comprensión y comunicación de los resultados obtenidos a partir del análisis de redes, permitiendo además la personalización de atributos visuales, como el color y tamaño de los nodos, con el fin de resaltar características particulares de la red en cuestión.

4.2. Entorno de desarrollo

Se ha trabajado en un sistema operativo *Ubuntu* y utilizando *Visual Studio Code* (VSCode) como entorno de desarrollo integrado (IDE) preferido⁴. VSCode se destaca por su versatilidad y extensibilidad, lo que me permite personalizarlo y adaptarlo a mis necesidades específicas. Su amplia selección de extensiones proporcionan herramientas adicionales y funcionalidades especializadas que enriquecen la experiencia de programación.

El lenguaje de programación elegido es *Python*, reconocido por su facilidad de uso y amplia gama de bibliotecas y frameworks disponibles. Sin embargo, también considero esencial el uso de *Bash*, un intérprete de comandos de Unix, para realizar diversas tareas y automatizaciones en el entorno de desarrollo.

Para realizar el proyecto,uento con un ordenador portátil de gama media equipado con un procesador *Intel® Core™ i5-11400H* y 16 GB de RAM. Estas especificaciones brindan un rendimiento adecuado para el desarrollo y la ejecución del software que acostumbro a usar⁵. Sin embargo, en algunos casos ha habido incidentes debido al elevado consumo de memoria que requiere el procesamiento de la masiva cantidad de datos a la que nos hemos enfrentado.

A continuación, se presentan algunas bibliotecas y herramientas utilizadas en el desarrollo de proyectos de software:

Pandas

Pandas es una biblioteca de análisis de datos de alto rendimiento que proporciona estructuras de datos y herramientas para manipular y analizar conjuntos de datos complejos.

tqdm

tqdm es una biblioteca que agrega una barra de progreso elegante y visual a los bucles iterativos, lo que facilita el seguimiento del progreso de las operaciones en tiempo real.

⁴La elección del sistema operativo y el IDE depende de las preferencias personales del autor

⁵Aunque el rendimiento puede variar dependiendo de los requisitos específicos del proyecto

Requests

Requests es una biblioteca que simplifica el manejo de solicitudes HTTP, permitiendo realizar peticiones a servidores web y recibir respuestas de manera sencilla.

BeautifulSoup4

BeautifulSoup4 es una biblioteca utilizada para extraer información de páginas web y realizar el análisis de datos web. Facilita la extracción de datos estructurados y no estructurados mediante técnicas de web scraping.

Selenium

Selenium es una biblioteca que automatiza la interacción con navegadores web, lo que permite realizar pruebas de aplicaciones web o realizar acciones específicas en páginas web de forma programática.

Networkx

Networkx es una biblioteca para el análisis de redes y grafos. Proporciona herramientas para la creación, manipulación y estudio de estructuras de redes complejas.

Matplotlib

Matplotlib es una biblioteca de visualización de datos en 2D que permite crear gráficos y visualizaciones de datos de alta calidad.

Pybraries

Pybraries es una biblioteca que actúa como un wrapper del API de libraries.io para Python.

Typing_extensions

Typing_extensions es una extensión del módulo typing de Python que proporciona funcionalidades adicionales para anotaciones de tipos en tiempo de ejecución.

pdoc

pdoc es una biblioteca que permite generar documentación automática a partir de los archivos de código fuente.

4.3. Computación en la nube: *Kaggle* y *Deepnote*

Jupyter Notebooks se ha convertido en una herramienta fundamental en el ámbito de la ciencia de datos y la programación interactiva. Estos notebooks permiten combinar código, texto explicativo y resultados visuales en un solo documento, lo que facilita la comunicación y colaboración en proyectos de análisis de datos. Los notebooks se ejecutan en un entorno interactivo, lo que permite explorar y experimentar con el código de manera iterativa, lo que resulta especialmente útil en tareas de análisis exploratorio de datos.

En cuanto a la computación en la nube, ha desempeñado un papel clave en el desarrollo de este TFG. Plataformas como *Kaggle* o *Deepnote* proporcionan servicios de notebooks basados en la nube, lo que significa que los usuarios pueden acceder a un entorno de desarrollo completo sin tener que preocuparse por configurar y mantener su propia infraestructura. Esto es especialmente beneficioso en proyectos que requieren una gran cantidad de recursos computacionales, como el procesamiento de grandes volúmenes de datos.

Además, la computación en la nube ha ayudado a reducir los costos asociados con la obtención y procesamiento de datos. La obtención de datos puede requerir tiempo, memoria y almacenamiento significativos, lo que puede ser costoso en términos de recursos locales. Al aprovechar la computación en la nube podemos acceder a recursos escalables y flexibles según sea necesario, lo que nos permite realizar análisis más eficientes y a gran escala sin incurrir en costos excesivos.⁶

⁶La escalabilidad y flexibilidad de los recursos en la computación en la nube se refiere a la capacidad de aumentar o disminuir la capacidad de cómputo y almacenamiento según las necesidades del proyecto, lo que permite un uso más eficiente de los recursos y un mejor control de costos.

4.4. Sistema de control de versiones

GitHub ha desempeñado un papel fundamental como plataforma de control de versiones Git en el ámbito del desarrollo de software colaborativo de código abierto. Como un estándar reconocido internacionalmente, GitHub, adquirido por *Microsoft*, ha proporcionado a los desarrolladores una infraestructura sólida para la gestión de proyectos. Además de su funcionalidad de repositorio Git público, GitHub ofrece herramientas integrales para el seguimiento y control de eventos relacionados con el desarrollo, lo que facilita la colaboración eficiente y transparente entre los miembros del equipo. Esta plataforma ha fomentado el desarrollo comunitario, impulsando la creación y mejora de proyectos de software en un entorno abierto y accesible para la comunidad global de desarrolladores.

GitHub, como plataforma de control de versiones basada en Git, permite a los desarrolladores almacenar y compartir sus repositorios de código, facilitando la colaboración y la contribución de múltiples personas a un proyecto. Además, ofrece herramientas como problemas, solicitudes de extracción y seguimiento de errores que permiten una comunicación efectiva entre los miembros del equipo y facilitan la gestión y resolución de problemas en el proceso de desarrollo de software.

El uso de GitHub ha fomentado el desarrollo comunitario y la creación de proyectos de software de calidad en un entorno colaborativo y transparente. Los desarrolladores pueden contribuir a proyectos existentes, realizar mejoras y correcciones de errores, y beneficiarse de la retroalimentación y la experiencia de otros miembros de la comunidad global de desarrolladores. Además, GitHub facilita la visibilidad y la accesibilidad de los proyectos, lo que permite a otros descubrir, aprender y utilizar el software desarrollado por la comunidad.

4.5. Integración continua y el control de calidad

La *integración continua* y el *control de calidad* desempeñan un papel crucial en el desarrollo de software. Para garantizar la calidad y la consistencia del proyecto, se ha utilizado *SonarCloud*⁷ como herramienta de control de calidad. Esta herramienta se integra con GitHub, lo que permite realizar un

⁷SonarCloud es una herramienta de control de calidad que proporciona análisis estático de código para identificar problemas y mejorar la calidad del código.

análisis automatizado de la calidad del código en cada commit. SonarCloud evalúa el código fuente en función de los estándares de calidad predefinidos y proporciona información detallada sobre posibles problemas, vulnerabilidades o malas prácticas. Esta integración continua de control de calidad asegura que el proyecto cumpla con los criterios de calidad deseados y permite abordar los problemas de manera oportuna.⁸

Además, se ha empleado *GitHub Pages* como una plataforma para alojar la documentación del código fuente de la biblioteca generada. GitHub Pages permite crear un sitio web estático que sirve como una fuente centralizada de información para los usuarios y desarrolladores del proyecto. Al alojar la documentación en GitHub Pages, se facilita el acceso y la navegación a través de la documentación, lo que mejora la usabilidad y la visibilidad del proyecto. Esta práctica de utilizar GitHub Pages para la documentación garantiza que la información esté siempre actualizada y disponible para todos los interesados en el proyecto.

4.6. Persistencia de datos

Se ha decidido seguir la metodología establecida en el Trabajo de Fin de Grado anterior, donde se emplean archivos CSV para almacenar los conjuntos de datos generados. Estos archivos CSV ofrecen una estructura tabular que permite representar de manera eficiente la lista de enlaces de paquetes y sus dependencias.⁹

Además de los archivos CSV, en algunos casos se ha optado por utilizar objetos serializados para el almacenamiento de datos. Esta elección se basa en la facilidad que proporcionan los objetos serializados para ser guardados y cargados en los entornos de desarrollo, como los Jupyter Notebooks utilizados en el proyecto. Al serializar los objetos, se logra una representación compacta que puede ser almacenada en archivos y posteriormente restaurada sin perder la integridad de los datos.¹⁰

Sin embargo, el gran volumen de los datos ha planteado desafíos en cuanto a su almacenamiento. El volumen de los datos generados ha requerido el

⁸El control de calidad se refiere al conjunto de procesos y técnicas utilizados para asegurar la calidad del software.

⁹CSV (*Comma-Separated Values*) es un formato de archivo que utiliza comas para separar los valores en una estructura tabular. Es ampliamente utilizado para el intercambio de datos en aplicaciones que requieren una estructura tabular sencilla.

¹⁰La serialización es el proceso de convertir un objeto en una secuencia de bytes que puede ser almacenada o transmitida, y posteriormente restaurada para obtener el objeto original. Esto facilita la persistencia de datos complejos en entornos de programación.

empleo de técnicas de compresión y división en *lotes* para asegurar su conservación eficiente. Mediante la compresión¹¹, se reduce el tamaño de los archivos de datos sin perder su contenido, lo que permite ahorrar espacio de almacenamiento. Por otro lado, la división en lotes consiste en dividir los datos en conjuntos más pequeños, lo cual facilita su manejo y procesamiento en entornos con recursos limitados.

4.7. Gestión y organización del proyecto

Inicialmente, se establecieron reuniones presenciales quincenales para discutir los objetivos de los *sprints* propuestos. A medida que nos acercábamos a la etapa final del proyecto, se optó por realizar reuniones semanales para una mayor agilidad en la toma de decisiones. Sin embargo, debido a la naturaleza del proyecto, gestionar adecuadamente los sprints ha sido un desafío, ya que en ocasiones fue necesario replantear la forma en que estábamos abordando las tareas e incluso retroceder para solucionar problemas que surgieron durante el proceso.¹²

Se ha utilizado *Microsoft Teams* como herramienta para facilitar las reuniones de forma remota, lo que permitió una comunicación efectiva y una colaboración fluida entre los miembros del equipo. Esta plataforma proporcionó un espacio para compartir documentos, discutir ideas y mantener un seguimiento de las tareas asignadas.

A lo largo del proyecto, se pueden distinguir varias etapas. En primer lugar, hubo una fase de toma de contacto, donde se adquirió un conocimiento inicial sobre los objetivos y el alcance del Trabajo de Fin de Grado. A continuación, se llevó a cabo una fase de investigación y aprendizaje, donde se profundizó en los conceptos teóricos de la ciencia de redes, aprovechando los conocimientos adquiridos en asignaturas como *Nuevas Tecnologías*.¹³

Otra etapa clave fue el estudio de estrategias para la obtención de datos. Dado que había diferentes fuentes disponibles, como archivos CSV, sitios web y APIs, se exploraron y seleccionaron las mejores opciones para obtener

¹¹La compresión de datos es el proceso de reducir el tamaño de un archivo o conjunto de datos sin perder su contenido o información. Existen diferentes algoritmos de compresión que se utilizan para lograr este objetivo.

¹²Un sprint es un período de tiempo durante el cual se realiza un conjunto de tareas o actividades dentro de un proyecto ágil. Se utiliza comúnmente en la metodología Scrum para la gestión de proyectos.

¹³Asignatura del grado de Ingeniería Informática en la UBU que proporciona una visión general de la ciencia de redes y sus aplicaciones.

los datos necesarios. Además, se desarrolló una herramienta en Python que permitió la extracción de datos de estas diversas fuentes de manera eficiente y automatizada.

Una vez obtenidos los datos, se procedió a realizar un análisis de los mismos, aplicando técnicas y algoritmos propios de la ciencia de redes para extraer información relevante y obtener conclusiones significativas. Este análisis proporcionó una base sólida para la posterior redacción de la memoria del proyecto.

5. Aspectos relevantes del desarrollo del proyecto

5.1. Análisis estático de las redes: Bowtie

El concepto de "*bowtie*" hace referencia a una representación gráfica de una red que exhibe una estructura en forma de corbata. Se trata de una visualización que segmenta la red en diversas componentes y describe la conectividad existente entre ellas.

- **Número de nodos (Nº nodes):** Indica la cantidad total de nodos o entidades individuales presentes en la red. Cada nodo representa un elemento o entidad específica dentro del contexto de la red.
- **Número de aristas (Nº edges):** Representa la cantidad total de aristas o conexiones existentes entre los nodos de la red. Cada arista denota una relación o interacción entre dos nodos.
- **Primera componente fuertemente conectada (1st SCC):** Constituye una porción de la red en la cual todos los nodos están interconectados mutuamente a través de rutas directas o indirectas. En otras palabras, se establece un camino desde cualquier nodo de esta componente hacia cualquier otro nodo presente en ella.
- **Segunda componente fuertemente conectada (2nd SCC):** Representa otra parte de la red donde todos los nodos se encuentran interconectados de manera mutua, sin embargo, no existen conexiones directas entre los nodos de la primera componente fuertemente conectada y los nodos de esta componente.

- **Componente de entrada (In component):** Corresponde a la sección de la red que abarca todos aquellos nodos desde los cuales se puede trazar al menos una ruta hacia la primera componente fuertemente conectada.
- **Componente de salida (Out component):** Se refiere a la porción de la red que comprende todos los nodos hacia los cuales existe al menos una ruta partiendo desde la primera componente fuertemente conectada.
- **Tubos (Tubes):** Son trayectorias directas que transcurren desde la componente de entrada hacia la componente de salida sin atravesar la primera componente fuertemente conectada. Estos tubos sirven como enlaces entre las componentes de entrada y salida, eludiendo la estructura de la corbata.
- **Tendril de entrada (In tendrils):** Representan aquellos nodos que están conectados a la componente de entrada, pero no forman parte de la primera componente fuertemente conectada ni de los tubos.
- **Tendril de salida (Out tendrils):** Hacen referencia a los nodos que se encuentran conectados a la componente de salida, sin embargo, no forman parte de la primera componente fuertemente conectada ni de los tubos.
- **Desconectados (Disconnected):** Son los nodos que no están conectados a ninguna otra componente de la corbata y no poseen conexiones entrantes o salientes con otros nodos en la red.

Estos conceptos proporcionan una descripción detallada de la estructura de la red desde la perspectiva del enfoque *bowtie*, permitiendo identificar las diferentes componentes y su nivel de interconexión.

A continuación se presentan los resultados obtenidos para cada uno de los repositorios de paquetes analizados.

Bioconductor y CRAN

Para el repositorio de *Bioconductor*, solo tenemos un conjunto de datos obtenidos mediante *Olivia Finder*.

Para el repositorio de *CRAN* tenemos 4 conjuntos de datos. [5.1](#)

- El primero usa los datos de *Libraries.io* (*Depends* e *Imports*) sin discriminar versiones de paquete, es decir, para un paquete dado tenemos en cuenta todas las dependencias que ha tenido en cada una de sus versiones. Cabe destacar que este punto de vista carece de sentido, pero se ha incluido para comparar los resultados del anterior *TFG* donde no se tuvo en cuenta este aspecto.
- El segundo conjunto de datos usa la última versión de cada paquete disponible e incluye las dependencias del tipo (*Depends* e *Imports*).
- El tercer conjunto de datos usa la última versión de cada paquete disponible e incluye las dependencias del tipo (*Depends*, *Imports*, *Suggest* y *Enhances*).
- El cuarto conjunto de datos es el obtenido mediante *Olivia Finder* para los tipos de dependencias (*Depends* e *Imports*).

	Bioconductor Scraped	CRAN Librariesio 1	CRAN Librariesio 2	CRAN Librariesio 3	CRAN Scraped
Nodes (n)	3509	16174	15647	16055	18671
Edges (m)	28320	117724	76207	107370	113273
1st SCC	1	1405	1	923	1
2nd SCC	1	6	1	13	1
In Component	124	381	79	333	6
Out Component	0	11746	0	11269	0
Tubes	0	444	0	666	0
In tendrils	2161	1680	14980	2373	17984
Out tendrils	0	481	0	442	0
Disconected	1223	37	587	49	680
Attack	2109	15123	14395	15056	17223
Attack/n	0.6010	0.9350	0.9200	0.9378	0.9224
Failure	24.8173	1454.5255	24.5910	957.2135	33.5440
Failure/n	0.0071	0.0899	0.0016	0.0596	0.0018

Tabla 5.1: Tabla de datos para Bioconductor y CRAN

Se ha omitido la columna *CRAN Librariesio 1* en la tabla anterior ya que no aporta información relevante.

- *Bioconductor* sigue siendo más pequeño en términos de *nodos* (3509) en comparación con las fuentes de *CRAN Librariesio 2*, *CRAN Librariesio 3* y *CRAN Scraped*.
- En cuanto al número de *aristas* (*edges*), *CRAN Librariesio 3* tiene el valor más alto (107,370), seguido de *CRAN Scraped* (113,273) y *CRAN Librariesio 2* (76,207).

- La cantidad de *nodos* en la *componente fuertemente conectada (1st SCC)* es similar en *Bioconductor*, *CRAN Librariesio 2* y *CRAN Scrapped*, mientras que *CRAN Librariesio 3* tiene un número más alto de *nodos* en esta categoría.
- *Bioconductor* tiene un número menor de *nodos* en la *segunda componente fuertemente conectada (2nd SCC)* en comparación con las fuentes de *CRAN Librariesio 2*, *CRAN Librariesio 3* y *CRAN Scrapped*.
- En términos de *componentes débilmente conectadas*, *CRAN Librariesio 3* tiene el número más alto de *nodos*, seguido de *CRAN Scrapped*, *CRAN Librariesio 2* y *Bioconductor*.
- El número de *tubos (tubes)* en la red es cero en todas las fuentes de información.
- *Bioconductor* tiene un número mayor de *nodos* en los componentes en forma de *tendril* tanto de entrada como de salida en comparación con las fuentes de *CRAN Librariesio 2*, *CRAN Librariesio 3* y *CRAN Scrapped*.
- La categoría de *nodos desconectados* muestra que *CRAN Librariesio 3* tiene un número más alto de *nodos desconectados*, mientras que *Bioconductor* y *CRAN Scrapped* tienen valores más bajos.
- El análisis de *ataque (attack)* muestra que *Bioconductor* tiene un valor más bajo en comparación con las fuentes de *CRAN Librariesio 2*, *CRAN Librariesio 3* y *CRAN Scrapped*. Esto indica una mayor resistencia a *ataques* en *Bioconductor*.
- El análisis de *falla (failure)* muestra que *Bioconductor* tiene un valor más bajo en comparación con *CRAN Librariesio 3* y *CRAN Scrapped*. Sin embargo, *CRAN Librariesio 2* tiene el valor más bajo de *fallas*. Esto indica una mayor *robustez* en *Bioconductor* y *CRAN Librariesio 2* en comparación con *CRAN Librariesio 3* y *CRAN Scrapped*.

PyPI

En el contexto de PyPI, se dispone de tres conjuntos de datos para el análisis: [5.1](#).

El *primer* conjunto de datos, al igual que en el caso anterior, utiliza los datos de *libraries.io* y considera como dependencia de un paquete a todos

los paquetes que hayan sido alguna vez dependencia de dicho paquete a lo largo de todo su histórico de versiones.

El *segundo* conjunto de datos realiza un filtrado de los datos de *libraries.io*, considerando únicamente las dependencias asociadas a la última versión de cada paquete.

El *tercer* conjunto de datos abarca la red de dependencias obtenida mediante *Olivia Finder*.

	PyPI Libraries 1	PyPI Libraries 2	PyPI Scraped
Nodes (n)	50766	49306	214469
Edges (m)	155369	134575	933955
1st SCC	7	4	283
2nd SCC	4	4	19
In Component	39	21	449
Out Component	62	5	138219
Tubes	13	1	2446
In tendrils	23815	27742	30261
Out tendrils	13	11	14941
Disconected	26817	21522	27870
Attack	22315	19212	145000
Attack/n	0.4396	0.3896	0.6761
Failure	15.7301	8.5733	489.5527
Failure/n	0.0003	0.0002	0.0023

Tabla 5.2: Tabla de datos para PyPI

En la siguiente comparacion omitimos el conjunto de datos *PyPI Libraries 1* debido a que no es comparable con los otros dos conjuntos de datos.

- El conjunto de datos *PyPI Libraries 2* tiene un total de 49,306 nodos, mientras que el conjunto de datos *PyPI Scraped* cuenta con 214,469 nodos. Se aprecia una diferencia significativa en términos de tamaño de red entre ambos conjuntos.
- En cuanto al número de aristas, *PyPI Scraped* presenta un valor considerablemente más alto con 933,955 aristas, en comparación con las 134,575 aristas del conjunto *PyPI Libraries 2*.
- En relación a las componentes fuertemente conectadas, tanto *PyPI Libraries 2* como *PyPI Scraped* poseen 4 nodos en la segunda componente fuertemente conectada (*2nd SCC*), mientras que la cantidad de nodos en la primera componente fuertemente conectada (*1st SCC*) varía significativamente, siendo 4 para *PyPI Libraries 2* y 283 para *PyPI Scraped*.

- En términos de componentes débilmente conectadas, *PyPI Scraped* presenta una mayor cantidad de nodos en la componente de entrada (*In Component*) con 449 nodos, en comparación con los 21 nodos de *PyPI Libraries 2*. Por otro lado, *PyPI Scraped* también cuenta con una mayor cantidad de nodos en la componente de salida (*Out Component*) con 138,219 nodos, mientras que *PyPI Libraries 2* tiene solo 5 nodos en esta categoría.
- El número de tubos (*Tubes*) es significativamente mayor en *PyPI Scraped* con 2,446 tubos, en comparación con el único tubo presente en *PyPI Libraries 2*.
- En cuanto a los componentes en forma de tendril, *PyPI Libraries 2* tiene 27,742 nodos en los tendril de entrada (*In tendrils*) y 11 nodos en los tendril de salida (*Out tendrils*), mientras que *PyPI Scraped* tiene 30,261 nodos en los tendril de entrada y 14,941 nodos en los tendril de salida.
- La categoría de nodos desconectados (*Disconnected*) muestra que *PyPI Libraries 2* tiene 21,522 nodos desconectados, mientras que *PyPI Scraped* tiene 27,870 nodos desconectados.
- En cuanto al análisis de ataque (*Attack*), *PyPI Libraries 2* tiene un valor de 19,212 y *PyPI Scraped* presenta un valor más alto de 145,000, lo que indica una mayor susceptibilidad en *PyPI Scraped* en comparación con *PyPI Libraries 2*.
- En el análisis de falla (*Failure*), *PyPI Libraries 2* tiene un valor de 8.5733, mientras que *PyPI Scraped* muestra un valor más alto de 489.5527. Esto sugiere una mayor robustez en *PyPI Libraries 2* en comparación con *PyPI Scraped*.
- Los valores normalizados (*Attack/n* y *Failure/n*) indican la proporción de ataques y fallas en relación al número total de nodos. En este sentido, *PyPI Scraped* presenta valores más altos en ambas métricas en comparación con *PyPI Libraries 2*.

NPM

En el contexto del repositorio *npm*, se disponen de cuatro conjuntos de datos distintos, cada uno con sus propias características: [5.1](#)

- El primer conjunto de datos se basa en la información recopilada por *libraries.io*. Este conjunto considera todas las dependencias que un paquete ha tenido a lo largo de su historial de versiones. Incluye tanto las dependencias de tipo *runtime* como las de desarrollo (*dev*).
- El segundo conjunto de datos realiza un filtrado del conjunto anterior, conservando únicamente las dependencias asociadas a la última versión de cada paquete. Además, se restringe el análisis a las dependencias de tipo *runtime* y desarrollo, excluyendo otras dependencias no relevantes para el contexto.
- El tercer conjunto de datos utiliza los datos generados por *Olivia Finder* para las dependencias de tipo *runtime*. *Olivia Finder* es una herramienta específica que permite identificar y analizar las dependencias en tiempo de ejecución de los paquetes en el repositorio *npm*.
- Por último, el cuarto conjunto de datos se genera también mediante *Olivia Finder*, pero incluye tanto las dependencias de tipo *runtime* como las de desarrollo (*dev*). Esto proporciona una visión más completa de las dependencias utilizadas en el entorno de desarrollo y en tiempo de ejecución de los paquetes del repositorio *npm*.

	NPM Librariesio 1	NPM Librariesio 2	NPM Scraped 1	NPM Scraped 2
Nodes (n)	1074508	1064531	1059758	1832943
Edges (m)	13052831	11405275	4855094	22036615
1st SCC	26486	13378	26	19579
2nd SCC	175	157	17	451
In Component	3849	1827	0	3718
Out Component	936295	940266	1	1626207
Tubes	3745	4260	0	7599
In tendrils	17891	19759	0	50120
Out tendrils	69604	60947	0	77588
Disconected	16638	24094	1059731	48132
Attack	975555	968059	258821	1683928
Attack/n	0.9079	0.9094	0.2442	0.9187
Failure	27193.8251	13633.8779	62.0866	20934.7775
Failure/n	0.0253	0.0128	0.0001	0.0114

Tabla 5.3: Tabla de datos para NPM

Se han omitido los datos de *NPM Librariesio 1* debido a que no se considera un conjunto de datos relevante para el análisis.

- El conjunto de datos *NPM Librariesio 2* tiene un total de 1,064,531 nodos, mientras que el conjunto de datos *NPM Scraped 1* cuenta con 1,059,758 nodos y el conjunto de datos *NPM Scraped 2* tiene la mayor

cantidad de nodos con 1,832,943. Se puede observar una diferencia significativa en términos de tamaño de red entre los conjuntos de datos.

- En cuanto al número de aristas, el conjunto de datos *NPM Scraped 2* presenta un valor considerablemente más alto con 22,036,615 aristas, en comparación con las 11,405,275 aristas del conjunto *NPM Librariesio 2*.
- En relación a las componentes fuertemente conectadas, tanto el conjunto *NPM Librariesio 2* como el conjunto *NPM Scraped 2* poseen 157 nodos en la segunda componente fuertemente conectada (*2nd SCC*), mientras que la cantidad de nodos en la primera componente fuertemente conectada (*1st SCC*) varía significativamente, siendo 13,378 para *NPM Librariesio 2* y 19,579 para *NPM Scraped 2*.
- En términos de componentes débilmente conectadas, el conjunto *NPM Scraped 2* presenta una mayor cantidad de nodos en la componente de entrada (*In Component*) con 3,718 nodos, en comparación con los 1,827 nodos del conjunto *NPM Librariesio 2*. Por otro lado, el conjunto *NPM Scraped 2* también cuenta con una mayor cantidad de nodos en la componente de salida (*Out Component*) con 1,626,207 nodos, mientras que *NPM Librariesio 2* tiene solo 940,266 nodos en esta categoría.
- El número de tubos (*Tubes*) es significativamente mayor en el conjunto *NPM Scraped 2* con 7,599 tubos, en comparación con los 4,260 tubos presentes en el conjunto *NPM Librariesio 2*.
- En cuanto a los componentes en forma de tendril, *NPM Librariesio 2* tiene 19,759 nodos en los tendril de entrada (*In tendrils*) y 60,947 nodos en los tendril de salida (*Out tendrils*), mientras que *NPM Scraped 2* tiene 50,120 nodos en los tendril de entrada y 77,588 nodos en los tendril de salida.
- La categoría de nodos desconectados (*Disconnected*) muestra que *NPM Librariesio 2* tiene 24,094 nodos desconectados, mientras que *NPM Scraped 1* tiene la mayor cantidad de nodos desconectados con 1,059,731.
- En cuanto al análisis de ataque (*Attack*), *NPM Scraped 2* tiene un valor de 1,683,928 y *NPM Librariesio 2* presenta un valor ligeramente menor de 968,059, lo que indica una mayor susceptibilidad a esta métrica en *NPM Scraped 2* en comparación con *NPM Librariesio 2*.

- En el análisis de falla (*Failure*), *NPM Scraped 2* muestra un valor más alto de 20,934.7775, mientras que *NPM Librariesio 2* tiene un valor de 13,633.8779. Esto sugiere una mayor robustez en *NPM Librariesio 2* en comparación con *NPM Scraped 2*.
- Los valores normalizados (*Attack/n* y *Failure/n*) indican la proporción de y fallas en relación al número total de nodos. En este sentido, *NPM Scraped 2* presenta valores más altos en ambas métricas en comparación con *NPM Librariesio 2*.

5.2. La red de dependencias de CRAN

CRAN (*Comprehensive R Archive Network*) es un *repositorio* en línea que alberga una amplia colección de paquetes de software para el *lenguaje de programación R*. R es un *entorno de programación* y un *lenguaje estadístico* ampliamente utilizado en la comunidad científica para el análisis y la visualización de datos. El lenguaje R se destaca por su *flexibilidad* y *extensibilidad*, lo que permite a los investigadores y científicos implementar *algoritmos estadísticos* avanzados y realizar *análisis exploratorios de datos*. Los paquetes almacenados en CRAN ofrecen una variedad de funcionalidades especializadas, incluyendo *modelado estadístico*, *gráficos*, *manipulación de datos* y *visualización*, lo que permite a los usuarios ampliar las capacidades base de R.

Descripción	Cantidad
Packages in librariesio	15154
Packages in scraped	18195
Common packages	11589
Packages in librariesio that are not in scraped	3565
Packages in scraped that are not in librariesio	6606

Tabla 5.4: Comparación de paquetes en CRAN entre los datos de libraries.io y los recolectados en este trabajo.

El análisis sobre los datos recopilados 5.4 indica que el conjunto de datos *scraped* ha experimentado un crecimiento notable en comparación con *libraries.io*, tanto en términos de la cantidad total de paquetes como en la inclusión de nuevos paquetes. Esta ampliación evidencia una recopilación más exhaustiva y actualizada de información. 5.1 5.2.

CRAN repository packages evolution

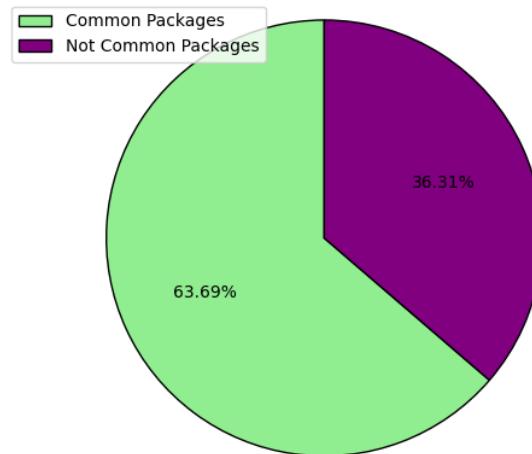


Figura 5.1: Comparacion de los paquetes comunes entre los dos conjuntos de datos.

CRAN repository packages evolution

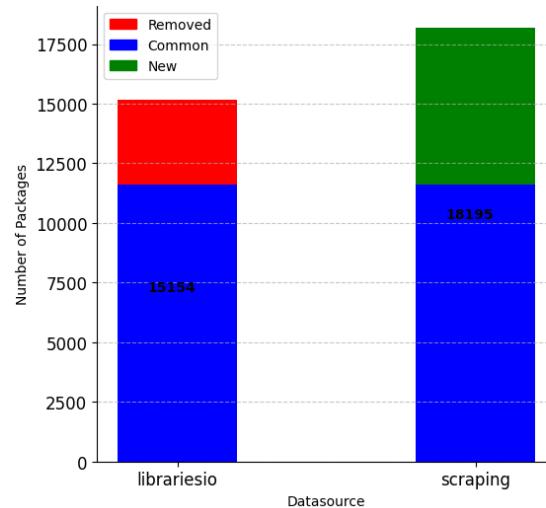


Figura 5.2: Comparacion del numero de paquetes.

Por consiguiente, se concluye que el conjunto de datos *scraped* proporciona una visión más completa y actualizada de los paquetes disponibles. Al

incluir tanto los paquetes en común como los paquetes adicionales respecto a *libraries.io*, *scraped* se configura como una fuente valiosa para investigaciones y análisis en el ámbito de estudio.

El tamaño

A continuación realizaremos una comparación de las medidas de tamaño entre los dos conjuntos de datos y como es el agrupamiento de los nodos en cada uno de ellos. 5.5

Medida	<i>libraries.io</i>	<i>scraped</i>
Number of nodes	15647	18671
Number of edges	76207	113273
Average degree	9.740	12.133
Average clustering coefficient	0.131	0.152

Tabla 5.5: Comparación de medidas de tamaño entre los dos conjuntos de datos.

La red *scraped* tiene un mayor número de nodos (18671) en comparación con la red *libraries.io* (15647). Esto indica que *scraped* contiene más elementos o entidades interconectadas en su estructura de red.

La red *scraped* también tiene un mayor número de aristas o enlaces (113273) en comparación con la red *libraries.io* (76207). Esto implica que *scraped* tiene más conexiones entre los nodos, lo que aumenta su grado de conectividad.

El grado promedio de los nodos en la red *scraped* es más alto (12.133) en comparación con el de la red *libraries.io* (9.740). Esto sugiere que, en promedio, cada nodo en *scraped* tiene más conexiones con otros nodos en comparación con los nodos en *libraries.io*.

El coeficiente de agrupamiento promedio en la red *scraped* es ligeramente más alto (0.152) que en la red *libraries.io* (0.131). Esto indica que, en promedio, los nodos en *scraped* tienen una mayor tendencia a formar grupos o comunidades más densamente interconectadas en comparación con los nodos en *libraries.io*.

El grado

Al realizar una comparativa de las distribuciones de grado entre ambos conjuntos de datos, se evidencia un leve incremento en el grado promedio.

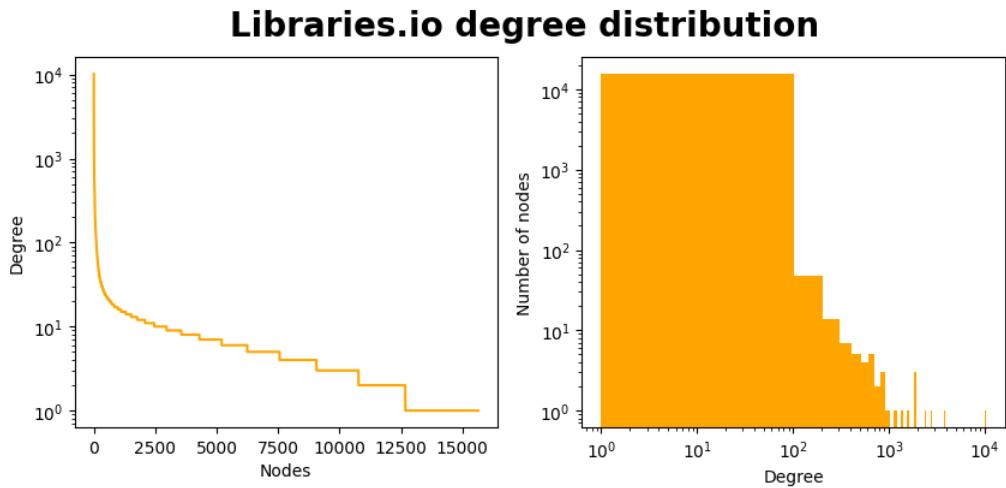


Figura 5.3: Distribucion de grado *libraries.io*.

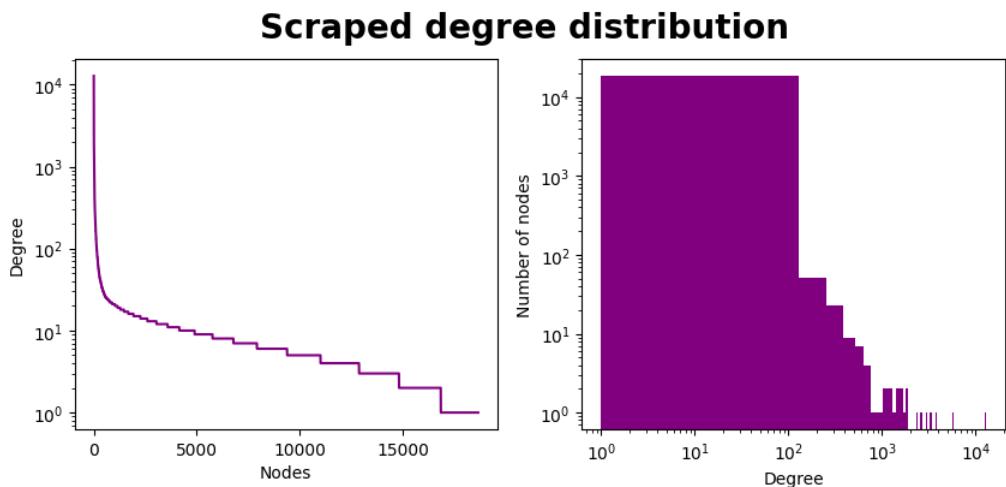


Figura 5.4: Distribucion de grado *scraped*.

El grado promedio del conjunto de datos *libraries.io* es de 4.87, mientras que en el conjunto *scraped* alcanza un valor de 6.06. Estos resultados indican un incremento en la conectividad y la relevancia de los paquetes más destacados en el conjunto de datos *scraped*. Tal variación en los valores promedio refleja el aumento en la importancia y la interconexión de los paquetes dentro de

este conjunto de datos, lo cual puede estar relacionado con su mayor tamaño y actualización. [5.3](#) [5.4](#)

Además, se observa que la tendencia a disminuir el número de dependencias es ligeramente más pronunciada que la tendencia a aumentarlas. Esto sugiere que, en la evolución de la red de dependencias, los paquetes tienden a reducir su dependencia directa o a reorganizar sus conexiones con otros paquetes, lo que puede ser resultado de procesos de *refactorización*, *optimización* o *consolidación*.

Grado de salida (*out degree*)

En el análisis de la distribución del grado de salida, no se observan diferencias significativas que puedan ser comparadas. Podemos inferir que ambas distribuciones siguen una tendencia similar, con la única distinción de que en el conjunto de datos nuevo se encuentra un mayor número de nodos [5.5](#) [5.6](#).

CRAN libraries.io out degree distribution

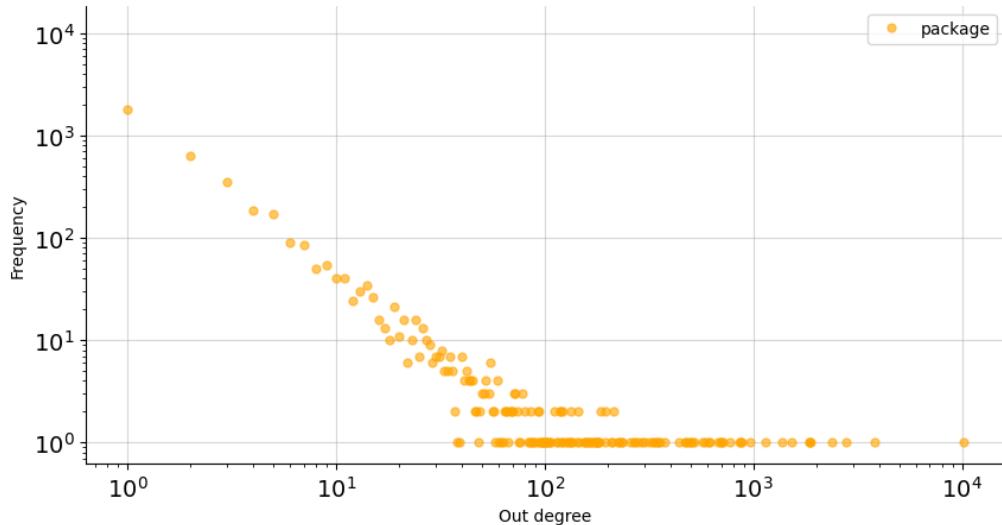


Figura 5.5: Distribucion de Out degree de *libraries.io*.

Al examinar el conjunto de paquetes con mayor grado de salida, se observa un aumento generalizado en todos ellos. Estos paquetes destacados representan una centralidad de grado significativa, lo que implica que son las dependencias más utilizadas dentro del sistema. Es comprensible que estos

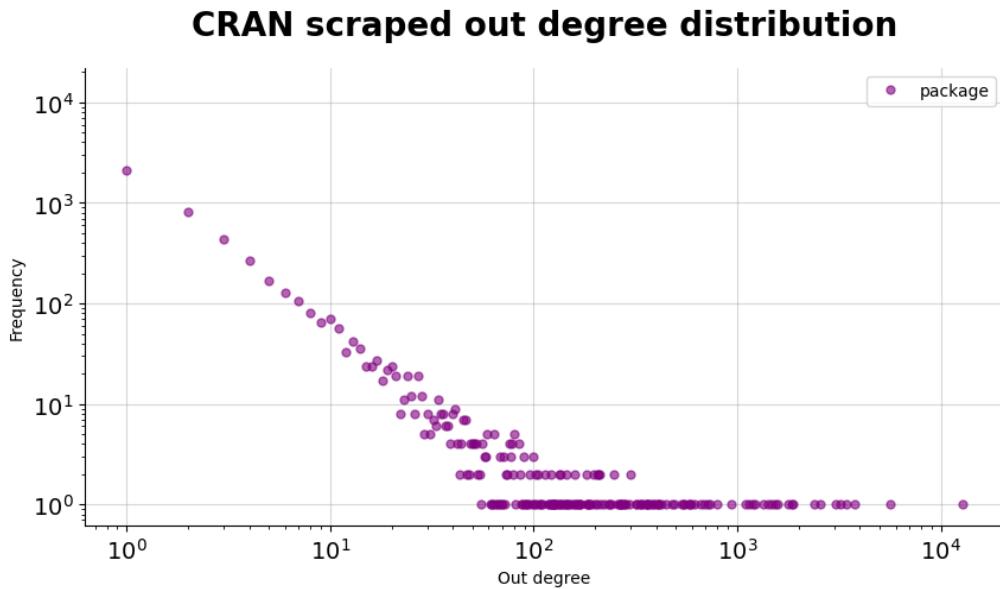


Figura 5.6: Distribucion de Out degree de *scraped*.

paquetes, al ser ampliamente utilizados, hayan mantenido una presencia constante en la red y hayan experimentado un incremento en el número de sus dependientes debido al surgimiento de nuevos paquetes. 5.7

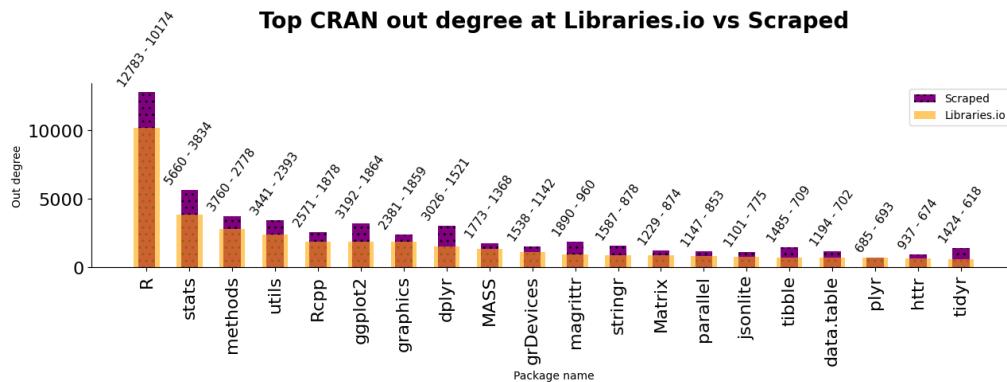


Figura 5.7: Top paquetes con mayor grado de salida en *libraries.io*.

Esta observación refuerza la importancia y la relevancia de estos paquetes clave en el ecosistema estudiado. Su estabilidad y el aumento en sus dependientes pueden atribuirse a su funcionalidad y a su amplia adopción por

parte de los usuarios. Además, el incremento en la cantidad de dependientes es un indicativo del crecimiento y la evolución continua del sistema, donde se generan nuevas relaciones de dependencia entre los paquetes existentes y los recién agregados.

Al realizar la comparativa utilizando el nuevo conjunto de datos, se observa que los principales representantes del ranking se mantienen presentes. Sin embargo, se han producido algunas variaciones en el orden de algunos puestos dentro del top. Además, se ha registrado la inclusión de paquetes que previamente se encontraban fuera del top. [5.8](#)

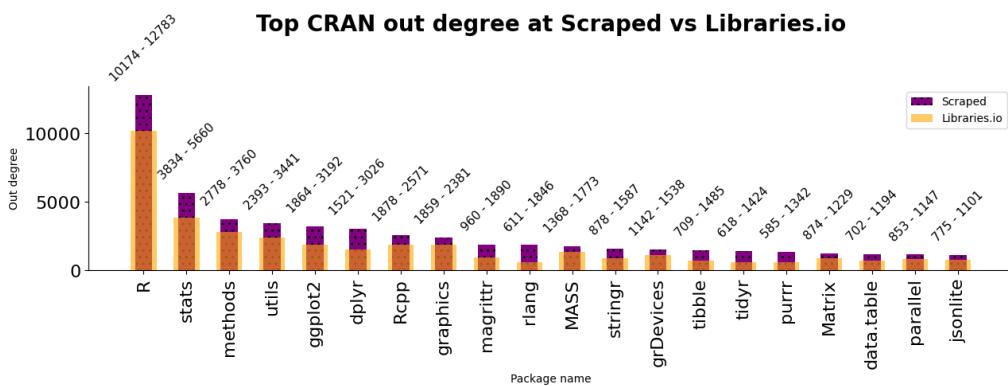


Figura 5.8: Top paquetes con mayor grado de salida en *scraped*.

Esta inclusión de paquetes puede atribuirse al incremento en el número de sus dependencias, el cual ha superado a aquellos que han salido del top en este periodo de tiempo analizado. Estos nuevos paquetes han logrado adquirir una mayor relevancia y han fortalecido su posición dentro del conjunto de datos. Este fenómeno puede ser consecuencia de su creciente adopción y de la expansión de su funcionalidad, lo que ha llevado a un incremento en el número de paquetes que los utilizan como dependencias.

Este incremento lo podemos ver representado en la siguiente figura [5.9](#).

Grado de entrada (*in degree*)

En el análisis de la distribución del grado de entrada, se observa un comportamiento común en ambos conjuntos de datos. Debido a su similitud, resulta difícil extraer conclusiones significativas. Sin embargo, se puede observar un ligero incremento en el nuevo conjunto de datos en comparación con el conjunto de datos de libraries.io. [5.10](#) [5.11](#)

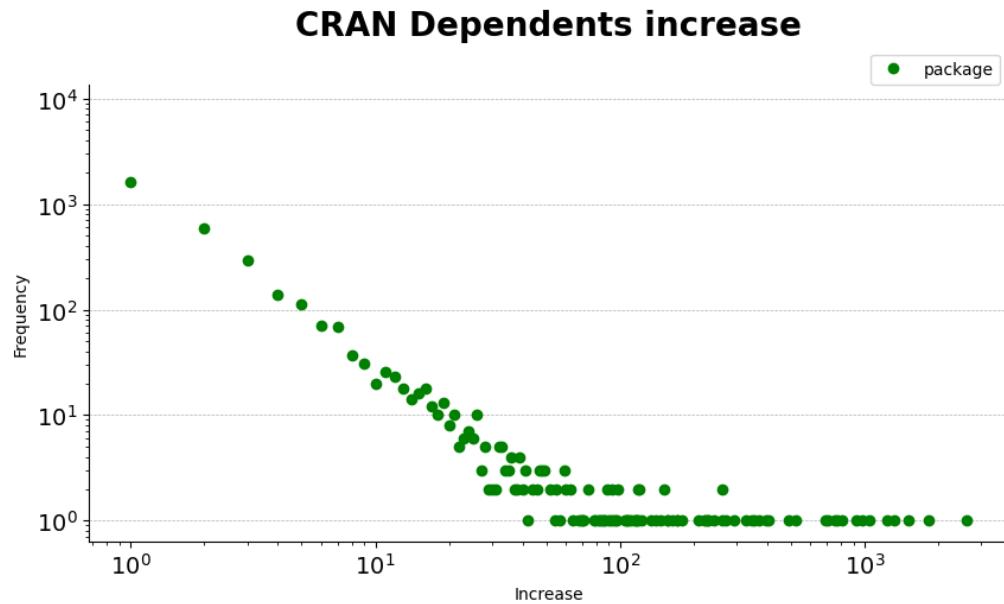


Figura 5.9: Distribucion de dependientes.

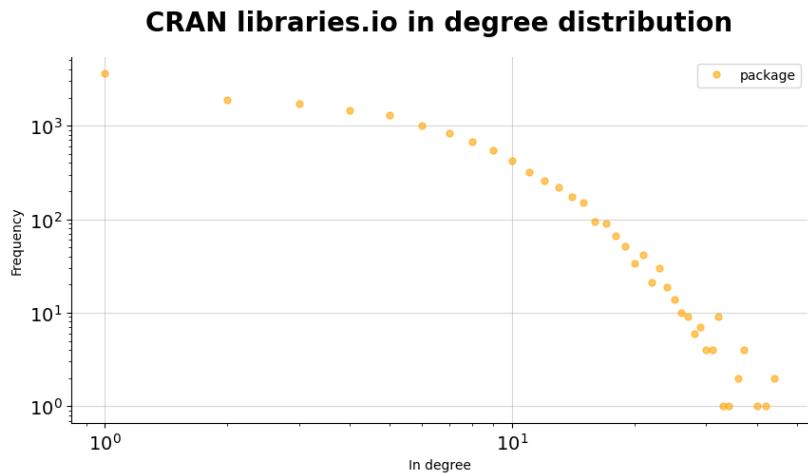


Figura 5.10: Distribucion de In degree de *libraries.io*.

Tomando como punto de referencia la frecuencia de nodos con un grado de entrada de valor 20, en el conjunto de datos de *libraries.io* se encuentran alrededor de 50 nodos, mientras que en el nuevo conjunto de datos se registran aproximadamente 100 nodos. Estos valores pueden sugerir un

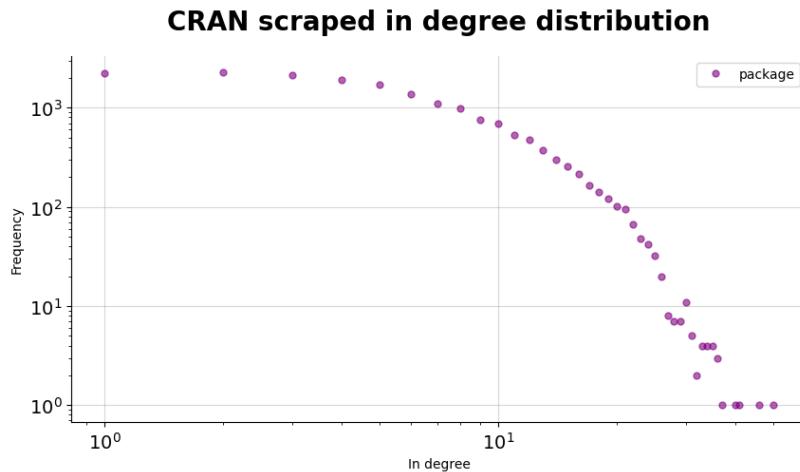


Figura 5.11: Distribucion de In degree de *scraped*.

aumento en la cantidad de paquetes que reciben un número determinado de dependencias.

Si representamos el *top* de *in degree* de los paquetes del conjunto de datos de *libraries.io*, podemos identificar aquellos paquetes que poseen un mayor número de dependencias. Estos paquetes podrían considerarse los más vulnerables en su primer nivel de dependencia, sin tener en cuenta la transitividad. Al analizar los resultados, se puede concluir que un porcentaje significativo de los paquetes destacados en este *top* han mantenido su presencia en la red a lo largo del tiempo.

Además, se observa una tendencia general a disminuir el número de dependencias para la mayoría de los casos en el *top*. Esto sugiere que, a medida que evoluciona la red de dependencias, algunos paquetes han logrado reducir su dependencia directa o han redistribuido sus conexiones con otros paquetes.

Desde la perspectiva del *top* de *in degree*, se observa una notable variación en la evolución de los paquetes. Aproximadamente la mitad de los individuos que conformaban el *top* han sido reemplazados. Estos paquetes de reemplazo se caracterizan por ser nuevos en la red, lo cual sugiere que están utilizando funcionalidades previamente desarrolladas para generar nuevas capacidades.

Es común que aparezcan nuevos paquetes en este *top*, dado el contexto de una red de carácter científico. A medida que el software madura, es habitual

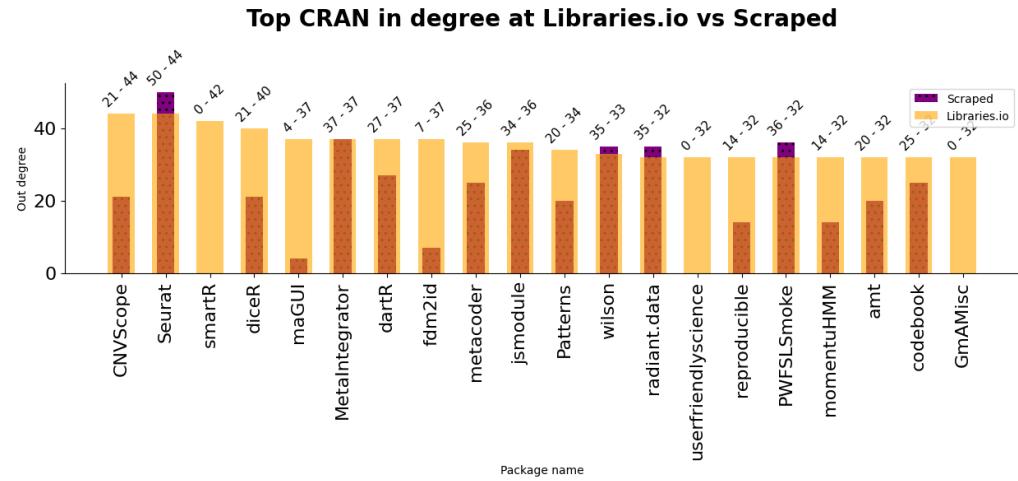


Figura 5.12: Top paquetes con mayor grado de entrada en *libraries.io*.

que estos paquetes más jóvenes se refactoricen con el tiempo, tendiendo a disminuir sus dependencias.

Por otro lado, existen paquetes que no solo se encontraban en el *top* anterior, sino que han incrementado sus dependencias. Este aumento puede ser indicativo de paquetes cuya funcionalidad aún está en desarrollo y continúa evolucionando. 5.13

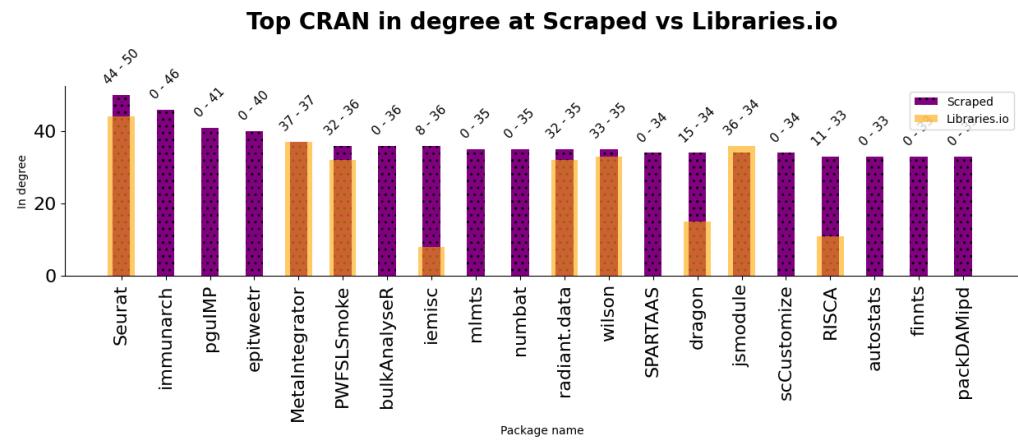


Figura 5.13: Top paquetes con mayor grado de entrada en *scrapped*.

Al observar la tendencia del incremento de dependencias en la red, se evidencia que, en general, es común que el número de dependencias de los paquetes no experimente cambios drásticos. La mayoría de los paquetes se sitúan en un rango de incremento de dependencias de aproximadamente ± 10 . 5.14

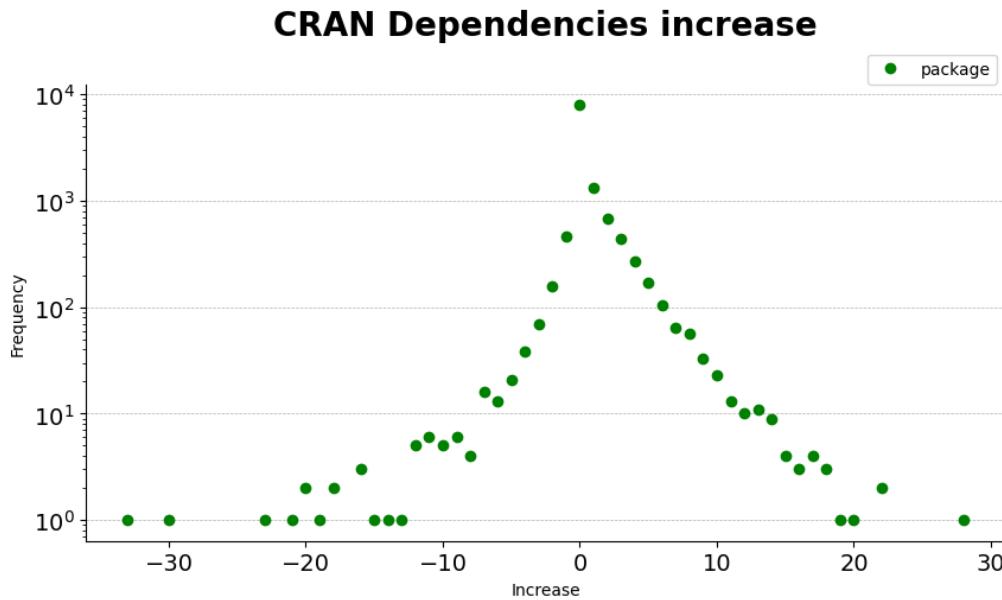


Figura 5.14: Incremento de dependencias.

El *PageRank*

A partir del análisis de la distribución de *PageRank* en la red, se observa la presencia de numerosos nodos con un *PageRank* bajo, y a medida que se incrementa el valor del *PageRank*, la frecuencia de nodos disminuye gradualmente. Este patrón revela la existencia de muchos nodos de baja importancia en la red, junto con un grupo reducido de paquetes que son considerados importantes debido a sus dependencias, las cuales, a su vez, son también dependencias relevantes en la red.

Al comparar las dos distribuciones, se aprecia una diferencia notable en la red de *libraries.io*, donde el valor máximo alcanzado por el *PageRank* de un paquete es mayor en comparación con la nueva red. Esta diferencia puede interpretarse como un indicio de que, en la evolución de *CRAN*, los paquetes más importantes se han estabilizado, mientras que han surgido otros

paquetes que están adquiriendo relevancia. Como resultado, el *PageRank* se ha distribuido de manera más equitativa entre los paquetes de la red. 5.15
5.16

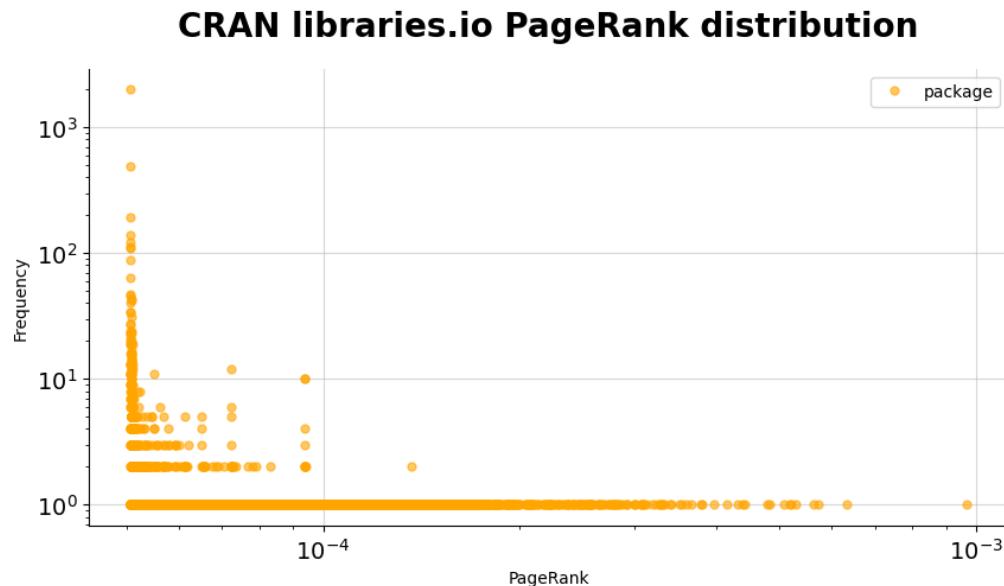


Figura 5.15: Distribucion de *PageRank* en *libraries.io*.

El top de mayor *PageRank* en *libraries.io* nos brinda una visión de los paquetes que se consideran como nodos centrales en la red de dependencias. Esto implica que son elementos fundamentales para la funcionalidad y el rendimiento de otros paquetes, dado que poseen un mayor número de dependencias directas e indirectas.

Estos paquetes tienden a ser dependientes de otros paquetes con alto *PageRank*. Además, desde el punto de vista de la vulnerabilidad, son considerados críticos debido a que suelen acumular una alta dependencia transitiva. Esto significa que cualquier cambio o problema en estos paquetes centrales puede tener un impacto significativo en todo el ecosistema de la red de dependencias.

Estos nodos de alto *PageRank* juegan un papel crucial en el mantenimiento y la estabilidad de la red de dependencias. Su importancia radica en su alta interconexion con otros paquetes similares. 5.17

En base a la introducción anterior, es esperable encontrar paquetes en el top de *libraries.io* que no estén presentes en el conjunto de datos de *scraped*.

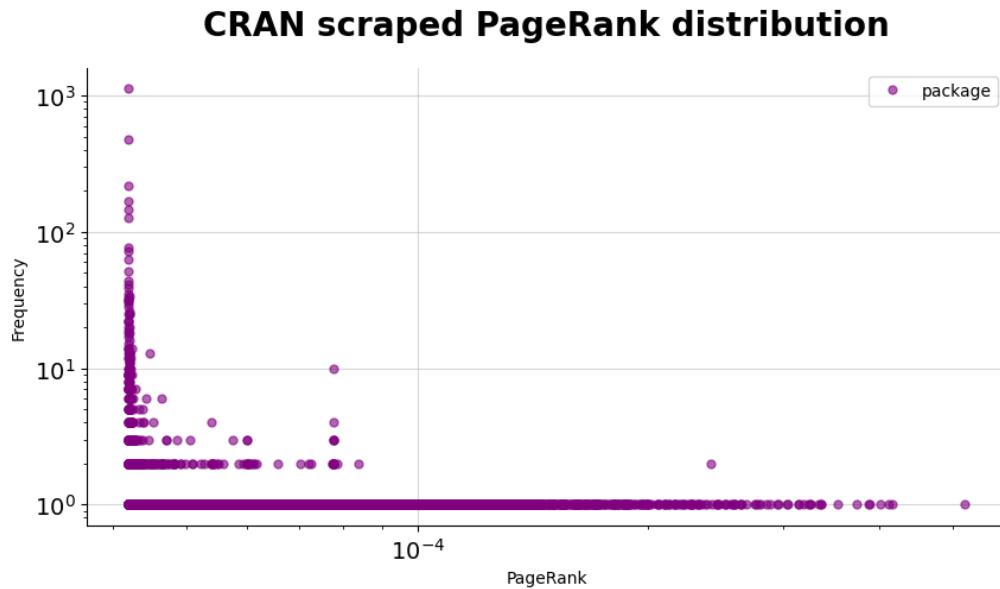


Figura 5.16: Distribucion de *PageRank* en *scraped*.

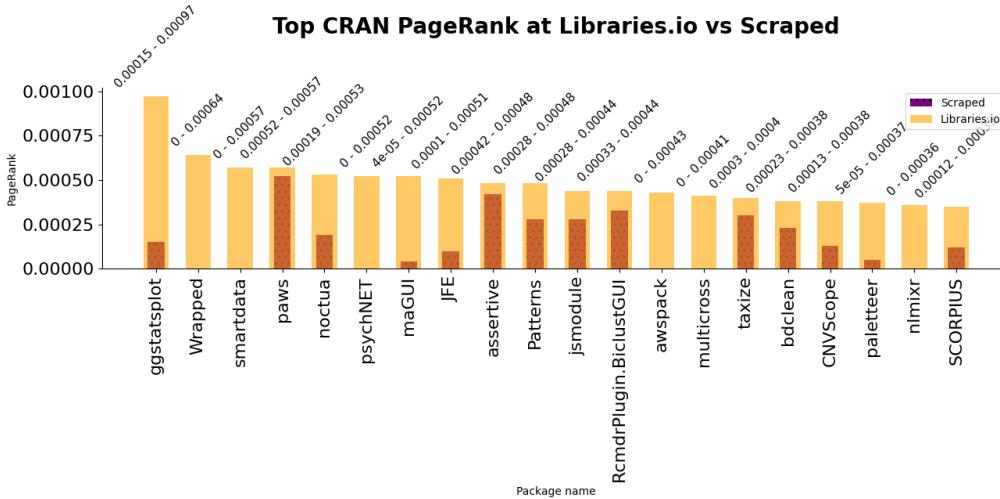


Figura 5.17: Top *PageRank* en *libraries.io*.

A partir del nuevo conjunto de datos 5.18, se observa un descenso general en los valores de PageRank en comparación con el conjunto de datos de *libraries.io*. Además, se destaca que un gran número de paquetes han ingresado al top de PageRank y no estaban presentes en *libraries.io*. Este

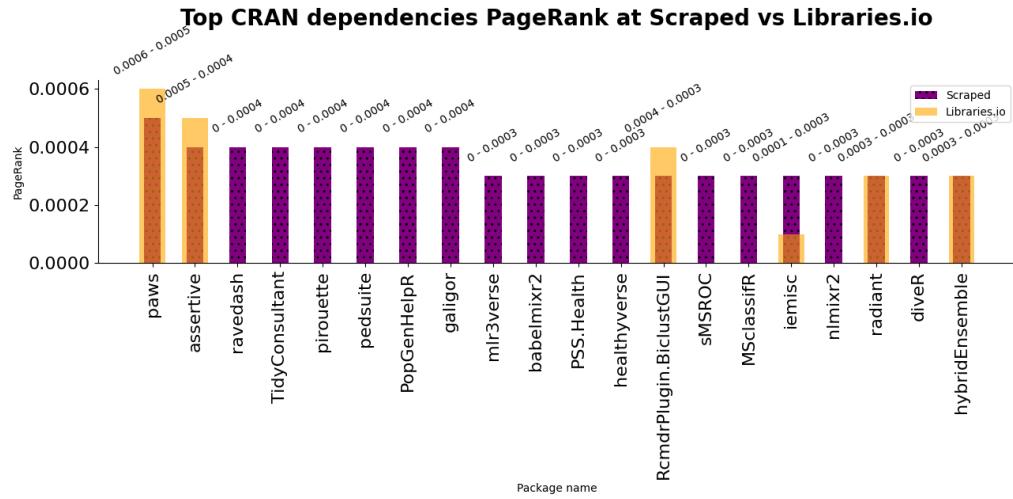


Figura 5.18: Top *PageRank* en *scraped*.

hallazgo refuerza la teoría de que el PageRank, en el contexto de una red de dependencias, puede considerarse un indicador de la vulnerabilidad de un paquete. Se puede inferir que estos paquetes recién incorporados tienden a tener una presencia transitoria en la red y, debido a su relativa falta de madurez, es probable que experimenten variaciones significativas a lo largo del tiempo.

Sin embargo, resulta más interesante centrar la atención en aquellos paquetes que mantienen un alto valor de PageRank a lo largo del tiempo. Estos paquetes indican la presencia de dependencias importantes y altamente transitivas en la red, lo que los hace potencialmente más vulnerables. Su capacidad para conservar su importancia en el contexto de la evolución de la red sugiere que desempeñan un papel fundamental en el funcionamiento y la estabilidad de otros paquetes. 5.19 5.20

Desde una perspectiva inversa, el uso del *PageRank* nos permite identificar qué paquetes son dependencia de otros paquetes importantes en la red de dependencias. Esto proporciona una visión de la centralidad en términos de popularidad de un paquete. Al analizar el *PageRank* bajo esta perspectiva, es posible determinar qué paquetes son altamente requeridos por otros paquetes y, por lo tanto, desempeñan un papel crucial en el funcionamiento de la red. 5.21

Los paquetes mostrados en este top son los mas populares del ecosistema de *R*. Estos paquetes son los mas utilizados por otros paquetes, lo que los

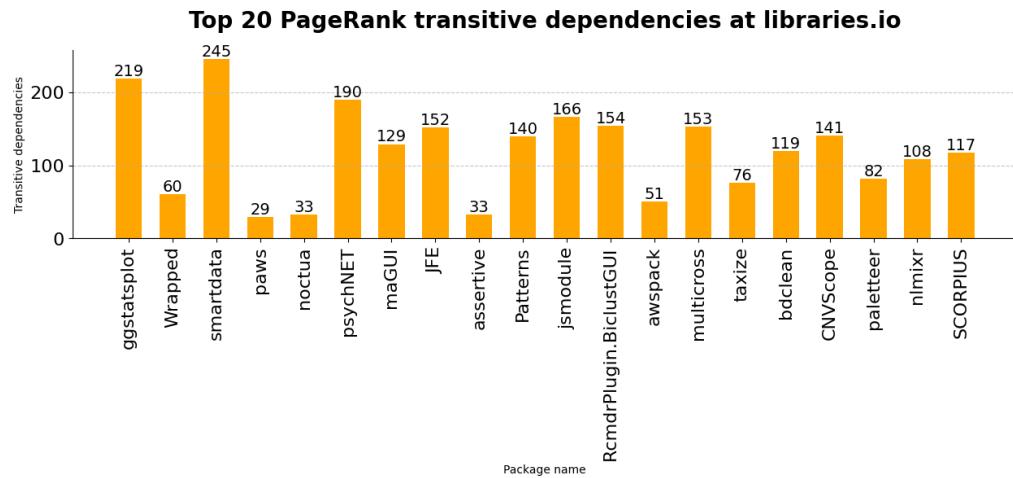


Figura 5.19: Top 20 *PageRank* numero de dependencias transitivas en libraries.io

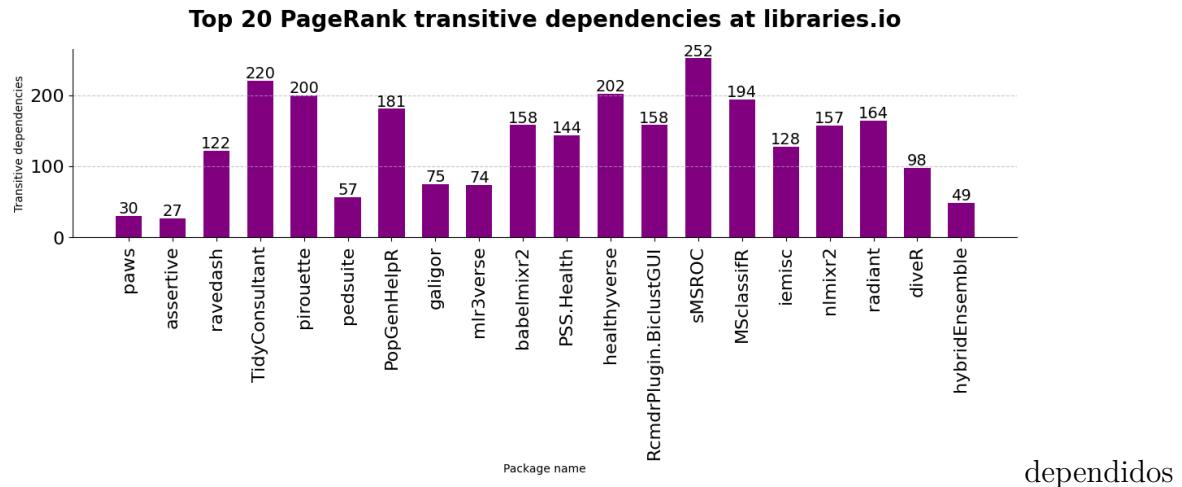
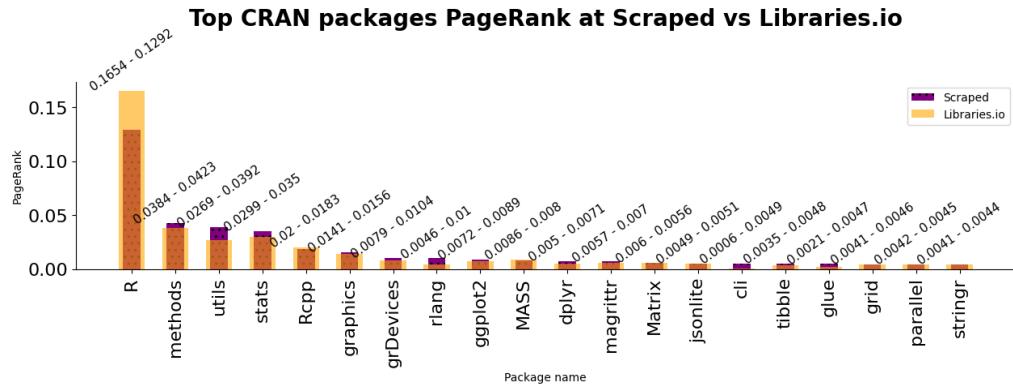
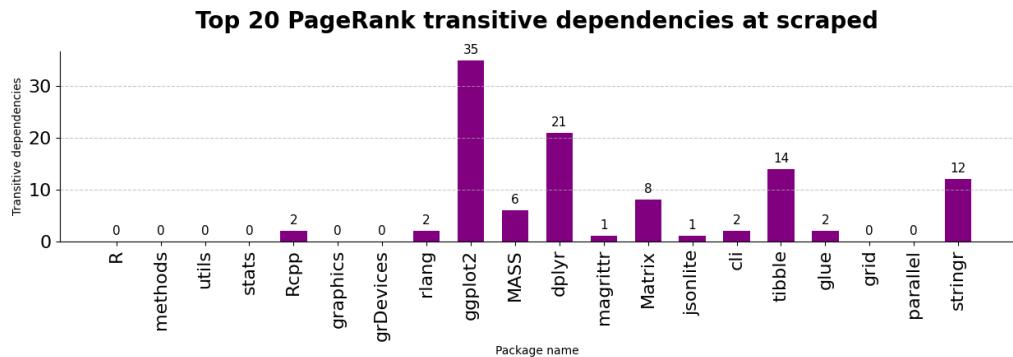


Figura 5.20: Top 20 *PageRank* numero de dependencias transitivas en scraped

hace fundamentales para el funcionamiento de la red de dependencias. No es raro que el paquete R, que implementa el *core* del lenguaje, sea el paquete mas popular.

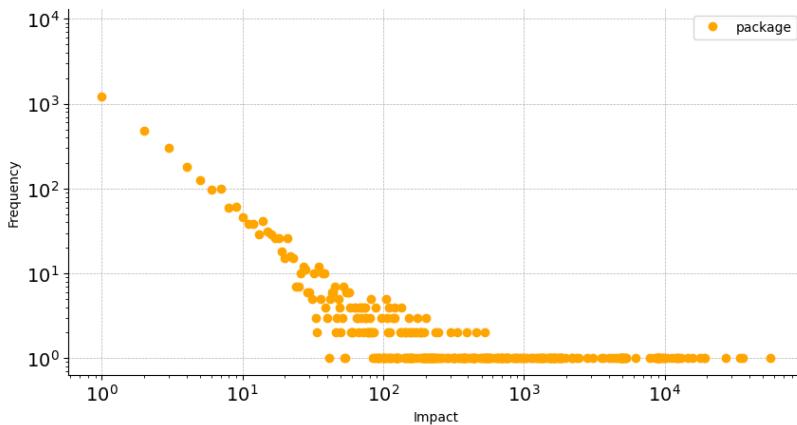
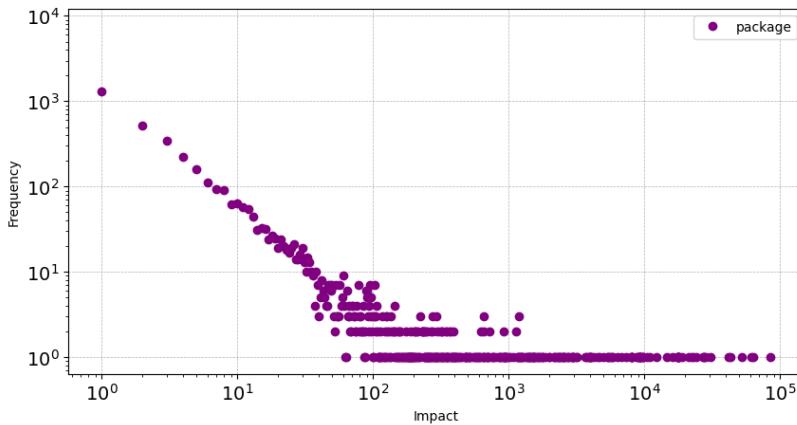
Estos paquetes a nivel de vulnerabilidad son los mas estables, ya que el numero de dependencias transitivas que tienen es relativamente bajo. 5.22

Figura 5.21: Top 20 *PageRank* paquetes en scrapedFigura 5.22: Top 20 *PageRank* paquetes numero de dependencias transitivas en scraped

Impacto (*Impact*)

Desde el punto de vista de esta métrica, podemos interpretar la vulnerabilidad de un paquete como el número de dependencias paquetes que se ven afectados por un cambio en el paquete. En este sentido, un paquete vulnerable es aquel que tiene un alto impacto en la red de dependencias.

A partir del análisis de la distribución del impacto, se observa una tendencia similar en ambos conjuntos de datos. Se aprecia un ligero incremento en el impacto en el nuevo conjunto de datos, pero en general los valores se mantienen estables. Este incremento puede atribuirse a la incorporación de nuevos paquetes en la red, los cuales han adoptado dependencias existentes, lo que ha aumentado el impacto de estas últimas. [5.23](#) [5.24](#)

CRAN Impact distribution for libraries.ioFigura 5.23: Distribución de *Impact* en libraries.io**CRAN Impact distribution for scraped**Figura 5.24: Distribución de *Impact* en scraped

En el top de paquetes con mayor *impacto* se encuentran aquellos que son considerados los más populares en la red, en términos de su influencia sobre otros paquetes. Es notable que los principales representantes de este top también aparecen en el top del *PageRank* a nivel de la red de paquetes.

Desde el punto de vista de la *dependencia transitiva*, estos paquetes son aquellos que tienen la mayor cantidad de dependientes en toda la red.

Al analizar el *impacto* de estos paquetes, se observa un aumento en su valor, en algunos casos significativo. Esto indica que estos paquetes han demostrado ser estables en el tiempo y en su implementación, y que su funcionalidad es altamente útil para la red.

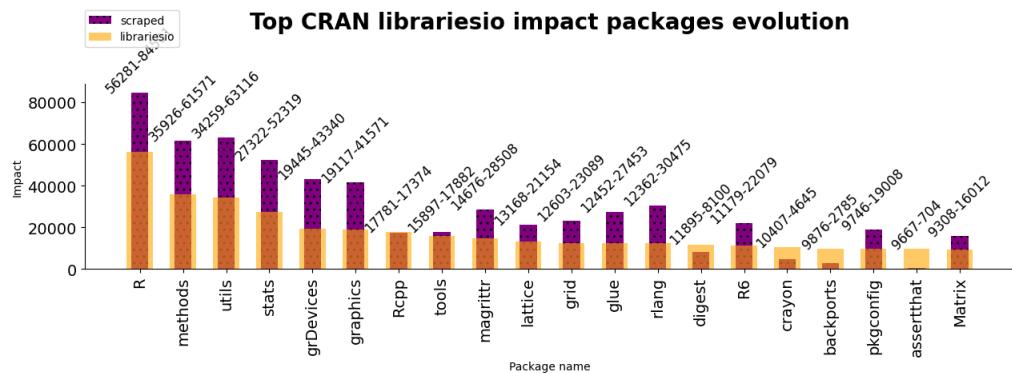


Figura 5.25: Top paquetes con mayor *Impact* en libraries.io

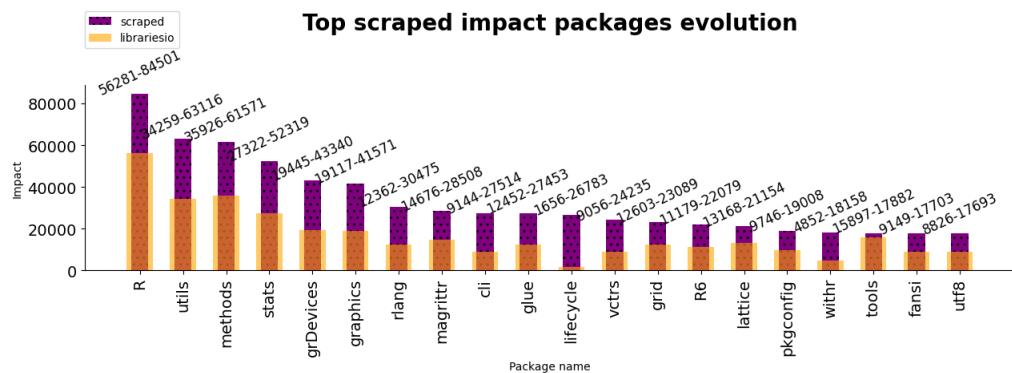


Figura 5.26: Top paquetes con mayor *Impact* en scraped

Por otro lado, aquellos paquetes en el top que han experimentado una reducción en su *impacto* podrían indicar la aparición de otros paquetes con funcionalidades similares o mejoradas, los cuales han sido elegidos como alternativas por los usuarios de la red. [5.25](#) [5.26](#)

En la tablas del incremento podemos ver que los paquetes que han tenido un mayor incremento y decremento en su impacto. [5.6](#) [5.7](#)

Package	libraries.io	scraped	increment
utils	34259	63116	28857
R	56281	84501	28220
methods	35926	61571	25645
lifecycle	1656	26783	25127
stats	27322	52319	24997
grDevices	19445	43340	23895
graphics	19117	41571	22454
cli	9144	27514	18370
rlang	12362	30475	18113
vctrs	9056	24235	15179

Tabla 5.6: Top 10 paquetes con mayor incremento en *Impact*

Package	libraries.io	scraped	increment
zeallot	9070	67	-9003
assertthat	9667	704	-8963
backports	9876	2785	-7091
crayon	10407	4645	-5762
reshape2	5217	1232	-3985
digest	11895	8100	-3795
plyr	6265	2710	-3555
lazyeval	4662	1193	-3469
formatR	1450	90	-1360
markdown	1511	324	-1187

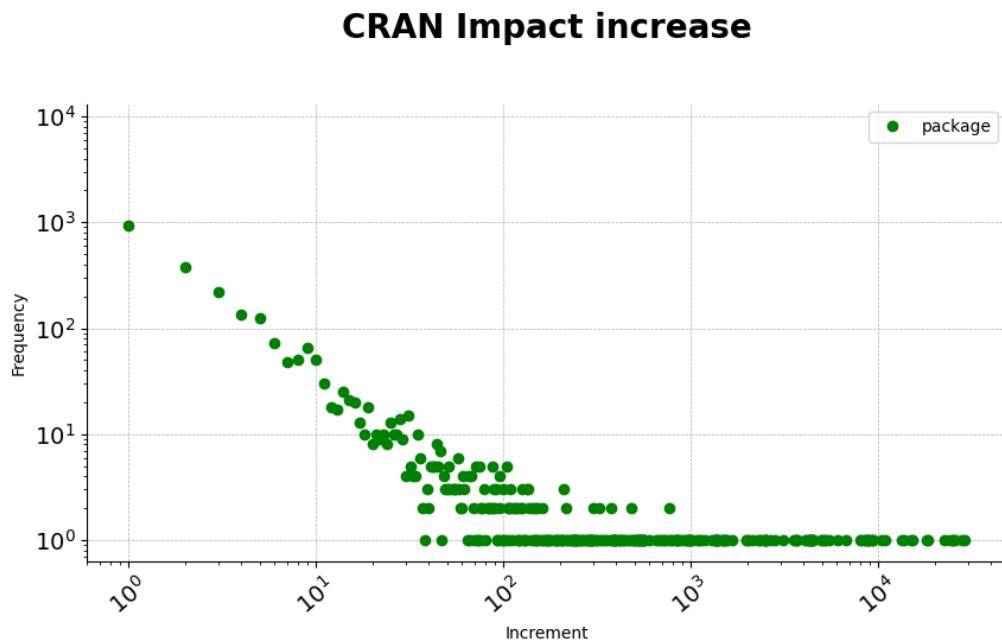
Tabla 5.7: Top 10 paquetes con mayor decrecimiento en *Impact*

Si lo representamos graficamente se obtiene una tendencia similar a las distribuciones anteriores. [5.27](#)

Alcance (*Reach*)

El alcance (*Reach*) de un paquete p, equivale al número de sucesores transitivos de p más 1 y mide el número de paquetes potencialmente afectados por un defecto en p.

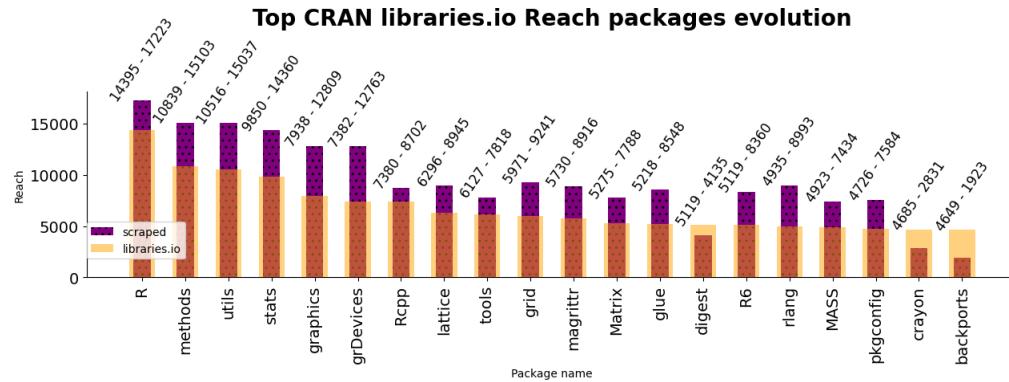
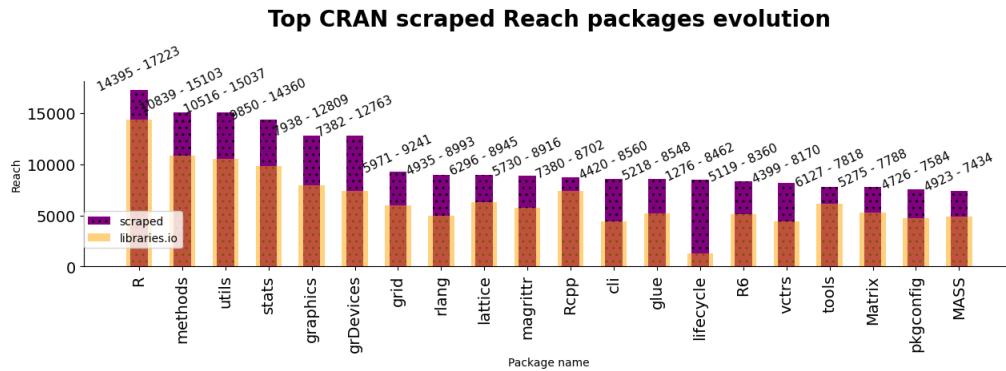
A la vista de las siguientes tablas podemos observar que paquetes son los que tienen un mayor reach. [5.8](#)

Figura 5.27: Incremento de *Impact*

libraries.io	scraped
R, 14395	R, 17223
methods, 10839	methods, 15103
utils, 10516	utils, 15037
stats, 9850	stats, 14360
graphics, 7938	graphics, 12809
grDevices, 7382	grDevices, 12763
Rcpp, 7380	grid, 9241
lattice, 6296	rlang, 8993
tools, 6127	lattice, 8945
grid, 5971	magrittr, 8916

Tabla 5.8: Top 10 paquetes con mayor *Reach*

Si comparamos los resultados para ambos conjuntos de datos nos damos cuenta de que los representantes de este top en su gran medida coinciden, esto es explicable ya que el *Reach* de un paquete depende de los paquetes que lo usan, y estos paquetes en R suelen ser los más populares. [5.28](#) [5.29](#)

Figura 5.28: Top paquetes con mayor *Reach* en libraries.ioFigura 5.29: Incremento de *Reach*

Si observamos los paquetes con mayor incremento en *Reach* 5.9 podemos ver que la mayoría estan presentes en el top, esta tabla nos da una idea de durante este periodo de tiempo que paquetes han ganado mas dependientes.

Es curioso ver que el paquete *isoband* haya incrementado tanto su *Reach*. Este paquete es una implementación de la generación de bandas de confianza para curvas y mapas de contorno. Resulta que este paquete fue protagonista de un incidente en el pasado:

El paquete *isoband* estuvo en riesgo de ser archivado en *CRAN*. La razón por la que este incidente causó revuelo es que *isoband* es una dependencia de *ggplot2* y cuando un paquete es eliminado de *CRAN*, todos los demás paquetes que dependen de él también son eliminados. Si *isoband* hubiera caído, *ggplot2* estaría en riesgo. Y esto habría desencadenado la eliminación

Paquete	libraries.io	scraped	increment
lifecycle	1276	8462	7186
grDevices	7382	12763	5381
farver	48	5020	4972
graphics	7938	12809	4871
isoband	5	4726	4721
utils	10516	15037	4521
stats	9850	14360	4510
splines	2016	6466	4450
generics	457	4895	4438
methods	10839	15103	4264

Tabla 5.9: Top 10 paquetes con mayor incremento en *Reach*

de aún más paquetes. En total, la eliminación de isoband habría llevado a la eliminación de 4747 paquetes[23]. Afortunadamente los desarrolladores de isoband pudieron solucionar el problema y el paquete no fue eliminado. [24].

Luego este incremento en el *Reach* puede ser debido a que los desarrolladores de paquetes que dependen de isoband se estan reenganchando al proyecto y actualizando sus paquetes para que sigan funcionando correctamente depues del incidente.

Por ultimo mostramos la distribucion de incremento de *Reach* 5.30.

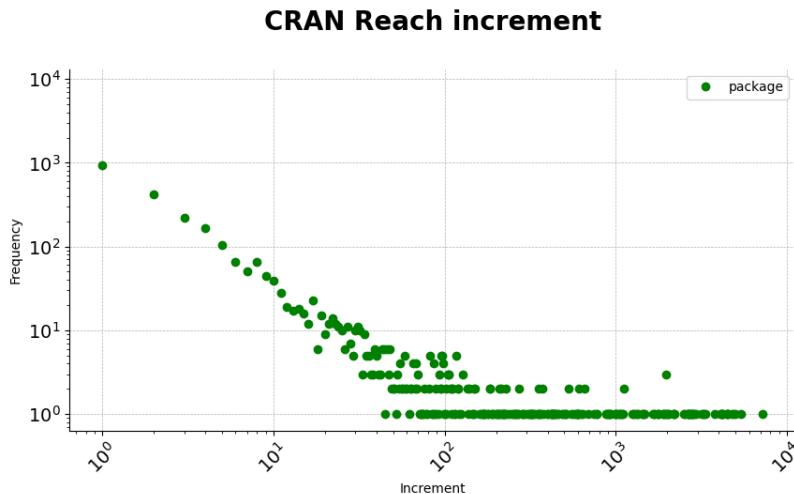


Figura 5.30: Incremento de *Reach*

5.3. La red de dependencias de Bioconductor

El repositorio *Bioconductor* para R es una plataforma científica de código abierto que ofrece herramientas y paquetes especializados para el análisis y la interpretación de datos genómicos. Surgió en 2001 para abordar desafíos específicos de la biología computacional y la genómica. *Bioconductor* es reconocido por su calidad, diversidad y enfoque colaborativo. Proporciona herramientas estadísticas y bioinformáticas para el análisis de expresión génica, variantes genéticas y más. Su importancia radica en su contribución al avance de la investigación genómica y promoción de la reproducibilidad y transparencia científica en el campo de la genómica y la biología computacional.

El análisis de *Bioconductor* es necesario como complemento al análisis de CRAN debido a que se enfoca específicamente en el análisis de un subconjunto del ecosistema R. Mientras que CRAN es el repositorio principal de paquetes para el lenguaje de programación R, *Bioconductor* se centra en ofrecer herramientas especializadas para el análisis de datos genómicos y biológicos.

Además, es importante destacar que, en el contexto de *Bioconductor*, se ha enfrentado el desafío de la falta de datos de referencia en *libraries.io*, un repositorio de información sobre paquetes de software. Esto ha llevado a la

necesidad de abrir nuevos caminos y proporcionar un conjunto de datos que no existía previamente. Al hacerlo, se está facilitando el análisis y el estudio de los paquetes de *Bioconductor*, y se está contribuyendo a la disponibilidad de datos valiosos para la comunidad.

El tamaño

La red de *Bioconductor* es una red relativamente pequeña^{5.10} en comparación con los valores de las otras redes que hemos analizado. Los valores de las métricas indican una red de dependencias de *Bioconductor* de tamaño bajo, con alta conectividad entre los nodos.

Medida	Valor
Number of nodes	3509
Number of edges	28320
Average degree	16.14
Average clustering coefficient	0.077

Tabla 5.10: Medidas de la red de dependencias de Bioconductor

El coeficiente de agrupamiento promedio de aproximadamente 0.0776 sugiere que la red tiene un nivel moderado de agrupamiento. Esto implica que algunos nodos están conectados en grupos o comunidades, pero no existe una fuerte tendencia hacia la formación de comunidades densamente interconectadas en toda la red. Esto puede indicar que existen diferentes grupos temáticos o funcionales dentro de la red de dependencias de *Bioconductor*, pero también existen conexiones entre estos grupos. Este coeficiente de agrupamiento es mucho menor que el de *CRAN*, que era de 0.15.

Respecto al grado medio podemos decir que es un poco mayor que el de *CRAN*, para el que teníamos un valor de 12.13.

El grado

A la vista de la distribución de *grado* 5.31, se puede observar una distribución de *ley de potencia*, lo cual es común en las *redes de dependencias*. En la gráfica, se puede apreciar que a partir de un *grado* de 50, el número de *nodos* comienza a reducirse de manera exponencial, quedando solo unos pocos representantes. Como era de esperar, se observan *hubs* o *nodos de alto grado*, con valores cercanos a 2000. Esta presencia de *hubs* indica la

existencia de elementos altamente conectados que desempeñan un papel crucial en la estructura de la red de dependencias.

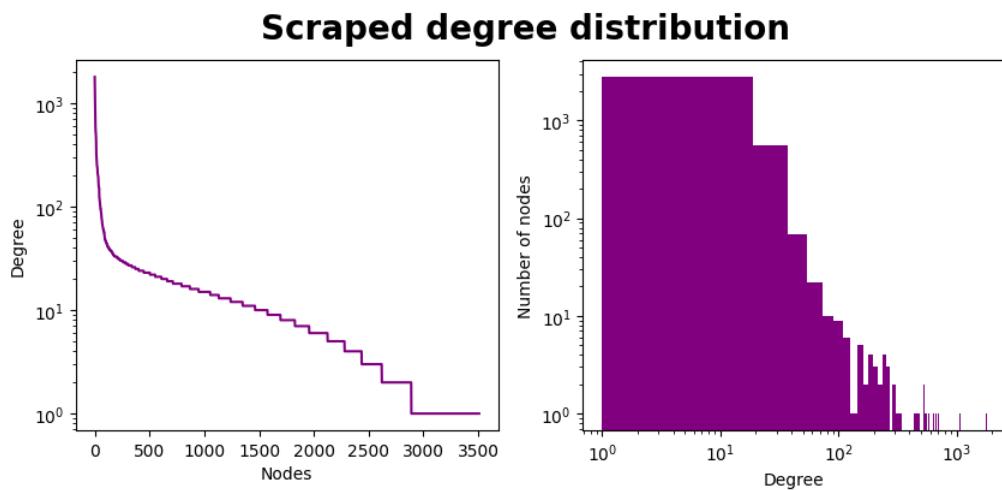


Figura 5.31: Distribucion de grado de la red de dependencias de Bioconductor

Figura 5.32: Distribucion de grado de la red de dependencias de Bioconductor

El grado de salida *Out degree*

La distribución del *grado de salida* en la red analizada presenta similitudes con la distribución de la red de CRAN. En este caso, el *grado de salida máximo* es notablemente inferior debido al menor tamaño de la red, pero la tendencia general es similar. Se puede apreciar que a partir de un *grado de salida* de 10, la frecuencia de individuos en la red se reduce significativamente. La mayoría de los nodos tienen un *grado de salida* inferior a 10, siendo estos el grupo más numeroso en la red.^{5.33}

En cuanto a los paquetes que representan el *mayor grado de salida*^{5.34} en la red analizada, se observa que la mayoría de ellos también aparecen en el *top* de la red de CRAN. Este hallazgo es coherente, dado que, como se mencionó anteriormente, Bioconductor es un subconjunto de los paquetes de R, y es esperable que estos paquetes tengan una importancia similar en ambos repositorios. Además, se observa que en este *top* se encuentran los paquetes más importantes y exclusivos de Bioconductor, como BiocGenerics o Biobase, entre otros. Esto nos indica que la red de Bioconductor está especializada en el ámbito de la bioinformática, ya que estos paquetes desempeñan un papel crucial en el análisis y procesamiento de datos biológicos. Su presencia

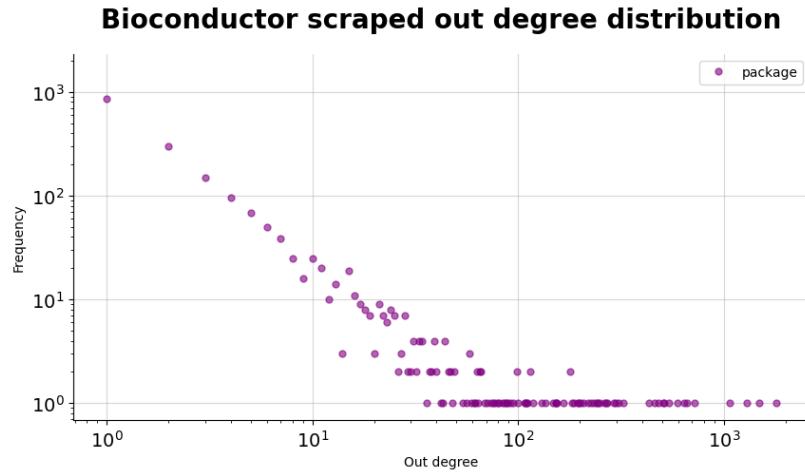


Figura 5.33: Distribucion de grado de salida de la red de dependencias de Bioconductor

en el *top* indica la importancia de la especialización de Bioconductor en este campo.

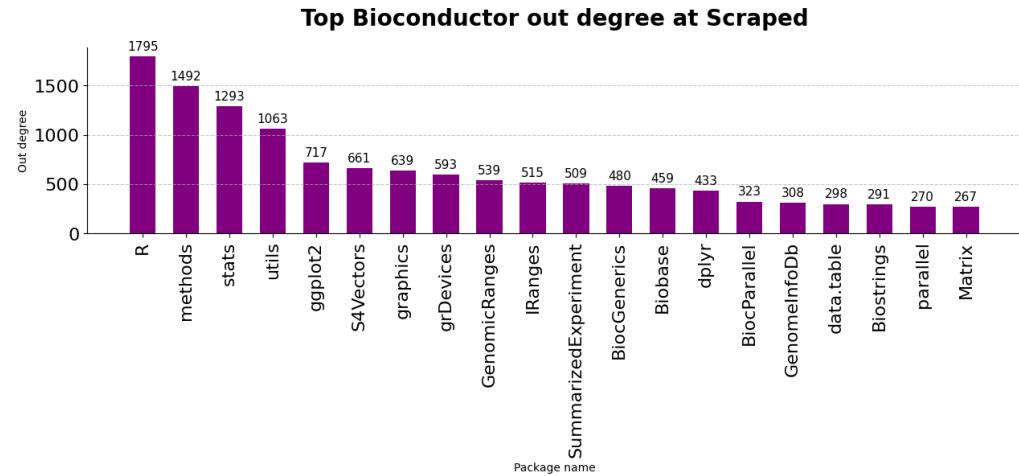


Figura 5.34: Top 10 de los paquetes con mayor grado de salida en la red de dependencias de Bioconductor

El grado de entrada *In degree*

La distribución de *in degree*^{5.35} en la red de Bioconductor muestra algunas diferencias con respecto a la red de CRAN. Se observa que hay un menor número de nodos en la red de Bioconductor en comparación con CRAN. Además, la frecuencia máxima de nodos con *in degree* ya no se encuentra en los valores 1 o 2, como ocurre en CRAN. En Bioconductor, se puede apreciar que hay más nodos con *in degree* en el rango de 10 a 20 en comparación con aquellos con un valor de 1.

Otro aspecto a tener en cuenta es que el *in degree* máximo en la red de Bioconductor tiene un valor de 85, mientras que en CRAN el valor máximo es de 50. Esto indica que en Bioconductor existen nodos con un mayor número de dependencias entrantes, lo que podría reflejar la complejidad y la interconexión de los paquetes en Bioconductor. Estas diferencias en la distribución de *in degree* evidencian las particularidades y características propias de la red de Bioconductor en comparación con la red de CRAN.

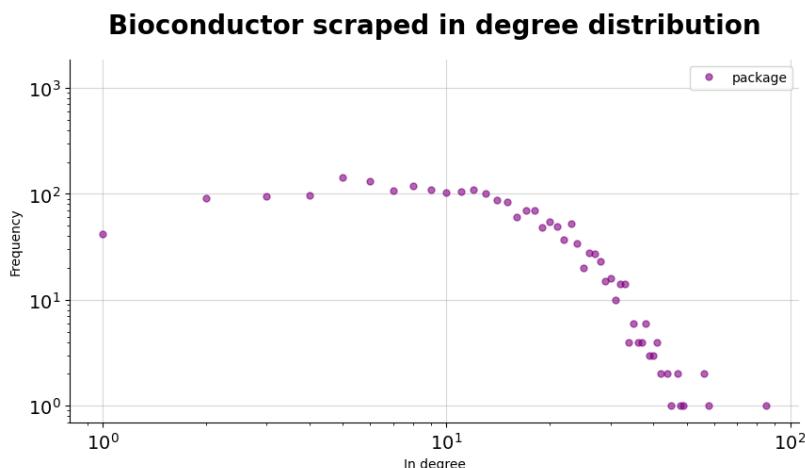


Figura 5.35: Distribucion de grado de entrada de la red de dependencias de Bioconductor

Los paquetes que se encuentran en el *top de grado de entrada*^{5.36} son aquellos que presentan un mayor número de dependencias en la red, sin tener en cuenta la transitividad de las conexiones. Esto implica que estos paquetes son altamente dependientes de otros paquetes en la red de Bioconductor en el primer nivel de profundidad del arbol de dependencias, lo que sugiere su importancia y relevancia en el ecosistema de Bioconductor.

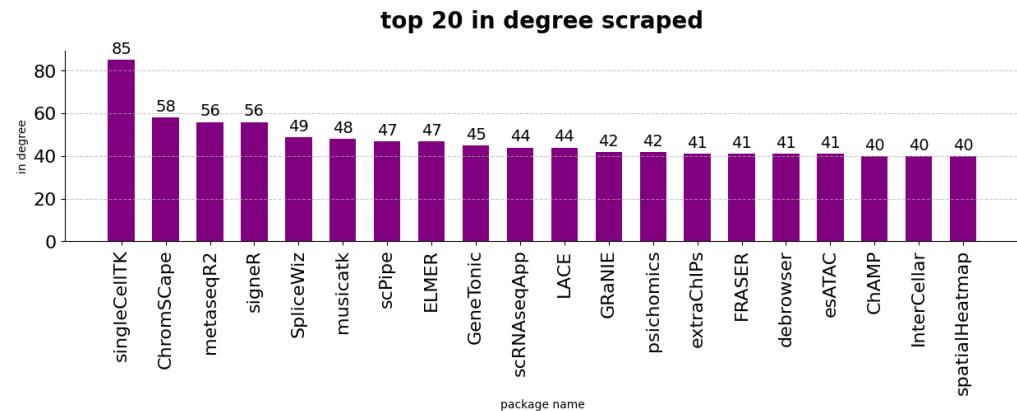


Figura 5.36: Top 10 de los paquetes con mayor grado de entrada en la red de dependencias de Bioconductor

El PageRank

La distribución de *PageRank* 5.37 en la red de *Bioconductor* no revela patrones claros y sencillos de analizar. Observando el gráfico, podemos determinar que la frecuencia máxima de nodos con el mismo valor de *PageRank* se sitúa aproximadamente entre 0.0002 y 0.0005. El rango normal de valores se encuentra entre 0.0002 y 0.002, aunque en algunos casos particulares se alcanzan valores de 0.003 o 0.004. Estos valores representan la importancia relativa de los nodos en la red, teniendo en cuenta tanto las conexiones directas como las conexiones indirectas a través de otros nodos en la red. Sin embargo, debido a la complejidad y tamaño de la red de *Bioconductor*, es necesario realizar un análisis más detallado y específico para comprender plenamente la distribución y significado de los valores de *PageRank* en esta red.

Respecto al top de dependencias con mayor *PageRank* 5.38, estos paquetes representan una vulnerabilidad significativa debido a su alto *in degree* y la interconexión entre ellos. Desde el punto de vista del autor, sería recomendable evitar depender de alguno de estos paquetes, ya que se encuentran en un entorno donde la probabilidad de vulnerabilidad es mayor. La dependencia de estos paquetes puede aumentar la propagación de posibles problemas o errores a lo largo de la red de *Bioconductor*. Por lo tanto, es esencial considerar alternativas y evaluar cuidadosamente las dependencias al desarrollar o utilizar aplicaciones basadas en esta red.

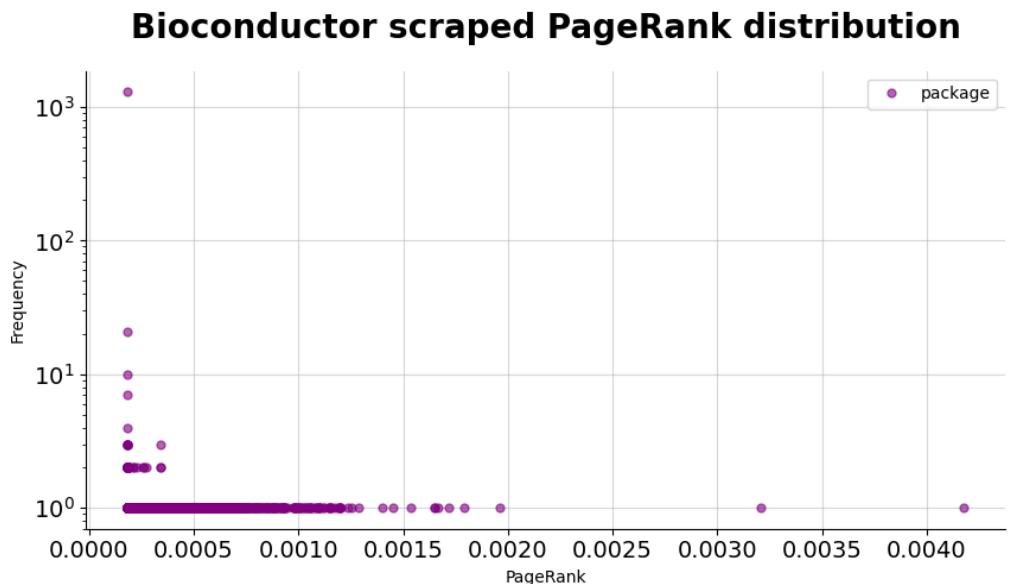


Figura 5.37: Distribucion de PageRank de la red de dependencias de Bioconductor

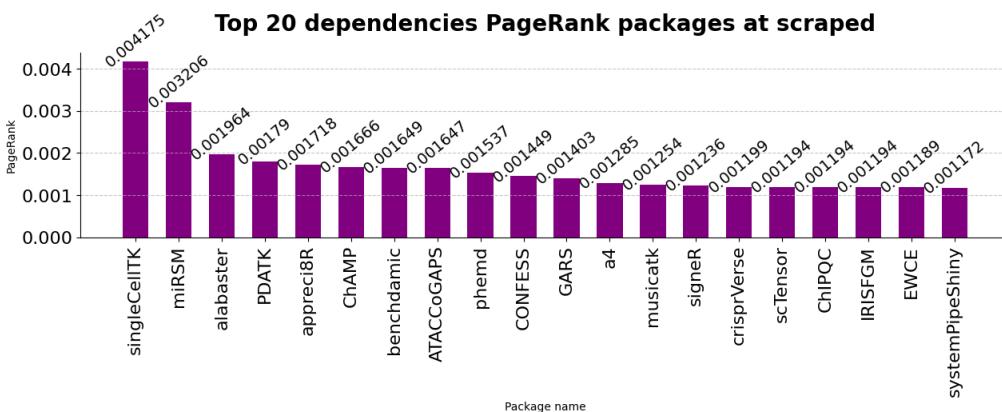


Figura 5.38: Top dependencias con mayor PageRank en la red de dependencias de Bioconductor

Realizando una inversionde la red de dependencias de Bioconductor, se obtiene la red de paquetes, desde este punto de vista el *PageRank* ahora esta ponderando la importancia de los paquetes en la red, es decir, aquellos paquetes que son dependencias de otros paquetes importantes, son considerados más relevantes.

rados como los mas importantes en la red. En la figura 5.39 se muestran los paquetes con mayor *PageRank* bajo este punto de vista de Bioconductor.

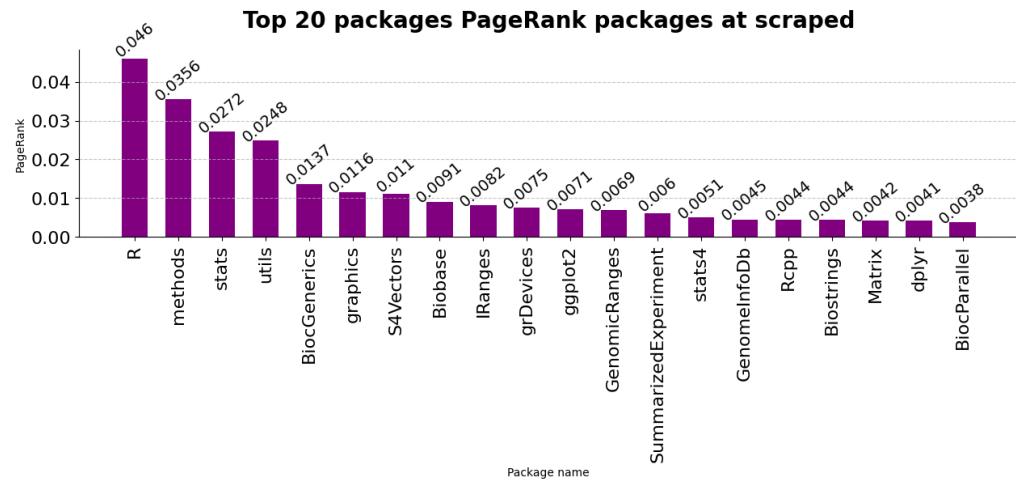


Figura 5.39: Top paquetes con mayor PageRank en la red de dependencias de Bioconductor

5.4. La red de dependencias de PyPI

En el ámbito del lenguaje de programación *Python*, nos enfocamos en el análisis de *PyPI* (*Python Package Index*). *PyPI* es un repositorio ampliamente adoptado en la comunidad de *Python* debido a su facilidad de uso, su naturaleza de código abierto y su extensa colección de paquetes.¹⁴

La facilidad de uso de *PyPI* se deriva de su diseño intuitivo y las funcionalidades que ofrece para la gestión de paquetes en *Python*. Los desarrolladores pueden acceder a *PyPI* como una fuente centralizada para descubrir, descargar e instalar una amplia variedad de paquetes y bibliotecas desarrollados por la comunidad.

En el análisis comparativo entre los datos proporcionados por *libraries.io* y los recolectados en este trabajo, se ha revelado una sorprendente tendencia en el número de paquetes presentes en *PyPI*. Se ha observado un aumento significativo en la cantidad de paquetes disponibles en *PyPI* en comparación con datos anteriores.

Este hallazgo sugiere un crecimiento notable en el ecosistema de paquetes de *Python* y evidencia el interés y participación de la comunidad en *PyPI*. Este aumento en el número de paquetes puede ser atribuido a diversos factores, como la creciente popularidad de *Python* como lenguaje de programación, el aumento en la adopción de *Python* en diferentes campos de aplicación y el creciente número de contribuyentes que comparten sus proyectos y soluciones a través de *PyPI*.

El análisis de los datos revela que aproximadamente el 57% de los paquetes que estaban presentes en el conjunto de datos antiguo de PyPI se han mantenido en el nuevo conjunto de datos¹⁵. Este porcentaje relativamente elevado indica que la red de paquetes en PyPI ha logrado mantener una cantidad considerable de paquetes de manera estable a lo largo del tiempo 5.4.

Es importante destacar que este conjunto de paquetes que se ha mantenido representa aproximadamente el 12.15 % del total de paquetes disponibles en PyPI en la actualidad. Esta proporción da una idea de la estabilidad relativa de la red, ya que una parte significativa de los paquetes ha logrado mantener su presencia en PyPI a pesar de los posibles cambios y actualizaciones¹⁶.

¹⁴Es importante mencionar que existen otros repositorios interesantes como *Conda*.

¹⁵Este porcentaje se basa en datos obtenidos experimentalmente.

¹⁶Los datos se refieren al momento de la última actualización y pueden estar sujetos a cambios futuros.

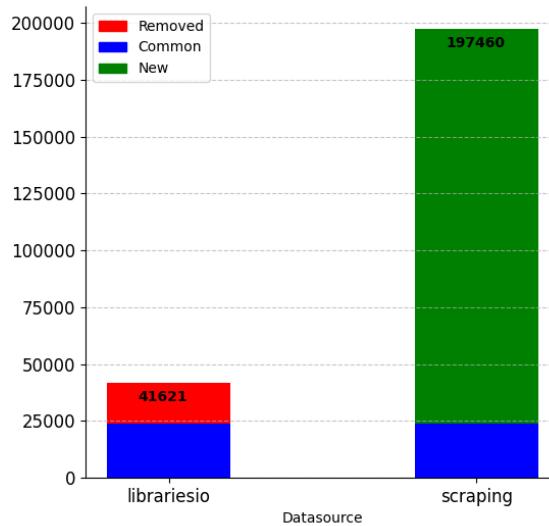
PyPI repository packages evolution

Figura 5.40: Comparacion de la cantidad de paquetes en PyPI entre los datos de libraries.io (2020) y los recolectados en este trabajo (2023).

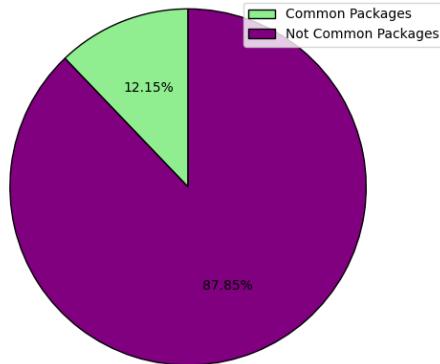
PyPI repository packages evolution

Figura 5.41: Paquetes comunes y no comunes entre los datos de libraries.io (2020) y los recolectados en este trabajo (2023).

Esta tendencia en la popularidad de Python se ve reflejada en las estadísticas realizadas por la empresa TIOBE [5.42¹⁷](https://www.tiobe.com/tiobe-index/python/).

¹⁷<https://www.tiobe.com/tiobe-index/python/>

Descripción	Cantidad
Paquetes en libraries.io	41,621
Paquetes en scraped	197,460
Paquetes comunes	24,001
Paquetes de libraries.io no disponibles en scraped	17,620
Paquetes en scraped que no estan en libraries.io	173,459

Tabla 5.11: Comparación de paquetes en PyPI entre los datos de libraries.io (2020) y los recolectados en este trabajo (2023).

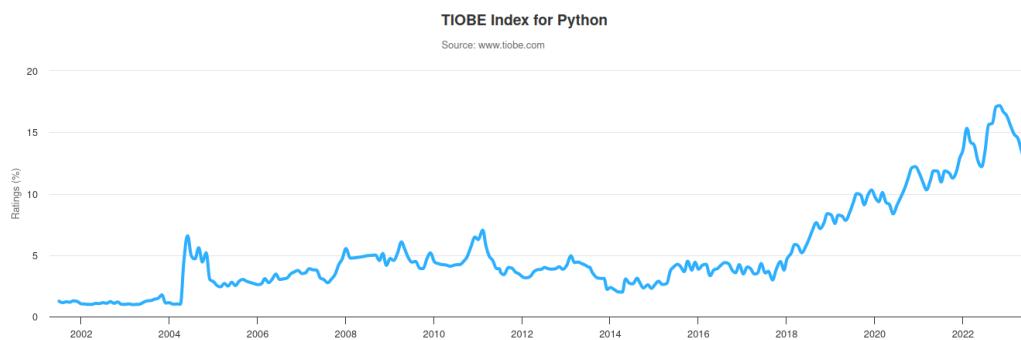


Figura 5.42: Popularidad de Python a lo largo del tiempo.

Grado de la red

Al examinar la distribución de grado en ambos conjuntos de datos, se observa que sigue una distribución de ley de potencias¹⁸.

Al analizar el gráfico, se evidencia que el grado máximo alcanzado por un paquete ha experimentado un incremento significativo. Tomando como punto de referencia el número de nodos con grado 1000, se puede apreciar una diferencia sustancial entre la red de *libraries.io* y los nuevos datos recolectados. Mientras que en *libraries.io* se registran menos de 10 paquetes con dicho grado, en los nuevos datos se han identificado aproximadamente 40 individuos¹⁹.

¹⁸La distribución de ley de potencias es una característica común en las redes de dependencias.

¹⁹Estos datos se basan en el análisis realizado en una fecha específica y pueden estar sujetos a cambios en el tiempo.

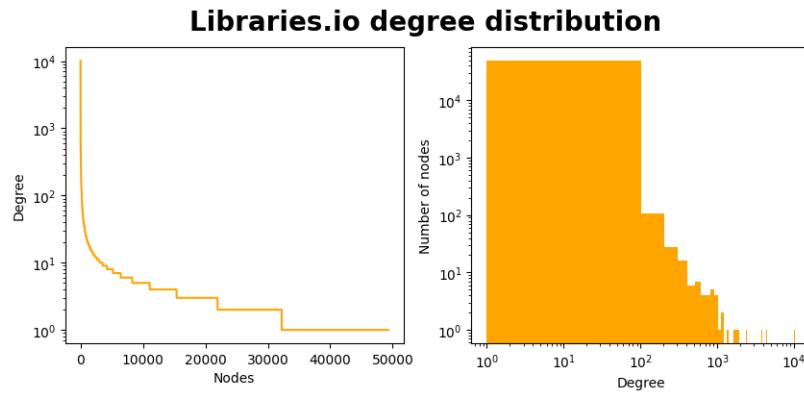


Figura 5.43: Distribucion de grado de PyPI para libraries.io.

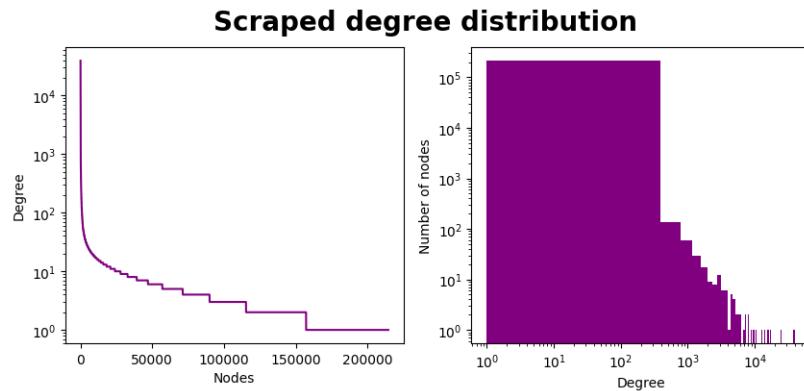


Figura 5.44: Distribucion de grado de PyPI para los datos recolectados en este trabajo (2023).

Además, al calcular el grado promedio de los grafos correspondientes a *libraries.io* y el nuevo conjunto, se obtiene un valor de 2.73 y 4.35, respectivamente. Estos valores indican que, en promedio, cada paquete en la red de *libraries.io* está conectado a alrededor de 2.73 otros paquetes, mientras que en los nuevos datos, cada paquete está conectado a aproximadamente 4.35 otros paquetes²⁰.

Estas estadísticas revelan cambios importantes en la estructura de la red de dependencias de paquetes. El incremento en el grado máximo y el aumento

²⁰Estos cálculos se realizaron utilizando una metodología específica y pueden variar dependiendo de la definición de conexión utilizada.

en el grado promedio indican una mayor interconectividad y complejidad en la red, lo cual puede ser atribuido al crecimiento y la evolución del ecosistema de paquetes en Python.

Es importante destacar que la distribución de ley de potencias y la presencia de paquetes con grados altos en la red tienen implicaciones significativas en términos de la propagación de dependencias y la influencia de ciertos paquetes en la comunidad²¹.

Grado de salida (*out degree*)

El *out degree* es una métrica que proporciona información sobre el número de dependientes de un paquete dado. En el contexto de *libraries.io*, analizando los datos, podemos identificar los paquetes que tienen más dependencias, es decir, los que están en el *Top* de las dependencias más utilizadas.

Los resultados obtenidos revelan una tendencia general en la cual las dependencias más populares han experimentado un aumento en su popularidad, siendo ahora requeridas por un mayor número de paquetes en la red de dependencias. En particular, se observa un incremento significativo en la popularidad de las bibliotecas *requests*, *numpy*, *pandas* y *pytest* en comparación con otros paquetes presentes en este ranking.

Estos hallazgos indican que estas bibliotecas han adquirido una mayor relevancia y utilidad en el desarrollo de proyectos y aplicaciones en el entorno de Python.

En este análisis de ranking, se observa que ciertos paquetes se han mantenido con respecto al ranking anterior, lo cual indica su relevancia a lo largo del tiempo. Estos paquetes incluyen *PyYAML*, *click*, *numpy*, *pandas*, *pytest*, *pyyaml*, *requests*, *setuptools* y *six*. Su presencia continua en el ranking sugiere que son dependencias fundamentales y ampliamente utilizadas en proyectos y aplicaciones de Python²².

Por otro lado, se identifican paquetes que han ascendido en el ranking en comparación con la clasificación anterior. Estos paquetes incluyen *black*, *coverage*, *flake8*, *matplotlib*, *odoo*, *python*, *scikit*, *scipy*, *sphinx*, *tqdm* y *typing*. Su ascenso en el ranking puede ser atribuido a su creciente popularidad

²¹Estas implicaciones pueden afectar la estabilidad, la modularidad y la confiabilidad del ecosistema de paquetes en Python.

²²La relevancia y utilidad de estos paquetes se basa en la percepción de la comunidad de desarrolladores de Python y puede variar dependiendo del contexto y los requisitos del proyecto

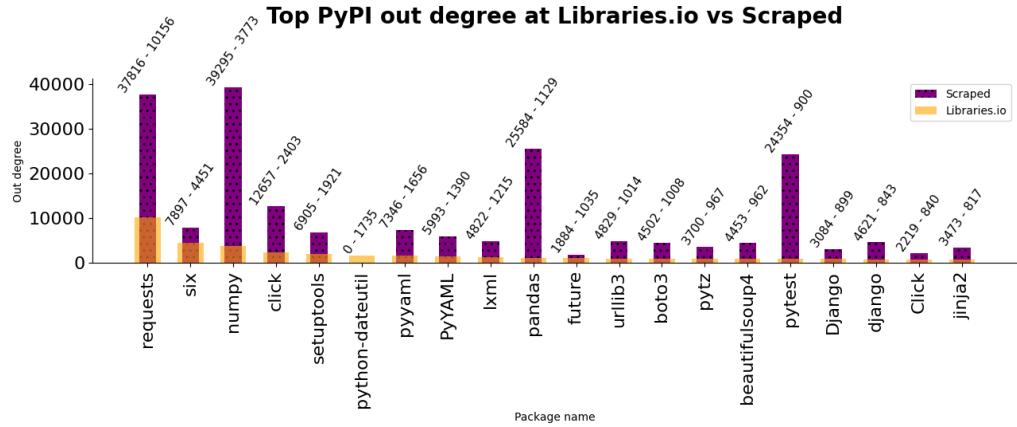


Figura 5.45: Top de paquetes con mayor grado de salida en PyPI para libraries.io (2020).

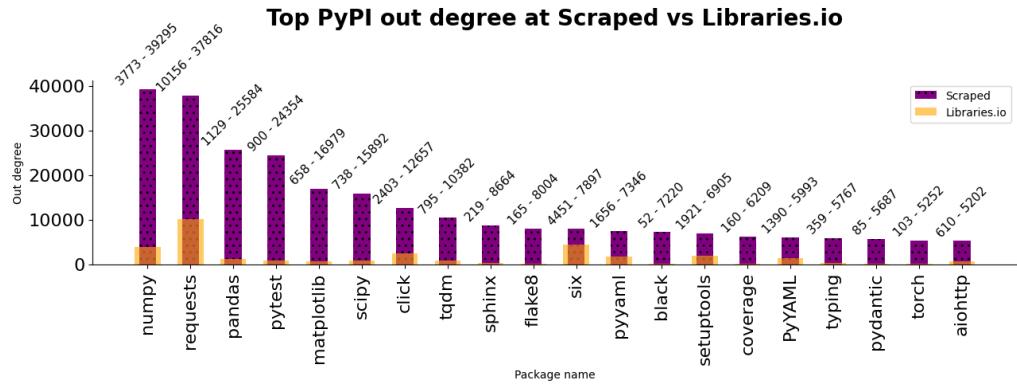


Figura 5.46: Top de paquetes con mayor grado de salida en PyPI para scraped (2023).

y utilidad en el desarrollo de proyectos de Python, ya que son bibliotecas ampliamente conocidas y utilizadas por la comunidad de desarrolladores²³.

Por último, se muestra la distribución de *out degree* para ambos conjuntos de datos 5.47 5.48

Al analizar los gráficos de las distribuciones de *out degree*, se observa una tendencia similar en ambos conjuntos. Se evidencia un incremento general

²³El ascenso en el ranking puede deberse a mejoras en funcionalidad, adopción en proyectos populares u otros factores que influyen en su popularidad

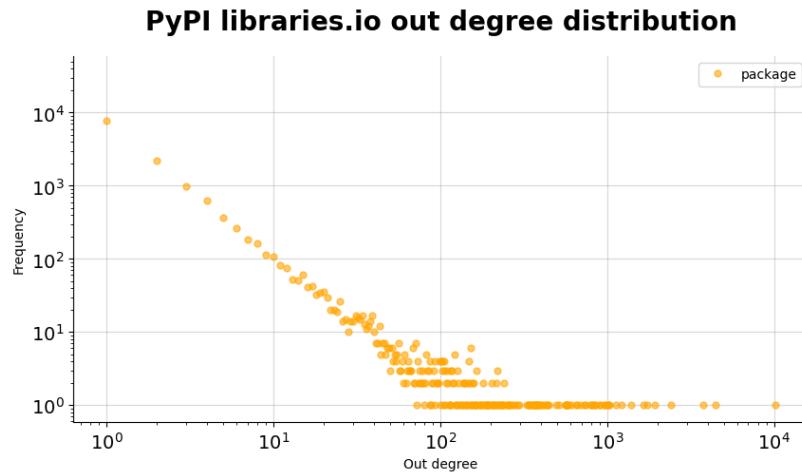


Figura 5.47: Distribución de *Out degree* para libraries.io (2020)

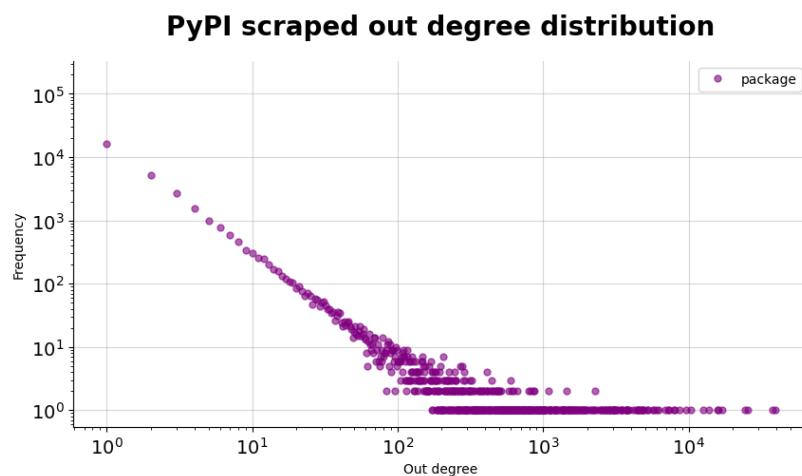


Figura 5.48: Distribución de *Out degree* para scraped (2023)

en el número total de paquetes, lo que indica un crecimiento continuo en el ecosistema de paquetes.

Se ha observado un aumento en el número de paquetes con un grado de salida bajo, lo que indica que estos paquetes tienen menos dependencias externas. Este fenómeno puede deberse a la introducción de paquetes más autónomos y autosuficientes en el ecosistema de Python, lo que reduce la necesidad de depender de otros paquetes para su funcionamiento.

Por otro lado, se ha identificado un incremento **5.49** en el grado de salida de los paquetes más populares. Esto indica que estos paquetes están siendo cada vez más utilizados como dependencias por otros paquetes en la comunidad de Python. Este aumento en el grado de salida de los paquetes populares puede ser atribuido a su funcionalidad ampliamente reconocida y popularidad.

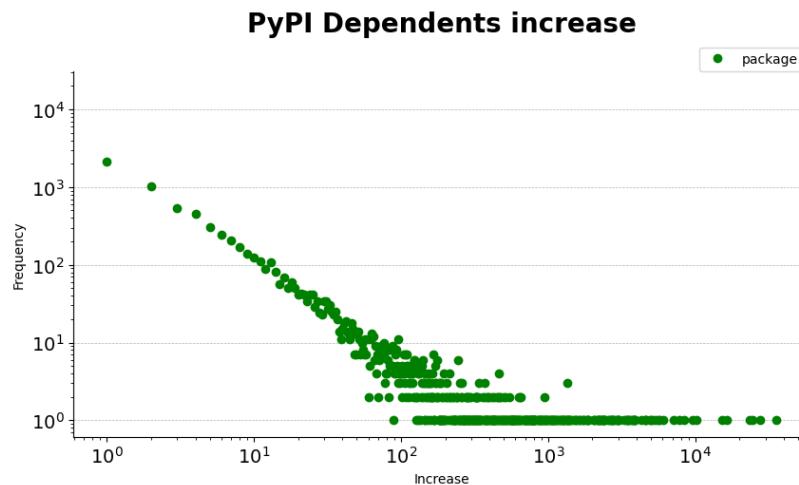


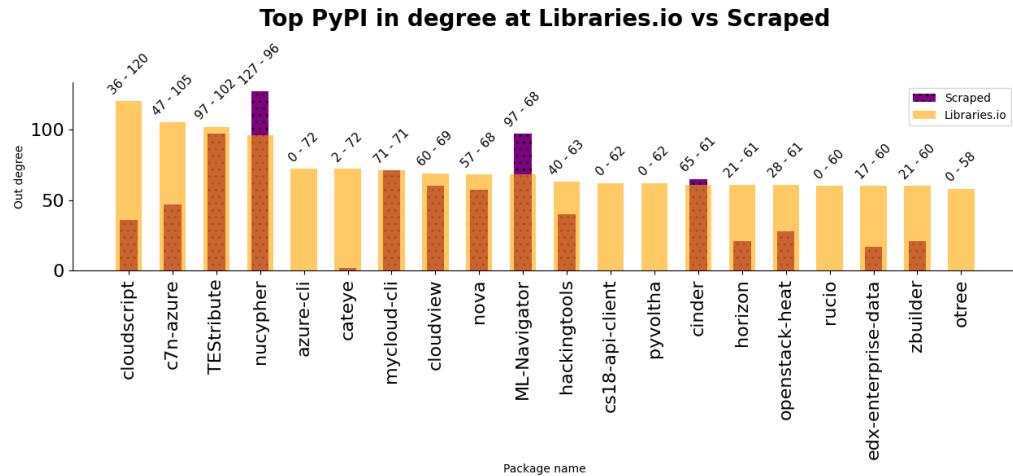
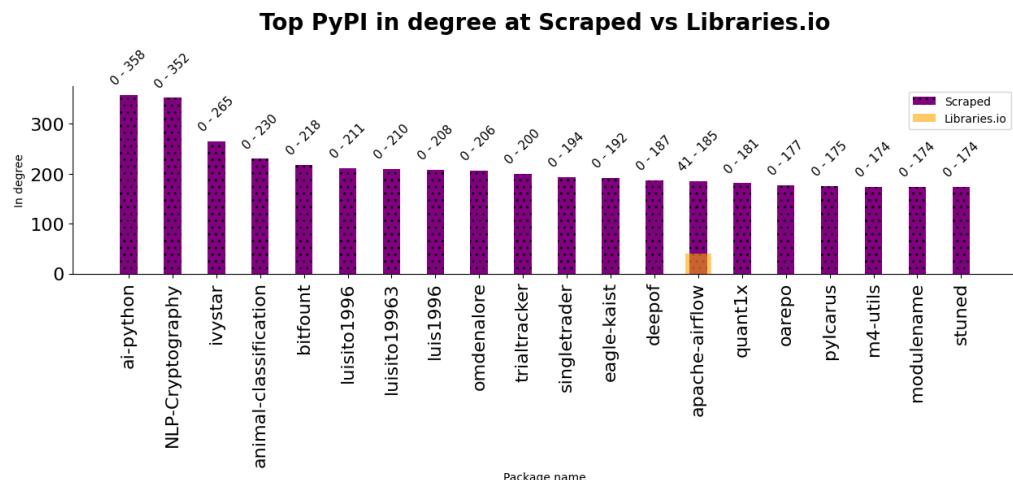
Figura 5.49: Incremento del *Out degree* en PyPI (2010-2023)

Grado de entrada (*In degree*)

Esta métrica nos da una idea del número de dependencias que tiene un paquete dado. Analizados los datos de *libraries.io* y representados sobre un grafico obtenemos los siguientes resultados: **5.50**

Se observa una tendencia decreciente en el número de dependencias entre los paquetes con mayor *In degree* dentro del conjunto de bibliotecas de *libraries.io*, lo cual sugiere una inclinación natural de los paquetes hacia una disminución de las dependencias requeridas. Además, se aprecia que una proporción significativa de estos paquetes, caracterizados por una elevada cantidad de dependencias, ha experimentado una desaparición, representando aproximadamente un 25 % de las instancias evaluadas. Por consiguiente, se puede inferir que, en general, la presencia de un alto número de dependencias no suele correlacionarse con la estabilidad de los paquetes en el repositorio.

Si realizamos este mismo análisis con el top 20 de los paquetes con más *In degree* en la actualidad **5.51**, podemos llegar a conclusiones similares. Se observa una tendencia decreciente en el número de dependencias de los

Figura 5.50: Top paquetes con mayor *In degree* en libraries.io (2020)Figura 5.51: Top paquetes con mayor *In degree* en scraped (2023)

paquetes más influyentes, lo cual sugiere una reducción en la cantidad de paquetes que dependen directamente de ellos. Esto puede indicar cambios en las estrategias de desarrollo, la aparición de alternativas o la evolución de la comunidad de desarrolladores.

Estos hallazgos resaltan la importancia de considerar tanto el In degree como el grado de salida de los paquetes al analizar la estabilidad y la evolución de la red de dependencias en el ecosistema de Python.

Como se puede apreciar, el 95 % de los paquetes pertenecientes a este conjunto presentan una ausencia de dependencias. Si profundizamos en el tema, podemos apreciar que estos paquetes son de reciente aparición. La falta de dependencias en los paquetes puede ser resultado de su diseño modular, el uso de bibliotecas internas o la falta de necesidad de dependencias externas.

Cabe destacar un caso particular, el paquete denominado *apache-airflow*²⁴, el cual ha experimentado un considerable aumento en el número de dependencias, pasando de 41 a 185. La explicación que se atribuye a este fenómeno es la incorporación de nuevas funcionalidades, dado que se trata de un paquete con cierta popularidad. No obstante, desde la perspectiva del autor de este Trabajo Final de Grado, se recomienda a los desarrolladores reducir al máximo este número de dependencias para mejorar su estabilidad²⁵.

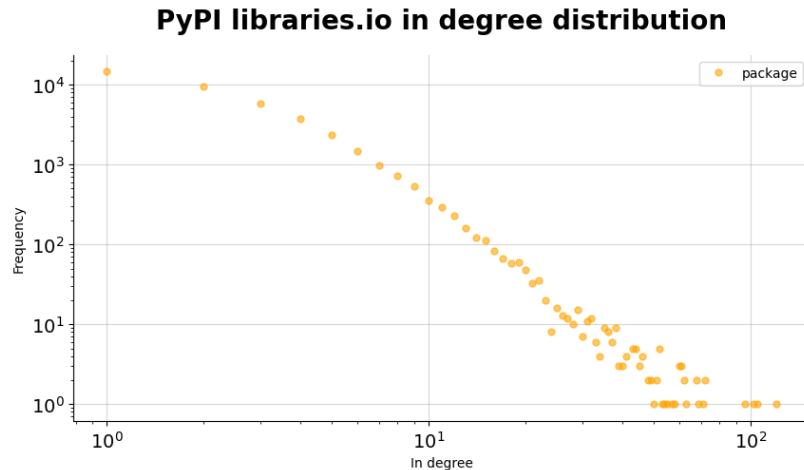


Figura 5.52: Distribucion del *In degree* en libraries.io (2020)

En el análisis de la distribución de *In degree*^{5.52}, se ha observado una alta frecuencia de nodos con un bajo grado, lo que indica la presencia de numerosos paquetes que no son utilizados como dependencias. Además, se ha identificado una clara tendencia descendente en la frecuencia a medida que aumenta el *In degree*. Esto claramente representa una *distribucion Power Law* [25].

²⁴<https://pypi.org/project/apache-airflow/>

²⁵El aumento en el número de dependencias puede aumentar la complejidad y la posibilidad de conflictos en el entorno de desarrollo. Se sugiere evaluar cuidadosamente las dependencias necesarias y buscar alternativas más ligeras o mejor optimizadas si es posible

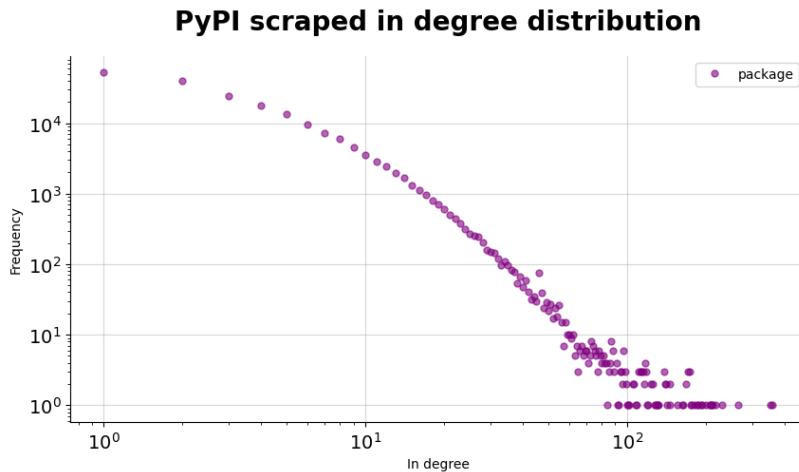


Figura 5.53: Distribucion del *In degree* en scraped (2023)

La disminución en la frecuencia se vuelve significativa a medida que se alcanza un *In degree* del orden de 10^2 , donde la frecuencia se reduce a un único nodo por caso. Esto implica que existe una disminución drástica en la cantidad de paquetes con un *In degree* alto, lo cual sugiere que son menos comunes aquellos paquetes que son ampliamente utilizados como dependencias por otros.

Si observamos la evolución 5.53, se observa una tendencia similar en ambos casos, aunque en el estado actual se evidencia un aumento en el número de nodos. La forma de la distribución se mantiene similar, pero se aprecia un considerable incremento en el *In degree*. Si consideramos la conclusión anteriormente obtenida, podemos constatar que también se cumple en este caso, dado que el incremento en la frecuencia implica una disminución en el grado de entrada.

Pagerank

La métrica de *PageRank*²⁶ Permite establecer un ranking de importancia respecto a los paquetes. A la vista de la distribución obtenida de la red de *libraries.io*, una conclusión que se puede extraer es que la mayoría de las dependencias no son consideradas importantes, ya que pertenecen al primer grupo, con una frecuencia considerablemente alta pero una baja relevancia

²⁶PageRank es un algoritmo utilizado para establecer un ranking de importancia en la web.

en términos de *PageRank*. Sin embargo, existe un grupo más reducido pero significativo que presenta una importancia media, pero son relevantes a nivel de que tienen múltiples enlaces provenientes de diferentes paquetes. Estas dependencias comunes desempeñan un papel crítico en la interconexión de los diferentes componentes de la red. Además, se destaca la presencia de un conjunto selecto de dependencias con un alto *PageRank*, lo que indica su gran popularidad y relevancia en la red de paquetes. En este grupo, las dependencias son enlazadas por muchas otras dependencias, pero no tienen dependencias propias.

En concreto, estas son las dependencias más importantes a tener en cuenta debido a que suelen ser común su aparición 5.54.

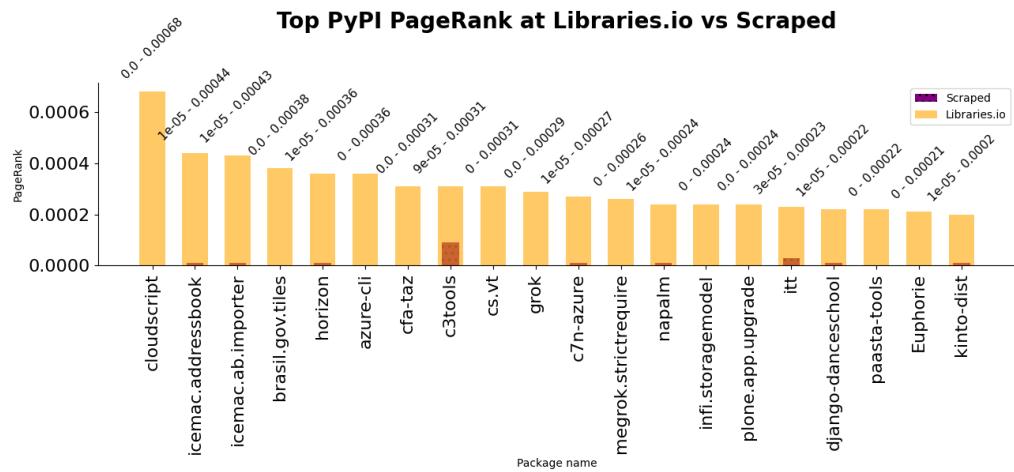


Figura 5.54: Top 20 *PageRank* en libraries.io (2020)

En términos de vulnerabilidad, los paquetes más críticos son aquellos que, si experimentan problemas o inestabilidad, pueden tener un impacto significativo en toda la red de dependencias. Una dependencia crítica con múltiples enlaces entrantes puede generar la propagación de errores o vulnerabilidades a través de los paquetes que dependen de ella²⁷.

Si visualizamos el número de dependencias transitivas 5.55 de estos paquetes, podemos obtener conclusiones más precisas. Existe un grupo de paquetes con un menor número de dependencias transitivas, lo que los hace

²⁷Una dependencia crítica es aquella que, al presentar problemas, puede afectar negativamente a otros paquetes que dependen de ella, causando errores o vulnerabilidades en cadena.

menos vulnerables en comparación con el resto. Además, tener un alto *PageRank* en estos paquetes implica que son más confiables en términos de dependencias²⁸.

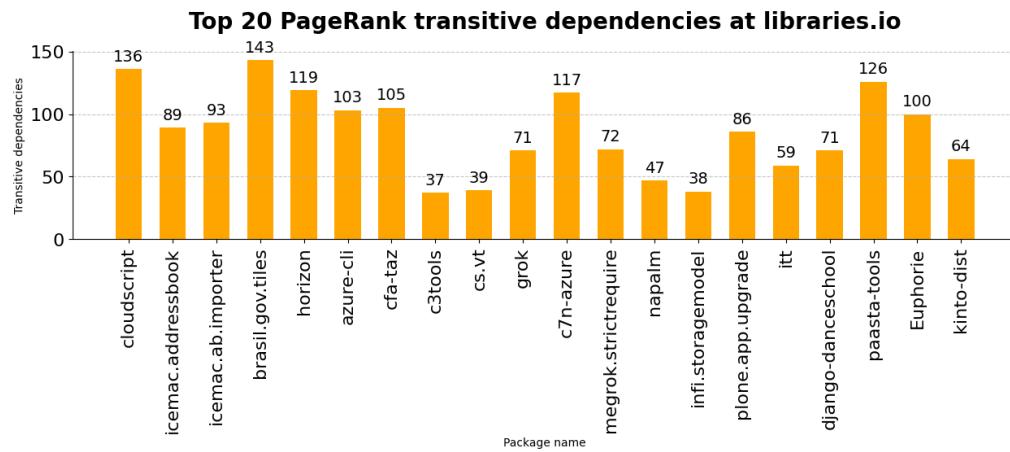


Figura 5.55: Dependencias transitivas del top 20 *PageRank* en libraries.io (2020)

También se observa que en este grupo de paquetes, las dependencias transitivas son en promedio más bajas²⁹.

Al analizar los resultados, se puede ver que los paquetes presentes en este grupo han experimentado una evolución considerable 5.56, con una reducción significativa en su *PageRank*³⁰. Esto se explica por la desaparición de algunos de estos paquetes, la aparición de otros nuevos que han reemplazado su importancia y la evolución de los propios paquetes hacia una mayor estabilización, lo que ha disminuido su vulnerabilidad.

Al examinar el nuevo conjunto de datos, se puede observar una tendencia similar a la anterior. El aumento en el número de paquetes se refleja en una alta frecuencia de paquetes con un bajo *PageRank*. Además, se observa una disminución general del valor del *PageRank* en la mayoría de la red. Esta

²⁸Un paquete con un alto *PageRank* y un bajo número de dependencias transitivas es considerado confiable y menos propenso a problemas de vulnerabilidad, ya que su estructura de dependencias es más simple y controlada.

²⁹El número de dependencias transitivas en este grupo de paquetes tiende a ser menor en comparación con otros grupos, lo que indica una estructura más ligera y menos compleja en términos de dependencias.

³⁰El *PageRank* de estos paquetes ha disminuido en comparación con mediciones anteriores.

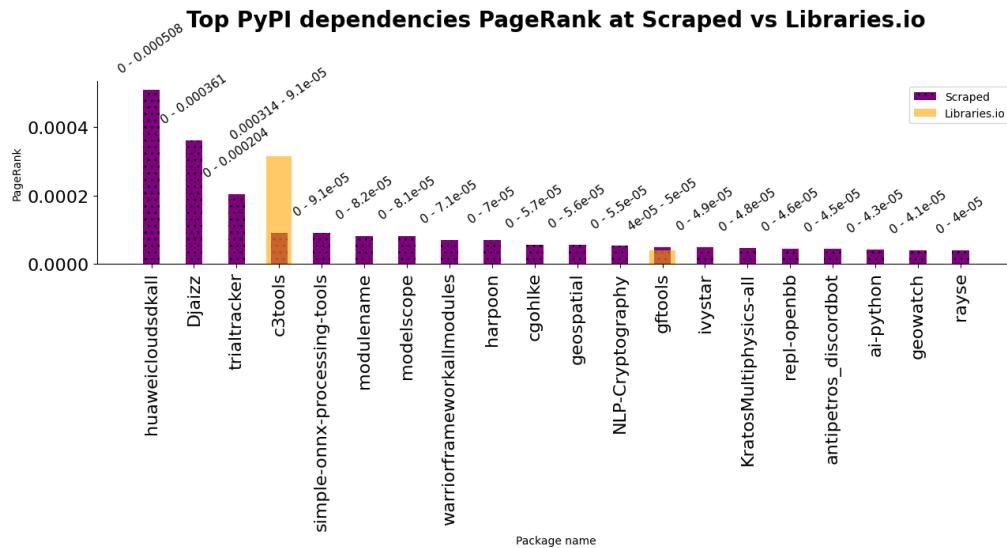


Figura 5.56: Top *PageRank* en scraped (2023)

disminución del *PageRank* puede interpretarse como una mejora en términos de vulnerabilidad.

Una conclusión que se puede extraer es que el crecimiento en el número de paquetes ha llevado a una mayor presencia de paquetes con un bajo *PageRank*. Esto sugiere que hay una mayor proporción de paquetes menos importantes en la red.

En este top se pueden observar las conclusiones previamente mencionadas en relación a la red de dependencias. La mayoría de los paquetes en este conjunto son nuevos, lo que ha llevado a una disminución del *PageRank* en comparación con el caso anterior. Sin embargo, es interesante destacar que algunos paquetes, como *c3tools* y *gftools*, se mantienen en el top, lo que sugiere que han resistido bien el paso del tiempo y podrían considerarse estables en la red, a pesar de tener una mayor probabilidad de vulnerabilidad.

Además, se puede observar que los tres paquetes principales en este conjunto tienen un *PageRank* considerablemente más alto que el resto. Esta diferencia en el *PageRank* podría indicar que estos paquetes son especialmente relevantes en la red de dependencias³¹.

³¹Los tres paquetes principales en este conjunto son altamente influyentes y desempeñan un papel crucial en la interconexión de otros paquetes en la red de dependencias.

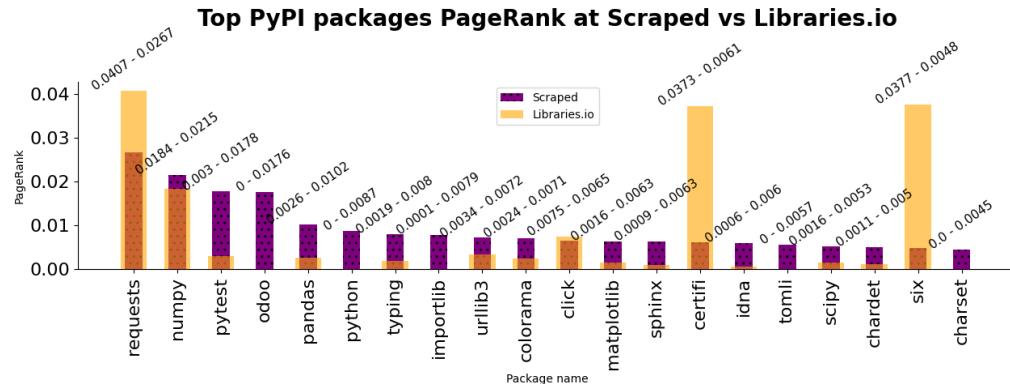


Figura 5.57: Top *PageRank* paquetes en scraped (2023)

Si invertimos el grafo de nuestra red de dependencias 5.57, podemos estudiar el *PageRank* desde el punto de vista de la relevancia del paquete en la red. Un alto *PageRank* implica que el paquete tiene una cantidad significativa de enlaces entrantes desde otros paquetes importantes. Esto sugiere que el paquete es visto como una fuente confiable de información o recursos dentro de la red. En otras palabras, es más probable que los otros paquetes dependan del paquete con un alto *PageRank* para obtener información o llevar a cabo determinadas tareas.

Un paquete con un alto *PageRank* puede ser considerado crucial en términos de la funcionalidad o el rendimiento de la red. Es probable que los otros paquetes dependan directa o indirectamente de él para llevar a cabo sus propias funciones o tareas.

Como se puede ver en el top que mostramos a continuación, aparecen los paquetes más conocidos y comúnmente usados de Python para los dos conjuntos de datos.

Impacto (Impact)

El impacto de un paquete se refiere al número de dependencias que se verían afectadas si ocurriera un defecto en ese paquete. Esta métrica podría utilizarse para evaluar la criticidad o importancia de un paquete en la red de dependencias y ayudar en la identificación de los paquetes que tienen un mayor impacto en el sistema en caso de fallos.

Tabla 5.12: Comparación entre paquetes obtenidos de libraries.io (2020) y scraped (2023) para la métrica *Impact*

libraries.io	scraped
six, 36757	numpy, 448177
certifi, 18739	six, 424014
requests, 17740	python, 422180
pyparsing, 14111	importlib, 420861
packaging, 13433	typing, 417287
appdirs, 12619	colorama, 416663
setuptools, 11803	matplotlib, 414520
python-dateutil, 9825	chardet, 413067
numpy, 7396	Cython, 412181
pytz, 6878	click, 411954

Resulta interesante apreciar que los paquetes del top siguen siendo prácticamente los mismos pese al notable incremento del impacto y que además esta métrica se relaciona bastante con el Pagerank a nivel de paquete.

PyPI Impact distribution for libraries.io

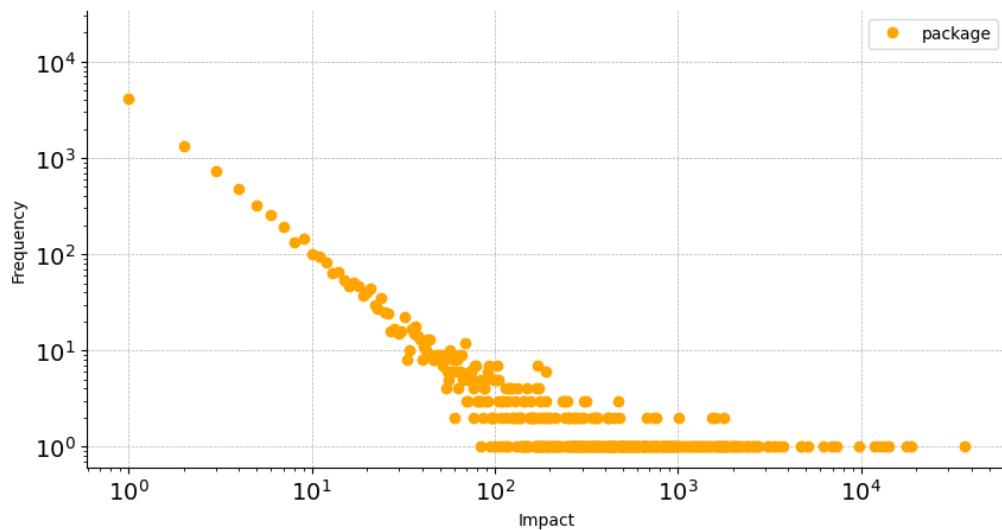


Figura 5.58: Distribución del impacto en la red de libraries.io (2020)

A la vista de la distribución del impacto en la red de libraries.io 5.58, se observa un patrón que se asemeja al comportamiento de la distribución del grado de salida (*out degree*). Se puede notar que existen numerosos paquetes con un impacto bajo, y la única explicación plausible es que estos paquetes no son ampliamente utilizados como dependencias en otros paquetes.

Es común que la mayoría de los paquetes tengan un impacto relativamente bajo, en el rango de alrededor de 10 paquetes. A medida que aumentamos el valor del impacto, la frecuencia de paquetes con un impacto alto disminuye significativamente.

Este patrón sugiere que la mayoría de los paquetes en la red de libraries.io no tienen una influencia crítica en las dependencias y, por lo tanto, su fallo o defecto tendría un impacto limitado en el sistema en general. Sin embargo, se identifican ciertos paquetes cuyo impacto es notablemente mayor, lo cual indica que son cruciales y tienen una influencia significativa en las dependencias.

PyPI Impact distribution for scraped

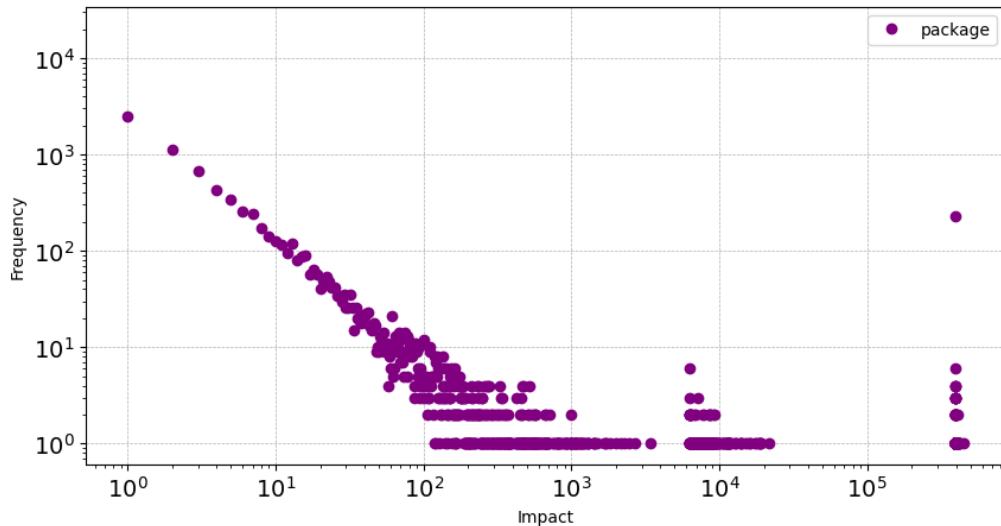


Figura 5.59: Distribución del impacto en la red scraped (2023)

Al evaluar el estado actual de la red, se observan cambios significativos en la distribución del impacto, que muestra similitudes con la distribución

del grado de salida (*out degree*), aunque también presenta variaciones y la formación de grupos con tendencias similares.

En particular, se ha observado un considerable aumento en el impacto general de los paquetes en la red. Ahora se identifica la presencia de un número considerable de paquetes con un impacto del orden de 10^2 , lo cual indica que su influencia en las dependencias ha aumentado significativamente.

Además, se distingue un segundo grupo más reducido de paquetes con un impacto alto, en el orden de 10000. Este grupo de paquetes merece especial atención debido a su impacto significativo en la red de dependencias.

Asimismo, se ha identificado otro grupo no existente anteriormente que resulta notable debido a su impacto elevado.

Al analizar el incremento del impacto en la red de dependencias, se observa una distinción entre dos casos. Por un lado, hay paquetes que han experimentado una disminución en su nivel de impacto o han mantenido un grado de *vulnerabilidad* estable a lo largo del tiempo. Por otro lado, existen paquetes que han experimentado un aumento significativo en su impacto.

Es notable que estos paquetes que han experimentado un aumento considerable en su impacto están interrelacionados y forman parte de componentes altamente conectados dentro de la red. Esta *interconexión* entre los paquetes permite un aumento grupal del impacto, amplificando así la magnitud de las consecuencias en caso de fallos o defectos.

Tabla 5.13: Comparación del incremento del impacto para de libraries.io (2020) y scraped (2023)

Paquete	libraries.io	scraped	incremento
numpy	7396	448177	440781
importlib	24	420861	420837
colorama	1795	416663	414868
typing	2434	417287	414853
matplotlib	748	414520	413772
Cython	75	412181	412106
chardet	1707	413067	411360
BeautifulSoup4	75	411391	411316
genshi	5	411290	411285
cssselect	260	411490	411230

A la vista del top de incremento del impacto se aprecia similitud entre los paquetes seleccionados, los cuales resultan muy familiares para los desa-

rrolladores que usamos el lenguaje Python ya que son paquetes muy usados en casi todo tipo de software.

Si analizamos el decremento se puede ver que no es tan acentuado como el incremento, no podemos sacar muchas conclusiones de ello más que estos paquetes han disminuido el número de dependencias transitivas que poseían, simplemente quedémonos con observar la tendencia y ver qué paquetes han sido los más afectados. [5.4](#)

Paquete	librariesio	scraped	incremento
python-dateutil	9825	0	-9825
importlib-metadata	4677	0	-4677
backports.functools-lru-cache	1944	0	-1944
async-timeout	1828	0	-1828
asn1crypto	2503	817	-1686
oslo.i18n	1503	0	-1503
futures	2277	816	-1461
oslo.utils	1242	0	-1242
singledispatch	1213	87	-1126
pyasn1-modules	1101	0	-1101

Tabla 5.14: Disminución del impacto en libraries.io (2020) y scraped (2023)

Reach

La métrica llamada *Reach*, que se refiere a la vulnerabilidad frente a fallos en una red de paquetes, se utiliza para medir el alcance de los paquetes afectados por un fallo aleatorio en la red. Se define como la media aritmética del alcance de los nodos en la red.

La vulnerabilidad de la red se cuantifica al calcular el número esperado de paquetes comprometidos por un fallo aleatorio, asumiendo que las probabilidades de fallo son independientes y siguen una distribución uniforme.

A nivel de paquete se refiere al número de paquetes que se verían afectados por un fallo en un paquete o alguna de sus dependencias transitivas, es decir el número de sucesores transitivos de un paquete más 1.³².

Bajo esta definición, al analizar el top 20 de paquetes con el mayor *Reach* en una red compuesta por aproximadamente 40000 nodos, resulta

³²El alcance a nivel de paquete se define como el número de paquetes que se verían afectados por un fallo en un paquete o alguna de sus dependencias transitivas.

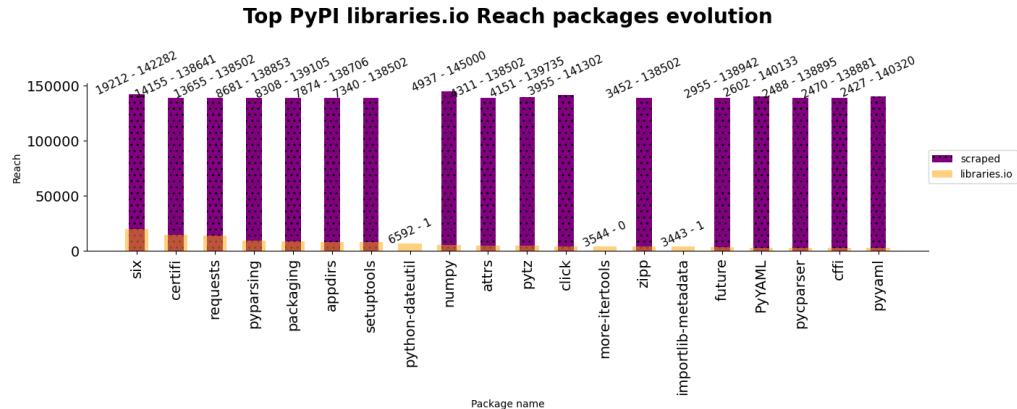


Figura 5.60: Top Reach en libraries.io (2020)

llamativo observar que algunos paquetes presentan un *Reach* tan elevado. Además, es importante destacar que estos paquetes se encuentran entre los más populares y utilizados en Python, lo cual justifica el valor alcanzado. Sin embargo, desde el punto de vista de la vulnerabilidad de la red, resulta preocupante, ya que un fallo en alguno de estos paquetes representaría un peligro significativo.

En relación a estos paquetes, en el estado actual de la red, resulta sorprendente el incremento que han experimentado en su alcance. Este incremento puede ser explicado por el crecimiento en el número de nodos de la red. Estos valores destacados para los paquetes en cuestión nos proporcionan una idea clara de la vulnerabilidad que introduce su presencia en la red. Si consideramos que *PyPI* actualmente cuenta con aproximadamente 200000 paquetes, un fallo en alguno de estos paquetes que se encuentran en el *top* del ranking tendría un impacto considerable en la integridad de la red.

Si comparamos las distribuciones de *reach* para los dos conjuntos de datos, podemos ver que tienen una tendencia similar a la distribución de *grado de salida*.

El nuevo conjunto de datos muestra la existencia de tres grupos distintos. El primer grupo presenta un valor de *Reach* bajo, lo que indica que no hay un nivel significativo de vulnerabilidad. En el segundo grupo, el valor de *Reach* se sitúa en el orden de 10000, lo cual representa un riesgo mayor. Aunque el número de paquetes pertenecientes a este grupo no es excesivamente alto, es importante tenerlo en cuenta debido a su nivel de vulnerabilidad. Por último, el tercer grupo se caracteriza por tener un valor de *Reach* muy alto.

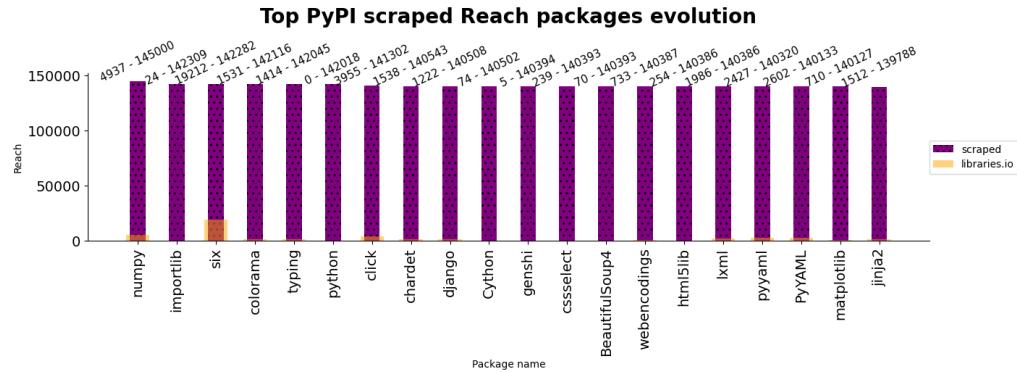


Figura 5.61: Top Reach en scraped (2023)

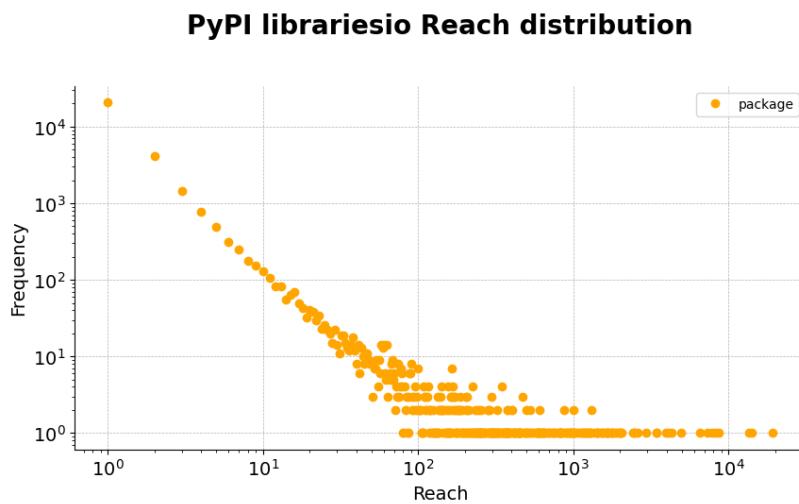


Figura 5.62: Distribución del Reach en libraries.io (2020)

Según mi interpretación, estos dos grupos de alto Reach representan nodos pertenecientes a componentes fuertemente conexos y son en los que habría que poner el foco para proteger la estabilidad de la red.

En relación al incremento del Reach, se pueden identificar dos tendencias distintas. En la primera tendencia, se observa que el Reach se mantiene relativamente estable, con fluctuaciones dentro de un rango aproximado de $\pm 15,000$. Por otro lado, el segundo grupo exhibe un notable aumento en el valor del Reach. Un fallo en un paquete perteneciente a este grupo podría generar graves problemas en la red. Este incremento puede ser atribuido al crecimiento en el número de nodos de la red. Como resultado de

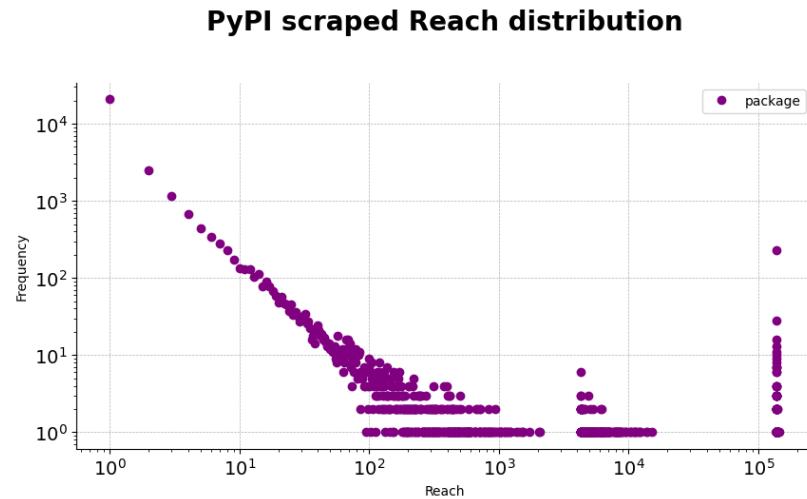


Figura 5.63: Distribución del Reach en scraped (2023)

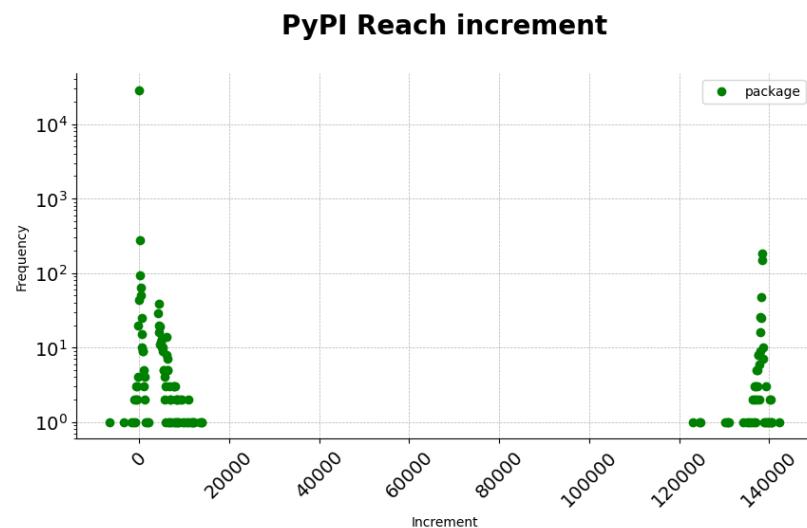


Figura 5.64: Incremento del Reach (2020-2023)

este crecimiento, han surgido dependencias transitivas que han contribuido significativamente a este aumento considerable en el Reach.

Componente fuertemente conexo

En un *componente fuertemente conexo*, todos los nodos están directa o indirectamente conectados entre sí. No importa si los caminos son directos

Size	Avg degree	Density	Diameter	Clustering coefficient	Transitive dependencies
283	8.890	0.015	14	0.196	206873
9	4.947	0.137	8	0.358	23807
8	3.000	0.214	6	0.222	6288
8	5.250	0.375	3	0.383	6784
8	4.750	0.339	5	0.498	6752
6	5.000	0.500	2	0.776	4536
6	4.666	0.466	3	0.470	4452
6	3.666	0.366	3	0.448	4440
5	2.800	0.350	4	0.000	3770
5	6.000	0.750	2	0.777	3855
5	3.200	0.400	4	0.386	3765
5	3.600	0.450	3	0.421	3755
4	3.000	0.500	3	0.406	5276
4	2.500	0.416	3	0.000	3016
4	3.000	0.500	3	0.406	3220
4	3.000	0.500	3	0.700	2988
4	3.000	0.500	3	0.406	3048
4	3.000	0.500	3	0.406	3064
4	4.500	0.750	2	0.800	2948
3	2.666	0.666	2	0.000	3780

Tabla 5.15: Componentes fuertemente conexos más grandes en scraped (2023)

o implican múltiples pasos a través de otros nodos, lo fundamental es que existe una ruta dirigida desde cualquier nodo al resto de los nodos del componente.

Bajo la red de libraries.io no se identifican componentes fuertemente conexos. Esto se debe principalmente al tamaño de la red, que es relativamente pequeño. Sin embargo, en el nuevo conjunto de datos, se ve claramente la existencia de componentes fuertemente conexos. [5.15](#)

A partir de estos datos, se pueden extraer varias conclusiones. Existen componentes fuertemente conexos de diversos tamaños, desde pequeños hasta muy grandes. Algunos componentes tienen una alta importancia medida por el pagerank del nodo principal. La densidad y el coeficiente de agrupamiento varían entre los componentes, lo que sugiere diferentes

patrones de conexiones. Además, el número de dependencias transitivas varía ampliamente en función del tamaño del componente.

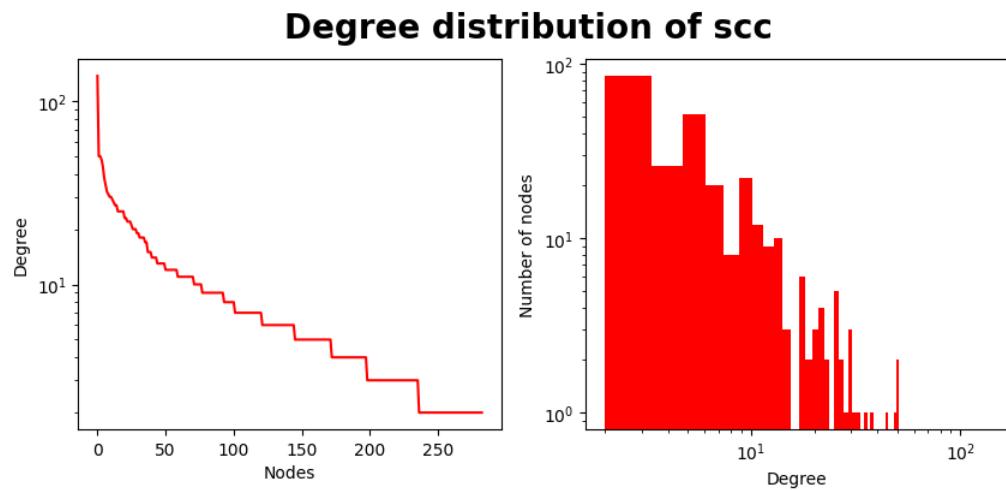


Figura 5.65: Distribución de grado del mayor componente fuertemente conexo (283 nodos) en scraped (2023)

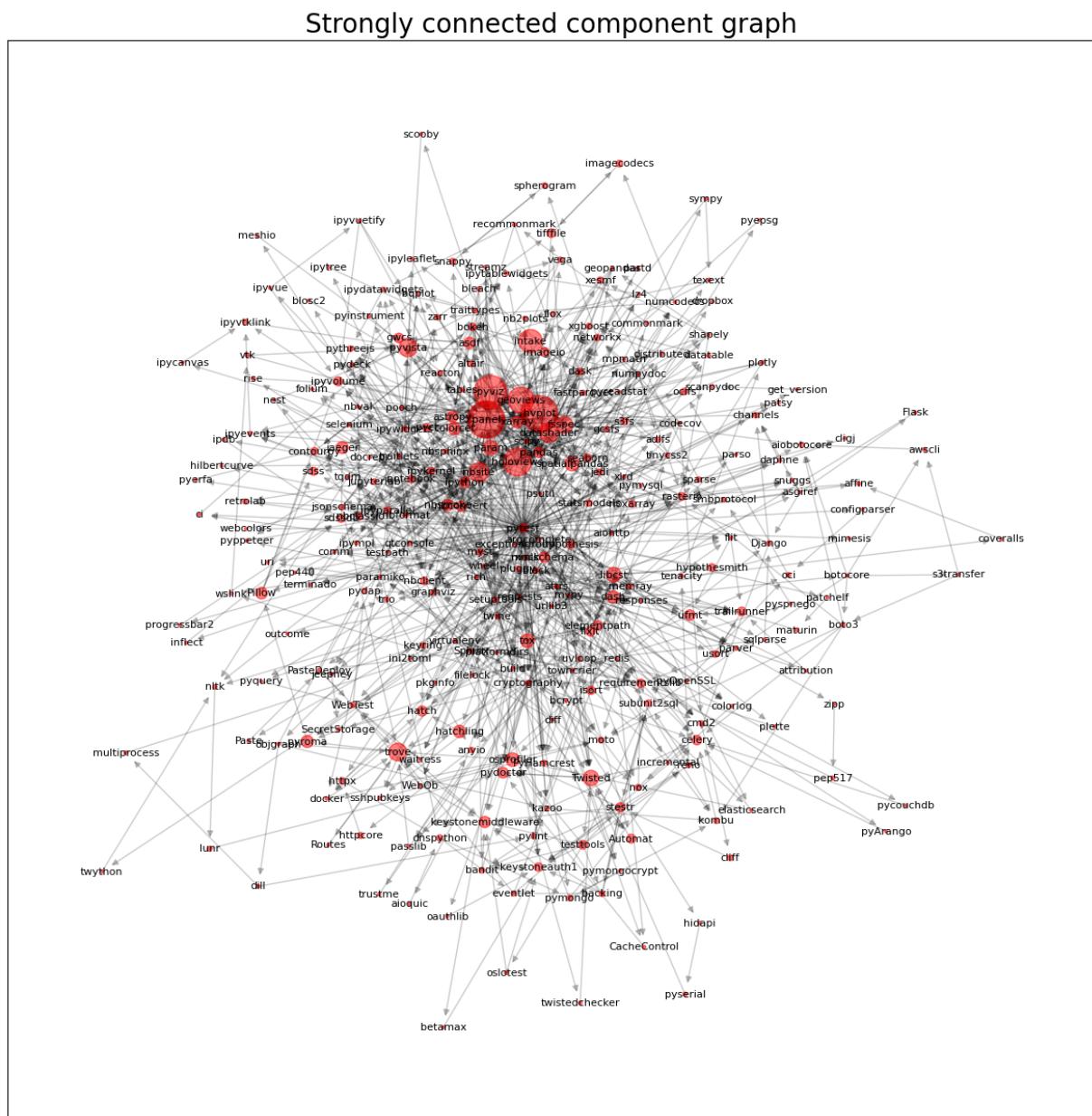


Figura 5.66: Mayor componente fuertemente conexo (283 nodos) en scraped (2023)

6. Trabajos relacionados

Los gestores de paquetes de software son herramientas esenciales que facilitan la reutilización y la construcción eficiente de sistemas de software. Estos gestores permiten a los desarrolladores y profesionales de la informática acceder a bibliotecas de código previamente desarrolladas, lo que agiliza el proceso de desarrollo al evitar tener que crear funcionalidades desde cero.

Dado que el software se distribuye a través de estos gestores de paquetes, es crucial para los investigadores y profesionales tener acceso a datos explícitos sobre las redes de dependencia de software. Estas *redes de dependencia* están compuestas por las relaciones entre los diferentes *paquetes de software* y pueden resultar opacas si no se cuenta con información precisa.

Para poder analizar y razonar sobre los ecosistemas y productos de software cada vez más complejos, los investigadores y profesionales dependen de conjuntos de datos públicos disponibles. Un ejemplo de ello es el conjunto de datos publicado en *libraries.io*, sin embargo, lamentablemente ha quedado desatendido y no se ha actualizado con regularidad.

Aquí es donde entra en juego el valor de este trabajo en particular, ya que proporciona un nuevo conjunto de datos que se centra en los gestores de paquetes *CRAN*, *Bioconductor*, *PyPI* y *npm*. Estos conjuntos de datos actualizados ofrecen una perspectiva más reciente y completa sobre la información disponible hasta la fecha. Esto es especialmente relevante para los autores de trabajos relacionados que han utilizado el conjunto de datos de *libraries.io* en sus investigaciones, ya que ahora tienen la oportunidad de actualizar sus estudios utilizando esta nueva información más actualizada. Con esto, se puede mejorar la calidad de la investigación y garantizar que los resultados y conclusiones reflejen con precisión el estado actual de los ecosistemas de software.

A continuación se presentan algunos de los trabajos relacionados que han utilizado el conjunto de datos de *libraries.io* en sus investigaciones.

- **On the impact of security vulnerabilities in the npm and rubygems dependency networks[28]**: Este artículo estudia empíricamente las vulnerabilidades de seguridad que afectan a los paquetes npm y RubyGems. Analiza cómo y cuándo se descubren y solucionan estas vulnerabilidades y cómo cambia su prevalencia con el tiempo. También analiza cómo los paquetes vulnerables exponen a sus dependientes directos e indirectos a vulnerabilidades.
- **Dependency solving is still hard, but we are getting better at it[2]**: Este artículo trata sobre la resolución de dependencias, que es un problema difícil (NP-completo) en todos los modelos de componentes no triviales debido a versiones mutuamente incompatibles de los mismos paquetes o conflictos de paquetes declarados explícitamente.
- **On the usage of JavaScript, Python and Ruby packages in Docker Hub images[27]**: Este artículo analiza empíricamente el uso de paquetes de terceros de JavaScript, Python y Ruby en imágenes de Docker Hub. Estudia cuán prevalentes, desactualizados y vulnerables son estos paquetes en imágenes de la comunidad que se basan en imágenes base de node, Python y Ruby.
- **Mining single statement bugs at massive scale[18]**: Este artículo presenta un enfoque para extraer y analizar bugs de una sola declaración de código fuente de proyectos de software.
- **Technical lag of dependencies in major package managers[22]**: Este artículo trata sobre el retraso técnico de las dependencias en los principales gestores de paquetes. Las bibliotecas de terceros utilizadas por un proyecto (dependencias) pueden quedar fácilmente desactualizadas con el tiempo, un fenómeno llamado retraso técnico.
- **Identifying critical projects via pagerank and truck factor[16]**: Este artículo trata sobre la identificación de proyectos críticos a través de PageRank y el factor de eje (Truck Factor). Recientemente, el equipo de código abierto de Google presentó el puntaje de criticidad, una métrica para evaluar la “influencia e importancia” de un proyecto en un ecosistema a partir de señales específicas del proyecto, como el número de dependientes, la frecuencia de confirmación, etc.

- **Identifying versions of libraries used in stack overflow code snippets[29]**: Este artículo trata sobre la identificación de versiones de bibliotecas utilizadas en fragmentos de código de Stack Overflow.
- **Intertwining Communities: Exploring Libraries that Cross Software Ecosystems[11]**: Este artículo trata sobre la exploración de bibliotecas que cruzan ecosistemas de software.

7. Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones

En esta conclusión se pretende dar una introducción al escaneo y análisis de repositorios software, destacando la dificultad asociada debido a la *elevada cantidad de datos*³³ y la necesidad de comprender la temática de los mismos. Se ha de tener en cuenta también el *tiempo de ejecución*³⁴ y el *gasto de recursos*³⁵ asociados a esta tarea. La computación en la nube es una solución a estos problemas, ya que permite el procesamiento de grandes volúmenes de datos de forma distribuida, escalable y eficiente.

Respecto a la persistencia de los datos, la elección entre una base de datos y archivos CSV para almacenar los datos de las redes de dependencias de paquetes depende de varios factores. Si se requiere una estructura de datos más compleja, consultas sofisticadas y escalabilidad a largo plazo, una base de datos es la opción más adecuada. Sin embargo, si se trabaja con conjuntos de datos más pequeños, se valora la simplicidad y la portabilidad, y no se necesitan capacidades avanzadas de gestión de datos, los archivos CSV pueden ser una solución práctica y eficiente. En nuestro caso, se ha optado por el uso de archivos CSV debido a la simplicidad de la estructura de datos y ser la estructura usada por OLIVIA para almacenar los datos

³³La cantidad de datos almacenados en los repositorios software puede ser enormemente extensa, lo que implica desafíos en términos de procesamiento y almacenamiento.

³⁴El tiempo requerido para realizar un escaneo y análisis exhaustivo de un repositorio puede ser considerable debido a la cantidad de operaciones y tareas involucradas.

³⁵El análisis de grandes volúmenes de datos puede requerir altos consumos de memoria RAM y espacio en disco, lo que puede afectar el rendimiento general del sistema.

de las redes de dependencias de paquetes. No obstante, se ha de tener en cuenta que el uso de archivos CSV puede implicar problemas, por lo que se recomienda el uso de una base de datos para el almacenamiento de los datos por las ventajas que ofrecen en términos de rendimiento y escalabilidad. Sobre el almacenamiento en GitHub, se ve necesario *particionar los datos* ya que existe una limitación en el tamaño máximo de archivo de 100 MB.

En cuanto a la extracción de datos, se ha de tener en cuenta que la extracción de datos de los repositorios software es una tarea compleja debido a la gran cantidad de información que se puede extraer de los mismos. Por ello, se ha de tener en cuenta que la extracción de datos debe ser *flexible* y *extensible*, ya que se pretende que sea capaz de adaptarse a diferentes repositorios software y a diferentes tipos de análisis.

En cuanto al análisis de los datos, el análisis de los datos es una tarea compleja debido a la naturaleza amplia y compleja de la ciencia de redes. Requiere el uso de técnicas y herramientas especializadas, así como un profundo conocimiento y comprensión de los principios de la ciencia de redes. Sin embargo, este análisis puede brindar información valiosa sobre las interdependencias y la estructura de los paquetes más importantes en los repositorios software, proponemos una visión general de los paquetes más importantes respecto a distintas métricas de centralidad y aportamos una visión de evolución de los paquetes al comparar los resultados de libraries.io con los obtenidos en este trabajo.

Por último, se propone el diseño de una herramienta capaz de llevar a cabo la recolección de datos, gestionando los posibles inconvenientes que puedan surgir durante el proceso. Se ha de tener en cuenta que el diseño de la herramienta debe ser extensible, ya que se pretende que sea capaz de adaptarse a diferentes repositorios software y a diferentes tipos de análisis y debido a su naturaleza open source y experimental, se ha realizado un esfuerzo adicional en la documentación del código y en el control de calidad del mismo.

7.2. Líneas de trabajo futuras

Mejoras en la herramienta

Con el objetivo de mejorar la herramienta desarrollada, se identifican diversas áreas de enfoque. En primer lugar, se propone realizar mejoras en el diseño y pulir detalles técnicos para optimizar su rendimiento y eficiencia. Esto implica revisar y refinar la arquitectura de la herramienta, identificar

posibles cuellos de botella y aplicar técnicas de optimización que permitan un procesamiento más rápido y una mayor escalabilidad.

Adicionalmente, se propone añadir soporte para nuevos repositorios, con el fin de ampliar la compatibilidad de la herramienta y permitir su aplicación en una variedad de entornos y plataformas. Esto implica adaptar la herramienta para interactuar con diferentes sistemas de control de versiones y repositorios, asegurando la interoperabilidad y facilitando su adopción por parte de un mayor número de usuarios.

Por último, es necesario replantearse la persistencia de datos. Esto implica evaluar la forma en que se almacenan y gestionan los datos recolectados durante el análisis de los repositorios software. Se pueden considerar opciones como el uso de bases de datos para un almacenamiento más eficiente y la implementación de mecanismos de respaldo y recuperación de datos para garantizar su integridad y disponibilidad a largo plazo.

Análisis de los datos

Se propone realizar un análisis más exhaustivo a partir de los datos recolectados. Esto implica la aplicación de técnicas de análisis de redes complejas para identificar patrones y tendencias en las redes de dependencias de paquetes. Ahí entra la comunidad científica, que puede aportar una visión más amplia y profunda de los datos que aportamos en este trabajo, así como una mayor comprensión de los mismos. Esto permitirá obtener información con la que actualizar sus investigaciones continuar con el estudio de las redes de dependencias de paquetes.

herramienta de visualización

Sería un punto interesante el desarrollo de una herramienta de visualización que permita representar gráficamente las redes de dependencias de paquetes y el cálculo de métricas asociadas. Esto implica el diseño e implementación de una interfaz gráfica de usuario que permita la interacción con la herramienta y la visualización de los resultados del análisis de los repositorios software. Esto permitirá a los usuarios explorar y comprender mejor las redes de dependencias de paquetes, así como identificar patrones y tendencias en los datos.

Bibliografía

- [1] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. Mining component repositories for installability issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 24–33. IEEE Press, 2015.
- [2] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. Dependency solving is still hard, but we are getting better at it. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 547–551. IEEE, 2020.
- [3] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 385–395, New York, NY, USA, 2017. Association for Computing Machinery.
- [4] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *Nature*, 406:378–382, 7 2000.
- [5] Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge University Press, Cambridge, 2016.
- [6] Christopher Bogart, Christian Kastner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. pages 86–89, 11 2015.
- [7] Paolo Boldi. How network analysis can improve the reliability of modern software ecosystems. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pages 168–172, 2019.

- [8] Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proceedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [9] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. When github meets cran: An analysis of inter-repository package dependency problems. 03 2016.
- [10] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.
- [11] Kanchanok Kanee, Raula Gaikovina Kula, Supatsara Wattanakriengkrai, and Kenichi Matsumoto. Intertwining communities: Exploring libraries that cross software ecosystems. *arXiv preprint arXiv:2303.09177*, 2023.
- [12] Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, January 2020.
- [13] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112, 2017.
- [14] Tom Mens, Alexandre Decan, and Maelick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. 02 2017.
- [15] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, Australia, 1998.
- [16] Rolf-Helge Pfeiffer. Identifying critical projects via pagerank and truck factor. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 41–45. IEEE, 2021.
- [17] Marton Posfai and Albert-Laszlo Barabasi. *Network Science*. Citeseer, 2016.

- [18] Cedric Richter and Heike Wehrheim. Tssb-3m: Mining single statement bugs at massive scale. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 418–422, 2022.
- [19] J. Sametinger. *Software Engineering with Reusable Components*. Springer Berlin Heidelberg, 1997.
- [20] Daniel Seto. dsr0018/olivia: OLIVIA - Open-source Library Indexes Vulnerability Identification and Analysis, nov 2022.
- [21] Daniel Seto-Rey, Jose Ignacio Santos-Martin, and Carlos Lopez-Nozal. Vulnerability of package dependency networks. *IEEE Transactions on Network Science and Engineering*, page 1–13, 2023. Cited by: 0.
- [22] Jacob Stringer, Amjad Tahir, Kelly Blincoe, and Jens Dietrich. Technical lag of dependencies in major package managers. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 228–237. IEEE, 2020.
- [23] Unknown. Cran and the isoband incident: Is your project at risk and how to fix it. <https://www.r-bloggers.com/2022/10/cran-and-the-isoband-incident-is-your-project-at-risk-and-how-to-fix-it/>. Acceso: 25 de junio de 2023.
- [24] Unknown. Issue comment: isoband. <https://github.com/r-lib/isoband/issues/33#issuecomment-1270766150>. Acceso: 25 de junio de 2023.
- [25] Wikipedia contributors. Power law — Wikipedia, the free encyclopedia, 2023. [Online; accessed 24-June-2023].
- [26] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR ’16, page 351–361, New York, NY, USA, 2016. Association for Computing Machinery.
- [27] Ahmed Zerouali, Tom Mens, and Coen De Roover. On the usage of javascript, python and ruby packages in docker hub images. *Science of Computer Programming*, 207:102653, 2021.
- [28] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, 27(5):107, 2022.

- [29] Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. Identifying versions of libraries used in stack overflow code snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 341–345. IEEE, 2021.