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 — 22 de junio de 2023
Tutor: Carlos López Nozal

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. profesor del departamento de nombre departamento, área de nombre área.

Expone:

Que el alumno D. Daniel Alonso Báscones, con DNI 71298886J, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado título de TFG.

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, 22 de junio de 2023

Vº. Bº. del Tutor:

D. Carlos López Nozal

Resumen

En este primer apartado se hace una **breve** presentación del tema que se aborda en el proyecto.

Descriptores

Palabras separadas por comas que identifiquen el contenido del proyecto Ej: servidor web, buscador de vuelos, android ...

Abstract

A **brief** presentation of the topic addressed in the project.

Keywords

keywords separated by commas.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

Uno de los principios más importantes de la ingeniería del software es aprovechar al máximo los componentes existentes [13]. 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), Packagist para PHP, 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 [**?**] 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 Schuleter, 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.

```
module.exports = leftpad;
function leftpad (str, len, ch) {
    str = String(str);
    var i = -1;
    if (!ch && ch !== 0) ch = ' ';
    len = len - str.length;
    while (++i < len) { str = ch + str; }
    return str;
}
```

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 [2] 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] [7] [4].

Es aquí donde entra la ciencia de redes 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 [9] [**?**] 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 çomplejas"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 [5], 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 [10], 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 [6], apareciendo un patrón de comportamiento que a priori se mantiene común en los repositorios de software. Por si eso no fuera suficiente, otro estudio [15] 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 [12]. 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 [3]. 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 [15] 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 [7] 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.

Las redes de dependencias de paquetes presentan características distintivas en comparación con las redes comúnmente estudiadas en la ciencia de redes, como las redes sociales, de información, tecnológicas o biológicas [11]. En primer lugar, los enlaces en estas redes pueden tener múltiples interpretaciones dentro de la misma red. Por un lado, actúan como canales de comunicación o difusión, similar a cómo ocurre en muchas redes sociales (lo cual explica, por ejemplo, la propagación de errores en la red). Por otro lado, representan requisitos necesarios para el funcionamiento de los nodos, pero de una manera diferente a otras redes tecnológicas. Tomemos como ejemplo una red eléctrica o de comunicaciones, donde los enlaces entrantes son necesarios para el funcionamiento de los nodos en la red. Sin embargo, en general, un solo enlace es suficiente para establecer una conexión entre un nodo y el conjunto de la red. En cambio, en las redes de dependencias, todos los enlaces son necesarios para el funcionamiento del nodo. Esto se debe a su naturaleza inherentemente transitiva, lo que significa que las relaciones son transitivas por definición, sin necesidad de que esta propiedad se exprese explícitamente en la estructura de la red. Esta característica es poco común en otros tipos de redes. Además, es posible considerar una red de dependencias como una red de información, donde los enlaces representan votos, en el sentido de que reflejan algún tipo de reconocimiento, al igual que los enlaces en Internet o las citas en las redes bibliográficas. En este sentido, las redes de dependencias comparten la propiedad de poder ser modeladas mediante una estructura acíclica sin introducir restricciones importantes, al igual que ocurre con las redes bibliográficas. La investigación sobre las redes de dependencias de paquetes es un campo relativamente nuevo y nuestra

comprensión de estas estructuras es limitada [9]. El conjunto de trabajos de referencia en este tema es aún reducido, y lo es aún más si nos centramos en aspectos específicos como el análisis de la vulnerabilidad, donde solo hemos encontrado publicaciones descriptivas. Hasta donde se sabe, no se ha propuesto un modelo teórico que permita avanzar en la comprensión de la vulnerabilidad de este tipo de redes, lo cual podría tener un impacto significativo en la seguridad y estabilidad del software a nivel global. Es necesario realizar más investigaciones en este campo para mejorar nuestra comprensión y desarrollar enfoques teóricos que aborden eficazmente los desafíos relacionados con la seguridad de las redes de dependencias de paquetes. En este trabajo, nos centraremos en los aspectos distintivos de las redes de dependencias de paquetes, como las múltiples interpretaciones de la red, las relaciones inherentemente transitivas y la posibilidad de modelarlas como redes acíclicas. Nuestro objetivo será estudiar la vulnerabilidad de estas redes frente a fallos. Definiremos la función de la red como la capacidad de proporcionar de manera ininterrumpida un conjunto de componentes de software reutilizables, de calidad y que funcionen según sus especificaciones. A través de nuestra investigación, intentaremos responder a preguntas como:

- ¿Hasta qué punto se ven afectados los proyectos software que emplean el repositorio ante la incidencia de fallos aleatorios (como por ejemplo la introducción de errores en los componentes)?
- ¿Cuál es la sensibilidad de una red determinada ante estas situaciones?
- ¿Existen estrategias óptimas para reducir el efecto perjudicial de los fallos en la red?

Por lo tanto, nuestra contribución consistirá en desarrollar un modelo teórico que permita analizar la vulnerabilidad frente a fallos en las redes de dependencias de paquetes. Este modelo se utilizará para implementar un conjunto de herramientas que ayudarán a responder las preguntas planteadas anteriormente. Estas herramientas se aplicarán a una captura de la estructura de dependencias de varios repositorios de paquetes reales. Por último, se realizará un análisis de los resultados obtenidos para obtener conclusiones relevantes. Los resultados de estos estudios proporcionan a la comunidad una visión muy poco conocida y novedosa desde el punto de vista de la redes de dependencias de paquetes, siendo como mínimo de interés académico para: Gestores centralizados de paquetes, para establecer políticas y procesos de control manuales o automáticos que mejoren la seguridad y estabilidad de

los repositorios. Desarrolladores de software en general, para valorar los diferentes riesgos introducidos por las dependencias empleadas en sus proyectos, y a los desarrolladores de paquetes en particular para comprender mejor su responsabilidad sobre el ecosistema. Desarrolladores de herramientas de calidad e integración continua, para definir cuantitativamente el concepto de vulnerabilidad en función del modelado de la red de dependencias de paquetes.

OLIVIA [14], 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 teoría de grafos. 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.

En la actualidad, OLIVIA está en proceso de ser publicado a nivel académico. Después de su desarrollo como proyecto de fin de carrera, se están realizando los esfuerzos necesarios para compartir sus resultados y contribuciones con la comunidad científica. Esta publicación permitirá 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, sentará 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.

Este TFG pretende ser una continuación de la labor realizada por su predecesor. 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 experimentales obtenidos.

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. Por lo tanto, se requieren herramientas especializadas y experiencia en análisis de datos para abordar este desafío.

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 puntos fuertes de este TFG se pretende 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 (Bioconductor y CRAN), python (Pypi) y javascript (NPM)

2. Objetivos del proyecto

- Realizar un análisis de la viabilidad de las diferentes estrategias de extracción de datos de los repositorios de paquetes, 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.

3. Conceptos teóricos

3.1. Defectos en productos software

Una especificación es una descripción de cómo se supone que un producto debe de estar construido para que funcione. Si hablamos de software, puede ser más o menos detallada, desde especificaciones matemáticas hasta manuales de usuario o descripciones funcionales. Un producto de software tiene un defecto cuando su comportamiento no es compatible con lo que se esperaba de él.

Cuando se construye un proyecto de ingeniería de software, se parte de una *Especificación de Requisitos de Software (SRS)*, que básicamente describe lo que se necesita del producto en general, tanto cosas funcionales como otros detalles complementarios, como temas legales, estándares de calidad y restricciones. Y si el diseño involucra el uso de componentes externos, la especificación de esos componentes tiene que ser compatible con la SRS del proyecto. Porque, en el contexto de ese proyecto, si algo no encaja bien, se considera como un defecto en ese componente, y eso a su vez afectaría el proyecto en sí. O sea, es como una cadena de defectos.

Entonces, cuando hablamos de defectos en el software, hay que tener en cuenta todo lo que pueda disminuir el valor del producto para sus usuarios. Ahí viene la parte más subjetiva, porque puede haber cosas que algunos usuarios consideren como defectos, como una biblioteca de cálculo que arroja resultados incorrectos, eso es un defecto objetivo, obviamente. Pero también pueden haber cambios en la licencia, en la usabilidad, en el cumplimiento de ciertos estándares de calidad o seguridad, o en la estructura de las interfaces externas, que para un usuario en particular también sean defectos.

Los defectos en el software pueden originarse por error o pueden ser el resultado de ataques informáticos que buscan sabotear o aprovecharse del software para obtener algún beneficio.

3.2. 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*.

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. 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.

3.5. Teoría de grafos

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 vértices*, conectados entre sí por *enlaces* llamados *aristas*. 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.

3.6. Grafos dirigidos y no dirigidos

En la teoría de grafos, existen dos tipos principales de grafos: los *grafos dirigidos* y los *grafos no dirigidos*[1]. Estas diferencias se refieren a la forma en que se establecen las conexiones entre los vértices del grafo.

[1]Nota: En la teoría de grafos existen otros tipos de grafos, pero nos enfocaremos en estos dos tipos principales en este contexto

Un *grafo no dirigido* es aquel en el que las aristas, que representan las conexiones entre los vértices, no tienen una dirección asociada. En otras palabras, la relación entre dos vértices es simétrica, lo que significa que si el vértice A está conectado con el vértice B, entonces el vértice B también está conectado con el vértice A. En este tipo de grafo, la comunicación o la relación entre los vértices se considera *bidireccional*. Un ejemplo común de un grafo no dirigido es una red social, donde los vértices representan personas y las aristas 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 aristas tienen una dirección asociada. Esto significa que la relación entre dos vértices puede ser asimétrica, es decir, el vértice A puede estar conectado con el vértice B, pero no necesariamente el vértice B está conectado con el vértice A. En este tipo de grafo, la comunicación o la relación entre los vértices se considera *unidireccional*. Un ejemplo común de un grafo dirigido es una red de transporte, donde los vértices representan ubicaciones y las aristas representan las rutas o los caminos que van en una dirección específica.

3.7. Métricas en la teoría de grafos

En general, las *métricas* son medidas o indicadores cuantitativos que se utilizan para evaluar y cuantificar diferentes aspectos de un objeto, sistema o fenómeno[2]. 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[3].

Grado

El *grado* de un vértice se refiere al número de aristas que están conectadas a ese vértice en particular[4]. El grado de un vértice es una métrica fundamental para comprender la conectividad y la importancia relativa de los vértices

[2]Las métricas también se conocen como medidas o indicadores de rendimiento

[3]Las métricas en la teoría de grafos son ampliamente utilizadas en campos como la informática, las redes sociales, la biología y la física, entre otros

[4]Nota: El grado de un vértice también se conoce como valencia

dentro de un grafo. En un grafo no dirigido, el grado de un vértice se calcula contando el número total de aristas incidentes a ese vértice. En un grafo dirigido, se distingue entre el grado de entrada (número de aristas entrantes) y el grado de salida (número de aristas salientes) de un vértice.

El grado de un vértice puede proporcionar información valiosa sobre la estructura y las propiedades del grafo en general. Por ejemplo, los vértices 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 vértices aislados (grado cero) o vértices 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 vértices 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 vértices 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*[5] 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.

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.

[5]Nota: Desarrollado por Larry Page y Sergey Brin

4. Técnicas y herramientas

Entorno de desarrollo

Se ha trabajado en un sistema operativo *Ubuntu* y utilizando *Visual Studio Code* (VSCode) como entorno de desarrollo integrado (IDE) preferido[6]. VSCode se destaca por su versatilidad y extensibilidad, lo que me permite personalizarlo y adaptarlo a mis necesidades específicas. Su amplia selección de extensiones me brinda herramientas adicionales y funcionalidades especializadas que enriquecen mi experiencia de programación.

El lenguaje de programación elegido es *Python*, reconocido por su facilidad de uso y amplia gama de bibliotecas y frameworks disponibles[7]. 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 llevar a cabo mis proyectos, cuento con un ordenador portátil de gama media equipado con un p rocesador *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 usar[8]. 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.

[6] La elección del sistema operativo y el IDE depende de las preferencias personales del autor

[7] Python se ha utilizado previamente en el Trabajo Final de Grado anterior

[8] Aunque el rendimiento puede variar dependiendo de los requisitos específicos del proyecto

A continuación se presenta una lista de los paquetes[9] utilizados, destacando sus funcionalidades:

- *Pandas*: 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*: 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.
- *Requests*: Una biblioteca que simplifica el manejo de solicitudes HTTP, permitiéndome realizar peticiones a servidores web y recibir respuestas de manera sencilla.
- *BeautifulSoup4*: Una biblioteca que se utiliza 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*: Una biblioteca que automatiza la interacción con navegadores web, lo que me permite realizar pruebas de aplicaciones web o realizar acciones específicas en páginas web de forma programática.
- *Networkx*: Una biblioteca para el análisis de redes y grafos. Proporciona herramientas para la creación, manipulación y estudio de estructuras de redes complejas.
- *Matplotlib*: Una biblioteca de visualización de datos en 2D que me permite crear gráficos y visualizaciones de datos de alta calidad.
- *Pybraries*: Una biblioteca que hace de wrapper de el API de *libraries.io*[10] para python.
- *Typing_extensions*: Una extensión del módulo typing de Python que proporciona funcionalidades adicionales para anotaciones de tipos en tiempo de ejecución.
- *pdoc*: Una biblioteca que me permite generar documentación automática a partir de mis archivos de código fuente.

[9] Paquetes de Python

[10] Libraries.io es una plataforma en línea que proporciona información y datos sobre diferentes bibliotecas y paquetes de software de código abierto

Jupyter Notebooks y Computación en la nube

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.[11]

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.[12]

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.[13]

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

[11] El análisis exploratorio de datos implica investigar y comprender los datos a través de la exploración visual, la estadística descriptiva y otras técnicas para obtener ideas y patrones clave.

[12] El procesamiento de grandes volúmenes de datos implica trabajar con conjuntos de datos masivos que pueden superar la capacidad de procesamiento de una sola máquina. La computación en la nube permite distribuir y escalar el procesamiento para manejar estos volúmenes de datos.

[13] La 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.

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.[14]

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.[15]

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.[16]

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

[14] El control de versiones es un sistema que registra y controla los cambios realizados en un proyecto a lo largo del tiempo. Permite realizar un seguimiento de las modificaciones, gestionar conflictos y recuperar versiones anteriores del código. Git es un sistema de control de versiones ampliamente utilizado en el desarrollo de software.

[15] Los problemas y las solicitudes de extracción son mecanismos utilizados en GitHub para informar y abordar problemas, sugerir cambios y revisar y fusionar contribuciones de código.

[16] La comunidad global de desarrolladores se refiere a la amplia red de personas que colaboran, comparten conocimientos y contribuyen al desarrollo de software en todo el mundo.

del proyecto, se ha utilizado *SonarCloud*[17] como herramienta de control de calidad. Esta herramienta se integra con GitHub, lo que permite realizar un 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.[18]

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.

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.[19]

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 brindan 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

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

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

[19] CSV (*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.

que puede ser almacenada en archivos y posteriormente restaurada sin perder la integridad de los datos.[20]

Sin embargo, la masividad de los datos ha planteado desafíos en cuanto a su almacenamiento. El volumen de los datos generados ha requerido el empleo de técnicas de compresión y división en *lotes* para asegurar su conservación eficiente. Mediante la compresión[21], 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.

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.[22]

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.[23]

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

[20] La 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.

[21] La 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.

[22] Un 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.

[23] Microsoft Teams es una plataforma de colaboración en línea que permite la comunicación y el trabajo en equipo a través de chat, videoconferencias, compartición de archivos y otras funcionalidades.

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*.[24]

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[25], se exploraron y seleccionaron las mejores opciones para obtener 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.

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

[25]API (Application Programming Interface) es un conjunto de reglas y protocolos que permite la comunicación y la interacción entre diferentes software o componentes de software. Permite el intercambio de datos y la ejecución de funciones entre sistemas diferentes.

5. Aspectos relevantes del desarrollo del proyecto

5.1. Análisis de los datos disponibles

Para llevar a cabo el análisis de las redes de dependencias de paquetes de software, contamos con valiosos datos proporcionados por *libraries.io*[8] en formato CSV. Estos datos nos permiten extraer la red de dependencias de los principales repositorios de paquetes de software[26]. Sin embargo, es importante destacar que estos datos no están actualizados a la fecha actual debido a la alta volatilidad 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.

Para aprovechar al máximo este conjunto de datos, es importante considerar la temporalidad de la información y su contexto histórico al realizar análisis y conclusiones. Esto nos permitirá comprender mejor la dinámica de las redes de dependencias y sus cambios a lo largo del tiempo.

[26] Algunos repositorios incluidos en los conjuntos de datos son: npm, Maven, Go, PyPI, NuGet, Packagist, Rubygems, Cargo, CocoaPods, Bower, Pub, CPAN, CRAN, Clojars, conda, Hackage, Hex, Meteor, Homebrew, Puppet, Carthage, SwiftPM, Julia, Elm, Dub, Racket, Nimble, Haxelib, PureScript, Alcatraz, Inqlude.

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 el ecosistema del software evoluciona rápidamente y se introducen constantemente nuevas versiones y cambios en los paquetes, es crucial contar con información que refleje de manera precisa el panorama actual de las dependencias entre los paquetes.

5.2. 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[27].
- **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[28].
- **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[29].

[27] El registro y obtención de la clave de API son necesarios para acceder a las funcionalidades de la API de libraries.io.

[28] La versión gratuita de libraries.io tiene ciertas limitaciones y las características avanzadas pueden requerir una suscripción de pago.

[29] El límite de velocidad impuesto por libraries.io está destinado a mantener un equilibrio en el uso de recursos y prevenir cargas excesivas 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[30].

Es importante tener en cuenta estos inconvenientes al utilizar la API de libraries.io para asegurar un uso adecuado y obtener los datos necesarios para nuestro estudio de manera eficiente y precisa.

5.3. Exploración de los repositorios

La exploración de los repositorios de paquetes desempeña un papel fundamental en nuestro estudio. A continuación, se presentan los puntos basicos 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 extracción[31].

- **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

[30] El uso de la paginación en las solicitudes a libraries.io permite controlar la cantidad de resultados obtenidos por página, pero puede implicar una posible pérdida de información en caso de resultados truncados.

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

de mantenimiento son aspectos esenciales para comprender la evolución de los paquetes a lo largo del tiempo[32].

5.4. Análisis empírico de las redes de dependencias de repositorios de paquetes

Análisis de la red de dependencias 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.[33]

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 (en concreto un 474 %).[34]

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*. [35]

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

[33] Es importante mencionar que existen otros repositorios interesantes como *Conda* o *Poetry*.

[34] Este crecimiento refleja el creciente interés y contribuciones de la comunidad de desarrolladores a *PyPI*.

[35] Es importante destacar que la comunidad de *Python* es conocida por su activa participación en la creación y mantenimiento de paquetes en *PyPI*, lo que contribuye al enriquecimiento continuo del ecosistema.

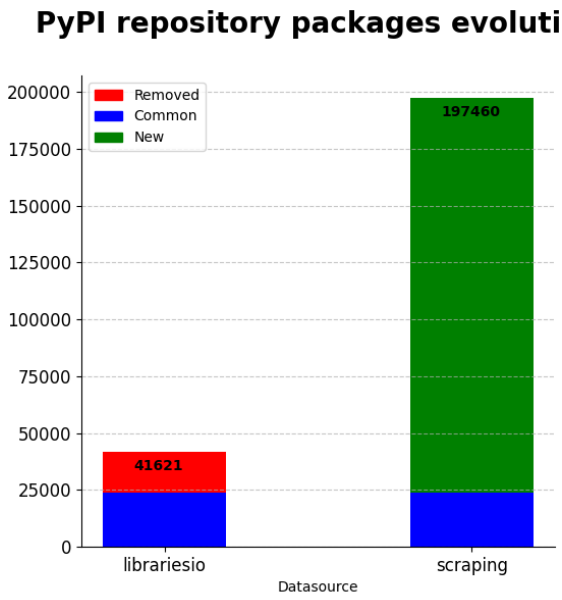

Figura 5.1: Comparacion de la cantidad de paquetes en PyPI entre los datos de libraries.io y los recolectados en este trabajo.

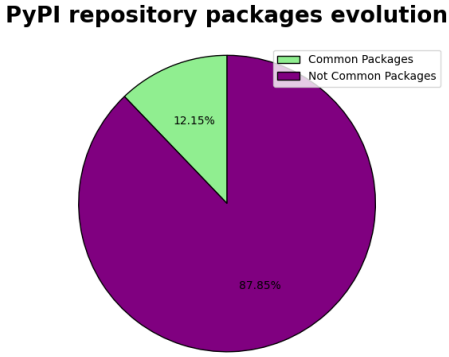

Figura 5.2: Paquetes comunes y no comunes actualmente.

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 datos [36]. 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.

[36]Este porcentaje se basa en datos obtenidos experimentalmente.

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

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 nos proporciona 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 actualizaciones[37].

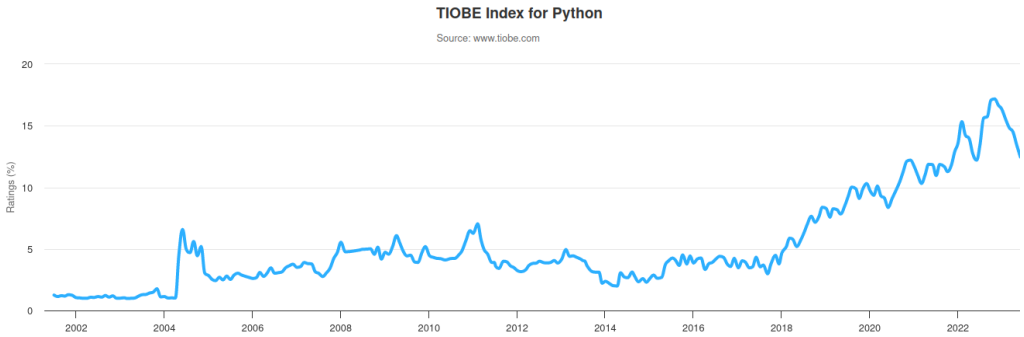

Figura 5.3: Popularidad de PyPI a lo largo del tiempo.

Esta tendencia en la popularidad de Python se ve reflejada en las estadísticas realizadas por la empresa TIOBE[38].

Grado de la red

Al examinar la distribución de grado en ambos conjuntos de datos, se observa que sigue una distribución de ley de potencias[39].

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

[38] https://www.tiobe.com/tiobe-index/python/

[39] La distribución de ley de potencias es una característica común en las redes de dependencias.

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* individuos [40].

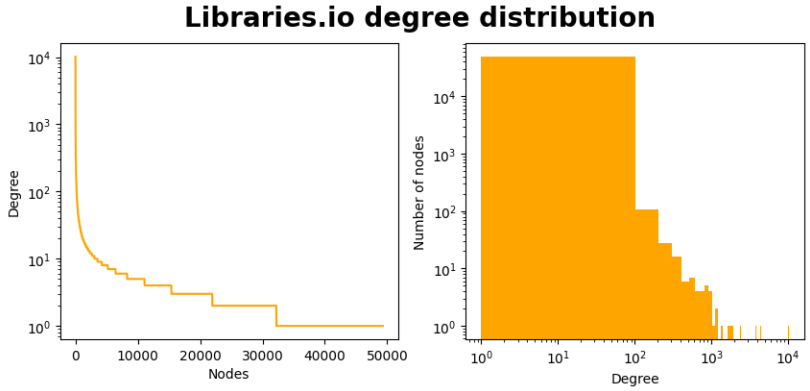

Figura 5.4: Distribucion de grado de PyPI para libraries.io.

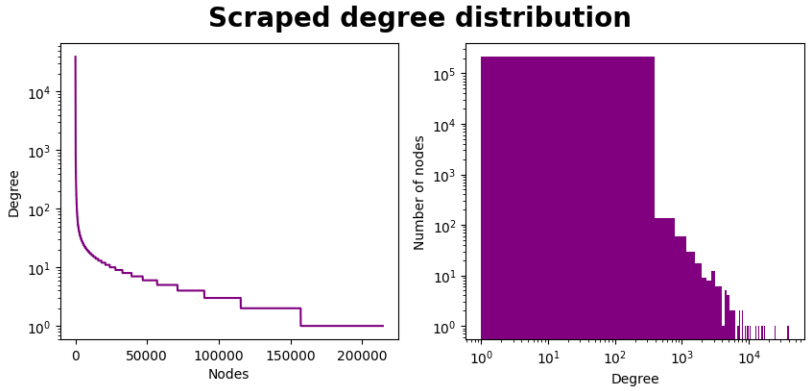

Figura 5.5: Distribucion de grado de PyPI para los datos recolectados.

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*,

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

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 paquetes[41].

Estas estadísticas revelan cambios importantes en la estructura de la red de dependencias de paquetes. El incremento en el grado máximo y el aumento 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 comunidad[42].

Grado de salida (out degree)

El *out degree* es una métrica que nos 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. La presencia destacada de estas bibliotecas en el ranking resalta su importancia dentro del ecosistema de paquetes de Python y sugiere que son ampliamente adoptadas por los desarrolladores.

En este análisis de ranking, se observa que ciertos paquetes se han mantenido con respecto al ranking anterior, lo cual indica su constante relevancia en la comunidad de desarrolladores de Python. Estos paquetes

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

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

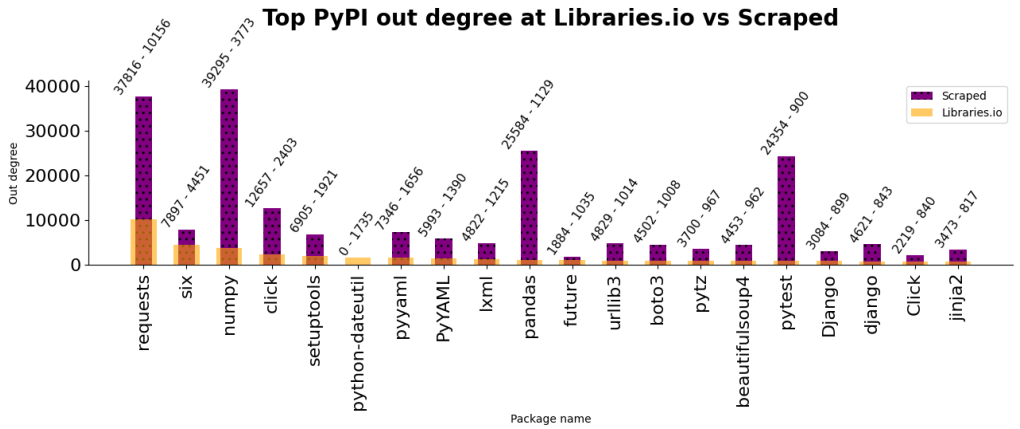

Figura 5.6: Top de paquetes con mayor grado de salida en PyPI para libraries.io

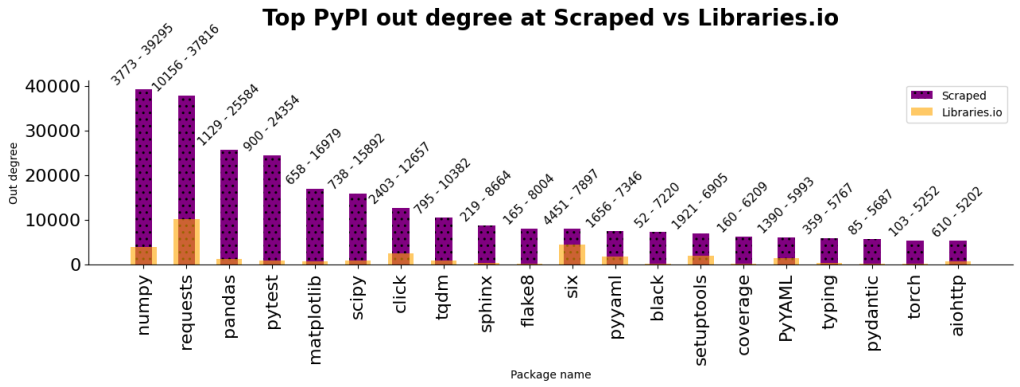

Figura 5.7: Top de paquetes con mayor grado de salida en PyPI para scraped.

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 Python[43].

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*.

[43]La 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

Su ascenso en el ranking puede ser atribuido a su creciente popularidad y utilidad en el desarrollo de proyectos de Python, ya que son bibliotecas ampliamente conocidas y utilizadas por la comunidad de desarrolladores[44].

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

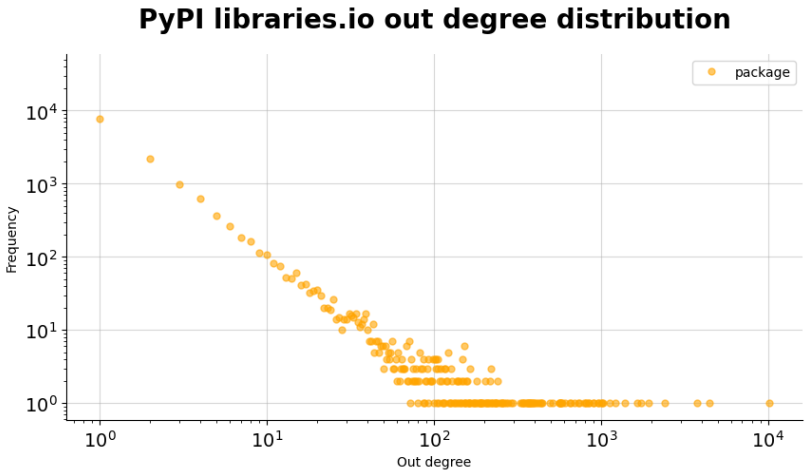

Figura 5.8: Distribución de *Out degree* para libraries.io

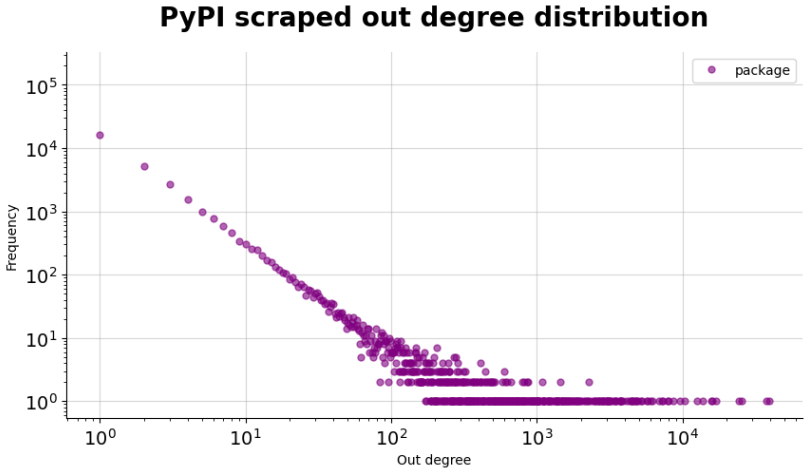

Figura 5.9: Distribución de *Out degree* para scraped

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

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

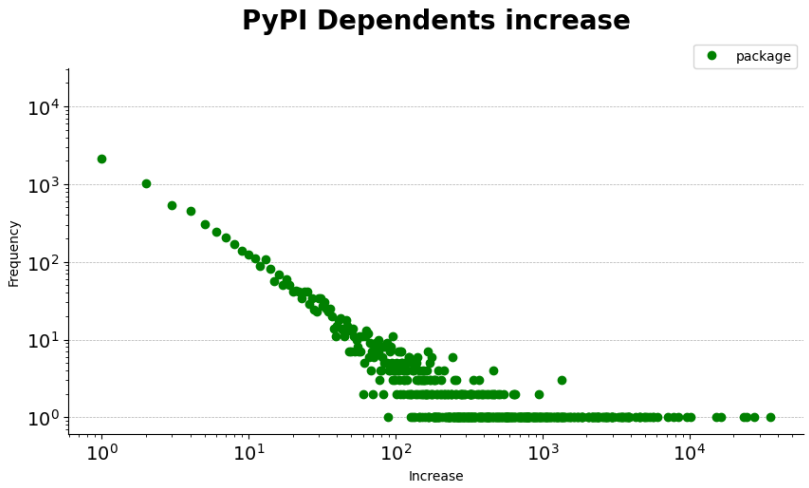

Figura 5.10: Incremento del out degree en PyPI

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:

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

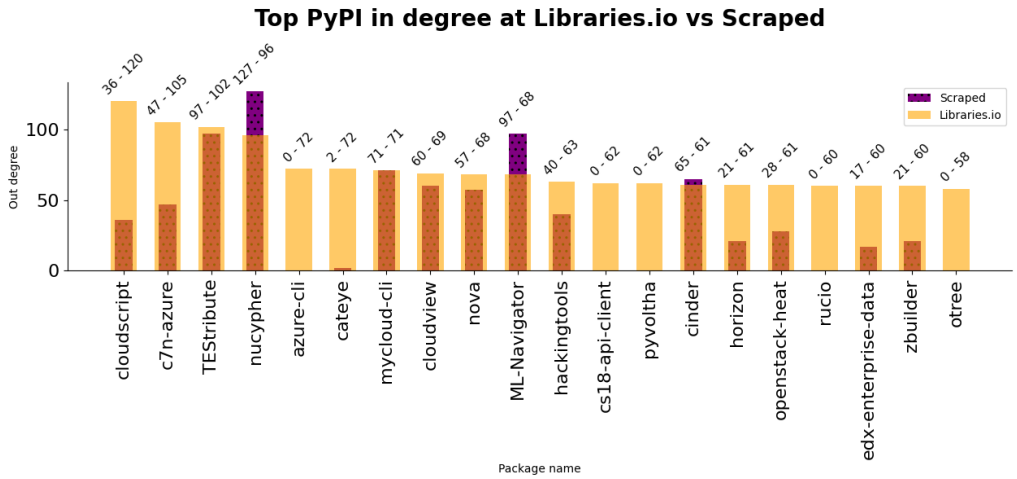

Figura 5.11: 20 paquetes con mayor In degree en libraries.io

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.

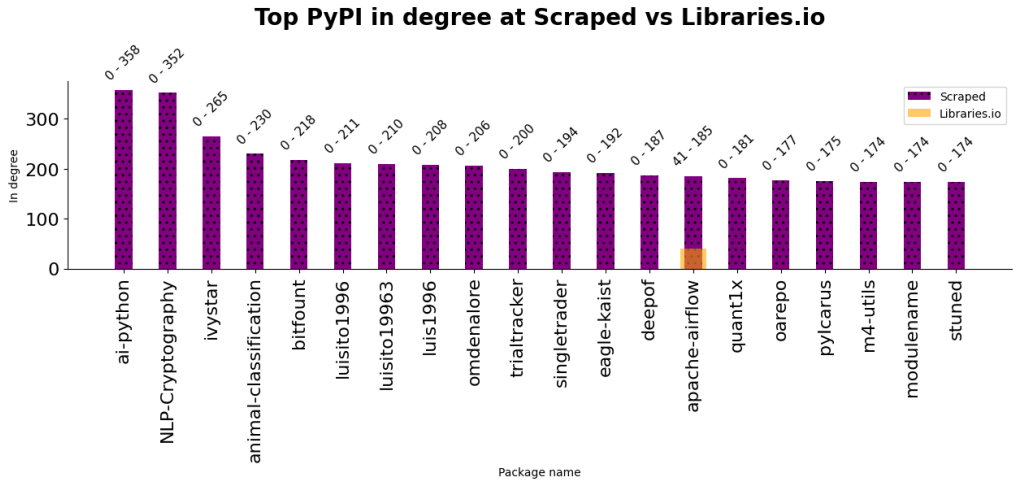

Figura 5.12: 20 paquetes con mayor In degree en scraped

Si realizamos este mismo análisis con el top 20 de los paquetes con más *in degree* en la actualidad, podemos llegar a conclusiones similares[45]. Se observa una tendencia decreciente en el número de dependencias de los 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[46].

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* [47], 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 estabilidad[48].

En el análisis de la distribución de In degree, 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*.

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

[45] El *in degree* se refiere al número de dependencias que tiene un paquete en la red de dependencias

[46] La disminución de las dependencias en los paquetes más influyentes puede ser resultado de optimizaciones, modularidad mejorada o cambios en las preferencias de los desarrolladores

[47] https://pypi.org/project/apache-airflow/

[48] El 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

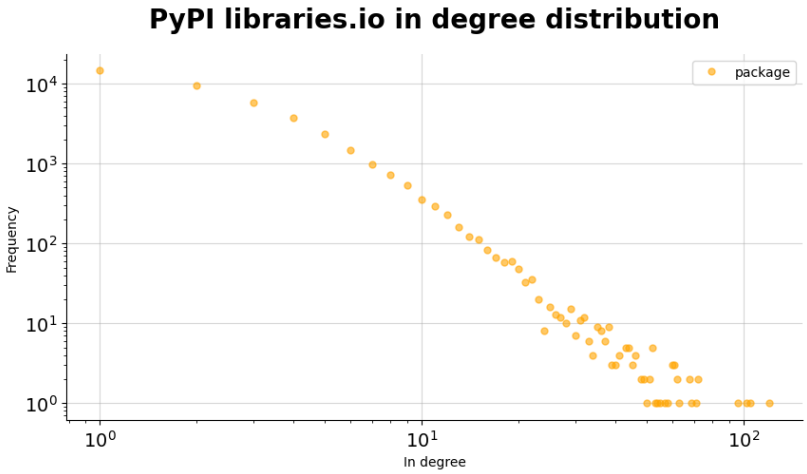

Figura 5.13: Distribucion del in degree en libraries.io

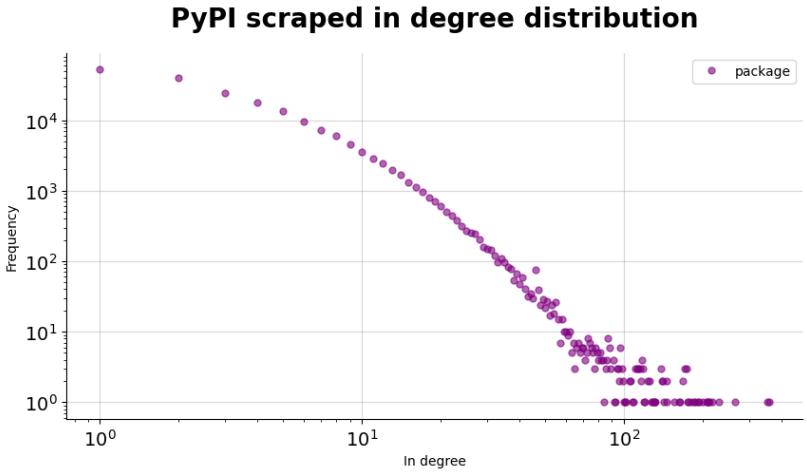

Figura 5.14: Distribucion del in degree en scraped

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, 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*[49] nos 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 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.

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 ella[50].

Si visualizamos el número de dependencias transitivas de estos paquetes, podemos obtener conclusiones más precisas. Existe un grupo de paquetes con un menor número de dependencias transitivas, lo que los hace menos vulnerables en comparación con el resto. Además, tener un alto *PageRank* en estos paquetes implica que son más confiables en términos de dependencias[51].

[49] PageRank es un algoritmo utilizado para establecer un ranking de importancia en la web.

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

[51] Un 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.

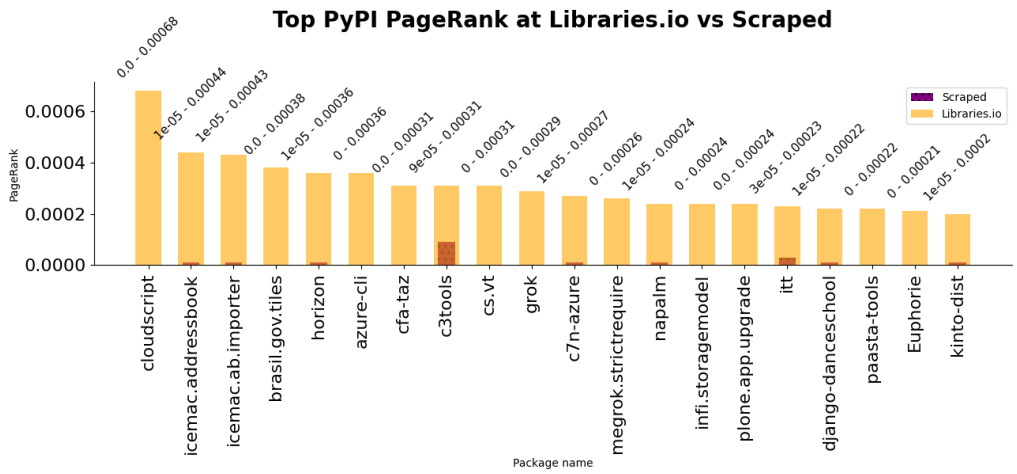

Figura 5.15: Top 20 pagerank en libraries.io

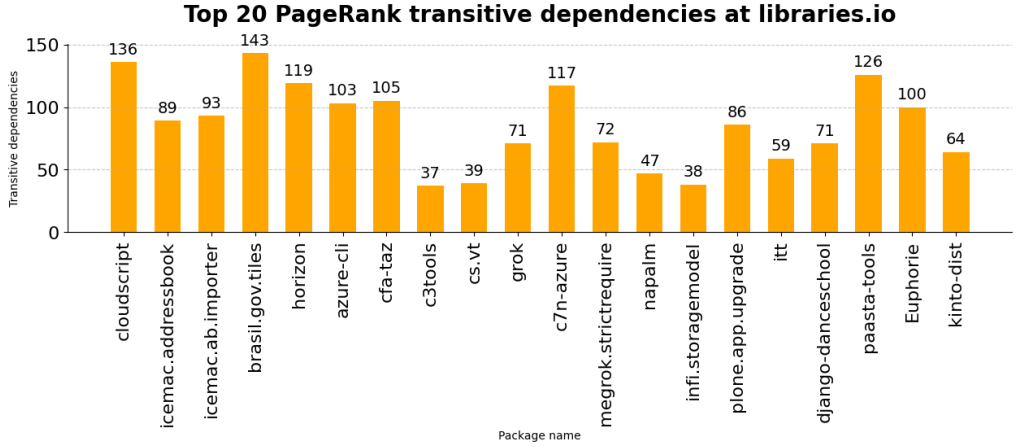

Figura 5.16: Dependencias transitivas del top 20 pagerank en libraries.io

También se observa que en este grupo de paquetes, las dependencias transitivas son en promedio más bajas[52].

Al analizar los resultados, se puede ver que los paquetes presentes en este grupo han experimentado una evolución considerable, con una reduc-

[52]El 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.

ción significativa en su *PageRank*[53]. 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.

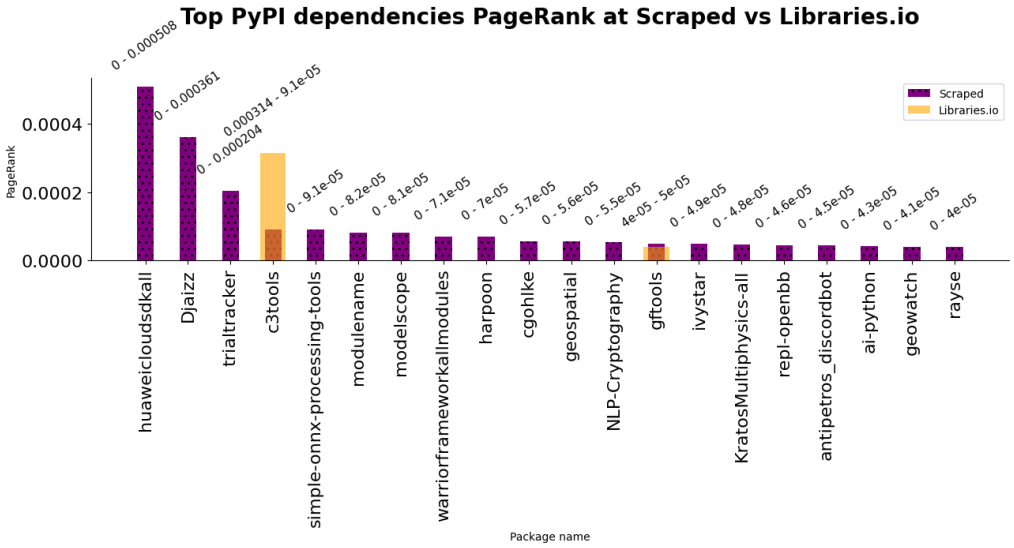

Figura 5.17: Top 20 pagerank en scraped

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 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

[53]El *PageRank* de estos paquetes ha disminuido en comparación con mediciones anteriores.

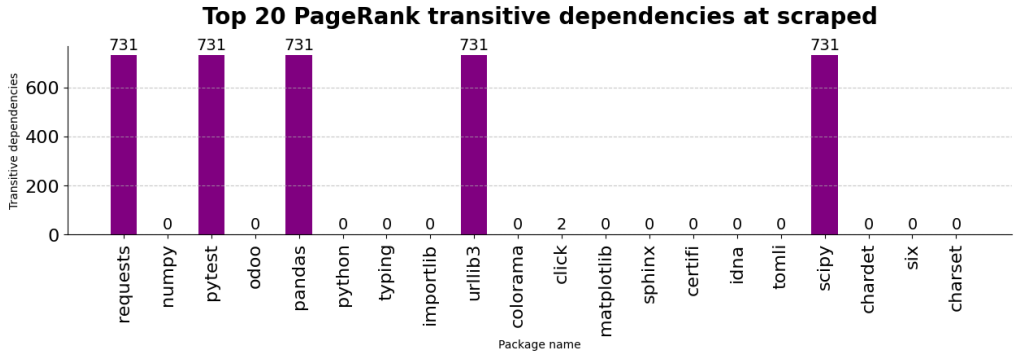

Figura 5.18: Dependencias transitivas del top 20 pagerank en sacraped

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 dependencias[54].

Si invertimos el grafo de nuestra red de dependencias, 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[55].

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

[54]Los 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.

[55]Un alto *PageRank* indica la importancia y la confianza que otros paquetes depositan en el paquete en cuestión, lo que lo convierte en una fuente valiosa de recursos y funcionalidades para otros paquetes en la red.

otros paquetes dependan directa o indirectamente de él para llevar a cabo sus propias funciones o tareas[56].

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

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 5.1: Comparación entre paquetes obtenidos de libraries.io y scraped para la metrica 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.

A la vista de la distribución del impacto en la red de libraries.io, 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[57].

[56] Los paquetes con un alto *PageRank* desempeñan un papel fundamental en la red, ya que otros paquetes dependen de ellos para realizar sus propias tareas y funciones, lo que los convierte en componentes esenciales de la red de dependencias.

[57] La distribución del impacto en la red de libraries.io muestra que hay muchos paquetes con un bajo impacto, lo cual sugiere que no son ampliamente utilizados como dependencias en otros paquetes.

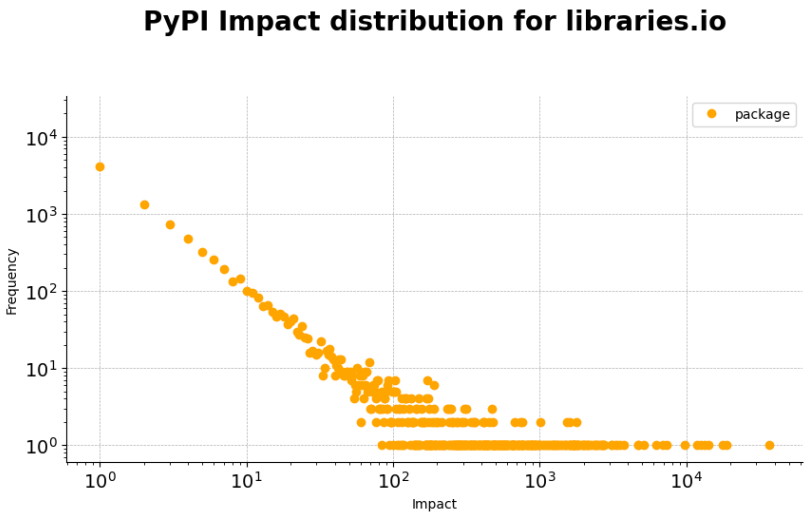

Figura 5.19: Distribución del impacto en la red de libraries.io

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.

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.

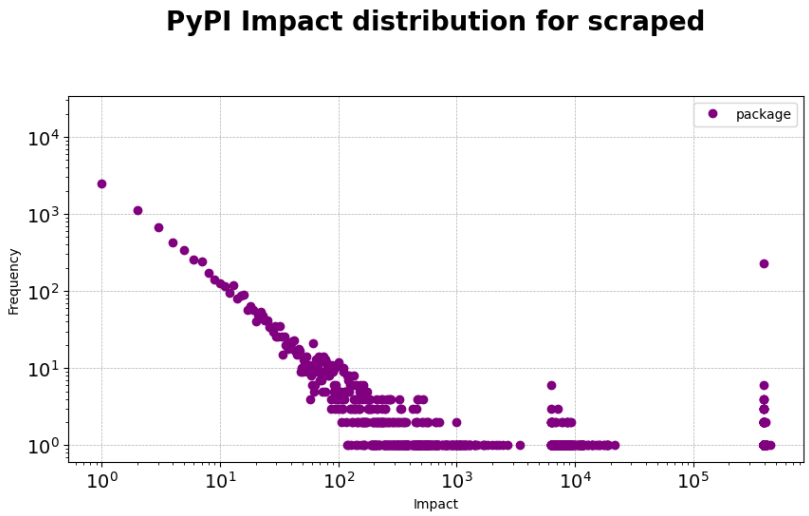

Figura 5.20: Distribución del impacto en la red scraped

Asimismo, se ha identificado otro grupo de tamaño medio-bajo pero que resulta notable debido a su impacto extremadamente elevado.

Tabla 5.2: Comparación del incremento del impacto para de libraries.io, scraped

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 desarrolladores 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.

Tabla 5.3: Disminución del impacto en libraries.io y scraped

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

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.

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[58].

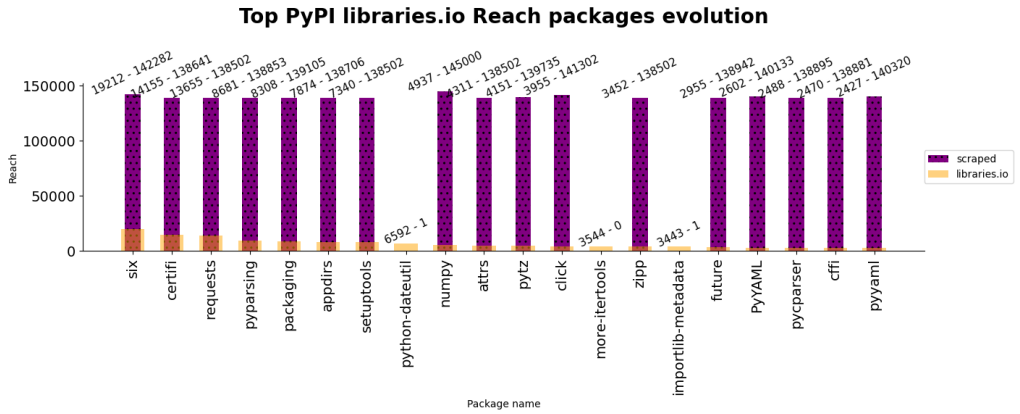

Figura 5.21: Top Reach en libraries.io

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

[58]El 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.

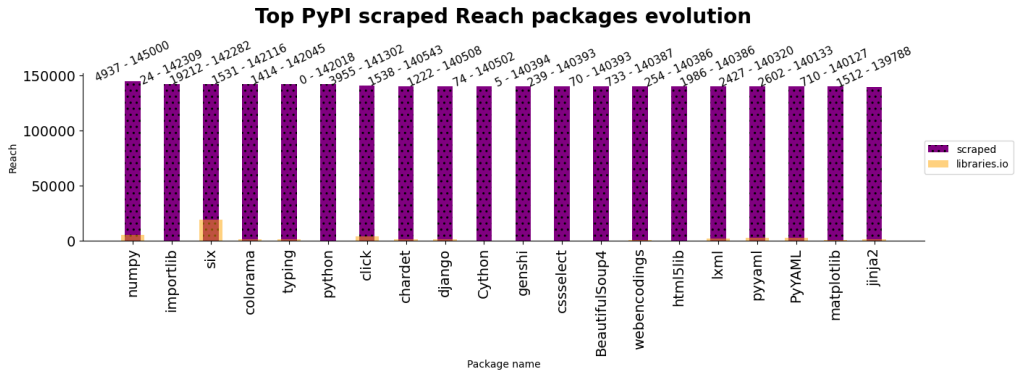

Figura 5.22: Top Reach en scraped

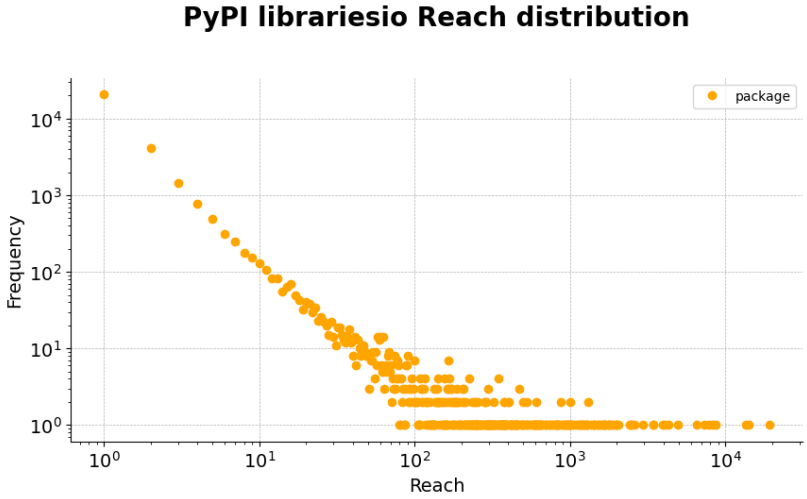

Figura 5.23: Distribución del Reach en libraries.io

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.

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.

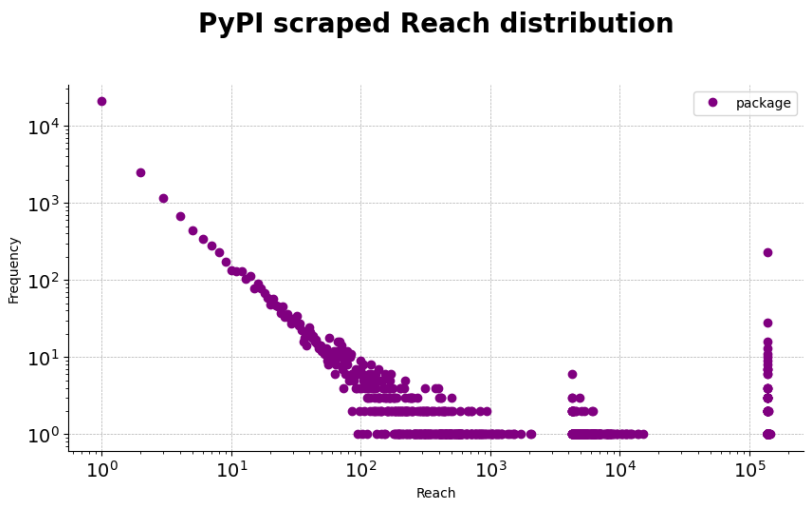

Figura 5.24: Distribución del Reach en scraped

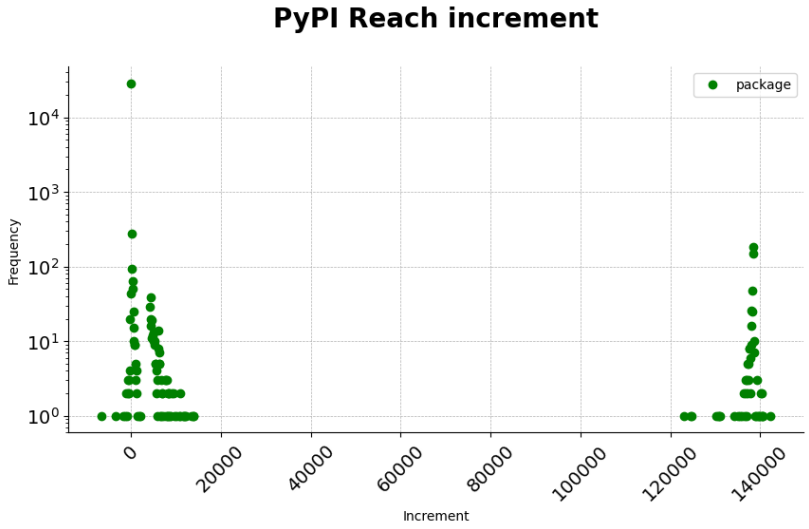

Figura 5.25: Incremento del Reach

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 ±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 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 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.

Size	Head	Head pagerank	Avg degree	Density	Diameter	Clustering coeff	Transitive deps
283	pytest	0.135569	8.890	0.015	14	0.196	206873
9	transformers	0.169269	4.947	0.137	8	0.358	23807
8	crds	0.221896	3.000	0.214	6	0.222	6288
8	lamindb	0.337734	5.250	0.375	3	0.383	6784
8	pycldf	0.329553	4.750	0.339	5	0.498	6752
6	grimoirelab	0.412447	5.000	0.500	2	0.776	4536
6	fdutil	0.315617	4.666	0.466	3	0.470	4452
6	omnitools	0.436878	3.666	0.366	3	0.448	4440
5	sunpy	0.351692	2.800	0.350	4	0.000	3770
5	pytablewriter	0.329746	6.000	0.750	2	0.777	3855
5	hist	0.315050	3.200	0.400	4	0.386	3765
5	pyontutils	0.318299	3.600	0.450	3	0.421	3755
4	featuretools	0.429209	3.000	0.500	3	0.406	5276
4	arnica	0.328053	2.500	0.416	3	0.000	3016
4	polyaxon	0.409227	3.000	0.500	3	0.406	3220
4	fastcore	0.371516	3.000	0.500	3	0.700	2988
4	shiny	0.429209	3.000	0.500	3	0.406	3048
4	esmerald	0.429209	3.000	0.500	3	0.406	3064
4	robotpy	0.390652	4.500	0.750	2	0.800	2948
3	dacy	0.486486	2.666	0.666	2	0.000	3780

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 funcion del tamaño del componente.

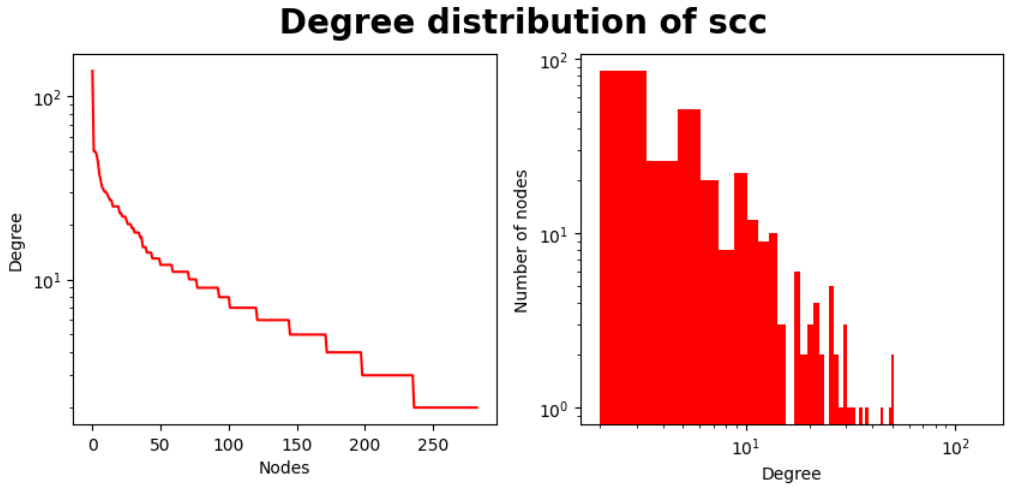

Figura 5.26: Distribucion de grado del mayor componente fuertemente conexo

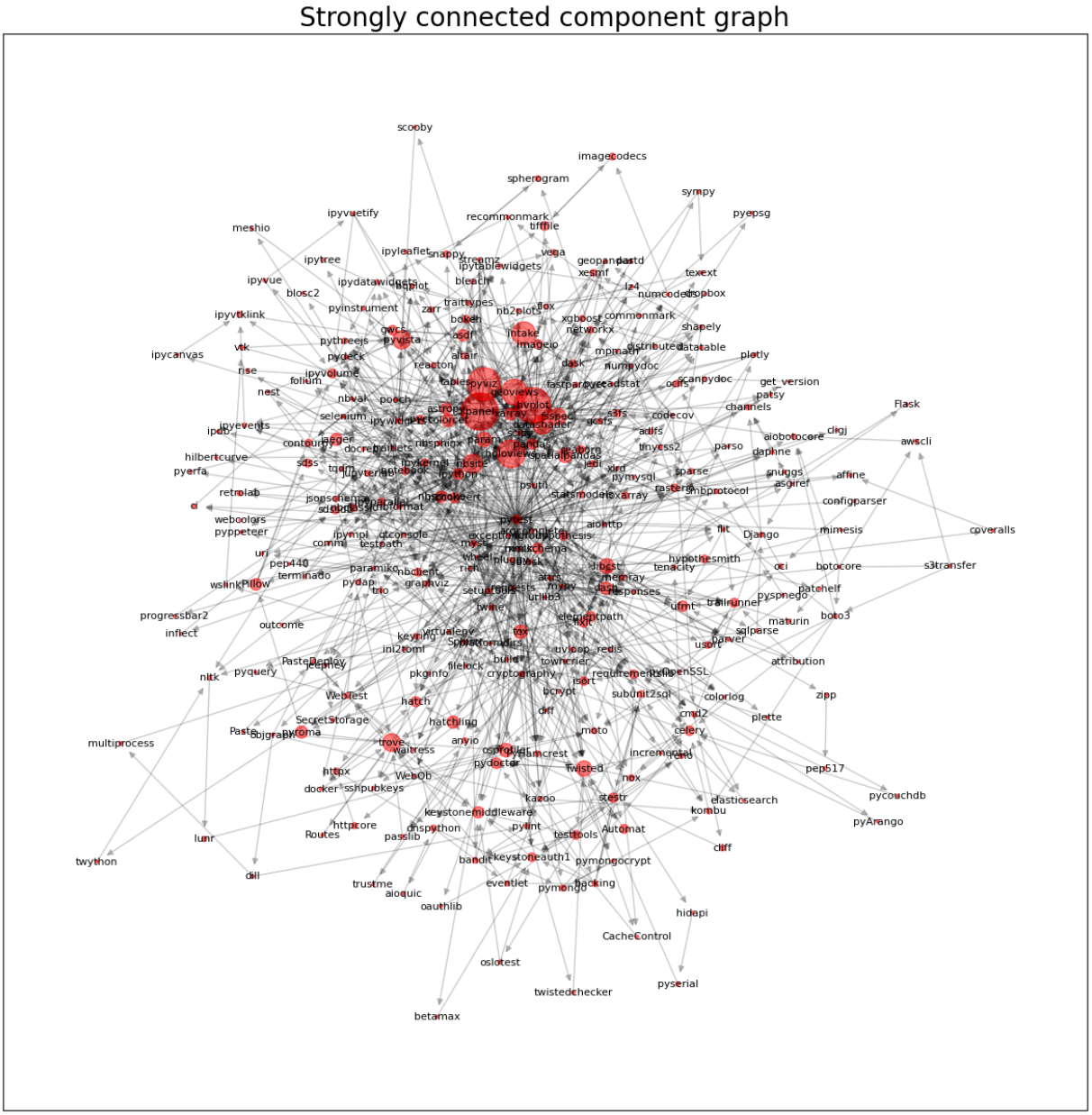

Figura 5.27: Mayor componente fuertemente conexo

6. Trabajos relacionados

Este apartado sería parecido a un estado del arte de una tesis o tesina. En un trabajo final grado no parece obligada su presencia, aunque se puede dejar a juicio del tutor el incluir un pequeño resumen comentado de los trabajos y proyectos ya realizados en el campo del proyecto en curso.

7. Conclusiones y Líneas de trabajo futuras

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

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] 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.

[3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *Nature*, 406:378–382, 7 2000.

[4] 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.

[5] Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. When github meets cran: An analysis of inter-repository package dependency problems. 03 2016.

[7] 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.

[8] Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, January 2020.

[9] 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.

[10] Tom Mens, Alexandre Decan, and Maelick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. 02 2017.

[11] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[12] Marton Posfai and Albert-Laszlo Barabasi. *Network Science*. Citeseer, 2016.

[13] J. Sametinger. *Software Engineering with Reusable Components*. Springer Berlin Heidelberg, 1997.

[14] Daniel Seto. dsr0018/olivia: OLIVIA - Open-source Library Indexes Vulnerability Identification and Analysis, nov 2022.

[15] 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.