UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE BACKEND PARA FRAMEWORK DASHAI

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

RODRIGO IGNACIO URREA LOYOLA

PROFESOR GUÍA:
FELIPE BRAVO MÁRQUEZ

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULE
ANDRÉS ABELIUK KIMELMAN

Este trabajo ha sido parcialmente financiado por ANID FONDECYT 11200290, Centro Nacional de Inteligencia Artificial CENIA FB210017 e Instituto Milenio Fundamento de los datos IMFD.

SANTIAGO DE CHILE
2023

Resumen

Hoy en día estamos evidenciando grandes avances en el campo del *Machine Learning*, tanto a nivel científico como empresarial. Cada día existen nuevos servicios que utilizan *Machine Learning* para generar mejores soluciones a problemas de transporte, salud, entretención, etc..

Este contexto ha permitido el desarrollo de una infinidad de *softwares* y *frameworks* que permiten desarrollar modelos de *Machine Learning* y dejarlos disponibles para su uso masivo. Sin embargo, existe una oferta escasa de herramientas que permitan hacer esto con modelos del estado del arte y de forma gratuita y amigable al usuario con poco conocimiento en programación.

Es gracias a esto que nace *DashAI* , un *framework* de código abierto, local y gratuito que permite entrenar modelos de *Machine Learning* de forma sencilla gracias a su interfaz gráfica y que soporta modelos del estado del arte debido a su arquitectura extensible.

No obstante, *DashAI* , a tiempo de realización de esta memoria, es un *software* que se encuentra aún en desarrollo y por tanto no está exento de problemas. Es así como nace la motivación de realizar este trabajo de título para resolver algunos de los problemas presentes en *DashAI* . Los problemas a abordar en este trabajo son el fuerte acoplamiento que este presenta entre su *frontend* y *backend*, la necesidad de definir puntos de guardado para los usuarios que utilicen la aplicación y la necesidad de evitar mantener al usuario en espera mientras se realiza el entrenamiento de modelos de *Machine Learning*.

Es frente a esto que el memorista propone resolver estos problemas mediante la elaboración de una *API RESTful* extensible capaz de separar el *frontend* del *backend* de *DashAI* , diseñar e implementar un modelo de datos que almacene la información necesaria para que un usuario pueda tener puntos de guardado que le permitan retomar procesos iniciados previamente y diseñar e implementar una cola de trabajos que permita ejecutar los entrenamientos solicitados por el usuario de forma asíncrona, permitiendo que la aplicación siga funcionando mientras se entrenan los modelos de *Machine Learning*.

En este documento se muestran todos los diseños elaborados por el memorista junto con las implementaciones desarrolladas para dejar dichos elementos disponibles en *DashAI* . Todo el desarrollo realizado por el memorista fue evaluado mediante la ejecución de *tests* unitarios, revisiones de ingenieros titulados y pruebas de integración con el resto del *framework*.

If you gaze long enough into an abyss,
the abyss will gaze back into you.
–Friedrich Nietzsche

Agradecimientos

Agradezco a mi madre, padre y padrastro por siempre estar ahí para mi, tanto en las buenas como en las malas y siempre motivarme a tomar nuevos desafios.

Agradezco a mi pareja Elizabeth por estar a mi lado y acompañarme tanto en mis días claros como oscuros. Sin su apoyo no me encontraría terminando esta etapa de mi vida.

Agradezco a mi profesor guía Felipe por confiar en mí para realizar este trabajo y motivarme a no desistir de convertirlo en mi trabajo de título.

Agradezco a mis amigos Alejandro, Catalán, Alonso, Nicolay y Cristóbal por alegrar mis días en la universidad.

Agradezco a mis compañeros de *DashAI* Oziel, Cristián, Pablo, Maximiliano, Ignacio por tan buen trabajo desarrollando y mejorando este proyecto.

Finalmente agradezco a todas las personas con las que he compartido en la universidad que han aportado en mi formación como profesional y como persona.

Tabla de Contenido

Índice de Ilustraciones

Code Listings

Capítulo 1

Introducción

1.1. Contexto

Hoy en día la sociedad está viviendo una revolución tecnológica, principalmente impulsada por el *Big Data* y el *Machine Learning*. Además se estima que este último genere un aumento de un 14 % del pip mundial al año 2030, según se índica en [15].

El *Machine Learning* es una subárea del área de la inteligencia artificial que busca generar algoritmos capaces de aprender a reconocer patrones y lograr con ello generar predicciones, este aprendizaje se logra mediante el uso de datos de ejemplo y de métricas que le indican al algoritmo que tan acertada es la predicción generada.

Para clasificar de mejor manera estos algoritmos, también llamados modelos, es que se usa el termino tarea o task en inglés. Una tarea es un problema que se busca resolver usando *Machine Learning* y se caracteriza por el tipo de dato de entrada (vector númerico, texto, imágenes, etc.) y por el tipo de dato que se espera predecir gracias al modelo (nuevamente esto puede ser un número, texto, etc.).

Tanto el *Big Data* como el *Machine Learning* afectan directamente el día a día de las personas, ya sea al momento de buscar una nueva serie o película que ver en una plataforma de *streaming*, al subir a trenes subterráneos autónomos o al recibir instantáneamente el diagnóstico de un examen médico.

Para lograr generar dichos sistemas se necesitó de mucho desarrollo teórico, tanto en las arquitecturas a utilizar, la forma de mejorar las predicciones del modelo y la forma de evaluar correctamente las predicciones generadas, sin embargo, también se necesitó implementar, entrenar y evaluar una infinidad de modelos, tanto para experimentar y generar con ello nueva teoría, como para generar un nuevo producto.

Es debido a esto último que se ha propiciado el auge de *frameworks* y herramientas destinados a facilitar y potenciar el desarrollo de modelos de *Machine Learning*. Estas se mencionan en profundidad en la Sección 2.1, pero de todos se pueden destacar 3 grandes grupos:

- Herramientas divulgativas: Orientadas a usuarios con poca experiencia en programación, les permiten a estos crear y utilizar modelos clásicos de *Machine Learning*.
- Herramientas científicas: Orientadas a investigadores con alta experiencia en programación, les permiten crear y utilizar modelos del estado del arte en *Machine Learning*.
- Herramientas *No-Code*: Orientadas a desarrolladores (con experiencia variable en programación) del área del *Machine Learning*, les permiten desplegar modelos desarrollados para ser usados en productos. Suelen ser alojadas en la nube, es decir, que el procesamiento no se realiza en el computador del usuario.

Ahora bien, cada uno de estos tipos de herramientas presentan desventajas o limitaciones en relación a las otras, estas se mencionan a continuación.

- Las herramientas divulgativas no presentan modelos del estado del arte y son difíciles de adaptar a las tareas actuales, esto se debe principalmente a que no están escritas en *Python* o que presentan una estructura poco extensible.
- Las herramientas científicas suelen presentar grandes barreras de entrada, lo que limita el número de usuarios que puede acceder a ellas.
- Las herramientas *No-Code* son pagadas además de no dar explicaciones al usuario sobre los resultados que se obtienen, lo que no resulta útil al momento de hacer investigación en el área.

Es en este escenario que nace la idea de crear *DashAI* , un *framework open source*, local y gratuito que permita generar modelos de *Machine Learning* de manera sencilla sin necesidad de tener grandes conocimientos en programación, pero que permita ser extendido por desarrolladores que si cuenten con dichos conocimientos para poder incorporar tareas y modelos del estado del arte.

1.2. Problema

Como se expone en la Sección 2.3, existen precursores de *DashAI* , es decir, implementaciones previas de *DashAI* desarrolladas bajo otros contextos pero con un objetivo similar, brindar una plataforma para entrenar y evaluar modelos de *Machine Learning*.

Estos ancestros de *DashAI* contaban con una serie de falencias en su diseño e implementación que son las que motivan el desarrollo de esta memoria. A continuación, se presentan las falencias mencionadas.

Como se menciona en la Sección 2.3, *TextPerimenter*, uno de los precursores de *DashAI* , es un *software* que paso por varios cambios de tecnología, en particular la librería utilizada para desarrollar la componente gráfica de la aplicación. Estos cambios de tecnología fueron costosos debido al fuerte acoplamiento que existe entre el *frontend* y el *backend* de dicha herramienta. Lo anterior llevo a una ralentización en el desarrollo de la aplicación y es uno de los problemas que se busca atacar con el desarrollo de este trabajo de título.

El termino *frontend* y *backend* se suelen usar en el ámbito del desarrollo web, el primero hace referencia a la componente gráfica de la aplicación, que es con la que interactúa el usuario, mientras que el segundo se refiere a la componente lógica de la aplicación, donde están codificados todos los procesos que se llevan a cabo.

Un segundo problema que se presenta en los precursores de *DashAI* es que estos no cuentan con una delimitación claras de las etapas que tiene el flujo de trabajo de *Machine Learning*, descritas en la Sección 2.2.

Esto lleva a que no sea posible la definición de puntos de guardado (o *checkpoints* en ingles) lo que conduce a que el usuario deba realizar todo el flujo de forma seguida. Este es el segundo problema que se busca atacar con la realización de esta memoria.

Un último problema que se presenta en las herramientas antes descritas es que estas no toman en cuenta el tiempo que demora el entrenamiento y evaluación de modelos de *Machine Learning* y por lo cual se deja al usuario en espera por el término del proceso, lo que atenta contra la usabilidad de la aplicación.

Como se menciona en [1], el entrenamiento de un *transformer* [19] puede tomar desde 1 a 130 minutos, es decir, en caso de querer entrenar dos modelos de este estilo se tendría que dejar al usuario en espera por alrededor de 4 horas. Este es el último problema que se busca abordar con el desarrollo de este trabajo.

1.3. Solución

Los problemas mencionados en la sección anterior son los que motivan el desarrollo de esta memoria, por lo que el propósito de este trabajo es desarrollar un *backend* capaz de dar solución a dichos problemas sin perder con ello la propiedad principal de *DashAI* , su extensibilidad.

Para ello es que propone el diseño y desarrollo de una *API*, es decir una interfaz de comunicación, que permita separar de manera definitiva el *frontend* del *backend*.

Junto con lo anterior se propone también el diseño de un modelo de datos compatible con el flujo de trabajo de Modelos de Machine Learning, mencionado en la Sección 2.2, que permita definir *checkpoints* en el flujo.

Y finalmente se propone la elaboración de un sistema de trabajos que pueda ejecutarse en un proceso distinto a la *API*, para que así esta pueda seguir funcionando, aunque se este entrenando un modelo.

Vale la pena destacar que *DashAI* es un proyecto que al momento de escribir esta memoria se encuentra en desarrollo, dicho desarrollo es llevado a cabo por un grupo de ingenieros civiles en computación, tesistas de magister en ciencias de datos y memoristas de ingeniería civil en computación, entre los que se incluye al memorista que escribe este trabajo, esto se explica más adelante en la Sección 1.5.

1.4. Objetivos

1.4.1. Objetivo General

El fin principal de esta memoria es Diseñar e Implementar el *backend* del *framework* *DashAI* , permitiendo que este sea extensible a las tareas que el usuario necesite abarcar.

1.4.2. Objetivos Especificos

Para lograr cumplir con el objetivo mencionado anteriormente, se proponen los siguientes objetivos específicos:

1. Diseñar e implementar una *API*, es decir, una interfaz de comunicación, que permita separar la lógica de negocios de la librería con la componente visual de la misma.
2. Diseñar un modelo de datos que permita separar en distintos procesos el uso de modelos.
3. Diseñar e implementar un sistema de trabajos que permita ejecutar el entrenamiento de modelos de *Machine Learning* de forma asíncrona, es decir, que la aplicación siga funcionando mientras se estén entrenando los modelos.

1.5. Metodología

Como se menciona en la Sección 1.3, *DashAI* es un proyecto colaborativo que se encuentra siendo desarrollado por memoristas, tesistas e ingenieros civiles en computación, donde cada integrante del equipo tiene a su cargo módulos distintos del *software*.

Para coordinar el desarrollo del *framework* se sigue la metodología *Shape Up* [18], la cual es similar a la metodología *Scrum*, pues se definen ciclos de trabajo acotados (entre 2 semanas a 1 mes) en donde cada integrante del equipo tiene tareas asignadas y cuyo progreso es revisado diariamente mediante la realización de reuniones de seguimiento o *dailys*. Sin embargo *Shape Up* se diferencia de *Scrum* en que esta no busca definir totalmente las tareas a desarrollar, si no que se le da libertad al desarrollador para atacar el problema desde su propia perspectiva y que este defina cómo desarrollar lo buscado. Esto último la hace ideal para trabajos exploratorios donde no se tiene una noción clara del producto final que se busca construir.

Además de lo anterior, todo desarrollo hecho por el equipo de *DashAI* es documentado a través de *Pull Request* (*PR*), las cuales son evaluadas en primera instancia por el sistema de integración continua del proyecto, el cual consiste en la ejecución de revisores de estilo de código, junto con la ejecución de los test presentes en el proyecto. Luego de ser aprobadas por el sistema de integración continua las *PR* son revisadas por alguno de los ingenieros presentes en el proyecto, los cuales pueden aprobar la *PR*, en cuyo caso de hace suben los cambios a la rama estable del proyecto, o pueden pedir corregir errores en la implementación.

1.6. Alcances y limitaciones

Debido al estado en el que se encuentra *DashAI* al momento de realizar esta memoria junto con el acotado tiempo que se tiene para realizar este trabajo de título es que se hace necesario la definición de alcances y limitaciones para esta memoria.

Con respecto a la extensibilidad de la solución desarrollada por el memorista frente a nuevas tareas, mencionada en la Sección 1.4, no se espera que este genere una arquitectura que permita soportar cualquier tarea de *Machine Learning*, si no que dada la existencia de una arquitectura que lo permita, los módulos desarrollados por el memorista no deben entorpecer ni limitar esa extensibilidad. Más adelante en la Subsección 3.1.2 se muestra la arquitectura que se tiene en *DashAI* para dotar de extensibilidad a la herramienta.

Dado que el trabajo que se propone abarcar por el memorista no contempla el desarrollo de una componente gráfica, no se considera la realización de pruebas con usuarios ni de encuestas. Sin embargo, si se espera el desarrollo de *tests* exhaustivos con el fin de comprobar que el trabajo realizado con el memorista cumpla con los objetivos que se proponen.

Debido a la motivación del software de ser fácil de utilizar y por lo tanto de instalar es que no se contempla un despliegue de la aplicación en algún servidor para su uso masivo, si no que se espera que sea usada de manera local por los usuarios, es debido a esto que tampoco se considera la evaluación de la *API* frente a múltiples solicitudes simultaneas ni problemas de conexión.

1.7. Estructura del documento

En el Capítulo 1 se exponen las condiciones que dan vida a esta memoria, junto con los problemas a atacar en el desarrollo de la misma y las soluciones propuestas a estos por el memorista. Se definen también los alcances y limitaciones del trabajo.

En el Capítulo 2 se muestran conceptos de vital importancia para la concepción de esta memoria, como lo son las herramientas que actualmente existen para atacar el problema propuesto, como se utilizan los modelos de *Machine Learning* actualmente y como ha sido el desarrollo de este software desde sus inicios, para indicar las razones de algunas decisiones que se han tomado.

En el Capítulo 3 se muestra el diseño del trabajo hecho por el memorista, primero exhibiendo una vista general del software con el fin de contextualizar al lector sobre DashAI, para luego dar énfasis en los módulos específicos desarrollados por el memorista, los cuales son la *API*, su base de datos y finalmente la integración de la librería con un sistema de trabajos asíncronos.

En el Capítulo 4 se expone en detalle todas las funcionalidades desarrolladas por el memorista junto con las decisiones tomadas y las razones detrás de ello.

En el Capítulo 5 se menciona como se llevó a cabo la implmentación del trabajo del memorista, volviendo a mencionar la metodología de trabajo ocupada en *DashAI* , mencionada en la Sección 1.5 y los problemas que surgieron al integrar estos cambios con el *framework*.

Finalmente, en el Capítulo 6 se expone una reflexión sobre el trabajo realizado por el memorista, haciendo una discusión sobre el cumplimiento de los objetivos, la utilidad de las tecnologías escogidas para desarrollar el trabajo y finalmente la mención de los pasos a seguir para seguir mejorando la librería.

Capítulo 2

Estado del Arte

Este capítulo busca introducir al lector nociones y conceptos de suma importancia para el entendimiento del trabajo realizado. En la Sección 2.1 se mencionan tecnologías actuales relacionadas con el uso de modelos de *Machine Learning*, junto con una comparativa entre ellas y *DashAI* . La Sección 2.2 busca mostrar al lector como se utilizan los modelos de *Machine Learning* típicamente, esto con el fin de justificar el diseño que se muestra en el Capítulo 3. Y finalmente en la Sección 2.3 se exponen los prototipos previos a *DashAI* , con el fin de mostrar algunos de los problemas que estos presentan.

2.1. Tecnologías similares

En esta sección se hace un repaso de las tecnologías existentes para trabajar con *Machine Learning*, separándolas según las clasificaciones mencionadas en la Sección 1.1, detallando ventajas y desventajas de estas. Al final de la sección se incluye un cuadro resumen que compara las herramientas entre ellas y con *DashAI* .

2.1.1. Herramientas Científicas

Estas son principalmente librerías que están orientadas a usuarios con gran experiencia en programación y permiten que estos puedan generar modelos modernos y complejos que puedan competir con el estado del arte en la tarea que busca resolver el usuario.

Todas las herramientas de este tipo son de código abierto y están escritas en el lenguaje de programación *Python*. Esta característica es completamente indispensable, pues de otra forma no se podría permitir a los usuarios generar modelos que compitan con el estado del arte, esto debido a que hoy en día todo el desarrollo e investigación en *Machine Learning* se hace en este lenguaje.

Algunas de las librerías más relevantes de este tipo son:

- Pytorch [13]: Esta librería está orientada al desarrollo y uso de redes neuronales profundas, dicha infraestructura es muy usada en la mayoría de las tareas relevantes hoy en día.
- Keras [11]: Herramienta similar a Pytorch, que busca reducir la cantidad de código necesaria para programar redes neuronales profundas.
- TensorFlow [4]: Esta librería es muy similar a Pytorch, es decir, está orientada al manejo de redes neurales profundas.
- Scikit-learn [14]: Esta librería se centra en el trabajo con datos tabulares (matrices numéricas), donde se encuentran disponibles una gran variedad de algoritmos clásicos para datos tabulares, como support-vector machines, random forests, gradient boosting y k-means.
- HugginFace [20]: Esta librería se centra en proveer el proceso conocido como *fine-tunning* a modelos basados en la arquitectura *transformers* [19], orientándolo principalmente a tareas de procesamiento de lenguaje natural.

La mayor fortaleza de este tipo de herramientas es que le permite a sus usuarios generar modelos completamente personalizados a la tarea que buscan resolver.

Ahora bien, dicha versatilidad repercute en que este tipo de herramientas presenta grandes barreras de entrada, principalmente para usuarios con poca experiencia en programación.

2.1.2. Herramientas Divulgativas

Estas son principalmente aplicaciones de escritorio que permiten a sus usuarios realizar la configuración, entrenamiento y evaluación de varios modelos de *Machine Learning*, asociados principalmente a la tarea de clasificación de datos tabulares.

Todas estas herramientas cuentan con una interfaz gráfica que les permiten a sus usuarios manejar los modelos de *Machine Learning* a través de clics en la pantalla. Este tipo de herramientas también suele presentar en la mayoría de sus menús configuraciones por defecto para facilitar y acelerar el uso de la aplicación.

Las herramientas más populares de este tipo son:

- WEKA [9]: Herramienta de código abierto programada en *Java* que permite realizar la configuración, entrenamiento, evaluación y predicción de modelos, todo a través de una interfaz gráfica interactiva. Enfocada principalmente en el procesamiento de datos tabulares.
- RapidMiner [12]: Herramienta pagada, también programada en el lenguaje *Java*, orientada al trabajo con datos tabulares y con una interfaz gráfica orientada a procesos.

- KNIME [6]: Herramienta de código abierto muy similar a RapidMiner.
- Orange [7]: Herramienta de código abierto similar a las herramientas anteriores pero desarrollada en el lenguaje *C++*. Esta aplicación es capaz de interactuar con algunas librerías de *Python* como *Scikit-learn* [14].

Gracias a la interfaz gráfica que manejan, este tipo de herramientas presenta barreras de entrada muchas más bajas que las herramientas científicas, lo que resulta muy atrayente para usuarios con poco conocimiento de programación.

Sin embargo, este tipo de herramientas también presenta dos grandes desventajas, la primera de ellas y más fundamental es que las herramientas no están escritas en el lenguaje de programación *Python* y la segunda es que estas presentan una arquitectura muy rígida. Estas desventajas provocan que sea engorroso el extender estas aplicaciones frente a nuevos modelos o nuevas tareas del estado del arte.

2.1.3. Herramientas *No-Code*

Estas son principalmente servicios de nube (o *cloud services* en inglés) y buscan combinar las fortalezas de las herramientas científicas con las de las herramientas divulgativas, es decir, ser lo suficiente versátiles para generar modelos del estado del arte pero con barreras de entrada bajas.

Algunos ejemplos de este tipo de servicios son:

- IBM Watson [3]: Servicio de nube enfocado en proveer a sus usuarios un entorno donde estos puedan entrenar y evaluar modelos, para ello los usuarios deben subir un conjunto de datos al servicio. El *software* además permite a sus usuarios usar los modelos ya entrenados para predecir nueva información.
- Hugging Face AutoTrain [2]: Servicio de nube enfocado en entrenar modelos de *Machine Learning* de forma automática, es decir, los usuarios solo deben proveer los datos y seleccionar los modelos a entrenar y la aplicación por sí sola ajusta los modelos para generar las mejores predicciones posibles.

Como ya se dijo antes este tipo de herramientas busca juntar lo mejor de las herramientas científicas con lo mejor de las herramientas divulgativas. Junto con lo anterior, este tipo de aplicaciones permite ejecutar el entrenamiento y evaluación de los modelos de *Machine Learning* fuera del computador del usuario, lo que resulta muy ventajoso debido a los altos tiempos que toman estos procesos, según se menciona en la Sección 1.2.

Ahora bien, la desventaja que tienen este tipo de herramientas es que no son de código abierto, lo que lleva a que los usuarios deban pagar altas tarifas por el uso de la aplicación (usualmente se realiza un pago dependiendo del tiempo de uso de los servidores al momento de entrenar modelos) y también que estos no permiten obtener mucha información sobre lo ocurrido con los modelos, lo que resulta de vital importancia para desarrolladores y científicos que apuntan a hacer modelos con resultados excepcionales.

2.1.4. Comparativas con DashAI

Como se menciona en el Sección 1.1, *DashAI* busca combinar todas las virtudes de las herramientas antes mencionadas, las cuales son:

- Ser de código abierto, esto con el objetivo de aumentar el número de usuarios que accedan a esta.
- Tener una ejecución local, esto con el fin de permitirle a sus usuarios obtener todos los datos que estimen relevantes de los modelos que entrenen, ya sea mediante el uso de la aplicación o de forma externa.
- Presentar una interfaz gráfica amigable con los usuarios, con el objetivo de disminuir las barreras de entrada de la aplicación.
- Ser extensible, esto con el fin de evitar la obsolescencia de la aplicación al no poder ajustarse al estado del arte.

Lo anterior se encuentra resumido en la siguiente figura.

	H. Científicas	H. Divulgativas	H. *No-Code*	DashAI
Open Source	✓	✓	✗	✓
Ejecución Local	✓	✓	✗	✓
Interfaz Gráfica	✗	✓	✓	✓
Extensible	✓	✓	✗	✓

Figura 2.1: Comparativa entre los distintos tipos de herramientas de *Machine Learning* y *DashAI*

2.2. Flujo de trabajo de Machine Learning

Esta sección busca presentar al lector cómo se estructura un flujo de trabajo clásico para trabajar con modelos de *Machine Learning*, para ello primero se expone un teorema que es una pieza fundamental en la investigación y desarrollo de modelos, para luego mostrar y explicar las etapas claves del flujo de trabajo de *Machine Learning*.

Existe en la literatura el llamado "*No free lunch theorem* "[21] el cual indica que no existe ningún modelo que sea el mejor para todos los problemas a resolver, esto debido a que las suposiciones tomadas para un problema no seguirán siendo ciertas para otro. Este teorema lleva a que siempre sea necesario buscar el mejor modelo para resolver un determinado problema.

Como es necesario experimentar cada vez que se quiere encontrar un modelo que solucione un determinado problema es que nace la noción de flujo de trabajo (o *workflow* en inglés) para utilizar modelos de *Machine Learning*.

Existen varios flujos de trabajo en la literatura, pero sin lugar a duda el más usado, tanto por investigadores como por las herramientas presentadas en la Sección 2.1, es el presentado en [10], el cual consiste en los siguientes pasos:

- Cargar datos: esta etapa consiste en seleccionar qué datos se usarán en los siguientes pasos, aquí lo más importante es que los datos a utilizar deben ser compatibles con el problema que uno busca resolver. También se suele aplicar una subetapa de limpieza y extracción de atributos, que consisten en transformar los datos (eliminar valores nulos, eliminar mayúsculas en caso de texto, etc.) con el fin de mejorar el rendimiento de los modelos a entrenar.
- Entrenar los modelos: esta etapa, como bien indica su nombre, consiste en el entrenamiento de modelos con el fin de encontrar cuál de ellos es el que genera mejores resultados para el problema que se busca resolver. Para lograr mejorar las predicciones de los modelos se realiza un proceso de ajuste de parámetros, el cual consiste en modificar variables del modelo, dicha modificación se parametriza por una función de pérdida, que indica que tan buenas fueron las predicciones del modelo.
- Evaluar los modelos: esta etapa se realiza luego de haber entrenado los modelos y consiste en el cálculo de una serie de métricas, es decir, funciones que indican qué tan cercanas a la realidad son las predicciones hechas por el modelo.

El flujo descrito anteriormente es un elemento fundamental para *DashAI* , pues permite estandarizar el uso de modelos de *Machine Learning*, posibilitando definir pasos y procesos necesarios para cumplir con las etapas antes mencionadas.

2.3. Precursores de DashAI

Para finalizar esta sección se presentan los distintos prototipos por los que pasó *DashAI* , esto con el fin de mostrar al lector los problemas que dichos prototipos tienen y lograr establecer una conexión entre estos y el trabajo realizado por el memorista, explicado en profundidad en el Capítulo 3 y el Capítulo 4. Cabe destacar que el memorista participó activamente en el diseño y elaboración de todos los prototipos que se mencionan.

2.3.1. clasificador-sentencias

Proyecto entre el Instituto Milenio Fundamento de los Datos (*IMFD*) y la Corte Suprema de Chile (*CS* para abreviar), que tiene como objetivo el desarrollo de una herramienta que permita a sus usuarios entrenar y evaluar modelos de *Machine Learning* desarrollados por los mismos usuarios, todo en el marco de la tarea de clasificación de texto, específicamente la categorización de sentencias judiciales con el fin de elaborar un buscador de sentencias judiciales. El enlace al repositorio es `https://github.com/felipesilvadv/clasificador-sentencias`.

La *CS* antes de la realización del proyecto se encontraba haciendo uso del servicio de nube *IBM Watson* [3], el cual no les era del todo útil pues no permitió que sus desarrolladores pudieran analizar los modelos que generaban.

El trabajo fue desarrollado por un grupo de estudiantes de ingeniería civil en computación y magíster en computación contratados por el *IMFD*, entre los cuales se encontraba el memorista, quienes contaban con el apoyo de un ingeniero del *IMFD* para realizar consultas sobre las tecnologías a utilizar en el desarrollo.

Este prototipo se encuentra escrito en el lenguaje de programación *Python*, no cuenta con una interfaz gráfica, pero sin con una *API* desarrollada con la librería *Flask*, la cual permite a sus usuarios realizar peticiones para entrenar modelos ya cargados en la aplicación.

Como ya se mencionó el *software* solo es compatible con la tarea de clasificación de texto, pero si es posible extender los modelos que se pueden utilizar.

Como la aplicación no cuenta con una interfaz gráfica su uso resulta complejo y requiere de mucho conocimiento sobre el funcionamiento de esta y de la escritura de peticiones gigantes. Es por ello que este prototipo nace la idea de contar con una interfaz gráfica que facilite la realización de las peticiones.

2.3.2. TextPerimenter

Herramienta sucesora de *clasificador-sentencias* desarrollada exclusivamente por el memorista en el marco de un trabajo dirigido bajo la tutela del profesor Felipe Bravo. Esta aplicación tiene objetivos similares a su predecesora, pero con la intención de ser más amigables con los usuarios, mediante la elaboración de una interfaz gráfica y poder ser extendida a más tareas, principalmente aquellas relacionadas con texto, de donde surge su nombre. El enlace al repositorio es `https://github.com/DashAISoftware/DashAIOld`, la implementación se encuentra específicamente la rama llamada *TextPerimenterLite*.

Este prototipo se encuentra escrito principalmente en el lenguaje de programación *Python*, usando la librería *Django* para manejar el *backend*. Con respecto a la interfaz gráfica esta pasó por una serie de modificaciones, en sus inicios se usaron los lenguajes *JavaScript* y *HTML* (tecnologías estándar en la elaboración de aplicaciones web con *Django*), pero luego se optó por utilizar la librería de *Python Dash*, debido a la popularidad que había ganado en dicho momento para la elaboración de aplicaciones web orientadas al análisis de datos.

Es al realizar estos cambios en la tecnología usada para la componente gráfica de la aplicación que se evidencia un gran problema que se venía arrastrando en el desarrollo de ambos prototipos, el cual es que no existe una separación clara entre la componente lógica de la aplicación y su interfaz gráfica. Dicho acoplamiento lleva a la idea de elaborar una *API* que permita separar completamente estas dos componentes, a modo de agilizar el desarrollo de la aplicación.

Capítulo 3

Diseño

Este capítulo tiene como propósito mostrar al lector los diseños elaborados por el memorista para cumplir con los objetivos planteados en la Sección 1.4.

Para ello este capítulo se estructura en dos secciones, la primera busca mostrar al lector la estructura que tiene *DashAI* y luego de ello se presenta en profundidad los diseños elaborados por el memorista.

Como se menciona en la Sección 1.5, *DashAI* es un proyecto colaborativo desarrollado por un equipo de ingenieros y estudiantes, es por ello que para contextualizar el trabajo hecho por el memorista es que se hace necesario exponer las partes que componen a la librería.

3.1. Visión general de DashAI

Esta sección tiene cómo objetivo presentar al lector como se estructura *DashAI* y qué partes lo componen, para de esa forma generar un contexto que permita explicar más fácilmente los módulos diseñados por el memorista.

La sección se divide en tres subsecciones, la primera expone la arquitectura de *DashAI* , describiendo brevemente los elementos que la componen, luego se muestran los componentes que forman parte de la herramienta y finalmente se muestran el uso típico de la aplicación.

Vale la pena destacar que mucho de lo que se expone en esta sección no fue realizado por el memorista, si no por el resto del equipo de trabajo de *DashAI* . El trabajo realizado exclusivamente por el memorista se encuentra explicado más adelante, en la Sección 3.2.

3.1.1. Arquitectura

DashAI presenta una versión modificada de la clásica arquitectura web, también conocida como arquitectura de 3 capas. Los elementos que componen esta arquitectura se pueden apreciar en la Figura 3.1.

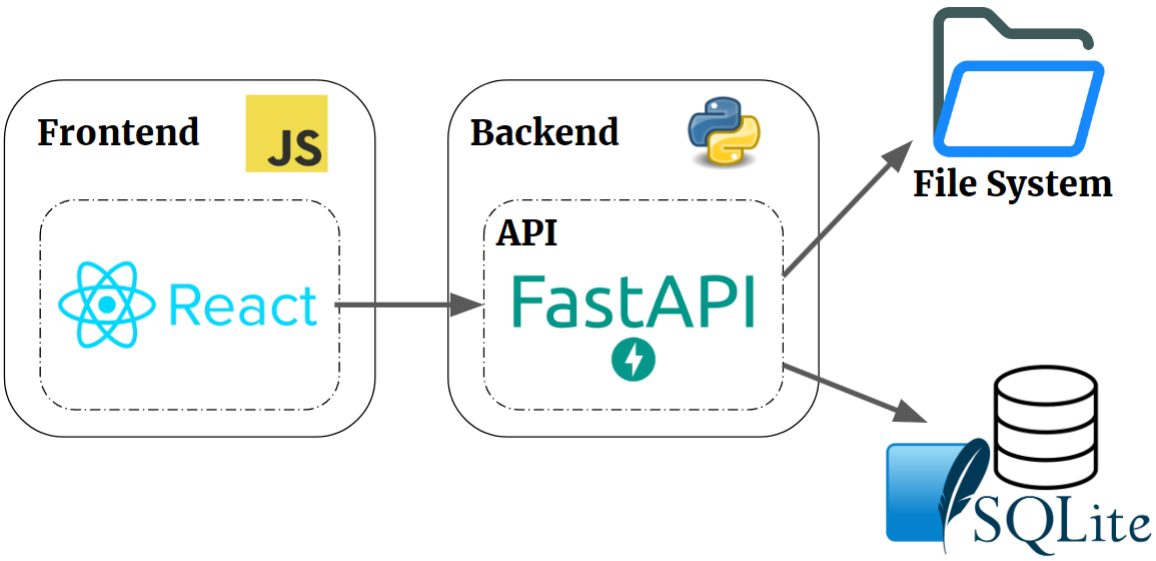

Figura 3.1: Arquitectura de *DashAI* .

- Frontend: Permite a los usuarios manejar todo el flujo de trabajo de modelos de *Machine Learning*, descrito en la Sección 2.2, mediante clics y la completación de formularios. Se encuentra escrito en el lenguaje *JavaScript* usando la librería *React*.
- Backend: Este se puede dividir en dos piezas fundamentales que funcionan en conjunto, la *API* y el núcleo o *core* de la librería. La *API* se encarga de estandarizar la comunicación entre el *frontend* y el *core* de *DashAI* , realizar las peticiones a la base de datos y crear y leer archivos del sistema de archivos del usuario. El núcleo de *DashAI* contiene la lógica de negocio de la aplicación, es decir, es donde se crean, entrenan y evalúan los modelos de *Machine Learning*. Este elemento se encuentra escrito en el lenguaje de programación *Python* y utiliza varias de las herramientas descritas en la Sección 2.1, junto con la librería *FastAPI*.
- Base de datos: Este se encarga de almacenar toda la información relevante del flujo de trabajo de modelos de *Machine Learning*, con el objetivo de poder retomar un flujo no terminado y también el poder realizar una comparación entre distintos modelos entrenados. Se utiliza para ello el gestor de bases de datos relacionales *SQLite*.
- Sistema de archivos: Gracias a que la aplicación está pensada para ser ejecutada de manera local, es posible contar con el sistema de archivos del usuario. Este se utiliza para almacenar los conjuntos de datos provistos por el usuario y los modelos que este busca entrenar. Estos archivos no se almacenan en la base de datos debido a que no es una buena práctica almacenar archivos pesados en bases de datos relacionales, esto según [17].

Para iniciar la aplicación tanto el frontend como el backend se ejecutan en el mismo proceso, esto con el fin de facilitar el uso de la herramienta. Para ello se utiliza la librería de *Python uvicorn*, la cual se usa comúnmente para disponibilizar servicios web de manera local.

3.1.2. Componentes

En esta subsección se presentan algunos de los componentes más relevantes de *DashAI* y que son usados más adelante por el memorista tanto en su diseño como en la implementación del mismo.

Dichos componentes pueden ser agrupados en tres grupos, componentes extensibles, componentes de datos y componentes variados, estos se describen a continuación.

3.1.2.1. Componentes Extensibles

Estos componentes son los responsables de dotar a *DashAI* de extensibilidad. Todos estos presentan interfaces abstractas que indican como pueden ser utilizados y presentan una serie de implementaciones que siguen dichas interfaces. Estos son:

- Dataloader: Componente que permite transformar un conjunto de datos por el usuario en un objeto de tipo *Dataset* (tipo provisto por la librería *HuggingFace*[20]), que posteriormente se convierte en un *DashAIDataset*, clase desarrollada por el equipo de *DashAI* y que permite estandarizar el formato de un conjunto de datos.
- Model: Componente que representa un modelo de *Machine Learning*. Gracias a la interfaz que define, este componente puede ser utilizado en el flujo de trabajo de modelos de *Machine Learning*, pues provee métodos para entrenar, realizar predicciones, almacenar el componente en un archivo y cargarlo desde un archivo.
- Task: Este componente representa la tarea que el usuario busca resolver con el uso de *Machine Learning*, como se explica más adelante, esta componente permite relacionar componentes son compatibles entre sí.
- Metric: Componente que permite evaluar las predicciones generadas por los modelos, le entrega al usuario un valor cuantitativo de qué tan correctas son las predicciones del modelo.
- JobQueue: Componente parecido a una cola que permite almacenar trabajos para ser ejecutados posteriormente. Un trabajo (o *job* en inglés) representa una invocación de función, guarda en su estructura una función y los parámetros a utilizar para invocarla, se suelen ocupar cuando se quiere ejecutar la función en otro momento o en otro proceso.

Actualmente *DashAI* cuenta con algunas implementaciones para dichas componentes, estas se muestran en la Figura 3.2.

El propósito de este tipo de componentes es permitirle al usuario seleccionar la que más se adecue a la tarea que busca resolver o en caso de no encontrar ninguna, que el usuario puede generar una implementación que le sea útil en su problema.

Ahora bien no todos los componentes se pueden utilizar en conjunto, un ejemplo de esto es que no es posible usar un modelo que espera recibir texto para clasificar imágenes. Para evitar dichos errores *DashAI* relaciona componentes con la *Task* con la que son compatibles.

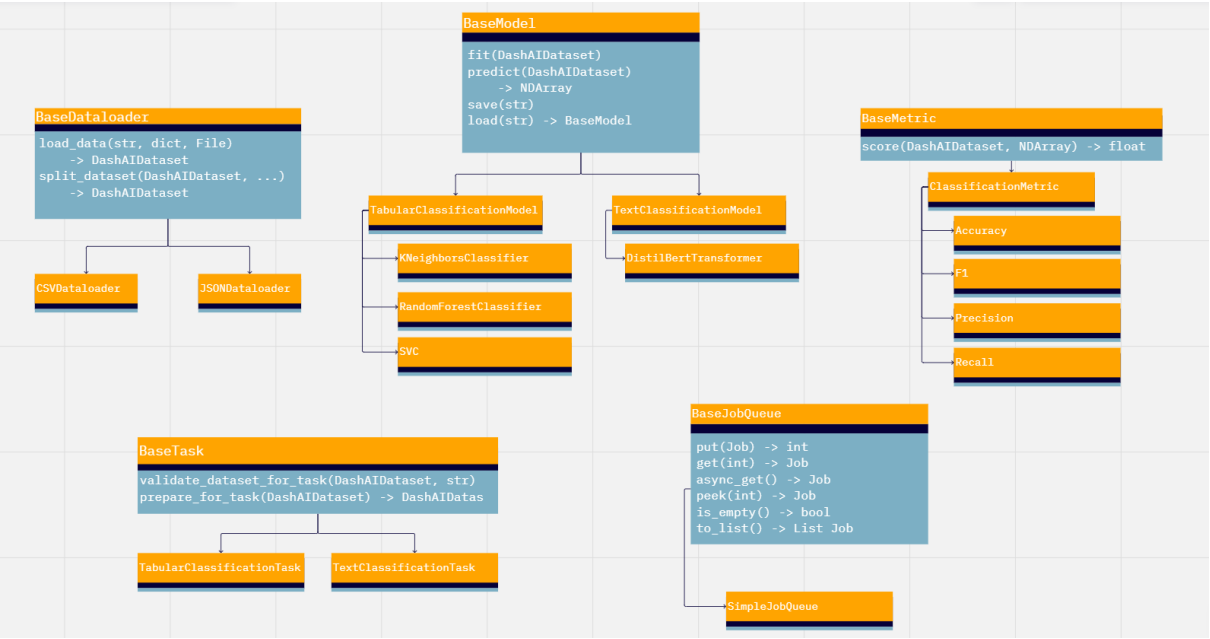

Figura 3.2: Diagrama de clases de DashAI.

Todos los componentes y sus relaciones se encuentran almacenados en un objeto de tipo *ComponentRegistry*, dicho tipo lo provee *DashAI* y presenta métodos que permiten realizar consultas para recuperar componentes relevantes.

3.1.2.2. Componentes de Datos

Este tipo de componente se relaciona directamente con el modelo de datos que utiliza *DashAI* en su base de datos. El modelo de datos de *DashAI* se encuentra en la Figura 3.3.

A continuación se presenta una breve descripción de las entidades presentes en el modelo de datos, estas son explicadas con más detalle en la Subsección 3.2.2

- Dataset: Entidad que almacena metadatos relevantes de un conjunto de datos subido por el usuario, se puede destacar que contiene la dirección donde se encuentra almacenado el *DashAIDataset* relativo al conjunto de datos subido por un usuario.
- Experiment: Entidad que representa un experimento realizado por un usuario, es decir, una tarea y los modelos a utilizar para resolverla.
- Run: Entidad que almacena metadatos relevantes de un modelo de *Machine Learning*, se puede destacar que almacena el nombre de la componente que representa y los parámetros a utilizar para entrenarla.

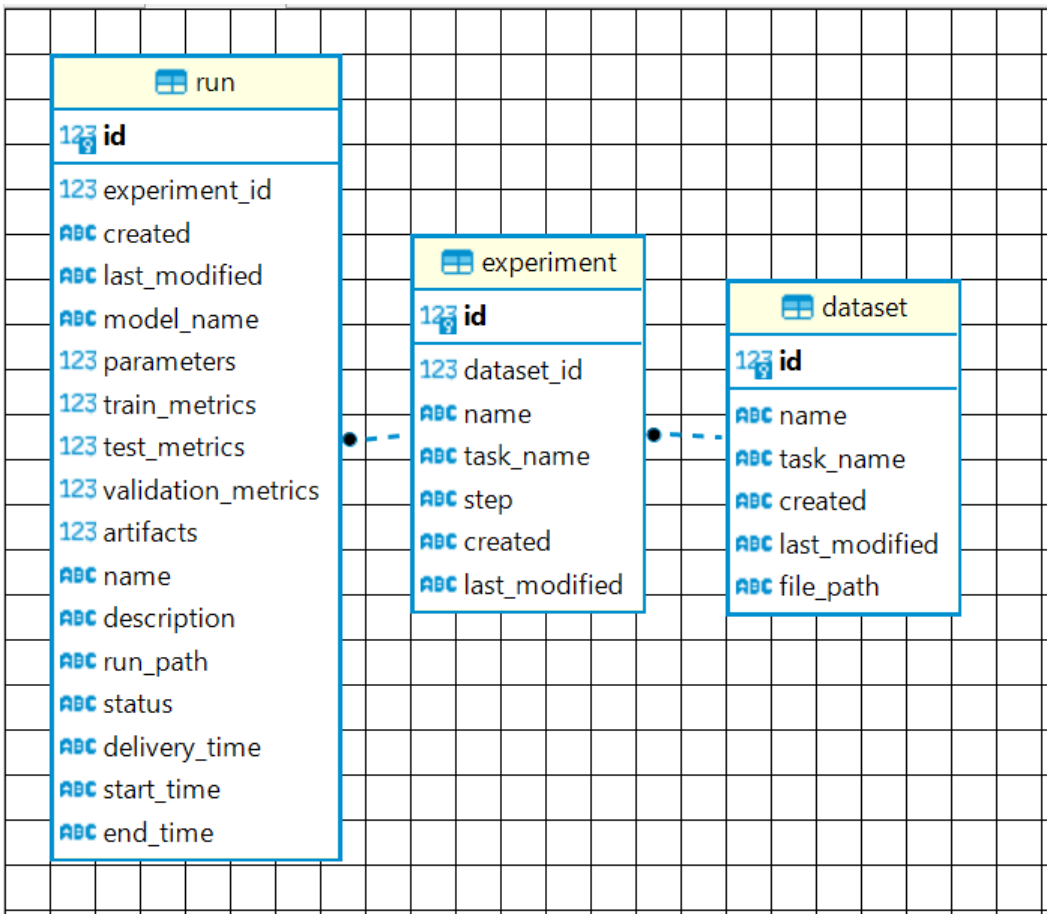

Figura 3.3: Modelo de datos de DashAI.

3.1.2.3. Componentes Variados

Este tipo de componentes no pertenecen a ninguna de las dos categorías expuestas anteriormente pero forman parte fundamental de *DashAI* . Estos son:

- *ComponentRegistry*: Componente parecida a un diccionario que permite almacenar componentes extensibles y sus relacionales y permite recuperarlos a través de la invocación de métodos.
- *DashAIDataset*: Estructura que representa un conjunto de datos subido por el usuario. Tiene como función estandarizar el formato de los datos utilizado en la herramienta. Para generarlo se hace el uso de una implementación del componente *Dataloader*.

3.1.3. Flujo

En esta subsección se muestran los pasos típicos que debe seguir un usuario para completar el entrenamiento de un modelo de *Machine Learning* en *DashAI* . Para ello se muestran imágenes de la interfaz gráfica de la aplicación junto con un breve texto explicando los procesos que se exponen.

Cabe destacar que el memorista no formó parte del desarrollo de la interfaz gráfica y por tanto esta no forma parte del trabajo de esta memoria, sin embargo, se hace necesario mostrarla para aclarar al lector como se espera que sea usado el *framework*.

Una vez un usuario instala y ejecuta *DashAI* , se muestra en su navegador una vista similar a la que aparece en la Figura 3.4, en ella el usuario puede elegir:

- Subir un conjunto de datos.
- Crear un experimento.
- Revisar resultados de experimentos previos.

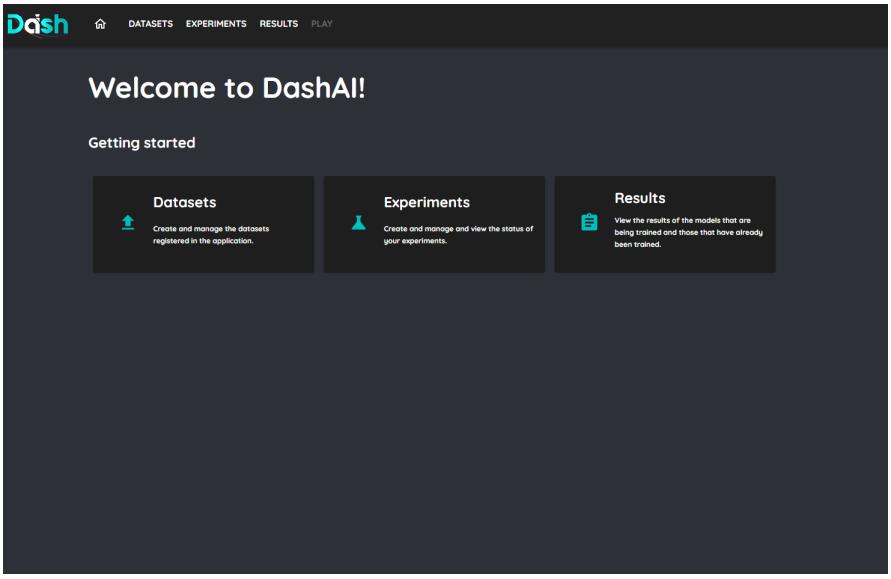

Figura 3.4: Página de inicio de DashAI.

Como se menciona en la Sección 2.2, lo primero que se debe tener para utilizar modelos de *Machine Learning* es un conjunto de datos, para ello el usuario debe ir a la pestaña *Datasets* y presionar en *New Dataset*, lo que lo lleva a las vistas que se muestran en la Figura 3.5, Figura 3.6 y Figura 3.7.

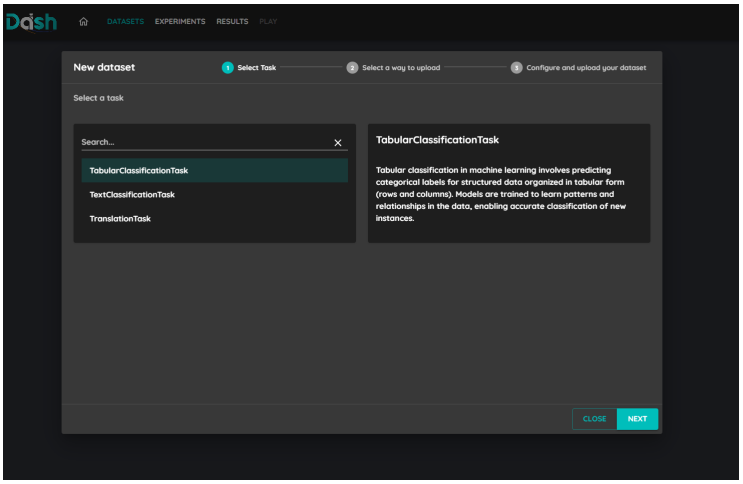

Figura 3.5: Subir un conjunto de datos, seleccionar la tarea.

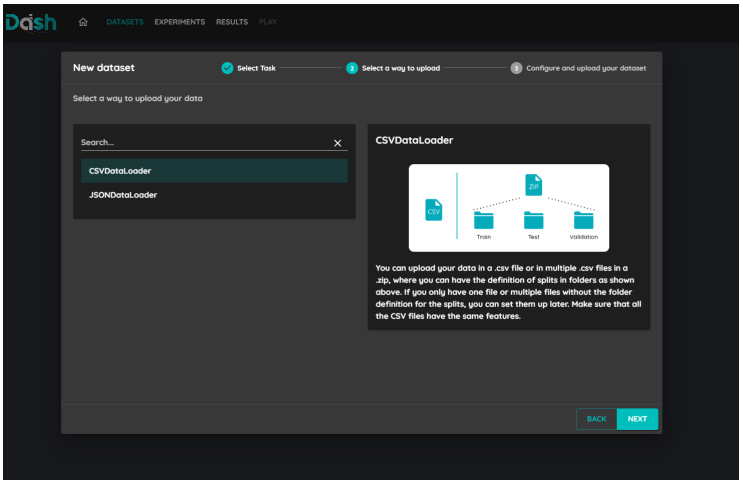

Figura 3.6: Subir un conjunto de datos, seleccionar el Dataloader a utilizar.

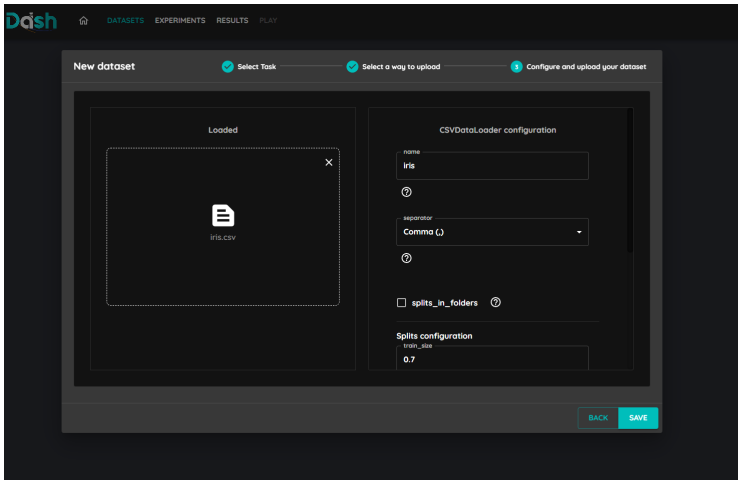

Figura 3.7: Subir un conjunto de datos, subir el archivo y nombrarlo.

Una vez el usuario el usuario ya cuenta con su conjunto de datos subido en *DashAI* este debe proceder a crear un experimento, dirigiéndose a la pestaña *Experiments* y presionando el botón *New Experiment*, esto lo lleva a las vistas que se muestran en la Figura 3.8, la Figura 3.9 y la Figura 3.10.

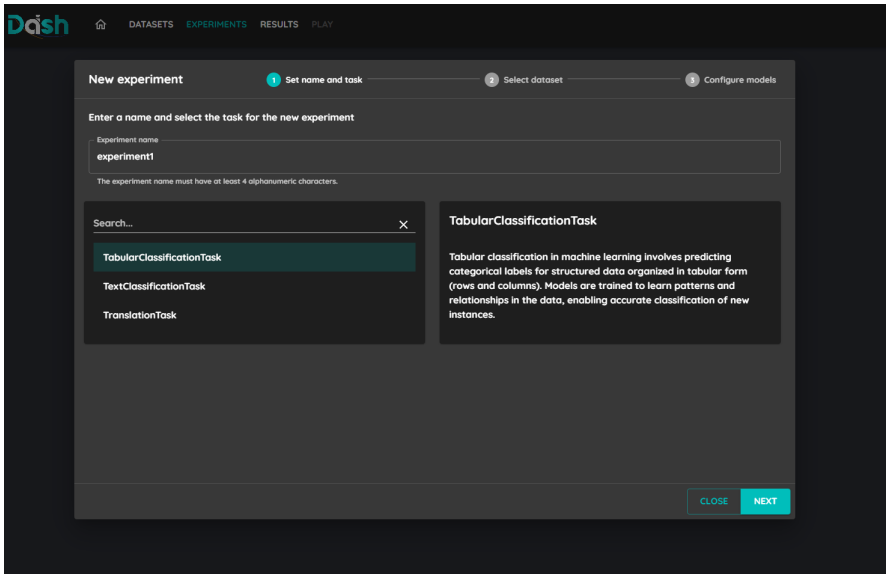

Figura 3.8: Crear un experimento, seleccionar la tarea.

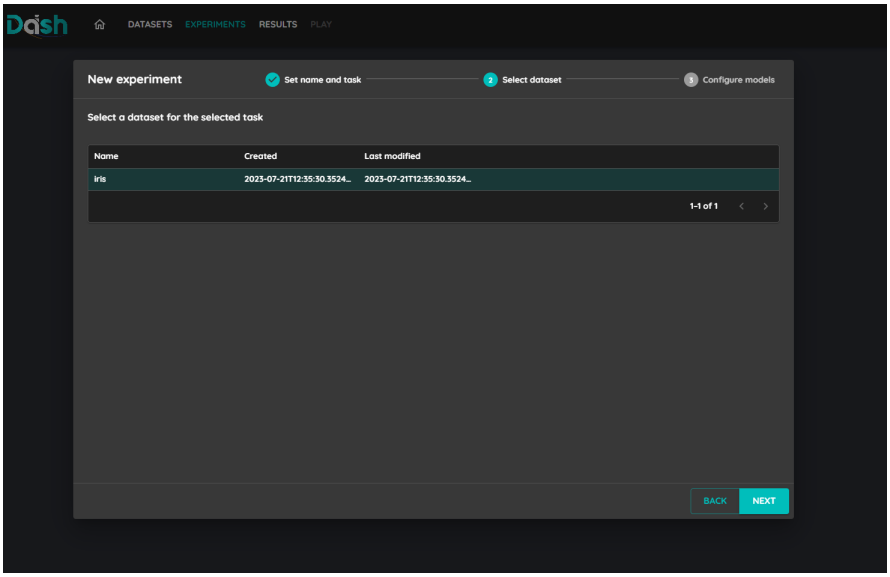

Figura 3.9: Crear un experimento, seleccionar el conjunto de datos a utilizar.

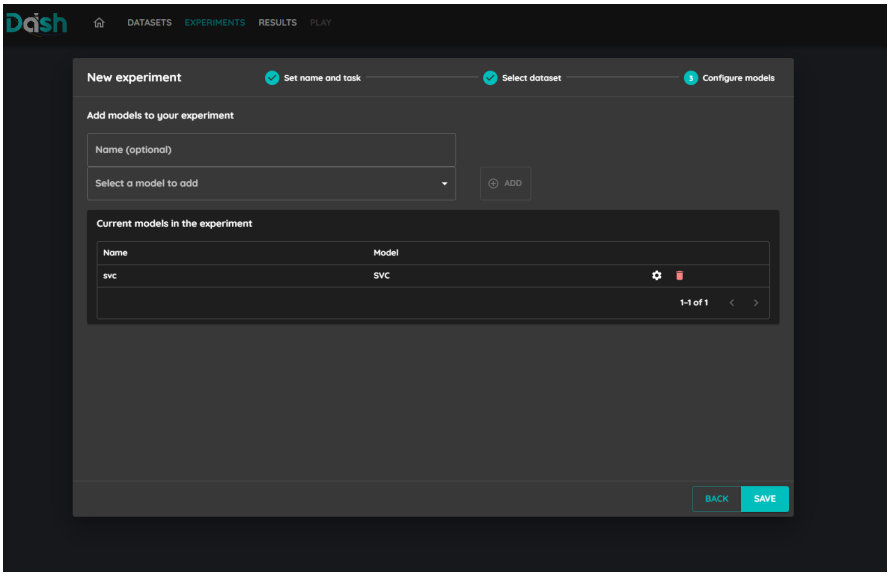

Figura 3.10: Crear un experimento, seleccionar los modelos a utilizar.

Una vez hecho esto el usuario puede mandar a entrenar los modelos del experimento, para ello debe presionar el botón *play* del experimento lo que lo lleva a la vista de la Figura 3.11, donde puede decidir qué modelos del experimento entrenar.

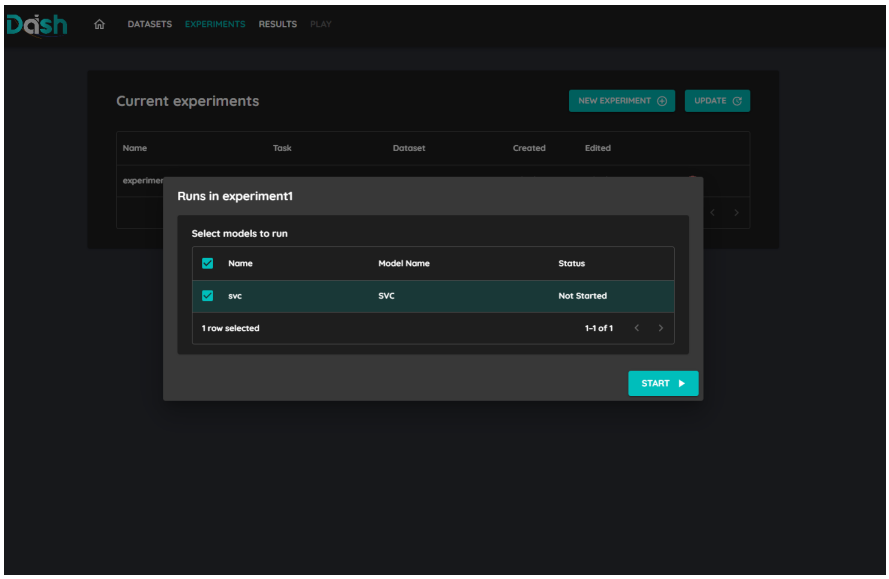

Figura 3.11: Mandar a entrenar los modelos de un experimento.

Finalmente el usuario puede revisar los resultados de los experimentos ya entrenados, para ello debe dirigirse a la pestaña *Results* y seleccionar el experimento a revisar, lo que lo lleva a una vista similar a la que se muestra en la Figura 3.12.

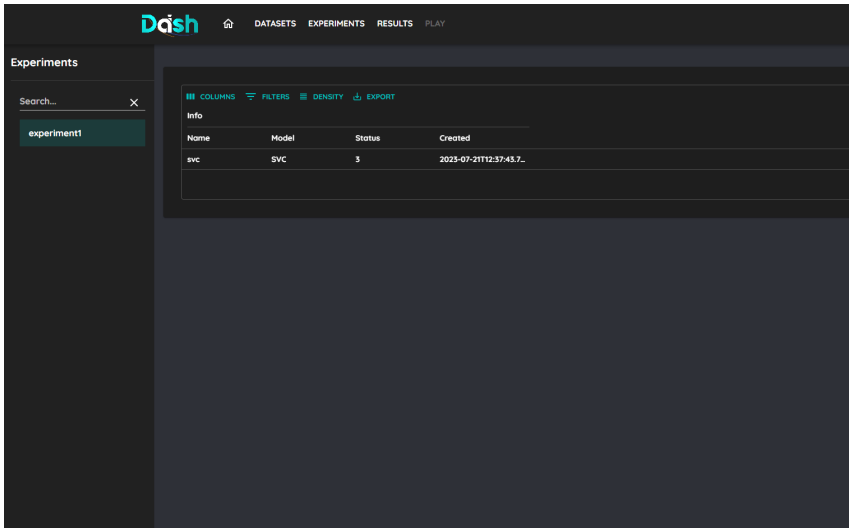

Figura 3.12: Revisar resultados, resultados de todos los modelos del experimento.

Además el usuario puede revisar los resultados de uno de los modelos del experimento presionando dos veces en el nombre del modelo, lo que lo lleva a la vista que se muestra en la Figura 3.13.

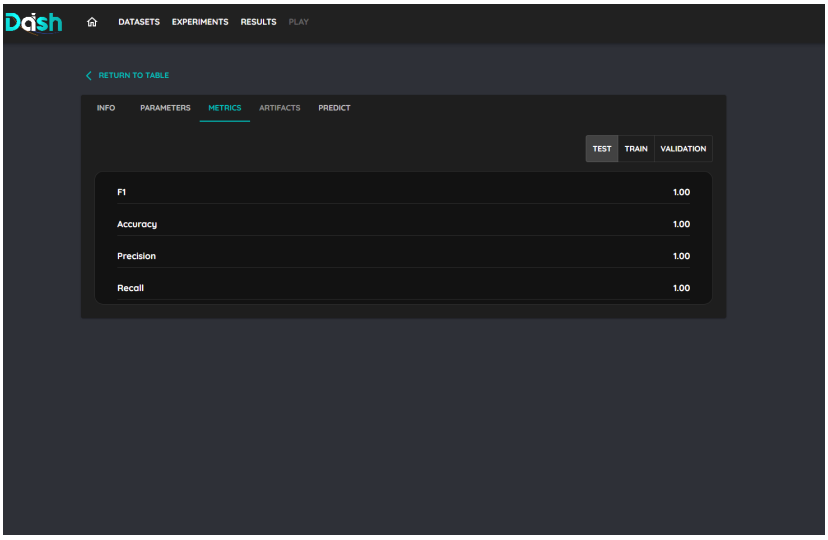

Figura 3.13: Revisar resultados, resultados de una *Run*.

3.2. Diseño del memorista

En esta sección se explica en profundidad el diseño de los módulos desarrollados por el memorista, los cuales fueron la *API*, el modelo de datos, la cola de procesos y los test asociados a estos. También se indican las razones de las decisiones de diseño tomadas y las tecnologías que se decidieron utilizar para desarrollarlas.

3.2.1. API

Como se ha dicho a lo largo del documento, el lenguaje de programación *Python* es sumamente importante en el área del *Machine Learning*, pues es el lenguaje más usado en las investigaciones relacionados al área y en el desarrollo de soluciones orientadas a *Machine Learning*.

Lo anterior lleva a que se escoja a *Python* como el lenguaje para desarrollar el *backend* de *DashAI* y por lo tanto su *API*. En cuanto a la librería a utilizar para desarrollar la *API* se escoge *FastAPI*, debido a la gran popularidad que tiene y lo sencillo que resulta implementar la *API*.

La *API* diseñada por el memorista presenta una arquitectura RESTful, esto quiere decir, según se menciona en [16], que la *API* presenta la siguientes características:

- Uso explícito de métodos *HTTP*: Esto quiere decir que cada método *HTTP* se utiliza para un único propósito, lo que facilita el uso de la *API*. El método GET se utiliza para obtener un recurso, el método POST para crearlo, el método PUT para actualizarlo y el método DELETE para eliminarlo del servidor. Un recurso es un objeto con el que trabaja la aplicación web, puede ser la información de un usuario, el detalle de un producto, etc..
- Sin estado: Esta característica apunta a que las peticiones hechas a la *API* no dependan de otras hechas anteriormente. Gracias a esta propiedad las *APIs* se vuelven más escalables y simplifican el diseño de las mismas.
- *URIs* similares a estructura de carpetas: Una URI (sigla en inglés de *Uniform Resource Identifier*) es una secuencia de caracteres que identifica un recurso web. Con esta característica se busca que las *URIs* declaradas sean lo más autoexplicativas posible, esto con el fin de facilitar el uso de la misma para sus usuarios. Algunas recomendaciones son el uso de sustantivos y el nombre de recursos que provee la aplicación web.
- Representar recursos en formato XML o JSON: Esto consiste en utilizar alguno de estos dos formatos para representar los recursos que provee la aplicación, estos formatos son de uso estándar en la web, lo que genera que la *API* pueda fácilmente ser compatible con los usuarios que la utilicen.

Siguiendo con lo anterior, los *endpoints* de la *API* desarrollada se estructuran en base a los recursos que provee el *backend* de *DashAI* , los cuales son *Dataset*, *Experiment*, *Run* y *Job*.

Para cada uno de los recursos mencionados previamente se implementan los métodos *HTTP* descritos más arriba, permitiendo manejarlos mediante una interfaz CRUD (sigla en inglés de *Create Read Update Delete*).

3.2.2. Modelo de datos

Como se menciona en la Subsección 3.1.1, para administrar la base de datos se utiliza el gestor de bases de datos relacionales *SQLite*. Se prefiere utilizar dicho gestor antes de otros más consolidados en el desarrollo de aplicaciones como *MySQL* o *PostgreSQL* debido a que no requiere de ninguna instalación por parte del usuario, lo que aumenta el número de usuarios que puede acceder al *framework*.

El modelo de datos diseñado por el memorista se encuentra en la Figura 3.3, a continuación se describen las entidades presentes en él y los campos que estas contienen.

3.2.2.1. Dataset

Entidad de la base de datos que almacena metadatos relevantes de una conjunto de datos subidos por el usuario.

- *id*: entero que identifica el *dataset*.
- *name*: nombre dado por el usuario para identificar el *dataset*.
- *task_name*: nombre de la tarea asociada al conjunto de datos, permite filtrar los *datasets* que son compatibles con la tarea que busca resolver el usuario.
- *created*: marca de tiempo que indica el instante en el que fue creado el *dataset*.
- *last_modified*: marca de tiempo que indica el instante en el que fue modificado por última vez el *dataset*.
- *file_path*: ruta donde se encuentra guardado el *DashAIDataset* asociado al *dataset*.

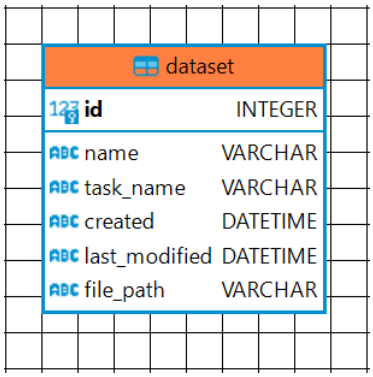

Figura 3.14: Entidad *dataset* del modelo de datos de DashAI.

3.2.2.2. Experiment

Entidad de la base de datos que representa un experimento realizado por el usuarios. Contiene metadatos relevantes a la tarea y los modelos seleccionados por el usuario.

- *id*: entero que identifica el *experiment*.
- *dataset_id*: entero que identifica el *dataset* asociado al experimento.
- *name*: nombre dado por el usuario para identificar el experimento.
- *task_name*: nombre de la tarea asociada al experimento.
- created: marca de tiempo que indica el instante en el que fue creado el *experiment*.
- *last_modified*: marca de tiempo que indica el instante en el que fue modificado por última vez el *experiment*.

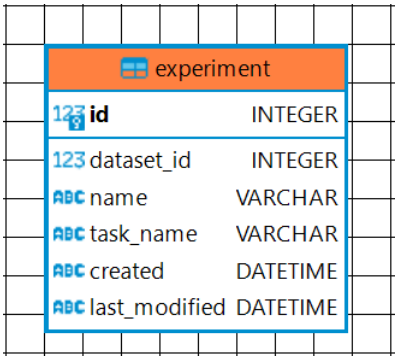

Figura 3.15: Entidad *experiment* del modelo de datos de DashAI.

3.2.2.3. Run

Entidad de la base de datos que almacena metadatos de un modelo de *Machine Learning* seleccionado por el usuario para incluirlo en un experimento.

- *id*: entero que identifica la *run*.
- *experiment_id*: entero que identifica el *experiment* asociado a la *run*.
- *created*: marca de tiempo que indica el instante en el que fue creada la *run*.
- *last_modified*: marca de tiempo que indica el instante en el que fue modificada por última vez la *run*.
- *model_name*: nombre del modelo de *Machine Learning* que la *run* representa.
- parameters: campo de tipo *JSON* (tipo parecido a un diccionario) que almacena el valor dado por el usuario a los parámetros del modelo que representa la *run*.
- *train_metrics*: campo de tipo *JSON* que contiene los resultados obtenidos por el modelo en la partición de entrenamiento del conjunto de datos. Contiene una entrada (llave y valor) por cada métrica relacionada con la tarea que el modelo busca resolver.
- *test_metrics*: campo de tipo *JSON* que contiene los resultados obtenidos por el modelo en la partición de evaluación del conjunto de datos. Contiene una entrada (llave y valor) por cada métrica relacionada con la tarea que el modelo busca resolver.

- *validation_metrics*: campo de tipo *JSON* que contiene los resultados obtenidos por el modelo en la partición de validación del conjunto de datos. Contiene una entrada (llave y valor) por cada métrica relacionada con la tarea que el modelo busca resolver.
- *artifacts*: campo de tipo *JSON* que contiene gráficos y estadísticas relacionadas a los resultados del modelo.
- *name*: nombre dado por el usuario para identificar la *run*.
- *description*: descripción dada por el usuario para detallar la *run*.
- *run_path*: ruta en la que se encuentra almacenado el modelo de *Machine Learning* asociado a la *run*.
- *status*: cadena de caracteres que indica el estado en el que se encuentra la *run*.
- *delivery_time*: instante de tiempo en que el la *run* fue encolada en la cola de trabajos para ser entrenada.
- *start_time*: instante de tiempo en que la *run* comienza a entrenarse.
- *end_time*: instante de tiempo en que la *run* termina su entrenamiento.

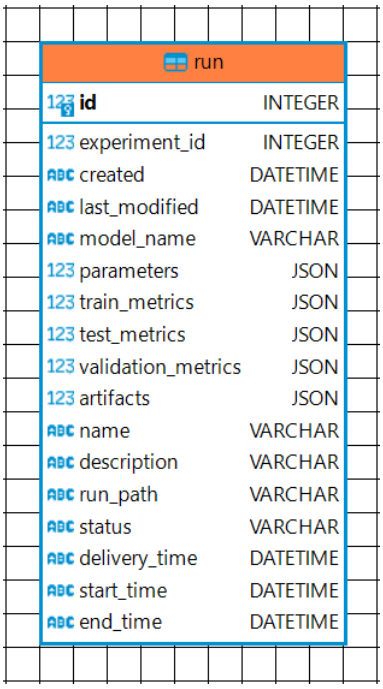

Figura 3.16: Entidad run del modelo de datos de DashAI.

En el modelo de datos descrito anteriormente tanto la relación entre *Dataset* y *Experiment* como entre *Experiment* y *Run* es *One-to-Many*, es decir, un *Dataset* puede estar relacionado con *n Experiments*, pero un *Experiment* sólo puede estar relacionado con un *Dataset*, es análogo para la relación entre *Experiment* y *Run*.

El objetivo de este modelo de datos es permitir que el usuario mantenga puntos de guardado (o *checkpoints* en inglés) de tal manera que este pueda retomar un entrenamiento que ha dejado pendiente o también revisar los resultados de un entrenamiento previo.

En el modelo de datos existen los campos *file_path* y *run_path* relacionados a las entidades *dataset* y *run* respectivamente. Dichos campos almacenan la ruta donde se encuentra un archivo que *DashAI* transforma un *DashAIDataset* o *Model*. Si bien se podría almacenar directamente el archivo en la base de datos, según explican en [17] no es recomendable hacerlo debido a que el escribir y leer en una base de datos es lento en comparación a realizar esas operaciones en el sistema de archivos, por lo que se opta por almacenar los archivos en el sistema de archivos y guardar la ruta de dichos archivos en la base de datos.

3.2.3. Cola de trabajos

Como se menciona en la Sección 1.2, el entrenamiento de modelos de *Machine Learning* puede tomar varios minutos e incluso horas en completarse, es por ello que nace en *DashAI* la necesidad de ejecutar dicho proceso de forma independiente a la *API*, para que el usuario pueda seguir utilizando la aplicación.

Es para resolver dicho problema que se propone la elaboración de una cola de procesos o trabajos. Un trabajo (o *job* en inglés) representa la invocación de una función, como se menciona en la Subsección 3.1.2.

La cola de trabajos diseñada por el memorista consiste en una cola *FIFO* (sigla en inglés de *First In First Out*) que almacena temporalmente los *jobs* creados por el usuario, hasta que estos son recuperados por otro proceso para ser ejecutados. Para interactuar con la cola de trabajos se define el *ADT* (sigla en inglés de *Abstract Data Type*) que se muestra en la Figura 3.17.

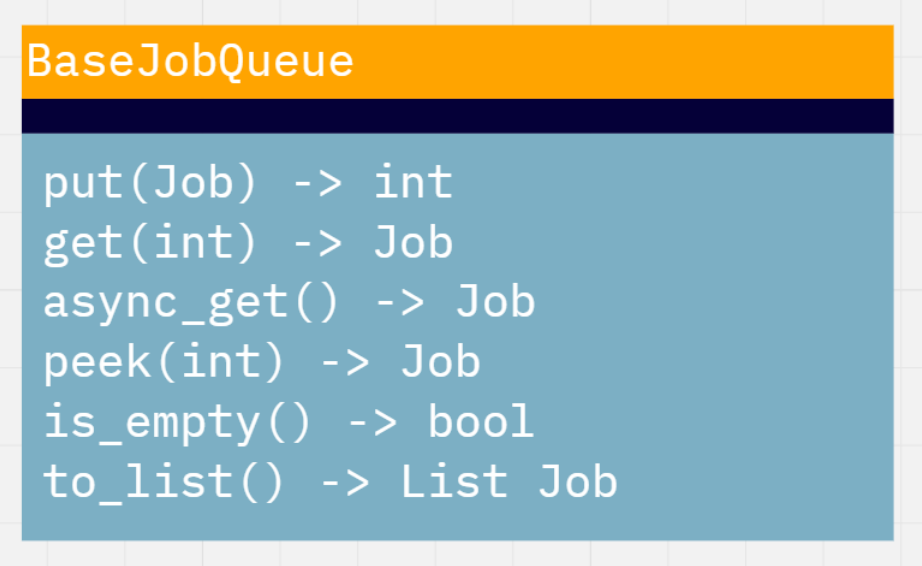

Figura 3.17: ADT de JobQueue.

A continuación se detallan lo métodos presentes en el ADT de la cola de trabajos:

- `put(Job)`: Encola un *job* en la cola de trabajos, recibe como argumento el trabajo a encolar y retorna un entero que permite identificar al *Job* encolado.

- `get(int)`: Permite extraer un trabajo de la cola, para ello recibe como argumento el identificador del *job* a extraer y retorna el trabajo extraido. En caso de no entregar un identificador se extrae el primer *job* presente en la cola. Entrega un error en caso de que la cola se encuentre vacia.
- `async_get()`: Extrae el primer trabajo de la cola, pero en caso de que la cola este vacia este método espera asíncronamente hasta que se encole un nuevo trabajo.
- `peek(int)`: Mismo funcionamiento que el método *get(int)* salvo que en vez de extraer los *jobs* de la cola este los recupera sin eliminarlos de la cola de trabajos.
- `is_empty()`: Predicado que permite saber si la cola se encuentra vacia o no. Retorna *True* en caso de que la cola este vacia.
- `to_list()`: Método que permite recuperar (sin extraer) todos los trabajos presentes en la cola, retornando un lista con dichos *jobs*.

El usuario puede administrar los trabajos de la cola de trabajos a través de la *API*, con ella el usuario puede encolar y cancelar *jobs*, además de ver los trabajos que se encuentran actualmente en la cola. La *API* no permite modificar trabajos activos en la cola pues solo se podría modificar la *Run* que se quiere entrenar, lo que sería en esencia un *job* distinto.

Para recuperar los trabajos de la cola y ejecutarlos, es decir, entrenar los modelos de *Machine Learning*, se utiliza un proceso que corre en el mismo proceso de la *API* pero de forma asíncrona.

La asincronía, también llamada concurrencia, es un término usado en computación para caracterizar el comportamiento de un proceso. Es común que ciertos procesos tengan que esperar a que se realicen ciertas operaciones lentas para continuar su ejecución, un ejemplo de esto es cuando se lee un archivo o se espera la respuesta de un servidor. En el caso de procesos asíncronos, el computador utiliza dicho tiempo de espera para ejecutar otro proceso pendiente y de esta forma aprovechar de mejor forma la CPU.

En el caso de una *API*, la asincronía se puede ver reflejada en la forma que esta contesta a las peticiones de los usuario, un *endpoint* asíncrono se comporta de manera normal pero continúa realizando procesamiento aun después de contestar la petición del usuario, lo que permite que este pueda realizar nuevas peticiones a la *API*. Este funcionamiento se encuentra representado en la Figura 3.18

Gracias al proceso asíncrono mencionado anteriormente es posible realizar el entrenamiento de modelos sin bloquear la *API*, de forma que el usuario puede seguir utilizando el *framework*.

Para lograr ejecutar el proceso de forma asíncrona se utilizan *BackgroundTasks*, una herramienta provista por la librería *FastAPI* para ejecutar procesos luego de responder una petición del usuario.

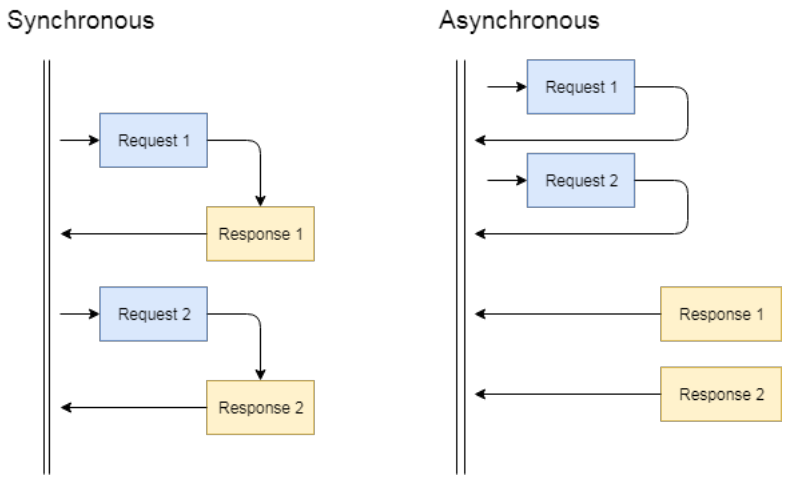

Figura 3.18: Ejemplo de *API* síncrona vs *API* asíncrona.

3.2.4. Testing

Como se menciona en la Sección 1.6, si bien no se contemplan pruebas con usuarios para evaluar este trabajo si se espera la realización de *testing* para comprobar el correcto funcionamiento de los módulos implementados por el memorista.

Un *test* es una función que busca evaluar un programa, para ello verifica el retorno del programa y las modificaciones que este hace a variables mutables, por ejemplo una base de datos o el sistema de archivos.

Los *test* desarrollados por el memorista se encuentran escritos en el lenguaje *Python*, siguiendo el estilo declarado por la librería *Pytest*.

Para que *Pytest* considere que una función es un *test* esta debe cumplir que su nombre tenga como prefijo o sufijo la palabra *test*. La forma de comprobar que se cumple la funcionalidad esperada por el programa es mediante el uso de la *keyword assert expr*, la cual arroja un error en caso de que la expresión entregada evalúa a falso.

Pytest permite también la creación de *fixtures*, las cuales son funciones que se ejecutan antes que los *test* y tienen como objetivo preparar el contexto en el cual se ejecuta el *test*. Para que una función sea considerada un *fixture* por *Pytest* esta debe tener el decorador *pytest.fixture*.

Un decorador en *Python* es una función que se coloca por sobre la definición de otra, anteponiéndole el signo . Dicha función se ejecuta tomando como argumento la función que la sucede.

Los *test* diseñados por el memorista se enfocan en evaluar los códigos de retorno de la *API* frente a casos de uso normal y situaciones de borde, junto que verificar que esta genere los cambios necesarios en la base de datos y el sistema de archivos. Para evaluar la implementación de la cola de trabajos se verifica que esta cumpla con el *ADT* que se encuentra definido en la Subsección 3.2.3.

Capítulo 4

Implementación

En este capítulo se muestra el código desarrollado por el memorista para implementar el diseño expuesto en el Capítulo 3.

En la primera sección del capítulo se exhibe la implementación del modelo de datos descrito en la Subsección 3.2.2.

En la segunda sección se presenta la implementación de la *API* y los *test* respectivos a esta, para ello la sección se divide en subsecciones relacionadas a los *endpoints* desarrollados.

En la tercera sección se muestra la implementación de la cola de trabajos y elementos relacionados a esta junto con los test unitarios que evalúan su funcionalidad.

4.1. Modelo de datos

Para implementar el modelo de datos y lograr dejar este disponible para su uso desde el *backend* de *DashAI* se utiliza la librería de *Python SQLAlchemy* [5]. Esta librería permite declarar el modelo de datos mediante la implementación de clases.

Las clases implementadas deben heredar de la clase *DeclarativeBase* provista por la librería y definir campos de igual nombre y tipo que los declarados en el modelo de datos. Además se deben declarar campos extras que indiquen la relación existente entre las entidades.

En el apéndice A se encuentran las clases implementadas para cada una de las entidades del modelo de datos presente en la Figura 3.3.

De la implementación se puede destacar que los campos *created* y *last_modified* se inicializan de forma automática, y este último modifica su valor cada vez que se actualiza algún campo de una instancia de la entidad.

También vale la pena mencionar al campo *status*, que toma su valor de alguna de las opciones de la enumeración *RunStatus*, la cual presenta los siguientes valores:

- *NOT_STARTED* indica que la *Run* fue creada pero no se ha mandado a entrenar.
- *DELIVERED* indica que la *Run* fue entregada a la cola de trabajos para ser entrenada y evaluada.
- *STARTED* indica que la *Run* ya está siendo entrenada.
- *FINISHED* indica que la *Run* ya terminó su entrenamiento.
- *ERROR* indica que algún proceso relacionado con la *Run* falló.

Para modificar el campo *status* se hace uso de los métodos:

- *set_status_as_delivered*: cambia el valor del campos *status* a *DELIVERED* y el valor del campo *delivered_time* al instante actual.
- *set_status_as_started*: cambia el valor del campos *status* a *STARTED* y el valor del campo *start_time* al instante actual.
- *set_status_as_finished*: cambia el valor del campos *status* a *FINISHED* y el valor del campo *end_time* al instante actual.
- *set_status_as_error*: cambia el valor del campo *status* a *ERROR*.

4.2. API

En esta sección se muestran las implementaciones hechas por el memorista en relación a la *API* de *DashAI* . Para ello esta sección se divide en subsecciones relacionadas con las *URIs* definidas en la *API*, las cuales son `/dataset`, `/experiment`, `/run` y `/job`.

En cada una de dichas subsecciones se describe la funcionalidad principal que busca proveer la *URI* y los *endpoints* que se implementan para ello, estos se encuentran agrupados según el método *HTTP* que implementan.

Para cada *endpoint* se indica su objetivo, los argumentos que recibe, el resultado que entrega y los distintos errores que se pueden producir durante su ejecución. También se incluye un diagrama que explica su ejecución y una breve descripción de los test asociados a este.

4.2.1. Dataset

Esta *URI* se encuentra relacionada directamente al manejo de los conjuntos de datos que se suben y utilizan en *DashAI* . Con esta *URI* es posible crear, obtener, modificar y eliminar conjuntos de datos del *framework*.

Vale la pena recordar que en *DashAI* existen dos tipos de representaciones para un conjunto de datos, como se menciona en Subsección 3.1.2, si bien esta *URI* guarda estrecha relación con el *Dataset* de la base de datos, esta también se encarga de la creación y guardado del *DashAIDataset* asociado.

4.2.1.1. GET

Los *endpoints* disponibles para la *URI* `/dataset` con método *HTTP* GET son:

4.2.1.1.1 `/dataset/`

Este *endpoint* permite obtener todos los *Datasets* presentes en la base de datos de *DashAI* .

No recibe argumentos del usuario y retorna una lista de diccionarios que representan cada una de las instancias de *Datasets* presentes en la base de datos.

Puede fallar debido a un error de conexión con la base de datos, en cuyo caso se le entrega un error *HTTP* 500 al usuario, indicando el problema.

El código del *endpoint* se encuentra en la Subsubsección B.1.1.1, el diagrama presente en la Figura 4.1 expone el flujo del *endpoint*.

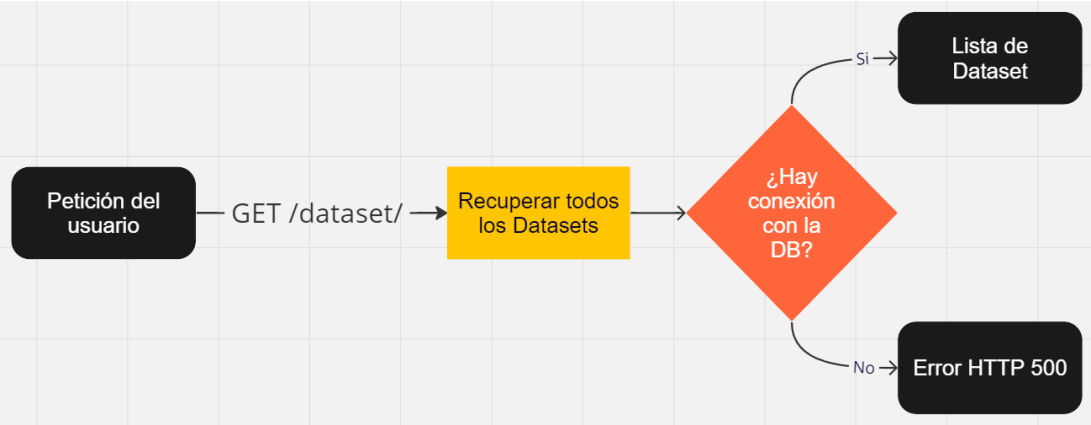

Figura 4.1: Diagrama de flujo *endpoint* GET `/dataset/`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.1.1.1, aquí se utiliza el *endpoint* para recuperar los dos *Datasets* creados por el *test* presente en el Párrafo D.1.1.2.1. El *endpoint* retorna una lista por lo que se verifica que en su primer componente se encuentre el *Dataset* con nombre *test_csv* y en su segundo componente el *Dataset* con nombre *test_csv2*.

4.2.1.1.2 `/dataset/"dataset_id"`

Este *endpoint* permite obtener el *Dataset* que tiene el campo *id* con valor *dataset_id* de la base de datos.

Recibe como argumento el *id* del *Dataset* a recuperar y retorna un diccionario que representa al *Dataset* consultado por el usuario.

Puede fallar en caso de no encontrar el *Dataset* buscado, entregando un error *HTTP* 404 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.1.1.2, el diagrama presente en la Figura 4.2 expone el flujo del *endpoint*.

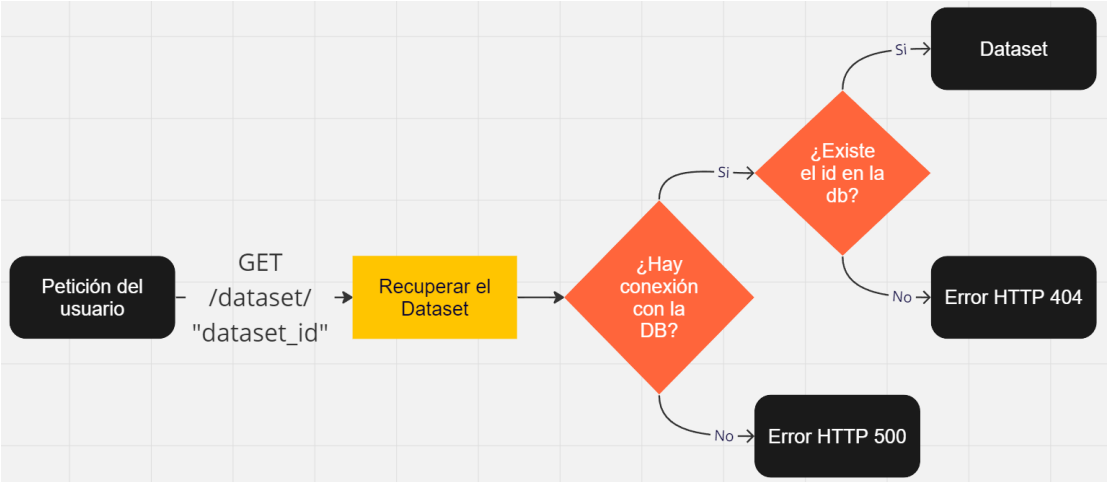

Figura 4.2: Diagrama de flujo *endpoint* GET `/dataset/"dataset_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.1.1.2, aquí se utiliza el *endpoint* para recuperar un *Dataset* inexistente, validando que se entregue un error *HTTP* 404.

4.2.1.2. POST

Para el método *HTTP* POST sólo se implementa un *endpoint* de la *URI* `/dataset`, el cual es:

4.2.1.2.1 `/dataset/`

Este *endpoint* permite que el usuario suba un conjunto de datos que él posea para ser utilizado posteriormente dentro del flujo de trabajo de *DashAI* .

Recibe como argumento parámetros de configuración del proceso, entre los que se destaca el Dataloader y parámetros para particionar el conjunto de datos, y el conjunto de datos a subir, el cual se puede subir mediante un archivo o entregando la URL donde se puede descargar. Retorna un diccionario que representa al *Dataset* creado.

Puede fallar en caso de encontrarse otro *Dataset* con el mismo nombre, entregando un error *HTTP* 409, en caso de no poder leer el archivo, entregando un error *HTTP* 500 y en caso de no poder conectarse a la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.1.2.1, el diagrama presente en la Figura 4.3 expone el flujo del *endpoint*.

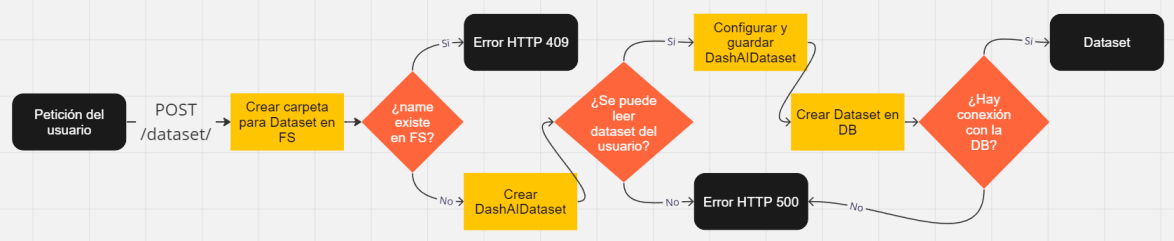

Figura 4.3: Diagrama de flujo *endpoint* POST `/dataset/`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.1.2.1, en él se crean dos instancias de *Dataset* relativas al conjunto de datos iris [8] con distinto nombre. Luego se consulta a la *API* por dichos *Datasets*, verificando que tengan los parámetros declarados al momento de crearlos.

4.2.1.3. PATCH

Para el método *HTTP* PATCH sólo se implementa un *endpoint* de la *URI* `/dataset`, el cual es:

4.2.1.3.1 `/dataset/"dataset_id"`

Este *endpoint* permite que el usuario modifique alguno de los metadatos presentes en la instancia de la entidad *Dataset* con *id dataset_id*.

Recibe como argumento el *id* del *Dataset* a modificar, el nuevo nombre y la nueva tarea del *Dataset* y retorna un diccionario que representa al *Dataset* modificado. Tanto el nombre como la tarea son argumentos opcionales.

Puede fallar en caso de que no se entregue ni nombre ni tarea como argumento, entregando un error *HTTP* 304 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.1.3.1, el diagrama presente en la Figura 4.4 expone el flujo del *endpoint*.

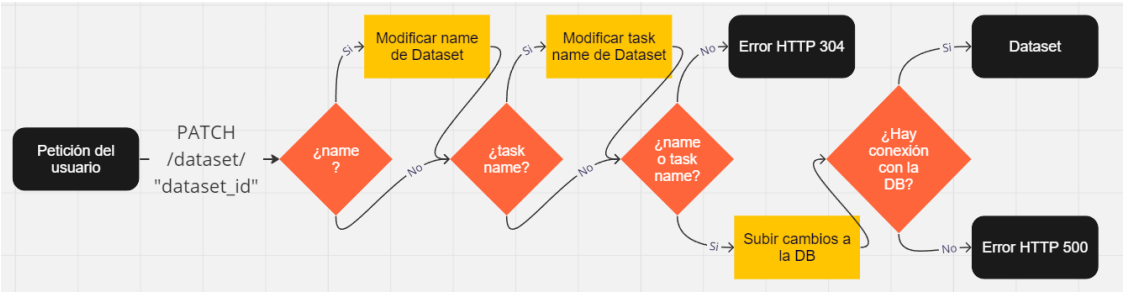

Figura 4.4: Diagrama de flujo *endpoint* PATCH `/dataset/"dataset_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.1.3.1, en este *test* se modifica tanto el nombre como la tarea asociada al *Dataset* con *id* 2 y luego se verifica que dicha modificación haya ocurrido en la base de datos.

4.2.1.4. DELETE

Para el método *HTTP* DELETE sólo se implementa un *endpoint* de la *URI* `/dataset`, el cual es:

4.2.1.4.1 `/dataset/"dataset_id"`

Este *endpoint* permite que el usuario elimine un conjunto de datos de la aplicación, tanto el *DashAIDataset* almacenado en el sistema de archivos como el *Dataset* guardado en la base de datos.

Recibe como argumento el *id* del *Dataset* a eliminar y retorna un mensaje al usuario que indica que el conjunto de datos fue eliminado satisfactoriamente.

Puede fallar en caso de no encontrar el *Dataset* buscado, entregando un error *HTTP* 404, en caso de no encontrar el DashAIDataset almacenado en el sistema de archivos, entregando un error *HTTP* 500 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.1.4.1, el diagrama presente en la Figura 4.5 expone el flujo del *endpoint*.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.1.4.1, en este *test* se eliminan los dos *Datasets* creados en el *test* que se encuentra en el Párrafo D.1.1.2.1 y se verifica el código de retorno que entregan las peticiones a la *API*.

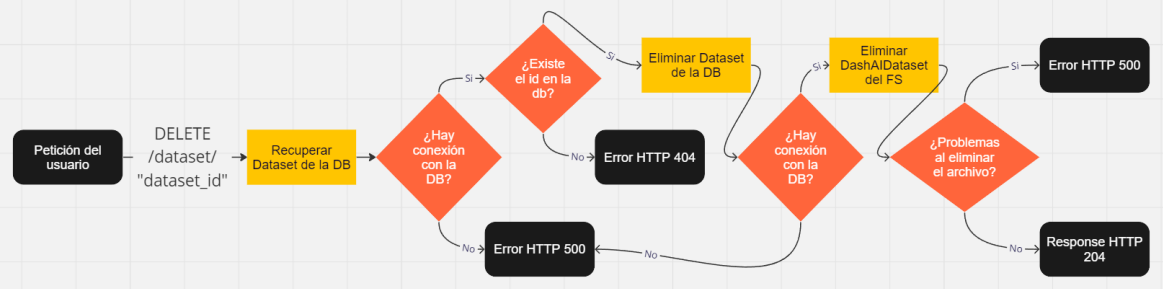

Figura 4.5: Diagrama de flujo *endpoint* DELETE `/dataset/"dataset_id"`.

4.2.2. Experiment

Esta *URI* se encuentra relacionada con la entidad *Experiment* del modelo de datos de *DashAI* . Con esta *URI* es posible crear, obtener, modificar y eliminar *Experiments* del *framework*.

4.2.2.1. GET

Los *endpoints* disponibles para la *URI* `/experiment` con método *HTTP* GET son:

4.2.2.1.1 `/experiment/`

Este *endpoint* permite obtener todos los *Experiments* presentes en la base de datos de *DashAI* .

No recibe argumentos del usuario y retorna una lista de diccionarios que representan cada una de las instancias de *Experiments* presentes en la base de datos.

Puede fallar debido a un error de conexión con la base de datos, en cuyo caso se le entrega un error *HTTP* 500 al usuario, indicando el problema.

El código del *endpoint* se encuentra en la Subsubsección B.2.1.1, el diagrama presente en la Figura 4.6 expone el flujo del *endpoint*.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.2.2.1, aquí se utiliza el *endpoint* para recuperar los dos *Experiments* creados por el *test* presente en el Párrafo D.1.2.3.1, verificando que ambos *Experiments* están asociados con el *Dataset* con *id dataset_id* usado para crear los experimentos.

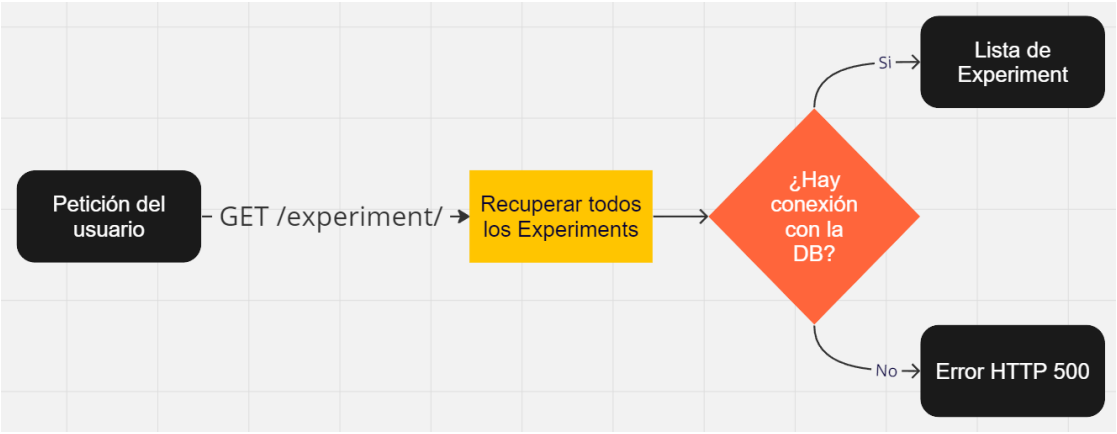

Figura 4.6: Diagrama de flujo *endpoint* GET `/experiment/`.

4.2.2.1.2 `/experiment/"experiment_id"`

Este *endpoint* permite obtener el *Experiment* que tiene el campo *id* con valor *experiment_id* de la base de datos.

Recibe como argumento el *id* del *Experiment* a recuperar y retorna un diccionario que representa al *Experiment* consultado por el usuario.

Puede fallar en caso de no encontrar el *Experiment* buscado, entregando un error *HTTP* 404 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.2.1.2, el diagrama presente en la Figura 4.7 expone el flujo del *endpoint*.

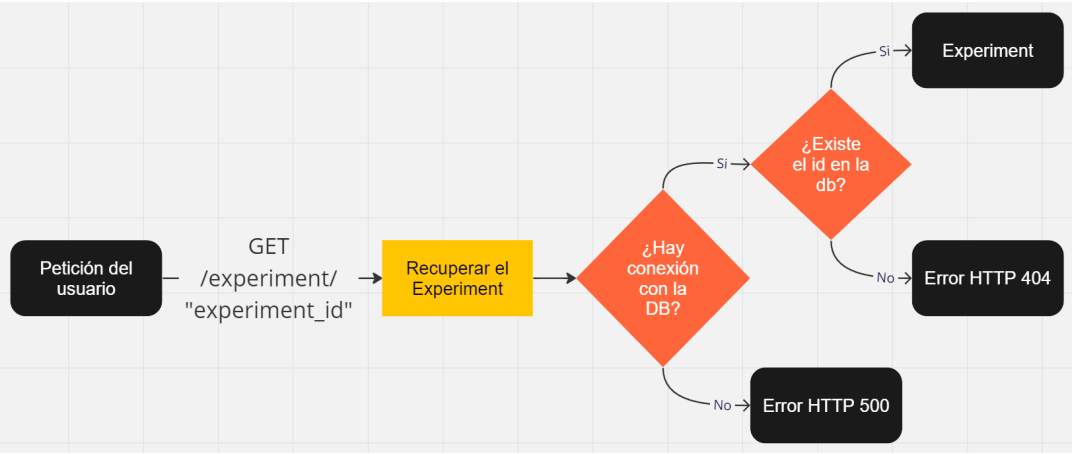

Figura 4.7: Diagrama de flujo *endpoint* GET `/experiment/"experiment_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.2.2.2, aquí se utiliza el *endpoint* para recuperar un *Experiment* inexistente, validando que se entregue un error *HTTP* 404.

4.2.2.2. POST

Para el método *HTTP* POST sólo se implementa un *endpoint* de la *URI* `/experiment`, el cual es:

4.2.2.2.1 `/experiment/`

Este *endpoint* permite que el usuario cree un experimento en *DashAI* .

Recibe como argumento el *id* del *Dataset* asociado, el nombre de la tarea relacionada al experimento y el nombre del experimento. Retorna un diccionario que representa al *Experiment* creado.

Puede fallar en caso de no encontrar el *Dataset* con *id dataset_id* en la base de datos, entregando un error *HTTP* 404 y en caso de no poder conectarse a la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.2.2.1, el diagrama presente en la Figura 4.8 expone el flujo del *endpoint*.

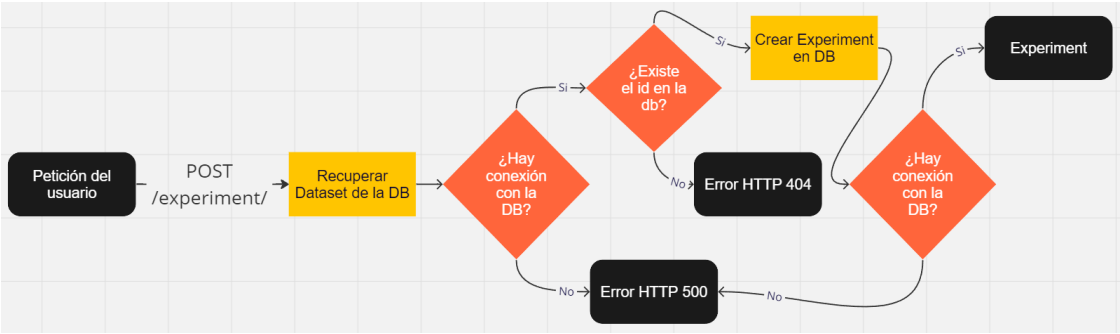

Figura 4.8: Diagrama de flujo *endpoint* POST `/experiment/`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.2.3.1, en él se crean dos instancias de *Experiment* con distinto nombre. Luego se consulta a la *API* por dichos *Experiments*, verificando que tengan los parámetros declarados al momento de crearlos.

4.2.2.3. PATCH

Para el método *HTTP* PATCH sólo se implementa un *endpoint* de la *URI* `/experiment`, el cual es:

4.2.2.3.1 `/experiment/"experiment_id"`

Este *endpoint* permite que el usuario modifique alguno de los parámetros presentes en la instancia de la entidad *Experiment* con *id experiment_id*.

Recibe como argumento el *id* del *Experiment* a modificar, el nuevo nombre, el nuevo *dataset_id* y la nueva tarea del *Experiment* y retorna un diccionario que representa al *Experiment* modificado. Tanto el nombre, como el *dataset_id*, como la tarea son argumentos opcionales.

Puede fallar en caso de que no se entregue ninguno de los parámetros opcionales, entregando un error *HTTP* 304 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.2.3.1, el diagrama presente en la Figura 4.9 expone el flujo del *endpoint*.

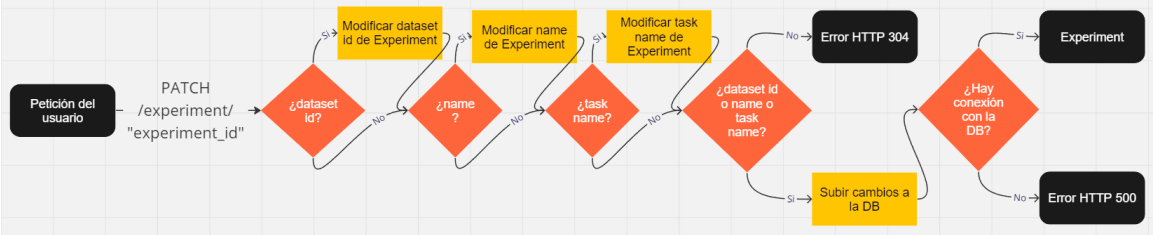

Figura 4.9: Diagrama de flujo *endpoint* PATCH `/experiment/"experiment_id"`.

Existen dos *tests* asociados a este *endpoint*, estos se pueden encontrar en el Párrafo D.1.2.4.1. El primero de ellos verifica que sea posible modificar los tres parámetros opcionales, mientras que, el segundo verifica que en caso de entregar un nuevo valor para otro campo este no se modifica.

4.2.2.4. DELETE

Para el método *HTTP* DELETE sólo se implementa un *endpoint* de la *URI* `/experiment`, el cual es:

4.2.2.4.1 `/experiment/"experiment_id"`

Este *endpoint* permite que el usuario elimine un *Experiment* de la aplicación.

Recibe como argumento el *id* del *Experiment* a eliminar y retorna un mensaje al usuario que indica que el experimento fue eliminado satisfactoriamente.

Puede fallar en caso de no encontrar el *Experiment* buscado, entregando un error *HTTP* 404 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.2.4.1, el diagrama presente en la Figura 4.10 expone el flujo del *endpoint*.

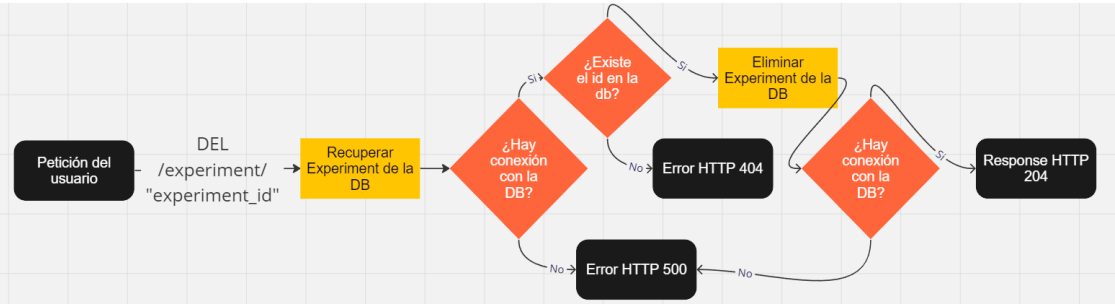

Figura 4.10: Diagrama de flujo *endpoint* DELETE `/experiment/"experiment_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.2.5.1, en este *test* se eliminan los dos *Experiments* creados en el *test* que se encuentra en el Párrafo D.1.2.3.1 y se verifica el código de retorno que entregan las peticiones a la *API*.

4.2.3. Run

Esta *URI* se encuentra relacionada con la entidad *Run* del modelo de datos de *DashAI* . Con esta *URI* es posible crear, obtener, modificar y eliminar *Runs* del *framework*.

4.2.3.1. GET

Los *endpoints* disponibles para la *URI* `/run` con método *HTTP* GET son:

4.2.3.1.1 `/run/`

Este *endpoint* permite obtener un subconjunto de las *Run* presentes en la base de datos de *DashAI* .

Recibe como argumento el *id* del *Experiment* asociado a las *Runs* que se busca recuperar y retorna una lista de diccionarios que representan a las instancias de las *Runs* asociadas al *Experiment* con *id* *experiment_id*. El parámetro *experiment_id* es opcional y en caso de no entregarlo se retornan todas las *Runs* presentes en la base de datos.

Puede fallar en caso de no encontrar el *Experiment* con *id* *experiment_id*, entregando un error *HTTP* 404 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.3.1.1, el diagrama presente en la Figura 4.11 expone el flujo del *endpoint*.

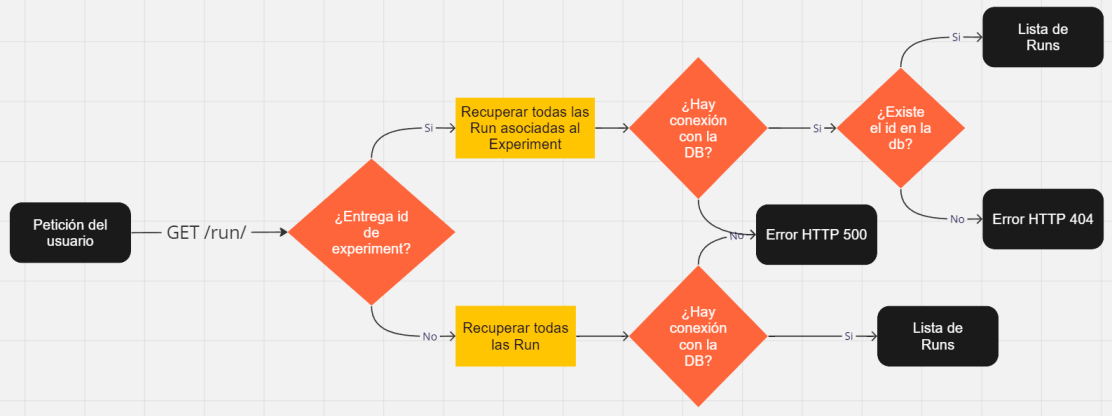

Figura 4.11: Diagrama de flujo *endpoint* GET `/run/`.

Los *tests* asociados a este *endpoint* se pueden encontrar en el Párrafo D.1.3.2.1. En el primero de ellos se utiliza el *endpoint* para recuperar las dos *Runs* creadas por el test presente en el Párrafo D.1.3.3.1, verificando que el primero presente el nombre *Run1* y el segundo el nombre *Run2*. El segundo test verifica que en caso de entregar un *id* de un *Experiment* inexistente se retorna el error correspondiente.

4.2.3.1.2 `/run/"run_id"`

Este *endpoint* permite obtener la *Run* que tiene el campo *id* con valor *run_id* de la base de datos.

Recibe como argumento el *id* de la *Run* a recuperar y retorna un diccionario que representa a la *Run* consultada por el usuario.

Puede fallar en caso de no encontrar la *Run* buscada, entregando un error *HTTP* 404 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.3.1.2, el diagrama presente en la Figura 4.12 expone el flujo del *endpoint*.

Los *tests* asociados a este *endpoint* se pueden encontrar en el Párrafo D.1.3.2.2. El primero de ellos verifica que al recuperar las *Runs* creadas en el test presente en el Párrafo D.1.3.3.1 estas tengan el nombre correcto, mientras que el segundo se utiliza para recuperar una *Run* inexistente, validando que se entregue un error *HTTP* 404.

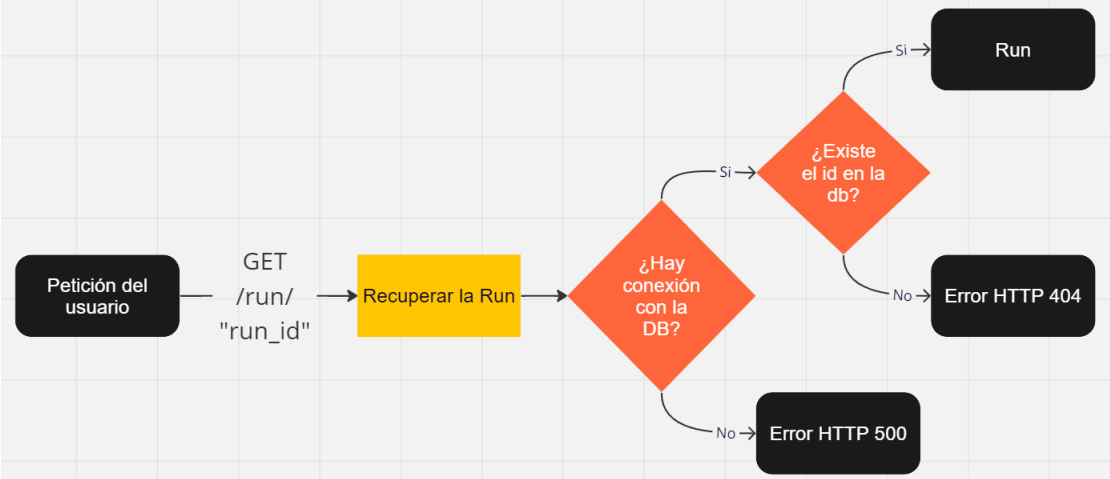

Figura 4.12: Diagrama de flujo *endpoint* GET `/run/"run_id"`.

4.2.3.2. POST

Para el método *HTTP* POST sólo se implementa un *endpoint* de la *URI* `/run`, el cual es:

4.2.3.2.1 `/run/`

Este *endpoint* permite que el usuario cree una Run en *DashAI* .

Recibe como argumento el *id* del *Experiment* asociado, el nombre del modelo de *Machine Learning* relacionado a la *Run*, los parámetros a entregar al modelo, el nombre y descripción de la Run. Retorna un diccionario que representa a la *Run* creada. La descripción es un argumento opcional, se rellena con un *string* vacío en caso de no entregarse.

Puede fallar en caso de no encontrar el *Experiment* con *id* *experiment_id* en la base de datos, entregando un error *HTTP* 404 y en caso de no poder conectarse a la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.3.2.1, el diagrama presente en la Figura 4.13 expone el flujo del *endpoint*.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.3.3.1, en él se crean dos instancias de *Run* con distinto nombre y parámetros. Luego se consulta a la *API* por dichas *Runs*, verificando que tengan los parámetros declarados al momento de crearlas.

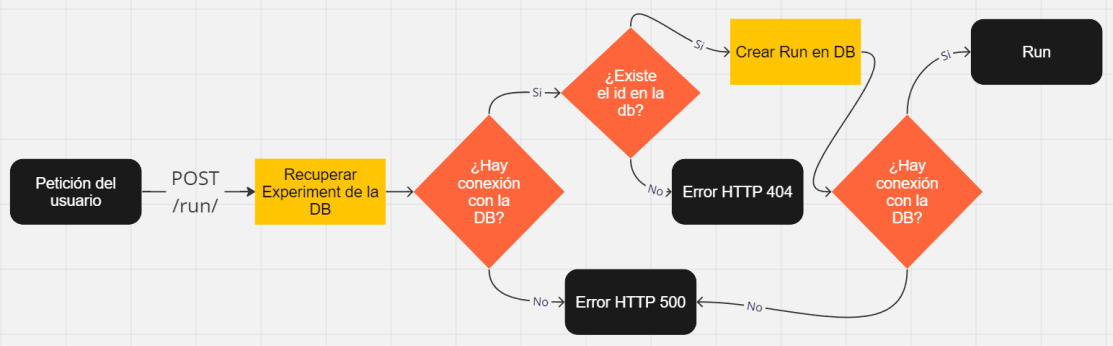

Figura 4.13: Diagrama de flujo *endpoint* POST `/run/`.

4.2.3.3. PATCH

Para el método *HTTP* PATCH sólo se implementa un *endpoint* de la *URI* `/run`, el cual es:

4.2.3.3.1 `/run/"run_id"`

Este *endpoint* permite que el usuario modifique alguno de los parámetros presentes en la instancia de la entidad *Run* con *id run_id*.

Recibe como argumento el *id* de la *Run* a modificar, el nuevo nombre, la nueva descripción y los nuevos parámetros de la *Run* y retorna un diccionario que representa a la *Run* modificada. Tanto el nombre, como la descripción, como los parámetros son argumentos opcionales.

Puede fallar en caso de que no se entregue ninguno de los parámetros opcionales, entregando un error *HTTP* 304 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.3.3.1, el diagrama presente en la Figura 4.14 expone el flujo del *endpoint*.

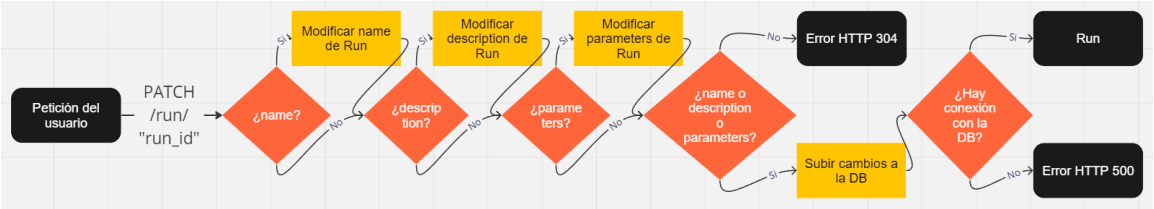

Figura 4.14: Diagrama de flujo *endpoint* PATCH `/run/"run_id"`.

Existen dos *tests* asociados a este *endpoint*, estos se pueden encontrar en el Párrafo D.1.3.4.1. El primero de ellos verifica que sea posible modificar los tres parámetros opcionales, mientras que, el segundo verifica que en caso de entregar un nuevo valor para otro campo este no se modifica.

4.2.3.4. DELETE

Para el método *HTTP* DELETE sólo se implementa un *endpoint* de la *URI* `/run`, el cual es:

4.2.3.4.1 `/run/"run_id"`

Este *endpoint* permite que el usuario elimine una *Run* de la aplicación, tanto su instancia en la base de datos como el archivo del modelo entrenado asociado a la *Run* (en caso de que exista).

Recibe como argumento el *id* de la *Run* a eliminar y retorna un mensaje al usuario que indica que la *Run* fue eliminada satisfactoriamente.

Puede fallar en caso de no encontrar la *Run* buscada, entregando un error *HTTP* 404, en caso de no encontrar el archivo donde se encuentra guardado el modelo asociado a la *Run*, entregando un error *HTTP* 500 y en caso de no poder establecer conexión con la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.3.4.1, el diagrama presente en la Figura 4.15 expone el flujo del *endpoint*.

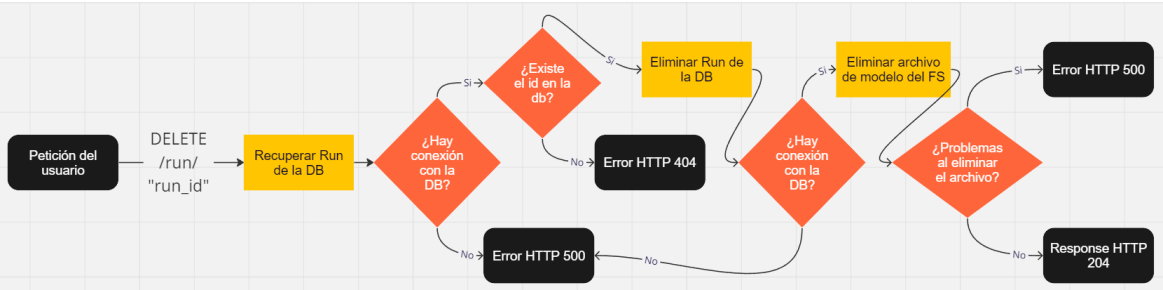

Figura 4.15: Diagrama de flujo *endpoint* DELETE `/run/"run_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.3.5.1, en este *test* se eliminan las dos *Runs* creadas en el *test* que se encuentra en el Párrafo D.1.3.3.1 y se verifica el código de retorno que entregan las peticiones a la *API*.

4.2.4. Job

Esta *URI* se encuentra relacionada con la cola de trabajos y la estructura *Job* mencionada en la Subsección 3.2.3. Con esta *URI* es posible encolar, eliminar y obtener trabajos de la cola de trabajos.

No es posible modificar un trabajo presente en la cola pues aquello sería en esencia crear un nuevo *job*, debido a que los parámetros modificables de un *job* son su tipo y sus *kwargs*, de los cuales el único modificable es el *run_id*.

4.2.4.1. GET

Los *endpoints* disponibles para la *URI* `/job` con método *HTTP* GET son:

4.2.4.1.1 `/job/`

Este *endpoint* permite obtener todos los *Jobs* presentes en la cola de trabajos.

No recibe argumentos y retorna una lista de diccionarios que representan a los trabajos presentes en la cola de trabajos.

Este método no puede arrojar errores al usuario.

El código del *endpoint* se encuentra en la Subsubsección B.4.1.1, el diagrama presente en la Figura 4.16 expone el flujo del *endpoint*.

Figura 4.16: Diagrama de flujo *endpoint* GET `/job/`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.4.2.1, aquí se utiliza el *endpoint* para recuperar los dos *Jobs* creados por el *test* presente en el Párrafo D.1.4.3.1, verificando que ambos *Jobs* están asociados con la *Run* con *id* *run_id* usada para crear los trabajos.

4.2.4.1.2 `/job/"job_id"`

Este *endpoint* permite obtener el *Job* que tiene el campo *id* con valor *job_id* de la cola de trabajos.

Recibe como argumento el *id* del *Job* a recuperar y retorna un diccionario que representa al *Job* consultado por el usuario.

Puede fallar en caso de no encontrar el *Job* buscado en la cola de trabajos, entregando un error *HTTP* 404.

El código del *endpoint* se encuentra en la Subsubsección B.4.1.2, el diagrama presente en la Figura 4.17 expone el flujo del *endpoint*.

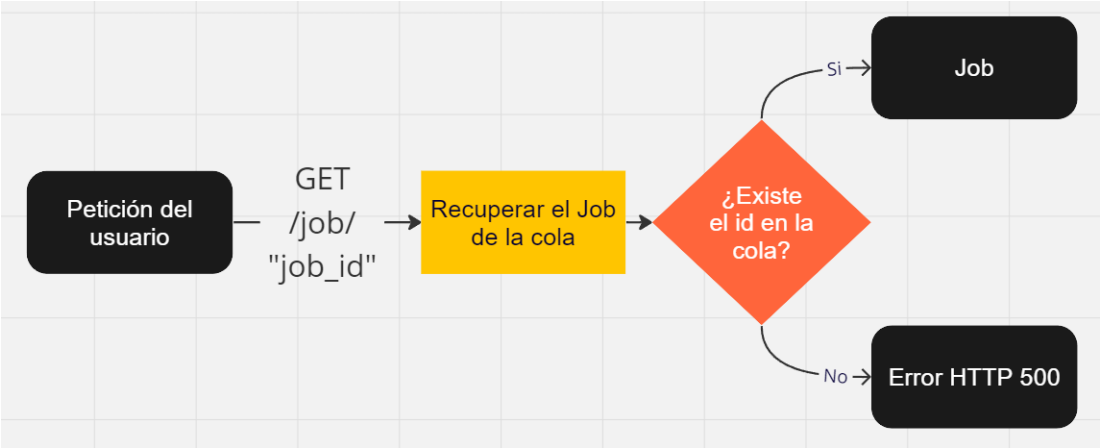

Figura 4.17: Diagrama de flujo *endpoint* GET `/job/"job_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.4.2.2, aquí se utiliza el *endpoint* para recuperar un *Job* inexistente, validando que se entregue un error *HTTP* 404.

4.2.4.1.3 `/job/start/`

Este *endpoint* inicia el procesamiento de los trabajos encolados en la cola de trabajos.

Recibe como argumento un booleano que indica si se debe detener el procesamiento de trabajos cuando la cola está vacía o si se debe esperar hasta que llegue un nuevo trabajo y retorna un mensaje al usuario que indica que se inició el procesamiento de trabajos de forma exitosa. El parámetro booleano es opcional y por defecto se indica que la cola espere por nuevos trabajos cuando esté vacía.

Este método no puede arrojar errores al usuario.

El código del *endpoint* se encuentra en la Subsubsección B.4.1.3, este consiste en iniciar el procesamiento de la cola de trabajos mediante la invocación asíncrona de la función `job_queue_loop`.

El código de dicha función se encuentra en la Subsección B.4.4, este consiste en tomar trabajos de la cola de trabajos y ejecutarlos.

El único trabajo actualmente implementado en *DashAI* es el de entrenar y evaluar modelos de *Machine Learning*, que tiene asociada la función `execute_run` cuyo código se encuentra en la Subsección B.4.5, el diagrama presente en la Figura 4.18 expone el flujo de dicha función.

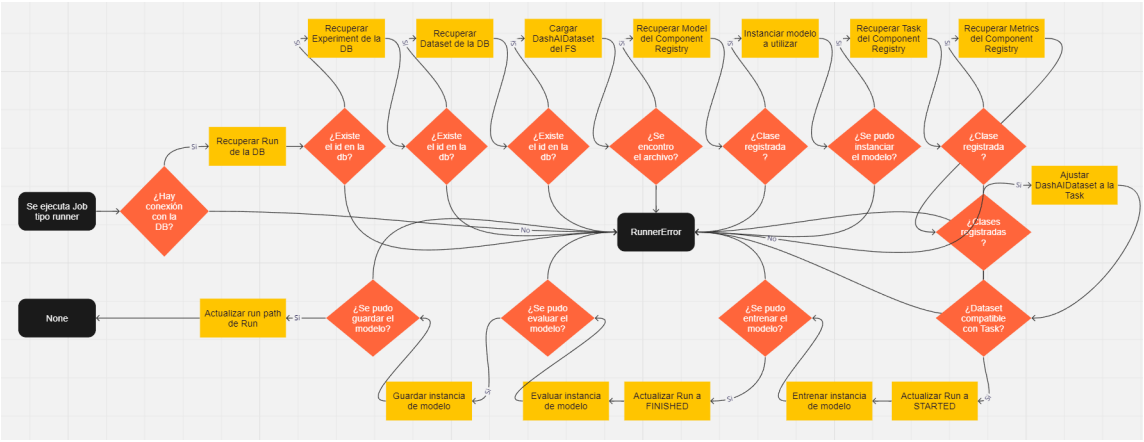

Figura 4.18: Diagrama de flujo de la función `execute_run`.

El *test* al *endpoint* antes mencionado se puede encontrar en el Párrafo D.1.4.2.3, en él se crean dos *Jobs* y se inicia el procesamiento de trabajos. Luego se verifica que el primer trabajo haya terminado de forma exitosa y que el segundo trabajo termina en error debido a que el modelo de *Machine Learning* asociado presenta errores.

4.2.4.2. POST

Para el método *HTTP* POST sólo se implementa un *endpoint* de la *URI* `/job`, el cual es:

4.2.4.2.1 `/job/runner/`

Este *endpoint* permite que el usuario encola en la cola de trabajos un *Job* de entrenamiento de una *Run*.

Recibe como argumento el *id* de la *Run* asociada al *Job* y retorna un diccionario que representa al *Job* encolado en la cola de trabajos.

Puede fallar en caso de no encontrar la *Run* con *id run_id* en la base de datos, entregando un error *HTTP* 404 y en caso de no poder conectarse a la base de datos, entregando un error *HTTP* 500.

El código del *endpoint* se encuentra en la Subsubsección B.4.2.1, el diagrama presente en la Figura 4.19 expone el flujo del *endpoint*.

Los *tests* asociados a este *endpoint* se pueden encontrar en el Párrafo D.1.4.3.1. En el primero de ellos se encolan dos *Jobs* iguales en la cola de trabajos. Luego se consulta a la *API* por dichos *Jobs*, verificando que tengan los parámetros declarados al momento de crearlos, mientras que en el segundo se solicita crear un *Job* asociado a una *Run* inexistente, verificando el código de error retornado por la *API*.

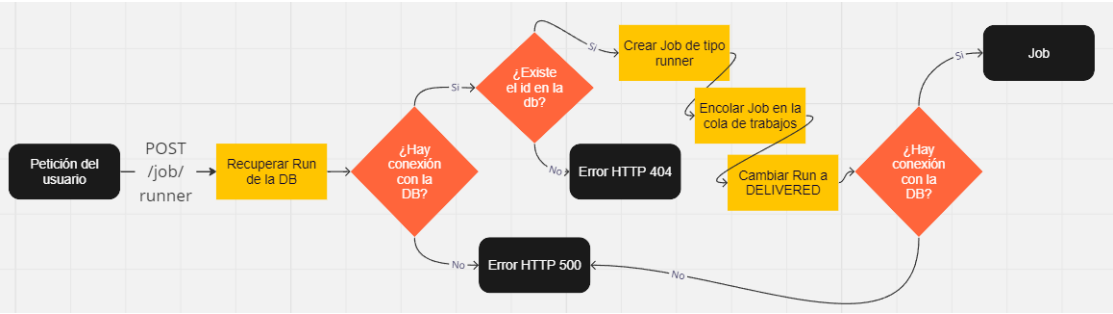

Figura 4.19: Diagrama de flujo *endpoint* POST `/job/`.

4.2.4.3. PATCH

No existe ningún endpoint para el método *HTTP* PATCH relativo a la *URI* `job`, esto debido a que modificar un *Job* representa lo mismo que eliminar un *Job* existente y crear uno nuevo con los nuevos parámetros.

4.2.4.4. DELETE

Para el método *HTTP* DELETE sólo se implementa un *endpoint* de la *URI* `/job`, el cual es:

4.2.4.4.1 `/job/"job_id"`

Este *endpoint* permite que el usuario cancele un *Job* presente en la cola de trabajos, impidiendo su ejecución.

Recibe como argumento el *id* del *Job* a cancelar y retorna un mensaje al usuario que indica que el *Job* fue eliminado satisfactoriamente.

Puede fallar en caso de no encontrar el *Job* buscada en la cola de trabajos, entregando un error *HTTP* 404.

El código del *endpoint* se encuentra en la Subsubsección B.4.3.1, el diagrama presente en la Figura 4.20 expone el flujo del *endpoint*.

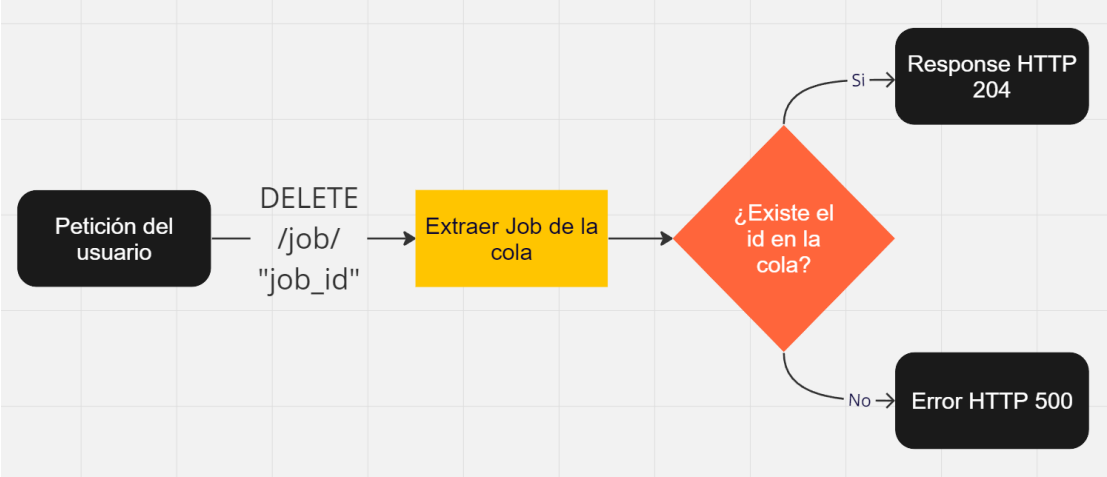

Figura 4.20: Diagrama de flujo *endpoint* DELETE `/job/"job_id"`.

El *test* asociado a este *endpoint* se puede encontrar en el Párrafo D.1.4.4.1, en este *test* se cancelan los dos *Jobs* encolados en el *test* que se encuentra en el Párrafo D.1.4.3.1 y se verifica el código de retorno que entregan las peticiones a la *API*, junto con los *Jobs* restantes en la cola luego de cada cancelación.

4.3. Cola de trabajos

En esta sección se muestra cómo se materializa la cola de trabajos que se presenta en la Subsección 3.2.3, para ello la sección se divide en dos subsecciones que muestran cada uno de los elementos necesario para generar la implementación de la cola de trabajos, en la primera se expone la estructura de los trabajos, mientras que, en la segunda se expone la implementación de la interfaz de la cola de trabajos y la instancia concreta de dicha interfaz.

4.3.1. Estructura Job

La estructura de los trabajos que se manejan en la cola de trabajos se encuentra en la Subsección C.1.1. Esta consiste en una clase que hereda de la clase *BaseModel* que provee la librería *Pydantic*. Dicha clase presenta los siguientes campos:

- `id`: identificador del *Job*.
- `func`: función a ejecutar, esta recibe como argumento un entero que representa el *id* de la *Run* que se desea procesar y un objeto de tipo *Session* que permite realizar conexiones con la base de datos.
- `job_type`: tipo de job que se quiere ejecutar, estos tipos están dados por la enumeración JobType, cuyo código se encuentra en Subsección C.1.2. Actualmente solo se encuentra implementado el tipo *runner*, que representa la petición de entrenamiento y evaluación de un modelo de *Machine Learning*.

- `kwargs`: diccionario que contiene los argumentos que se deben entregar a la función `func` al momento de ejecutarla. En el caso de un *Job* de tipo *runner* este diccionario contiene el *id* de la *Run* a procesar y el objeto de tipo *Session* que permite conectarse con la base de datos.

Esta estructura también presenta el método `ser_job` que permite serializar el *Job* con el fin de poder ser entregado a los usuarios de la *API*, para ello este método entrega un diccionario que omite el campo *func* y el objeto *Session* debido a que ambos no son serializables.

4.3.2. Implementación JobQueue

Para implementar el *ADT* de la cola de trabajos que se muestra en la Figura 3.17 se desarrolla la clase abstracta *BaseJobQueue* que se encuentra en la Subsección C.2.1 de los anexos. En ella se define cada uno de los métodos declarados en el *ADT* indicando su firma y propósito.

Para dicha interfaz se implementa la clase concreta *SimpleJobQueue* que se encuentra en la Subsección C.2.2. Dicha clase hace uso de la clase *Queue* provista por la librería integrada *asyncio* de *Python*. La clase *Queue* presenta los siguientes métodos usados en la implementación de la clase *SimpleJobQueue*:

- `put_no_wait`: permite encolar un objeto en la cola.
- `get`: extrae un objeto de la cola, en caso de que la cola esté vacía espera hasta que llegue un nuevo elemento a la cola.
- `get_no_wait`: extrae un objeto de la cola, en caso de que la cola esté vacía arroja un error.
- `empty`: predicado que indica si la cola está vacía.

A continuación se detalla cómo se implementa cada uno de los métodos declarados en el *ADT* de la cola de trabajos.

4.3.2.1. `put(Job)`

Método que permite encolar un *Job* en la cola de trabajos. Recibe como argumento el *Job* a encolar y retorna un entero que identifica al *Job* encolado.

La invocación de este método no puede arrojar errores.

Este método recibe el *Job* a encolar, calcula un identificador para el trabajo usando la función `uuid4` de la librería integrada *uuid*de *Python*, agrega el identificador al *Job*, encola el *Job* usando la función `put_no_wait` de la clase *Queue* y retorna el identificador generado.

El *test* asociado a este método se puede encontrar en la Subsubsección D.2.1.1. Dicho *test* verifica que los identificadores entregados por el método sean siempre distintos.

4.3.2.2. `get(int)`

Método que permite extraer un *Job* de la cola de trabajos. Recibe como argumento el identificador del *Job* a extraer y retorna el *Job* solicitado. El identificador es un parámetro opcional, en caso de no entregarlo se extrae el primer elemento de la cola.

Puede fallar en caso de que la cola se encuentre vacía o en caso de que no se encuentre el *Job* buscado, en ambos casos se entrega un error de tipo *JobQueueError*.

El diagrama de la figura Figura 4.21 muestra el funcionamiento de este método.

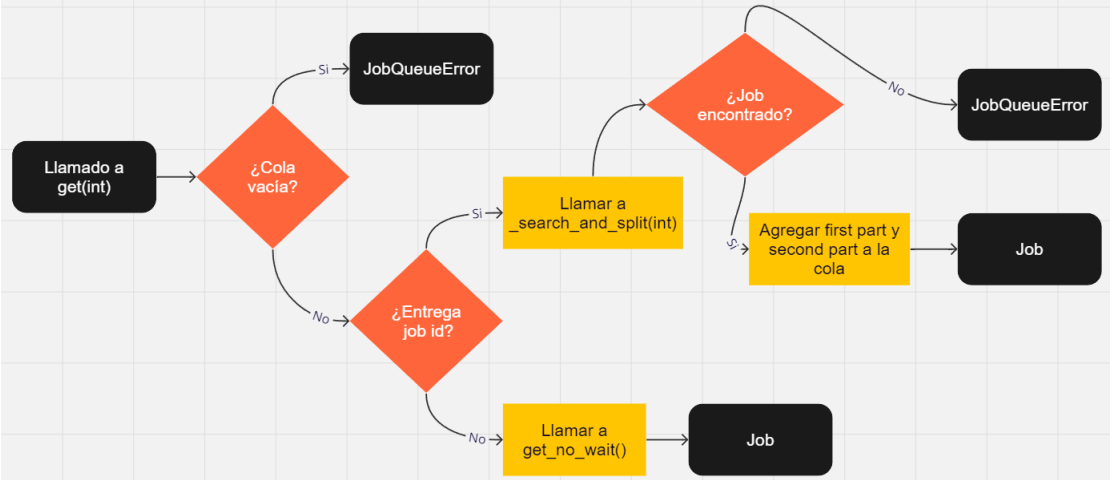

Figura 4.21: Diagrama de flujo método `get(int)` de la clase *SimpleJobQueue*.

Los *tests* asociados a este método se pueden encontrar en la Subsubsección D.2.1.2. El primero de ellos evalúa que al llamar a este método se retornen los trabajos solicitados y que estos sean extraídos de la cola de trabajos. El segundo verifica el error resultante de invocar el método al encontrarse la cola vacía. EL tercero verifica el error resultante de solicitar un trabajo inexistente en la cola de trabajos.

4.3.2.3. `async_get()`

Método que extrae el primer elemento de la cola de trabajos y espera asíncronamente en caso de que la cola esté vacía. No recibe argumentos y entrega el primer *Job* de la cola.

La invocación de este método no puede arrojar errores.

Este método llama directamente al método `get` de la clase *Queue*, retornando lo entregado por dicho método.

El *test* asociado a este método se encuentra en la Subsubsección D.2.1.3. Dicho *test* verifica que el trabajo entregado por el método sea el solicitado. No se puede evaluar si el método actúa de forma asíncrona pues *Pytest* siempre espera a que termine la ejecución de un método asíncrono para continuar la ejecución del test.

4.3.2.4. `peek(int)`

Método que permite recuperar un *Job* presente en la cola de trabajos sin extraerlo. Recibe como argumento el identificador del trabajo a recuperar y retorna el trabajo recuperado. En caso de no entregar un identificador se recupera el primer trabajo de la cola sin extraerlo.

Puede fallar en caso de que la cola se encuentre vacía o en caso de que no se encuentre el *Job* buscado, en ambos casos se entrega un error de tipo *JobQueueError*.

El diagrama de la figura Figura 4.22 muestra el funcionamiento de este método.

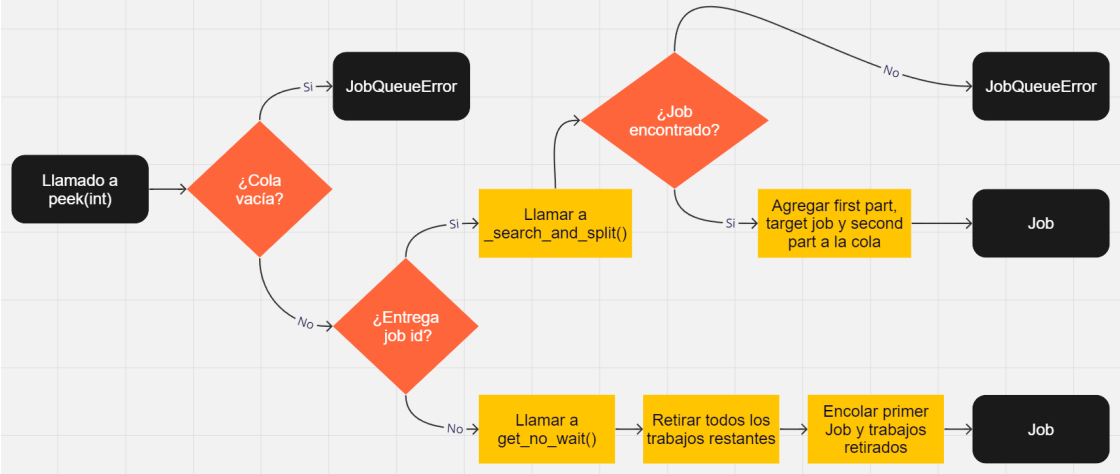

Figura 4.22: Diagrama de flujo método `peek(int)` de la clase *SimpleJobQueue*.

Los *tests* asociados a este método se pueden encontrar en la Subsubsección D.2.1.4. El primero de ellos evalúa que al llamar a este método se retornen los trabajos solicitados y que estos no sean extraídos de la cola de trabajos. El segundo verifica el error resultante de invocar el método al encontrarse la cola vacía. EL tercero verifica el error resultante de solicitar un trabajo inexistente en la cola de trabajos.

4.3.2.5. `is_empty()`

Método que indica si la cola se encuentra o no vacía. No recibe argumentos y retorna un booleano que en caso de ser verdadero indica que la cola está vacía.

La invocación de este método no puede arrojar errores.

Este método llama directamente al método `empty` de la clase *Queue*, retornando el valor entregado por dicho método.

El *test* asociado a este método se puede encontrar en la Subsubsección D.2.1.5, en él se verifica que el método entregue *True* cuando no posee elementos la cola y *False* en el caso contrario.

4.3.2.6. `to_list()`

Método que permite recuperar todos los trabajos presentes en la cola de trabajos. No recibe argumentos y entrega una lista con los *Jobs* recuperados de la cola de trabajos.

La invocación de este método no puede arrojar errores.

Este método extrae todo los trabajos de la cola usando el método `get_no_wait` de la clase *Queue*, los almacena en una lista, luego vuelve a encolar los trabajos en el mismo orden que tenían inicialmente y retorna la lista generada.

El *test* asociado a este método se puede encontrar en la Subsubsección D.2.1.6, en él se verifica que la lista entregada por el método retorne los trabajos en el mismo orden que tenían dentro de la cola y que el método funcione cuando la cola de trabajos está vacía.

4.3.3. `_search_and_split(int)`

Método privado de la clase *SimpleJobQueue* (en *Python* no es posible definir métodos privados, sin embargo, este método no está definido en la clase abstracta *BaseJobQueue* por lo que no se espera que sea usado fuera de la clase) que permite dividir la cola de trabajos en dos partes, usando un trabajo presente en ella como *pivote*. Se usa al momento de buscar un cierto trabajo en la cola debido a que la clase *Queue* no define método que permitan realizar esto, si no que la única forma de buscar un elemento en la cola es extrayéndolos hasta encontrar el buscado.

Recibe como argumento el identificador del trabajo buscado y retorna un tupla con tres valores, el primero es la primera parte de la cola, el segundo es el trabajo buscado y el tercero es la segunda parte de la lista.

Puede fallar en caso de que la cola se encuentre vacía o en caso de que no se encuentre el Job buscado, en ambos casos se entrega un error de tipo *JobQueueError*.

El diagrama de la figura Figura 4.23 muestra el funcionamiento de este método.

Como este método es un método privado de la clase *SimpleJobQueue* no se considera elaborar *tests* que verifiquen la funcionalidad prometida, si no que dicha funcionalidad se evalúa al evaluar los métodos que lo utilizan, como `get(int)` y `peek(int)`.

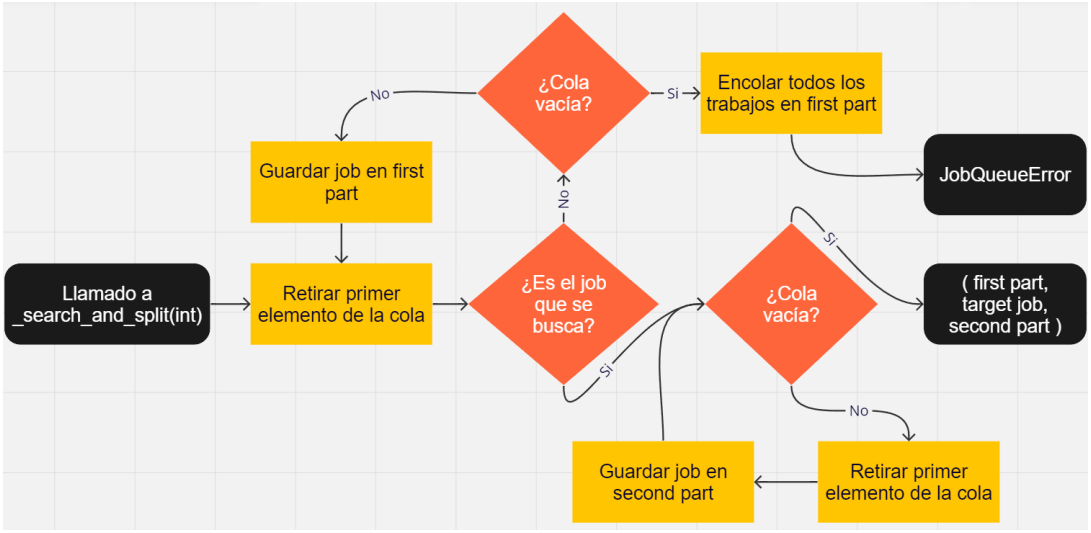

Figura 4.23: Diagrama de flujo método `_search_and_split(int)` de la clase *SimpleJobQueue*.

4.4. Resumen

En este capítulo se mostró el código implementado por el memorista relativo al diseño que se presenta en el Capítulo 3. Primero se presentó el código relativo al modelo de datos, luego se mostró el código de la *API*, explicando cada uno de los *endpoints* desarrollados y finalmente se mostró la implementación de la cola de trabajos.

Capítulo 5

Puesta en marcha

En este capítulo se explica con más detalle los procesos de evaluación y corrección que ocurrieron para incluir los cambios hechos por el memorista en *DashAI* .

Como se menciona en la Sección 1.5, *DashAI* es un proyecto que a tiempo de escribir esta memoria se encuentra siendo desarrollado por un grupo de ingenieros civiles en computación y estudiantes de la misma carrera, donde cada uno tiene a cargo distintos módulos del *software*.

Lo anterior lleva a que las soluciones hechas por cada integrante no sean del todo compatibles con el resto del *framework*.

Para evitar que estos problemas se produzcan en la versión estable de la librería es que existen cuatro etapas de evaluación de código e integración, las cuales se describen a continuación.

La primera etapa consiste en las reuniones diarias (o *dailys* en inglés), a estas reuniones concurren todos los desarrolladores de *DashAI* para comentar los cambios realizados el día anterior y lo que se espera desarrollar durante el día. Es en estas reuniones donde los distintos integrantes de *DashAI* discuten cómo realizar las implementaciones necesarias para hacer crecer al *software* sin generar incompatibilidades con módulos previos o en desarrollo.

Luego de esta etapa el desarrollador comienza a realizar las implementaciones correspondientes al módulo que tiene a su cargo, para ello en *DashAI* se sigue la práctica de ingeniería de software llamada *Test Driven Development*, la cuál declara que para desarrollar una nueva funcionalidad se deben seguir los siguiente pasos:

- Escribir un *test* sobre la funcionalidad que se espera implementar, este *test* debe fallar pues la funcionalidad aún no está implementada.
- Implementar la funcionalidad buscando que el *test* escrito sea aprobado.
- Refactorizar el código escrito, para evitar duplicación de código y seguir buenas prácticas de escritura de código.

Siguiendo esta práctica es que se llega a la segunda etapa de evaluación de código, la cual consiste en la ejecución de los *test* presentes en el proyecto, tanto los agregados por el desarrollador como los agregados previamente, aquí el desarrollador debe asegurarse de que todos los *test* pasen antes de subir sus cambios al repositorio remoto del proyecto.

DashAI también cuenta con herramientas de formateo de código, también conocidas como *linters*, las que verifican que el código escrito siga formatos de escritura de código como los declarados por la guía de estilo *PEP 8* de *Python*. Estas si bien no apuntan a evaluar la funcionalidad del código, buscan estandarizar el estilo utilizado para escribirlo, una característica importante para proyectos donde participan grupos grandes de personas donde el código escrito puede variar mucho de persona en persona.

Una vez se aprueban los *tests* y el *linting* los cambios son subidos al repositorio remoto del proyecto. Este se encuentra hospedado en *GitHub*, el enlace al repositorio es `https://github.com/DashAISoftware/DashAI`.

El repositorio se encuentra organizado en ramas (o *branches* en inglés), las cuales consisten en distintas versiones del código presente en el repositorio. La rama *main* es donde se encuentra la versión estable del proyecto, la rama *staging* es donde se mezclan los cambios hechos por los desarrolladores, con el fin de integrarlos y ser enviados a la rama *main*. Además de estas existen una serie de ramas donde se encuentran los cambios específicos realizados por cada integrante de *DashAI* .

Una vez un desarrollador termina una implementación este debe crear una *PR* (sigla en inglés de *Pull Request*), que consiste en una petición de incluir los cambios en una rama del proyecto, en el caso de *DashAI* los cambios se llevan a la rama *staging*, también se incluye una breve descripción de los cambios realizados. Dicha *PR* es revisada por alguno de los ingenieros en computación de *DashAI* , quien puede aprobar los cambios o solicitar correcciones.

El repositorio de *DashAI* cuenta también con *GitHub Actions*, las cuales son piezas de código que se ejecutan cada vez que se crea una *PR* o se suben cambios a una rama particular. En *DashAI* se ocupan para ejecutar el *linter* y los test cada vez que se crea un *PR*, con el fin de asegurar que los cambios presentes en la *PR* cumplen con ambas evaluaciones.

Luego de ser aprobada la *PR*, los cambios desarrollados son llevados a la rama *staging* donde se prueba la integración de estos junto con el resto de módulos del *software*. Estas pruebas se realizan de forma manual ejecutando la aplicación y realizando el caso de uso clásico de la aplicación.

Como los cambios desarrollados por el memorista estaban relacionados principalmente la *API*, que es un elemento que interactúa directamente con la interfaz gráfica, se tuvo que tener una serie de reuniones y discusiones con el integrante de *DashAI* a cargo del *frontend* para lograr una buena integración con este componente.

Al momento de escribir esta memoria la mayoría de los cambios relativos a la *API* y el modelo de datos se encuentran incluidos en la rama staging e integrados con el resto del *software*. Esto se puede apreciar en las capturas expuestas en la Subsección 3.1.3, que exponen la aplicación relativa al código presente en la rama *staging*.

Sin embargo, los cambios relativos a la cola de tareas y el *endpoint* de los *jobs* no se encuentran incluidos en la rama *staging* si no que se encuentran en la rama *back/job-queue*. Esto se debe principalmente a que en el tiempo próximo a la entrega de esta memoria *DashAI* llevará a cabo una serie de reuniones con instituciones tecnológicas con el objetivo de levantar fondos, esto lleva a que se tenga que asegurar la integridad del *software* que se presentará.

Capítulo 6

Conclusión

El objetivo de este capítulo es presentar al lector reflexiones frente al trabajo realizado junto con trazar un camino para futuras mejoras al trabajo hecho por el memorista.

Para ello el capítulo se encuentra divido en tres secciones, la primera busca mostrar lo logrado y no logrado por la implementación desarrollada por el memorista en relación a los objetivos planteado en la Sección 1.4 y finalmente una reflexión sobre qué podría hacer el memorista en caso de que pudiera iniciar de cero este trabajo. La segunda sección presenta un análisis de la utilidad de las tecnologías utilizadas para la implementación desarrollada. Y finalmente la tercera sección busca enumerar aspectos que se pueden mejorar de la implementación generada y nuevas funcionalidades se podrían agregar para potenciar al software *DashAI* .

6.1. Restrospectiva

El trabajo realizado por el memorista logra cumplir de manera satisfactoria con los objetivos planteados en la Sección 1.4, a continuación se listan estos para explicar la razón de su cumplimiento.

Como resultado de este trabajo se obtiene una *API* funcional que permite reproducir el flujo de trabajo de *Machine Learning* que se presenta en la Sección 2.2. La *API* gracias a su estructura *RESTful* permite al *frontend* generar las visualizaciones que se muestran en Subsección 3.1.3 sin depender de cómo funciona el *backend*, con lo que se cumple el objetivo específico 1.

La *API* también permite al usuario mantener puntos de guardado al usar la aplicación, pudiendo este tanto retomar experimentos previos como revisar resultados de experimentos ya procesados. Con lo anterior se puede dar por cumplido el objetivo específico 2.

Además, la *API* desarrollada por el memorista presenta un comportamiento asíncrono al momento de entrenar modelos de *Machine Learning*, lo que permite que el usuario pueda seguir recibiendo respuestas del servidor, con lo que se cumple el objetivo específico 3. Es importante destacar que aún cuando se cumple con el objetivo la solución generada no es del todo satisfactoria, pues la cola de trabajos no se encuentra respaldada en la base de datos, lo que produce que se pierdan los *jobs* encolados en caso de que se cierre la herramienta. Se espera que se pueda atacar dicha falencia en una futura iteración del *software*.

Por último se tiene que la *API* implementada es lo suficientemente flexible, en el sentido que se menciona en Sección 1.6, para poder soportar nuevos modelos, tareas, métricas, etc.. Esto se puede ver en que dentro del código de la *API* sólo se espera que los componentes utilizados cumplan con las interfaces expuestas en la Figura 3.2.

Con todo lo anterior se puede decir que el trabajo realizado por el memorista cumple satisfactoriamente con el objetivo general de este trabajo de título, el cual es Diseñar e Implementar el *backend* del *framework DashAI* , permitiendo que este sea extensible a las tareas que el usuario necesite abarcar.

Cabe destacar también que los test desarrollados contribuyeron enormemente en la detección temprana de errores y también que serán de gran ayuda para detectar futuros errores gracias a que se encuentran incluidos dentro del sistema de integración continua con el que cuenta el repositorio de *DashAI* .

Como reflexión personal el memorista considera como un éxito tanto el desarrollo como los resultados del trabajo de título, debido a que este le permitió poner en práctica muchos de los conocimientos obtenidos en los cursos de la carrera de Ingeniería Civil en Computación, lo que no pudo lograr de manera satisfactoria al desarrollar sus prácticas profesionales debido a los trabajos que se le asignaban y a la corta duración de las mismas.

Para finalizar esta sección, si el memorista tuviera la oportunidad de volver a empezar este trabajo, buscaría familiarizarse más con las herramientas presentadas en la Sección 2.1, debido a que muchas de ellas presentaban buenas soluciones a problemas que el memorista trató de resolver por su cuenta y que por lo mismo le resultaron más complejas de lo que podrían haber sido.

6.2. Diagnóstico de las tecnologías utilizadas

En esta sección se muestra qué tan adecuadas resultaron las tecnologías escogidas en la elaboración de este trabajo.

El uso del lenguaje *Python* nunca fue puesto a discusión debido a que es el lenguaje más usado en el desarrollo de *Machine Learning*, aun así, este lenguaje resulta adecuado debido a la gran cantidad de librerías desarrolladas en este lenguaje, lo que permite al memorista poder escoger la que se adecue mejor al problema que se busca resolver.

La librería de *Python FastAPI* facilita enormemente el desarrollo de la *API* y de la cola de trabajos, debido a lo fácil que resulta escribir los *endpoints* y que esta cuenta con la herramienta *BackgroundTasks* para el procesamiento asíncrono.

El gestor de bases de datos *SQLite* permite contar con una base de datos sin aumentar el costo de instalación de la librería además de presentar un buen rendimiento para los bajos volúmenes de datos que se manejan actualmente en *DashAI* , por lo que se considera como acertada la desición de utilizarla en el desarrollo.

La librería de *Python SQLAlchemy*[5] cuenta con interfaces intuitivas para el manejo de bases de datos, junto con presentar la posibilidad de migrar de gestor de manera transparente para el *software*, lo que se espera sea una necesidad en el futuro de *DashAI* . Es por ello que se considera una buena elección haber realizado este trabajo con la librería antes mencionada.

Finalmente la librería de *Python Pytest* facilita la creación de test unitarios y la reutilización del código necesario para implementarlos (mediante el uso de *fixtures*), lo que lleva a considerar como buena idea el uso de esta librería. Sin embargo esta librería no permite evaluar el comportamiento asíncrono de una *API*, por lo que queda pendiente la exploración de otras librerías que sí entregan dicha funcionalidad.

En conclusión la mayoría de las tecnologías utilizadas logro cumplir satisfactoria con los requerimientos pedidos.

6.3. Trabajo Futuro

Si bien este trabajo logra cumplir con los objetivos propuestos, esto no quiere decir que no se encuentre abierto a mejoras que permitan perfeccionar la solución desarrollada y hacer crecer a *DashAI* .

Algunas de estas mejoras son las siguientes:

- Validar parámetros entregados por el usuario a través de los *endpoints* de la *API*: El trabajo realizado no contempla la realización de esta validación por temas de tiempo, pero se espera que en el futuro *DashAI* cuente con templates de *Pydantic* o herramientas similares que permitan validar la información entregada por el usuario.
- Persistencia de los *jobs*: Actualmente en *DashAI* los *jobs* generados solo son almacenados en la cola de procesos, la cual no persiste luego del cierre del *framework*, esto se puede arreglar guardando metadatos en la entidad *Run* o incluso creando una nueva entidad para almacenar *jobs*, sin embargo se detectó que aquello se encontraba fuera de los alcances de este trabajo.

Vale la pena destacar que además de lograr solucionar los problema mencionados en la Sección 1.2, este trabajo abre el camino para la realización de nuevos proyectos relacionados a mejorar *DashAI* , a continuación se mencionan los más relevantes desde el punto de vista del memorista, algunos de los cuales a tiempo de escritura de esta memoria ya se encuentran en etapa de desarrollo por miembros del equipo de *DashAI* .

- Incluir un módulo que permita la optimización de hiper parámetros.
- Permitir al usuario modificar los conjuntos de datos subidos por esto, con el fin de aplicar procedimientos de limpieza, estandarización, etc..
- Generar una nueva implementación de cola de trabajos, la cuál se encuentre conectada con un servicio como *Celery* que permita paralelizar el entrenamiento de modelos.

Bibliografía

[1] Bert 101 - state of the art nlp model explained. `https://huggingface.co/blog/bert-101`. Accessed: 2023-07-17.

[2] Hugging Face AutoTrain. `https://huggingface.co/autotrain`. Accessed: 2022-12-05.

[3] IBM Watson Studio. `https://www.ibm.com/cloud/watson-studio`. Accessed: 2022-12-05.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[5] Michael Bayer et al. Sqlalchemy-the database toolkit for python. *URL http://www. sqlalchemy. org/. Accessed on the 13th of November*, 2012.

[6] Michael R Berthold, Nicolas Cebron, Fabian Dill, Thomas R Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. Knime-the konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*, 11(1):26–31, 2009.

[7] Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, et al. Orange: data mining toolbox in python. *the Journal of machine Learning research*, 14(1):2349–2353, 2013.

[8] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: `https://doi.org/10.24432/C56C76`.

[9] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[10] Chip Huyen. *Designing machine learning systems*. O'Reilly Media, Inc., 2022.

[11] Nikhil Ketkar. Introduction to keras. In *Deep learning with Python*, pages 97–111. Springer, 2017.

[12] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940, 2006.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[15] Anand S Rao and Gerard Verweij. Sizing the prize: What's the real value of ai for your business and how can you capitalise. *PwC Publication, PwC*, pages 1–30, 2017.

[16] Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33(2008):18, 2008.

[17] Russell Sears, Catharine Van Ingen, and Jim Gray. To blob or not to blob: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168*, 2007.

[18] Ryan Singer. *Shape Up: Stop Running in Circles and Ship Work that Matters*. Basecamp, 2019.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[21] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

Anexo A

Código modelo de datos

A.1. Dataset

```python
class Dataset(Base):
    __tablename__ = "dataset"
    """
    Table to store all the information about a dataset.
    """
    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(
        String,
        unique=True,
        nullable=False,
    )
    task_name: Mapped[str] = mapped_column(String, nullable=False)
    created: Mapped[DateTime] = mapped_column(
        DateTime, default=datetime.now
    )
    last_modified: Mapped[DateTime] = mapped_column(
        DateTime,
        default=datetime.now,
        onupdate=datetime.now,
    )
    file_path: Mapped[str] = mapped_column(String, nullable=False)
    experiments: Mapped[List["Experiment"]] = relationship()
```

Code Listing A.1: Clase de la tabla dataset

A.2. Experiment

```python
class Experiment(Base):
    __tablename__ = "experiment"
    """
    Table to store all the information about a model.
    """
```

```
    id: Mapped[int] = mapped_column(primary_key=True)
    dataset_id: Mapped[int] = mapped_column(
        ForeignKey("dataset.id")
    )
    name: Mapped[str] = mapped_column(
        String, unique=True, nullable=False
    )
    task_name: Mapped[str] = mapped_column(String, nullable=False)
    created: Mapped[DateTime] = mapped_column(
        DateTime, default=datetime.now
    )
    last_modified: Mapped[DateTime] = mapped_column(
        DateTime,
        default=datetime.now,
        onupdate=datetime.now,
    )
    runs: Mapped[List["Run"]] = relationship()
```

Code Listing A.2: Clase de la tabla experiment

A.3. Run

```
class Run(Base):
    __tablename__ = "run"
    """
    Table to store all the information about a specific run of
    a model.
    """
    id: Mapped[int] = mapped_column(primary_key=True)
    experiment_id: Mapped[int] = mapped_column(
        ForeignKey("experiment.id")
    )
    created: Mapped[DateTime] = mapped_column(
        DateTime, default=datetime.now
    )
    last_modified: Mapped[DateTime] = mapped_column(
        DateTime,
        default=datetime.now,
        onupdate=datetime.now,
    )
    # model and parameters
    model_name: Mapped[str] = mapped_column(String)
    parameters: Mapped[JSON] = mapped_column(JSON)
    # metrics
    train_metrics: Mapped[JSON] = mapped_column(
        JSON, nullable=True
    )
    test_metrics: Mapped[JSON] = mapped_column(
        JSON, nullable=True
    )
    validation_metrics: Mapped[JSON] = mapped_column(
        JSON, nullable=True
    )
```

```
    # artifacts
    artifacts: Mapped[str] = mapped_column(JSON, nullable=True)
    # metadata
    name: Mapped[str] = mapped_column(String)
    description: Mapped[str] = mapped_column(
        String, nullable=True
    )
    run_path: Mapped[str] = mapped_column(String, nullable=True)
    status: Mapped[Enum] = mapped_column(
        Enum(RunStatus),
        nullable=False,
        default=RunStatus.NOT_STARTED,
    )
    delivery_time: Mapped[DateTime] = mapped_column(
        DateTime, nullable=True
    )
    start_time: Mapped[DateTime] = mapped_column(
        DateTime, nullable=True
    )
    end_time: Mapped[DateTime] = mapped_column(
        DateTime, nullable=True
    )

    def set_status_as_delivered(self):
        """Updates the status of the run to delivered
        and set delivery_time to now"""
        self.status = RunStatus.DELIVERED
        self.delivery_time = datetime.now()

    def set_status_as_started(self):
        """Updates the status of the run to started
        and set start_time to now"""
        self.status = RunStatus.STARTED
        self.start_time = datetime.now()

    def set_status_as_finished(self):
        """Updates the status of the run to finished
        and set end_time to now"""
        self.status = RunStatus.FINISHED
        self.end_time = datetime.now()

    def set_status_as_error(self):
        """Updates the status of the run to error"""
        self.status = RunStatus.ERROR
```

Code Listing A.3: Clase de la tabla run

```
class RunStatus(Enum):
    NOT_STARTED = 0
    DELIVERED = 1
    STARTED = 2
    FINISHED = 3
    ERROR = 4
```

Code Listing A.4: Enumeración RunStatus

Anexo B

Código API

B.1. Endpoint Dataset

B.1.1. GET

B.1.1.1. /dataset/

```python
@router.get("/")
async def get_datasets(db: Session = Depends(get_db)):
    """Return all the available datasets in the database.

    Returns
    -------
    List[dict]
        A list of dict containing datasets.
    """
    try:
        all_datasets = db.query(Dataset).all()

    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e

    return all_datasets
```

Code Listing B.1: Endpoint GET /dataset/

B.1.1.2. /dataset/"dataset_id"

```python
@router.get("/{dataset_id}")
async def get_dataset(dataset_id: int, db: Session = Depends(get_db)):
```

```
    """Return the dataset with id dataset_id from the database.

    Parameters
    ----------
    dataset_id : int
        id of the dataset to query.

    Returns
    -------
    JSON
        JSON with the specified dataset id.
    """
    try:
        dataset = db.get(Dataset, dataset_id)
        if not dataset:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Dataset not found",
            )

    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e

    return dataset
```

Code Listing B.2: Endpoint GET `/dataset/"dataset_id"`

B.1.2. POST

B.1.2.1. `/dataset/`

```
@router.post("/", status_code=status.HTTP_201_CREATED)
async def upload_dataset(
    db: Session = Depends(get_db),
    params: str = Form(),
    url: str = Form(None),
    file: UploadFile = File(None),
):
    """
    Endpoint to upload datasets from user's input file or url.

    Parameters
    ----------
    params : str
        Dataset parameters in JSON format inside a string.
    url : str, optional
        For load the dataset from an URL.
    file : UploadFile, optional
        File uploaded
```

```
19
20      Returns
21      -------
22      JSON
23          JSON with the new dataset on the database
24      """
25      params = parse_params(params)
26      dataloader = dataloaders[params.dataloader]
27      folder_path = f"{settings.USER_DATASET_PATH}/{params.dataset_name}"
28
29      try:
30          os.makedirs(folder_path)
31      except FileExistsError as e:
32          log.exception(e)
33          raise HTTPException(
34              status_code=status.HTTP_409_CONFLICT,
35              detail="A dataset with this name already exists",
36          ) from e
37
38      try:
39          dataset = dataloader.load_data(
40              dataset_path=folder_path,
41              params=params.dataloader_params.dict(),
42              file=file,
43              url=url,
44          )
45          columns = dataset["train"].column_names
46          outputs_columns = params.outputs_columns
47
48          if len(outputs_columns) == 0:
49              inputs_columns = columns[:-1]
50              outputs_columns = [columns[-1]]
51          else:
52              inputs_columns = [x for x in columns if x not in
    outputs_columns]
53
54          dataset = to_dashai_dataset(dataset, inputs_columns,
    outputs_columns)
55
56          if not params.splits_in_folders:
57              dataset = dataloader.split_dataset(
58                  dataset,
59                  params.splits.train_size,
60                  params.splits.test_size,
61                  params.splits.val_size,
62                  params.splits.seed,
63                  params.splits.shuffle,
64                  params.splits.stratify,
65                  outputs_columns[0],  # Stratify according
66                  # to the split is only done in classification,
67                  # so it will correspond to the class column.
68              )
69
70          save_dataset(dataset, f"{folder_path}/dataset")
71
72      except OSError as e:
```

```
        log.exception(e)
        shutil.rmtree(folder_path, ignore_errors=True)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to read file",
        ) from e

    try:
        folder_path = os.path.realpath(folder_path)
        dataset = Dataset(
            name=params.dataset_name,
            task_name=params.task_name,
            file_path=folder_path,
        )
        db.add(dataset)
        db.commit()
        db.refresh(dataset)
        return dataset

    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.3: Endpoint POST `/dataset/`

B.1.3. PATCH

B.1.3.1. `/dataset/"dataset_id"`

```
@router.patch("/{dataset_id}")
async def update_dataset(
    dataset_id: int,
    db: Session = Depends(get_db),
    name: Union[str, None] = None,
    task_name: Union[str, None] = None,
):
    """Update a dataset name or task.

    Parameters
    ----------
    dataset_id : int
        id of the dataset to update.

    Returns
    -------
    JSON
        JSON containing the updated record
    """
    try:
        dataset = db.get(Dataset, dataset_id)
```

```
        if name:
            setattr(dataset, "name", name)
        if task_name:
            setattr(dataset, "task_name", task_name)
        if name or task_name:
            db.commit()
            db.refresh(dataset)
            return dataset
        else:
            raise HTTPException(
                status_code=status.HTTP_304_NOT_MODIFIED,
                detail="Record not modified",
            )
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.4: Endpoint PATCH /dataset/"dataset_id"

B.1.4. DELETE

B.1.4.1. /dataset/"dataset_id"

```
@router.delete("/{dataset_id}")
async def delete_dataset(dataset_id: int, db: Session = Depends(get_db)):
    """Return the dataset with id dataset_id from the database.

    Parameters
    ----------
    dataset_id : int
        id of the dataset to delete.

    Returns
    -------
    Response with code 204 NO_CONTENT
    """
    try:
        dataset = db.get(Dataset, dataset_id)
        if not dataset:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Dataset not found",
            )

        db.delete(dataset)
        db.commit()

    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
```

```
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e

    try:
        shutil.rmtree(dataset.file_path, ignore_errors=True)
        return Response(status_code=status.HTTP_204_NO_CONTENT)

    except OSError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to delete directory",
        ) from e
```

Code Listing B.5: Endpoint DELETE `/dataset/"dataset_id"`

B.2. Endpoint Experiment

B.2.1. GET

B.2.1.1. `/experiment/`

```
@router.get("/")
async def get_experiments(db: Session = Depends(get_db)):
    """Return all experiments stored in the database.

    Returns
    -------
    List[dict]
        A list of dict containing experiments.
    """
    try:
        all_experiments = db.query(Experiment).all()
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
    return all_experiments
```

Code Listing B.6: Endpoint GET `/experiment/`

B.2.1.2. `/experiment/"experiment_id"`

```
@router.get("/{experiment_id}")
async def get_experiment(experiment_id: int, db: Session = Depends(get_db)
    ):
    """Return the experiment with id experiment_id from the database.
```

```
    Parameters
    ----------
    experiment_id : int
        id of the experiment to query.

    Returns
    -------
    JSON
        JSON with the specified experiment id.
    """
    try:
        experiment = db.get(Experiment, experiment_id)
        if not experiment:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Experiment not found",
            )
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
    return experiment
```

Code Listing B.7: Endpoint GET `/experiment/"experiment_id"`

B.2.2. POST

B.2.2.1. `/experiment/`

```
@router.post("/", status_code=status.HTTP_201_CREATED)
async def upload_experiment(
    dataset_id: int,
    task_name: str,
    name: str,
    db: Session = Depends(get_db),
):
    """
    Endpoint to create experiments.

    Parameters
    ----------
    dataset_id : int
        Id of the Dataset linked to the experiment.
    task_name : str
        Name of the Task linked to the experiment.
    name : str
        Name of the experiment

    Returns
    -------
```

```
    JSON
        JSON with the new experiment on the database

    Raises
    ------
    HTTPException
        If the dataset with id dataset_id is not registered in the DB.
    """
    try:
        dataset = db.get(Dataset, dataset_id)
        if not dataset:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND, detail="Dataset not
    found"
            )
        experiment = Experiment(dataset_id=dataset_id, task_name=task_name
    , name=name)
        db.add(experiment)
        db.commit()
        db.refresh(experiment)
        return experiment
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.8: Endpoint POST `/experiment/`

B.2.3. PATCH

B.2.3.1. `/experiment/"experiment_id"`

```
@router.patch("/{experiment_id}")
async def update_dataset(
    experiment_id: int,
    db: Session = Depends(get_db),
    dataset_id: Union[int, None] = None,
    task_name: Union[str, None] = None,
    name: Union[str, None] = None,
):
    """Update the experiment information with id experiment_id from the
    database.

    Parameters
    ----------
    experiment_id : int
        id of the experiment to delete.

    Returns
    -------
    JSON
```

```
        JSON containing the updated record
    """
    try:
        experiment = db.get(Experiment, experiment_id)
        if dataset_id:
            setattr(experiment, "dataset_id", dataset_id)
        if task_name:
            setattr(experiment, "task_name", task_name)
        if name:
            setattr(experiment, "name", name)
        if dataset_id or task_name or name:
            db.commit()
            db.refresh(experiment)
            return experiment
        else:
            raise HTTPException(
                status_code=status.HTTP_304_NOT_MODIFIED,
                detail="Record not modified",
            )
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.9: Endpoint PATCH `/experiment/"experiment_id"`

B.2.4. DELETE

B.2.4.1. `/experiment/"experiment_id"`

```
@router.delete("/{experiment_id}")
async def delete_experiment(experiment_id: int, db: Session = Depends(
    get_db)):
    """Delete the experiment with id experiment_id from the database.

    Parameters
    ----------
    experiment_id : int
        id of the experiment to delete.

    Returns
    -------
    Response with code 204 NO_CONTENT
    """
    try:
        experiment = db.get(Experiment, experiment_id)
        if not experiment:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Experiment not found",
            )
```

```
        db.delete(experiment)
        db.commit()
        return Response(status_code=status.HTTP_204_NO_CONTENT)
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.10: Endpoint DELETE `/experiment/"experiment_id"`

B.3. Endpoint Run

B.3.1. GET

B.3.1.1. `/run/`

```
@router.get("/")
async def get_runs(
    experiment_id: Union[int, None] = None,
    db: Session = Depends(get_db),
):
    """Retrieve a list of runs from the DB.

    The runs can be filtered by experiment_id if the parameter is passed.

    Parameters
    ----------
    experiment_id: Union[int, None], optional
        If specified, the function will return all the runs associated
    with
        the experiment, by default None.

    Returns
    -------
    List[dict]
        A list with the information of all selected runs.

    Raises
    ------
    HTTPException
        If the experiment is not registered in the DB.
    """
    try:
        if experiment_id is not None:
            runs = db.scalars(
                select(Run).where(Run.experiment_id == experiment_id)
            ).all()
            if not runs:
                raise HTTPException(
```

```
                    status_code=status.HTTP_404_NOT_FOUND,
                    detail="Runs assoc with Experiment not found",
                )
        else:
            runs = db.query(Run).all()
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
    return runs
```

Code Listing B.11: Endpoint GET `/run/`

B.3.1.2. `/run/"run_id"`

```
@router.get("/{run_id}")
async def get_run_by_id(run_id: int, db: Session = Depends(get_db)):
    """Return the run with the specified id.

    Returns
    -------
    dict
        All the information of the selected run.

    Raises
    ------
    HTTPException
        If the run is not registered in the DB.
    """
    try:
        run = db.get(Run, run_id)
        if not run:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Run not found",
            )
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
    return run
```

Code Listing B.12: Endpoint GET `/run/"run_id"`

B.3.2. POST

B.3.2.1. `/run/`

```python
@router.post("/", status_code=status.HTTP_201_CREATED)
async def upload_run(
    experiment_id: int,
    model_name: str,
    name: str,
    parameters: dict,
    description: Union[str, None] = None,
    db: Session = Depends(get_db),
):
    """
    Endpoint to create a run.

    Parameters
    ----------
    experiment_id : int
        Id of the Experiment linked to the run.
    model_name : str
        Name of the Model linked to the run.
    name : str
        Name of the run
    parameters : dict
        Parameters to instantiate the run.
    description: Union[str, None], optional
        A brief description of the run, by default None.

    Returns
    -------
    dict
        A dictionary with the new run on the database

    Raises
    ------
    HTTPException
        If the experiment with id experiment_id is not registered in the DB.
    """
    try:
        experiment = db.get(Experiment, experiment_id)
        if not experiment:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND, detail="Experiment not found"
            )
        run = Run(
            experiment_id=experiment_id,
            model_name=model_name,
            parameters=parameters,
            name=name,
            description=description,
        )
        db.add(run)
        db.commit()
        db.refresh(run)
        return run
    except exc.SQLAlchemyError as e:
        log.exception(e)
```

```
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.13: Endpoint POST `/run/`

B.3.3. PATCH

B.3.3.1. `/run/"run_id"`

```
@router.patch("/{run_id}")
async def update_run(
    run_id: int,
    db: Session = Depends(get_db),
    run_name: Union[str, None] = None,
    run_description: Union[str, None] = None,
    parameters: Union[dict, None] = None,
):
    """Update the run information with id run_id from the DB.

    Parameters
    ----------
    run_id : int
        The id of the run to update.
    run_name : Union[str, None], optional
        The new name of the run, by default None.
    run_description : Union[str, None], optional
        The new description of the run, by default None.
    parameters : Union[dict, None], optional
        The new parameters of the run, by default None.

    Returns
    -------
    dict
        A dict containing the updated record

    Raises
    ------
    HTTPException
        If no parameters passed.
    """
    try:
        run = db.get(Run, run_id)
        if run_name:
            setattr(run, "name", run_name)
        if run_description:
            setattr(run, "description", run_description)
        if parameters:
            setattr(run, "parameters", parameters)
        if run_name or run_description or parameters:
            db.commit()
            db.refresh(run)
```

```
            return run
        else:
            raise HTTPException(
                status_code=status.HTTP_304_NOT_MODIFIED, detail="Record not modified"
            )
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
```

Code Listing B.14: Endpoint PATCH /run/"run_id"

B.3.4. DELETE

B.3.4.1. /run/"run_id"

```
@router.delete("/{run_id}")
async def delete_run(run_id: int, db: Session = Depends(get_db)):
    """Delete the run with id run_id from the DB.

    Parameters
    ----------
    run_id : int
        id of the run to delete.

    Returns
    -------
    Response with code 204 NO_CONTENT

    Raises
    ------
    HTTPException
        If the run is not registered in the DB.
    HTTPException
        If the run was trained but the run_path does not exists.
    """
    try:
        run = db.get(Run, run_id)
        if not run:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND, detail="Run not found"
            )
        db.delete(run)
        if run.status == RunStatus.FINISHED:
            os.remove(run.run_path)
        db.commit()
        return Response(status_code=status.HTTP_204_NO_CONTENT)
    except exc.SQLAlchemyError as e:
        log.exception(e)
```

```
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
    except OSError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to delete directory",
        ) from e
```

Code Listing B.15: Endpoint DELETE `/run/"run_id"`

B.4. Endpoint Job

B.4.1. GET

B.4.1.1. `/job/`

```
@router.get("/")
async def get_jobs():
    """Return all the jobs in the job queue.

    Returns
    ----------
    List[dict]
        A list of dict containing the Jobs.
    """
    all_jobs = job_queue.to_list()
    return all_jobs
```

Code Listing B.16: Endpoint GET `/job/`

B.4.1.2. `/job/"job_id"`

```
@router.get("/{job_id}")
async def get_job(job_id: int):
    """Return the selected job from the job queue

    Parameters
    ----------
    job_id: int
        id of the Job to get.

    Returns
    ----------
    dict
        A dict containing the Job information.

    Raises
```

```
    ----------
    HTTPException
        If is not posible to get the job from the job queue.
    """
    try:
        job = job_queue.peek(job_id)
    except JobQueueError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Job not found",
        ) from e
    return job
```

Code Listing B.17: Endpoint GET `/job/"job_id"`

B.4.1.3. `/job/start/`

```
@router.post("/start/")
async def start_job_queue(
    background_tasks: BackgroundTasks, stop_when_queue_empties: bool =
    False
):
    """Start the job queue to begin processing the jobs inside the jobs
    queue.
    If the param stop_when_queue_empties is True, the loop stops when the
    job queue
    becomes empty.

    Parameters
    ----------
    stop_when_queue_empties: Optional bool
        boolean to set the behavior of the loop.

    Returns
    -------
    Response
        response with code 202 ACCEPTED
    """
    background_tasks.add_task(job_queue_loop, stop_when_queue_empties)
    return Response(status_code=status.HTTP_202_ACCEPTED)
```

Code Listing B.18: Endpoint GET `/job/start/`

B.4.2. POST

B.4.2.1. `/job/runner/`

```
@router.post("/runner/", status_code=status.HTTP_201_CREATED)
async def enqueue_runner_job(run_id: int, db: Session = Depends(get_db)):
    """Create a runner job and put it in the job queue.

```

```
    Parameters
    ----------
    run_id : int
        Id of the Run to train and evaluate.
    Returns
    -------
    dict
        dict with the new job on the database
    """
    try:
        run: Run = db.get(Run, run_id)
        if not run:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND, detail="Run not
found"
            )
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e
    job = Job(
        func=execute_run,
        type=JobType.runner,
        kwargs={"run_id": run_id, "db": db},
    )
    job_queue.put(job)

    try:
        run.set_status_as_delivered()
        db.commit()
    except exc.SQLAlchemyError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Internal database error",
        ) from e

    return job
```

Code Listing B.19: Endpoint POST `/job/runner/`

B.4.3. DELETE

B.4.3.1. /job/"job_id"

```
@router.delete("/")
async def cancel_job(job_id: int):
    """Delete the job with id job_id from the job queue.

    Parameters
    ----------
```

```
    job_id : int
        id of the job to delete.

    Returns
    -------
    Response
        response with code 204 NO_CONTENT

    Raises
    ----------
    HTTPException
        If is not posible to get the job from the job queue.
    """
    try:
        job_queue.get(job_id)
    except JobQueueError as e:
        log.exception(e)
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Job not found",
        ) from e
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

Code Listing B.20: Endpoint DELETE `/job/"job_id"`

B.4.4. `job_queue_loop`

```
async def job_queue_loop(stop_when_queue_empties: bool):
    """Loop function to execute all the pending jobs in the job queue.
    If the the param stop_when_queue_empties is True, the loop returns
   when
    the queue empties, else it waits until  new jobs come in.

    Parameters
    ----------
    stop_when_queue_empties: bool
        boolean to set the while loop condition.
    """
    while not job_queue.is_empty() if stop_when_queue_empties else True:
        try:
            job: Job = await job_queue.async_get()
            job.func(**job.kwargs)
        except exc.SQLAlchemyError as e:
            log.exception(e)
        except RunnerError as e:
            log.exception(e)
```

Code Listing B.21: Función `job_queue_loop`

B.4.5. `exec_run`

```python
def execute_run(run_id: int, db: Session):
    """Function to train and evaluate a Run.
    It retrieves all the objects associated with the run and then it:
    - Trains the model.
    - Evaluate the model.
    - Save the trained model.

    Parameters
    ----------
    run_id: int
        id of the run to execute.

    Raises
    ----------
    RunnerError
        If an entity does not exist in the DB.
    RunnerError
        If the dataset does not exist in its path.
    RunnerError
        If unable to find a component in ComponentRegistry.
    RunnerError
        If the preparation of the dataset fails.
    RunnerError
        If the connection with the database fails.
    RunnerError
        If the training of the model fails.
    RunnerError
        If the evaluation of the model fails.
    RunnerError
        If the saving of the model fails.

    """
    run: Run = db.get(Run, run_id)
    if not run:
        raise RunnerError(f"Run {run_id} does not exist in DB.")
    try:
        experiment: Experiment = db.get(Experiment, run.experiment_id)
        if not experiment:
            raise RunnerError(f"Experiment {run.experiment_id} does not
    exist in DB.")
        dataset: Dataset = db.get(Dataset, experiment.dataset_id)
        if not dataset:
            raise RunnerError(f"Dataset {experiment.dataset_id} does not
    exist in DB.")

        try:
            loaded_dataset: DashAIDataset = load_dataset(f"{dataset.
    file_path}/dataset")
        except Exception as e:
            log.exception(e)
            raise RunnerError(
                f"Can not load dataset from path {dataset.file_path}",
            ) from e

        try:
            run_model_class = component_registry[run.model_name]["class"]
```

```
54          except Exception as e:
55              log.exception(e)
56              raise RunnerError(
57                  f"Unable to find Model with name {run.model_name} in
    registry.",
58              ) from e
59
60          try:
61              model: BaseModel = run_model_class(**run.parameters)
62          except Exception as e:
63              log.exception(e)
64              raise RunnerError(
65                  f"Unable to instantiate model using run {run_id}",
66              ) from e
67
68          try:
69              task: BaseTask = component_registry[experiment.task_name]["
    class"]()
70          except Exception as e:
71              log.exception(e)
72              raise RunnerError(
73                  f"Unable to find Task with name {experiment.task_name} in
    registry",
74              ) from e
75
76          try:
77              selected_metrics = {
78                  component_dict["name"]: component_dict
79                  for component_dict in component_registry.
    get_components_by_types(
80                      select="Metric"
81                  )
82              }
83              selected_metrics = _intersect_component_lists(
84                  selected_metrics,
85                  component_registry.get_related_components(experiment.
    task_name),
86              )
87              metrics: List[BaseMetric] = [
88                  metric["class"] for metric in selected_metrics.values()
89              ]
90          except Exception as e:
91              log.exception(e)
92              raise RunnerError(
93                  "Unable to find metrics associated with"
94                  f"Task {experiment.task_name} in registry",
95              ) from e
96
97          try:
98              prepared_dataset = task.prepare_for_task(loaded_dataset)
99          except Exception as e:
100             log.exception(e)
101             raise RunnerError(
102                 f"Can not prepare Dataset {dataset.id} for Task {
    experiment.task_name}",
103             ) from e
```

```
104
105         try:
106             run.set_status_as_started()
107             db.commit()
108         except exc.SQLAlchemyError as e:
109             log.exception(e)
110             raise RunnerError(
111                 "Connection with the database failed",
112             ) from e
113
114         try:
115             # Training
116             model.fit(prepared_dataset["train"])
117         except Exception as e:
118             log.exception(e)
119             raise RunnerError(
120                 "Model training failed",
121             ) from e
122
123         try:
124             run.set_status_as_finished()
125             db.commit()
126         except exc.SQLAlchemyError as e:
127             log.exception(e)
128             raise RunnerError(
129                 "Connection with the database failed",
130             ) from e
131
132         try:
133             model_metrics = {
134                 split: {
135                     metric.__name__: metric.score(
136                         prepared_dataset[split],
137                         model.predict(prepared_dataset[split]),
138                     )
139                     for metric in metrics
140                 }
141                 for split in ["train", "validation", "test"]
142             }
143         except Exception as e:
144             log.exception(e)
145             raise RunnerError(
146                 "Metrics calculation failed",
147             ) from e
148
149         run.train_metrics = model_metrics["train"]
150         run.validation_metrics = model_metrics["validation"]
151         run.test_metrics = model_metrics["test"]
152
153         try:
154             os.makedirs(settings.USER_RUN_PATH, exist_ok=True)
155             run_path = os.path.join(settings.USER_RUN_PATH, str(run.id))
156             model.save(run_path)
157         except Exception as e:
158             log.exception(e)
159             raise RunnerError(
```

```
                "Model saving failed",
            ) from e

        try:
            run.run_path = run_path
            db.commit()
        except exc.SQLAlchemyError as e:
            log.exception(e)
            run.set_status_as_error()
            db.commit()
            raise RunnerError(
                "Connection with the database failed",
            ) from e
    except Exception as e:
        run.set_status_as_error()
        db.commit()
        raise e
```

Code Listing B.22: Función `exec_run`

Anexo C

Código cola de trabajos

C.1. Estructura Job

C.1.1. Job

```python
class Job(BaseModel):
    """Model for abstracting a job."""

    id: Optional[int] = None
    func: Callable[[int, Session], None]
    type: JobType
    kwargs: dict

    @model_serializer
    def ser_job(self) -> Dict[str, Any]:
        """Returns a dict representation of the Job.

        Returns
        ----------
        dict
            dictionary with most of the fields of the Job.
        """
        return {"id": self.id, "type": self.type, "run_id": self.kwargs["
    run_id"]}
```

Code Listing C.1: Estructura Job

C.1.2. JobType

```python
class JobType(Enum):
    """Enumeration for job types."""

    runner = 0
```

Code Listing C.2: Enumeración JobType

C.2. JobQueue

C.2.1. BaseJobQueue

```python
class BaseJobQueue(metaclass=ABCMeta):
    """Abstract class for all Jobs Queues."""

    @abstractmethod
    def put(self, job: Job) -> int:
        """Put a job at the end of the queue.

        Parameters
        ----------
        job: Job
            Job to put in the queue.

        Returns
        ----------
        int
            The id of the job.
        """
        raise NotImplementedError

    @abstractmethod
    def get(self, job_id: int | None = None) -> Job:
        """Extract the job with id job_id from the queue.
        If the id is not specified, it extracts the first job in the queue
    .

        Parameters
        ----------
        job_id: Optional int
            id of the job to get.

        Returns
        ----------
        Job
            Extracted job from the queue.

        Raises
        ----------
        JobQueueError
            If the queue is empty.
        JobQueueError
            If there is not job with job_id in the queue.
        """
        raise NotImplementedError

    @abstractmethod
    async def async_get(self) -> Coroutine[Any, Any, Job]:
        """Tries to extract a Job from the queue,
        if the queue is empty waits until it has a Job to extract.

        Returns
```

```
50          ----------
51          Job
52              Extracted job from the queue.
53          """
54          raise NotImplementedError
55
56      @abstractmethod
57      def peek(self, job_id: int | None = None) -> Job:
58          """Retrieve the job with id job_id without removing it from the
    queue.
59          If the id is not specified, it retrieves the first job in the
    queue.
60
61          Parameters
62          ----------
63          job_id: Optional int
64              id of the job to peek.
65
66          Returns
67          -------
68          Job
69              Retrived Job from the queue.
70
71          Raises
72          ----------
73          JobQueueError
74              If the queue is empty.
75          JobQueueError
76              If there is not job with job_id in the queue.
77          """
78          raise NotImplementedError
79
80      @abstractmethod
81      def is_empty(self) -> bool:
82          """Predicate that indicates if the queue is empty.
83
84          Returns
85          ----------
86          bool
87              If the queue is empty.
88          """
89          raise NotImplementedError
90
91      @abstractmethod
92      def to_list(self) -> List[Job]:
93          """List all the jobs in the queue and returns them.
94
95          Returns
96          ----------
97          list Job
98              All the Jobs in the queue.
99          """
100         raise NotImplementedError
```

Code Listing C.3: Clase abstracta BaseJobQueue

C.2.2. SimpleJobQueue

```python
class SimpleJobQueue(BaseJobQueue):
    """JobQueue implementation using asyncio Queue."""

    queue: Queue = Queue()

    def _search_and_split(self, job_id: int) -> tuple[List[Job], Job, List[Job]]:
        """Split the queue using the job with id job_id as a pivot.

        Parameters
        ----------
        job_id: int
            Id of the Job used to split the queue.

        Returns
        ----------
        tuple(List Job, Job, List Job)
            A tuple containg the first part of the queue,
            the pivot Job and the second part of the queue.

        Raises
        ----------
        JobQueueError
            If the queue is empty.
        JobQueueError
            If there is not job with job_id in the queue.
        """
        first_part = []
        target_job: Job = self.queue.get_nowait()
        while target_job.id != job_id and not self.is_empty():
            first_part.append(target_job)
            target_job = self.queue.get_nowait()

        if target_job.id != job_id:
            for job in [*first_part, target_job]:
                self.queue.put_nowait(job)
            raise JobQueueError(
                f"Error trying to get job {job_id}: the job is not in the queue."
            )

        second_part = []
        while not self.is_empty():
            second_part.append(self.queue.get_nowait())

        return (first_part, target_job, second_part)

    def put(self, job: Job) -> int:
        job.id = uuid.uuid4().int
        self.queue.put_nowait(job)
        return job.id

    def get(self, job_id: int | None = None) -> Job:
        if self.is_empty():
```

```
            raise JobQueueError(
                f"Error trying to get job {job_id}: the async queue is empty."
            )

        if job_id:
            (first_part, target_job, second_part) = self._search_and_split(job_id)
            for job in [*first_part, *second_part]:
                self.queue.put_nowait(job)
            return target_job
        else:
            return self.queue.get_nowait()

    async def async_get(self) -> Coroutine[Any, Any, Job]:
        return await self.queue.get()

    def peek(self, job_id: int | None = None) -> Job:
        if self.is_empty():
            raise JobQueueError(
                f"Error trying to get job {job_id}: the async queue is empty."
            )

        if job_id:
            (first_part, target_job, second_part) = self._search_and_split(job_id)
            for job in [*first_part, target_job, *second_part]:
                self.queue.put_nowait(job)
            return job
        else:
            target_job: Job = self.queue.get_nowait()
            tmp_queue = Queue()
            tmp_queue.put_nowait(target_job)
            while not self.is_empty():
                tmp_queue.put_nowait(self.queue.get_nowait())
            self.queue = tmp_queue
            return target_job

    def is_empty(self) -> bool:
        return self.queue.empty()

    def to_list(self) -> List[Job]:
        job_list = []
        while not self.is_empty():
            job_list.append(self.queue.get_nowait())
        for job in job_list:
            self.queue.put_nowait(job)
        return job_list
```

Code Listing C.4: Clase SimpleJobQueue

Anexo D

Código tests

D.1. API

D.1.1. Endpoint Dataset

D.1.1.1. GET

D.1.1.1.1 /dataset/

```
def test_get_all_datasets(client: TestClient):
    response = client.get("/api/v1/dataset/")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data[0]["name"] == "test_csv"
    assert data[0]["task_name"] == "TabularClassificationTask"
    assert data[1]["name"] == "test_csv2"
```

Code Listing D.1: Test endpoint GET /dataset/

D.1.1.1.2 /dataset/"dataset_id"

```
def test_get_wrong_dataset(client: TestClient):
    response = client.get("/api/v1/dataset/31415")
    assert response.status_code == 404, response.text
    assert response.text == '{"detail":"Dataset not found"}'
```

Code Listing D.2: Test endpoint GET /dataset/"dataset_id"

D.1.1.2. POST

D.1.1.2.1 /dataset/

```python
def test_create_csv_dataset(client: TestClient):
    script_dir = os.path.dirname(__file__)
    test_dataset = "iris.csv"
    abs_file_path = os.path.join(script_dir, test_dataset)
    with open(abs_file_path, "rb") as csv:
        response = client.post(
            "/api/v1/dataset/",
            data={
                "params": """{  "task_name": "TabularClassificationTask",
                                "dataloader": "CSVDataLoader",
                                "dataset_name": "test_csv",
                                "outputs_columns": [],
                                "splits_in_folders": false,
                                "splits": {
                                    "train_size": 0.8,
                                    "test_size": 0.1,
                                    "val_size": 0.1,
                                    "seed": 42,
                                    "shuffle": true,
                                    "stratify": false
                                },
                                "dataloader_params": {
                                    "separator": ","
                                }
                            }""",
                "url": "",
            },
            files={"file": ("filename", csv, "text/csv")},
        )
        response = client.post(
            "/api/v1/dataset/",
            data={
                "params": """{  "task_name": "TabularClassificationTask",
                                "dataloader": "CSVDataLoader",
                                "dataset_name": "test_csv2",
                                "outputs_columns": [],
                                "splits_in_folders": false,
                                "splits": {
                                    "train_size": 0.5,
                                    "test_size": 0.2,
                                    "val_size": 0.3,
                                    "seed": 42,
                                    "shuffle": true,
                                    "stratify": false
                                },
                                "dataloader_params": {
                                    "separator": ","
                                }
                            }""",
                "url": "",
            },
            files={"file": ("filename", csv, "text/csv")},
        )
    assert response.status_code == 201, response.text
    response = client.get("/api/v1/dataset/1")
    assert response.status_code == 200, response.text
```

```
    data = response.json()
    assert data["name"] == "test_csv"
    assert data["task_name"] == "TabularClassificationTask"
    response = client.get("/api/v1/dataset/2")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["name"] == "test_csv2"
```

Code Listing D.3: Test endpoint POST `/dataset/`

D.1.1.3. PATCH

D.1.1.3.1 `/dataset/"dataset_id"`

```
def test_modify_dataset(client: TestClient):
    response = client.patch(
        "/api/v1/dataset/2",
        params={"name": "test_modify_name", "task_name": "UnknownTask"},
    )
    assert response.status_code == 200, response.text
    response = client.get("/api/v1/dataset/2")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["name"] == "test_modify_name"
    assert data["task_name"] == "UnknownTask"
```

Code Listing D.4: Test endpoint PATCH `/dataset/"dataset_id"`

D.1.1.4. DELETE

D.1.1.4.1 `/dataset/"dataset_id"`

```
def test_delete_dataset(client: TestClient):
    response = client.delete("/api/v1/dataset/1")
    assert response.status_code == 204, response.text

    response = client.delete("/api/v1/dataset/2")
    assert response.status_code == 204, response.text
```

Code Listing D.5: Test endpoint DELETE `/dataset/"dataset_id"`

D.1.2. Endpoint Experiment

D.1.2.1. Fixtures

D.1.2.1.1 `dataset_id`

```python
@pytest.fixture(scope="module", name="dataset_id")
def fixture_dataset_id(session: sessionmaker):
    db = session()
    # Create Dummy Dataset
    dummy_dataset = Dataset(
        name="DummyDataset",
        task_name="TabularClassificationTask",
        file_path="dummy.csv",
    )
    db.add(dummy_dataset)
    db.commit()
    db.refresh(dummy_dataset)
    yield dummy_dataset.id

    # Delete the dataset
    db.delete(dummy_dataset)
    db.commit()
```

Code Listing D.6: Fixture dataset_id test endpoints `/experiment`

D.1.2.2. GET

D.1.2.2.1 `/experiment/`

```python
def test_get_all_experiments(client: TestClient, dataset_id: int):
    # Get all the experiments available in the back
    response = client.get("/api/v1/experiment")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data[0]["dataset_id"] == dataset_id
    assert data[1]["dataset_id"] == dataset_id
```

Code Listing D.7: Test endpoint GET `/experiment/`

D.1.2.2.2 `/experiment/"experiment_id"`

```python
def test_get_wrong_experiment(client: TestClient):
    # Try to retrieve a non-existent experiment an get an error
    response = client.get("/api/v1/experiment/31415")
    assert response.status_code == 404, response.text
    assert response.text == '{"detail":"Experiment not found"}'
```

Code Listing D.8: Test endpoint GET `/experiment/"experiment_id"`

D.1.2.3. POST

D.1.2.3.1 `/experiment/`

```python
def test_create_experiment(client: TestClient, dataset_id: int):
    # Create Experiment using the dummy dataset
    response = client.post(
        f"/api/v1/experiment/?dataset_id={dataset_id}&"
        f"task_name=TabularClassificationTask&name=ExperimentA",
    )
    assert response.status_code == 201, response.text
    response = client.post(
        f"/api/v1/experiment/?dataset_id={dataset_id}"
        f"&task_name=TabularClassificationTask&name=Experiment2",
    )
    assert response.status_code == 201, response.text

    response = client.get("/api/v1/experiment/1")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["dataset_id"] == dataset_id
    assert data["task_name"] == "TabularClassificationTask"
    assert data["name"] == "ExperimentA"

    response = client.get("/api/v1/experiment/2")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["dataset_id"] == dataset_id
    assert data["task_name"] == "TabularClassificationTask"
    assert data["name"] == "Experiment2"
```

Code Listing D.9: Test endpoint POST `/experiment/`

D.1.2.4. PATCH

D.1.2.4.1 `/experiment/"experiment_id"`

```python
def test_modify_experiment(client: TestClient, dataset_id: int):
    # Modify an existent experiment
    response = client.patch(
        "/api/v1/experiment/2?task_name=UnknownTask&name=Experiment123",
    )
    assert response.status_code == 200, response.text

    # Get the experiment
    response = client.get("/api/v1/experiment/2")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["dataset_id"] == dataset_id
    assert data["task_name"] == "UnknownTask"
    assert data["name"] == "Experiment123"
    assert data["created"] != data["last_modified"]


def test_modify_experiment_step(client: TestClient):
    response = client.patch(
        "/api/v1/experiment/2",
        data={"params": """{"step": "STARTED"}""", "url": ""},
```

```
    )
    assert response.status_code == 304, response.text
```

Code Listing D.10: Test endpoint PATCH `/experiment/"experiment_id"`

D.1.2.5. DELETE

D.1.2.5.1 `/experiment/"experiment_id"`

```
def test_delete_experiment(client: TestClient):
    # Delete all the experiments in the db
    response = client.delete("/api/v1/experiment/1")
    assert response.status_code == 204, response.text
    response = client.delete("/api/v1/experiment/2")
    assert response.status_code == 204, response.text
```

Code Listing D.11: Test endpoint DELETE `/experiment/"experiment_id"`

D.1.3. Endpoint Run

D.1.3.1. Fixtures

D.1.3.1.1 `experiment_id`

```
@pytest.fixture(scope="module", name="experiment_id")
def fixture_experiment_id(session: sessionmaker):
    db = session()
    # Create Dummy Dataset
    dummy_dataset = Dataset(
        name="DummyDataset",
        task_name="TabularClassificationTask",
        file_path="dummy.csv",
    )
    db.add(dummy_dataset)
    db.commit()
    db.refresh(dummy_dataset)
    # Create Dummy Experiment
    dummy_experiment = Experiment(
        dataset_id=dummy_dataset.id,
        task_name="TabularClassificationTask",
        name="Test Experiment",
    )
    db.add(dummy_experiment)
    db.commit()
    db.refresh(dummy_experiment)
    yield dummy_experiment.id

    # Delete the dataset and experiment
    db.delete(dummy_experiment)
    db.delete(dummy_dataset)
```

```
    db.commit()
```

Code Listing D.12: Fixture experiment_id test endpoints `/run`

D.1.3.2. GET

D.1.3.2.1 `/run/`

```
def test_get_all_runs(client: TestClient, experiment_id: int):
    response = client.get(f"/api/v1/run/?experiment_id={experiment_id}")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data[0]["experiment_id"] == experiment_id
    assert data[1]["experiment_id"] == experiment_id

def test_get_wrong_runs(client: TestClient):
    # Try to retrieve a list of runs from a non-existent experiment an get
    an error
    response = client.get("/api/v1/run/?experiment_id=31415")
    assert response.status_code == 404, response.text
    assert response.text == '{"detail":"Runs assoc with Experiment not
   found"}'
```

Code Listing D.13: Test endpoint GET `/run/`

D.1.3.2.2 `/run/"run_id"`

```
def test_get_run(client: TestClient):
    response = client.get("/api/v1/run/1")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["name"] == "Run1"
    response = client.get("/api/v1/run/2")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["name"] == "Run2"

def test_get_wrong_run(client: TestClient):
    # Try to retrieve a non-existent run an get an error
    response = client.get("/api/v1/run/31415")
    assert response.status_code == 404, response.text
    assert response.text == '{"detail":"Run not found"}'
```

Code Listing D.14: Test endpoint GET `/run/"run_id"`

D.1.3.3. POST

D.1.3.3.1 `/run/`

```python
def test_create_run(client: TestClient, experiment_id: int):
    # Create Run using the dummy Experiment
    response = client.post(
        f"/api/v1/run/?experiment_id={experiment_id}&"
        f"model_name=KNeighborsClassifier&name=Run1",
        json={
            "n_neighbors": 5,
            "weights": "uniform",
            "algorithm": "auto",
        },
    )
    assert response.status_code == 201, response.text
    response = client.post(
        f"/api/v1/run/?experiment_id={experiment_id}&"
        f"model_name=KNeighborsClassifier&name=Run2",
        json={
            "n_neighbors": 3,
            "weights": "uniform",
            "algorithm": "kd_tree",
        },
    )
    assert response.status_code == 201, response.text
    response = client.get("/api/v1/run/1")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["experiment_id"] == experiment_id
    assert data["model_name"] == "KNeighborsClassifier"
    assert data["name"] == "Run1"
    assert data["status"] == 0
    assert data["parameters"] == {
        "n_neighbors": 5,
        "weights": "uniform",
        "algorithm": "auto",
    }

    response = client.get("/api/v1/run/2")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["experiment_id"] == experiment_id
    assert data["model_name"] == "KNeighborsClassifier"
    assert data["name"] == "Run2"
    assert data["status"] == 0
    assert data["parameters"] == {
        "n_neighbors": 3,
        "weights": "uniform",
        "algorithm": "kd_tree",
    }
```

Code Listing D.15: Test endpoint POST `/run/`

D.1.3.4. PATCH

D.1.3.4.1 `/run/"run_id"`

```python
def test_modify_run(client: TestClient):
    response = client.patch(
        "/api/v1/run/1?run_name=RunA",
        json={
            "n_neighbors": 3,
            "weights": "uniform",
            "algorithm": "kd_tree",
        },
    )
    assert response.status_code == 200, response.text

    response = client.get("/api/v1/run/1")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["name"] == "RunA"
    assert data["status"] == 0
    assert data["parameters"] == {
        "n_neighbors": 3,
        "weights": "uniform",
        "algorithm": "kd_tree",
    }
    assert data["created"] != data["last_modified"]


def test_modify_run_model(client: TestClient):
    response = client.patch(
        "/api/v1/run/2?model_name=UnknownModel",
    )
    assert response.status_code == 304, response.text
```

Code Listing D.16: Test endpoint PATCH `/run/"run_id"`

D.1.3.5. DELETE

D.1.3.5.1 `/run/"run_id"`

```python
def test_delete_run(client: TestClient):
    # Delete all the runs in the db
    response = client.delete("/api/v1/run/1")
    assert response.status_code == 204, response.text
    response = client.delete("/api/v1/run/2")
    assert response.status_code == 204, response.text
```

Code Listing D.17: Test endpoint DELETE `/run/"run_id"`

D.1.4. Endpoint Job

D.1.4.1. Fixtures

D.1.4.1.1 `Dummy Task`

```python
class DummyTask(BaseTask):
    name: str = "DummyTask"

    def prepare_for_task(self, dataset):
        return {
            "train": {"input": [], "output": []},
            "validation": {"input": [], "output": []},
            "test": {"input": [], "output": []},
        }
```

Code Listing D.18: Dummy Task used in test endpoint `/job`

D.1.4.1.2 Dummy Models

```python
class DummyModel(BaseModel):
    COMPATIBLE_COMPONENTS = ["DummyTask"]

    @classmethod
    def get_schema(cls):
        return {}

    def save(self, filename):
        joblib.dump(self, filename)

    def load(self, filename):
        return

    def predict(self, data):
        return {}

    def fit(self, data):
        return


class FailDummyModel(BaseModel):
    COMPATIBLE_COMPONENTS = ["DummyTask"]

    @classmethod
    def get_schema(cls):
        return {}

    def save(self, filename):
        return

    def load(self, filename):
        return

    def predict(self, data):
        return {}

    def fit(self, data):
        raise Exception("Always fails")
```

Code Listing D.19: Dummy Models used in test endpoint `/job`

D.1.4.1.3 Dummy Metric

```python
class DummyMetric(BaseMetric):
    COMPATIBLE_COMPONENTS = ["DummyTask"]

    @staticmethod
    def score(true_labels: list, probs_pred_labels: list):
        return 1
```

Code Listing D.20: Dummy Task used in test endpoint `/job`

D.1.4.1.4 override_registry

```python
@pytest.fixture(scope="module", autouse=True)
def override_registry():
    original_registry = component_registry._registry
    original_relationships = component_registry._relationship_manager

    test_registry = ComponentRegistry(
        initial_components=[
            DummyTask,
            DummyModel,
            FailDummyModel,
            DummyMetric,
        ]
    )

    # replace the default dataloaders with the previously test dataloaders
    component_registry._registry = test_registry._registry
    component_registry._relationship_manager = test_registry.
   _relationship_manager

    yield test_registry

    # cleanup: restore orginal registers
    component_registry._registry = original_registry
    component_registry._relationship_manager = original_relationships
```

Code Listing D.21: Fixture `override_registry` test endpoints `/job`

D.1.4.1.5 dataset_id

```python
@pytest.fixture(scope="module", name="dataset_id")
def fixture_dataset_id(client: TestClient):
    script_dir = os.path.dirname(__file__)
    test_dataset = "iris.csv"
    abs_file_path = os.path.join(script_dir, test_dataset)
    with open(abs_file_path, "rb") as csv:
        response = client.post(
            "/api/v1/dataset/",
            data={
                "params": """{  "task_name": "DummyTask",
```

```
                                    "dataloader": "CSVDataLoader",
                                    "dataset_name": "test_csv2",
                                    "outputs_columns": [],
                                    "splits_in_folders": false,
                                    "splits": {
                                        "train_size": 0.5,
                                        "test_size": 0.2,
                                        "val_size": 0.3,
                                        "seed": 42,
                                        "shuffle": true,
                                        "stratify": false
                                    },
                                    "dataloader_params": {
                                        "separator": ","
                                    }
                                }""",
                "url": "",
            },
            files={"file": ("filename", csv, "text/csv")},
        )
    assert response.status_code == 201, response.text
    dataset = response.json()

    yield dataset["id"]

    response = client.delete(f"/api/v1/dataset/{dataset['id']}")
    assert response.status_code == 204, response.text
```

Code Listing D.22: Fixture `dataset_id` test endpoints `/job`

D.1.4.1.6 `experiment_id`

```
@pytest.fixture(scope="module", name="experiment_id")
def fixture_experiment_id(session: sessionmaker, dataset_id: int):
    db = session()

    experiment = Experiment(
        dataset_id=dataset_id, name="DummyExperiment", task_name="
    DummyTask"
    )
    db.add(experiment)
    db.commit()
    db.refresh(experiment)

    yield experiment.id

    db.delete(experiment)
    db.commit()
    db.close()
```

Code Listing D.23: Fixture `experiment_id` test endpoints `/job`

D.1.4.1.7 `run_id`

```python
@pytest.fixture(scope="module", name="run_id")
def fixture_run_id(client: TestClient, experiment_id: int):
    response = client.post(
        f"/api/v1/run/?experiment_id={experiment_id}&"
        f"model_name=DummyModel&name=DummyRun",
        json={},
    )
    assert response.status_code == 201, response.text
    run = response.json()

    yield run["id"]

    response = client.delete(f"/api/v1/run/{run['id']}")
    assert response.status_code == 204, response.text
```

Code Listing D.24: Fixture `run_id` test endpoints `/job`

D.1.4.1.8 `failed_run_id`

```python
@pytest.fixture(scope="module", name="failed_run_id")
def fixture_failed_run_id(session: sessionmaker, experiment_id: int):
    db = session()

    run = Run(
        experiment_id=experiment_id,
        model_name="FailDummyModel",
        parameters={},
        name="DummyRun2",
    )
    db.add(run)
    db.commit()
    db.refresh(run)

    yield run.id

    db.delete(run)
    db.commit()
    db.close()
```

Code Listing D.25: Fixture `failed_run_id` test endpoints `/job`

D.1.4.2. GET

D.1.4.2.1 `/job/`

```python
def test_get_all_jobs(client: TestClient, run_id: int):
    # Get all the experiments available in the back
    response = client.get("/api/v1/job")
    assert response.status_code == 200, response.text
    data = response.json()
    assert data[0]["run_id"] == run_id
```

```
    assert data[1]["run_id"] == run_id
```

Code Listing D.26: Test endpoint GET `/job/`

D.1.4.2.2 `/job/"job_id"`

```
def test_get_wrong_job(client: TestClient):
    # Try to retrieve a non-existent experiment an get an error
    response = client.get("/api/v1/job/31415")
    assert response.status_code == 404, response.text
    assert response.text == '{"detail":"Job not found"}'
```

Code Listing D.27: Test endpoint GET `/job/"job_id"`

D.1.4.2.3 `/job/start/`

```
def test_execute_jobs(client: TestClient, run_id: int, failed_run_id: int)
    :
    response = client.post(f"/api/v1/job/runner/?run_id={run_id}")
    assert response.status_code == 201, response.text

    response = client.post(f"/api/v1/job/runner/?run_id={failed_run_id}")
    assert response.status_code == 201, response.text

    response = client.get("/api/v1/run")
    data = response.json()
    for run in data:
        assert run["status"] == 1
        assert run["delivery_time"] is not None
        assert run["start_time"] is None
        assert run["end_time"] is None

    response = client.post("/api/v1/job/start/?stop_when_queue_empties=
    True")
    assert response.status_code == 202, response.text

    response = client.get(f"/api/v1/run/{run_id}")
    data = response.json()
    assert data["status"] == 3
    assert isinstance(data["train_metrics"], dict)
    assert "DummyMetric" in data["train_metrics"]
    assert data["train_metrics"]["DummyMetric"] == 1
    assert data["train_metrics"] == data["validation_metrics"]
    assert data["train_metrics"] == data["test_metrics"]
    assert data["run_path"] is not None
    assert os.path.exists(data["run_path"])
    assert data["status"] == 3
    assert data["delivery_time"] is not None
    assert data["start_time"] is not None
    assert data["end_time"] is not None

    response = client.get(f"/api/v1/run/{failed_run_id}")
    data = response.json()
```

```
    assert data["status"] == 4
    assert data["delivery_time"] is not None
    assert data["start_time"] is not None
    assert data["end_time"] is None
```

Code Listing D.28: Test endpoint GET `/job/start/`

D.1.4.3. POST

D.1.4.3.1 `/job/runner/`

```
def test_enqueue_jobs(client: TestClient, run_id: int):
    response = client.post(f"/api/v1/job/runner/?run_id={run_id}")
    assert response.status_code == 201, response.text
    created_job = response.json()
    assert created_job["type"] == 0
    assert created_job["run_id"] == run_id

    response = client.get(f"/api/v1/job/{created_job['id']}")
    assert response.status_code == 200, response.text
    gotten_job = response.json()
    assert gotten_job["id"] == created_job["id"]
    assert gotten_job["type"] == created_job["type"]
    assert gotten_job["run_id"] == created_job["run_id"]

    response = client.post(f"/api/v1/job/runner/?run_id={run_id}")
    assert response.status_code == 201, response.text
    created_job_2 = response.json()
    assert created_job_2["id"] != created_job["id"]

    response = client.get("/api/v1/job")
    assert response.status_code == 200, response.text
    gotten_jobs = response.json()
    assert gotten_jobs[0]["id"] == created_job["id"]
    assert gotten_jobs[1]["id"] == created_job_2["id"]

def test_job_with_wrong_run(client: TestClient):
    response = client.post("/api/v1/job/runner/?run_id=31415")
    assert response.status_code == 404, response.text
    assert response.text == '{"detail":"Run not found"}'
```

Code Listing D.29: Test endpoint POST `/job/runner/`

D.1.4.4. DELETE

D.1.4.4.1 `/job/"job_id"`

```
def test_cancel_jobs(client: TestClient):
    response = client.get("/api/v1/job")
    assert response.status_code == 200, response.text
    gotten_jobs = response.json()
```

```
5
6       response = client.delete(f"/api/v1/job/?job_id={gotten_jobs[0]['id']}"
        )
7       assert response.status_code == 204, response.text
8
9       response = client.get("/api/v1/job")
10      assert response.status_code == 200, response.text
11      jobs = response.json()
12      assert len(jobs) == len(gotten_jobs) - 1
13      assert jobs[0]["id"] != gotten_jobs[0]["id"]
14      assert jobs[0]["id"] == gotten_jobs[1]["id"]
15
16      response = client.delete(f"/api/v1/job/?job_id={gotten_jobs[1]['id']}"
        )
17      assert response.status_code == 204, response.text
18
19      response = client.get("/api/v1/job")
20      assert response.status_code == 200, response.text
21      jobs = response.json()
22      assert jobs == []
```

Code Listing D.30: Test endpoint GET `/job/"job_id"`

D.2. Cola de trabajos

D.2.1. SimpleJobQueue

D.2.1.1. `put`

```
1  def test_enqueue_jobs(job_queue: BaseJobQueue):
2      job_1 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
3      job_1_id = job_queue.put(job_1)
4      assert job_1.id == job_1_id
5
6      job_2 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
7      job_2_id = job_queue.put(job_2)
8      assert job_2.id == job_2_id
9      assert job_1_id != job_2_id
10
11     job_3 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
12     job_3_id = job_queue.put(job_3)
13     assert job_3.id == job_3_id
14     assert job_1_id != job_3_id
15     assert job_2_id != job_3_id
```

Code Listing D.31: Test método `put` clase SimpleJobQueue

D.2.1.2. `get`

```python
def test_get_jobs(job_queue: BaseJobQueue):
    job_1 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_1_id = job_queue.put(job_1)
    job_2 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_2_id = job_queue.put(job_2)
    job_3 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_3_id = job_queue.put(job_3)

    assert job_queue.get().id == job_1_id
    assert job_queue.get(job_3_id).id == job_3_id

    jobs = job_queue.to_list()
    assert len(jobs) == 1
    assert jobs[0].id == job_2_id

def test_get_from_empty_queue(job_queue: BaseJobQueue):
    with pytest.raises(JobQueueError):
        job_queue.get()
    with pytest.raises(JobQueueError):
        job_queue.get(job_id=0)

def test_get_nonexistent_job(job_queue: BaseJobQueue):
    job = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_queue.put(job)
    with pytest.raises(JobQueueError):
        job_queue.get(job_id=-1)
```

Code Listing D.32: Test método `get` clase SimpleJobQueue

D.2.1.3. async_get

```python
@pytest.mark.asyncio()
async def test_async_get_job(job_queue: BaseJobQueue):
    job_1 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_1_id = job_queue.put(job_1)

    job = await job_queue.async_get()
    assert job.id == job_1_id
```

Code Listing D.33: Test método `async_get` clase SimpleJobQueue

D.2.1.4. peek

```python
def test_peek_jobs(job_queue: BaseJobQueue):
    job_1 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_1_id = job_queue.put(job_1)
    job_2 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_2_id = job_queue.put(job_2)
    job_3 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
    job_3_id = job_queue.put(job_3)

    assert job_queue.peek().id == job_1_id
    assert job_queue.peek(job_3_id).id == job_3_id
```

```
11
12      jobs = job_queue.to_list()
13      assert len(jobs) == 3
14      assert jobs[0].id == job_1_id
15      assert jobs[1].id == job_2_id
16      assert jobs[2].id == job_3_id
17
18  def test_peek_from_empty_queue(job_queue: BaseJobQueue):
19      with pytest.raises(JobQueueError):
20          job_queue.peek()
21      with pytest.raises(JobQueueError):
22          job_queue.peek(job_id=0)
23
24  def test_peek_nonexistent_job(job_queue: BaseJobQueue):
25      job = Job(func=lambda x: x, type=JobType.runner, kwargs={})
26      job_queue.put(job)
27      with pytest.raises(JobQueueError):
28          job_queue.peek(job_id=-1)
```

Code Listing D.34: Test método `peek` clase SimpleJobQueue

D.2.1.5. `is_empty`

```
1  def test_empty_queue(job_queue: BaseJobQueue):
2      assert job_queue.is_empty()
3
4      job = Job(func=lambda x: x, type=JobType.runner, kwargs={})
5      job_queue.put(job)
6
7      assert not job_queue.is_empty()
```

Code Listing D.35: Test método `is_empty` clase SimpleJobQueue

D.2.1.6. `to_list`

```
1  def test_queue_to_list(job_queue: BaseJobQueue):
2      jobs = job_queue.to_list()
3      assert isinstance(jobs, list)
4      assert jobs == []
5
6      job_1 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
7      job_1_id = job_queue.put(job_1)
8      jobs = job_queue.to_list()
9      assert len(jobs) == 1
10     assert jobs[0].id == job_1_id
11
12     job_2 = Job(func=lambda x: x, type=JobType.runner, kwargs={})
13     job_2_id = job_queue.put(job_2)
14     jobs = job_queue.to_list()
15     assert jobs[0].id == job_1_id
16     assert jobs[1].id == job_2_id
```

Code Listing D.36: Test método `to_list` clase SimpleJobQueue