



Universidad  
Rey Juan Carlos

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA  
TELECOMUNICACIÓN

Curso Académico 2021/2022

Trabajo Fin de Grado

METODOLOGÍAS Y HERRAMIENTAS PARA  
DESPLIEGUE DE PROYECTOS DE APRENDIZAJE  
AUTOMÁTICO

Autor : Meritxell Maider Díaz Coque

Tutor : Dr. José Felipe Ortega Soto



# Trabajo Fin de Grado

Metodologías y Herramientas para Despliegue de Proyectos de Aprendizaje  
Automático

**Autor :** Meritxell Maider Díaz Coque

**Tutor :** Dr. José Felipe Ortega Soto

La defensa del presente Proyecto Fin de Carrera se realizó el día                      de  
de 202X, siendo calificada por el siguiente tribunal:

**Presidente:**

**Secretario:**

**Vocal:**

y habiendo obtenido la siguiente calificación:

**Calificación:**

Fuenlabrada, a                      de                      de 202X

---

*Dedicado a  
mi madre y pareja*



## Agradecimientos

En primer lugar quiero dar las gracias por el apoyo incondicional de toda una vida a mi mejor amiga, mi madre. Gracias a ella estudié esta carrera, no solo porque sin ella no sabría que existe este doble grado, sino porque me ha ayudado a superar los momentos más difíciles en el camino. Con su ayuda siempre logro todo lo que me propongo.

A mi gato, Pepe, por hacerme ver que es bueno desconectar de vez en cuando y por estar literalmente en todo momento a mi lado mientras realizaba este proyecto.

A mi pareja, Víctor, por apoyarme cada día desde que nos conocimos. Por estar en todo momento cuando las cosas van bien y no tan bien, por ser esa balsa de tranquilidad que siempre me escucha y, en concreto en este proyecto, por aconsejarme sobre cómo mejorar los casos prácticos y ayudarme durante varias tardes a leer esta memoria para que esté todo perfecto. Gracias a sus consejos he aprendido mucho.

También quiero dar las gracias a la familia de mi pareja, en especial a sus padres. Me han tratado desde el primer momento como una más de la familia y me han acogido en su casa en esos huecos de varias horas de mi horario.

Por último, y no por ello menos importante, a mi tutor Felipe, por todo lo que me ha enseñado en este proyecto, por su tiempo y paciencia en responderme dudas y por nuestras reuniones que siempre se alargaban.





# Resumen

Actualmente no existe una metodología unificada para llevar a cabo la creación de modelos de aprendizaje máquina (*machine learning*, ML) y su despliegue a producción de manera óptima, ni automatizada. Algunas empresas logran llevar a cabo este proceso de manera ineficiente, invirtiendo grandes cantidades de recursos en ello.

Este proyecto pretende poner de manifiesto la situación actual en la que se encuentran las empresas, analizar las etapas de las que se compone el ciclo de vida de un modelo de ML y, por último, proponer y aclarar los elementos, funciones y tecnologías necesarias para llevar a cabo el proceso anterior de manera automática sin apenas intervención manual.

Para ello, se analizan las herramientas que facilitan llevar a cabo este proceso, y se utilizan tres de ellas (MLflow, Apache Airflow y Data Version Control) para ilustrar, de manera práctica, cómo debe llevarse a cabo la creación y despliegue de modelos de ML.

Finalmente, se concluye con un análisis de aquellos elementos del proceso automatizado que aún no es posible implementar, debido a la inexistencia de tecnologías que los lleven a cabo, junto con una comparativa entre las fortalezas y debilidades de las herramientas utilizadas en los casos prácticos implementados. En concreto, para las tres herramientas mencionadas, se puede ver que MLflow y Data Version Control se solapan en lo que respecta a la ejecución de experimentos, mientras que Airflow ofrece una funcionalidad alternativa a la hora de orquestar de forma automática los distintos pasos del flujo de trabajo con datos.



# Summary

Nowadays, there is no standard methodology or definition for creating machine learning (ML) models and deploying them in production systems, in an optimal and automated way. Some companies manage to carry out this process inefficiently, investing large amounts of resources during this process.

This project aims to highlight the current situation faced by numerous companies, analyse the stages that constitute the life cycle of an ML model and, then, propose and clarify the required elements, functions and technologies to carry out the above process automatically, almost without any manual intervention.

To this end, the tools that facilitate this process are analysed, and three of them (MLflow, Apache Airflow and Data Version Control) are used to illustrate, from a practical perspective, how the creation and deployment of ML models is accomplished.

Finally, this project concludes with a study of those elements of the automated process that cannot be implemented yet, due to the absence of technologies to undertake them, along with a comparison between strengths and weaknesses of the tools involved in the implemented practical cases. Specifically, for the three aforementioned tools, it can be seen that MLflow and Data Version Control overlap in terms of experiment execution, whereas Airflow offers an alternative functionality, regarding the automated orchestration of different tasks.



# Índice general

<b>1</b>	<b>Introducción</b>	<b>1</b>
1.1	Machine learning en producción	2
1.1.1	Big Data y aprendizaje máquina en las organizaciones	4
1.2	Objetivos del proyecto	6
1.3	Planificación temporal	7
1.4	Estructura de la memoria	8
<b>2</b>	<b>Estado del arte</b>	<b>11</b>
2.1	Data Version Control	12
2.2	MLflow	14
2.3	Apache Airflow	16
2.4	DataRobot	18
2.5	OpenML	20
2.6	Delta Lake	21
<b>3</b>	<b>Marco teórico</b>	<b>25</b>
3.1	Escenarios de MLOps	26
3.1.1	Procedimiento manual	26
3.1.2	Entrega continua de modelos	27
3.1.3	Proceso totalmente automatizado	28
3.2	Herramientas asociadas al proceso automatizado	32
3.3	Automatización usando <i>pipelines</i>	35
<b>4</b>	<b>Diseño e implementación práctica</b>	<b>37</b>
4.1	Datos y procesado	38
4.1.1	Datos	38
4.1.2	Procesado	40
4.1.3	Parámetros	41
4.2	Metodología común	43
4.2.1	Estructura general	43

4.2.2	Organización en carpetas . . . . .	44
4.3	Criterios a comparar . . . . .	45
4.4	Caso de uso del presente proyecto . . . . .	47
<b>5</b>	<b>Primer caso práctico</b>	<b>49</b>
5.1	Diseño . . . . .	49
5.2	Implementación . . . . .	52
5.2.1	Consideraciones de implementación . . . . .	55
5.3	Resultados . . . . .	57
<b>6</b>	<b>Segundo caso práctico</b>	<b>61</b>
6.1	Diseño . . . . .	62
6.2	Implementación . . . . .	64
6.2.1	Consideraciones de implementación . . . . .	66
6.3	Resultados . . . . .	69
<b>7</b>	<b>Conclusiones</b>	<b>73</b>
7.1	Consecución de objetivos . . . . .	77
7.2	Aplicación de lo aprendido . . . . .	77
7.3	Lecciones aprendidas . . . . .	78
7.4	Trabajos futuros . . . . .	78
<b>A</b>	<b>Requisitos de ejecución de los casos prácticos</b>	<b>81</b>
A.1	Primer caso práctico . . . . .	81
A.2	Segundo caso práctico . . . . .	82

# Índice de figuras

1.1	Ciclo de vida de un modelo de ML . . . . .	3
1.2	Fases en la planificación de ciencia de datos . . . . .	5
1.3	Diagrama de Gantt que ilustra la planificación temporal . . . . .	8
2.1	Cambio de enfoque de datos y modelos sin versionar a un escenario donde todo está organizado y versionado por git y DVC . . . . .	13
2.2	Esquema de los módulos y funcionalidad que abarca de MLflow . . . . .	15
2.3	Esquema de la relación entre los distintos módulos de MLflow . . . . .	16
2.4	Esquema de los conceptos clave de Apache Airflow . . . . .	17
2.5	Estructura de Apache Airflow . . . . .	18
2.6	Esquema de DataRobot . . . . .	19
2.7	Ubicación de Delta Lake en el almacenamiento de datos . . . . .	22
3.1	Organización de los equipos técnicos en la empresa . . . . .	26
3.2	Esquema del desarrollo y despliegue de un modelo de ML totalmente automatizado	29
3.3	Diagrama de bloques que muestra la relación entre las distintas áreas . . . . .	31
3.4	Diagrama de bloques genérico de las herramientas asociadas a un proceso total- mente automatizado . . . . .	32
3.5	Tareas y su implementación usando <i>pipelines</i> . . . . .	35
4.1	Esquema de las herramientas a utilizar en los casos prácticos. . . . .	38
4.2	Indicadores utilizados en la elaboración de la clasificación de mejores universida- des del mundo de THE . . . . .	39
5.1	Diseño simplificado de DVC en el primer caso práctico . . . . .	50
5.2	Captura de pantalla del DAG de Airflow del primer caso práctico . . . . .	52
5.3	Captura de pantalla del DAG de DVC del primer caso práctico . . . . .	53
5.4	Métricas analizadas en el primer caso práctico . . . . .	54
5.5	Matriz de confusión generada por DVC . . . . .	54
5.6	Evolución de la métrica precisión generada por DVC . . . . .	55
5.7	Evolución de la métrica precisión en los experimentos Sf3 y Snest64 generado por DVC . . . . .	56

---

6.1	Configuración de almacenamiento elegida en MLflow . . . . .	62
6.2	Diseño simplificado llevado a cabo con MLflow en el segundo caso práctico . . .	63
6.3	Captura de pantalla del DAG del segundo caso práctico . . . . .	65
6.4	Captura de pantalla de las <i>runs</i> del segundo caso práctico . . . . .	66
6.5	Captura de pantalla del <i>model registry</i> ofrecido por MLflow . . . . .	67
6.6	Captura de pantalla de la información almacenada en el <i>model registry</i> sobre el modelo generado en el experimento <i>cwurData</i> . . . . .	68
6.7	Captura de pantalla de la comparativa entre los modelos almacenados en el <i>model registry</i> . . . . .	69



## Índice de tablas

4.1	Tabla en la que se detallan parámetros asociados a cada experimento desarrollado	42
5.1	Tabla que muestra las funcionalidades que provee la herramienta DVC . . . . .	60
6.1	Tabla que muestra las funcionalidades que provee la herramienta MLflow . . . . .	71
7.1	Tabla comparativa de las dimensiones analizadas entre los dos experimentos desarrollados . . . . .	75
7.2	Tabla comparativa de las funcionalidades que proveen las herramientas DVC y MLflow . . . . .	76



# Capítulo 1

## Introducción

La ciencia de datos y el aprendizaje máquina [5, 3] (*machine learning*, ML) se han convertido en herramientas indispensables para incrementar la competitividad y el rendimiento de organizaciones de todo tipo. Algunas de las actividades en las que se implementa son el conocimiento del cliente, el desarrollo de un producto o la optimización de procesos [7].

Según el Estudio de transformación digital de la empresa española [12], el 69% de las empresas españolas ya han emprendido procesos de transformación digital, mientras que el 31% aún no ha emprendido ningún tipo de transformación en este sentido. Ello puede ser debido a que un 25% presenta dificultades o falta de conocimientos para llevarlo a cabo. Aproximadamente el 50% de ejecutivos coinciden en que no se encuentran preparados para abordar el proceso, debido a su complejidad, falta de expertos o dificultad en la integración con los sistemas existentes.

En concreto, se estima que el impacto del Big Data en la empresa se encuentra en un porcentaje del 88%. Afectando especialmente a sectores como las telecomunicaciones, internet, banca, seguros, finanzas o alimentación. Por ello, se espera que la inversión realizada en el apartado tecnológico sea creciente en los años venideros.

Un estudio llevado a cabo por la Comisión Europea [10] revela que el 51% de las empresas españolas no tienen intención de implantar el uso de la inteligencia artificial (IA), mientras que el 42% de las empresas europeas implementan alguna herramienta que usa IA. Todo ello provoca una tasa de adopción en España inferior al 13%.

Otro dato interesante que arroja este estudio es que la implantación de IA en España en empresas medianas es de un 57%, mientras que pequeña empresa cae hasta el 30%.

Por estas razones aparece la Estrategia Nacional de Inteligencia Artificial impulsada por el Ministerio de Ciencia e Innovación [11]. Su objetivo consiste en trabajar en las políticas nacionales para impulsar el desarrollo y uso del IA, favoreciendo la colaboración privada y pública.

Otro estudio que investiga el estado del ML en la empresa [9] revela que entre 2012 y 2017 la demanda de puestos de técnicos de ML se disparó en un 650% en la plataforma LinkedIn.

Sin embargo, ese mismo estudio indica que en 2018 el 18% de las empresas encuestadas contrató a 11 expertos o más, mientras que tan solo el 2% de las empresas contrató a más de 1.000 científicos de ML. Esta cifra tan solo sube al 3% en el año 2020, lo que implica una mejora muy pequeña en comparación a la demanda de este tipo de empleos. Ese mismo estudio muestra que compañías de todos los tamaños invierten entre 8 y 90 días en desarrollar un único modelo de ML. Además, el 36% de las empresas encuestadas, indican que sus científicos gastan un 25% de su tiempo únicamente en desplegar el modelo, mientras que un 64% invierte más de la mitad, es decir, como mínimo se pierde una cuarta parte de los esfuerzos en tareas de infraestructura. Finalmente, el estudio concluye con el hecho de que únicamente un 22% de empresas encuestadas han sido capaces de desplegar correctamente el modelo de ML en producción.

Por tanto, de estos estudios se concluye que el desarrollo, y en especial el despliegue, de modelos de ML para la adopción de IA es un problema para el cual las empresas aún no se encuentran preparadas. Por ello, surge el concepto de operaciones de ML (MLOps), el cual proporciona principios y herramientas para llegar a cabo este proceso. Este TFG se centrará en utilizar estas técnicas y herramientas para el desarrollo y despliegue de modelos de ML.

En lo que resta de capítulo se realizará una revisión de las fases que componen un modelo de ML, se introducirá el concepto de MLOps, se hará hincapié en la importancia de llevar a cabo un correcto procesamiento de datos, se especificarán los objetivos del proyecto, la planificación temporal y la estructura de la presente memoria.

## 1.1 Machine learning en producción

El ciclo de vida de un modelo de ML se compone de tres partes: descubrimiento, desarrollo y despliegue. Estos pasos son igual de importantes y fallar en uno de ellos puede desencadenar errores en fases posteriores. En la figura 1.1 se esquematiza este proceso.

La fase de descubrimiento, marcada en azul en la imagen anterior, consiste en identificar el problema y determinar qué es lo que puede resolver el modelo de ML y qué no. En esta etapa es especialmente importante la colaboración entre los técnicos de ML y los expertos del ámbito empresarial [6, capítulo 8].

En ella se determinan las métricas e indicadores a tener en cuenta en el modelo. Es especialmente importante determinar los datos de los que se dispone, en concreto se deben responder preguntas como ¿cuál es la distribución de los datos?; ¿alguna de las entradas están correladas? o ¿disponemos de la suficiente cantidad de datos?.

La segunda etapa o desarrollo, marcada en amarillo en la figura 1.1, consiste por un lado en el procesamiento de los datos a fin de que estos sean válidos para el modelo que se pretende implementar, y por otro lado consiste en desarrollar el algoritmo de ML en el que se determinan los hiperparámetros, se exploran casos límite y se emplean técnicas de regularización. Por último, se valida el modelo desarrollado y se presenta a las partes interesadas de la compañía.

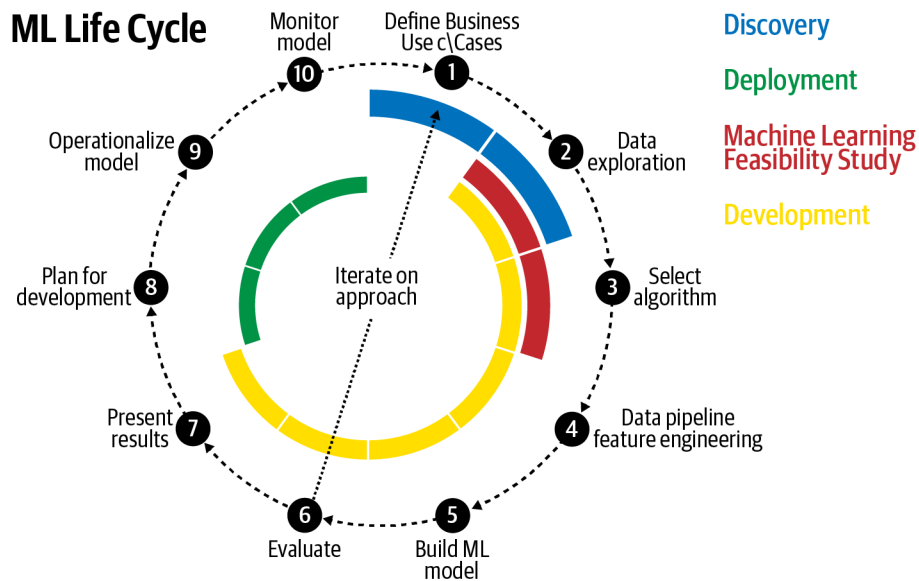


Figura 1.1: Ciclo de vida de un modelo de ML. Tomado de [6, figura 8-2].

La literatura ha obviado la fase de despliegue, marcada en verde en la figura 1.1. El paso del tiempo ha demostrado que esto es un gran error, lo que ha desembocado en el contexto explicado anteriormente. Previo al despliegue del modelo es importante tener en cuenta algunos factores, por ejemplo se deben resolver algunas de las siguientes preguntas: ¿deben tenerse en cuenta cuestiones especiales en cuanto al tiempo de respuesta para el usuario?; ¿cómo debe gestionarse el reentrenamiento del modelo? o ¿es la conectividad a la red un problema?.

La falta de profundización que la literatura realiza en esta fase se debe a la gran diferencia de objetivos entre la utilización del aprendizaje automático en el mundo académico y en las organizaciones. En el primer caso lo que se desea es entender y mejorar el conocimiento sobre esta materia, mientras que en el segundo, el objetivo principal es lograr el máximo beneficio posible.

Esta gran dificultad contrasta con la falta de atención comentada anteriormente, ya que es necesario integrar distintas partes de la organización tanto, técnicas como logísticas. Es por ello que este TFG se va a centrar en este punto y en las herramientas que pueden hacer el proceso más sencillo y automático.

El siguiente paso dentro de esta fase consiste en poner en funcionamiento el modelo, *operationalize*. Este apartado también es conocido por la industria como **MLOps**, que incluye aspectos como automatizar, supervisar, gestionar y mantener los modelos en producción. Se trata de un paso muy importante ya que permite que los modelos sean escalables y robustos en la organización [37].

El concepto de MLOps nace debido a la problemática desarrollada anteriormente y se puede definir como el conjunto de tecnologías y prácticas, presentes en todo el ciclo de vida del modelo de ML, utilizadas para gestionar de forma coherente y eficiente los modelos de ML con los demás elementos técnicos y no técnicos. El objetivo es lograr comercializar los productos y obtener el máximo potencial en el mercado.

Por último, es necesario llevar a cabo un proceso de monitorización con el fin de asegurar que el rendimiento obtenido es el esperado. Un claro ejemplo de la importancia de este paso son los cambios que puedan ocurrir en los datos de entrada, debido a variaciones en el comportamiento del cliente.

El origen de la problemática comentada en cuanto a la adopción de ML en la empresa es debida a que es substancialmente diferente el desarrollo de aplicaciones basadas en aprendizaje automático frente a software tradicional. En concreto, se deben tener en cuenta tres posibles niveles de cambio: datos, modelo de ML y código. Por tanto, un cambio en alguno de los anteriores aspectos puede desencadenar problemas en los demás. Sin embargo, la adopción de los principios de MLOps permite monitorizar esos cambios y actuar en consecuencia.

Algunas de los problemas más comunes que MLOps permite resolver, son los siguientes, aunque no son los únicos [13]:

- Una vez desplegado podemos observar que, con el paso del tiempo, el modelo comienza a comportarse de manera extraña, siendo necesario reentrenar con nuevos datos, si es que estos están disponibles.
- Tras examinar los datos disponibles, podríamos reconocer que es difícil obtener los datos necesarios para resolver el problema que habíamos definido previamente, por lo que habría que reformular el problema.
- A veces el objetivo por el que se planteó el problema puede cambiar durante el desarrollo del proyecto y decidimos cambiar el algoritmo de aprendizaje automático para entrenar el modelo.

### 1.1.1 Big Data y aprendizaje máquina en las organizaciones

La ciencia de datos es un campo que, a primera vista, puede parecer reciente, sin embargo se ha desarrollado durante años. Esta disciplina se puede definir como aquella que se encarga de analizar, procesar y almacenar conjuntos complejos de información, más conocidos como Big Data, ya bien por su tamaño o dificultad a la hora de interpretarlos. Sobre todo, entra en escena cuando las técnicas de procesamiento habituales no surten efecto y, por tanto, es necesario buscar nuevas alternativas [8, capítulo 3].

La ciencia de datos combina matemáticas, estadística, informática y la experiencia para llevar a cabo algunas de estas tareas: identificación de nuevos mercados, predicciones más precisas, optimización en el proceso de aprendizaje máquina y mejora en la toma de decisiones, entre otros.

Para obtener buenos resultados en la tarea que queramos resolver utilizando ML, es necesario alimentar nuestros modelos con datos cuya relevancia y estructura sea la adecuada, lo cual se consigue mediante las técnicas de ciencia de datos. Mediante la aplicación de las mismas las empresas persiguen alguno de los siguientes objetivos:

- Enriquecer sus datos internos con conocimiento externo y conocer su posición en el mercado global. De esta forma la compañía no infiere información del mercado, sino, que directamente percibe el mercado.
- Convertir datos en información para poder generar conocimiento. Esto hace que se respondan preguntas del tipo: qué está pasando en el negocio, cómo de bueno es el funcionamiento de la empresa y por qué la empresa trabaja en el nivel de mercado en el que se encuentra.
- La compañía puede adaptarse a los cambios en las necesidades del entorno haciendo uso del Big Data y los sistemas de gestión de procesos empresariales (BPMS en inglés). De esta forma, con las herramientas adecuadas, de los datos se puede obtener previsión en el mercado.

En concreto, para aplicar las técnicas de ciencias de datos a un conjunto sin estructura y en bruto, es necesario seguir los pasos se detallados en la figura 1.2.

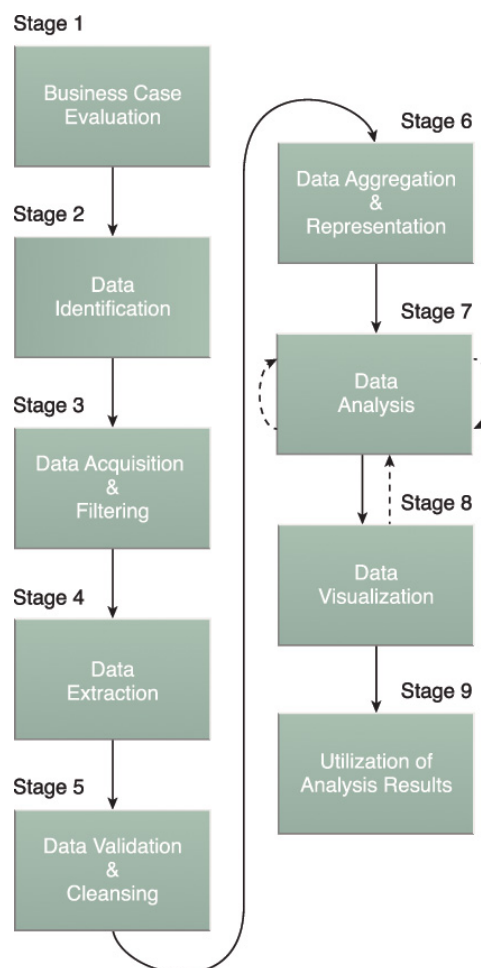


Figura 1.2: Fases en la planificación de ciencia de datos. Los distintos pasos representados se detallan en el texto principal. Tomado de [8, figura 3.6].

1. La etapa de **evaluación del caso de negocio** trata de definir y entender correctamente la motivación y objetivos de la problemática en cuestión. En esta fase, es importante identificar si

el caso a tratar es realmente un problema de Big Data ya que, de ser así, debe estar relacionado con volumen, velocidad o variedad en los datos.

2. La fase de **identificación de datos** consiste en identificar los conjuntos de datos (*datasets*) necesarios y sus fuentes. Según el propósito que se persiga los *datasets* a utilizar pueden ser internos o externos a la compañía.
3. La etapa de **adquisición y filtrado de datos** consiste en tomar los datos de las fuentes identificadas en la etapa anterior, para después filtrar y eliminar los datos corruptos o que carezcan de valor para el objetivo perseguido.
4. La fase de **extracción de datos** se encarga de extraer datos muy dispares y transformarlos en un formato único, especialmente cuando los datos obtenidos sean externos.
5. La **validación y limpieza de datos** consiste en establecer reglas para realizar su validación y completar aquellos datos incompletos. Por ejemplo, es típico tener *datasets* redundantes, lo cual puede ser aprovechado para completar campos vacíos o para validar ciertos parámetros.
6. La etapa de **agregación y representación de datos** consta, principalmente, de dos tareas: obtener los datos en un formato único e integrar los diversos *datasets*, de manera que se consiga una perspectiva única. No es de extrañar que *datasets* distintos se compongan de campos repetidos, referenciados mediante nombres diferentes. Por ello, es necesario integrarlos bajo un identificador y formato común.
7. En la fase de **análisis de datos** se intentan identificar patrones que pueden no ser evidentes en dichos datos, generalmente de forma iterativa. En concreto, existen dos aproximaciones. Por un lado, el análisis de datos confirmados postula una hipótesis de antemano y la prueba o refuta según el resultado del estudio. Por otro lado, el análisis exploratorio no plantea hipótesis alguna y tan solo busca entender la causa que ha generado esos datos.
8. La **visualización de datos** consiste en la comunicación gráfica de los resultados del análisis de la etapa anterior. El objetivo es que los trabajadores de la empresa puedan interpretar correctamente los resultados que se desprenden del análisis.
9. La etapa final, o etapa de **utilización de los resultados** del análisis, consiste en determinar cómo y dónde se pueden aprovechar mejor los resultados alcanzados y así optimizar su explotación.

## 1.2 Objetivos del proyecto

Tal y como se ha indicado en la sección anterior, en este proyecto se analiza la situación actual de la gestión del ciclo de vida de proyectos de ML en entornos empresariales. Además, se proponen soluciones enfocadas, en particular, a la implementación y gestión de la etapa de despliegue de los modelos de aprendizaje máquina desarrollados.



- **Objetivos generales (OG):** proponer una primera aproximación teórica para el despliegue de proyectos de ML en producción en el ámbito empresarial. Para ello:
  - OG1: identificar y analizar las diferentes fases del ciclo de vida de proyectos de aprendizaje automático en producción y detallar las tareas a realizar en cada fase.
  - OG2: detallar las funciones a implementar por cada elemento del flujo de trabajo que crea y despliegue modelos de manera automática.
  - OG3: encontrar herramientas software que permitan dar soporte a las principales funciones y tareas dentro de este ciclo de vida y flujo de trabajo, ubicándolas dentro del flujo de trabajo.
  - OG4: localizar funciones y tareas cuyo soporte todavía no esté respaldado por proyectos o soluciones específicas.
- **Objetivos específicos (OE):** ilustrar mediante dos casos prácticos la implementación de los principios de **MLOps** a lo largo de todo el ciclo de vida de un modelo de ML, ilustrado en la figura 1.1. Para ello, se han utilizado algunas de las herramientas detalladas en el capítulo 2. En concreto:
  - OE1: diseñar e implementar un flujo de trabajo de ejemplo, que muestra el desarrollo de un proyecto de aprendizaje automático en un entorno de un laboratorio de pruebas.
  - OE2: diseñar e implementar un flujo de trabajo, que ilustre un caso de paso a producción de un proyecto de aprendizaje automático desde una fase de laboratorio.

## 1.3 Planificación temporal

La planificación temporal realizada en este proyecto se detalla en la figura 1.3. Aproximadamente, se han invertido unas 800 horas en la realización de este TFG. Esto es debido, por un lado, a haber comenzado pronto tanto a realizarlo como a redactar la memoria y, por otro lado, al deseo de la alumna de entregar un proyecto elaborado. Adicionalmente, dada la naturaleza del proyecto, las tareas de revisión del estado del arte e implementación han requerido una dedicación alta.

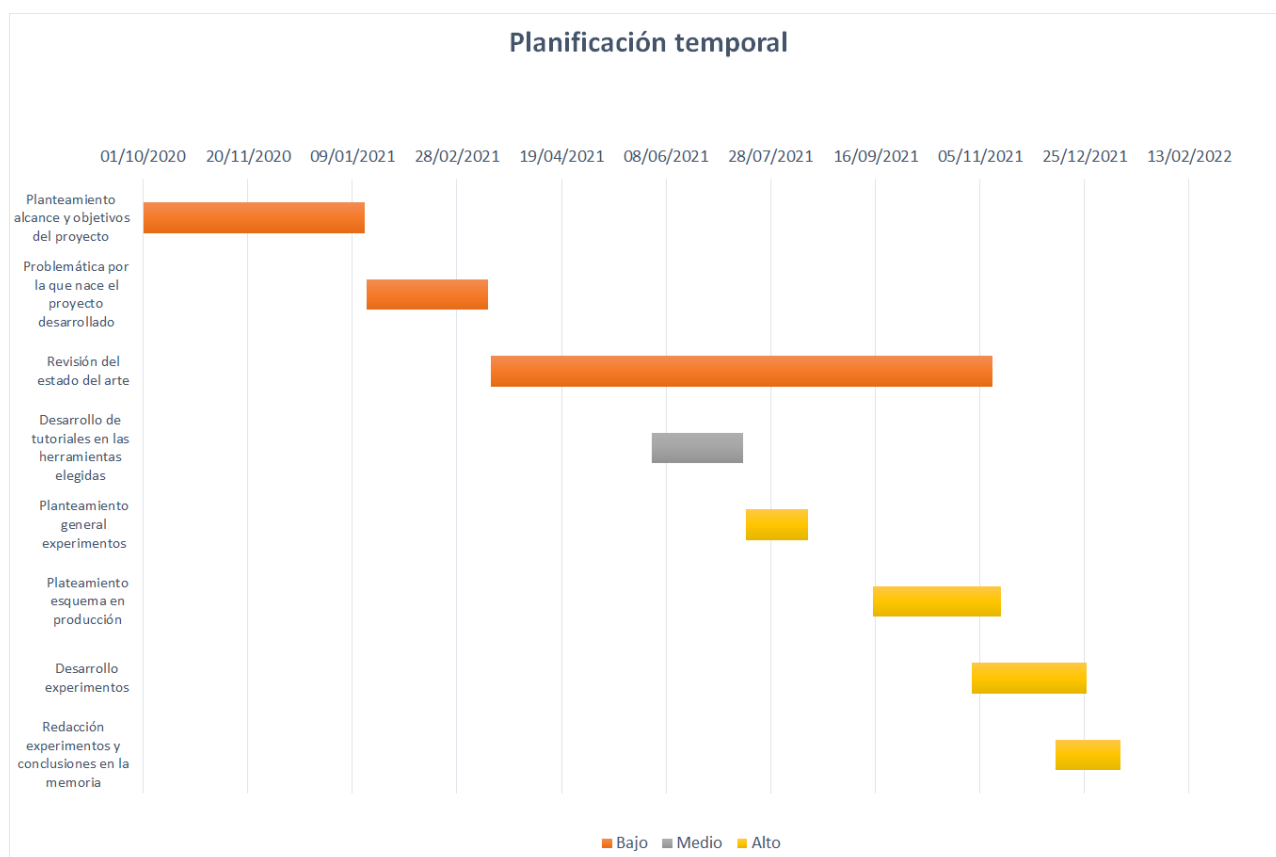


Figura 1.3: Diagrama de Gantt que ilustra la planificación temporal llevada a cabo en este proyecto. En el eje horizontal se representan las fechas en las cuales se ha desarrollado y en el eje vertical se representan las tareas llevadas a cabo. Asimismo se ilustra el nivel de esfuerzo desempeñado en cada tarea: bajo (menos de 0.5h/día de media, en color naranja), medio (3h/día de media, en color gris) y alto (6h/día de media, en color amarillo)

## 1.4 Estructura de la memoria

La estructura del presente proyecto se organiza de la siguiente forma:

- En el capítulo 1 se ha realizado una contextualización del problema que se presenta en muchas organizaciones, a la hora de trasladar un modelo de ML de un entorno de pruebas a producción. También se han detallado las fases de las que se compone el ciclo de vida de un modelo de ML y se hace hincapié en la importancia de llevar a cabo un correcto análisis de los datos y de los pasos necesarios a seguir.
- A continuación, se presenta el estado del arte en el capítulo 2. Se realiza una definición, análisis y objetivos planteados por **MLOps**. Concretamente, se explica una visión general de las tres herramientas en las cuales se centra este proyecto, Data Version Control, MLflow y Apache Airflow. También se presentan otras herramientas relacionadas con los principios de MLOps.

- En el capítulo 3 se desarrolla el esquema teórico junto con todos los elementos necesarios para afrontar el proceso de creación y despliegue de modelos de ML en el ámbito empresarial. En concreto, se repasan los diferentes escenarios existentes según el grado de implementación de MLOps y se realizan varias propuestas conceptuales para llevar a cabo todo el proceso de manera automatizada.
- En el capítulo 4 se explica el origen y procesamiento de los datos elegidos y se presenta el planteamiento general utilizado en los dos casos prácticos desarrollados en este proyecto. En particular, se plantea la estructura común a ambos y se detallan las dimensiones o criterios objetivos con los que realizar un análisis de resultados.
- En el capítulo 5 se analiza el primer caso práctico. Se plantea un diseño teórico general, a continuación se implementa dicho diseño, y, por último, se evalúan los resultados en función de los criterios planteados en el capítulo anterior.
- Análogamente, en el capítulo 6 se lleva a cabo el diseño, implementación y exposición de resultados llevados a cabo en el segundo caso práctico.
- Por último, en el capítulo 7, se ofrecen unas conclusiones generales de los experimentos desarrollados, así como del estudio de la situación de MLOps en el ámbito empresarial. Como complemento, se detallan los conocimientos adquiridos en este proyecto, se evalúa el grado de consecución de los objetivos iniciales planteados en el capítulo 1 y se esbozan líneas de trabajo futuras.

Adicionalmente, se pueden consultar los siguientes recursos en líneas asociados a este proyecto:

- Página web del proyecto: <https://meridiaz.github.io/MLOps-Evaluation/>.
  - Primer caso práctico: <https://meridiaz.github.io/MLOps-Evaluation/exp1.html>.
  - Segundo caso práctico: <https://meridiaz.github.io/MLOps-Evaluation/exp2.html>.
- Repositorio de código: <https://github.com/meridiaz/MLOps-Evaluation>.



## Capítulo 2

### Estado del arte

Este proyecto aborda un área de conocimiento emergente y de gran interés. Por ello, es necesario hacer una revisión del estado del arte de la disciplina de **MLOps**. A tal fin, este capítulo se encuentra estructurado de la siguiente forma: en primer lugar, se realizará una definición y especificación de los objetivos que persigue MLOps; en segundo y último lugar, se explica el alcance y funciones de algunas de las herramientas más relevantes en este campo.

En el capítulo 1 se ha ofrecido una pequeña introducción a la problemática por la cual aparece MLOps. En la sección 1.1 se define este término, que tiene su origen en operaciones de desarrollo, *development operations* (DevOps). Esta disciplina define una metodología para lograr integración y colaboración entre desarrolladores de software y administradores de sistemas. Con ello se logra desarrollar software de menor coste, mayor calidad y con una mayor automatización [17].

Los creadores de MLOps aportan la siguiente definición: *“MLOps must be language-, framework-, platform-, and infrastructure-agnostic practice. MLOps should follow a “convention over configuration” implementation”... We need to establish best practices and tools to test, deploy, manage, and monitor ML models in real-world production. In short, with MLOps we strive to avoid “technical debt” in machine learning applications”* [37]. Por tanto, se trata de un conjunto de herramientas y principios que se encuentran en las tres fases de desarrollo de un modelo de ML: diseño, desarrollo del modelo y despliegue. Su objetivo final es integrar los equipos de ciencia de datos e ingenieros de ML con los administradores del sistema o equipos de operación para agilizar, automatizar y mejorar su funcionamiento en producción.

Es importante destacar que existe discrepancia entre las definiciones de MLOps para la industria y la academia. En la sección 1.1 se introduce una breve definición del concepto y se indica que aparece en la última etapa del ciclo de vida del modelo, el despliegue. Esta es la definición que aporta la industria sobre MLOps. Sin embargo, en el párrafo anterior se explica lo que la literatura entiende por MLOps. En este caso, se hace hincapié en que MLOps aparece en todo el ciclo de vida del modelo pues, debido a que es un proceso cíclico, como muestra la figura 1.1, todas las etapas influyen en la fase de despliegue.

Tal y como se ha detallado en el capítulo anterior, el despliegue de los modelos de ML supone un reto importante para las empresas debido a la gran cantidad de recursos que se invierte en ello. Por esta razón, el uso de MLOps resulta decisivo para facilitar el despliegue y mantenimiento automatizando de las partes más complicadas [1]. Por todo ello, debe incluir herramientas que cumplan los siguiente principios [37]:

- Los datos, modelos ML y código han de ser gestionados, mediante un sistema de control de versiones.
- Se deben encapsular los modelos de ML, para facilitar su despliegue y su puesta en marcha en otros entornos.
- Deben crearse *pipelines* que aseguren la integración continua y entrega continua, *Continuous Integration and Continuous Delivery/Deployment (CI/CD)*.

De manera genérica, dentro del área de la ingeniería de software un *pipeline* consiste en una secuencia de tareas que se ejecutan de forma automatizada y secuencial. En el campo de MLOps esta definición se amplía, debiendo, además, incluir tareas de construcción y despliegue de forma eficiente y fiable [24]. En el capítulo 3 se realiza una explicación más detallada y aplicada.

- Es preciso que automáticamente se generen experimentos, debiendo ser estos reproducibles en cualquier entorno, y que se desplieguen modelos de ML.
- Se ha de evaluar el desempeño de los modelos, de forma previa a su despliegue.
- Se debe monitorizar los modelos desplegados (*continuous monitoring*, CM).
- Los modelos deben ser almacenados en un lugar centralizado y ser identificados correctamente.

En las restantes secciones se introducirán las herramientas que hacen posible la aplicación de estos principios.

## 2.1 Data Version Control

Data Version Control (DVC) es una herramienta de software libre que se centra en la fase más experimental del ciclo de vida de un modelo [33]. Algunas de las tareas que permite llevar a cabo son:

- Control simultáneo de versiones de datos y código del modelo, permitiendo cambiar fácilmente entre ellas. Para llevar a cabo esta tarea, DVC funciona conjuntamente con git, de manera que DVC versiona los modelos y grandes volúmenes de datos, mientras que en git tan solo se almacenan unos ficheros con metadatos y el código del modelo de ML en cuestión. En la figura 2.1 se esquematiza esta función:

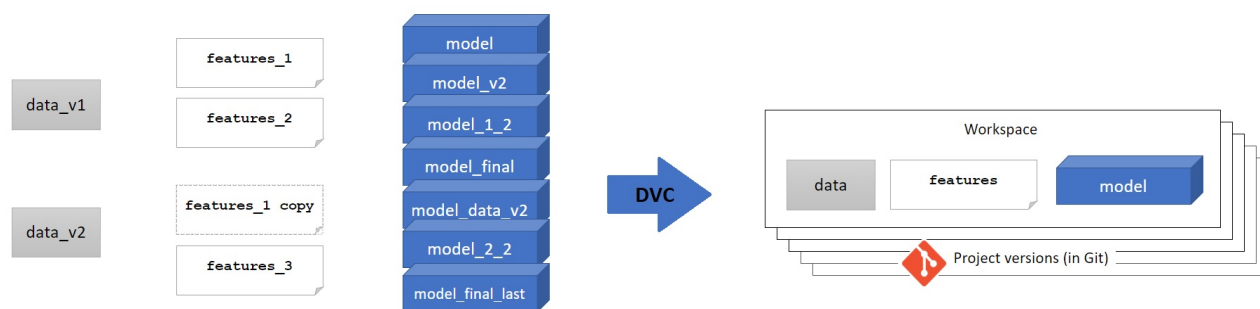


Figura 2.1: Cambio de enfoque de datos y modelos sin versionar a un escenario donde todo está organizado y versionado por git y DVC. A la izquierda complejidad exponencial, a la derecha historial de código, datos y modelo realizado por DVC. Adaptado de [33].

- Compartir grandes volúmenes de datos o código, haciendo uso de un almacenamiento en la nube como Amazon S3, perteneciente a Amazon Web Services (AWS), Google Drive o Google Cloud Storage, entre otros. Esto permite separar los datos del código propiamente dicho.
- Obtención de datos de diferentes repositorios DVC, lo que se conoce como *Data registry*. Este concepto permite centralizar en una sola ubicación los datos, lo cual confiere mayor seguridad y control sobre los cambios y accesos que tengan lugar en ellos.
- Creación de *pipelines*, con el fin de automatizar y optimizar el proceso de obtención de características, entrenamiento y validación. Tal y como se ha explicado en la sección 1.1.1, es necesario transformar y filtrar los datos antes de ser alimentados al modelo, por lo que es recomendable incluir una tarea que realice esta función.

Además, para optimizar los periodos de ejecución, DVC únicamente ejecuta aquellas tareas del *pipeline* que hayan sufrido cambios en alguna de sus dependencias como parámetros, elementos de entrada o código asociado a ellas, lo que permite ahorrar recursos.

- Creación de servidores de uso compartido. En el mundo de la ciencia de datos es típico el uso de servidores compartidos que ejecuten los experimentos, lo que permite mejorar la reutilización de recursos, como el acceso a la unidad de procesamiento gráfico (*graphical processing unit*, GPU). DVC permite cambiar fácilmente el espacio de trabajo entre los diferentes usuarios.
- Gestión y planificación de experimentos, de manera que no es necesario almacenar todos ellos en git, sino tan solo aquellos que decidamos compartir. DVC permite cambiar diversos parámetros del modelo fácilmente, de forma que, una vez ejecutados, muestra una comparativa entre ellos para almacenar únicamente aquellos que se desee.
- Análisis de métricas del modelo de los diferentes experimentos que se hayan ejecutado. DVC genera de forma automática comparativas entre las métricas y parámetros de los experimentos.

## 2.2 MLflow

Otra herramienta de software libre es MLflow, la cual permite llevar a cabo el seguimiento y empaquetamiento de experimentos, con el fin de hacerlos reproducibles en otros entornos y almacenar, versionar, empaquetar y desplegar a producción los modelos de ML. El objetivo de esta herramienta es el siguiente: “*MLflow aims to take any codebase written in its format and make it reproducible and reusable by multiple data scientists*” [35].

MLflow se subdivide en distintos módulos según el problema a abordar [35, 48, 22].

- **MLflow Tracking** realiza un seguimiento de las ejecuciones o *runs* llevadas a cabo. El objetivo es realizar un seguimiento de métricas, modelos, parámetros, gráficas y ficheros asociados esas *runs*. Además, permite agrupar esas ejecuciones en categorías para mejorar la organización del área de trabajo. Todo ello puede ser visualizado gráficamente mediante la UI que proporciona.

Puesto que almacenar los ficheros de datos y modelos puede requerir una gran capacidad de memoria, MLflow permite configurar un almacenamiento remoto en servicios como Amazon S3, Azure Blob Storage, Google Cloud Storage, FTP server, SFTP Server, NFS y HDFS.

- **MLflow Projects** empaqueta el código fuente del proyecto, de forma que sea reutilizable y reproducible para compartirlo con otros científicos de datos o transferirlo a producción. Un proyecto de MLflow no es más que un fichero YAML, llamado `MLproject`, que especifica los puntos de entrada del proyecto (si los tiene) y el nombre de los ficheros que contienen las dependencias (por ejemplo el fichero que contiene dependencias de Anaconda, es decir, los paquetes necesarios de Python para ejecutar el proyecto). En este fichero puede definirse un punto de entrada llamado “`main`”, que se ejecute por defecto si no se especifica lo contrario. En el mismo, se puede configurar un *script* para que ejecute ordenadamente cada punto de entrada, pasándole los argumentos especificados, con lo que se lograría la implementación de un *pipeline*.

Gracias a esto, se pueden compartir los experimentos de ML con otros desarrolladores y ejecutarlos en otro entorno, bajo las mismas condiciones en las que fueron creados.

- **MLflow Models** permite gestionar y desplegar modelos de un conjunto de librerías de ML (llamadas *flavours*), como pueden ser scikit-learn, TensorFlow, Keras, PyTorch, y PySpark, en diversas plataformas de servicio de modelos, por ejemplo AWS Sagemaker, AzureML o incluso localmente. De esta forma, los datos son enviados a la plataforma para que el modelo realice la predicción.

Al registrar un modelo MLflow también se crean, entre otros, dos ficheros, `MLmodel` y `model.pkl`. El primero contiene todos los metadatos necesarios para desplegarlo, mientras que el segundo incluye una versión serializada del modelo que se ha entrenado.



Asimismo, este módulo permite definir lo que se conoce como el *model signature*. Se trata de un esquema que define cómo deben ser las entradas y salidas del modelo. De esta forma, una vez que se despliega se comprueba si los datos de entrada proporcionados son del tipo esperado y tienen el nombre correcto al compararlos con el *signature* del modelo.

Por tanto, MLflow Models permite abstraerse de la biblioteca en la que se implementa el modelo para poder desplegarlo en cualquier plataforma. Las opciones que provee MLflow a la hora de desplegar modelos son a través de una API REST local, o bien mediante función en Python para las plataformas Microsoft Azure ML, Amazon SageMaker o Apache Spark UDF.

- **MLflow Model Registry** proporciona un almacén central de modelos (*model store* o *model registry*), para gestionar de forma colaborativa el ciclo de vida completo de un modelo de ML, de forma que se facilita la cooperación entre científicos de datos y administradores del sistema.

Algunas características que este módulo permite visualizar son, entre otras: la línea cronológica de versiones y experimentos que han producido determinado modelo, detalle del *stage* en que se encuentra, ya bien en fase de pruebas o directamente en producción, y controlar el acceso a la plataforma de registros.

MLflow Model Registry es muy similar al anterior, MLflow Models, en lo que respecta al registro del modelo. Sin embargo, una diferencia clave es el que Model Registry permite visualizar en un lugar centralizado todos los modelos disponibles junto con su versión y *stage*.

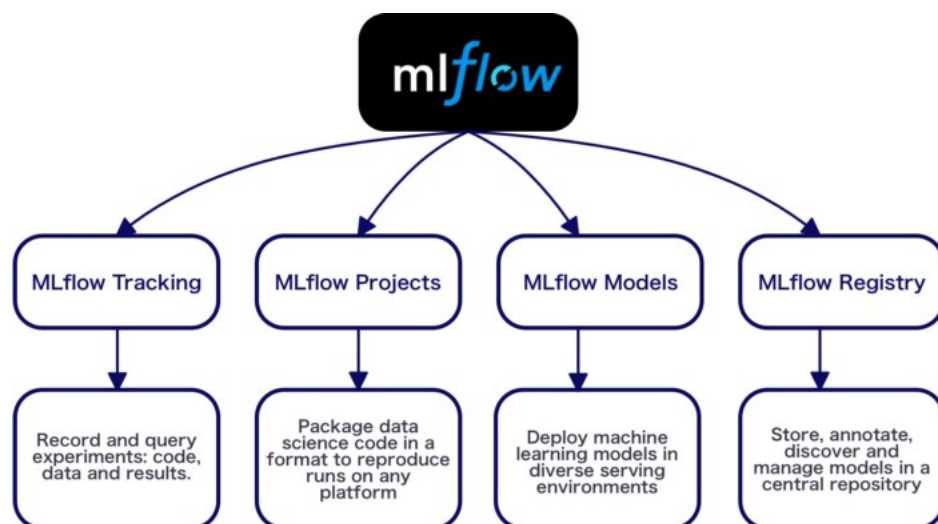


Figura 2.2: Esquema de los módulos y funcionalidad que abarca de MLflow. Tomado de [22].

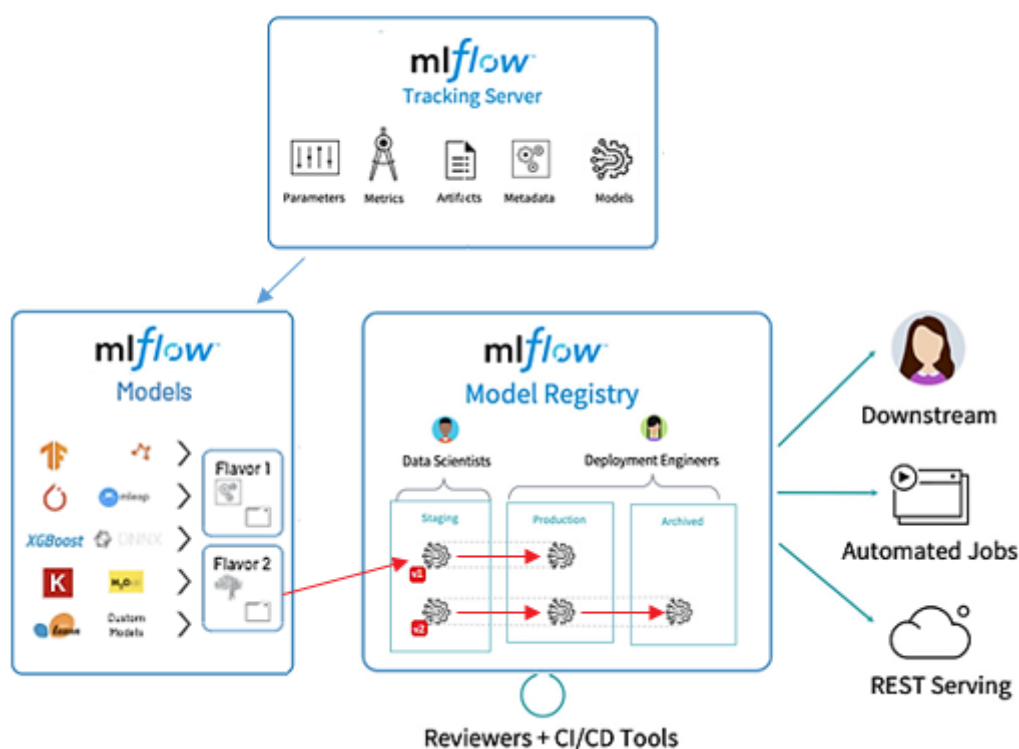


Figura 2.3: Esquema de la relación entre los distintos módulos de MLflow. MLflow Tracking se encarga de realizar un seguimiento en cada ejecución de métricas, parámetros o artefactos, los cuales pueden englobar gráficas o modelos. A continuación, tenemos MLflow Models que se encarga de gestionar y desplegar los modelos desde distintos *flavors* como Keras a diversas plataformas. Y por último, MLflow Model Registry en el que se realiza un seguimiento del versionado de cada modelo y de su *stage*. Todo ello permite desplegar fácilmente los modelos y automatizar tareas. Tomado de [14, 23].

## 2.3 Apache Airflow

Por otro lado, si lo que se busca es la automatización en la programación de flujos de trabajo, es necesario hablar de la herramienta Apache Airflow [28]. De manera general, se puede definir como un software libre orquestador, esto es, una herramienta que indica en qué momento y cómo se debe ejecutar una tarea.

Airflow utiliza un grafo acíclico dirigido (*directed acyclic graph*, DAG), que define una secuencia de tareas, de manera que nos permite abstraer el código de esas tareas que se llevan a cabo. Es decir, su trabajo consiste en orquestar la ejecución de tareas respetando las dependencias existentes entre ellas. Así, supongamos que queremos ejecutar tres tareas A, B y C. El DAG nos permite configurar cómo queremos ejecutarlas, por ejemplo, deseamos que B se lance si la ejecución de A no ha fallado y que C se lance 5 minutos después de la ejecución de B. El concepto clave es que el DAG no se preocupa de lo que hacen internamente las tareas que lo componen. Su trabajo es asegurarse de que sucedan en el momento adecuado, en el orden correcto, o con el manejo correcto de cualquier

problema inesperado. De este modo, se puede cambiar fácilmente el código que se desea ejecutar en cada tarea sin cambiar la estructura del DAG.

A su vez, cada tarea engloba a un operador, que se encarga de ejecutar el código que define la tarea, mientras que en sí la tarea se encarga de que el trabajo se haya realizado correctamente. En la figura 2.4 se realiza un pequeño resumen de estos conceptos.

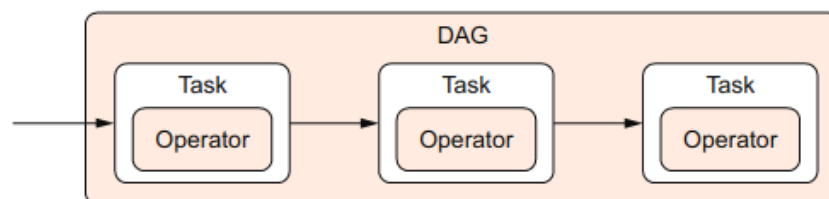


Figura 2.4: Esquema de los conceptos clave de Apache Airflow. El DAG se compone de tareas, mientras que las tareas están formadas por operadores que ejecutan el código. Tomado de [4, figura 2.4].

Airflow puede ser utilizado, entre otros, en los siguientes casos [20, 27]:

- Programar una secuencia de tareas, donde cada una de ellas es dependiente de las anteriores.
- Programar los DAG para que se ejecute en fechas o intervalos de tiempo concretos, de manera automática.

Airflow usa lo que se conoce como *interval-based scheduling windows*, esto es, ejecuta la tarea correspondiente tan pronto como el intervalo de tiempo para el que está definida se encuentre en el pasado. Por ejemplo, si el DAG está configurado para ejecutarse cada hora, el slot asociado a las 19h se ejecutará en cualquier momento pasadas las 19:59h. Esta aproximación permite afrontar el problema de procesamiento de grandes volúmenes de datos de manera incremental, como es el caso de datos en *streaming*.

- Dispone de una interfaz gráfica, UI, que permite gestionar y obtener una visión general de todas las operaciones que se están llevando a cabo, así como de sus configuraciones.
- La función X-COM permite compartir información entre diferentes tareas, usando una base de datos propia, de manera que es posible comunicar las tareas entre sí. Sin embargo, este tipo de comunicación no es la recomendada pues, por diseño, las tareas deben estar aisladas y ser autocontenidas.

Para llevar a cabo estas tareas, Airflow se compone de tres partes, las cuales se representan en la figura 2.5:

- Organizador de tareas (*scheduler*): este es el corazón de Airflow, que se encarga de leer los ficheros en los que se definen los DAGs, comprueba cuándo deben ser ejecutados y encola las tareas teniendo en cuenta las dependencias entre ellas.
- Ejecutor de tareas (*worker*): lleva a cabo todas las tareas que se encuentran en la cola.

- Servidor web de Airflow (*webserver*): muestra los resultados y el flujo de trabajo de manera gráfica al usuario.

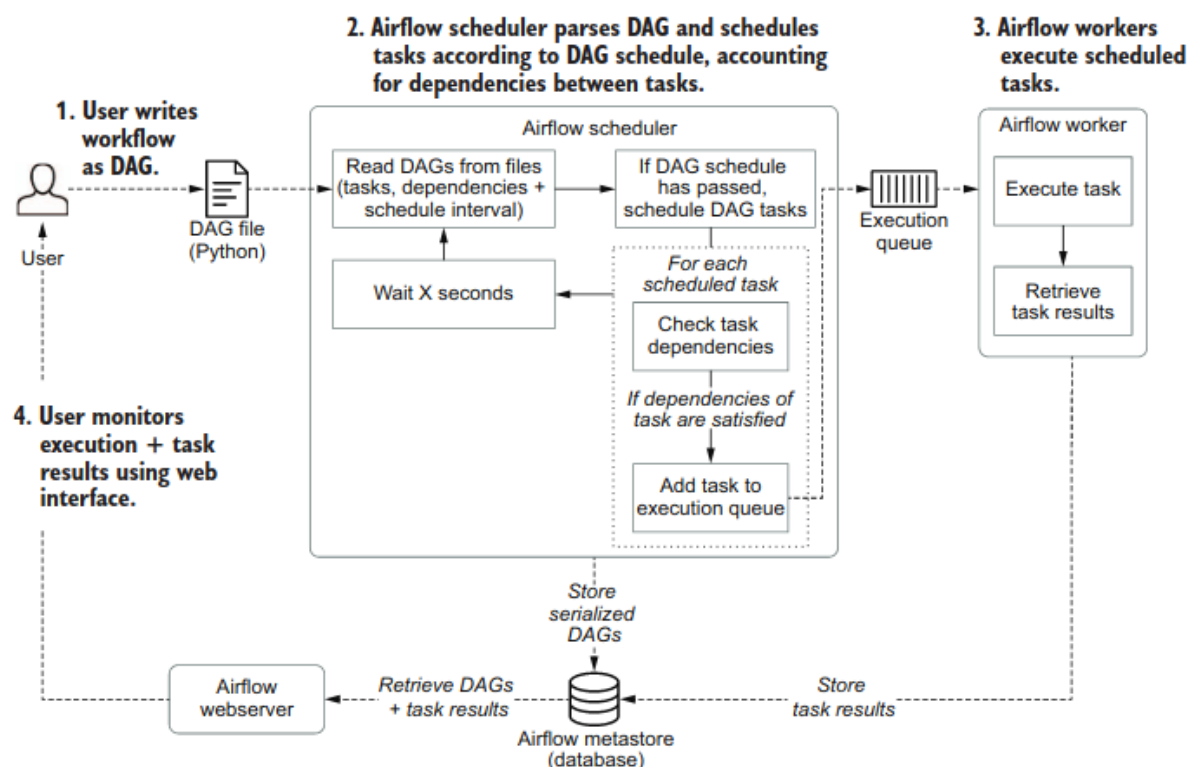


Figura 2.5: Estructura de Apache Airflow. El usuario escribe un fichero de código, normalmente en Python, el organizador de tareas lo lee y encola las tareas que se deban llevar a cabo, el ejecutor recoge esas tareas de la cola y las ejecuta, y, por último, se visualiza el resultado en el servidor web que proporciona Airflow. Tomado de [4, figura 1.9].

## 2.4 DataRobot

Se trata de una herramienta propietaria totalmente gráfica y automatizada. Integra todo el proceso reflejado en la figura 3.2. En concreto, se encarga del procesamiento, entrenamiento y selección de modelos, despliegue automático y monitorización en una única herramienta [31]. En la figura 2.6 se muestra todo su alcance.

En primer lugar DataRobot pide importar los datos a usar, puede ser de una base de datos, de un fichero local o de una URL. A continuación, mediante una interfaz gráfica, permite visualizar y analizar los datos, a través de lo que denominan *Exploratory Data Analysis* o EDA. Esta función permite categorizar, aplicar transformaciones o crear conjuntos de características para usarlos en los siguientes pasos. Además, EDA también indica la calidad de los datos y muestra problemas típicos, como es la presencia de *outliers*.

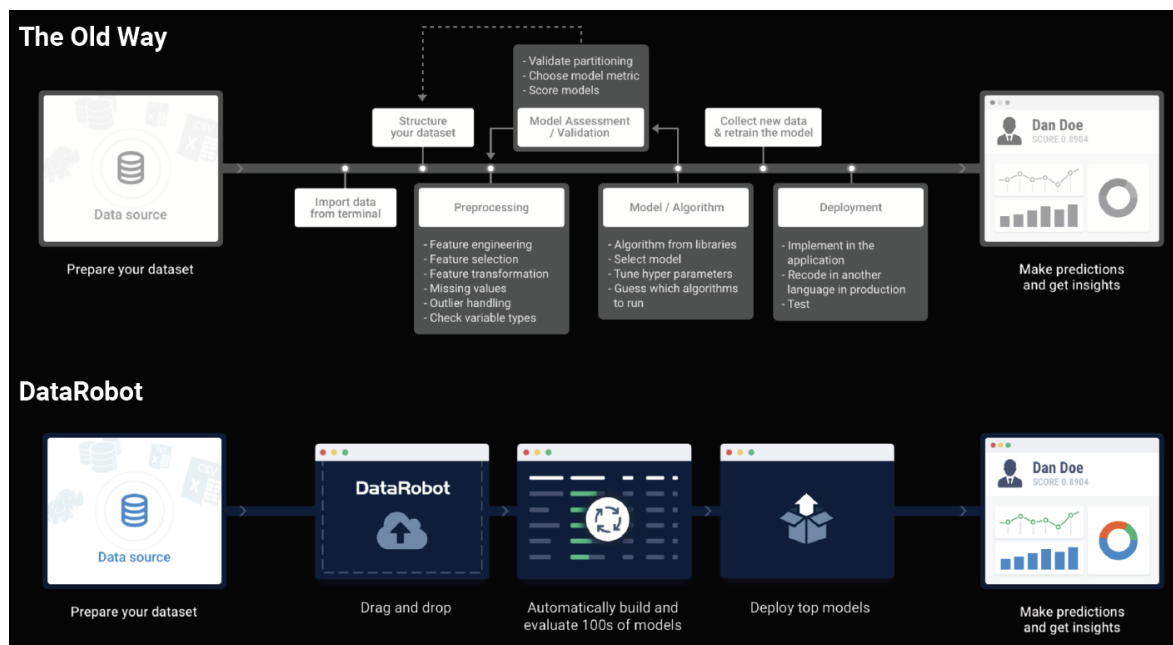


Figura 2.6: Esquema de DataRobot. Esta herramienta automatiza de manera gráfica todo el ciclo de vida de un modelo de ML (véase la figura 1.1), desde procesado de datos, creación del modelo y despliegue hasta la monitorización en tiempo real. Tomado de [15].

En segundo lugar, DataRobot pregunta al usuario por la característica objetivo de la predicción del modelo. Una vez seleccionada, esta herramienta elige de forma automática el modelo que mejor se ajusta a la variable seleccionada y divide los datos para las posteriores pruebas. A continuación, comienza a entrenar y muestra varias estadísticas sobre la variable objetivo, como la correlación con las demás características. Por último, almacena los modelos creados en la sección “models”, y elige los mejores para llevar a cabo el despliegue a producción.

La sección “models” también incluye la posibilidad de analizar en mayor profundidad el modelo y características seleccionadas. Para ello muestra algunas estadísticas, por ejemplo, indica las características que mayor impacto tienen en la predicción, los parámetros elegidos para la creación del modelo y las predicciones, junto con su distribución y una explicación de los resultados.

En tercer lugar, DataRobot permite compartir las predicciones y crear una aplicación con el mejor modelo seleccionado. Primeramente, DataRobot muestra las predicciones realizadas para entender mejor el resultado obtenido. A continuación, lanza varias simulaciones para optimizar los parámetros del modelo y obtener las mejores métricas. Y, por último, ofrece la posibilidad de compartir el resultado mediante una API y gestionar el acceso a este.

En cuarto lugar, DataRobot permite transferir los modelos a un *model registry* con el fin de prepararlo para el despliegue. Este *model registry* permite visualizar las métricas del modelo desplegado, los participantes en el proceso y la fecha de su creación. A continuación, muestra un seguimiento

del número de modelos desplegados, predicciones realizadas y la calidad de estas, permite identificar *data drifts* y por último ejecuta predicciones en segundo plano haciendo que el modelo desplegado pueda ser sustituido automáticamente si se encuentra uno mejor.

Por último, Datarobot permite visualizar en tiempo real las tendencias de los nuevos datos, así como sus métricas, para llevar a cabo un reemplazo del modelo en caso necesario. Con el fin de permitir un mejor seguimiento, proporciona alertas sobre *data drifts* o predicciones que se encuentran fuera de rango, entre otros. Todo ello permite la aplicación de los principios de **MLOps** directamente sobre el proyecto.

La ventaja más clara que ofrece esta herramienta es que brinda una gran flexibilidad a la hora de crear los modelos. Se puede utilizar como un asistente que, en unos pocos pasos, crea, entrena y despliega un modelo, o bien, como una herramienta para profundizar en los distintos parámetros del proceso, consiguiendo modelos muy precisos.

Sin embargo, su desventaja más clara es la falta de interoperabilidad. Al tratarse de software propietario, el cliente no puede extraer los productos acabados como, por ejemplo, los modelos de ML ya entrenados, para migrarlos a otra herramienta. Ello ocasiona que ante cualquier situación adversa que se dé, como una subida en el coste de la licencia, el cliente deba asumirlo con el fin de no perder todo el trabajo llevado a cabo.

## 2.5 OpenML

Es una herramienta de software libre que permite organizar datos, modelos y experimentos, para compartirlos con otros usuarios de la plataforma. En OpenML se distinguen varios conceptos [40]:

- **Datasets:** tablas de valores, donde las columnas se denominan características o covariables y cuyos datos deben ser numéricos o de tipo String. OpenML automáticamente analiza los datos, comprueba posibles problemas, permite visualizarlos y provee diversas estadísticas sobre ellos.

Cada dataset está definido por un identificador único y un estado, que indica si el dataset ya ha sido aprobado por el administrador, puede usarse o se encuentra desactivado por problemas encontrados en él.

- **Tareas:** engloban el conjunto de datasets, tipo de tarea de ML a resolver (como puede ser clasificación o *clustering*), la variable objetivo a predecir, número de muestras con las que validar el modelo y una forma de evaluación.

Las tareas son legibles por la máquina, y, por tanto, pueden ser interpretadas directamente, de manera que el programador se centra en obtener el mejor algoritmo posible para resolver esa tarea. A su vez, OpenML se encarga de organizar los resultados, evaluarlos y versionar las

distintas soluciones proporcionadas por los usuarios de la plataforma para resolver esa tarea específica.

- **Flows:** son algoritmos de librerías concretas, como `scikit-learn`, que resuelven tareas. OpenML organiza estos *flows* de manera que muestra el tipo de tarea que resuelve (por ejemplo clasificación), los hiperparámetros que se pueden configurar, discusiones de los usuarios y una wiki indicando, entre otras cosas, en qué datasets puede ser usado. Por ejemplo, el algoritmo Random Forest de la librería Weka es un *flow*.
- **Runs o experimentos:** es la aplicación de un *flow* a una tarea concreta sobre un dataset determinado. De manera que en la página de cada tarea se puede ver el dataset que le corresponde, los distintos *runs* ejecutados junto al *flow* usado y las métricas obtenidas. Por ejemplo, un *run* sería utilizar el algoritmo o *flow* Random Forest de la librería Weka, para tratar de predecir el tipo de flor a partir de unas características físicas.

En el futuro, esta herramienta incluirá la posibilidad de crear estudios, estos son combinaciones de *datasets*, *flows* y *runs* para colaborar con otros usuarios o simplemente llevar a cabo un seguimiento del trabajo realizado [40].

Con todo ello OpenML permite versionar tanto código, modelos y datos, con un objetivo doble: colaborar en el proceso de creación de modelos con otros usuarios y realizar experimentos reproducibles.

## 2.6 Delta Lake

Herramienta de software libre que se encuentra en la capa inmediatamente superior a un sistema de almacenaje de tipo *data lake*, como muestra la figura 2.7. Su objetivo es conferir fiabilidad a los datos, permitir un manejo escalable de estos y proveer transacciones atómicas, consistentes, aisladas y durables (ACID). Por tanto, esta herramienta actúa sobre la problemática expuesta en la subsección 1.1.1.

Es importante realizar una revisión del concepto *data lake*. Históricamente los datos han sido almacenados en bases de datos o *data warehouse*<sup>1</sup>. Este tipo de almacenamiento filtra los datos y les confiere estructura antes de ser guardados. En contraste, en el caso de *data lake*, se almacena todo, tanto si son datos estructurados o sin estructura. Esta distinción provoca que la calidad del dato sea muy distinta entre ambas opciones.

Algunas de las diferencias clave entre estos dos conceptos son [29, 41]:

---

<sup>1</sup>Conviene mencionar que los conceptos base de datos y *data warehouse* no son sinónimos. Este último se encarga de realizar análisis de los datos almacenados en las distintas bases de datos que conforman un negocio. Un *data warehouse* permite llevar a cabo el análisis de los datos de manera adecuada y estructurada, mientras que una base de datos únicamente está preparada para almacenar datos y consultarlos



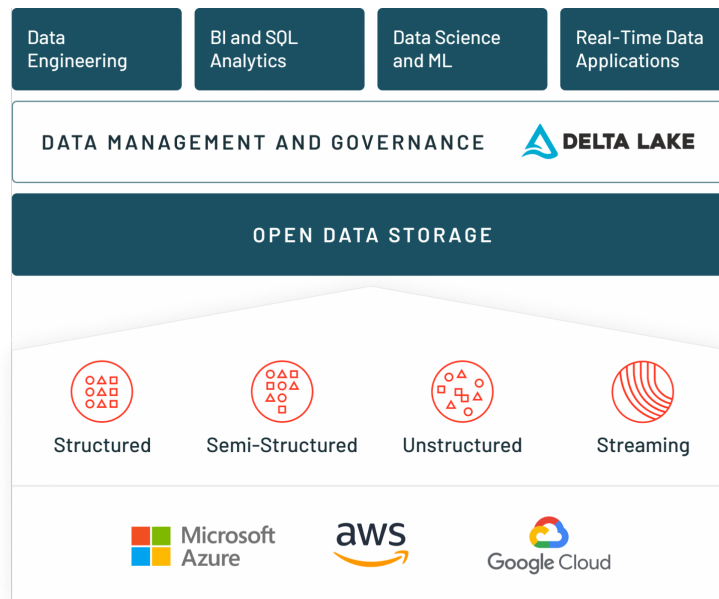


Figura 2.7: Ubicación de Delta Lake en el almacenamiento de datos. Se encuentra entre el *data lake* y los usuarios que hacen uso de estos datos. Tomado de [30]

- **Procesamiento.** Antes de almacenar los datos en un *data warehouse* es necesario proveerlos de estructura, lo que se conoce como *schema-on-write*. En un *data lake* se almacenan directamente y, en el momento en el que sean usados, se les provee de estructura, se trata de *schema-on-read*.
- **Agilidad.** Al ser un repositorio muy estructurado, un *data warehouse* hace complicada la tarea de cambiar dicha estructura, mientras que un *data lake* permite reconfigurar en tiempo real los modelos y consultas asociados a estos datos.
- **Seguridad.** La tecnología que implementa un *data warehouse* es más madura que la de un *data lake*, por lo que los protocolos para asegurar esos datos se encuentran más probados.
- **Usuarios.** En un *data warehouse* los usuarios directos son analistas de negocio, mientras que en el caso del *data lake* son científicos de datos.

En esta situación, aparecen los siguientes problemas asociados al *data lake* [18]:

- Los datos suelen ser bastante desordenados. Normalmente los datos que se recopilan se almacenan sin aún conocer cual será su propósito o la razón por la que son guardados.
- En gran medida el desorden proviene de que los datos se componen de ficheros de poco tamaño y de distintas naturalezas.
- Es bastante frecuente que los datos estén corruptos y sea necesario llevar a cabo un versionado para volver al estado anterior.

Con el fin de solucionar los problemas anteriormente expuestos, Delta Lake ofrece las funcionalidades siguientes [2, 32]:



- **Almacenaje de datos en formato Apache Parquet.** Se trata de un formato rápido de codificación y decodificación de código abierto.
- **Manejo escalable de datos y metadatos.** Delta Lake hace posible escalar los datos facilitando el acceso simultáneo de varios usuarios a los distintos ficheros o tablas. Para ello, Delta Lake utiliza un fichero llamado `Delta log`. En él se escribe al final de cada operación un estado válido de todos los ficheros o tablas usadas, cumpliendo con los principios de transacciones ACID.
- **Histórico de versiones.** Delta Lake almacena un historial completo de cualquier cambio que tenga lugar en los datos. Permitiendo a los desarrolladores acceder a versiones pasadas.

Ello permite a los científicos de datos elaborar experimentos reproducibles, por ejemplo, es típico en ML requerir una versión antigua de los datos para comparar un algoritmo de entrenamiento nuevo y uno antiguo sobre los mismos datos.

- **Validación y evolución de los esquemas.** Los *dataframes* que utiliza Delta Lake contienen un esquema que especifica su tamaño, tipo de datos, columnas presentes y metadatos.

Delta Lake utiliza *schema enforcement* o validación del esquema, consiste en evitar que los usuarios contaminen las tablas con datos erróneos o insertando datos en columnas que no existen. Para ello utiliza validación *schema-on-write*, de esta forma se comprueba en cada nueva escritura si los nuevos datos son compatibles con el esquema proporcionado. Con ello, se consigue que el esquema no cambie de manera imprevista y no deseada con el tiempo [16].

Por otro lado, *schema evolution* o evolución de esquemas permite añadir nuevas columnas a los datos de manera dinámica. Delta Lake automáticamente crea el nombre de la nueva columna e indica el tipo que corresponde, lo que permite añadir nuevas columnas sin corromper las ya existentes.

- **Apoyo en las operaciones de borrado y mezcla.** Delta Lake permite actualizar, borrar o mezclar tablas de manera atómica y mediante una sintaxis muy similar a la que provee SQL. Gracias a ello posibilita almacenar de forma rápida cambios que tengan lugar en los datos.
- **Unificación de datos de *streaming* y *batch*.** Ambos tipos de datos provienen de fuentes muy diferenciadas. En el caso de *batch* el volumen de datos es elevado y se ha recolectado a lo largo del tiempo, para, posteriormente, realizar el procesamiento. Por otro lado, los datos en *streaming* suceden en tiempo real y su procesamiento y análisis en requerido casi al instante. Debido a ello era necesario hacer lecturas distintas según el tipo de dato.

Delta Lake permite leer y escribir ambos tipos de datos en la misma tabla, incluso simultáneamente, lo que soluciona el problema de leer ambos tipos de datos de manera separada.

Adicionalmente, existe una variedad de herramientas propietarias, como Neptune [39], que proporcionan comparativas entre los modelos, compartición y programación de experimentos y monitorización de modelos desplegados. Los interesados en obtener una perspectiva general de las herramientas de **MLOps** clasificadas según su propósito pueden consultar [19].

## Capítulo 3

### Marco teórico

En este capítulo se presenta, tentativamente, un esquema de diseño de cómo utilizar estas herramientas de **MLOps** en un sistema real en producción. En primer lugar, se realizará una revisión de la organización típica de los equipos de trabajo en las organizaciones. En segundo lugar, se revisarán los diferentes escenarios de MLOps, comenzando por un escenario totalmente manual hasta llegar a uno completamente automatizado. En tercer lugar, se revisará el concepto de *pipeline*. Por último, se describirán los experimentos a desarrollar en los capítulos 5 y 6.

A la hora de implementar un modelo de ML en el ámbito empresarial intervienen diferentes equipos técnicos, según la fase y tarea en la que se encuentre el proyecto. En concreto, en la fase de desarrollo intervienen los ingenieros de ML y los científicos de datos, mientras que en la fase de despliegue participan los ingenieros de software y los equipos de operación o administradores del sistema. Ambas partes se encuentran unidas por el repositorio de código en el que trabajan conjuntamente, ver figura 3.1.

Los ingenieros de ML y los científicos de datos, también llamados equipo de desarrolladores de modelos, se encargan del análisis de datos y de las fases de construcción, entrenamiento, pruebas y validación de los modelos. Una vez hecho esto, suben el proyecto a un repositorio de código. A continuación, los ingenieros de software se encargan de integrarlo con la infraestructura ya existente, mientras que el equipo operativo se encarga de monitorizar la aplicación y proveer *feedback* al equipo desarrollador de modelos [1].

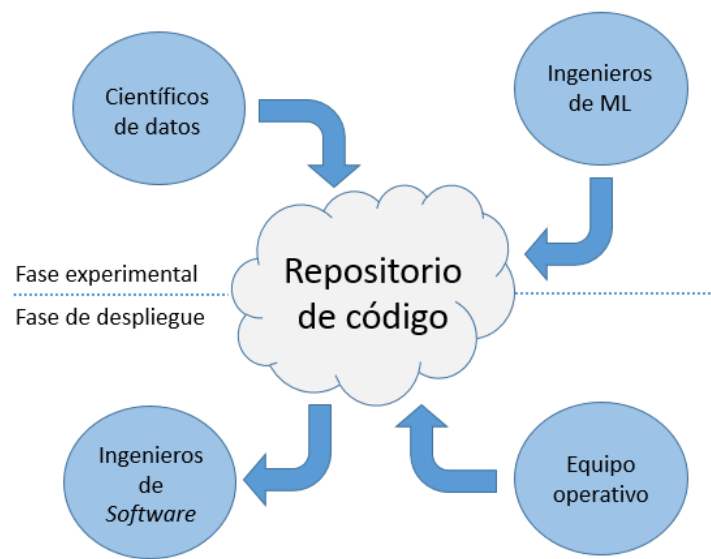


Figura 3.1: Organización de los equipos técnicos en la empresa a lo largo de todo el ciclo de vida de un modelo de ML.

### 3.1 Escenarios de MLOps

Un escenario de **MLOps** es la situación existente en una organización para llevar a cabo el proceso de desarrollo y despliegue a producción de un proyecto de ML [38]. Existen tres escenarios distintos que representan los diferentes niveles de automatización: procedimiento manual sin integración de principios de MLOps, entrega continua de modelos (*continuous delivery*, CD) y procedimiento totalmente automatizado (**CI/CD**).

#### 3.1.1 Procedimiento manual

El procedimiento manual es el escenario donde el procesamiento de datos y la creación de modelos son llevados a cabo manualmente, mientras que en producción los ingenieros de software deben integrar manualmente la aplicación en la infraestructura existente. Esta situación sin integración de principios de **MLOps**, capítulo 2, ocasiona los siguientes problemas:

- Es necesario analizar de manera periódica y continua las nuevas tendencias en los datos, así como llevar a cabo repetitivamente el proceso de entrenamiento, pruebas y validación, porque los datos cambian y los modelos que eran válidos anteriormente ya no lo son. Todo ello conlleva una gran inversión de recursos a lo largo del tiempo y una pérdida de beneficio para la compañía.
- Es posible que, con el tiempo, la arquitectura del modelo cambie, y, por tanto, los ingenieros de software deban volver a integrar el modelo en la aplicación ante cada cambio que tenga lugar. Un ejemplo de esta situación es que se produzca un cambio en el estado del arte, por

lo que si se desea hacer uso de todo el potencial de la disciplina es necesario introducir los nuevos cambios.

Como se puede ver no solo es costoso el proceso de implementación del modelo, sino que se hace aún más complejo el mantener actualizada la aplicación a lo largo del tiempo. Debido a algunos de estos problemas aparece el concepto de MLOps, capítulo 2, con el fin de automatizar tareas y facilitar la cooperación entre los equipos de desarrollo y de operación [1, capítulo 3].

### 3.1.2 Entrega continua de modelos

El escenario con entrega continua de modelos es el punto intermedio entre el procedimiento manual y uno totalmente automatizado. Se utilizan *pipelines* para el desarrollo y despliegue continuo de modelos (CD) de manera que los modelos son creados, entrenados, validados y desplegados mucho más rápidamente. Para implementarlo habría que llevar a cabo algunas modificaciones respecto al escenario manual anterior:

- Implementación de un almacén de características, *feature store*. Consiste en llevar a cabo el mismo procedimiento de procesamiento y limpieza para todos los datos brutos que se recojan. De esta forma, todos los datos estarán estandarizados a una definición común y serán almacenados en un lugar centralizado.

Es común encontrar fuentes donde *feature store* se utiliza como sinónimo de *data lake*, introducido en la sección 2.6. En ambos casos se hace referencia al lugar centralizado donde residen los datos en bruto.

- Implementar un *pipeline*, que lleve a cabo una construcción y análisis automatizado de los modelos. De esta forma, el equipo de desarrolladores de modelos seleccionaría el modelo y especificaría algunos hiperparámetros. El *pipeline* procesaría los datos tomados del *feature store* utilizando un procedimiento concreto para el modelo elegido, ya que diferentes modelos pueden requerir un procesamiento de datos específico. A continuación, construiría, entrenaría y validaría el modelo. Incluso durante la etapa de validación el propio *pipeline* podría encontrar los hiperparámetros óptimos para el modelo. Esta propuesta es lo que se conoce como *orchestrated experiment* o *automated model building and analysis*.

Puede suceder que, tras crear y probar el modelo, no se obtenga un rendimiento satisfactorio, por lo que es necesario volver a repetir todo el proceso de creación del modelo.

- De la misma forma, se puede definir otro *pipeline* que despliegue los modelos en la aplicación de forma automática y reentrenarlos en caso necesario sin necesidad de modificar los hiperparámetros y repetir la etapa experimental. Esta configuración se conoce en la literatura como *automated pipeline* o *automated training pipeline*.

El reentrenamiento del modelo puede ser activado manualmente, por un entrenamiento programado, porque el rendimiento del modelo ha disminuido, o porque aparecen nuevas tendencias en los datos.

- Almacén central de modelos o *model registry* que almacena y versiona tanto los modelos (incluidos sus pesos o parámetros). De esta forma se despliega el modelo que mejor resultado haya obtenido, logrando establecer el nexo de unión entre los desarrolladores de modelos y los equipos de operación.

El *model registry* pertenece a un ámbito local, de manera que únicamente almacena modelos entrenados de un proyecto concreto. Sin embargo, también es posible considerar un almacén de modelos cuyo alcance sea toda la empresa, es lo que se conoce como *model catalog*. En este caso no se almacenan los parámetros, o pesos, del modelo, sino, que se pretende llevar a cabo una bitácora de las decisiones llevadas a cabo en el desarrollo de un modelo para resolver un problema concreto (algoritmo de ML utilizado, hiperparámetros del mismo, etc.). Más adelante en el capítulo se realiza una definición más detallada.

Estas modificaciones consiguen automatizar el proceso de crear y desplegar el modelo sin necesidad de repetir todo el ciclo ante cualquier cambio. Sin embargo, aún existen algunos problemas que requieren una solución manual, como por ejemplo la necesidad de reconstruir manualmente ciertas partes del *pipeline* ante cambios en la estructura del código o la inexistencia de mecanismos que prueben dicho *pipeline* y eliminen errores. Todo ello hace necesario pasar al siguiente escenario.

### 3.1.3 Proceso totalmente automatizado

Por último, un proceso totalmente automatizado es un escenario en el que, se lleva a cabo el *testing* del *pipeline* u otros componentes, CI, hasta la automatización en la creación y despliegue del modelo o CD. Por tanto, sería necesario realizar las siguientes modificaciones respecto al escenario anterior:

- *Testing* del *pipeline*. Es importante probar tanto el modelo como el *pipeline*, con el fin de asegurar que las salidas proporcionadas son las esperadas por la aplicación. Por ejemplo, es necesario comprobar el correcto funcionamiento en el cálculo y análisis de las métricas del modelo, pues si dicho modelo tiene un buen desempeño pero hay un fallo en análisis o cálculo, puede causar problemas y retrasos en el despliegue. Otro problema más común a tener en cuenta sería la revisión de un correcto procesamiento de los datos, ya que puede acarrear un bajo rendimiento del modelo.
- Repositorio de código. En este almacén se guarda el código de los *pipelines* y modelos desarrollados. Los equipos pueden desarrollar varios de ellos para diferentes propósitos y almacenarlos aquí.

Es típico asociar directamente repositorio de código con la herramienta GitHub. Pero en este caso no es así. En GitHub no se pueden almacenar grandes volúmenes de código (como por ejemplo grandes modelos de ML) por lo que es necesario buscar otras alternativas, en ese sentido en la sección 3.2 se propone una solución.

- Opcionalmente se puede incluir un almacén de *pipelines* o *package store*, esto es un lugar centralizado donde se encuentran los diferentes *pipelines* listos para ser utilizados en el despliegue de los modelos en producción. Esta unidad puede verse como una aplicación de CD a los *pipelines*.

Cabe destacar que los *packages* no se encuentran estandarizados y, por tanto, existen varias formas de empaquetado. Así, un *package* puede ser tan solo el *pipeline* de despliegue a producción, o en tal caso su especificación indicando entradas y salidas de cada tarea que lo compone, o puede ser el dicho *pipeline* y el modelo entrenado que se pretende desplegar, de manera que en este *package* se encuentra todo lo necesario para implementar en producción el modelo seleccionado, haciendo reproducibles los productos desarrollados en el área más experimental al entorno de producción.

Por tanto, el *package* contendrá todas las dependencias, como paquetes software necesarios o archivos de configuración, y ficheros ejecutables para desplegar directamente los modelos. Un ejemplo de estos paquetes son los contenedores que Docker ofrece.

Gracias a este escenario se puede mantener un ciclo totalmente automatizado sin apenas intervención manual. Con ello se logra mantener a punto la aplicación ante cualquier cambio, por mínimo que sea, ya bien se trate de los datos, modelo o código. El esquema final se ilustra en la figura 3.2 y los pasos representados son los siguientes:

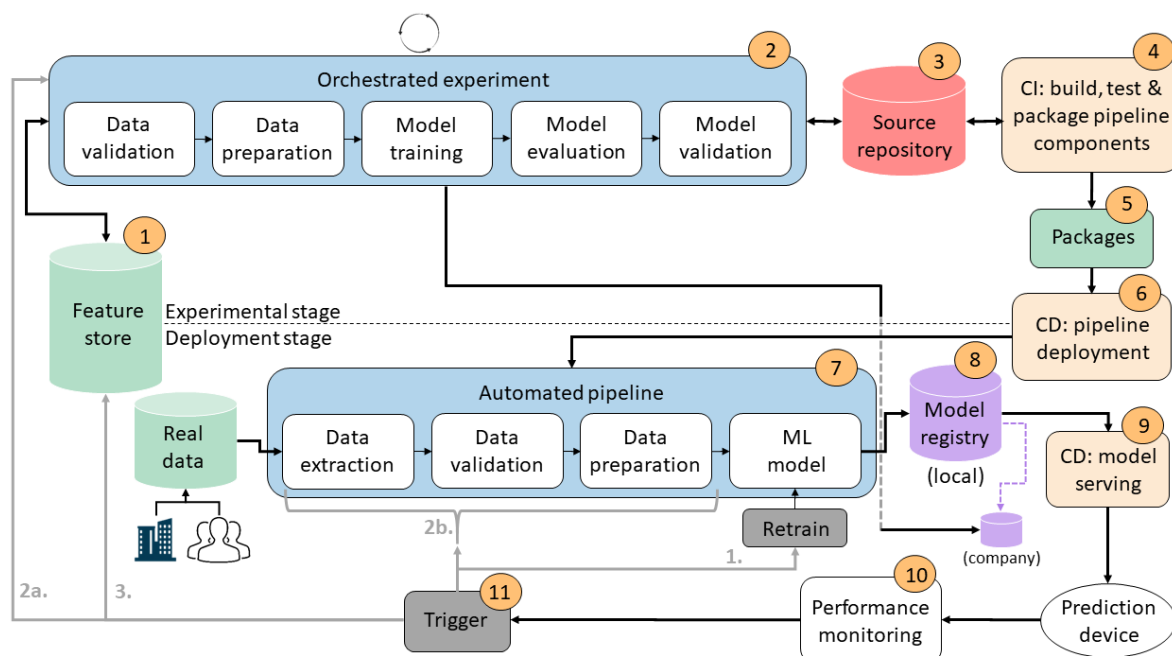


Figura 3.2: Esquema del desarrollo y despliegue de un modelo de ML totalmente automatizado con CI/CD. Basado en [6, figura 8-5] y [1, figura 3-4].

1. Estandarización común de los datos procesados en características, para almacenarlas en el *feature store*. Estas características pueden ser utilizadas para entrenar nuevamente el *automated pipeline*.
2. Creación, entrenamiento, pruebas y validación del modelo automáticamente. Este paso es cíclico, ya que si no se obtiene el rendimiento o el resultado deseado, se debe ejecutar de nuevo el *pipeline*.
3. Almacenar en un repositorio centralizado el código de los *pipelines* y modelos desarrollados. Es posible que los equipos experimentales implementen distintos modelos para varios propósitos, que serían almacenados en el repositorio de código. En la sección 3.2 se aclara qué herramientas pueden implementar este repositorio.
4. Probar el modelo y el *pipeline* con el fin de evitar errores relacionados con la fase más experimental que puedan afectar al rendimiento del modelo una vez desplegado. Este paso es clave para lograr CI.
5. Almacenar en un depósito central paquetes que han superado las pruebas anteriores y se encuentran listos para ser desplegados.
6. Se selecciona el paquete del *package store* y es desplegado a producción de manera automática. Además, en esta etapa pueden incluirse más pruebas con el fin de asegurar una futura correcta integración del *pipeline* en la aplicación.
7. Este módulo logra un objetivo doble: procesar los datos para los cuales se requieren realizar predicciones y actualizar el modelo o el *pipeline* cuando se produce un desencadenante. Como se puede ver en la figura 3.2, este *pipeline* es alimentado con los datos procedentes de *real data*. Este elemento representa los datos para los cuales los consumidores de la empresa desean obtener predicciones. A diferencia del *feature store* estos datos no son conocidos de antemano por parte de la compañía y, por ello, no pueden ser utilizados en la creación del modelo.

En el caso de que se haya producido un desencadenante o *trigger* se puede realizar una acción, como reentrenar el modelo automáticamente. A continuación, se detallan las cuatro posibles causas por las que se pueda producir este desencadenante y sus respectivas acciones:

1. Reentrenamiento automático del modelo ya desplegado con datos que incluyan las nuevas tendencias. Es lo que se conoce como *data drift* o cambios en los datos de entrada.
2. Cambios en alguno de los componentes del modelo, ya sea:
  - 2a. Cambio en las entradas del modelo, para ello es necesario repetir la etapa del *orchestrated experiment* y realizar un nuevo procesamiento de estas nuevas entradas. Este caso debe ser revisado manualmente.
  - 2b. Cambio en el *pipeline* en producción, por ejemplo puede ser necesario realizar un nuevo procesamiento de datos de las entradas del modelo. Este caso es llevado de forma automática.



3. Cambio en el modelo, en este caso se comienza de nuevo el ciclo de creación, entrenamiento y despliegue del modelo. Debe ser llevado a cabo manualmente.
8. Almacenar y versionar en un lugar centralizado, o *model registry*, los distintos modelos que ya han sido entrenados junto a sus métricas y parámetros.
9. Poner en servicio los modelos con los mejores rendimientos para que el usuario final comience a hacer uso de la aplicación.
10. Monitorización continua del desempeño del modelo y recopilación de los nuevos datos proporcionados por los usuarios, los cuales serán almacenados en el *feature store*.
11. En el caso de que el modelo desplegado presente un rendimiento por debajo del adecuado es necesario identificar la causa correspondiente y tomar una de las acciones especificadas en el séptimo paso.

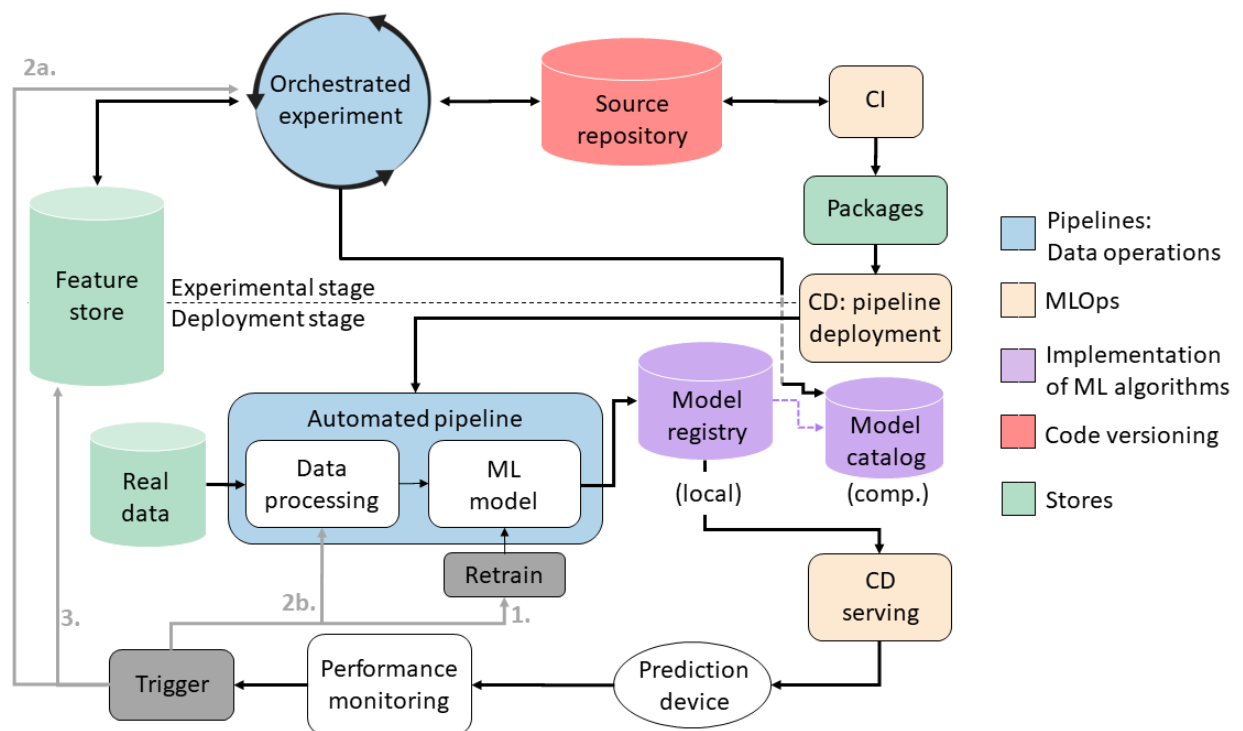


Figura 3.3: Diagrama de bloques que muestra la relación entre las distintas áreas presentadas en este proyecto.

En la figura 3.3 se muestra la relación existente entre los tipos de elementos en un escenario totalmente automatizado de manera más abstracta y general que en la figura anterior.

De manera general, los *pipelines* se utilizan para automatizar y modularizar el código. El *pipeline* en la fase experimental, u *orchestrated experiment*, dicta las transformaciones a realizar sobre los

datos que lo atraviesan, tomados del *feature store*, o en su caso del *data lake*, para después construir y entrenar el modelo seleccionado.

A continuación, al resultado del *pipeline* experimental se le aplican tareas con el fin de cumplir los principios de **MLOps**, en este caso se trata de **CI/CD** con el fin de probar todos los componentes y automatizar el despliegue del modelo. De nuevo, el modelo atraviesa otro *pipeline*, o *automated pipeline*, con el fin de facilitar la mantenibilidad del modelo ante cambios en la tendencia de los datos que puedan suceder con el tiempo. Previamente a que el usuario final pueda acceder a la aplicación, el modelo es registrado y versionado en el *model registry* permitiendo organizar la evolución de los modelos, facilitar la cooperación entre los distintos equipos técnicos y facilitar la transferencia del modelo a producción.

## 3.2 Herramientas asociadas al proceso automatizado

En la sección anterior se detallan los distintos escenarios de **MLOps** y se presenta el esquema final del ciclo totalmente automatizado. Sin embargo, no se indican las herramientas software que implementan este proceso, por ello es necesario establecer el nexo de unión entre la subsección 3.1.3 y el capítulo 2 de estado del arte.

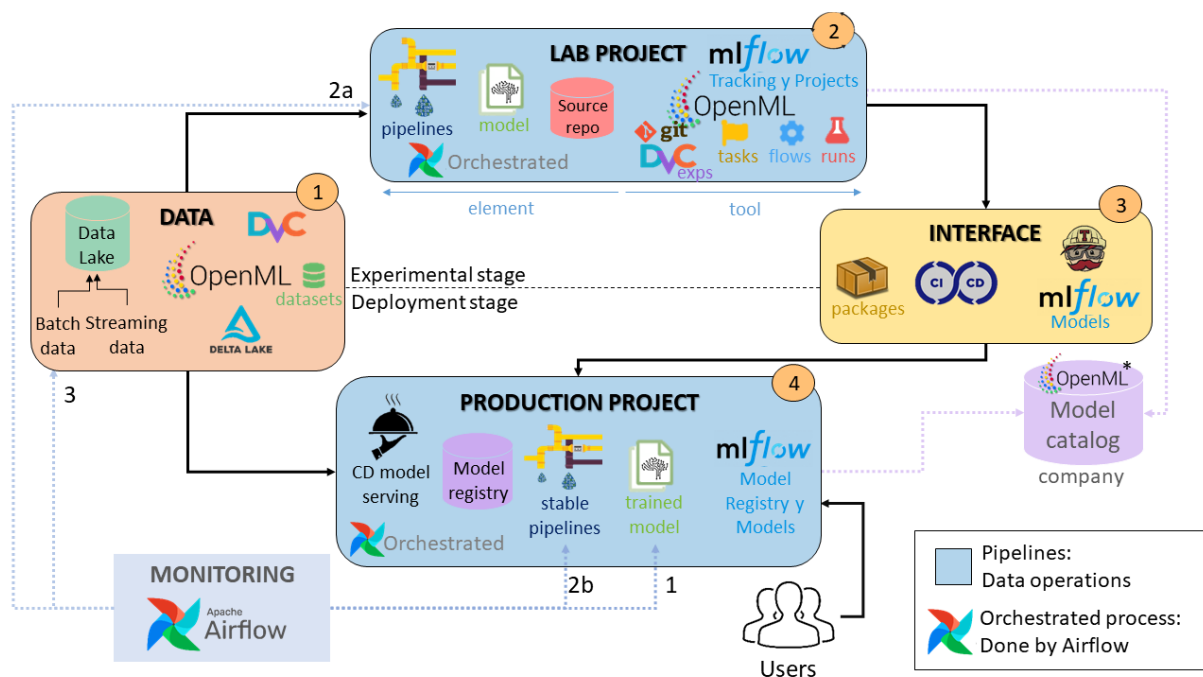


Figura 3.4: Diagrama de bloques genérico de las herramientas asociadas a un proceso totalmente automatizado. Se ilustran los cuatro grandes bloques que componen el proceso y las herramientas que implementan las funcionalidades asociadas a esos bloques. (\*) En el *model catalog* se ha incluido la herramienta OpenML pues, según indican en su documentación, en futuras versiones es posible que se incluya las funciones necesarias para implementar este almacén.

En la figura 3.4 se ilustra el procedimiento totalmente automatizado a alto nivel de abstracción e indicando las herramientas asociadas en cada etapa. Los pasos ilustrados son los siguientes:

1. En primer lugar se encuentran los datos, tal y como se ha explicado en el capítulo 2, las compañías incluyen un *data lake* en el que almacenan cualquier dato que consideren que pueda ser necesario en el futuro. Estos datos pueden ser de tipo *batch* o de tipo *streaming*.

Debido al gran volumen de datos es necesario organizarlos y limpiarlos. Para ello existen herramientas que versionan los datos como DVC, los gestionan y organizan como Delta Lake, o directamente proporcionan datos ya procesados y listos para usar como OpenML datasets.

Es importante prestar atención a esta etapa, pues la calidad del modelo implementado depende directamente de los datos con los que es alimentado, subsección 1.1.1.

2. A continuación, en la segunda etapa, se debe llevar a cabo la elaboración del modelo. Lo ideal es que el equipo de desarrolladores de modelos seleccionen el modelo y los hiperparámetros, y que el *pipeline* construya, pruebe y valide el modelo de forma automatizada (ver la subsección 3.1.2).

Como se puede ver se encuentran elementos ya mencionados como el *pipeline*, el modelo que se pretende crear y el repositorio de código en el que se registran las distintas versiones del código.

En esta etapa se encuentran herramientas como Apache Airflow, que se encarga de ejecutar el *pipeline* compuesto por las tareas descritas anteriormente. DVC y git pueden implementar el repositorio de código, gracias a ello la primera organiza, compara, versiona y almacena experimentos<sup>1</sup>, datos y modelos, mientras que en git se versionan los ficheros de metadatos generados por DVC y se comparten los experimentos. MLflow con el módulo Tracking permite realizar un seguimiento de las ejecuciones y métricas asociadas, mientras que, con el módulo Projects, permite empaquetar los experimentos y hacerlos reproducibles en otros entornos. Y, por último, OpenML con las tareas, flows y runs permite visualizar cómo otros usuarios han resuelto problemas que puedan estar relacionados.

Tras esto, los hiperparámetros asociados al modelo, junto con un registro de las decisiones tomadas, son almacenados en el *model catalog*. Tal y como se ha mencionado, es el lugar centralizado donde se encuentran todos los modelos desarrollados por la compañía.

Cabe destacar que esta etapa es cíclica, por lo que, si se obtiene un modelo cuyo rendimiento no es el deseado, es necesario repetirla de nuevo.

3. La siguiente etapa consiste en llevar a cabo las pruebas necesarias a los distintos componentes, mediante tareas de CI, antes de transferir el modelo a producción de manera automatizada, con *pipelines* de CD.

---

<sup>1</sup>Entendido por experimento al *pipeline* y conjunto de parámetros, que producen un determinado modelo de ML

El elemento clave son los *packages*, como se ha explicado antes, que consisten en versiones empaquetadas el modelo entrenado, o el modelo entrenado y el *pipeline* de producción.

En esta etapa se encuentra el módulo MLflow Models y la herramienta Travis. La primera es utilizada para empaquetar el modelo de ML y desplegarlo en servicios en la nube como AWS o AzureML, mientras que Travis es utilizada para llevar a cabo las tareas de pruebas de los distintos elementos de forma automatizada.

4. Por último, la etapa que cierra el ciclo es el *production project*, cuya funcionalidad consiste en mantener actualizado el modelo desplegado.

Se encuentran elementos como el modelo ya entrenado, *pipelines* estables, el *model registry* y el elemento que propiamente transfiere a producción el modelo para que los usuarios hagan uso de este.

De nuevo, en esta etapa se encuentra Apache Airflow que se encarga de ejecutar el *pipeline* de procesamiento sobre los datos para los que se requiere una predicción por parte del modelo desplegado. Por otro lado, MLflow Model Registry mantiene un registro cronológico de las versiones de los modelos y los transfiere a producción, mientras que el módulo MLflow Models permite desplegarlo localmente o en plataformas concretas.

Tal y como sucedía en la segunda etapa, en el caso de que las métricas del modelo disminuyan por debajo de cierto umbral y se decida que para solucionarlo se debe reentrenar el modelo o cambiar ciertas partes del *pipeline*, la toma de decisiones llevada a cabo y dicho modelo sin entrenar deben ser almacenados en el *model catalog*.

En la figura 3.4 existen dos brechas de funcionalidad, representadas por colores más tenues:

1. **Monitorización.** La monitorización del modelo desplegado puede ser llevada a cabo por la herramienta Apache Airflow, de manera que puede notificar a los científicos cuando las métricas disminuyan por debajo de cierto umbral. Esta caída en el rendimiento puede ser solucionada por acciones bien diferenciadas y explicadas en el texto principal asociado a la figura 3.2: reentrenamiento del modelo desplegado, cambios en las entradas, cambio en el *pipeline* en producción o cambio en el modelo.

Sin embargo, aún no se ha propuesto una herramienta o técnica para decidir cuál de las cuatro soluciones es necesario implementar cuando un modelo disminuye su rendimiento.

2. **Model catalog:** almacén que guarda los modelos sin entrenar junto con las decisiones que han llevado a seleccionar dicho modelo. Asimismo, también es posible almacenar modelos entrenados a modo de ejemplo.

El *model catalog* puede verse como un *model registry* cuyo alcance es toda la compañía, pero no almacena los pesos, o parámetros, que generan el modelo, aunque sí los hiperparámetros. Además, no solo versiona los modelos si no que justifica la toma de decisiones llevada a cabo

para desarrollar un modelo concreto. De esta forma, distintos proyectos de la misma compañía pueden beneficiarse de los avances realizados por otros equipos técnicos.

Actualmente no existen herramientas de código abierto que permitan implementar este almacén. Sin embargo, la herramienta propietaria RStudio Connect sí implementa esta función y OpenML probablemente en el futuro también lo incorpore.

### 3.3 Automatización usando *pipelines*

A lo largo del presente capítulo se ha mencionado repetidamente el concepto de *pipeline*, el cual se introdujo en el capítulo 2.

Este es un artefacto clave en la automatización de tareas que consigue simplificar código complejo en tareas más simples. En realidad el concepto *pipeline* tiene un doble sentido: por un lado se refiere a la secuencia de tareas abstractas que es necesario implementar, y por otro, se refiere a la instancia y gestión automatizada de esas tareas, lo cual puede ser llevado a cabo por una herramienta como Airflow. En la figura 3.5 esquematiza esta idea.

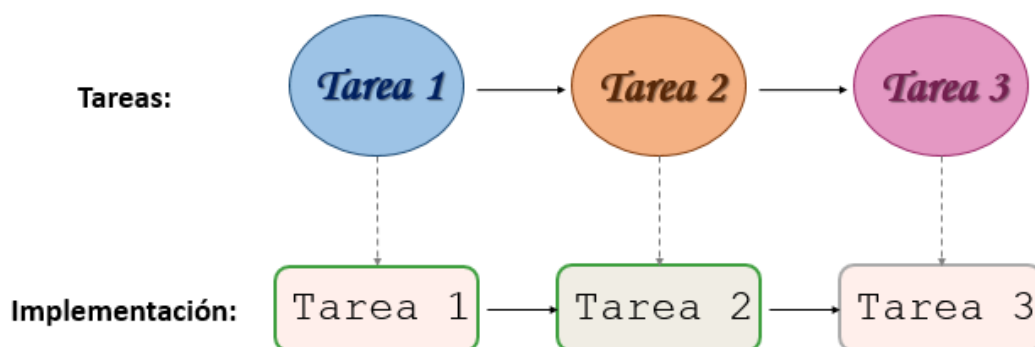


Figura 3.5: Los *pipelines* son tareas abstractas y la implementación de estas en software para su ejecución y gestión, en herramientas como Airflow.

Gracias a estos artefactos es posible obtener los siguientes beneficios [25]:

- **Facilidad a la hora de entender cada tarea.** Debido a que un problema grande ha sido dividido en varios pasos, es más sencillo entender de un vistazo cada tarea que lo compone.
- **Facilidad a la hora de entender todo el *pipeline* en conjunto.** Gracias a la división en tareas se pueden abstraer los detalles y entender el conjunto a alto nivel.
- **Modularidad.** Si las tareas han sido implementadas de una manera flexible, es posible utilizarlas en otros contextos, reduciendo la cantidad de código necesario.
- **Facilidad a la hora de realizar cambios en el código.** Gracias a la división en tareas es posible modificar únicamente una parte del código dejando intacto el resto.

- **Acceso a resultados intermedios.** Debido a que el código está dividido en tareas, es sencillo guardar el resultado de cada una de ellas, haciendo más sencillo validar ese *pipeline*.
- **Escalabilidad.** Gracias a la gestión de los *pipelines* que permiten algunas herramientas y paquetes es posible escalar el número de esos *pipelines* en el mundo empresarial, pues en caso de que no existiese un gestor de estos *pipelines* su manejo y gestión serían muy complejos y poco eficientes.

Un ejemplo aplicado de estas ventajas sucede cuando un trabajador se va de vacaciones. En el caso de que el código no estuviese modularizado y apareciese un error en la parte correspondiente al trabajador ausente, el resto del equipo debe lidiar con el problema intentando entender grandes y complejos fragmentos de código difíciles de manejar. Sin embargo, si se usan *pipelines* el error puede ser encontrado y solventado fácilmente en una de las tareas que lo componen, pues éstas estarán estandarizadas y organizadas.

## Capítulo 4

# Diseño e implementación práctica

En el presente capítulo se plantearán las líneas comunes y generales de los casos prácticos planteados en este proyecto. En primer lugar, se detallarán los *datasets* seleccionados junto con el procesamiento llevado a cabo en ambos experimentos. En segundo lugar, se describirán las necesidades por las que se desarrollan dichos casos. En tercer lugar, se describe la metodología común a ambos. Y, por último, se plantearán las dimensiones o criterios a comparar entre ellos.

En capítulos anteriores, 2 y 3, se ha presentado una amplia gama de soluciones tecnológicas que pueden ser utilizadas en un ciclo totalmente automatizado, figuras 3.3 y 3.4. Siendo la meta final desplegar modelos de ML en producción usando los principios propuestos por **MLOps**, capítulo 2. El objetivo de estos experimentos es probar tres tecnologías de software libre para evaluar su funcionamiento de manera práctica. Sin embargo, con el objetivo de ilustrar en profundidad la funcionalidad que provee cada herramienta se llevará a cabo un planteamiento diferente en determinados pasos del experimento, como por ejemplo sucede en la creación de variables necesarias para la ejecución de los experimentos. En la figura 4.1 se esquematizan las tres herramientas a utilizar en los experimentos: DVC, MLflow y Apache Airflow.

Los dos casos prácticos desarrollados se centran especialmente en el apartado *lab project* de la figura 3.4. El objetivo es comparar y mostrar las herramientas que contribuyen a desarrollar esta fase del diagrama. Sin embargo, el segundo experimento también aborda las fases de *interface* y *production project*, pues recordemos que es una de las fases de mayor dificultad en su implementación en las empresas, capítulo 1.

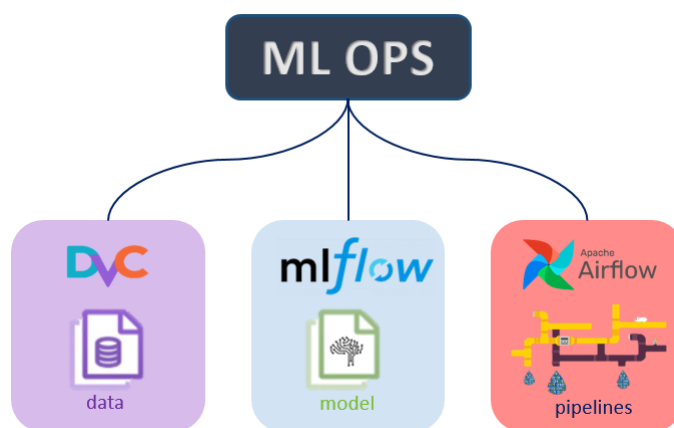


Figura 4.1: Esquema de las herramientas a utilizar en los casos prácticos.

## 4.1 Datos y procesado

El problema abordar, y por tanto el modelo que se pretende desarrollar, consiste en la predicción de si una universidad está o no entre las mejores del mundo. Para ello, utilizaremos los *datasets* especificados en la subsección 4.1.1 en los cuales se indica la posición mundial que ocupa cada universidad. Con esta información se abordará como un problema de clasificación, donde la etiqueta 0 será que la universidad no está entre las mejores, mientras que si su valor es 1 la universidad está entre las mejores del mundo (el número de universidades consideradas como “mejores” varía según el *dataset* y el experimento tal y como se puede ver en la tabla 4.1, parámetro `world_rank`),

Los datos usados en ambos casos prácticos se han descargado de la plataforma Kaggle en [este enlace](#). Estos *datasets* contienen tres clasificaciones llevadas a cabo por instituciones distintas sobre las mejores universidades a nivel mundial. Cada una de ellas evalúa diferentes aspectos de las universidades con el fin de asignarles una posición en la clasificación mundial.

Asimismo los datos descargados incluyen dos *datasets* adicionales con los que completar los datos de las instituciones anteriores. Sin embargo, debido a la cantidad de valores faltantes y a la falta de unificación de determinados campos, como por ejemplo el nombre de la universidad, se ha decidido no usar ninguno de ellos. Es conveniente recordar que el objetivo de este proyecto no es llevar a cabo un procesado riguroso de los datos.

### 4.1.1 Datos

Las clasificaciones incluidas en el *dataset* son las siguientes:

- **Times Higher Education World University Ranking (THE):** listado de mejores universidades del mundo realizado por la revista de educación inglesa Times Higher Education. Se trata de una clasificación de alto prestigio que proporciona datos muy usados a nivel universitario. Ofrece diversas clasificaciones a nivel mundial o regional y según área de conocimiento, entre otros [46].



Utiliza un total de 13 indicadores de rendimiento divididos en cinco bloques, figura 4.2: enseñanza (en el que a su vez se encuentra el indicador que mide la relación entre número de trabajadores y estudiantes, entre otros), investigación, citas, perspectiva internacional (en el que a su vez se encuentra el indicador que mide el número de estudiantes internacionales, entre otros) e ingresos del sector.

El *dataset* utilizado incluye estos cinco grandes bloques y otras variables, como número total de estudiantes y la relación sobre 100 entre número de mujeres y hombres que estudian en la universidad [44]. Cabe destacar que esta clasificación contiene valores no numéricos en la posición global para determinadas universidades a lo largo de todos los años del *dataset* (2011-2016), por ejemplo en lugar de encontrar la posición de una universidad puede aparecer el rango en el que se encuentra (301-350 o 601-800).

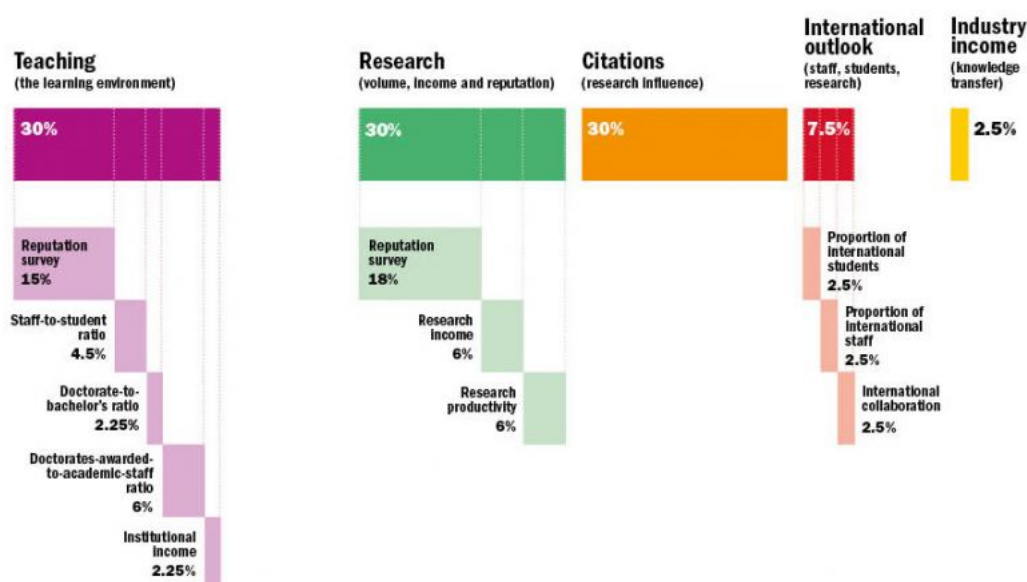


Figura 4.2: Indicadores utilizados en la elaboración de la clasificación de mejores universidades del mundo de THE. Tomado de [44].

- **Academic Ranking of World Universities o Shanghai Ranking:** se trata de otro listado de gran prestigio recopilado por un grupo de especialistas de la Universidad Jiao Tong de Shanghai, China. Su objetivo es valorar la calidad de las 1000 mejores universidades del mundo. Las universidades se clasifican en 54 áreas contenidas en las ramas de ciencias naturales, ingeniería, ciencias de la vida, ciencias Médicas o ciencias sociales.

Los indicadores considerados para elaborar esta clasificación se agrupan en los siguientes bloques: investigación, influencia de la investigación, colaboración internacional, calidad de la investigación y premios académicos internacionales [42].

Algunas de las variables que se integran en el *dataset* son número de alumnos que obtienen premios de prestigio internacionales, número de premios conseguidos por profesores e investigadores o número de publicaciones citadas en numerosas ocasiones, entre otros. De nuevo, esta clasificación contiene valores no numéricos en la posición global para determinadas universidades a lo largo de todos los años del *dataset* (2005-2015), por ejemplo en lugar de encontrar la posición de una universidad puede aparecer el rango en el que se encuentra (301-400 o 401-500).

- **Center for World University Rankings (CWUR)**: se trata de un listado de universidades menos conocido y con origen en Arabia Saudí. Esta clasificación es realizada por una organización de consultoría que proporciona asesoramiento a gobiernos y universidades con el fin de mejorar los resultados educativos y de investigación [45, 26].

Algunas de las variables más relevantes de este *dataset* son calidad de la educación, empleo de los alumnos, calidad del profesorado, resultados de investigación e influencia.

Cabe destacar que esta clasificación proporciona un listado de las 2000 mejores universidades del mundo. Sin embargo, los datos recogidos en este *dataset* únicamente contienen las 100 mejores universidades para los años 2012 y 2013, mientras que para los años 2014 y 2015 contienen las 1000 mejores universidades.

Todos los *datasets* descargados contienen las columnas año, clasificación mundial, nombre de la universidad, país al que pertenece dicha universidad y puntuación total sobre 100 obtenida aplicando en cada caso los criterios considerados.

#### 4.1.2 Procesado

Los datos descargados requieren llevar a cabo un procesado, explicado en detalle en la subsección 1.1.1, con el fin de alimentar correctamente el modelo. Sin embargo, cabe señalar que el objetivo de este proyecto no es llevar a cabo un procesado riguroso ni exacto.

El objetivo final de este procesado es completar o eliminar aquellos registros que no contengan valores, unificar el tipo correcto de datos en cada columna y renombrar algunas columnas con el fin de que sea más explicativo su título <sup>1</sup>. De nuevo, el problema se debe afrontar según la institución que realizó el *dataset*.

De manera general en todos los *datasets* se han tenido en cuenta los siguientes criterios:

1. Se eliminarán íntegramente aquellas columnas que contengan la mayoría de datos faltantes de todo el *dataset*.
2. Aquellos registros sin cifra serán sustituidos por el valor de la siguiente fila.

---

<sup>1</sup>Este procesado de datos toma en gran parte los pasos llevados a cabo por el usuario de Kaggle Youssef Adem, <https://www.kaggle.com/youssefadem/data-cleaning-and-plotting-detailed-prediction>

3. Se han eliminado todos los registros en los que en la columna clasificación total contenían algún valor no numérico.
4. Aquellas columnas que deban ser de un tipo de dato concreto se han configurado con el valor que corresponda. Por ejemplo, si una columna únicamente contiene números enteros se indicará este tipo de dato.

Aplicación de los criterios según el *dataset*:

- **THE.** Las decisiones tomadas incluyen los criterios 2, 3 y 4. Y se ha dividido la columna relación entre mujeres y hombres estudiantes en dos columnas separadas, pues la columna original contenía valores no numéricos.
- **Shanghái.** Siguiendo el primer criterio expuesto anteriormente, se ha eliminado la columna puntuación total, pues contiene casi un 70% del total de valores faltantes del *dataset*. Además también se han utilizado los criterios 2, 3 y 4.
- **CWUR.** Únicamente se hecho uso del criterio 1 en la columna impacto debido a que contiene la mayoría de valores faltantes.

Una vez procesados los datos, se ha creado una nueva columna, *chances*, y se le ha asignado una etiqueta con valor 0 o 1. La etiqueta 1 indica que la universidad considerada se encuentra entre las mejores universidades, mientras que la etiqueta 0 indica el caso contrario. Para generarlas se ha tenido en cuenta el valor indicado en el parámetro `world_rank` de la tabla 4.1. A continuación, se han tenido en cuenta exclusivamente el número de columnas indicado en el parámetro `max_features`.

Por último, estos datos han sido divididos en datos de entrenamiento, pruebas y validación, y se han almacenado en la carpeta `data/prepared_data`. Los datos de entrenamiento tienen por objetivo entrenar el modelo, y contendrán datos de todos los años excepto del último. Los datos de validación se usarán para evaluar el rendimiento de modelo y contendrán los valores del último año disponible. Y, por último, los datos de pruebas solo serán utilizados en el último experimento con el fin de hacer consultas a los modelos una vez que estos se encuentren desplegados. En este caso los datos de entrenamiento y validación son tomados aleatoriamente en un 85% y 15%, respectivamente, de los datos de todos los años excepto del último.

### 4.1.3 Parámetros

En esta subsección se describen los parámetros que pueden ser modificados en cada experimento de ambos casos prácticos.

- `world_rank`. Máximo número de universidades consideradas como las mejores en cada *dataset*. A diferencia de `max_rank`, este parámetro contiene solo las universidades que se consideran las mejores del mundo, no el número total de universidades a evaluar. Debe ser un número entero mayor que 0. Por defecto su valor es 50.

Nombre exp.	Max_rank	World_rank	Max_features	train_data	n_est
cwurData	100	50	12	cwurData	100
timesData	100	50	12	timesData	100
shanghaiData	100	50	12	shanghaiData	100
Cmr300	300	100	12	cwurData	100
Tmr300	300	100	12	timesData	100
Smr300	300	100	12	shanghaiData	100
Cf3	100	50	3	cwurData	100
Tf3	100	50	3	timesData	100
Sf3	100	50	3	shanghaiData	100
Cnest64	100	50	12	cwurData	64
Tnest64	100	50	12	timesData	64
Snest64	100	50	12	shanghaiData	64

Tabla 4.1: Tabla que detalla parámetros asociados a cada experimento desarrollado. En el texto principal se detalla el significado de cada parámetro

- **Max\_rank.** Máximo número de universidades a considerar en cada *dataset*. Debe ser un número entero mayor que 0 y superior al parámetro `world_rank`. Por defecto su valor es 100.
- **Max\_features.** Máximo número de columnas de cada *dataset* con el que alimentar al modelo de ML. Debe ser un número entero mayor que 0. Por defecto su valor es de 12.
- **Train\_data.** Los datos descargados proporcionan tres *datasets* diferentes según la institución creadora de los datos, para más información consultar la subsección 4.1.1. Este parámetro hace referencia al nombre de la institución que ha generado los datos con los cuales entrenar el modelo. Es una cadena de texto y debe ser uno de los siguientes valores: `cwurData` (valor por defecto), `shanghaiData` o `timesData`.
- **Número de estimadores, n\_est.** Este parámetro hace referencia al número de estimadores usados para entrenar el modelo de clasificación. Es un número entero mayor que 0 cuyo valor por defecto es 100.

En ambos casos prácticos se han desarrollado un total de 12 experimentos. En la tabla 4.1 se detalla el valor asociado a cada parámetro.

## 4.2 Metodología común

En el presente proyecto se han desarrollado dos casos de uso o experimentos. En ambos se hace uso de Airflow con el fin de construir, automatizar y visualizar las distintas tareas que componen el *pipeline* asociado a cada experimento. Sin embargo, en el primer experimento se integrarán Airflow (sección 2.3) y DVC (sección 2.1), mientras que en el segundo serán Airflow y MLflow (sección 2.2).

Se ha decidido desarrollar los casos prácticos de esta manera persiguiendo los siguientes objetivos: visualizar el funcionamiento de cada herramienta, evaluar los puntos en los que las herramientas DVC y MLflow se solapan y cómo cada una afronta la resolución de determinados problemas y, por último, evaluar las decisiones de diseño llevadas a cabo en un caso y otro.

### 4.2.1 Estructura general

Con el fin de evaluar de forma equitativa ambos experimentos comparten una estructura común:

1. **Comprobación de prerequisites.** Se encarga de evaluar si se cumplen los requisitos previos, detallados en el archivo `README.md` en GitHub asociado a cada caso. En ambos casos esta función es implementada por código Python en el DAG de cada experimento en Airflow.

Cabe destacar que algunos lenguajes de programación, como R, ofrecen la posibilidad de implementar herramientas de validación que pueden ser ejecutadas externamente a los programas principales. Esta herramienta no se encuentra disponible en Python por lo que se implementará como una tarea de Airflow o como código de Python en el programa principal. Se dirige a los lectores interesados a [21] para más información.

2. **Descarga de *datasets*.** Esta fase tiene por objetivo la descarga de los datos. En cada experimento esta fase es implementada de manera distinta, se detallará en el capítulo correspondiente de cada experimento la solución afrontada en cada caso.
3. **Procesado de los *datasets* descargados.** Debido a la falta de ciertos valores en algunas muestras o tipo incorrecto de los mismos, es necesario llevar a cabo un proceso de filtrado y limpieza. En ambos casos este proceso es llevado a cabo de la misma forma. En la sección 4.1 se detallan las soluciones tomadas en la limpieza de los datos.
4. **Creación de experimentos.** El objetivo es llevar a cabo una serie de experimentos o pruebas (entendiéndose por experimento al conjunto del *pipeline* y parámetros, definidos en la subsección 4.1.3) con el fin de, posteriormente, comparar entre sí todos los experimentos correspondientes al mismo caso.

Los experimentos se componen de la fase de procesamiento de datos, entrenamiento de un modelo de clasificación, concretamente se trata del algoritmo *RandomForest* de la librería *scikit-learn* [43], el cual es entrenado con los datos filtrados. Todas estas etapas serán realizadas por un *pipeline* en ambos casos prácticos.

5. **Evaluación de experimentos.** En ambos casos se compara la matriz de confusión y las métricas exhaustividad, valor-F, exactitud y precisión. Sin embargo, el modo de llevar a cabo esta evaluación se afronta de distinta manera en cada caso práctico.
6. **Confirmación por correo electrónico** de la correcta ejecución del DAG de cada experimento. De nuevo, en ambos casos es implementado como una tarea del tipo *EmailOperator*. Airflow por defecto solo ejecuta una tarea si la anterior a ella ha finalizado con un estado exitoso. De manera que si el flujo de trabajo llega a esta tarea de confirmación, significa que el caso práctico se ejecutó correctamente.

## 4.2.2 Organización en carpetas

Tal y como se ha indicado en la subsección anterior la estructura llevada a cabo en ambos casos prácticos es la misma. Esto también sucede en la organización software de las carpetas.

Ambos experimentos se encuentran en este [repositorio de código](#). Las carpetas *dvc* y *airflow* contienen algunos ejemplos implementados para entender el funcionamiento de estas herramientas previamente al desarrollo de los experimentos. Por otro lado, en las carpetas *exp1* y *exp2* se encuentran ambos casos prácticos, cuya estructura es la siguiente:

- Fichero *my\_env.yml*. Fichero que contiene todas las dependencias de Anaconda y paquetes necesarios para ejecutar correctamente cada experimento.
- Fichero *README.md*. Fichero que contiene todas las instrucciones y comandos necesarios para ejecutar el experimento. Se recomienda leer este fichero antes de ejecutar ningún experimento.
- Carpeta *src*. Contiene el código fuente necesario para desarrollar cada uno de los casos. Ambos experimentos contienen los ficheros *featurization.py*, encargado de procesar los datos descargados y obtener sus características, *train.py* encargado de entrenar el modelo de ML, *dag\\_expN.py* que contiene el código Python del DAG de Airflow y el fichero *exps.sh* que contiene la declaración de cada uno de los 12 experimentos indicados en la tabla 4.1.
- Carpeta *data*. A su vez contiene otras dos carpetas, las cuales no se encuentran en el repositorio de código de git porque no deben ser versionadas mediante este sistema, en el capítulo asociado a cada caso práctico se detalla cómo se ha afrontado su versionado.
  - Carpeta *downloaded\_data* contiene todos los datos en bruto descargados directamente del origen.

- Carpeta `prepared_data` contiene los datos procesados, mediante el fichero `featurization.py`, separados por archivos según el *dataset* de origen y según si se tratan de datos de entrenamiento, validación o pruebas.

Ambos casos prácticos han sido desarrollados en el lenguaje de programación Python y Bash. Ejecutados en un ordenador portátil con sistema operativo Linux Ubuntu versión 20.04.3 LTS, 8GB de memoria RAM, procesador intel i5-6200U, unidad central de procesamiento (*central processing unit*, CPU) de 64 bits a 2.30GHz y memoria interna de 237 GB.

## 4.3 Criterios a comparar

Existen diversos criterios o dimensiones que se pueden emplear para evaluar los experimentos desarrollados, los cuales serán calificados en un rango 0-4 puntos según la escala Likert [47]. La puntuación máxima de 4 puntos se corresponden a un nivel de “totalmente de acuerdo”, mientras que 0 puntos se corresponde con el nivel “totalmente en desacuerdo”. Se tendrán en cuenta los siguientes criterios:

1. **Facilidad de uso.** Esta dimensión tiene por objetivo evaluar la sencillez en la ejecución del experimento y la facilidad de acceso a documentación para la utilización de las herramientas detalladas anteriormente.
2. **Escalabilidad.** Esta dimensión pretende evaluar cómo de escalables son los dos experimentos desarrollados.

En las compañías pueden existir multitud de DAGs y ficheros fuente, es importante que la solución diseñada provea una adecuada gestión de todos estos archivos.

3. **Requisitos software.** Esta dimensión evalúa la facilidad en la instalación de las herramientas y las posibles problemáticas surgidas por incompatibilidades en las dependencias de las mismas.
4. **Capacidades.** Este criterio busca analizar y comparar las funciones o capacidades de dos herramientas bajo análisis, DVC y MLflow. Airflow no se incluye ya que su función es distinta a las realizadas por DVC y MLflow, y por ello sirve para la orquestación de ambos experimentos. Además, en la sección de resultados de cada caso se analizan sus capacidades.

Cada una de las funcionalidades que analiza este criterio será calificada con una puntuación entre 0 y 4 puntos. La calificación total de este criterio se realizará calculando la media aritmética entre las puntuaciones de las distintas funcionalidades. En concreto las funcionalidades que se analizarán serán las siguientes:

- (a) **Intercambio de experimentos.** Esta función se encarga de analizar si es posible compartir los distintos experimentos desarrollados con otros científicos.

- (b) **Instalación automática de las dependencias necesarias.** En este caso el objetivo es evaluar si la herramienta instala de forma automática las dependencias necesarias para ejecutar cada experimento.
- (c) **Comprobación de experimentos repetidos.** Esta función evalúa si la herramienta es capaz de comprobar si alguna tarea del experimento a realizar ya ha sido ejecutada anteriormente y, por tanto, no es necesario repetirla.
- (d) **Aplicación de experimentos en el área de trabajo.** En este caso se evalúa si es posible ejecutar un experimento fuera del área de trabajo y posteriormente aplicarlo en el mismo.
- (e) **Interfaz gráfica.** Se evalúa si la herramienta a analizar provee una interfaz gráfica en la que visualizar los distintos experimentos.
- (f) **Visualización de gráficas.** Esta función evalúa si la herramienta a analizar permite visualizar gráficas de valores, como por ejemplo métricas.
- (g) **Búsqueda y comparación de experimentos.** Esta función evalúa si la herramienta permite buscar experimentos y compararlos entre todos los ejecutados a través de una API.
- (h) **Despliegue de modelos.** En este caso se pretende evaluar si la herramienta en cuestión despliega un modelo seleccionado.
- (i) **Almacén de modelos.** Esta función evalúa si la herramienta posee un *model registry* con el que realizar un seguimiento cronológico y de versionado de los modelos desarrollados.
- (j) **Creación de *pipelines*.** Se pretende evaluar si la herramienta permite crear *pipelines* y en ellos se comprueba en la ejecución de cada etapa las dependencias existentes con las anteriores etapas.

Debido a la importancia de esta dimensión se realizará una tabla comparativa, diferente a la de los criterios, en la que se indicará en verde si la herramienta asociada a cada experimento cumple la función por defecto (4 puntos si lo cumple muy bien, 3 puntos si lo cumple pero es más sencillo su uso en la otra herramienta), en amarillo si dicha funcionalidad se puede implementar desarrollando software por parte del usuario (2 puntos) y en rojo si esa funcionalidad no existe para dicha herramienta (0 puntos). Dicha tabla será completada en la sección de resultados de cada caso práctico.



## 4.4 Caso de uso del presente proyecto

En esta última sección se explicará en detalle la funcionalidad que se usará de cada herramienta en cada caso práctico.

### 1. Primer caso práctico: fase de desarrollo del modelo en un entorno de laboratorio o pruebas.

- **DVC:**
  - Seguimiento de los datos y definición de un almacenamiento remoto de DVC en la carpeta local `dvcstore` y git en [este repositorio de GitHub](#).
  - Creación de un *pipeline* que procese los datos, construya y valide el modelo de ML.
  - Creación y ejecución de experimentos definidos en la tabla 4.1 y comparación y compartición de aquellos que mejores métricas obtienen.
- **Airflow:**
  - Ejecución de cada tarea definida en el *pipeline* de DVC.
  - Descarga de datos del origen.
  - Comprobación sobre la creación del *pipeline* de DVC.

### 2. Segundo caso práctico: paso de un entorno de pruebas a producción.

- **MLflow:**
  - Utilización del módulo MLflow Projects para realizar las distintas etapas: descarga de datos, procesamiento de los mismos, entrenamiento del modelo y validación del mismo en cada uno de los experimentos propuestos, tabla 4.1.
  - Uso del módulo MLflow Tracking para obtener las mejores ejecuciones de los experimentos ejecutados por MLflow Projects.
  - Uso del módulo MLflow Model Registry para realizar el versionado de los mejores modelos seleccionados en el paso anterior.
  - Utilización del módulo MLflow Models para desplegar los mejores modelos a producción y realizar consultas.
  - Comparación de las métricas obtenidas por los modelos descargados de manera manual y gráfica.
- **Airflow:** Automatización en la ejecución secuencial de cada módulo indicado en el punto anterior.



## Capítulo 5

### Primer caso práctico

En el presente capítulo se detalla la metodología específica y resultados alcanzados en el desarrollo de este primer caso práctico. En primer lugar, se detallará a alto nivel el diseño empleado en este primer caso práctico. En segundo lugar, se explicará la implementación práctica para ejecutar el diseño planteado y se detallan algunas decisiones de diseño relevantes, y, por último, se analizarán los resultados alcanzados teniendo en cuenta las dimensiones planteadas en el capítulo 3.

El objetivo de este caso de uso es ilustrar cómo se deben implementar las fases de *Data* y en especial, *lab project*, figura 3.4, de manera automatizada. Con ello se logra la creación del modelo, que en las futuras etapas será empaquetado y desplegado a producción.

Este experimento utiliza las herramientas DVC y Airflow. La primera se encarga de la creación del *pipeline* en el que se incluyen las tareas de limpieza de datos o *featurization*, entrenamiento o *train* del modelo y evaluación o *evaluate* del mismo usando los datos de validación; creación de experimentos que modifican uno o varios parámetros; compartir esos experimentos y comparar sus métricas. Por otro lado, Airflow ejecuta las tareas descritas anteriormente de forma secuencial y automática.

#### 5.1 Diseño

Previo al desarrollo de la implementación es necesario llevar a cabo un diseño teórico sobre el desarrollo del experimento. En la figura 5.1 se adjunta un esquema simplificado del diseño llevado a cabo en este primer caso práctico.

La cinco etapas principales son las siguientes:

1. **Descarga de datos.** Este paso descarga los datos desde la plataforma Kalgge hasta el área de trabajo.

A continuación, se realiza con DVC un seguimiento de la carpeta descargada (paso 1.1), lo que provoca dos sucesos: creación del fichero `.dvc` y almacenamiento de dicha carpeta en la

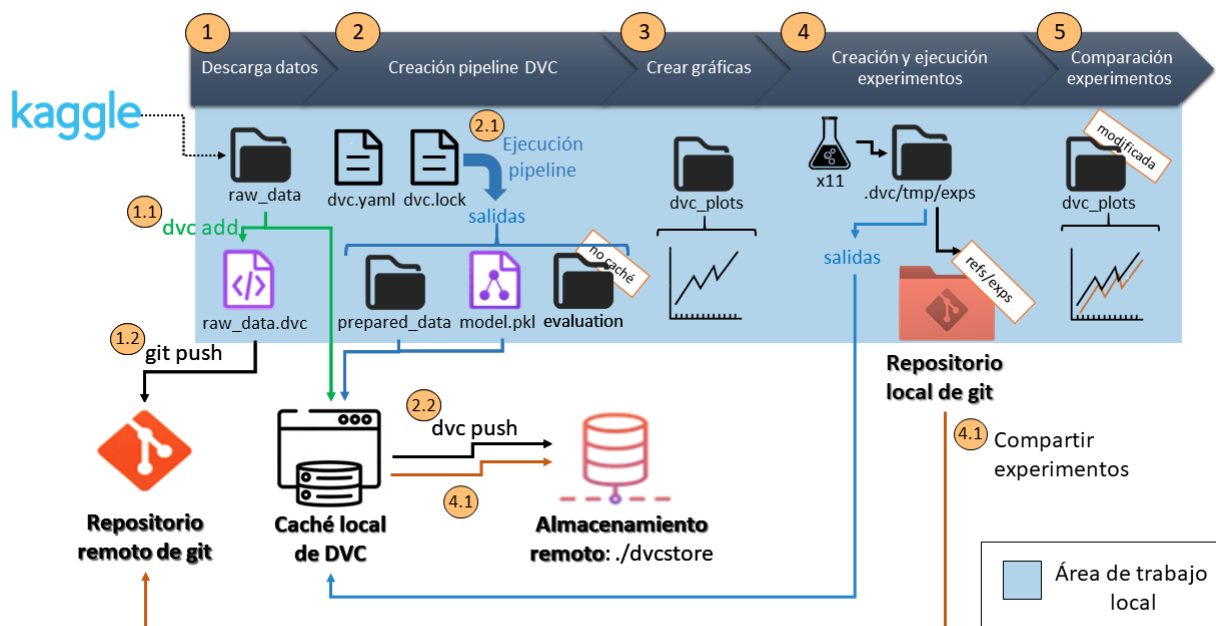


Figura 5.1: Diseño simplificado de DVC en el primer caso práctico. Se compone de cinco pasos principales: descarga de datos, que incluye seguirlos con DVC y git, creación y ejecución del *pipeline*, creación de gráficas, construcción y ejecución de 11 experimentos, los cuales serán compartidos a través del remoto de git y, por último, comparación de dichos experimentos.

caché que DVC mantiene localmente. Este fichero `.dvc` contiene metadatos de los archivos y directorios seguidos por DVC siendo de un tamaño reducido. Por último, este fichero `.dvc` puede ser almacenado en el repositorio local de git y después ser subido al repositorio remoto (paso 1.2).

2. **Creación del *pipeline* de DVC.** Los dos primeros pasos detallados a continuación son ejecutados secuencialmente por DVC.

Primeramente tiene lugar la propia construcción del *pipeline* lo que crea los ficheros `dvc.yaml` y `dvc.lock`. El primero de ellos contiene la declaración del *pipeline* de DVC incluyendo los parámetros (declarados junto a su valor en el fichero `params.yaml`), comandos a ejecutar, dependencias, así como las entradas y salidas de cada etapa, mientras que el segundo contiene las referencias internas que DVC asigna a cada fichero o directorio que sea la salida de alguna etapa definida en el fichero `dvc.yaml`.

El *pipeline* diseñado contiene las etapas de limpieza de datos, cuya salida son los datos procesados; entrenamiento del modelo, cuya salida es el modelo serializado y, por último, la etapa de validación, cuya salida son los ficheros de métricas (valor-F, precisión, exhaustividad, exactitud, valor del área bajo la curva *Receiver Operating Characteristic*, ROC, y curvas ROC y precisión-exhaustividad).

En el segundo paso, ilustrado en la figura anterior como 2.1, DVC ejecuta de manera automática el *pipeline* durante el proceso de su creación (utilizando los valores por defecto indicados en el fichero `params.yaml`), esto ocasiona que en el área de trabajo aparezcan las salidas correspondientes a cada etapa ejecutada, en concreto la carpeta `prepared_data`, el archivo `model.pkl` y la carpeta `evaluation`. DVC automáticamente sigue y almacena en la caché local estas salidas. Sin embargo, en este caso se ha indicado que la carpeta `evaluation` no sea almacenada en la caché, pues debido a su tamaño puede ser versionada por git.

Por último, paso 2.2 en la figura 5.1, es posible subir todos los archivos que se encuentren en la caché de DVC al almacenamiento remoto que esta herramienta permite configurar. En este caso se ha elegido que sea un almacenamiento local en una carpeta llamada `dvcstore`.

Cabe destacar que, por simplicidad, en la figura 5.1 no se ha representado que la carpeta `evaluation` y los archivos `dvc.yaml` y `dvc.lock` deben ser subidos al repositorio remoto de git con el fin de que el *pipeline* y las salidas de la etapa de evaluación sean versionados. Asimismo, por simplicidad, tampoco se ha incluido la caché local dentro del área azul, a pesar de que pertenece exclusivamente al área de trabajo.

3. **Creación de las gráficas** correspondientes al *pipeline* que acaba de ser ejecutado. Para ello DVC genera una carpeta, `dvc_plots`, que contiene todas las figuras generadas, como archivos `html`. En este experimento se generará la matriz de confusión y la evolución de las métricas precisión y exhaustividad del modelo desarrollado. La intención de incluir este paso es para ilustrar la capacidad de DVC a la hora de realizar gráficas sobre los resultados presentes en el área de trabajo, lo cual es distinto a la comparación de experimentos (fuera del área de trabajo).
4. **Creación y ejecución de experimentos.** La creación y gestión de experimentos es uno de los aspectos más destacables de DVC, ya que permite ejecutarlos fuera del área de trabajo con el fin de no sobrescribir ningún archivo y compartirlos de manera sencilla.

En este caso se han implementado 11 experimentos, tabla 4.1, los cuales no serán ejecutados directamente en el área de trabajo, ya que hemos configurado DVC para que los ejecute en un directorio temporal que se encuentra en `.dvc/tmp/exps`. A continuación, estos experimentos son almacenados por DVC automáticamente en una referencia personalizada de git en el repositorio local de código [34], mientras que sus salidas son almacenadas automáticamente por DVC en la caché local.

Por último, se comparten solo 4 de estos experimentos (paso 4.1). Lo que ocasiona dos sucesos: DVC almacena los experimentos en la referencia personalizada llamada `exps` del repositorio remoto y las salidas de esos experimentos son guardadas en el almacenamiento remoto configurado, en este caso `./dvcstore`.

5. **Comparación de experimentos.** En este último paso se crean las gráficas necesarias para visualizar las diferencias en la evolución de las métricas y matrices de confusión. De nuevo DVC las almacena en la carpeta `dvc_plots`.

## 5.2 Implementación

Para llevar a cabo la implementación de este experimento primero se ha planteado la estructura del DAG de Airflow, figura 5.2, que incluye los operadores: `BashOperator`, `EmailOperator`, `PythonBranchOperator` y `PythonOperator`. A continuación, en cada tarea de Airflow se han integrado los pasos detallados en la figura 5.1. Sin embargo, estas tareas de Airflow no tienen por qué coincidir con los pasos detallados en dicha figura.

Cabe destacar que DVC es una herramienta que provee una API de Python muy escasa, ya que su principalmente se realiza a través de comandos, de manera análoga a git. Por ello, la mayoría de tareas del DAG de Airflow de este caso práctico son `BashOperator`.

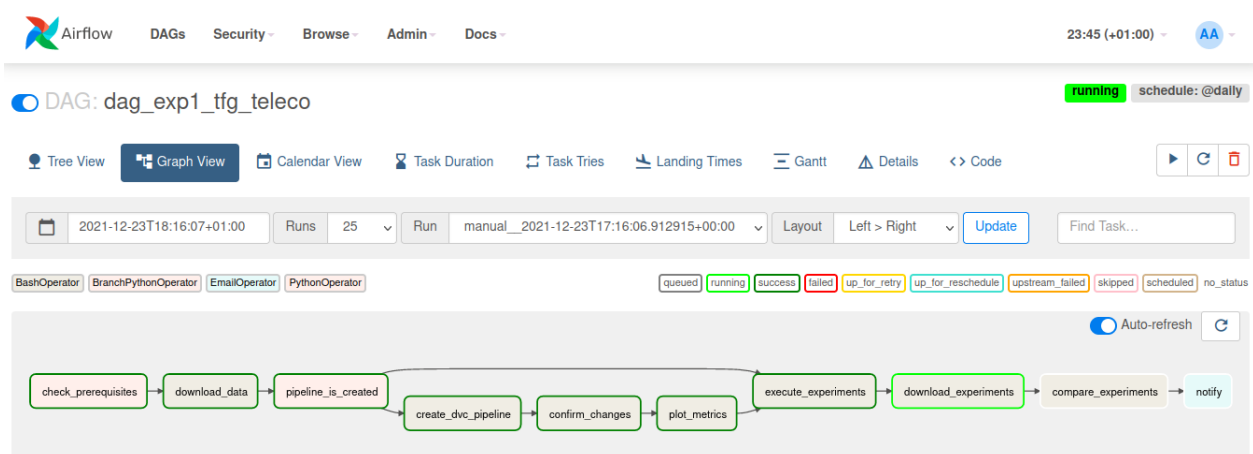


Figura 5.2: Captura de pantalla del DAG de Airflow del primer caso práctico. Como se puede ver el flujo de trabajo ha tomado la rama inferior, la cual es ejecutada si el *pipeline* de DVC no ha sido creado con anterioridad.

1. **Comprobación de prerequisites** (`PythonOperator`). Tal y como se indicó en la subsección 4.2.1, el objetivo de esta tarea es comprobar que todos los requisitos necesarios para ejecutar correctamente este caso práctico se cumplen. Algunos de ellos incluyen la declaración de las variables de Airflow, instalación de DVC y de los paquetes necesarios.
2. **Descarga de datasets** (`BashOperator`). Esta tarea descarga los datos de la plataforma de Kaggle, para más información se dirige a la subsección 4.1.1.
3. **Comprobación sobre la creación del *pipeline* de DVC** (`PythonBranchOperator`). Para ello se comprueba si los archivos `dvc.yaml` y `dvc.lock` se encuentran en el directorio de trabajo. Si ambos archivos existen en el área de trabajo el flujo de trabajo fluye por la rama superior saltando a la tarea ejecutar experimentos, mientras que si uno de ellos no se encuentra,

se ejecuta la rama inferior saltando a la tarea creación del *pipeline* de DVC. Por tanto, las tareas creación del *pipeline* de DVC, confirmación de cambios y generación de gráficas solo se ejecutan en la rama inferior.

4. **Creación del *pipeline* de DVC (BashOperator).** Esta tarea se encarga de crear el *pipeline*, el cual se refleja en el archivo `dvc.yaml` y que genera el siguiente DAG figura 5.3. Como se puede ver, esta figura también ilustra las dependencias existentes entre las tareas de DVC.

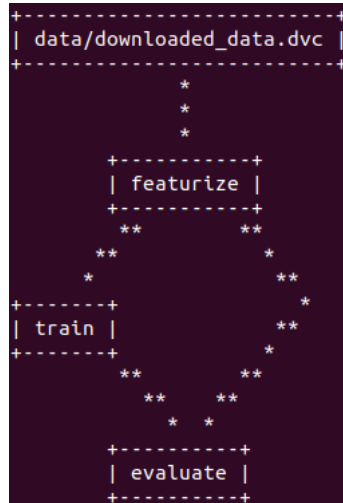


Figura 5.3: Captura de pantalla del DAG de DVC del primer caso práctico. Este DAG describe un *pipeline* que incluye las etapas de obtención de características, entrenamiento y evaluación del modelo creado. Como se puede ver la etapa *featurize* depende de los datos en bruto descargados de Kaggle y sus salidas son los datos de entrenamiento y validación. La etapa de entrenamiento tiene una dependencia con las características extraídas en el caso anterior, pues genera aquellos datos con los que será entrenado el modelo. Y, por último, la tarea de evaluación depende del modelo generado en la fase de entrenamiento para el cual se evaluará su rendimiento con los datos de validación, generados en la etapa de *featurize*.

5. **Confirmación de cambios (BashOperator).** Se encarga de seguir con DVC y git los archivos correspondientes. En el repositorio remoto de código se encuentran los ficheros `params.yaml`, `dvc.lock`, `dvc.yaml` y `downloaded_data.dvc` y la carpeta `src/`, mientras que en la caché local y almacenamiento remoto de DVC se encuentran el fichero `model.pkl` y las carpetas `data/prepared_data` y `data/downloaded_data`.

De esta forma se está subiendo indirectamente al repositorio remoto de código la primera ejecución del *pipeline*, lo cual se puede considerar el primer experimento desarrollado en este primer caso.

6. **Generación de gráficas (BashOperator).** Esta tarea se encarga de mostrar las métricas, figura 5.4, y crear las gráficas asociadas a ellas que han tenido lugar al ejecutar la etapa de evaluación del *pipeline* de DVC. Algunos ejemplos son la matriz de confusión, figura 5.5, y la

evolución de la precisión del modelo, figura 5.6. Esta tarea finaliza con la subida al repositorio remoto de la carpeta `evaluation`.

Experiment	Created	scores.json:precision	scores.json:recall	scores.json:f1	scores.json:accuracy	scores.json:roc_auc
workspace	-	0.94118	0.97959	0.96	0.96	0.9952
master	03:07 PM	0.94118	0.97959	0.96	0.96	0.9952

Figura 5.4: Métricas analizadas en el primer caso práctico: valor-F, exhaustividad, ROC, precisión y exactitud. Estas métricas han sido generadas por el experimento nombrado como `cwurData`.

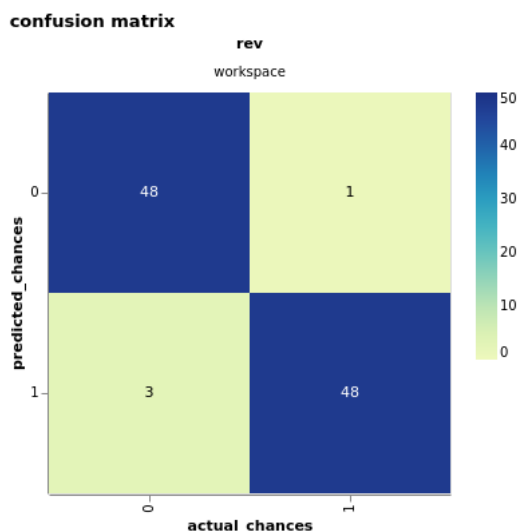


Figura 5.5: Matriz de confusión generada por DVC. El eje horizontal representa los valores reales. Mientras que, en el eje vertical se encuentran los valores predichos por el modelo desarrollado ante los datos de validación. Esta matriz de confusión ha sido generada por el experimento nombrado como `cwurData`, tabla 4.1

- Creación y ejecución de experimentos (BashOperator).** Esta tarea se encarga de crear y ejecutar 11 experimentos en una ruta temporal local. A continuación, se suben 4 de ellos al repositorio remoto de código (Cmr300, Sf3, Snest64 y Tmr300) y, de cara a la preparación de la siguiente tarea, se borran todos ellos del área de trabajo. Se han elegido estos experimentos porque son los que mejores métricas obtenían y para mostrar diversos parámetros seleccionados.
- Descarga de experimentos (BashOperator).** Con el fin de ilustrar la compartición de experimentos que provee DVC, se descargan los cuatro experimentos subidos al repositorio remoto de código en el área de trabajo.
- Comparación de experimentos (BashOperator).** Se comparan los experimentos descargados entre sí, dos a dos, y con el resultante de la primera ejecución del *pipeline*. Para ello, se realizan gráficas comparativas de las métricas y se muestran las diferencias entre sus parámetros, figura 5.7.



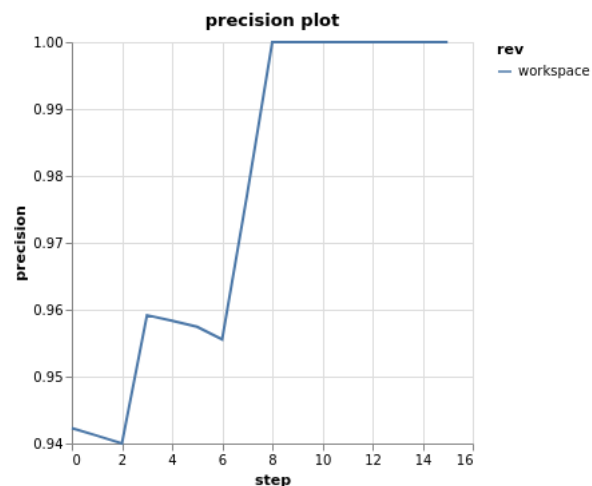


Figura 5.6: Evolución de la métrica precisión generada por DVC en la fase de evaluación. El eje horizontal representa el paso que el modelo elige para evaluar esta métrica, mientras que en el eje vertical se encuentra el valor de la precisión ante los datos de validación. Esta gráfica ha sido generada por el experimento nombrado como `cwurData`, tabla 4.1

Por último, se realiza una limpieza para preparar los repositorios y el área de trabajo para futuras ejecuciones: esta tarea elimina del repositorio remoto de código y del directorio de trabajo todos los experimentos, de forma que en futuras ejecuciones del DAG no haya experimentos anteriores. Asimismo, se elimina de la caché local aquellos elementos que no se encuentren en el área de trabajo.

10. **Notificación por correo electrónico** (`EmailOperator`). Esta tarea notifica por correo electrónico la correcta ejecución del DAG de Airflow.

### 5.2.1 Consideraciones de implementación

Esta subsección tiene por objetivo clarificar y justificar algunas de las decisiones clave llevadas a cabo en el proceso de implementación de este caso.

1. **Variables necesarias para la ejecución del caso como variables de Airflow.** Se ha elegido que las variables generales necesarias para la ejecución de este caso como variables integradas en Airflow. Tal y como se indicará en la próxima sección, esta decisión no es escalable. Sin embargo, es útil para ilustrar en profundidad las posibilidades de Airflow en cuanto a la gestión en la creación y acceso a estas variables.
2. **Creación de la tarea `pipeline_is_created` como `PythonBranchOperator`.** En lugar de haber seleccionado este tipo de tarea, que es más compleja de entender e implementar, DVC permite sobrescribir la escritura de un `pipeline` ya existente mediante comandos, o simplemente mediante edición manual. Sin embargo, se ha elegido esta otra implementación persiguiendo un

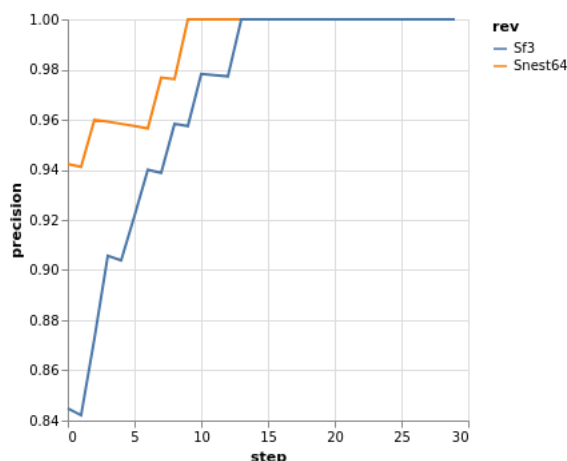


Figura 5.7: Evolución de la métrica precisión en los experimentos Sf3 y Snest64 generado por DVC. El eje horizontal representa el paso que el modelo elige para evaluar esta métrica. Mientras que, en el eje vertical se encuentra el valor de la precisión ante los datos de validación.

objetivo doble: profundizar en las posibilidades que ofrece Airflow en la creación de tareas y evitar construir el mismo *pipeline* periódicamente.

3. **Reintento de ejecución de la tarea `create_dvc_pipeline` ante fallo.** Esta tarea del DAG de Airflow se ha configurado para que, si su ejecución ha fallado, se notifique por correo electrónico y se intente ejecutar de nuevo 2 veces con un intervalo entre ellas de 1 minuto. De esta forma, de nuevo, se profundiza en la gestión que Airflow lleva a cabo en la ejecución de las tareas.
4. **Descarga de datos en una tarea de Airflow y no como una etapa del *pipeline* de DVC.** Es importante tener en cuenta que DVC por defecto ejecuta exclusivamente aquellas tareas que hayan sufrido alguna modificación en las dependencias marcadas en el fichero `dvc.yaml`. Estas suelen ser parámetros o los propios ficheros fuente. Sin embargo, DVC permite configurar que determinadas etapas del *pipeline* sean siempre ejecutadas.

La descarga de datos no implica ningún cambio en ninguna dependencia o parámetro, únicamente se lleva a cabo para obtener periódicamente los datos actualizados. Por ello y para ilustrar cómo DVC genera el fichero `.dvc` (ya que DVC no crea este fichero si es una salida de una etapa del *pipeline*), se ha elegido implementarlo como una tarea de Airflow.

5. **Creación del *pipeline* de DVC vía comandos.** DVC ofrece la posibilidad de crear el fichero `dvc.yaml` editándolo manualmente o mediante comandos. En este caso se ha elegido esta última opción persiguiendo un triple objetivo: implementación más sencilla, solución del mismo problema en el próximo experimento e ilustrar las opciones que ofrece Airflow al permitir dividir flujos de trabajo mediante una tarea del tipo `PythonBranchOperator`.

Esta elección acarrea problemas de escalabilidad, pues si se desea cambiar la ruta de los archivos de salida o dependencias de alguna etapa del *pipeline*, es necesario modificar manualmente todas sus apariciones en el fichero `dvc.yaml`.

6. **Selección de ejecución diaria del DAG.** Se ha seleccionado este intervalo de tiempo para simular la llegada de nuevos datos con los que alimentar el modelo, tal y como sucede en las organizaciones. Por ello, y para no incurrir en demasiadas ejecuciones del DAG, se ha elegido un intervalo diario.

7. **Elección de almacenamiento remoto en una ruta local.** DVC ofrece la posibilidad de configurar un almacenamiento remoto en plataformas como Google Cloud o AWS, entre otros.

Inicialmente se eligió Google Drive ya que es una de las pocas opciones gratuitas. Sin embargo, DVC requiere autenticarse al subir archivos a la plataforma. Airflow durante la ejecución de cada tarea no ofrece la posibilidad de pedir datos al usuario, como una contraseña, por lo que se ha elegido un almacenamiento local, la carpeta `dvcstore`, que no requiere autenticación de ningún tipo.

8. **Elección de ejecución de experimentos en una ruta temporal.** DVC por defecto ejecuta los experimentos en el área de trabajo. Sin embargo, con el fin de probar varias de las posibilidades que ofrece se han probado las siguientes opciones:

- **Encolado de experimentos.** DVC permite declarar experimentos y ejecutarlos más adelante en un directorio temporal. Para ello, almacena estos experimentos en `git stash`. Estas son ramas que almacenan el directorio de trabajo en un entorno separado donde realizar pruebas.

Sin embargo, al tratar de ejecutar las tareas encoladas, DVC no funciona en todas las ocasiones. De hecho, en su propia documentación indica que la ejecución paralela de experimentos puede ser inestable en algunas ocasiones.

- **Ejecución en un directorio temporal.** Con el fin de no sobrescribir el área de trabajo, se han ejecutado en un directorio temporal local. De esta forma es posible comparar esos experimentos con el área de trabajo. Además, se puede seleccionar el experimento deseado y aplicarlo en el área de trabajo para después confirmarlo en git y descartar el resto.

9. **Versión 2.7.3 de DVC.** Los experimentos desarrollados en este caso práctico fueron ejecutados en la versión de DVC 2.7.3. Sin embargo, al actualizar a la última versión disponible, 2.8.3, surgieron errores internos de DVC a la hora de mostrar los experimentos llevados a cabo en el proyecto.

## 5.3 Resultados

DVC es una herramienta intuitiva y muy cómoda a la hora de ejecutar experimentos localmente y compartirlos con otros científicos. Asimismo, hace muy sencillo el hecho de compartir y versionar grandes volúmenes de información en un almacenamiento remoto. Sin embargo, es una herramienta en la que a veces no se encuentra suficiente ayuda en línea.

Por otro lado, Airflow, al tratarse de una herramienta con interfaz gráfica, hace la ejecución de flujos de trabajo mucho más cómoda y manejable. No solo gracias a la división de problemas complejos en tareas, sino también a la visualización de la ejecución de cada tarea, figura 5.2.

A continuación se analizarán las dimensiones seleccionadas en la sección 4.3:

1. **Facilidad de uso.** DVC es una herramienta bastante cómoda de utilizar. Con ejemplos en cada sección de la documentación, tutoriales y vídeos sobre sus principales características. Además, gracias al hecho de proporcionar una experiencia similar a la de git, se torna bastante intuitiva. Sin embargo, se trata de una herramienta inmadura y con una comunidad de usuarios pequeña, por lo que resolver determinados problemas asociados a ella se hace complejo.

Por otro lado, Airflow también proporciona documentación en línea, organizada según la versión, con algunos tutoriales aunque con ejemplos escasos. A pesar de ello, su uso está mucho más extendido y hay multitud de ejemplos y tutoriales que los usuarios particulares proporcionan.

Por último, Airflow no permite hacer uso de todo el potencial de DVC. Se trata del caso de subir información al almacenamiento remoto en, por ejemplo, Google Drive. DVC permite llevarlo a cabo si la sesión no ha expirado, si no, es necesario introducir un código o contraseña. Debido a que Airflow no permite interactuar con el usuario ni pedirle información durante la ejecución de las tareas, esta funcionalidad de DVC debe ser llevada a cabo manualmente y de manera externa a Airflow.

2. **Escalabilidad.** Tal y como se ha ilustrado en el desarrollo de este caso, DVC obliga a crear un fichero `params.yaml` en el que se detallan el valor de los parámetros de los que dependen las etapas del *pipeline*. Gracias a ello, si se desea añadir, borrar o modificar alguno de los parámetros únicamente es necesario acceder a este fichero. Además, Python ofrece un paquete que maneja fácilmente el acceso a ficheros YAML.

Otro inconveniente que plantea DVC es la falta de una API a través de la que consultar los distintos experimentos y elegir el más conveniente. De esta forma se podría haber consultado mediante código cuáles de los distintos experimentos ofrecían las mejores métricas para compartirlos, en lugar de hacer una consulta manual.

Por último y tal y como se ha indicado anteriormente, el planteamiento de este experimento incluía la declaración de las variables necesarias para su ejecución en Airflow. Esta solución es poco escalable en el ámbito empresarial, y para ponerlo de manifiesto se plantea el siguiente ejemplo: supongamos que cada DAG necesita hacer uso de tres variables para poder ejecutar y todas ellas son declaradas en Airflow. Supongamos también un total de 20 ficheros DAG. En total habría declaradas 60 variables en Airflow, y, a pesar de que se muestran en la interfaz gráfica y se puede acceder a ellas fácilmente, no es manejable ni gestionable tal cantidad de

variables. Esta decisión se tomó así para ilustrar esta capacidad de Airflow. En el próximo caso práctico se plantea una solución a este problema.

3. **Requisitos software.** DVC no proporciona una instalación vía Pip o Anaconda, si no que las opciones que provee no son tan cómodas o sencillas comparadas con la instalación de MLflow o Airflow.

Por otro lado, la integración entre ambas herramientas no ha sido compleja de realizar, ya que no ha habido problemas de versiones en las dependencias internas de cada una de ellas.

4. **Capacidades.** DVC es una herramienta que se encuentra muy presente en la fase más experimental del desarrollo de un modelo, por ello, no provee soluciones para el despliegue y gestión de modelos a producción. En la tabla 5.1 se muestran las funcionalidades que esta herramienta provee.

Asimismo es conveniente destacar que la API Python que ofrece es escasa, y en el caso de la gestión de los experimentos inexistente. De nuevo, hubiera sido útil poder comparar los experimentos a través de una API con el fin de, por ejemplo, compartir los mejores de ellos o aplicarlos en el área de trabajo.

Por tanto, de este primer caso práctico se concluye:

- DVC es una herramienta de manejo sencillo, intuitiva y muy adecuada para la ejecución y compartición de experimentos y grandes volúmenes de datos.
- DVC puede ser utilizada para versionar ficheros de código o datos de gran tamaño, mientras que git se encarga de versionar la declaración de los *pipelines* de DVC y ficheros de metadatos.
- DVC presenta algunos fallos de software al tratarse de una herramienta poco madura, además, la cantidad de contenidos y ejemplos desarrollados por la comunidad es reducida.
- Airflow es una herramienta muy adecuada para automatizar y modularizar procesos, simplemente declarándolo en un fichero.
- La compatibilidad entre Airflow y DVC es posible aunque en determinados casos no se puede hacer uso de toda la funcionalidad que provee DVC.
- Experimento poco escalable, debido a la elección de usar variables presentes en Airflow y a la carencia de DVC de proporcionar una API Python más completa, a través de la cual seleccionar los mejores experimentos.

Función	Evaluación para DVC	Puntuación
Función 1	Permite compartir experimentos usando los remotos de git y el almacenamiento remoto configurado	4
Función 2	No instala automáticamente los paquetes necesarios para ejecutar los experimentos	0
Función 3	Comprueba en su caché si alguna etapa del experimento ya ha sido ejecutada, en caso afirmativo no la vuelve a repetir	4
Función 4	Permite ejecutar en una carpeta temporal los experimentos y aplicar en el área de trabajo los deseados	4
Función 5	No proporciona interfaz gráfica	0
Función 6	Permite visualizar gráficas en formato <i>HTML</i> , previamente guardando los valores en un fichero	3
Función 7	No permite buscar los experimentos con mejores métricas, para por ejemplo aplicarlos en el área de trabajo, ya que no ofrece una API Python para ello	0
Función 8	No despliega modelos a producción	0
Función 9	No ofrece <i>model registry</i>	0
Función 10	Sí ofrece la posibilidad de crear un <i>pipeline</i>	4
Promedio		1.9

Tabla 5.1: Tabla que muestra las funcionalidades que provee la herramienta DVC. En color verde se muestra si dicha herramienta implementa la funcionalidad por defecto (puntuación 3 y 4), en amarillo si el usuario puede implementar código para conseguir obtener esa funcionalidad (puntuación 2) y en color rojo si dicha funcionalidad no existe para esa herramienta (puntuación 0 y 1).

## Capítulo 6

### Segundo caso práctico

En este capítulo se detalla el desarrollo llevado a cabo en el segundo caso práctico del presente proyecto. Se sigue una estructura similar a la planteada en el caso anterior, en primer lugar se realiza el diseño del experimento a realizar en el que se indica la configuración elegida de MLflow, a continuación se explica la implementación llevada a cabo para lograr ese diseño, y, por último, se analizan los resultados obtenidos según los criterios explicados en la sección 4.3.

El objetivo de este caso es doble: ilustrar cómo deben ser implementadas de manera simplificada las fases de *lab project*, *interface* y *production project* detalladas en la figura 4.1. Estas fases se desarrollarán de manera automatizada y se buscará corregir los problemas relativos a la implementación planteados en el caso anterior, subsección 5.2.1. En la implementación se crea el modelo, se empaqueta (en este caso solo se empaqueta el modelo serializado, un fichero que lo describe y los ficheros que detallan las dependencias necesarias para su uso) posibilitando simular su despliegue a producción. Por tanto, se logra la creación del modelo, su empaquetamiento y simular su despliegue a producción.

Este caso práctico utiliza las herramientas MLflow y Airflow. La primera se encarga de crear un *pipeline* y llevar a cabo 12 experimentos, ver tabla 4.1. Para ello, se hará uso del módulo MLflow Projects, en el que se incluyen las tareas de descarga de datos, limpieza de los mismos y entrenamiento del modelo. A continuación, se eligen los modelos que mejor rendimiento han proporcionado (según el *dataset* considerado) en los datos de validación, y se almacenan en el *model registry*, haciendo uso del módulo MLflow Model Registry. Por último, cada modelo es desplegado de una forma diferente y se realizan predicciones usando los datos de pruebas, estas son después comparadas gráficamente usando la UI de MLflow. La segunda herramienta, Airflow, se encarga de ejecutar de forma automatizada y secuencial cada una de las tareas de anteriores.

## 6.1 Diseño

Previo al desarrollo de la implementación de este caso es preciso, en primer lugar, elegir la forma de almacenamiento entre las que ofrece MLflow y, en segundo lugar, plantear el diseño teórico a seguir en este caso práctico. En las figuras 6.1 y 6.2 se adjuntan el almacenamiento seleccionado y el diseño planteado, respectivamente.

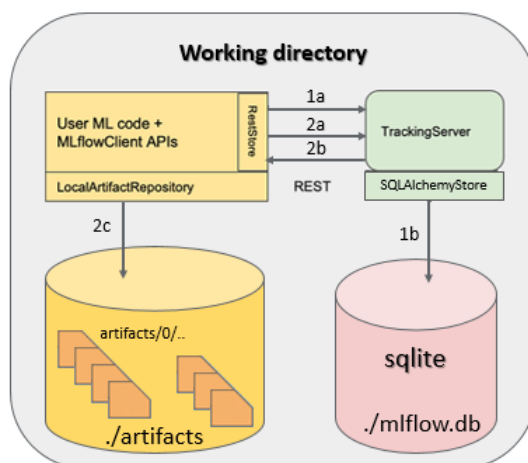


Figura 6.1: Configuración de almacenamiento elegida en MLflow. Se ha elegido como almacén de artefactos una carpeta local, llamada *artifacts*, y como *backend store* una base de datos SQLITE llamada *mlflow.db*. En el texto principal se detallan la secuencia de peticiones y respuestas representadas. Figura adaptada de [36, escenarios 3 y 4].

Tal y como se puede visualizar en la figura 6.1, MLflow distingue dos tipos de almacenamiento: *backend store* y almacén de artefactos. El primero almacena **entidades** que incluyen métricas, parámetros, etiquetas o incluso los modelos registrados en el *model registry*, lo que requiere almacenar sus versiones, notas, esquemas y *stage*, entre otros. En cambio el almacén de artefactos se encarga de almacenar ficheros, modelos o imágenes, entre otros.

MLflow en ambos casos elige por defecto una carpeta que crea en el directorio de trabajo llamada `mlruns`. Sin embargo, permite configurar un almacenamiento remoto de artefactos que puede ser alojado en plataformas como AWS, Google Cloud, Azure o en una carpeta local. En este caso se ha elegido esta última. Por otro lado, el *backend store* debe ser alojado como una carpeta o base de datos del sistema, en este caso se ha elegido una base de datos local SQLITE.

Para llevar a cabo el almacenamiento descrito, MLflow interactúa con el servidor de seguimiento a través de peticiones REST, figura 6.1:

- Almacenamiento de entidades. El cliente MLflow, el cual puede ser creado mediante una API Python, crea una instancia de `RestStore` para enviar una petición REST al servidor de seguimiento con el fin de almacenar entidades (1a). Este servidor crea una instancia de `SQLAlchemyStore` para almacenar las entidades en la base de datos (1b).



- Almacenamiento de artefactos. El cliente MLflow usa una instancia de RestStore para enviar una petición con el fin de obtener la localización del almacén de artefactos (2a). El servidor de seguimiento responde con la ruta local a este almacén (2b). Por último, el cliente de MLflow crea una instancia de LocalArtifactRepository y salva los artefactos deseados en la ruta local indicada por el servidor de seguimiento (2c).

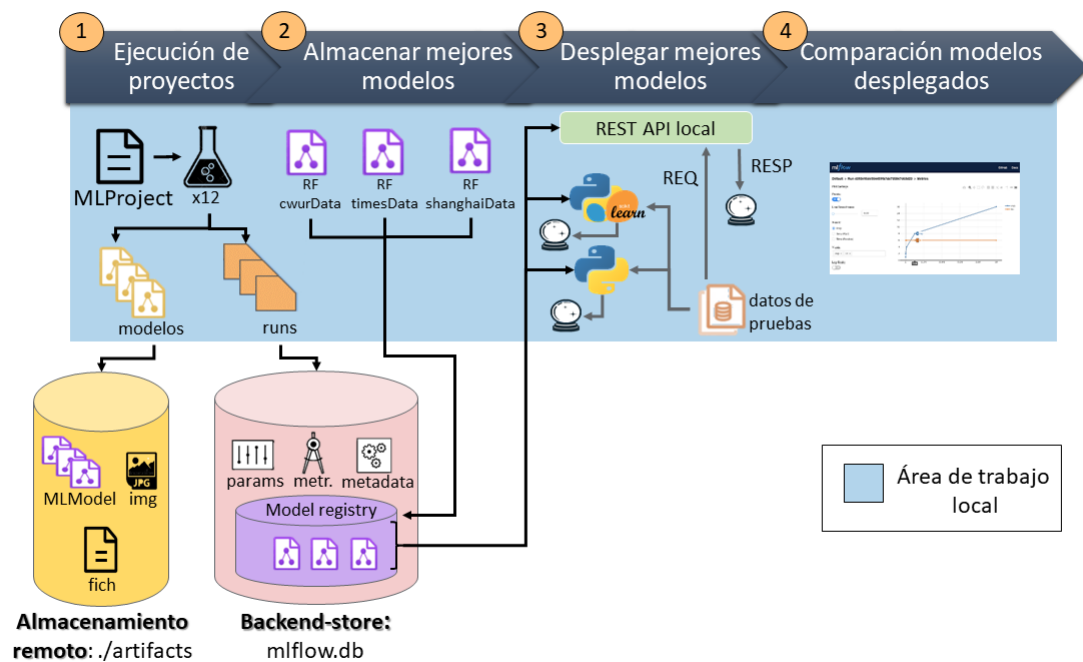


Figura 6.2: Diseño simplificado llevado a cabo con MLflow en el segundo caso práctico. Se compone de cuatro pasos principales: ejecución del proyecto, para ello se ejecuta el *pipeline* diseñado en el fichero *MLproject* 12 veces según los experimentos planteados, tabla 4.1; se almacenan en el *model registry* los mejores modelos obtenidos en los experimentos anteriores; se despliegan de tres formas distintas y se realizan predicciones, y, por último, se comparan usando la UI de MLflow.

Por otro lado, el diseño planteado que utiliza el almacenamiento descrito anteriormente incluye las siguientes etapas principales, figura 6.2:

1. **Ejecución de experimentos.** Para ello, se ejecuta el punto de entrada “main” del fichero `MLproject`, el cual ejecuta el programa `src/main.py`, que define el *pipeline* que incluye todos los pasos a realizar. Este *pipeline* se ejecuta 12 veces según los parámetros planteados en la tabla 4.1. En concreto, se compone de las siguientes etapas:
  - Descarga de los datos desde la plataforma Kaggle.
  - Limpieza y obtención de características de los datos descargados. La salida de esta etapa es almacenada por MLflow Tracking en el almacén de artefactos.

- Entrenamiento del modelo, el cual es almacenado usando el módulo MLflow Models en el almacén de artefactos, mientras que sus métricas (precisión, exactitud, exhaustividad y valor-F) y parámetros son almacenados por MLflow Tracking en el *backend store*.

El entrenamiento del modelo se lleva a cabo con los datos de entrenamiento, mientras que las métricas almacenadas son obtenidas utilizando los datos de validación.

2. **Registrar los mejores modelos.** El objetivo es buscar los mejores modelos entre las *runs* ejecutadas en el paso anterior, según el *dataset* empleado (CWUR, THE o Shangái; ver sección 4.1) y almacenarlos en el *model registry*.

Por simplicidad, en la figura 6.2 no se ha indicado que los mejores modelos se buscan entre aquellos entrenados en la etapa anterior.

3. **Despliegue de los mejores modelos.** En esta etapa se despliegan los modelos almacenados en el *model registry* y se realizan predicciones usando los datos de pruebas. Para ello, se emplean las tres formas que MLflow provee: REST API local que recibe los datos sobre los que predecir en formato JSON y la predicción realizada se encuentra en el mismo formato, una función Python (MLflow PyFunc) según el *flavour* del modelo, sklearn en este caso, o mediante una función Python genérica agnóstica de la librería que haya creado el modelo (MLflow PyFunc), estas dos últimas reciben y devuelven datos representados por la librería Pandas.

De nuevo, por simplicidad y de cara a la siguiente etapa no se han representado que las métricas asociadas a estas predicciones han sido almacenadas en una *run* nueva.

4. **Comparación de los mejores modelos desplegados,** para ello, se comparan manualmente desde la interfaz gráfica que ofrece MLflow.

## 6.2 Implementación

Para llevar a cabo la implementación de este experimento primero se ha planteado la estructura del DAG de Airflow, figura 6.3, que incluye los operadores BashOperator, PythonOperator e EmailOperator. A continuación, en cada tarea de Airflow se han integrado los pasos detallados en la figura 6.2.

1. **Comprobación de prerequisites** (PythonOperator). Tal y como se indicó en la subsección 4.2.1, el objetivo de esta tarea es comprobar que todos los requisitos necesarios para la ejecución de este caso práctico se cumplen. Algunos de ellos incluyen haber lanzado el servidor de seguimiento o comprobar si es válida la ruta en la que llevar a cabo la ejecución.
2. **Ejecución del proyecto** (BashOperator). Esta tarea se encarga de la ejecución de los 12 experimentos detallados en la tabla 4.1. Para ello, se hace uso del módulo MLflow Projects el cual instala automáticamente las dependencias necesarias para su ejecución detalladas en el fichero `my_env.yaml`.

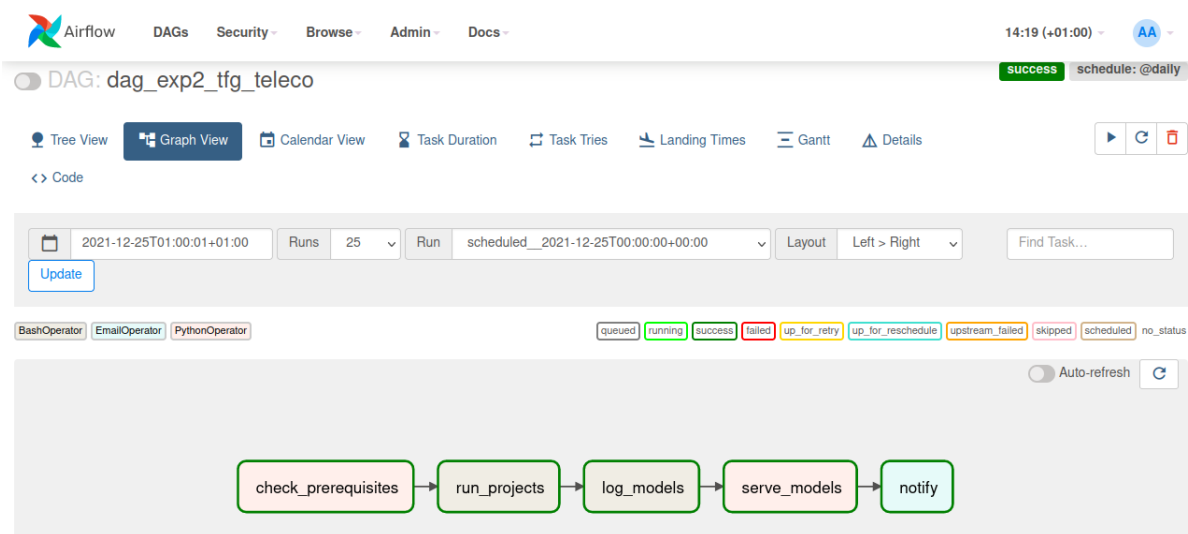


Figura 6.3: Captura de pantalla del DAG del segundo caso práctico. Se compone de cinco tareas, cuya funcionalidad se describe en el texto principal.

Se utiliza el módulo MLflow Tracking para almacenar métricas de cada experimento y ficheros obtenidos en las etapas de descarga y extracción de características, mientras que se usa el módulo MLflow Models para almacenar el modelo entrenado. Éste guarda los modelos en el almacén de artefactos, para ello crea los siguientes ficheros:

- Fichero `MLmodel` que contiene datos como la librería o *flavour* usado para generar este modelo, fecha de creación, esquema de los datos de entrada y salida y *run* que ha generado este modelo.
- Fichero `model.pkl` que contiene el modelo serializado.
- Fichero `conda.yaml` que contiene las dependencias de Anaconda necesarias para utilizar este modelo en otros entornos.
- Fichero `requirements.txt` que contiene las dependencias Python necesarias para utilizar este modelo en otros entornos.

Tras la ejecución de los experimentos, la UI de MLflow contendría las siguientes *runs*, figura 6.4.

3. **Almacenamiento de los mejores modelos (BashOperator).** Gracias a la API Python de MLflow es posible obtener todas las *runs* ejecutadas en el paso anterior y almacenar los modelos asociados a ellas con mejores métricas. Estos modelos son los asociados a los experimentos `cwurData`, `Snest64` y `timesData`, tabla 4.1.

A continuación, estos modelos son almacenados en el *model registry* usando el módulo MLflow Model Registry, figuras 6.5 y 6.6.

Experiments

Search Experiments

Default

**RF cwurData**

RF timesData

RF shanghaiData

Experiment ID: 1

Artifact Location: ./artifacts/1

Notes

Showing 16 matching runs

Refresh Compare Delete Download CSV

Columns

metrics.rmse < 1 and params.model = "tr...

Search Filter Clear

Start Time	Run Name	User	Source	Version	Models	Metrics	Parameters
6 days ago	Best model	meri	log_best_model	4f5a5	-	0.96 0.96 0.941 0.98	RF ...
6 days ago	-	meri	MLOps-Evaluation	b83d0	-	- - - -	-
6 days ago	-	meri	MLOps-Evaluation	b83d0	-	- - - -	-
6 days ago	-	meri	MLOps-Evaluation	b83d0	-	- - - -	-
6 days ago	-	meri	MLOps-Evaluation	ca7b9	-	- - - -	-
6 days ago	-	meri	main.py	ca7b9	RFClass-cw.../1	0.96 0.96 0.941 0.98	-
6 days ago	-	meri	main.py	ca7b9	-	- - - -	-
6 days ago	-	meri	main.py	ca7b9	-	- - - -	-

Figura 6.4: Captura de pantalla de las *runs* del segundo caso práctico. La interfaz se ha organizado según el *dataset* considerado, en este caso se muestran las cinco ejecuciones asociadas a los datos CWUR, cuatro de ellas debidas a la ejecución del fichero *MLproject* y una correspondiente a la búsqueda del mejor modelo.

4. **Servir modelos** (PythonOperator). Esta tarea pretende simular el despliegue de los modelos almacenados en el *model registry*. En este caso se ha elegido un despliegue en una REST API local, como una función Python del *flavour* de *sklearn* y como una función Python genérica, con la intención de ilustrar las tres posibilidades que ofrece MLflow para este propósito.

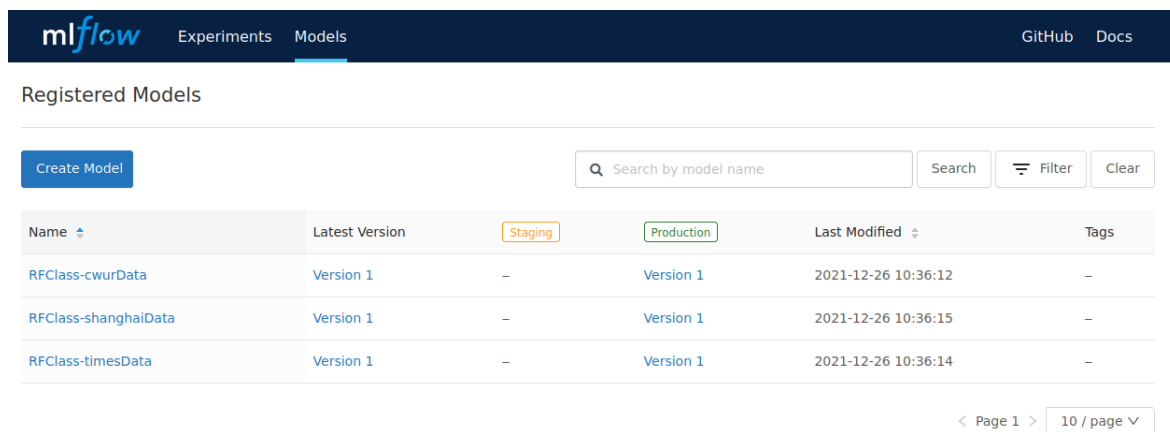
A continuación, se han almacenado las métricas obtenidas usando el módulo MLflow Tracking utilizando los datos de pruebas.

Por último y de manera externa a Airflow, se ha comparado manualmente el resultado obtenido por los modelos desplegados ante esos datos de pruebas usando la UI que ofrece MLflow, figura 6.7.

5. **Notificación por correo electrónico** (EmailOperator). Esta tarea notifica al correo electrónico configurado en el fichero *params.yaml* la correcta ejecución del DAG.

### 6.2.1 Consideraciones de implementación

Esta subsección tiene por objetivo justificar algunas de las decisiones tomadas en la implementación anteriormente detallada. Debido a las similitudes con el caso anterior, la elección de seleccionar una ejecución periódica del DAG de Airflow diaria responde a la motivación ya detallada en la subsección 5.2.1.



The screenshot shows the MLflow Model Registry interface. At the top, there's a navigation bar with 'mlflow', 'Experiments', and 'Models' tabs. Below this, the 'Registered Models' section is visible. It includes a 'Create Model' button, a search bar with the placeholder 'Search by model name', and buttons for 'Search', 'Filter', and 'Clear'. The main content is a table with the following columns: 'Name', 'Latest Version', 'Staging' (with a dropdown arrow), 'Production' (with a dropdown arrow), 'Last Modified', and 'Tags'. The table lists three models: 'RFClass-cwurData', 'RFClass-shanghaiData', and 'RFClass-timesData'. Each model has 'Version 1' as its latest version, is in a 'Staging' state, and has a 'Last Modified' timestamp from 2021-12-26. The 'Tags' column shows a hyphen '-' for all models. At the bottom right, there's a pagination control showing '< Page 1 >' and '10 / page'.

Name	Latest Version	Staging	Production	Last Modified	Tags
RFClass-cwurData	Version 1	-	Version 1	2021-12-26 10:36:12	-
RFClass-shanghaiData	Version 1	-	Version 1	2021-12-26 10:36:15	-
RFClass-timesData	Version 1	-	Version 1	2021-12-26 10:36:14	-

Figura 6.5: Captura de pantalla del *model registry* ofrecido por MLflow. Se detallan todos los modelos almacenados, en este caso tres, su versión, estado y última modificación, entre otros

### 1. Variables necesarias para la ejecución del experimento almacenadas en un fichero *YAML*.

Las variables necesarias para la ejecución de este caso han sido almacenadas en un fichero *YAML* en la carpeta del sistema *airflow*. En este fichero, llamado *params.yaml*, se indica el ID del DAG, el nombre de cada parámetro y su valor asociado.

### 2. Comprobación sobre la ubicación del fichero *params.yaml*. Tal y como se ha indicado en la sección 4.2.1 antes de la ejecución del experimento es necesario comprobar una serie de requisitos previos. En este caso la comprobación sobre si el fichero *params.yaml* se encuentra en la ruta correcta se realiza antes de la definición del DAG y de manera externa a la tarea *check\_prerequisites*. Esta decisión logra que el DAG ni sea ejecutado si esta condición no se cumple.

### 3. Elección de un almacén de artefactos local. A diferencia del caso anterior, todas las opciones que MLflow permite configurar de almacenamiento remoto son de pago. Por esa razón se ha elegido implementar una carpeta local llamada *artifacts*. El proceso de almacenamiento se representa en la figura 6.1.

### 4. Despliegue de modelos. De manera similar al punto anterior, la mayoría de opciones que MLflow proporciona para el despliegue de modelos son para ser llevadas a cabo en plataformas de pago, por lo que estas han sido analizadas en un ámbito local.

Cabe destacar que, en este caso, no es necesario llevar a cabo el *pipeline* de procesamiento de datos en producción, sobre los que queremos realizar predicciones (figura 3.2). Esto es porque ya han sido procesados previamente.

### 5. Elección del *backend store* como base de datos. Tal y como se ha indicado en la sección anterior, se ha seleccionado como *backend store* una base de datos en lugar de una ruta a una carpeta local. Esta decisión ha sido tomada porque MLflow no permite utilizar ni visualizar el *model registry* si no se utiliza una base de datos local.

mlflow Experiments Models

Version 1 ▾

Registered Models > RFClass-cwurData > Version 1

Registered At: 2021-12-26 10:36:12 Creator: Stage: Production ▾

Last Modified: 2021-12-26 10:36:12 Source Run: Run 7518819b751541afa07dbb22acc4a5c0

▾ Description [Edit](#)

RF classifier on cwurData. In tags tab metrics and parameters used to create this model are specified

▸ Tags

▾ Schema

Name	Type
Inputs (6)	
publications	long
citations	long

Figura 6.6: Captura de pantalla de la información almacenada en el *model registry* sobre el modelo generado en el experimento *cwurData*. Se detalla información sobre su fecha de creación, modificación, *run* que lo originó, *tags* y esquema registrado sobre su datos, entre otros.

6. **Implementación del *pipeline* usando una ‘caché’.** MLflow no comprueba si los pasos ejecutados en el *pipeline*, representado en el punto de entrada “main” del fichero MLproject, ya han sido ejecutados con anterioridad. Por esa razón, se ha implementado una comprobación sobre si la etapa a ejecutar ya ha sido realizada con anterioridad para los mismos parámetros, código y con un estado exitoso. Con ello se logra un objetivo doble: no repetir etapas que ya han sido ejecutadas y comparar de manera equitativa, en el próximo capítulo los *pipelines* ofrecidos por DVC y los proyectos de MLflow.

Gracias a la implementación de esta caché ha sido posible incluir la descarga de los datos en el *pipeline*, pues vía código se puede indicar que se ejecute siempre esta etapa.

7. **Uso de la versión 2.1.1 de Airflow y versiones superiores a la 1.0 de MLflow.** Se han seleccionado estas versiones de ambas herramientas porque otras versiones presentaban problemas de compatibilidad entre ellas.

Por otro lado, Airflow en su última versión, 2.2.3, permite en las tareas del tipo `BashOperator` definir el directorio en el cual ejecutar el comando. Esta funcionalidad es útil para no cambiar repetidamente mediante comandos la ruta de ejecución de estos. Sin embargo, esta versión no ha sido implementada porque presentaba conflictos con la versión usada en la creación del DAG.

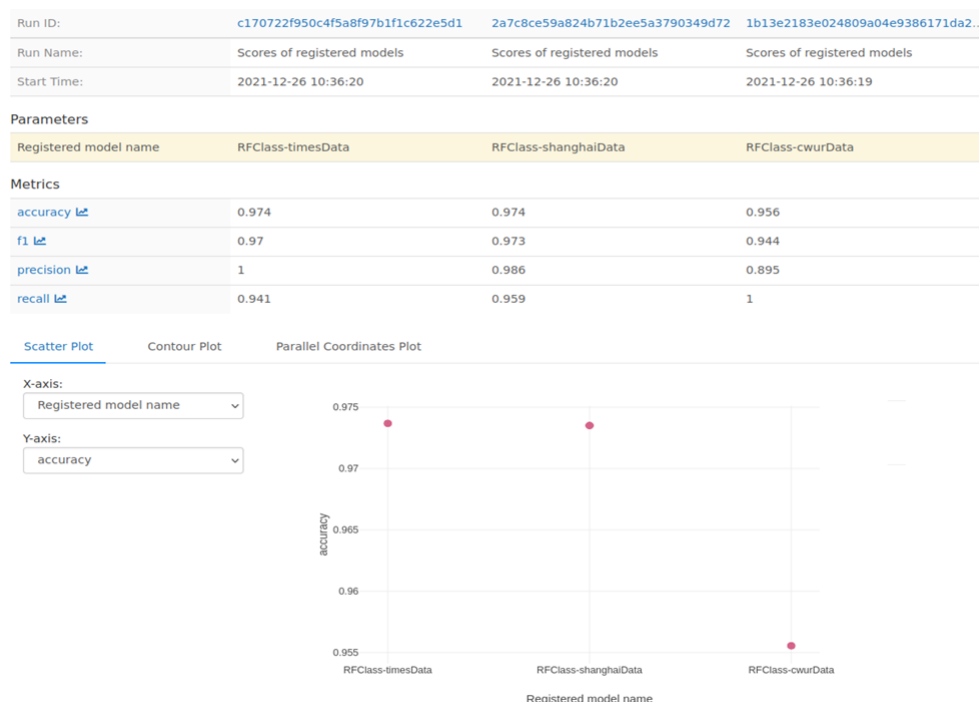


Figura 6.7: Captura de pantalla de la comparativa entre los modelos almacenados en el *model registry*. Se detalla el *run* que ha generado cada modelo, fecha de creación y métricas obtenidas, entre otros.

## 6.3 Resultados

MLflow es una herramienta sencilla de usar y permite resolver algunas dificultades como crear experimentos reproducibles en otros entornos y empaquetar el modelo.

Por otro lado, Airflow de nuevo resulta muy cómoda gracias a la división en tareas que se realiza y a proveer una interfaz gráfica muy potente.

1. **Facilidad de uso.** MLflow es una herramienta que provee una documentación en línea bastante extensa con algunos tutoriales generales y ejemplos en cada sección. Además, posee una comunidad que proporciona más ejemplos y tutoriales sobre aspectos concretos de MLflow, por lo que resolver problemas asociados a ella son sencillos de solventar. Sin embargo, debido a la extensión de la documentación proporcionada, esta se encuentra, en algunos casos, desordenada. Por ejemplo, al buscar cómo desplegar modelos MLflow no existe una única sección con las distintas opciones, sino que la información se encuentra duplicada o faltante en distintas secciones de su documentación.

A pesar de ello, MLflow es una herramienta sencilla de utilizar que no ocasiona grandes dificultades a la hora de hacer uso de su funcionalidad. En ese sentido, el proporcionar una interfaz gráfica facilita su uso.

Por otro lado, Airflow permite hacer uso de todo el potencial de MLflow, por lo que no ha sido necesario modificar ninguna tarea en ese sentido. MLflow permite configurar vía API

Python el token y el usuario para acceder a una plataforma remota como Azure, lo que evita introducir contraseñas vía comandos como sucedía en el caso anterior.

2. **Escalabilidad.** A diferencia del caso anterior en el que las variables eran definidas en Airflow, en este experimento se han detallado en el fichero `params.yaml`. Gracias a este planeamiento es posible escalar el número de DAGs y variables asociadas a ellos consiguiendo un objetivo doble: acceder fácilmente al valor de estas variables y modificar todas las variables asociadas a cada DAG en un único fichero. De nuevo, recordar que esta es una característica de decisión de diseño con la intención de ilustrar las capacidades de la herramienta, en ambos experimentos se podría haber utilizado esta solución, ya que es más óptima en términos de escalabilidad.

Además, en el fichero `MLproject` se han declarado al comienzo del mismo los valores por defecto de todas las variables usadas en dicho fichero. Con ello se consigue que, si se desea modificar o consultar alguno de estos valores, tan solo es necesario acceder a este archivo.

3. **Requisitos software.** Tanto MLflow como Airflow pueden ser instaladas por medio del comando `Pip` o configurando una dependencia de Anaconda. Gracias a ello es posible hacer una instalación sencilla de ambas herramientas.

Por otro lado, la compatibilidad entre MLflow y Airflow ha sido complicada de llevar a cabo. Los problemas ocasionados se debían a la presencia de conflictos en las versiones de sus dependencias internas. Sin embargo, es posible la integración seleccionando cuidadosamente las versiones de ambas herramientas.

4. **Capacidades.** MLflow es una herramienta que se encuentra en todas las fases del despliegue de un modelo, desde el *lab project* hasta el *production project*, mediante una interfaz gráfica muy sencilla de utilizar. En la tabla 6.1 se muestra la evaluación de las funcionalidades definidas en la sección 4.3 que esta herramienta provee.

Cabe destacar la buena gestión que MLflow realiza de las dependencias software de sus experimentos, lo que permite hacerlos reproducibles en otros entornos.

Por otro lado, MLflow no proporciona una forma de crear *pipelines*. Lo que origina que tampoco proporcione una caché que compruebe las dependencias entre etapas o si alguna tarea de ese *pipeline* ya ha sido ejecutada. Por ello, ambos han tenido que realizarse de forma manual.

Por tanto, de este experimento se concluye:

- Fácil manejo de MLflow gracias a la API Python que provee, aunque la búsqueda de determinados conceptos en la documentación puede ser confusa.
- MLflow soluciona grandes problemas como la necesidad de hacer reproducibles los experimentos en otros entornos, gracias al módulo MLflow Projects; realizar un seguimiento de los modelos y sus versiones, mediante el módulo MLflow Model Registry y empaquetar el



Función	Evaluación para MLflow	Puntuación
Función 1	Permite compartir experimentos o <i>runs</i> y los ficheros, modelos o parámetros asociados usando el almacén de artefactos y <i>backend</i> , pero no permite compartir <i>runs</i> concretas	3
Función 2	MLflow instala automáticamente las dependencias indicadas en un fichero <i>YAML</i>	4
Función 3	No comprueba si alguna etapa del experimento ya ha sido ejecutada. Sin embargo, se puede implementar manualmente consultando vía API las ejecuciones anteriores	2
Función 4	Todos los experimentos se ejecutan en el área de trabajo	0
Función 5	Proporciona una UI muy intuitiva	4
Función 6	Permite visualizar gráficas a través de su interfaz gráfica	4
Función 7	Permite obtener todas las <i>runs</i> ejecutadas juntos a los parámetros y archivos asociados	4
Función 8	Sí despliega modelos de diversas formas y en varias plataformas	4
Función 9	Sí ofrece <i>model registry</i>	4
Función 10	No ofrece la posibilidad de crear un <i>pipeline</i> , es necesario una implementación manual para ello	2
Promedio		3.1

Tabla 6.1: Tabla comparativa de las funcionalidades que provee la herramienta MLflow. En color verde se muestra si dicha herramienta implementa la funcionalidad por defecto (puntuación 3 y 4), en color amarillo si el usuario puede implementar código para conseguir obtener esa funcionalidad (puntuación 2) y en color rojo si dicha funcionalidad no existe para esa herramienta (puntuación 0 y 1).

código del modelo con el fin de abstraerlo de la librería que lo implementa, gracias al módulo MLflow Models.

- Mayor escalabilidad de este caso gracias a la API Python que MLflow provee y debido al diseño realizado usando un fichero, `params.yaml`, que contiene todas las variables necesarias para su ejecución.
- La compatibilidad entre MLflow y Airflow es posible una vez superados sus problemas de versiones, de manera que Airflow permite hacer uso de todo el potencial de MLflow.

## Capítulo 7

### Conclusiones

En este capítulo se recogen las conclusiones obtenidas en el presente proyecto. En concreto, se hace un resumen de los principales resultados, se analiza la consecución de objetivos planteados en el la sección 1.2, se detallan las asignaturas cuyos conocimientos han ayudado en el desarrollo de este proyecto, las habilidades obtenidas y trabajos futuros que plantea la realización de este TFG.

A lo largo de este proyecto se ha presentado una propuesta para llevar a cabo el proceso totalmente automatizado en la creación y despliegue de modelos a producción. Asimismo se han realizado dos casos prácticos que pretenden ilustrar la implementación de algunas partes de dicho diagrama. Las conclusiones más destacables son:

- Inexistencia de una definición común de la función que deben desempeñar los distintos elementos de un proceso totalmente automatizado, figura 3.2. Durante la realización de este proyecto se ha descubierto que la literatura no ofrece una definición clara de la función específica que debe desempeñar cada elemento del diagrama. Por ello, este TFG pretende clarificar y definir las tareas a realizar en cada elemento.
- A pesar del diagrama planteado para llevar a cabo un proceso totalmente automatizado, figura 3.2, existen algunas brechas de funcionalidad que aún no han sido resueltas debido a la inexistencia de herramientas o técnicas que permitan abordarlos. Se trata de la toma de decisiones automática en la caída del rendimiento del modelo y la implementación del *model catalog*.
- Es posible automatizar el proceso usando la herramienta Airflow. Las etapas de creación y despliegue pueden ser llevada por MLflow, mientras que la creación del modelo y versionado de datos y modelos pueden ser realizadas por DVC, ver las tablas 7.1 y 7.2 para obtener una perspectiva resumida de los resultados obtenidos en cada caso práctico.
- Airflow permite automatizar y modularizar los flujos de trabajo de manera sencilla. Además, ofrece una interfaz gráfica que facilita la gestión y ejecución de las diversas tareas. Lo que permite localizar errores o mejorar determinadas tareas.

- DVC es una herramienta intuitiva y sencilla de utilizar, en parte gracias a su gran paralelismo con git. Permite llevar a cabo el versionado de grandes volúmenes de información como datos y modelos. Sin embargo, aún es una herramienta en pleno crecimiento por lo que su comunidad de usuarios es pequeña.

A lo largo de este proyecto se han intentado analizar unos criterios que cubran por completo la problemática que resuelve **MLOps**. Por ello, DVC al encontrarse en una fase más experimental, no cubre la totalidad de la problemática analizada y obtiene puntuaciones más bajas en algunas de las funciones analizadas.

- MLflow es una herramienta cómoda de utilizar que permite abordar algunos de los problemas más complejos que se plantean, como hacer reproducibles los experimentos y empaquetar los modelos independientemente de la librería que los ha creado. Además, permite realizar el despliegue de los modelos en plataformas en la nube y alimentar los modelos con los datos para los cuales se desean hacer predicciones de manera sencilla.
- La integración entre Airflow y DVC es posible de realizar pero la primera limita algunas funciones de la segunda.
- La integración entre Airflow y MLflow es posible y no presenta grandes dificultades, salvo que algunas versiones de las herramientas no son compatibles entre sí.
- La compatibilidad entre DVC y MLflow no es posible. Esto es debido a que se solapan en algunas de las funciones que ofrecen y la forma de llevarlas a cabo es distinta. Por tanto, la integración entre ambas es posible pero puede no merecer la pena ya que no es óptimo.

Criterio	Experimento 1	Experimento 2
1. Facilidad de uso	2 puntos Comentarios: Sencillo de usar, pero en algunas funcionalidades DVC no es compatible con Airflow	4 puntos Comentarios: Muy sencillo de usar
2. Escalabilidad	1 punto Comentarios: Poco escalable	4 puntos Comentarios: Muy escalable
3. Requisitos software	3 puntos Comentarios: Poco cómoda la descarga de DVC. No presentan conflictos las dependencias de Airflow y DVC	3 puntos Comentarios: Facilidad en la descarga de MLflow y buena compatibilidad con Airflow en determinadas versiones
4. Capacidades	1.9 puntos Comentarios DVC: -Compartir experimentos con otros científicos, de manera sencilla -Comprobación de experimentos repetidos -Aplicación de experimentos en el área de trabajo -Ejecución de experimentos fuera del área de trabajo -Visualización de gráficas, ya bien para ilustrar valores de un único experimento o para comparar dos experimentos entre sí -Permite crear <i>pipelines</i>	3.1 puntos Comentarios MLflow: -Compartir experimentos con otros científicos -Instalación automática de dependencias -Comprobación de experimentos repetidos vía software -Interfaz gráfica -Visualización de gráficas -Despliegue de modelos -Almacén de modelos -Creación de <i>pipelines</i> vía software
Puntuación final	1.98	3.53

Tabla 7.1: Tabla comparativa de las dimensiones analizadas entre los dos experimentos desarrollados que incluyen las herramientas Airflow y DVC (experimento 1) y Airflow y MLflow (experimento 2). Se refiere a los lectores interesados a la sección 4.3 para un mayor detalle de cada criterio.

Función	Puntuaciones	
	Exp. 1: DVC	Exp. 2: MLflow
1. Compartir experimentos	4	3
2. Instalación automática de dependencias	0	4
3. Comprobación experimentos repetidos	4	2
4. Aplicación experimentos en el área de trabajo	4	0
5. Interfaz gráfica	0	4
6. Visualización de gráficas	3	4
7. Búsqueda y comparación de experimentos	0	4
8. Despliegue de modelos	0	4
9. Almacén de modelos	0	4
10. Creación de <i>pipelines</i>	4	2
Puntuación final	1.9	3.1

Tabla 7.2: Tabla comparativa de las funcionalidades que proveen las herramientas DVC y MLflow. Se dirige al lector a la sección 4.3 para mayor detalle en las funciones realizadas y código de colores

## 7.1 Consecución de objetivos

Esta sección recapitula los resultados obtenidos en este proyecto, y se evalúa si cumplen los objetivos planteados en la sección 1.2.

En la sección 1.1 se realiza un análisis de las distintas etapas de las que se compone un modelo de ML (OG1) y de las tareas que se deben desarrollar en cada etapa. Asimismo, se hace hincapié en la importancia del concepto de **MLOps** para llevar a cabo el paso a producción.

En la subsección 3.1.3 se detallan los distintos elementos que integran todo el proceso de creación y despliegue de modelos a producción en el ámbito empresarial (OG2). Asimismo, se indican las funciones que debe realizar cada elemento. En la figura 3.2 se representa dicho proceso.

A continuación, en la sección 3.2 se asocian las herramientas de software libre, introducidas en el capítulo 2, con cada elemento del proceso (OG3). Indicando cómo esta herramienta implementa la funcionalidad que debe desempeñar cada elemento del proceso automatizado. En esta sección también se indican que existen dos brechas de funcionalidad que aún no han sido resueltas (OG4).

Por último, para alcanzar los objetivos específicos planteados se han implementado dos casos prácticos que usan algunas de las herramientas planteadas en el capítulo 2.

En el primer caso práctico (OE1) se ilustra cómo debe llevarse a cabo la etapa experimental del proceso. Se ha utilizado la herramienta DVC para llevar a cabo un versionado de los datos, modelos, ejecución de experimentos y crear el *pipeline* que construye y valida los modelos. Por otro lado, Airflow se ha utilizado para modularizar las tareas que lleva a cabo DVC.

En el segundo y último caso práctico (OE2) se ilustra cómo debe llevarse a cabo la etapa experimental y en especial el paso a producción del proceso totalmente automatizado. Para ello, se ha utilizado la herramienta MLflow para crear el *pipeline* que construye y valida los modelos, almacenar dichos modelos en el *model registry*, desplegar los modelos desde este almacén a producción de tres formas distintas para realizar consultas y comparar gráficamente el resultado de estos modelos desplegados. De nuevo, Airflow es utilizado para modularizar estas tareas y visualizar gráficamente su ejecución.

## 7.2 Aplicación de lo aprendido

Para llevar a cabo la realización de este proyecto ha sido necesario hacer uso de algunos conocimientos aportados durante el estudio del grado asociados a las siguientes asignaturas:

1. La asignatura de Sistemas y Aplicaciones Telemáticas (SAT). Gracias a ella he aprendido a usar el lenguaje de programación Python y a entender conceptos relacionados con aplicaciones web, como por ejemplo el significado de API REST o el uso de ficheros JSON en este ámbito.

2. La asignatura Ingeniería de Sistemas de Información (ISI). Gracias a ella he aprendido sobre manejo de git y bases de datos, lo que me ha ayudado en el manejo de DVC y uso de la base de datos de MLflow.
3. La asignatura Procesamiento Digital de la Información (PDI). Gracias a ella he aprendido sobre procesamiento de datos con el fin de aplicarlos a estimadores, lo que me ha ayudado a realizar el proceso de limpieza de los datos descargados.

### 7.3 Lecciones aprendidas

Durante el desarrollo de este proyecto he aprendido lo siguiente:

1. Proceso de creación y despliegue de modelos en el ámbito empresarial. Estas tareas distan radicalmente del enfoque que el mundo académico proporciona, y por tanto, los problemas que se deben resolver son distintos y deben ser analizados de diferente forma.
2. Dificultad de las empresas a la hora de desplegar modelos. Previamente al desarrollo de este proyecto no conocía esta problemática ni la profundidad de la misma.
3. Búsqueda de información en la documentación de las herramientas elegidas. Durante el desarrollo de este proyecto he aprendido a desenvolverme mejor en la búsqueda de información en inglés en la documentación y otros recursos, como libros o vídeos, sobre las herramientas utilizadas.
4. Redacción de una memoria técnica. Previo al desarrollo de este proyecto no conocía la forma en la que una memoria o texto científico debían ser redactados y explicados.
5. Utilización de herramientas enfocadas a la creación y despliegue de modelos en el ámbito empresarial.

### 7.4 Trabajos futuros

En este proyecto se ha propuesto una arquitectura y explicado las tareas que deben llevar a cabo cada uno de sus componentes, en el desarrollo y despliegue de modelos de ML en el ámbito empresarial. Asimismo, se ha ilustrado de manera simplificada este proceso, usando algunas herramientas de software libre. Sin embargo, existen algunas tareas que no ha sido posible abarcar en este proyecto:

- Implementación práctica de todos los elementos de ciclo totalmente automatizado, figura 3.2, usando herramientas de software libre. En este proyecto solamente se han ilustrado la implementación del elemento que despliega modelos, repositorio de código, creación de paquetes de modelos en la etapa *interface*, *model registry* y despliegue local sin incluir el *pipeline* de producción, todo ello de manera automatizada. Sin embargo, no se ha implementado el *feature store*, no se ha implementado CI, no se han analizado los *packages* que contengan *pipelines* y



modelos, no se ha implementado un *package store*, no se ha implementado el *pipeline* de procesamiento de datos de producción y, por último, no se ha monitorizado todo el proceso para analizar la bajada en el rendimiento.

- Explorar las opciones de despliegue de modelos y almacenamiento de datos en plataformas como AWS o Google Cloud.
- Creación de herramientas que implementen el *model catalog* y solventar el problema en la toma de decisiones ante la bajada de rendimiento de un modelo (por ejemplo, por un cambio en los datos o *data drift*).
- Explorar otras herramientas no analizadas en este proyecto como pueden ser OpenML o Delta Lake.
- Resolver el problema de mantener un rendimiento óptimo de modelos grandes y complejos que son desplegados en sistemas distribuidos [6, capítulo 5]. Este problema aparece cuando el usuario final carece de conexión a Internet y, por tanto, es necesario desarrollar un modelo de ML que compagine la exactitud y el tamaño.

Un ejemplo de esta problemática es una aplicación de actividad física, la cual realiza recomendaciones al usuario según su nivel. En ese caso, tendríamos excesiva latencia si un modelo que se encuentre en la nube tuviera que realizar predicciones continuamente.

Por esa razón, un modelo de ML se puede dividir en dos partes, una de ellas optimizada para ocupar el menor tamaño posible y ser utilizada por el usuario final, y la otra más compleja y precisa para ser desplegada en la nube. Esto es lo que se conoce como predicción en dos fases.



## Apéndice A

# Requisitos de ejecución de los casos prácticos

### A.1 Primer caso práctico

En esta sección se detallan los requisitos software que se deben reunir para la correcta ejecución del primer caso práctico. En [este enlace](#) se pueden encontrar estos requisitos explicados más en detalle.

- **Instalación de DVC.** Para ello se puede optar por tres opciones: descarga del fichero binario, instalación por medio de snap o clonando el repositorio de código. Se puede consultar [este enlace](#) para más información.
- **Instalación de dependencias.** El fichero `my_env.yaml` contiene la declaración de un entorno de Anaconda con todas las dependencias necesarias, incluyendo la instalación de Airflow. Para crearlo y activarlo se deben ejecutar los siguientes comandos:

```
conda env create -f my_env.yaml
conda activate exp1
```

- **Configuración de Airflow.** Si es la primera vez que se ejecuta Airflow es necesario inicializar la base de datos que utiliza y crear un usuario con permisos de administrador en dicha base de datos. Se puede consultar [este enlace](#) para más información.
- **Lanzamiento de Airflow.** Para ejecutar Airflow es necesario lanzar el servidor web y el ejecutor de tareas:

```
airflow webserver
airflow scheduler
```

- **Ejecución del DAG.**

- **Creación de una contraseña SSH.** Para más información visitar el [siguiente enlace](#).
- **Registrar la contraseña creada en GitHub.** Para más información visitar el [siguiente enlace](#).
- **Activar la contraseña SSH.** Si la contraseña no está activada se debe ejecutar el siguiente comando e introducir la frase de acceso elegida:

```
ssh-add
```

- **Ejecución de comandos.** Para este caso práctico se han de haber ejecutado anteriormente los comandos: `git init`, `dvc init` y `git remote add <nombre_remoto> <link_remoto_ssh>`. Todos ellos deben ser ejecutados en la ruta local en la que se desea ejecutar este caso práctico.
- **Mover o copiar el fichero `src/dag_exp1.py` a la carpeta del sistema `~/airflow/dags`**
- **Configuración de variables necesarias.** Por último, es necesario definir las siguientes variables de Airflow: `path_to_repo_exp1`, `name_repo_ssh` y `email_to_notify_exp1`. Esta última variable es utilizada para notificar por correo electrónico la correcta o la incorrecta ejecución del DAG y debe contener una dirección de correo electrónico válida. Para ello, es necesario [configurar un servidor SMTP](#) en el archivo `airflow.cfg`. Sin embargo, esta configuración de un servidor SMTP es opcional y no es necesaria para la correcta ejecución del DAG.

## A.2 Segundo caso práctico

En esta sección se detallan los requisitos software que se deben reunir para la correcta ejecución del segundo caso práctico. En [este enlace](#) se pueden encontrar estos requisitos explicados más en detalle.

- **Instalación de dependencias.** El fichero `airflow_env.yaml` contiene la declaración de un entorno de Anaconda con todas las dependencias necesarias incluyendo la instalación de Airflow y MLflow. Para crearlo y activarlo se deben ejecutar los siguientes comandos:

```
conda env create -f airflow_env.yaml
conda activate exp2
```

Por otro lado, el fichero `my_env.yaml` contiene aquellas dependencias que MLflow necesita para ejecutar los puntos de entrada declarados en el fichero `MLproject`.

- **Configuración de Airflow.** Si es la primera vez que se ejecuta Airflow es necesario inicializar la base de datos que utiliza y crear un usuario con permisos de administrador en dicha base de datos.
- **Lanzamiento de Airflow.** Para ejecutar Airflow es necesario lanzar el servidor web y el ejecutor de tareas:

```
airflow webserver
airflow scheduler
```

- **Ejecución del DAG.**
  - **Ejecución de comandos.** Para este caso práctico se han de haber ejecutado anteriormente los comandos: `git init` y `bash src/server_tracking_server.sh`. Todos ellos deben ser ejecutados en la ruta local en la que se desea ejecutar este caso práctico.
  - **Mover o copiar el fichero `src/dag_exp2.py` a la carpeta del sistema `~/airflow/dags` y mover o copiar el fichero `src/params.yaml` a la carpeta `~/airflow/`**
  - **Configuración de variables necesarias.** En el fichero `params.yaml` es necesario configurar las siguientes variables:
    - \* La variable `email_to_notify_exp2` debe contener una dirección de correo electrónico válida a la que notificar la correcta o incorrecta ejecución del DAG.
    - \* La variable `path_to_repo_exp2` contiene la ruta local en la cual se ejecutará el segundo caso práctico.



## Glosario de términos

**CI/CD** (Integración continua y entrega continua o *continuous integration continuous delivery/deployment*) Consiste en la automatización en el desarrollo, construcción y despliegue del modelo (CD) así como el *testing* del *pipeline* (CI). En el caso de CD se utilizan dos *pipelines*:

- El primero se encarga del procesado de los datos para, a continuación, construir, entrenar, probar y validar el modelo elegido. De manera que los desarrolladores de modelos únicamente eligen el modelo y el *pipeline* se encarga de construirlo.
- El segundo *pipeline* es aquel que despliega los modelos en la aplicación y los reentrena en caso necesario. Esta aproximación permite llevar a cabo un mantenimiento mucho más sencillo al no ser necesario desplegar un modelo cada vez que este disminuye su rendimiento.

Por otro lado, CI consiste en probar el modelo de ML y el *pipeline* con el fin de asegurar que las salidas que proporcionan son las esperadas por la aplicación. También incluye tareas de creación del *pipeline*. 12, 26, 29, 32

**feature store** Almacén central de características en el que se encuentran todos los datos a los cuales se les ha aplicado el mismo procedimiento de procesado y limpieza. De esta forma, todos los datos estarán estandarizados a una definición común. 27, 30–32, 78

**MLOps** (Operaciones de ML) Conjunto de tecnologías y prácticas utilizadas para gestionar de forma coherente y eficiente los modelos de ML a lo largo de sus tres fases de desarrollo: diseño, desarrollo del modelo y despliegue. Su objetivo final es integrar a los equipos de ciencia de datos e ingenieros de ML con los administradores del sistema o equipos operativos, con el fin de automatizar y monitorizar continuamente todo el proceso. 2–4, 7–9, 11, 12, 20, 24–27, 32, 37, 74, 77

**model catalog** Almacén central que guarda los modelos sin entrenar junto con sus hiperparámetros y las decisiones que han llevado a implementar dicho modelo. Asimismo, también es posible almacenar modelos entrenados a modo de ejemplo. De esta forma se consigue que los equipos técnicos se beneficien de los avances realizados por otros equipos de la misma compañía. . 28, 32–34, 73, 79

- model registry** Almacén central de modelos que guarda y versiona tanto los modelos o pesos como sus métricas e hiperparámetros. Ello permite mantener un lugar centralizado al que acceder a todas las versiones del modelo, permitiendo ahorrar tiempo y esfuerzo a los diferentes equipos de desarrolladores. [XII](#), [15](#), [19](#), [28](#), [31](#), [32](#), [34](#), [46](#), [60–69](#), [71](#), [77](#), [78](#)
- package store** Almacén central en el que se encuentran todos los paquetes estables listos para ser utilizados en el despliegue de los modelos en producción. Esta unidad puede verse como una aplicación de CD a los *pipelines* contenidos en los *packages*. [29](#), [30](#), [79](#)
- pipeline** Artefacto que ejecuta tareas de manera automática y secuencial. Permite simplificar grandes y complejos fragmentos de código en tareas más simples. Ello permite obtener un código reproducible y modularizado, así como mayor facilidad a la hora de entenderlo, realizar cambios o localizar errores. En la fase de desarrollo se utilizan para llevar a cabo el procesamiento de datos, mientras que en la fase de producción se utiliza para facilitar la mantenibilidad de la aplicación. [XI](#), [12–14](#), [25](#), [27–36](#), [43](#), [44](#), [46](#), [47](#), [49–56](#), [58–61](#), [63](#), [67](#), [68](#), [70](#), [71](#), [75–79](#), [85](#), [86](#)



# Bibliografía

## Libros

- [1] Sridhar Alla y Suman Kalyan Adari. *Beginning MLOps with MLFlow*. USA: Apress, 2020. ISBN: 9781484265482.
- [2] Tathagata Das Denny Lee y Vini Jaiswal. *Delta Lake: The Definitive Guide*. USA: O'Reilly Media, mayo de 2020. ISBN: 9781098104597.
- [3] Peter A. Flach. *Machine Learning. The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012. ISBN: 978-1-10-742222-3.
- [4] Bas Harenslak y Julian de Ruiter. *Data Pipelines with Apache Airflow*. USA: Manning Publications, abr. de 2021. ISBN: 9781617296901.
- [5] John D. Kelleher, Brian Mac Namee y Aoife D'arcy. *Fundamentals of Machine Learning for Predictive Analytics*. 2nd ed. USA: MIT Press, oct. de 2020. ISBN: 9780262044691.
- [6] Valliappa Lakshmanan, Sara Robinson y Michael Munn. *Machine Learning Design Patterns*. USA: O'Reilly Media, oct. de 2020. ISBN: 9781098115784.
- [7] Foster Provost y Tom Fawcett. *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking*. USA: O'Reilly Media, sep. de 2013. ISBN: 9781449361327.
- [8] Wajid Khattak Thomas Erl y Paul Buhler. *Big Data Fundamentals: Concepts, Drivers and Techniques*. USA: Prentice Hall, 2016. ISBN: 9780134291079.

## Estudios técnicos

- [9] Algorithmia, Inc. *2020 state of enterprise machine learning*. Inf. téc. Algorithmia, Inc., 2020.
- [10] Dirección General de Redes de Comunicación, Contenido y Tecnologías (Comisión Europea), Ipsos e International Centre for Innovation Technology and Education (iCite). *European enterprise survey on the use of technologies based on artificial intelligence*. Inf. téc. Comisión Europea, 2020.
- [11] Ministerio de Ciencia, Innovación y Universidades. *Estrategia española de I+D+I en inteligencia artificial*. Inf. téc. Ministerio de Ciencia, Innovación y Universidades, 2019.

- [12] Territorio Creativo. *Estudio de Transformación Digital de la empresa española*. Inf. téc. Good Rebels, 2015.

## Otras fuentes

- [13] Continuous Delivery Foundation (CDF). *MLOps Roadmap 2020*. Fecha de consulta: 26 de febrero de 2021. URL: <https://github.com/cdfoundation/sig-mlops/blob/master/roadmap/2020/MLOpsRoadmap2020.md>.
- [14] Tomas Nykodym y Clemens Mewald Aaron Davidson. *Announcing MLflow Model Serving on Databricks*. Fecha de consulta: 25 de julio de 2021. URL: <https://databricks.com/blog/2020/06/25/announcing-mlflow-model-serving-on-databricks.html>.
- [15] PUE Blog. *Transforma tu negocio con PUE y DataRobot, la plataforma líder en AI para el ámbito empresarial*. Fecha de consulta: 10 de noviembre de 2021. URL: <https://blog.pue.es/datarobot-plataforma-lider-ai-ambito-empresarial-transforma-negocio/>.
- [16] Brenner Heintz Burak Yavuz y Denny Lee. *Diving Into Delta Lake: Schema Enforcement & Evolution*. Fecha de consulta: 20 de noviembre de 2021. URL: <https://databricks.com/blog/2019/09/24/diving-into-delta-lake-schema-enforcement-evolution.html>.
- [17] José Ruiz Cristina. *Qué es DevOps (y sobre todo qué no es DevOps)*. Fecha de consulta: 9 de julio de 2021. URL: <https://www.paradigmadigital.com/techbiz/que-es-devops-y-sobre-todo-que-no-es-devops/>.
- [18] Brian Custer. *What is Delta Lake in databricks?* Fecha de consulta: 20 de noviembre de 2021. URL: <https://3cloudsolutions.com/post/what-is-delta-lake-in-databricks/>.
- [19] EthicalML. *Awesome production machine learning*. Fecha de consulta: 26 de febrero de 2021. URL: <https://github.com/EthicalML/awesome-production-machine-learning#model-serving-and-monitoring>.
- [20] Julien Kervizic. *10 Benefits to using Airflow*. Fecha de consulta: 26 de febrero de 2021. URL: <https://medium.com/analytics-and-data/10-benefits-to-using-airflow-33d312537bae>.
- [21] Mark P.J. van der Loo. *The Data Validation Cookbook*. Fecha de consulta: 23 de diciembre de 2021. URL: <https://cran.r-project.org/web/packages/validate/vignettes/cookbook.html#Preface>.
- [22] Fernando López. *Track Your ML models as a Pro, Track them with MLflow*. Fecha de consulta: 26 de junio de 2021. URL: <https://towardsdatascience.com/track-your-ml-models-as-pro-track-them-with-mlflow-11fc5486e389>.

- [23] Jules Damji y Clemens Mewald Mani Parkhe Sue Ann Hong. *Databricks Extends MLflow Model Registry with Enterprise Features*. Fecha de consulta: 25 de julio de 2021. URL: <https://databricks.com/blog/2020/04/15/databricks-extends-mlflow-model-registry-with-enterprise-features.html>.
- [24] Dan Merron. *Deployment Pipelines (CI/CD) in Software Engineering*. Fecha de consulta: 9 de agosto de 2021. URL: <https://www.bmc.com/blogs/deployment-pipeline>.
- [25] Chris Moradi. *Maintainable ETLs: Tips for Making Your Pipelines Easier to Support and Extend*. Fecha de consulta: 30 de agosto de 2021. URL: <https://multithreaded.stitchfix.com/blog/2019/05/21/maintainable-etls/>.
- [26] Universidade de Oporto. Reitoria. Serviço de Melhoria Contínua. *A Universidade do Porto no CWUR World University Rankings 2015*. Fecha de consulta: 21 de diciembre de 2021. Jul. de 2015. URL: <https://repositorio-aberto.up.pt/bitstream/10216/84158/2/136702.pdf>.
- [27] Écrit par. *Apache Airflow: What is it and why you should start using it*. Fecha de consulta: 26 de febrero de 2021. URL: <https://www.invivoo.com/apache-airflow-what-it-is/>.
- [28] Apache Airflow Development Team. *Apache Airflow*. Fecha de consulta: 26 de febrero de 2021. URL: <https://airflow.apache.org/docs/apache-airflow/stable/concepts.html>.
- [29] AWS team. *What is a data lake?* Fecha de consulta 20 de noviembre de 2021. URL: <https://aws.amazon.com/es/big-data/datalakes-and-analytics/what-is-a-data-lake/>.
- [30] Databricks Team. *Delta Lake*. Fecha de consulta: 20 de noviembre de 2021. URL: <https://databricks.com/product/delta-lake-on-databricks>.
- [31] DataRobot team. *DataRobot tours*. Fecha de consulta: 10 de noviembre de 2021. URL: <https://www.datarobot.com/tours/>.
- [32] Delta Lake Team. *Build Lakehouses with Delta Lake*. Fecha de consulta: 20 de noviembre de 2021. URL: <https://delta.io/>.
- [33] DVC Development Team. *Data Version Control*. Fecha de consulta: 26 de febrero de 2021. URL: <https://dvc.org/doc/use-cases/versioning-data-and-model-files>.
- [34] Git Team. *10.3 Git Internals - Git References*. Fecha de consulta 4 de enero de 2022. URL: <https://git-scm.com/book/en/v2/Git-Internals-Git-References>.
- [35] MLFlow Development Team. *MLFlow*. Fecha de consulta: 26 de febrero de 2021. URL: <https://www.mlflow.org/docs/latest/concepts.html>.
- [36] MLFlow Development Team. *MLflow Tracking*. Fecha de consulta 29 de diciembre de 2021. URL: <https://www.mlflow.org/docs/latest/tracking.html>.
- [37] MLOps Team. *MLOps Infrastructure Stack*. Fecha de consulta: 26 de febrero de 2021. URL: <https://ml-ops.org/content/state-of-mlops>.

- [38] MLOps Team. *MLOps Principles*. Fecha de consulta: 9 de agosto de 2021. URL: <https://mlops.org/content/mlops-principles>.
- [39] Neptune Team. *Metadata store for MLOps, built for research and production teams that run a lot of experiments*. Fecha de consulta 29 de diciembre de 2021. URL: <https://neptune.ai/product#how-it-works>.
- [40] OpenML team. *OpenML doc*. Fecha de consulta: 11 de noviembre de 2021. URL: <https://docs.openml.org/>.
- [41] PowerData team. *Data Warehouse: todo lo que necesitas saber sobre almacenamiento de datos*. Fecha de consulta: 20 de noviembre de 2021. URL: <https://www.powerdata.es/data-warehouse>.
- [42] Shanghai Ranking Team. *ShanghaiRanking's Global Ranking of Academic Subjects Methodology 2021*. Fecha de consulta: 21 de diciembre de 2021. URL: <https://www.shanghairanking.com/methodology/gras/2021>.
- [43] Sklearn Team. *Sklearn RandomForestClassifier*. Fecha de consulta 4 de enero de 2022. URL: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html>.
- [44] THE Team. *THE World University Rankings 2021: methodology*. Fecha de consulta: 21 de diciembre de 2021. URL: <https://www.timeshighereducation.com/world-university-rankings/world-university-rankings-2021-methodology>.
- [45] Universidad de Valladolid. *CWUR*. Fecha de consulta: 21 de diciembre de 2021. URL: <https://rank.uva.es/ranking/cwur/>.
- [46] Universidad de Valladolid. *THE*. Fecha de consulta: 21 de diciembre de 2021. URL: <https://rank.uva.es/ranking/the/>.
- [47] Wikipedia. *Escala Likert*. Fecha de consulta 15 de enero de 2022. URL: [https://es.wikipedia.org/wiki/Escala\\_Likert](https://es.wikipedia.org/wiki/Escala_Likert).
- [48] Matei Zaharia. *Introducing MLflow: an Open Source Machine Learning Platform*. Fecha de consulta: 20 de julio de 2021. URL: <https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html>.