

TRABAJO FIN DE MÁSTER

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA.

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL



E.T.S. INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

PREDICCIÓN AUTOMÁTICA DEL RESULTADO DE INTEGRACIÓN CONTINUA EN EL DESARROLLO DE
SOFTWARE MODERNO

AUTOMATIC PREDICTION OF CONTINUOUS INTEGRATION OUTCOME IN MODERN SOFTWARE
DEVELOPMENT

Realizado por

Joaquín Alejandro España Sánchez

Tutorizado por

Gabriel Jesús Luque Polo

Francisco Javier Servant Cortés



UNIVERSIDAD
DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

—
Universidad de Málaga,
Málaga, Septiembre de 2024

Índice general

1. Introducción	4
2. Antecedentes y trabajos relacionados	5
2.1. Antecedentes	5
2.1.1. El ciclo de vida de la Integración Continua.	5
2.1.2. Características de las <i>builds</i>	6
2.1.3. El costo de la Integración Continua.	8
2.2. Trabajos relacionados	9
3. Objetivos y preguntas de investigación	11
4. Descripción del problema	13
5. Detalles de la propuesta	14
5.1. Construcción del modelo	14
5.1.1. Hiperparámetros.	16
5.2. Evaluación del modelo	16
5.2.1. Validación Cruzada.	17
5.2.2. Umbral de decisión.	18
5.3. <i>Features</i> empleadas	18
5.3.1. Cálculo de <i>features</i> durante la predicción.	20
5.4. Interfaz gráfica	22
5.5. Detalles técnicos de la implementación	24
5.5.1. Tecnologías empleadas.	24
5.5.2. <i>GitHub</i> REST API.	24
5.5.3. Procesamiento de <i>builds</i>	25
6. Experimentación	26
6.1. Diseño experimental	27
6.1.1. Técnicas a comparar.	27
6.1.2. Descripción del <i>dataset</i>	27
6.1.3. Procedimiento para el entrenamiento, prueba y validación cruzada	29
6.1.4. Métricas de evaluación.	30
6.2. Resultados	31
7. Amenazas a la validez	31
7.1. Validez de Constructo.	31
7.2. Validez Interna.	31
7.3. Validez Externa	32
8. Conclusiones y trabajos futuros	32

Resumen – En el contexto del desarrollo de *software* moderno, la Integración Continua (CI) es una práctica ampliamente adoptada que busca automatizar el proceso de integración de cambios de código en un proyecto. A pesar de ofrecer numerosas ventajas, implementarla conlleva una serie de costos significativos que deben ser abordados para garantizar la eficiencia a largo plazo. La fase de Integración Continua puede resultar costosa tanto en términos de recursos computacionales como económicos, llevando a grandes empresas como Google y Mozilla a invertir millones de dólares en sus sistemas de CI [12]. Han surgido numerosos enfoques para reducir el costo asociado a la carga computacional evitando ejecutar construcciones que se espera que sean exitosas [9]. Sin embargo, estos enfoques no son precisos, llegando a hacer predicciones erróneas que omiten ejecutar construcciones que realmente fallan. Además de los costos asociados con la carga computacional y económica de la CI, otro problema al que se enfrentan los equipos de desarrollo de *software* es el tiempo que deben esperar para obtener *feedback* del resultado del proceso de CI [10]. Este tiempo de espera en ocasiones puede ser significativo y puede afectar negativamente a la productividad y eficiencia del equipo, así como a la capacidad de respuesta ante problemas y ajustes rápidos en el desarrollo. Así, en este trabajo nuestro objetivo es reducir el costo computacional en CI, al mismo tiempo que maximizamos la observación de construcciones fallidas. Para ello, se ha realizado un estudio sobre las técnicas existentes [9,11,17,2,5,3], y se ha propuesto una implementación, *JAES24*, que busca contribuir a las mismas. Este nuevo enfoque amplía el estado del arte de técnicas existentes que hacen uso de *Machine Learning* para la predicción de construcciones fallidas, mejorando sus resultados y ofreciendo un punto diferenciador, una interfaz gráfica. Dicha interfaz permite interactuar de forma sencilla con el sistema, abstrayendo la complejidad de los algoritmos de predicción y ofreciendo una forma intuitiva y sencilla de realizar predicciones basadas en un repositorio concreto. Posteriormente, se han realizado una serie de experimentos para verificar y validar la efectividad de *JAES24* en comparación con otras técnicas existentes. Finalmente, se desarrollan unas conclusiones sobre los resultados obtenidos y se proponen posibles líneas de trabajo futuro.

Palabras clave: Integración Continua, Predicción de Builds, Aprendizaje Automático, Ahorro de Costos, Características de Builds

Abstract – In the context of modern software development, Continuous Integration (CI) is a widely adopted practice that aims to automate the process of integrating code changes in a project. Despite offering numerous advantages, implementing CI involves significant costs that need to be addressed to ensure long-term efficiency. The Continuous Integration phase can be costly in terms of computational and economic resources, leading large companies like Google and Mozilla to invest millions of dollars in their CI systems [12]. Several approaches have emerged to reduce the cost associated with computational load by avoiding running builds that are expected to be successful [9]. However, these approaches are not accurate, often making erroneous predictions that skip running builds that actually fail. In addition to the costs associated with computational and economic load of CI, another problem faced by software development teams is the time they have to wait to get feedback on the CI process outcome [10]. This waiting time can sometimes be significant and can negatively impact team productivity and efficiency, as well as the ability to respond to issues and make quick adjustments in development. Therefore, the objective of this work is to reduce the computational cost in CI while maximizing the observation of failed builds. To achieve this, a study has been conducted on existing techniques [9,11,17,2,5,3], and an implementation, *JAES24*, has been proposed to contribute to them. This new approach extends the state of the art of existing techniques that use *Machine Learning* for predicting build failures, improving their results and offering a distinguishing feature, a graphical interface. This interface allows for easy interaction with the system, abstracting the complexity of the prediction algorithms and providing an intuitive and simple way to make predictions based on a specific repository. Subsequently, a series of experiments have been conducted to verify and validate the effectiveness of *JAES24* in comparison with other existing techniques. Finally, conclusions are drawn from the results obtained, and possible future research directions are proposed.

Keywords: Continuous Integration, Build Prediction, Machine Learning, Cost Saving, Build Features

1. Introducción

La Integración Continua (*Continuous Integration*, CI) es una práctica de desarrollo de *software* que busca automatizar el proceso de fusión de cambios de código en un proyecto, donde cada integración es verificada mediante la ejecución automática de pruebas. Este proceso busca la detección temprana de errores y mejorar la calidad del *software*, permitiendo una integración más frecuente y rápida del trabajo de todos los desarrolladores. Las buenas prácticas de CI [3] permiten una rápida detección de errores y su resolución, un *feedback* rápido, la reducción de errores que provienen de tareas manuales, unas tasas de *commits* y *pull requests* más altas, una calidad del *software* mayor, reconocer errores en producción temprano antes del despliegue, etc. Numerosos son sus ámbitos de aplicación: *software* empresarial, desarrollo de aplicaciones web, proyectos de código abierto, aplicaciones móviles, etc. Todo ello, haciendo uso de las distintas herramientas que existen en el mercado [16], como *GitHub Actions*, *Jenkins*, *Travis CI*, *CircleCI*, *Azure DevOps*, entre otras.

Para contextualizar el problema que nos ocupa, vamos a describir algunos términos relevantes para el entendimiento del mismo. A lo largo del trabajo, nos referiremos como *build* al proceso automático mediante el cual el código fuente se compila, se ejecutan las pruebas, y se genera un artefacto *software*, ya sea un ejecutable, un contenedor, un paquete, etc., que está listo para ser desplegado o usado en producción. Cada *build* es lanzada por lo que se denomina comúnmente *trigger*, que puede ser un:

- **Commit:** representa una “instantánea” del estado del proyecto en un momento específico, guardando las modificaciones que se han hecho a los archivos desde el último *commit*. Cada vez que el desarrollador realiza un *commit*, se dispara una nueva *build*.
- **Pull request:** un *pull request* o solicitud de incorporación de cambios es una solicitud formal para fusionar cambios propuestos en una rama de desarrollo a otra rama, que generalmente es la rama principal. Este tipo de solicitud permite la revisión de los cambios realizados, su discusión, y aprobación del código por parte de otros desarrolladores antes de integrarlo con la rama principal. En este caso, al crear o actualizar un *pull request*, se lanza una *build* para verificar que el código cumple con los estándares de calidad.
- **Schedule:** se pueden programar *builds* para que se ejecuten en un intervalo de tiempo regular, independientemente de si hubo o no cambios en el código.

Existen numerosos sistemas de CI en la actualidad, *GitHub Actions*, *Jenkins*, *Travis CI*, *CircleCI*, *Azure DevOps*, entre otros, sin embargo, en este trabajo nos centraremos en *GitHub Actions*. *GitHub Actions* es el sistema de CI más utilizado en la actualidad, y al cual muchos otros sistemas migraron debido a sus características, especialmente *Travis CI*. En 2020, *Travis CI* decidió imponer numerosas restricciones a su plan gratuito para proyectos *software* de código abierto [16], siendo este uno de los principales motivos para su migración hacia *GitHub Actions*. Además, existen otras razones para esta migración, como puede ser utilizar una herramienta de CI más confiable, mejor integración con soluciones *self-hosted*, mejor soporte para múltiples plataformas, la reducción de la cantidad de uso compartido de la herramienta, tener más funcionalidades, etc.

El ciclo de vida de la Integración Continua, a pesar de ofrecer numerosas ventajas, conlleva grandes costos asociados debido a los recursos computacionales [6] necesarios para ejecutar las construcciones, comúnmente denominadas *builds*. A lo largo de este trabajo, nos referiremos como costo computacional al hecho de ejecutar una *build*, es decir, el proceso de construir el *software* y ejecutar todas las pruebas cuando la CI es lanzada. Este costo asociado se acentúa en empresas de gran tamaño, donde el número de *builds* que se ejecutan diariamente es muy elevado [4,15]. Ahorrar en dicho costo computacional se convierte por tanto en un objetivo clave para las mismas. Optimizando la cantidad de *builds* que se ejecutan, podemos lograr una reducción significativa de este costo, ya que se habrán consumido menor cantidad de recursos. Además, hay que sumarle el tiempo de espera que los desarrolladores deben soportar cuando el tiempo de ejecución de la *build* es elevado, pudiendo ralentizar el tiempo de respuesta ante problemas y ajustes rápidos en el desarrollo.

En los últimos años, han surgido numerosos enfoques centrados en reducir el costo computacional asociado a la ejecución de CI [12,9,11,17,2,5]. La idea principal de estos enfoques es reducir el número de *builds* que se

ejecutan, prediciendo el resultado antes de su ejecución y, por lo tanto, ahorrándose ese costo computacional. Las *builds* predichas como construcciones exitosas (*build pass*) no se ejecutan, mientras que las predichas como construcciones fallidas (*build failure*) sí se ejecutan. De esta forma, se mantiene el valor conceptual de la CI, que es la detección temprana de errores, pero reduciendo el costo computacional asociado en el proceso. Este estudio toma como punto de partida el algoritmo de *Machine Learning SmartBuildSkip* [9]. La idea principal es realizar una contribución a este algoritmo, realizando un estudio de las *features* que se usan para la predicción, y añadiendo nuevas *features* más significativas que puedan mejorar estudios existentes. Además, se ha creado una aplicación web sencilla con la que el usuario puede interactuar de forma directa a través de una interfaz gráfica, abstrayendo la complejidad de los algoritmos de predicción y ofreciendo una forma intuitiva y sencilla de realizar predicciones basadas en un repositorio concreto. Por lo tanto, este estudio se enmarca en el desarrollo de *software* moderno, específicamente en el ámbito de la Integración Continua y la predicción automática del resultado de dicha integración.

La memoria queda organizada de la siguiente forma: en primer lugar, se realiza un estudio del estado del arte que sitúa los antecedentes previos a la Integración Continua y la predicción automática de resultados de *builds*. Posteriormente, se establecen los objetivos y preguntas de investigación que pretende este estudio responder. A continuación, se describe en detalle el problema a resolver, los principales obstáculos que se plantean y sus posibles soluciones. Acto seguido, se desarrolla con detalle nuestro enfoque al problema, describiendo las tecnologías usadas y el desarrollo de la solución. Después se presentan las pruebas y resultados obtenidos, comparando la solución con otras existentes, a modo de validar y verificar la aportación de nuestra solución. Seguidamente, se comentan las amenazas a la validez, una parte esencial en cualquier trabajo de investigación. Este apartado nos permite identificar y discutir posibles limitaciones que podrían afectar a la validez de los resultados y a las conclusiones. Por último, se dan unas conclusiones sobre los resultados obtenidos y se proponen posibles líneas de trabajo futuro.

2. Antecedentes y trabajos relacionados

En esta sección se comentan los principales conceptos necesarios para entender el resto del documento, así mismo como un repaso a las técnicas que existen en la literatura para abordar el problema tratado en este trabajo.

2.1. Antecedentes

Este trabajo se centra en la implementación, evaluación y mejora del algoritmo de predicción de CI propuesto en [9]. Para comprender mejor el contexto en el que se desarrolla, primero vamos a presentar algunos de los conceptos básicos de CI y del problema que nos ocupa. En primer lugar, se describirá el ciclo de vida de CI, junto a las dos casuísticas que pueden darse en el proceso de integración. En segundo lugar, hablaremos sobre la extracción de características, un aspecto fundamental para algoritmos de predicción. Por último, se hablará del consumo de recursos computacionales que supone la implementación de CI, de ahí la principal motivación de este trabajo, la reducción de dichos costes.

2.1.1. El ciclo de vida de la Integración Continua. La Integración Continua es un proceso iterativo en el cual varios contribuidores hacen cambios sobre un mismo código base añadiendo nuevas funcionalidades, para luego integrarlas a la misma línea temporal de desarrollo, de forma controlada y automatizada. Cada integración se realiza a través de la compilación, construcción y ejecución de pruebas automatizadas sobre el código fuente [6]. Aunque pueda parecerlo, la Integración Continua no es un proceso trivial, en [4] se describen las buenas prácticas de CI que deben seguirse para garantizar la calidad del *software*, algunas de las cuales han sido fuertemente adoptadas en el sector, como por ejemplo:

1. **Punto de código fuente único:** para facilitar la integración de cualquier desarrollador a un proyecto, es fundamental que este pueda obtener el código fuente actualizado del proyecto. La mejor práctica es utilizar un sistema de control de versiones como fuente única del código. Todos los archivos necesarios para la construcción del sistema, incluidos *scripts* de instalación, archivos de configuración, etc., deben estar en el repositorio.
2. **Automatización de *builds*:** para proyectos pequeños, construir la aplicación puede ser tan sencillo como ejecutar un único comando. Sin embargo, para proyectos más complejos o con dependencias externas, la construcción puede ser un proceso complicado. El uso de *scripts* de construcción automatizados es esencial para manejar estos procesos, llegando a analizar qué partes del código necesitan ser recompiladas, y gestionando dependencias para evitar recompilar innecesariamente.
3. **Desarrollo de pruebas unitarias o de validación interna:** compilar el código no es suficiente para asegurar que funciona correctamente, por lo que se implementan pruebas automatizadas. Normalmente, estas se dividen en pruebas unitarias, que prueban partes específicas del código; y pruebas de aceptación, que prueban el sistema completo. Aunque este proceso no puede garantizar la ausencia total de errores, ofrece un mecanismo efectivo de mejorar la calidad del *software* mediante la detección y corrección continua de fallos.

Sin embargo, en el mundo real, la forma de aplicar cada una de estas técnicas y la prioridad con la que se aplica puede estar fuertemente influenciada por la cultura empresarial donde se desarrolle. En [3], se realiza un caso de estudio con tres empresas donde se percibe que la adopción de las prácticas de CI no es homogénea. Por ejemplo, con respecto a tener un único punto de código fuente, algunas prefieren minimizar los conflictos de fusión (*merge conflicts*) que el beneficio poco claro de usar un único repositorio. Por otro lado, en cuanto a las pruebas unitarias, existen diferencias debido a limitaciones en las herramientas (poco factibles para realizar pruebas de interfaz de usuario), a las percepciones prácticas (el trabajo necesario para las pruebas de integración supera los beneficios percibidos) y al contexto del proyecto (pruebas centradas en datos requieren comunicación con servicios externos).

Ejemplo práctico: Un desarrollador hace un *commit* (una instantánea de los cambios realizados) y mediante una acción de *push*, lo envía al repositorio central. El servidor de CI [16] (Jenkins, Travis CI, GitHub Actions, etc.) detecta automáticamente este nuevo *commit* y desencadena el *pipeline* de CI. El servidor extrae el nuevo código del repositorio y comienza a construir la aplicación, lo que denominamos la fase de construcción o *build*. Esta parte puede incluir la compilación del código fuente, la instalación de dependencias, etc. Una vez que la aplicación está construida, se ejecutan una serie de pruebas automatizadas (*Self-Testing code*) [6]. Dichas pruebas pueden ser pruebas unitarias, pruebas de integración, pruebas funcionales o pruebas de interfaz de usuario. Dependiendo del resultado de las fases anteriores, podemos encontrarnos dos casos:

- **La *build* ha sido exitosa:** todas las pruebas han pasado con éxito. En este caso, el servidor de CI puede desplegar la aplicación en un entorno de pruebas o producción.
- **La *build* ha fallado:** alguna de las pruebas ha fallado. En este caso, el servidor de CI suele notificar a los desarrolladores y detiene el despliegue de la aplicación.

2.1.2. Características de las *builds*. Al ejecutarse una *build*, se pueden extraer de ella una serie de características con las que algoritmos de predicción pueden predecir el resultado de la integración. Tener un conjunto de *features* bien seleccionadas y significativas mejorará la precisión de los modelos. La mayoría de estudios utilizan características extraídas directamente de la base de datos de TravisTorrent [2], sin embargo, estas características no son las mejores para predecir *builds* que fallan, es decir, *builds failures*. Algunos enfoques [2,5] hacen uso de características basadas en la *build* actual, la *build* anterior y el histórico ligado a todas las ejecuciones de *builds* anteriores. El primer estudio en utilizar técnicas de *Machine Learning* para predecir el resultado de CI fue realizado por Hassan et al. [5]. En su estudio, utilizó características basadas en la *build* actual y la anterior, para la *build* anterior usó:

- *prev_bl_cluster*: el cluster de la *build* anterior.
- *prev_tr_status*: el estado de la *build* anterior.
- *prev_gh_src_churn*: el número de líneas de código fuente cambiadas en la *build* anterior.
- *prev_gh_test_churn*: el número de líneas de código de test cambiadas en la *build* anterior.

Para la instancia de *build* actual, se usaron características como:

- *gh_team_size*: el tamaño del equipo.
- *cmt_buildfilechangelcount*: número de cambios en el archivo de script de construcción.
- *gh_other_files*: número de archivos no relacionados con el código fuente.
- *gh_src_churn*: número de líneas de código fuente cambiadas.
- *gh_src_files*: número de archivos de código fuente.
- *gh_files_modified*: número de archivos modificados.
- *gh_files_deleted*: número de archivos eliminados.
- *gh_doc_files*: número de archivos de documentación.
- *cmt_methodbodychangelcount*: número de cambios en el cuerpo del método.
- *cmt_methodchangelcount*: número de cambios en la cabecera del método.
- *cmt_importchangelcount*: número de cambios en los *imports*.
- *cmt_fieldchangelcount*: número de cambios en los atributos de clase.
- *day_of_week*: día de la semana del primer *commit* de la *build*.
- *cmt_classchangelcount*: número de clases cambiadas.
- *gh_files_added*: número de archivos añadidos.
- *gh_test_churn*: número de líneas de código de test cambiadas.

En [2] se reutilizaron gran cantidad de estas features mencionadas añadiendo las relacionadas con el histórico de ejecuciones de *builds*: tanto por ciento de compilaciones fallidas, incremento del *fail* ratio en la última *build* con respecto al ratio de la penúltima, porcentaje de *builds* exitosas desde la última *build* fallida, etc. Como vemos, se utilizan un gran número de *features* para la predicción, sin embargo, el objetivo no es la reducción de los costos de CI ni la importancia de cada una de ellas en relación con los *build failures*. El hecho de que se utilicen tantas *features* relacionadas con la *build* anterior, hace que predecir una *build* como fallida esté fuertemente relacionado con el resultado de la *build* anterior, que debería ser fallida. Esto hace que exista una limitación para la detección de los primeros *build failures* [9], ya que estos dependen mucho del resultado de la *build* anterior y, por definición, estarán siempre precedidos por una *build* exitosa.

Los *build failures* pueden categorizarse en una serie de tipos. Se ha identificado un total de 14 categorías de build failures, clasificadas según el tipo de error que las origina [15]. En el estudio se demostró que más del 80% de los errores se producían en la fase de ejecución de pruebas o *tests*. En el estudio se pretende identificar las causas que originan los *build failures*, para lo que se usan 16 métricas de la literatura y se descubre lo siguiente:

- Se respalda la hipótesis de que los *build failures* pueden aumentar con la complejidad de los cambios.
- Cambios objetivamente insignificantes pueden romper la *build*.
- Hay poca evidencia de que la fecha y hora de un cambio tenga un aspecto negativo o positivo en los resultados.
- Los autores que *commitean* menos frecuentemente tienden a causar menos *build failures*.
- Normalmente, las *builds* lanzadas a través de *pull requests* fallan con mayor frecuencia que las lanzadas por cambios directamente subidos a través de *push* a la rama principal.
- No existe evidencia que demuestre que trabajar en paralelo a un *pull request* afecte al resultado de la *build*.
- La mayoría de los errores se producen consecutivamente. Las fases más inestables de compilación generan fallos en la CI.

Todos estos resultados se obtuvieron a través de un estudio empírico con 14 proyectos de código abierto basados en Java que usan Travis CI. Por último, en [8] se realiza un estudio a gran escala con 3.6 millones de *builds* en el que se demuestra que factores como la cantidad de cambios en el código fuente, el número de *commits*, el número de archivos modificados o la herramienta de integración usada tienen una relación estadísticamente significativa con las compilaciones fallidas.

2.1.3. El costo de la Integración Continua. La implementación de la Integración Continua, a pesar de ofrecer numerosas ventajas, también supone un coste computacional y económico. Además del costo computacional que supone ejecutar la CI, debemos sumarle el costo del tiempo no productivo de los desarrolladores si estos no saben como proceder sin saber el resultado de la integración. Hilton et al. [6] estudiaron los beneficios y costes de aplicar CI en proyectos de código abierto. En su estudio, observaron que entre los proyectos *open-source* que no usaban CI, el principal motivo no era el costo técnico, si no que los desarrolladores no estaban familiarizados con CI. Otra de las razones de no usar CI era la falta de *tests* automáticos, un aspecto fundamental en CI. Además, calcularon el costo de mantenimiento de la CI, para lo que midieron el número de cambios en los archivos de configuración. Observaron que el número medio de modificaciones en archivos de configuración se elevaba a 12, siendo frecuente que se realizaran cambios en la configuración de CI. Una de las principales razones para estos cambios en archivos de configuración era la presencia de versiones obsoletas en las dependencias. Por último, observaron un hecho curioso con respecto al tiempo de ejecución medio de las *builds*: las *builds* exitosas son, en promedio, más rápidas que aquellas que fallan. Intuitivamente, podría esperarse lo contrario, ya que un error debería de interrumpir el proceso antes, aunque se necesita una mayor investigación para averiguar estas razones.

Klotins et al. [13] realizaron un trabajo con múltiples casos de estudio en el que encontraron que la aplicación de CI mejoraba notablemente los procesos de desarrollo internos en las empresas, sin embargo, se destaca la necesidad de evaluar las consecuencias de aplicar este tipo de desarrollo desde una perspectiva del cliente. El hecho de actualizar a los clientes a entregas continuas es un obstáculo importante ya que pueden existir acuerdos previos, y renegociar dichos acuerdos conlleva el riesgo de perder clientes y causar inestabilidad en la empresa. En el estudio se ha observado la necesidad de adoptar prácticas de CI para mejorar los procesos de desarrollo *software* en las empresas, sin embargo, estas se enfrentan a desafíos comunes en su implementación:

1. **Beneficios internos vs. externos:** se reconocen los beneficios internos de aplicar CI, como la agilización de los procesos y la liberación de recursos. Sin embargo, extender estos beneficios a los clientes y la adaptación de los modelos de negocio existentes representan un obstáculo mayor.
2. **Cultura organizacional:** la implementación de CI requiere cambios significativos en los procesos y en la cultura organizacional. Esto requiere compromiso por parte de los equipos de desarrollo y directivos.
3. **Clientes:** convencer a los clientes para aceptar entregas más frecuentes y compartir más datos es fundamental para sacar partido a las ventajas de CI. Sin embargo, los clientes pueden ofrecer resistencia al cambio y esto puede poner en peligro las relaciones con los mismos.

En [7] se presenta un estudio en el que se preguntó a trabajadores de la empresa *Atlassian* sobre sus percepciones sobre los fallos de CI. Entre los trabajadores, una gran mayoría (46%) consideraba como muy o extremadamente difícil resolver los fallos de CI, una minoría (13%) consideraba que no era difícil resolverlos, y el resto calificó la complejidad como moderada. Además, comentaban que los fallos de CI podían afectar tanto al flujo de trabajo individual como a la empresa. Los trabajadores notaban que estos fallos podían aumentar el tiempo de trabajo e incluso interrumpir el flujo de desarrollo. Otro impacto posible es la reducción de la productividad, ya que alguien tiene que dedicar tiempo a investigar por qué falló la *build* y solucionarlo. Este tipo de problemas puede ocasionar que las revisiones o las correcciones rápidas tarden más tiempo de lo esperado, poniendo en peligro el tiempo de lanzamiento (*release*). Además, se menciona que factores técnicos como humanos desempeñan un papel fundamental en la adopción de CI.

2.2. Trabajos relacionados

Existen numerosos estudios que buscan reducir el costo asociado a la Integración Continua mediante la creación de *predictors* [5,17,9,19,2,18,11,12,14]. Hassan et al. [5] estudiaron un total de 402 proyectos Java con información de 256,055 *builds*, procedentes de la base de datos de TravisTorrent. En su estudio se utilizan características extraídas directamente de la base de datos de TravisTorrent y otras propias, relacionando *features* propias de la *build actual* y la anterior. En su propuesta, primero se realiza una selección de *features* basada en la evaluación de la importancia de las mismas mediante el *Information Gain (IG)*. Con ello seleccionan las más discriminativas del conjunto de *features* (Sección 2.1.2). Posteriormente, construyen un clasificador que usa *random forest* para clasificar las *builds* en exitosas y fallidas. Este estudio fue el primer enfoque en utilizar técnicas de *Machine Learning* para predecir el resultado de la CI, sin embargo, no estaba centrado en reducir los costos asociados a la misma ni a los *build failures*, los casos positivos que más interés tienen.

En [17] se propone una solución novedosa que usa Programación Genética Multi-Objetivo, sin utilizar técnicas de aprendizaje automático. Su enfoque consiste en recopilar *builds* exitosas y fallidas de un proyecto, obtener información de *TravisTorrent* que contiene información sobre *builds* de *Travis CI* y, a partir de ahí, se toman esos datos como entrada para generar un conjunto de reglas predictivas que anticipen el resultado de la compilación de CI con la mayor precisión posible. Finalmente, entra en juego el algoritmo de programación genética multiobjetivo, el cual va generando un conjunto de soluciones, cada una de ellas con su conjunto de reglas de predicción, por ejemplo, una combinación de umbrales asignados a cada métrica. Dicha combinación de métricas-umbrales está conectada a operadores lógicos. Todas las muestras generadas en la solución son evaluadas usando dos objetivos: maximizar la tasa de verdaderos positivos y, minimizar la tasa de falsos positivos. En cada iteración se van cambiando los operadores, generando nuevas soluciones, hasta llegar a una condición de parada y devolviendo la solución óptima. En el estudio encontraron que características como el tamaño del equipo, la información de la última *build* o el tipo de archivos cambiados, pueden indicar el potencial fallo de una *build*. A pesar de obtener buenos resultados, solo se centran en 10 proyectos de lenguajes Java y Ruby, haciendo poco generalizables sus resultados. Además, el ratio de *failures* que presentan estos proyectos es relativamente elevado, lo que puede ocasionar que el algoritmo no sea tan efectivo en proyectos con ratios de *failures* bajos.

La piedra angular de nuestro estudio se basa en el trabajo de **Servant et al.** [9]. En este estudio, se propone un algoritmo que utiliza técnicas de *Machine Learning* para la predicción de CI, con el objetivo de reducir los costos asociados a la misma. Su teoría parte de dos hipótesis principales:

- H_1 : la mayoría de las *builds* devuelven un resultado exitoso. Por lo general, las *builds* exitosas son más numerosas que las fallidas. Es decir, siempre habrá mayor ratio de *builds* que pasan la CI que *builds* que fallan.
- H_2 : muchas *builds* fallidas en CI ocurren consecutivamente después de otra *build* fallida.

Teniendo en cuenta estas dos hipótesis encontramos que, si la primera es cierta, al saltarse todas aquellas *builds* que se predigan como exitosas, se reducirá el coste considerablemente. En caso de que la segunda sea cierta, entonces si se predice que las *builds* subsecuentes a una *build* fallida también fallarán, se predecirán correctamente una buena parte de las *builds* fallidas. En el estudio se introduce por primera vez el término de *first failures*, que hace referencia a las primeras *builds* que fallan en una subsecuencia de *builds failures*. En enfoques anteriores, existe una limitación para la predicción de estas primeras *builds* que fallan, ya que dependen fuertemente del resultado de la *build* anterior, haciendo que sean complicadas de predecir.

En el algoritmo, se utilizan *features* que son propias de la *build* y sirven para predecir *build failures* en un mismo proyecto y, por otro lado, se usan *project features*, que sirven para realizar lo que se denomina *cross-project predictions*. Con respecto a las *build features*, estas son propias de la *build* en cuestión y servirán para realizar predicciones sobre el mismo repositorio que estemos analizando. En el estudio, se mencionan algunas más, pero finalmente se seleccionan las siguientes:

- **SC**: el número de líneas de código fuente cambiadas desde la última *build*.
- **FC**: el número de archivos modificados desde la última *build*.
- **TC**: el número de líneas de *tests* cambiadas desde la última *build*.
- **NC**: el número de *commits* desde la última *build*.

En cuanto a las *project features*, estas son útiles cuando queremos predecir el resultado de la CI en un proyecto que tiene un número escaso de *builds*, bien porque sea reciente, no se hayan ejecutado en su duración gran número de *builds*, etc. Este último problema es lo que suele denominarse en sistemas de información como el “arranque en frío” (*cold start*), cuando no se puede extraer información útil para los usuarios debido a que todavía no se ha reunido suficiente información. Para ello, se usan modelos generados a partir de otros proyectos entrenados con estas *features*. En el estudio, se mencionan algunas *project features*, pero finalmente se seleccionan las siguientes:

- **TD**: el número medio de líneas en los casos de prueba por cada 1000 líneas ejecutables de código de producción.
- **PS**: el número medio de líneas de código fuente de producción ejecutale en el repositorio a lo largo de la historia de uso de CI en el proyecto.
- **PA**: la duración entre la primera y la última *build* del proyecto

A continuación, se explica de forma gráfica el funcionamiento de su algoritmo, llamado SmartBuildSkip:

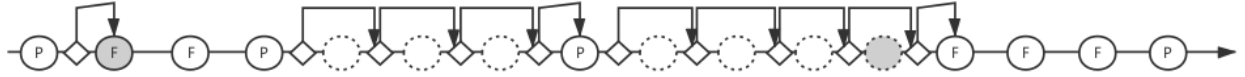


Figura 1. Línea temporal de SmartBuildSkip [9].

Cada círculo recoge el resultado real de la *build*. Los *first failures* están sombreados en gris. El símbolo de diamante indica que el predictor ha realizado una predicción. Las *builds* que se han saltado están indicadas con círculos discontinuos. Cuando una *build* se predice como *pass*, el algoritmo acumula los cambios de la *build* con la siguiente, lo que se indica mediante una flecha entre los círculos. Cuando se predice un *first failure*, *SmartBuildSkip* predice directamente como *fail* la *build* siguiente. Así, hasta que se encuentra un *pass*, lo que vuelve a reiniciar el algoritmo a la primera fase de predicción.

Se ha elegido este estudio [9] como base para nuestro trabajo porque es el primero que usa únicamente *features* que presentan una correlación significativa con las *build failures*. Además, con nuestro trabajo pretendemos indagar en la calidad de estas *features* y en mejorar los resultados obtenidos en el estudio original, bien mediante la adición de nuevas *features* o mediante la mejora del algoritmo de predicción.

Continuando con el orden cronológico del estado del arte, Saidani et al. [19] propone un predictor que utiliza Redes Neuronales Recurrentes (*RNN*) basadas en Memoria a Largo Plazo (*LSTM*). Su estudio se realiza como es habitual con diez proyectos de código abierto que usan el sistema de CI de *Travis CI*, sumando un total de 91330 *builds*. Estos revelan que este tipo de técnicas ofrecen mejores resultados que las de *Machine Learning*, obteniendo mejor rendimiento en términos de *AUC*, *F1-Score* y *accuracy* cuando se trata de validación entre proyectos. En [2] se propone una nueva solución en la que se usa un predictor que es dependiente del histórico de *builds* pasadas para poder hacer sus predicciones. En este estudio existen métodos de selección de *features* que seleccionan determinadas *features* en función del tipo de proyecto que se esté evaluando. En otro artículo, Ouni et al. [18] propone una solución de línea de comandos donde se consigue mejorar el estado del arte en términos de *F1-Score*. Sin embargo, en el estudio, solo se tiene en cuenta el estudio [5] comentado anteriormente, obviando todas las implementaciones posteriores y teniendo

una clara amenaza a la validez del mismo. Además, dada la arquitectura presentada, podemos apreciar que para la extracción de *features*, se utiliza un parseador de *HTML* con *Jsoup* y *Selenium*, lo cuál hace poco duradero el enfoque, ya que está fuertemente acoplado a la estructura *HTML* de *GitHub* y sus cambios.

Jin et al. [11] propone un nuevo predictor, *PreciseBuildSkip*, que mejora el ahorro de costo y la observación de *builds* fallidas, llegando a obtener valores de *recall* realmente buenos. En su implementación, incluyen dos variantes: la segura, que salva el 5.5 % de las *builds* y por lo general captura todas las construcciones fallidas, y una versión que mejora el ahorro de costos, salvando un 35 % de las *builds* mientras captura un 81 % de las observaciones de *builds* fallidas. Finalmente, Jin et al. [12] propone una solución que emplea técnicas de selección de *builds* y dos técnicas de selección de tests. Esta solución ejecuta seis técnicas existentes y luego usa los resultados como *features* para un clasificador *Random forest*. Entre sus resultados, se observa que:

- Se consiguió un mayor ahorro de costos con la mayor seguridad en comparación con técnicas anteriores.
- Tener un componente de selección de *tests* además de un componente de selección de compilación aumenta los ahorros de costos.
- Tener enfoques de selección de *tests* para predecir los resultados aumenta tanto la capacidad de ahorro de costos como la capacidad de observación de *build failures*.
- El algoritmo de bosque aleatorio es el que ofrece mejor rendimiento en la predicción.
- La *features* que recoge los fallos consecutivos fue la más efectiva para este enfoque.

Por otro lado, no existen proyectos documentados que usen técnicas de predicción de CI para el ahorro de costos, por lo que es complicado evaluar el impacto económico real que estas pueden causar. Liu et al. [14] utiliza simulación de procesos *software* y experimentos basados en simulación para evaluar el impacto de estos *predictors* de CI en un entorno más realista. Entre sus descubrimientos, vieron que existe poca diferencia entre los *predictors* del estado del arte y las estrategias aleatorias en términos de ahorro de tiempo. Sin embargo, en casos donde el ratio de *builds* fallidas es mayor, la estrategia aleatoria tendría un impacto negativo. Además, en proyectos donde la proporción de *failures* es muy pequeña, el uso de CI predictiva no es mucho mejor que saltar *builds* de forma aleatoria. A pesar de esto, se demuestra que el uso de técnicas de *predictive CI* puede ayudar a ahorrar el costo de tiempo para ejecutar CI, así como el tiempo promedio de espera antes de ejecutar la CI.

3. Objetivos y preguntas de investigación

En un estudio de carácter exploratorio como el que se propone, definir unos objetivos y preguntas de investigación se convierte en una tarea fundamental para la correcta orientación del trabajo. En este sentido, los objetivos nos permiten establecer una serie de metas a alcanzar, mientras que, las preguntas de investigación nos ayudan a centrar el estudio en aspectos concretos que queremos responder. Los objetivos de la investigación son los siguientes:

- **OB-1:** implementar un algoritmo de aprendizaje automático que genere un modelo predictivo (un *predictor*) basado en un conjunto de características *features* extraídas de las *builds*.
- **OB-2:** utilizar la *API* de GitHub para obtener datos relevantes sobre las *builds*, como su histórico, características asociadas, resultados anteriores de la integración continua.
- **OB-3:** desarrollar e implementar diferentes algoritmos de predicción con la selección de diferentes características con el objetivo de proporcionar múltiples opciones a la hora de predecir el resultado de la integración continua.
- **OB-4:** implementar una interfaz gráfica que sirva como punto de entrada de datos para el algoritmo de predicción y que permita visualizar los resultados obtenidos.

Antes de introducir las preguntas de investigación, es importante definir el significado de algunos términos clave para evaluar el desempeño de los modelos de predicción y la eficacia con la que cumplen su función. Cuando un algoritmo realiza una predicción, podemos encontrarnos con cuatro casos:

- *True Positive (TP)*: el modelo predice que la *build* fallará y, efectivamente, falla.
- *True Negative (TN)*: el modelo predice que la *build* pasará y, efectivamente, pasa.
- *False Positive (FP)*: el modelo predice que la *build* fallará, pero en realidad pasa.
- *False Negative (FN)*: el modelo predice que la *build* pasará, pero en realidad falla.

Valores reales	0	TP La build falla	FN La build falla
	1	FP La build pasa	TN La build pasa
0≡failure 1≡pass		0	1
		Valores predichos	

Figura 2. Matriz de confusión.

Con estos conceptos en mente, podemos definir las siguientes métricas de evaluación:

- *Accuracy*: mide la proporción de predicciones correctas realizadas por el modelo. Se calcula como la suma de los verdaderos positivos y verdaderos negativos dividida entre el total de predicciones realizadas.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

- *Precision*: mide la proporción de predicciones positivas correctas realizadas por el modelo. Se calcula como la suma de los verdaderos positivos dividida entre la suma de los verdaderos positivos y falsos positivos.

$$P = \frac{TP}{TP + FP} \quad (2)$$

- *Recall*: mide la proporción de instancias positivas que el modelo predice correctamente. Se calcula como la suma de los verdaderos positivos dividida entre la suma de los verdaderos positivos y falsos negativos.

$$R = \frac{TP}{TP + FN} \quad (3)$$

- *F1-score*: es la media armónica de *precision* y *recall*.

$$F1 = 2 \times \frac{P \times R}{P + R} \quad (4)$$

Las preguntas de investigación delimitan el alcance del estudio y ayudan a enfocar el trabajo en aspectos específicos del tema a investigar, evitando que nos desviemos hacia otras áreas no relevantes. Ayudan a clarificar qué se quiere lograr con la investigación y guían en el proceso metodológico, es decir, dependiendo de las preguntas de investigación, podremos determinar si necesitamos una metodología cuantitativa, cualitativa o mixta. Además, estas tienen una función estructural, ya que las secciones y capítulos siempre irán orientados a responder estas preguntas. A continuación se detallan las preguntas de investigación junto a las métricas usadas para su evaluación:

- **PI-1:** ¿Qué algoritmo de predicción produce los mejores resultados en la predicción automática del resultado de la integración continua?
 - **Métrica:** *accuracy*, *precision*, *recall* y *F1-score* del modelo.
- **PI-2:** ¿Qué características de las *builds* son más significativas en la predicción?
 - **Métrica:** importancia de cada *feature* a través de la interpretación de los coeficientes del modelo.

Finalmente, queda mencionar que en un modelo entrenado con una serie de *features*, los coeficientes del modelo representan la relación cuantitativa entre cada *feature* y la variable objetivo, en este caso, la predicción del resultado de la *build*. Por tanto, los coeficientes indican cómo se espera que cambie el valor de la predicción cuando la correspondiente *feature* cambia, manteniendo constante el resto de características.

4. Descripción del problema

La Integración Continua(CI) es una práctica esencial en el desarrollo de *software* moderno, que busca automatizar la fusión de cambios de código mediante la ejecución de pruebas automáticas. Esta práctica permite detectar errores de forma temprana y mejorar la calidad del *software*, facilitando una integración más frecuente y rápida del trabajo de los desarrolladores. Aunque la CI ofrece numerosas ventajas, su implementación conlleva significativos costos computacionales asociados, especialmente en empresas de gran tamaño donde se ejecutan un elevado número de *builds* diariamente. Estos costos no solo incluyen el costo de recursos de cómputo para ejecutar las *builds*, sino también el tiempo de espera al que se pueden enfrentar los desarrolladores durante el proceso de integración.

El objetivo principal de este trabajo es abordar el problema de la optimización de costos computacionales en CI mediante la predicción automática del resultado de las *builds*. Se utilizarán algoritmos de *Machine Learning* para predecir si una *build* concreta pasará o fallará antes de ser ejecutada. En especial, nos centraremos en predecir las *builds* que fallan, ya que son las más valiosas para los desarrolladores y de las cuales depende la puesta en producción o no del *software*. Este enfoque permite mantener la detección temprana de errores, que es el objetivo fundamental de CI, mientras se reduce el consumo de recursos computacionales.

Por lo tanto, este estudio se centra en desarrollar un modelo predictivo que permita optimizar el proceso de Integración Continua. Se pretende:

- Implementar un algoritmo de aprendizaje automático que permita predecir los resultados de las *builds* basándose en un conjunto de características. Se probarán varios algoritmos para determinar cuál es el que mejor resultados ofrece en nuestro problema.
- Analizar y seleccionar las características más significativas que influyen en la predicción. Para lo cual, se realizará un estudio de las *features* presentadas en otros estudios y se añadirán otras que puedan ser relevantes para nuestro problema.
- Evaluar la importancia que cada *feature* tiene sobre el modelo de predicción, lo que permitirá a los desarrolladores identificar qué aspectos de las *builds* son más significativos en la predicción de su resultado.

5. Detalles de la propuesta

En este apartado, se dará una visión global del contenido y funcionamiento de nuestra propuesta. Se realizará una descripción completa de nuestro enfoque, desde los conceptos de IA empleados hasta las herramientas y tecnologías empleadas para su resolución. En primer lugar, se detalla cómo se ha realizado la generación de los modelos, los tipos utilizados y sus características, así como cualquier concepto relacionado. Posteriormente, se describen las técnicas empleadas para evaluar el rendimiento de los modelos. Continuaremos con la explicación en detalle de las *features* empleadas, qué representan y cómo se ha realizado su cálculo. Finalmente, procederemos a explicar detalles técnicos de la implementación, como las tecnologías y recursos empleados, el procedimiento de extracción de datos, el entorno de ejecución, etc. En los siguientes apartados se describe con detalle cada uno de estos puntos.

5.1. Construcción del modelo

El Aprendizaje Automático (*Machine Learning*, ML) es un campo de la Inteligencia Artificial que se centra en desarrollar algoritmos y modelos que sean capaces de aprender y mejorar su desempeño en tareas específicas a partir de datos, sin la necesidad de ser explícitamente programadas. Los sistemas de ML identifican patrones en los datos y utilizan estos patrones para hacer predicciones o tomar decisiones sobre los datos. Podemos encontrarnos con dos tipos de aprendizaje automático:

- **Supervisado:** en este tipo de aprendizaje el modelo es entrenado utilizando un conjunto de datos etiquetados. En este contexto “etiquetado” significa que cada instancia en el conjunto de datos viene con una entrada (conjunto de características) y una salida conocida (etiqueta o valor objetivo). En nuestro problema, la entrada sería el conjunto de características (*features*) que vamos a usar y, la salida, el resultado de la integración continua (exitosa o fallida).
- **No supervisado:** en este tipo de aprendizaje el modelo es entrenado utilizando un conjunto de datos que no tiene etiquetas ni salidas predefinidas. A diferencia del supervisado, en el que se indica al modelo lo que debe predecir, en el no supervisado el modelo explora los datos en busca de patrones, estructuras, o relaciones ocultas.

Teniendo esto en cuenta, nos encontramos claramente ante un problema de aprendizaje supervisado, ya que tenemos las características propias de cada *build*, que sería el conjunto de entrada que proporcionamos al modelo y, por otro lado, los resultados de las ejecuciones, que conformarían las etiquetas o valores objetivo de nuestro problema.

Para poder responder a la pregunta de investigación **PI-1**, debemos explorar los distintos algoritmos de Aprendizaje Automático que existen y ver cuál de ellos ofrece mejores resultados en el problema que nos ocupa. En nuestro caso, nos encontramos con un claro problema de clasificación binaria en el ámbito del aprendizaje automático supervisado. En este contexto, tenemos dos clases posibles:

1. **Clase positiva** (fallo de la *build*): predicción de que la *build* fallará.
2. **Clase negativa** (éxito de la *build*): predicción de que la *build* tendrá éxito.

Por lo tanto, dado que nos encontramos con este tipo de problema, se ha decidido utilizar seis algoritmos de clasificación supervisada, entre los que nos encontramos:

1. **Árboles de Decisión (*Decision Trees*, DT).** Dividen el conjunto de datos en subconjuntos más pequeños y más simples, basándose en ciertas características o condiciones, que están representadas en un gráfico similar a un árbol. Cada nodo interno del árbol representa una característica (o atributo), y cada rama representa el resultado de la partición de los datos en función de esa característica. Las hojas del árbol contienen las etiquetas o valores objetivo.

2. **Bosques Aleatorios (*Random Forest, RF*)**. Es un algoritmo de aprendizaje supervisado que crea un conjunto de árboles de decisión durante el entrenamiento y realiza la predicción promediando las predicciones de cada árbol individual. Es una técnica de ensamblaje que combina múltiples modelos de aprendizaje para mejorar la precisión y la estabilidad del modelo.
3. **Regresión Logística (*Logistic Regression, LR*)**. Es un algoritmo de clasificación que se utiliza para predecir la probabilidad de que una variable dependiente pertenezca a una categoría particular. Aunque se llama regresión, en realidad es un algoritmo de clasificación binaria.
4. **Máquinas de Soporte Vectorial (*Support Vector Machines, SVM*)**. Es un algoritmo de clasificación que busca encontrar el hiperplano que mejor divide un conjunto de datos en dos clases. El hiperplano es la línea que maximiza el margen entre las dos clases. Si los datos no son linealmente separables, se puede utilizar un truco matemático llamado *kernel trick* para transformar los datos en un espacio de mayor dimensión donde sí sean separables.
5. **Vecinos más Cercanos (*K-Nearest Neighbors, KNN*)**. Es un algoritmo de clasificación que se basa en la idea de que los puntos de datos que son similares deben pertenecer a la misma clase. Para predecir la clase de un nuevo punto de datos, el algoritmo busca los k puntos de datos más cercanos en el conjunto de entrenamiento y asigna la clase más común entre esos vecinos.
6. **Redes Neuronales (*Neural Networks, NN*)**. Son un conjunto de algoritmos de aprendizaje automático que intentan imitar el funcionamiento del cerebro humano. Consiste en una red de nodos interconectados, llamados neuronas, que se organizan en capas. Está formada por una capa de entrada, una o varias capas ocultas y una capa de salida. Cada nodo está conectado a los demás y tiene su propia ponderación y umbral asociados. Concretamente, se ha utilizado un Perceptrón Multicapa, que es un tipo de red neuronal utilizado para tareas de clasificación. Cada neurona en un MLP se conecta a las de la capa anterior con pesos y aplica una función de activación a su suma ponderada.

Para realizar la predicción, no únicamente se le proporciona al modelo el conjunto de características de la *build* y se predice, si no que se sigue un procedimiento más complejo y que simula el funcionamiento de SmartBuildSkip [9]. Para comprenderlo, veamos el siguiente pseudocódigo:

Algorithm 1 *SmartBuildSkip con nuestra implementación*

```

1: Input: List of builds
2: Output: Predictions of builds outcomes
3: in_failure_sequence  $\leftarrow$  False ▷ Inicializar estado de secuencia de fallos
4: for each build in builds do
5:   if in_failure_sequence then
6:     Prediction  $\leftarrow$  Fail ▷ Predicción automática de fallo
7:     if build passes then ▷ Comprobar resultado
8:       in_failure_sequence  $\leftarrow$  False ▷ Error en la predicción, vuelve a predecir con ML
9:     end if
10:  else
11:    Prediction  $\leftarrow$  Machine Learning Prediction ▷ Predicción usando ML
12:    if Prediction = Pass then
13:      Accumulate changes with next build ▷ Se salta la build, acumulando cambios
14:    else
15:      if build fails then ▷ Comprobar resultado
16:        in_failure_sequence  $\leftarrow$  True ▷ Entra en predicción automática de fallo
17:      end if
18:    end if
19:  end if
20: end for

```

Como podemos observar, este algoritmo se divide en dos partes: una en la que predice automáticamente que la *build* fallará y otra en la que se realiza la predicción mediante el uso de un modelo de aprendizaje automático. Esta forma de proceder es debida a las dos hipótesis que comentamos en la Sección 2.2, que muchas *builds* fallan consecutivamente después de que otra haya fallado y que las *builds* exitosas siempre son más numerosas que las fallidas. Esto hace que nuestro algoritmo pueda saltarse mayor número de *builds* a la vez que captura una mayor cantidad de *build failures*. Consecuentemente, debido a la no ejecución de estas *builds*, se estará logrando una optimización de recursos computacionales.

5.1.1. Hiperparámetros. Cuando se define un algoritmo de clasificación, es importante seleccionar y ajustar con cuidado los hiperparámetros, ya que estos controlan aspectos clave del proceso de entrenamiento y pueden determinar la capacidad del modelo para generalizar a nuevos datos. Para la definición de nuestros algoritmos de clasificación, se ha utilizado la biblioteca *scikit-learn*, una popular biblioteca de *Python* para *Machine Learning* y análisis de datos. Por lo general, se han definido todos los algoritmos con los parámetros por defecto de la biblioteca *scikit-learn*, con las siguientes particularidades:

- En un conjunto de datos que contiene *builds*, donde el número de *builds* exitosas es mucho mayor que el de *builds* fallidas, estamos ante un problema de desequilibrio de clases. En este contexto, asignar distintos pesos a las clases es una estrategia útil para entrenar un modelo que sea más sensible en la detección de *builds failures*, a pesar de ser poco numerosas. Para ello, se ha definido un peso de 20:1 a favor de las *build failures* en Árbol de Decisión, Bosque Aleatorio, Regresión Logística y Máquinas de Soporte Vectorial.
- En Regresión Logística, se establece el número máximo de iteraciones que el algoritmo de optimización realizará antes de detenerse. En este caso se utiliza el valor 15.000 para asegurar que el algoritmo tenga suficiente tiempo para converger.
- En el algoritmo de Máquinas de Soporte Vectorial, hemos añadido el hiperparámetro que habilita el cálculo de probabilidades de pertenencia a cada clase. Cuando el parámetro *probability* está habilitado, el modelo ajusta de manera interna un modelo de probabilidad, permitiendo que las salidas del modelo sean las etiquetas de clase y sus probabilidades.
- Para el Perceptrón Multicapa (redes neuronales), se ha añadido, al igual que en Regresión Logística, un parámetro que indica el número máximo de iteraciones para entrenar la red neuronal. En este caso, se ha añadido con un valor de 15000.

Además de los parámetros antes mencionados, todos ellos tienen definida la semilla de reproducibilidad, que garantiza que los resultados sean reproducibles.

5.2. Evaluación del modelo

La evaluación de los modelos de clasificación tiene como objetivo medir y analizar el rendimiento de los modelos en la tarea de clasificación. Esta evaluación permite identificar con qué precisión el modelo puede predecir o clasificar nuevas instancias no vistas anteriormente, es decir, basándose en un conjunto de datos de prueba. Para realizar la evaluación de los modelos en nuestro problema, hemos seguido los siguientes pasos:

1. **División del conjunto de datos:** se divide el conjunto de datos, en este caso *features* de cada *build*, en dos partes: el conjunto de entrenamiento y el conjunto de prueba. El conjunto de entrenamiento se utiliza para entrenar el modelo, mientras que el conjunto de prueba se usa para evaluar su rendimiento. Dada la naturaleza de nuestro problema, no podemos hacer esta división de manera aleatoria, ya que las *builds* están relacionadas temporalmente entre sí, por lo que no sería realista realizar una predicción sobre una instancia antigua basándonos en *builds* que se hayan ejecutado recientemente. Por lo tanto, en nuestro caso se ha dividido el conjunto de datos igualmente en dos partes, pero siendo el conjunto de

entrenamiento más antiguo en su conjunto que el conjunto de prueba, que es más reciente. Por ejemplo, si se dividen los datos de entrenamiento y test en un 80 % y 20 % respectivamente, el conjunto de entrenamiento contendrá el 80 % de las *builds* más antiguas, mientras que el de prueba contendrá el 20 % de *builds* más reciente.

2. **Predicción:** una vez entrenado el modelo con el conjunto de entrenamiento, se realizan predicciones sobre el conjunto de prueba. Gracias a estas predicciones, podremos verificar si nuestros modelos se comportan bien frente a instancias nuevas o no vistas con anterioridad.
3. **Métricas de evaluación:** una vez realizadas las predicciones, hemos definido las métricas de evaluación, que nos ayudarán a determinar el rendimiento de nuestros modelos. Las métricas que hemos usado son: *accuracy*, *precision*, *recall*, *F1-score*, *confusion matrix* y *ROC curve*. Todas ellas han sido descritas en la Sección 3, exceptuando el área bajo la curva *ROC* (*Area Under the Curve*, *AUC*), que mide la capacidad de un modelo para distinguir entre clases positivas y negativas. Cuanto mayor sea el valor de *AUC*, mejor será el desempeño del modelo en la clasificación, ya que esta indica una mayor capacidad para separar correctamente las clases.

5.2.1. Validación Cruzada. La validación cruzada (*cross-validation*) es una técnica utilizada para la evaluación y selección de modelos, midiendo su rendimiento y capacidad de generalización. Su objetivo principal es evitar el sobreajuste (*overfitting*), que ocurre cuando un modelo aprende demasiado de los datos de entrenamiento y es incapaz de generalizar a nuevos datos. Esta consiste en dividir el conjunto de datos en k subconjuntos (*folds*), entrenar el modelo en $k - 1$ subconjuntos y evaluarlo en el subconjunto restante. Este proceso se repite k veces, de forma que cada subconjunto se utiliza una vez como conjunto de prueba. Al final, se promedian los resultados de las k iteraciones para obtener una estimación más precisa del rendimiento del modelo.

En nuestro caso particular, hemos tenido que hacer una ligera variación de esta técnica. En nuestro problema, la división del conjunto de datos en k subconjuntos no puede realizarse de manera aleatoria, ya que no tendría sentido realizar predicciones para instancias más antiguas utilizando para el entrenamiento instancias más recientes. Es decir, en nuestro problema existe una dependencia temporal y, por tanto, la secuencia y el orden de los datos son críticos para la validez de las predicciones. Los resultados y características de las *builds* están influenciados por las *builds* anteriores. Esto se debe a que cada *build* puede depender de cambios de código, configuraciones y otros factores que se acumulen o evolucionen con el tiempo. Por lo tanto, si se permite que las *builds* más antiguas predigan utilizando información de *builds* más recientes, se introduciría un sesgo temporal que no reflejaría el comportamiento real del sistema. Además de esto, usar datos de *builds* futuras para predecir *builds* antiguas no solamente introduce un sesgo, si no que también puede dar una falsa impresión de precisión del modelo, llevando a resultados artificialmente inflados de precisión durante esta fase de evaluación.

En nuestra implementación dividimos el conjunto en 11 partes de forma secuencial, de modo que el primer subconjunto contenga las *builds* más antiguas y el último las *builds* más recientes. El subconjunto de *folds* se recorre y en cada iteración, se entrena con la acumulación de subconjuntos anteriores, y se evalúa con el siguiente subconjunto, dejando el resto de subconjuntos sin utilizar. Siempre que se tengan k *folds* se realizarán $k - 1$ iteraciones, ya que en la última iteración se utilizará el último subconjunto como conjunto de prueba. Finalmente, dado que se habrán obtenido 10 resultados de las métricas de evaluación, se promedian para obtener una estimación más precisa del rendimiento del modelo.

A continuación, se presenta de forma gráfica el funcionamiento de nuestro algoritmo de *key-fold cross validation*:

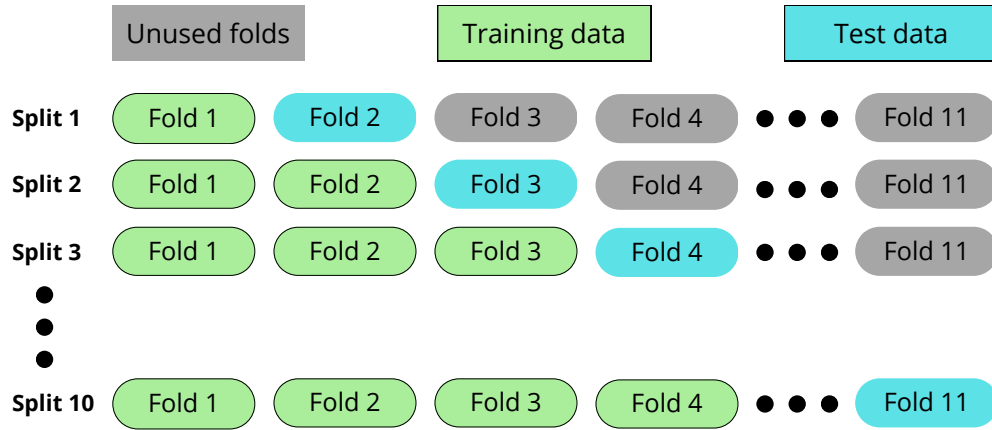


Figura 3. Funcionamiento de la técnica *key-fold cross validation*.

5.2.2. Umbral de decisión. La evaluación de los modelos a distintos grados de sensibilidad es especialmente relevante cuando se consideran problemas de clasificación con clases desbalanceadas o cuando el costo de los errores varía entre clases. En el problema que nos ocupa, se dan ambas condiciones: por lo general, el número de *builds* exitosas es muy superior al de *builds* fallidas y, el costo de predecir un falso negativo no es el mismo que el de predecir un falso positivo. Al ajustar la sensibilidad del modelo, se balancean las tasas de verdaderos positivos (*recall*) y falsos positivos, lo que permite optimizar el rendimiento del modelo para diferentes contextos y necesidades. El hecho de variar el umbral de decisión permite al desarrollador decidir hasta qué punto está dispuesto a obtener falsos positivos en las predicciones. Por ejemplo, si se asigna un umbral de decisión bajo, se aumentarán las tasas de verdaderos positivos, pero también aumentará el número de falsos positivos, es decir, el algoritmo predecirá un mayor número de veces que la *build* falla, haciendo al desarrollador ejecutar *builds* que realmente no fallaban. Por otro lado, si se asigna un umbral de decisión más conservador, se reducirán los falsos positivos, pero también con el riesgo de predecir alguna *build* como exitosa cuando realmente no lo es, teniendo el caso del falso negativo.

Muchos algoritmos de aprendizaje automático pueden predecir una probabilidad de pertenencia a una clase. Esto es útil porque proporciona una medida de la certeza o incertidumbre de una predicción y ofrece más detalles que solo predecir la etiqueta de la clase. En nuestro problema, es necesario convertir estas probabilidades en el valor de una clase. Esta conversión se basa en un parámetro llamado “umbral de decisión”. El valor por defecto de este umbral es 0,5 para probabilidades normalizadas en el intervalo $[0, 1]$. Por ejemplo, dadas las etiquetas de nuestro problema: 0 (la *build* falla) y 1 (la *build* pasa), si la probabilidad de que una *build* falle es igual o mayor a 0,5, se predecirá que la *build* es exitosa, mientras que si es menor a 0,5, se predecirá que la *build* falla.

En nuestra implementación, hemos evaluado el rendimiento de todos los modelos y variantes del mismo para valores del umbral de decisión comprendidos en el intervalo $[0, 1]$. Para ello, se han calculado las métricas de evaluación para cada valor del umbral y se han representado en gráficos para poder comparar el rendimiento de los modelos en función del umbral de decisión. De esta forma, se ha podido determinar cuál de los modelos es el que mejor rendimiento ofrece y cuál de las implementaciones es la que mejor resultado proporciona.

5.3. Features empleadas

Las *features* o características son las propiedades o atributos que se utilizan como entrada para entrenar un modelo. Son elementos fundamentales que permiten al modelo aprender patrones y tomar decisiones

basadas en los datos. Las *features* recogen información relevante de los datos, en este caso de las *builds*. La calidad y relevancia de las mismas afecta al rendimiento del modelo, por lo que escoger las adecuadas es fundamental. Al hacer la selección de las *features*, hay que tener en cuenta lo siguiente:

- Elegir *features* que sean relevantes y representativas del problema es importante para que el modelo sea preciso y eficiente. Además, ayuda a reducir la dimensionalidad del problema, lo que puede mejorar la eficiencia computacional y la interpretabilidad del modelo.
- Seleccionar gran cantidad de *features* puede llevar a un sobreajuste del modelo, donde el modelo aprende de patrones de datos ruidosos o irrelevantes. Esto puede llevar a un rendimiento bajo y a modelos menos generalizables

A continuación, se presentan las *features* estudiadas en nuestro problema.

Tabla 1. *Build features.*

<i>Clasificación</i>	<i>Feature</i>	<i>Descripción breve</i>
Consideradas en SmartBuildSkip	<i>Number Commits</i> (NC)	El número de <i>commits</i> desde la última <i>build</i> .
	<i>Files Changed</i> (FC)	El número de archivos modificados desde la última <i>build</i> , incluyendo archivos añadidos, modificados y eliminados.
	<i>Source Lines Changed</i> (LC)	El número de líneas de código modificadas desde la última <i>build</i> , incluyendo líneas añadidas y eliminadas.
	<i>Test Lines Changed</i> (LT)	El número de líneas de código de <i>test</i> modificadas desde la última <i>build</i> , incluyendo líneas añadidas y eliminadas.
Consideradas en SmartBuildSkip pero no utilizadas	<i>Performance Short</i> (PS)	La proporción de <i>builds</i> exitosas en las últimas cinco <i>builds</i> .
	<i>Performance Long</i> (PL)	La proporción de <i>builds</i> exitosas de todas las <i>builds</i> previas.
	<i>Time Frequency</i> (TF)	El intervalo de tiempo en horas desde la última <i>build</i> .
	<i>Failure Distance</i> (FD)	El número de <i>builds</i> exitosas desde la última <i>build</i> fallida.
	<i>Week Day</i> (WD)	El día de la semana en el que se ha ejecutado la <i>build</i> .
	<i>Day Hour</i> (DH)	La hora del día en la que se ha ejecutado la <i>build</i> .
Nuevas consideradas en este estudio	<i>Files Added</i> (FA)	El número de archivos añadidos desde la última <i>build</i> .
	<i>Files Modified</i> (FM)	El número de archivos modificados desde la última <i>build</i> .
	<i>Files Removed</i> (FR)	El número de archivos modificados desde la última <i>build</i> .
	<i>Source Lines Removed</i> (LR)	El número de líneas de código eliminadas desde la última <i>build</i> .
	<i>Source Lines Added</i> (LA)	El número de líneas de código añadidas desde la última <i>build</i> .
	<i>Unit Tests</i> (UT)	Si se han escrito pruebas unitarias desde la última <i>build</i> .
	<i>Commit Delay</i> (CD)	El tiempo transcurrido en horas entre <i>commits</i> de una misma <i>build</i> .

Como vemos en la Tabla 1, las *features* que se encuentran en la primera clasificación, son aquellas que realmente se han utilizado para el entrenamiento de los modelos en estudios previos [9]. Las *features* que se encuentran en la segunda clasificación son aquellas que se han considerado en SmartBuildSkip, pero que no se han llegado a utilizar para el entrenamiento de los modelos. Por último, las que se encuentran en la tercera clasificación son *features* nuevas propuestas en este estudio.

Para el cálculo de las *features* mencionadas en la Tabla 1, debemos tener en cuenta los siguientes aspectos:

- **Información para el entrenamiento:** cuando obtenemos las características de las *builds* directamente desde el repositorio de código fuente, tenemos acceso a la información completa de cada una de ellas. El cálculo de las *features* es sencillo y directo, ya que podemos acceder a la información de cada *build* que se ha ejecutado y extraer las características necesarias.
- **Cálculo de Features durante la predicción:** debemos recordar que, en el enfoque que planteamos, a veces no se ejecutan todas las *builds* que se han programado. Por lo tanto, no siempre se tiene acceso a la información real (*Ground Truth*) de haber ejecutado las *builds*. *Features* como el *performance short*, el *performance long* o el *failure distance* deben ser calculadas de forma diferente para poder simular un comportamiento práctico real.
- **Normalización:** es importante normalizar las *features* antes de entrenar el modelo. La normalización es un proceso que ajusta los valores de las *features* para que tengan una escala común. Esto es importante porque muchos algoritmos de aprendizaje automático son sensibles a la escala de las *features* y pueden dar resultados incorrectos si las *features* tienen escalas muy diferentes. En nuestra implementación, hemos utilizado dos métodos de normalización:
 1. Normalización *Min-Max*: como comentamos, es una técnica de escalado de datos que transforma los valores de un conjunto de datos dentro de un rango específico, normalmente en el intervalo $[0, 1]$, como en nuestro caso. La fórmula para la normalización *Min-Max* es la siguiente:

$$N_i = \frac{X_i - X_{min}}{X_{max} - X_{min}} \quad (5)$$

2. Normalización *Z-score* o estandarización: esta técnica transforma los valores de un conjunto de datos a una distribución con media 0 y desviación estándar 1. Para ello, se resta a cada valor la media de los datos y se divide entre la desviación estándar. La fórmula para la normalización *Z-score* es la siguiente:

$$N_i = \frac{X_i - \mu}{\sigma} \quad (6)$$

En nuestro caso, hemos utilizado un tipo de normalización u otro en función del modelo de clasificación que estemos utilizando. Por ejemplo, hemos usado normalización *Min-Max* para *k* vecinos más cercanos y para redes neuronales, y normalización *Z-score* para regresión logística y máquinas de soporte vectorial. Además, queda añadir que no se ha utilizado normalización en árboles de decisión ni en bosques aleatorios, ya que estos modelos crean reglas basadas en comparaciones entre valores de las características y no en sus magnitudes, lo que hace que la escala de las características no afecte a su rendimiento.

5.3.1. Cálculo de *features* durante la predicción. Para comprender bien este punto, vamos a diferenciar dos conceptos fundamentales: la *build* propuesta y la *build* ejecutada. La *build* propuesta es aquella que se desea predecir, independientemente de lo que prediga el algoritmo de predicción, y que por lo tanto todavía no ha sido ejecutada. La *build* ejecutada es aquella que previamente era una *build* propuesta (a predecir) y que ha sido ejecutada porque el algoritmo predijo que fallaría. Teniendo claros estos dos conceptos, pasemos a la explicación del cálculo de las *features* mencionadas anteriormente:

- **Performance Short (PS)**: para calcular este porcentaje de *builds* exitosas en las últimas cinco *builds*, cuando el algoritmo predice que la *build* propuesta pasará, se salta la ejecución de la *build* y se asume que el algoritmo ha acertado. Cuando el algoritmo predice que la *build* propuesta fallará, la *build* se ejecutará y, podremos ver el resultado real de la CI, considerando dicho valor.
- **Performance Long (PL)**: al igual que en el caso anterior, se calcula el porcentaje de *builds* exitosas de todas las *builds* propuestas hasta el momento. Si el algoritmo predice que la *build* propuesta pasará, se asume que ha acertado y si predice que fallará, se ejecuta la *build* y se considera el resultado real.
- **Failure Distance (FD)**: se calculará como el número de *builds* exitosas desde la última *build* fallida. Si el algoritmo predice que la *build* propuesta pasará, se asume que ha acertado y si predice que fallará, se ejecuta la *build* y se considera el resultado real.

A continuación, se representa de forma gráfica el cálculo de cada uno de ellos:



Figura 4. Cálculo del *Performance Short*.

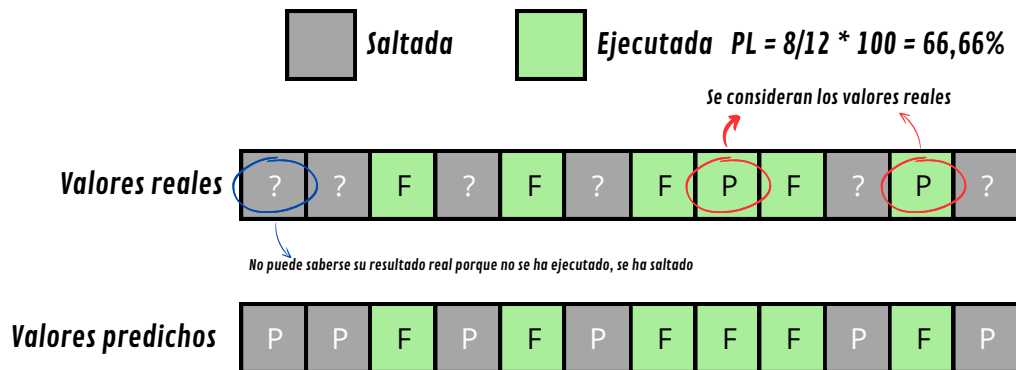


Figura 5. Cálculo del *Performance Long*.

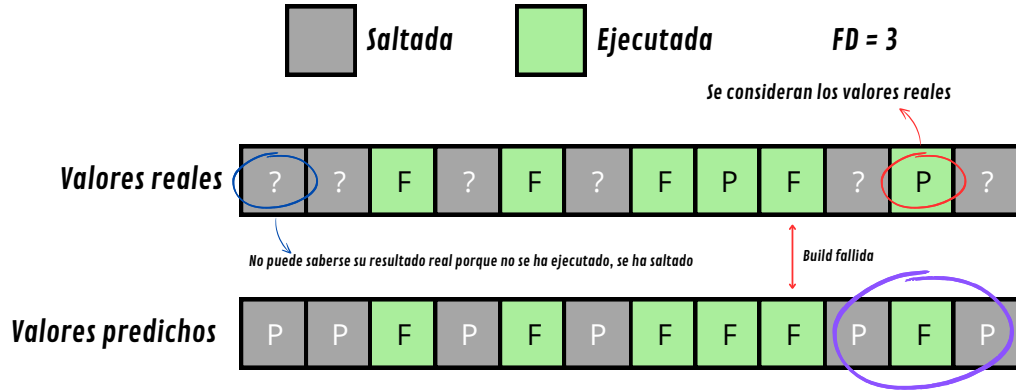


Figura 6. Cálculo del *Failure Distance*.

5.4. Interfaz gráfica

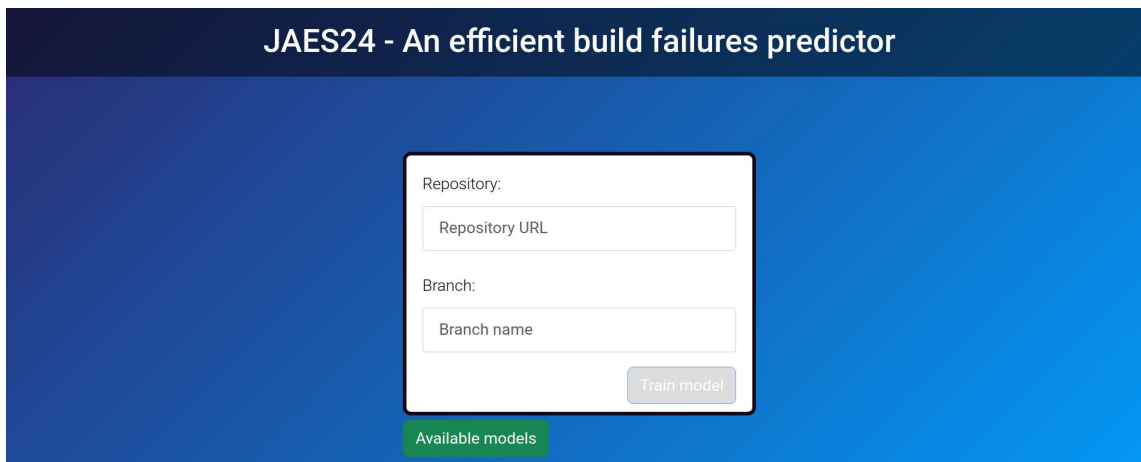
En nuestro proyecto, hemos implementado una interfaz gráfica simple que permite a los usuarios interactuar con la aplicación de forma intuitiva y sencilla, reduciendo así la interacción a bajo nivel con la aplicación. La interfaz se ha desarrollado con *Angular*, un framework de desarrollo de aplicaciones web desarrollado por *Google*. La interfaz consta de dos partes fundamentales, por un lado, un formulario que permite a los usuarios introducir la URL de un repositorio concreto de *GitHub* y la rama sobre la que desea predecir, y por otro lado, una tabla que muestra los datos de cada uno de los repositorios indicando si se encuentran disponibles sus modelos de predicción o no.

Cuando un usuario introduce la URL de un repositorio y la rama sobre la que desea predecir, nuestra aplicación internamente realiza los siguientes pasos:

1. Comprueba que la URL tiene un formato válido de acuerdo a las URLs de *GitHub*.
2. Comprueba si el repositorio y rama introducidos ya eran conocidos anteriormente por la aplicación. Si estos se encuentran en la base de datos, se devuelve un mensaje informativo y se detiene la ejecución. Si el repositorio no existe en la aplicación, se crea la estructura de directorios necesaria para el almacenamiento de los datos relativos a ese repositorio: *builds*, *features*, modelos, gráficos de evaluación, etc.
3. Si se continúa con la ejecución, se procede a la extracción de las *builds* del repositorio.
4. Una vez se han extraído las *builds*, se procede automáticamente a la extracción de *features* a partir de estas *builds*.
5. Finalmente, se procede de forma automática al entrenamiento de los modelos de acuerdo a las *features* seleccionadas en nuestro enfoque. Esto genera unos modelos de predicción listos para predecir.

Es importante mencionar que, todos los pasos anteriormente descritos se realizan de forma asíncrona, es decir, el usuario no tiene que esperar a que se complete un paso para poder seguir con el siguiente. Esto ofrece la posibilidad de introducir varios repositorios a analizar de forma simultánea. Además, cuando se ha realizado la extracción de *features*, el programa genera automáticamente los modelos de predicción para todos los algoritmos de clasificación estudiados en la Sección 5.1. Esto hace que el usuario no tenga que preocuparse de seleccionar un algoritmo de clasificación en concreto, ya que la aplicación automáticamente generará el modelo asociado.

A continuación, se muestra el formulario de entrada:



JAES24 - An efficient build failures predictor

Repository:
Repository URL

Branch:
Branch name

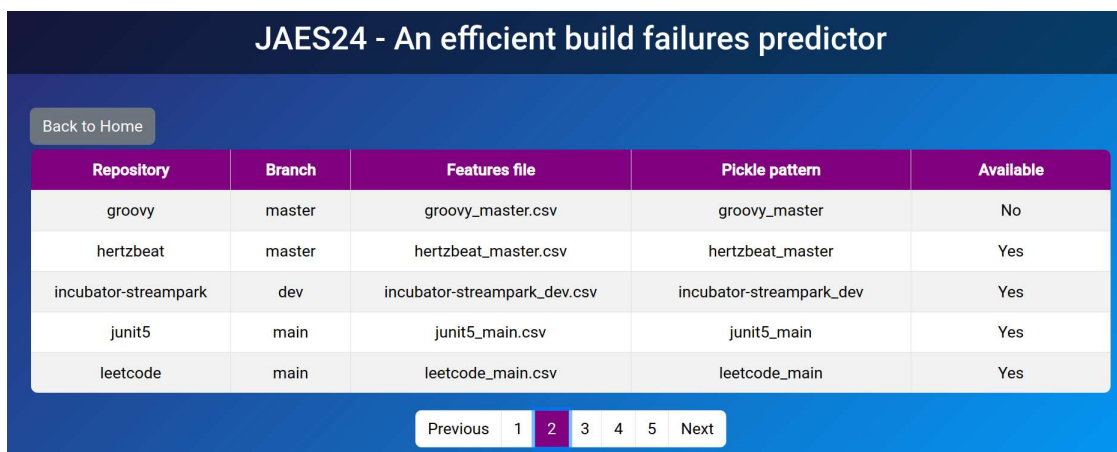
Train model

Available models

Figura 7. Formulario de entrada.

Como es lógico, la extracción de las *builds*, las *features* y el entrenamiento de los modelos, puede llevar un tiempo considerable. Este depende de la cantidad de *builds* a extraer y de la semántica de cada una de ellas, es decir, del tipo de evento que las origina y la complejidad interna de cada una. No es similar extraer *builds* originadas por un *pull request*, que probablemente contenga mayor número de *commits* y archivos modificados, que extraer *builds* originadas por un *push* simple. Por lo tanto, se ha diseñado una pequeña tabla que recoge información sobre el estado de cada uno de los repositorios introducidos. En ella, se muestra el nombre del repositorio, la rama sobre la que se desea predecir, el archivo donde se almacenan las *features* del repositorio, el patrón del nombre que seguirán los modelos generados y, la última y más importante, si el modelo se encuentra disponible o no.

A continuación, se muestra la tabla de repositorios:



JAES24 - An efficient build failures predictor

Back to Home

Repository	Branch	Features file	Pickle pattern	Available
groovy	master	groovy_master.csv	groovy_master	No
hertzbeat	master	hertzbeat_master.csv	hertzbeat_master	Yes
incubator-streampark	dev	incubator-streampark_dev.csv	incubator-streampark_dev	Yes
junit5	main	junit5_main.csv	junit5_main	Yes
leetcode	main	leetcode_main.csv	leetcode_main	Yes

Previous 1 2 3 4 5 Next

Figura 8. Tabla de repositorios disponibles y el estado de sus modelos.

5.5. Detalles técnicos de la implementación

En esta sección, se muestran detalles técnicos de la implementación. Se explican las tecnologías empleadas y las características que estas nos proporcionan para la resolución de nuestro problema, los recursos que utilizamos para extraer las *builds* de los repositorios y, finalmente, cómo se realiza la extracción de las *features* a partir de estas *builds*.

5.5.1. Tecnologías empleadas. Para la resolución de nuestro problema, hemos decidido utilizar las siguientes tecnologías:

- **Docker:** es una plataforma de código abierto que nos permite crear, desplegar y ejecutar aplicaciones en contenedores. Los contenedores son entornos de ejecución que por lo general son ligeros y portátiles, conteniendo todo lo necesario para que una aplicación pueda ejecutarse. En nuestra implementación, hemos creado tres contenedores: uno para albergar la base de datos, otro para el servidor *Flask* que contiene toda la lógica de nuestra aplicación y, por último, un contenedor para el cliente *Angular* donde residirá la interfaz gráfica.
- **Docker Compose:** es una herramienta que nos permite definir y ejecutar aplicaciones *Docker* de múltiples contenedores. Nos permite orquestar la comunicación y ejecución de los contenedores de forma sencilla y eficiente. Para ello, se utiliza un archivo de configuración con extensión *.yaml* donde se definen los servicios, redes, puertos, variables de entorno, volúmenes que se van a utilizar, etc. Concretamente, se han definido tres servicios, uno para la aplicación *Flask*, otro para *Angular* y otro para la base de datos.
- **Flask:** es un “microframework” de *Python* para el desarrollo de aplicaciones web. Se ha elegido porque es bastante ligero y fácil de usar, lo que nos permite centrarnos en la lógica de la aplicación sin tener que preocuparnos de detalles de la configuración. Además, nos permite una fácil gestión de las dependencias, algo fundamental para el uso de librerías como *Scikit-learn*, *Pandas* o *Matplotlib*.
- **Angular:** es un framework de desarrollo de aplicaciones web desarrollado por *Google*. Se ha elegido por su arquitectura modular y porque permite crear interfaces reutilizables.
- **PostgreSQL:** es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto. Se ha elegido por su fiabilidad, escalabilidad y por ser muy eficiente en la gestión de grandes volúmenes de datos. En nuestro caso, únicamente se ha usado para almacenar los valores de aquellos repositorios que han sido introducidos en la aplicación.

5.5.2. GitHub REST API. La *GitHub* REST API [1] es una interfaz de programación de aplicaciones (API) que permite a los desarrolladores interactuar de forma programática con los servicios de *GitHub*. Esta API da un soporte completo para realizar operaciones como:

- **Gestión de repositorios:** crear, modificar y eliminar repositorios.
- **Administración de issues y pull requests:** crear, modificar, comentar y cerrar *issues* o *pull requests*.
- **Gestión de usuarios y organizaciones:** obtener información de usuarios, modificar ajustes de la cuenta, manejar miembros de organizaciones o equipos de trabajo, etc.
- **Automatización de flujos de trabajo:** permite la integración de *GitHub* con otras aplicaciones y servicios, permitiendo lanzar flujos de trabajo de forma automática.

Todas estas operaciones, lógicamente, se podrán realizar siempre y cuando los usuarios estén correctamente autenticados y tengan los permisos necesarios en relación con los recursos sobre los que desea realizar la operación.

En nuestra solución, hemos realizado peticiones a esta API a través del uso de la librería *requests* de *Python*. Esta nos proporciona todo lo necesario para realizar peticiones HTTP de forma sencilla y eficiente. Además, para permitir un mayor número de peticiones por minuto a esta API, hemos utilizado un *token* de autenticación, que va incluido en la cabecera de cada petición que realizamos.

Tabla 2. *Endpoints de GitHub API REST usados.*

<i>Endpoint</i>	Descripción
<code>https://api.github.com/repos/OWNER/REPO/pulls</code>	Lista todos los <i>pull requests</i> de un repositorio específico.
<code>https://api.github.com/repos/OWNER/REPO/pulls/PULL_NUMBER</code>	Lista los detalles de un <i>pull request</i> dado su identificador numérico.
<code>https://api.github.com/repos/OWNER/REPO/pulls/PULL_NUMBER/commits</code>	Lista los <i>commits</i> de un <i>pull request</i> concreto.
<code>https://api.github.com/repos/OWNER/REPO/pulls/PULL_NUMBER/files</code>	Lista los archivos modificados en un <i>pull request</i> concreto.
<code>https://api.github.com/repos/OWNER/REPO/actions/runs</code>	Lista todas las <i>builds</i> ejecutadas en un repositorio.
<code>https://api.github.com/repos/OWNER/REPO/actions/runs/RUN_ID</code>	Lista una <i>build</i> específica dado su <i>run id</i> .
<code>https://api.github.com/repos/OWNER/REPO/commits/COMMIT_SHA</code>	Lista un <i>commit</i> específico dado su valor SHA.

En la Tabla 2, se muestran todos los *endpoints* de la API de *GitHub* que hemos utilizado en nuestra implementación. Si nos fijamos, existen partes en las urls que están marcadas con **OWNER**, **REPO**, **PULL_NUMBER**, **RUN_ID** y **COMMIT_SHA**. Estos valores son parámetros que se deben sustituir por los valores reales de los recursos sobre los que se quiere realizar la operación y que significan lo siguiente: **OWNER**, nombre del propietario del repositorio, ya sea una persona o una organización; **REPO**, nombre del repositorio del que se quiere obtener información; **PULL_NUMBER**, número identificador de un *pull request*; **RUN_ID**, identificador numérico de una *build*; **COMMIT_SHA**, valor SHA de un *commit*.

Obtener todas las *builds* que se han ejecutado en un repositorio es una tarea que puede parecer sencilla, ya que tenemos el quinto *endpoint* que se muestra en la Tabla 2, sin embargo, no es tan trivial como parece. Existen algunas restricciones como el límite máximo de resultados que la API puede devolver, el número de peticiones por minuto que se pueden realizar, la paginación de los resultados, etc., que hacen este proceso más complejo. En nuestra implementación, se utiliza lo que se denomina un *fine-grained personal access token*, que nos permite aumentar el número de peticiones por minuto a la API. Además, se ha implementado paginación para realizar las peticiones, ya que únicamente se pueden obtener un máximo de 100 *builds* por página y, además, al llegar a un límite de 1000 *builds*, la API no permite obtener más *builds* de forma directa. Finalmente, se ha considerado la posibilidad de que *GitHub* nos deniegue el acceso a la API por superar el límite diario de peticiones, para lo cual guardamos el estado de ejecución del programa, para reintentar pasado un tiempo definido la misma petición.

5.5.3. Procesamiento de *builds*. El procesamiento de las *builds* es el paso en el que se extraen las *features* de cada una de las *builds* extraídas en el paso anterior. Para ello, una vez las *builds* han sido extraídas del repositorio, estas se encuentran organizadas en un formato JSON y por ficheros correspondientes a cada uno

de los meses en los que se han ejecutado. Por ejemplo, para un proyecto llamado *junit5*, sus *builds* quedarían organizadas de la siguiente forma:

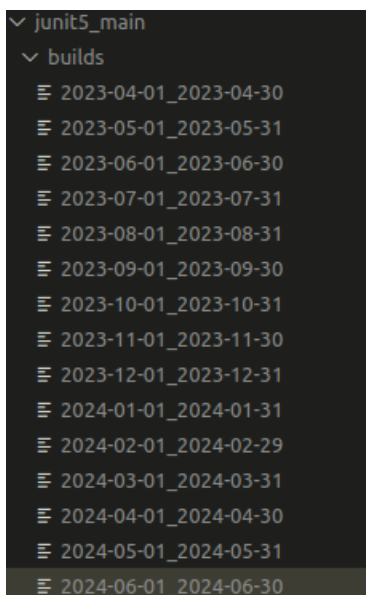


Figura 9. Organización de las *builds* de un proyecto.

Como vemos, por cada mes en el que se han ejecutado *builds*, se ha creado un fichero JSON conteniendo la información de todas las *builds* ejecutadas en ese mes. Además, por formato y organización, cada archivo está nombrado con el día de inicio y fin del mes concreto al que pertenece.

A pesar de tener extraída la información de las *builds*, muchos de los datos necesarios para el cálculo de algunas *features* no se encuentran disponibles directamente en la información extraída. *Features* como el número de *commits* (NC), el número de archivos modificados (FC), el número de líneas de código modificadas (LC), el número de líneas de código de *test* modificadas (LT), el número de archivos añadidos (FA), el número de archivos modificados (FM), el número de archivos eliminados (FR), el número de líneas de código eliminadas (LR), el número de líneas de código añadidas (LA), si se han escrito pruebas unitarias (UT) o el tiempo transcurrido entre *commits* de una misma *build* (CD), son necesarias inferirlas a partir de la información que se tiene.

6. Experimentación

En este apartado se realiza una explicación en profundidad de las pruebas que se han realizado para validar y verificar la implementación descrita en apartados anteriores. Esta sección incluye la explicación en detalle de todo el diseño experimental llevado a cabo, incluyendo las técnicas a comparar, la descripción del *dataset* utilizado para la experimentación, el procedimiento seguido para realizar el entrenamiento y prueba de los modelos, incluyendo el *key-fold cross-validation*, la descripción del cálculo de las distintas métricas de evaluación y, finalmente, los resultados obtenidos y su correspondiente análisis. Además, en esta última sección, se dará respuesta a las preguntas de investigación planteadas en la Sección 3, lo que constituye el objetivo principal de este estudio.

6.1. Diseño experimental

En esta sección se describe el diseño experimental llevado a cabo para la validación de la implementación propuesta.

6.1.1. Técnicas a comparar. En este estudio, se propone comparar tres técnicas de predicción diferentes. Por un lado, la propuesta en *SmartBuildSkip* [9] que usa bosques aleatorios en su algoritmo de predicción y, por otro lado, las dos técnicas propuestas en este estudio, JAES24. A continuación, se describen las tres técnicas a comparar:

- **SBS-Within:** técnica propuesta en *SmartBuildSkip* que utiliza bosques aleatorios en su algoritmo de *Machine Learning* para la predicción. Esta técnica tiene dos fases principales: una en la que el algoritmo predice de forma automática que la *build* fallará, por encontrarse en una secuencia de *build failures*, y otra en la que utiliza predicción mediante *Machine Learning*.
- **JAES24-Within:** técnica propuesta en este estudio que utiliza como modelo de ML árboles de decisión. El algoritmo de predicción se basa en la implementación propuesta de *SmartBuildSkip*, pero modificando el conjunto de *features* empleado: TF, NC, FC, LC, LA, LR, LT, WD y DH, el cual captura eventos temporales al realizar las *builds* y desgana los cambios realizados en la misma. Además, en este algoritmo se realiza la acumulación de *features*, lo cual permite acumular cambios para la predicción cuando una *build* es saltada por el algoritmo.
- **JAES24-Without:** técnica propuesta en este estudio que utiliza como modelo de ML árboles de decisión. Este modelo emplea el mismo conjunto de *features* descrito en el punto anterior, sin embargo, no utiliza como base la implementación realizada *SmartBuildSkip*, por lo que no se realiza acumulación de valores en las *features* cuando una *build* es saltada por el algoritmo, ni tampoco tiene dos fases de predicción. En este caso, simplemente se van realizando predicciones de forma individual para cada *build*.

Dependiendo del contexto sobre el que se realicen las predicciones, puede ser más adecuado utilizar una técnica u otra. Por ejemplo, si el proyecto sobre el que realizamos predicciones suele tener muchos *build failures* de forma consecutiva, será más beneficioso utilizar las técnicas *SBS-Within* o *JAES24-Within*, ya que estas tienen una fase en la predicción especialmente diseñada para detectar secuencias de *build failures*. Por otro lado, si el proyecto tiene *build failures* producidos más aleatoriamente a lo largo del proyecto, será más beneficioso utilizar la técnica *JAES24-Without*, ya que esta no tiene en cuenta secuencias de *build failures* en su algoritmo de predicción.

Realizar una comparación entre estas tres técnicas nos da una visión más amplia de los resultados obtenidos en este estudio, permitiéndonos identificar cual de ellas es más adecuada en términos de precisión, eficiencia o adaptabilidad a las características del problema. El hecho de considerar varias técnicas mejora la validez interna del estudio, ya que se descarta que los resultados sean atribuibles a una sola metodología.

6.1.2. Descripción del dataset. Para realizar la experimentación, se han utilizado 20 proyectos de código abierto disponibles de forma pública en *GitHub*. Todos los proyectos están basados en *Java* y han sido seleccionados de forma manual. Con el objetivo de tener una experimentación diversa, se han tenido en cuenta dos escenarios posibles:

1. **Escenario 1:** engloba a todos aquellos proyectos donde la CI falla con muy poca frecuencia. En nuestra solución, se ha considerado a todos estos proyectos como “proyectos difíciles”, y serán todos aquellos en los que la proporción de *build failures* es inferior al 10 % con respecto a las *builds* exitosas.
2. **Escenario 2:** engloba a todos aquellos proyectos donde la CI falla con una frecuencia “normal”. Consideramos como “proyectos normales” a todos aquellos en los que el porcentaje de *build failures* se encuentra comprendido entre el 10 % y el 25 % con respecto a las *builds* exitosas.

Al incluir tanto proyectos difíciles como normales, se pretende obtener una visión más amplia de la efectividad de nuestro algoritmo en diferentes condiciones y escenarios. Este enfoque es beneficioso por varias razones:

- **Evaluación en escenarios específicos:** separar ambos escenarios nos permite analizar el comportamiento del modelo en cada tipo de proyecto de forma más precisa.
- **Fortalezas y debilidades:** al evaluar cada conjunto de proyectos individualmente, podemos identificar fortalezas y debilidades del modelo en cada escenario.
- **Adaptabilidad del modelo:** al evaluar el modelo en diferentes escenarios, podemos observar si el modelo es capaz de adaptarse bien a ambos tipos de proyectos, o si realmente necesita un enfoque diferente para cada uno de ellos.

Se ha decidido catalogarlos como “proyectos difíciles” y “proyectos normales” según la capacidad que tendrá el algoritmo de aprender de los *build failures* en cada tipo de proyecto. En los proyectos difíciles, donde se espera que la proporción de *build failures* sea mucho menor a la de *builds* exitosas, el algoritmo tendrá que aprender de un número mucho menor de ejemplos de fallos, haciéndole la tarea de aprendizaje más difícil. Por otro lado, en los proyectos normales, donde la proporción de *build failures* es más alta, el algoritmo tendrá más ejemplos de *build failures* sobre los que aprender, esperando que la capacidad de aprendizaje sea mayor.

Como hemos mencionado anteriormente, se han seleccionado 20 proyectos de código abierto disponibles en *GitHub*. Cada uno de ellos ha sido seleccionado de forma manual, y se ha intentado que el número de *builds* sea lo más similar posible entre ellos. A continuación, se muestran dos gráficos que describen la proporción de *build failures* en cada uno de los proyectos seleccionados, tanto para proyectos difíciles como para proyectos normales.

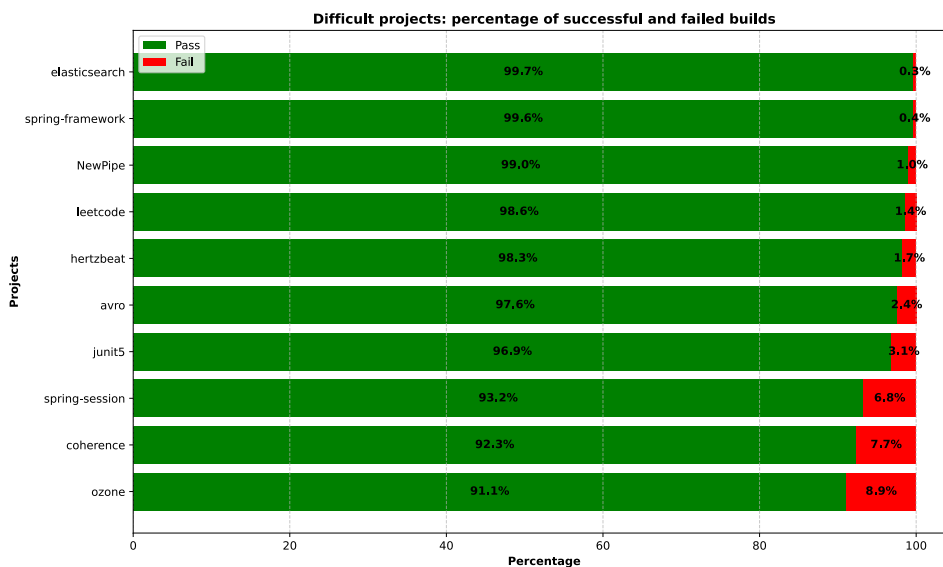


Figura 10. Proporción de *build failures* en proyectos difíciles

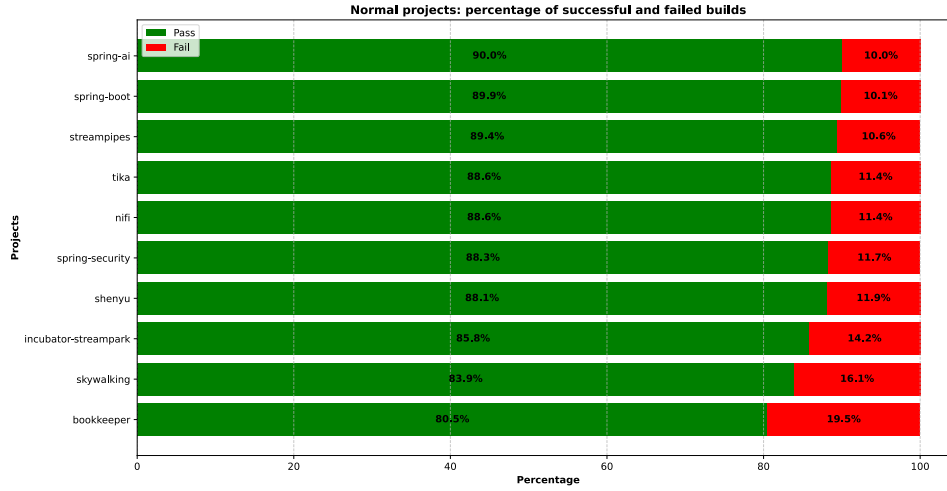


Figura 11. Proporción de *build failures* en proyectos normales

Como vemos, en los proyectos difíciles la proporción de *build failures* es inferior al 10 %, mientras que en los proyectos normales se encuentra entre el 10 % y el 25 %. En ambos casos, se ha intentado que el número de *builds* sea lo más similar posible entre los proyectos seleccionados.

6.1.3. Procedimiento para el entrenamiento, prueba y validación cruzada La evaluación de los modelos de clasificación busca medir y analizar el desempeño de los mismos en la tarea de clasificación. Esta evaluación permite determinar la capacidad del modelo para predecir o clasificar correctamente nuevas instancias no observadas previamente, utilizando un conjunto de datos de prueba.

Para cada uno de los 20 proyectos seleccionados, se ha realizado lo siguiente:

1. **Conjunto de entrenamiento y prueba:** se ha dividido el subconjunto de *builds* en dos subconjuntos, uno de entrenamiento y otro de prueba. El porcentaje de entrenamiento y prueba se ha fijado en 80 % y 20 % respectivamente. Se ha tenido especial cuidado en seleccionar el 80 % de las *builds* más antiguas para el subconjunto de entrenamiento y, el 20 % de las *builds* más recientes para el subconjunto de prueba. Esto es así por la dependencia temporal que existe en este tipo de problemas, donde no tiene sentido realizar predicciones basándonos en información futura. Además, para realizar este entrenamiento y prueba, se ha escogido el umbral de decisión por defecto, con valor 0,5.
2. **Key-fold cross-validation:** se ha utilizado la técnica de validación cruzada *key-fold cross-validation* para evaluar el rendimiento de los modelos de clasificación propuestos. Para cada proyecto, se han seleccionado sus *builds* y se han creado 11 subconjuntos a partes iguales. A continuación, se ha realizado la validación cruzada de la siguiente forma: se han ido seleccionando los *folds* de forma acumulativa para realizar el entrenamiento, mientras que con el *fold* siguiente se ha realizado la parte de *test*. Cuando este proceso se ha realizado con cada uno de los *folds*, habrá un total de 10 resultados disponibles, calculando así posteriormente la media de cada una de las métricas obtenidas en cada *fold*. Este proceso se ha realizado iterativamente para un total de 6 umbrales de decisión, que van desde el 0 hasta el 1, con incrementos de 0,2.

6.1.4. Métricas de evaluación Las métricas de evaluación constituyen una parte esencial en nuestro estudio, ya que gracias a ellas podemos medir y analizar el rendimiento de las técnicas propuestas. En este estudio, se han considerado cinco métricas de evaluación diferentes: *accuracy*, *precision*, *recall*, *F1-score* y *AUC-ROC*. Estas métricas ya fueron descritas en la Sección 3, por lo que no se describirán de nuevo en este apartado.

Es importante mencionar que, para realizar el cálculo de estas métricas, hemos hecho uso de métodos de la librería *metrics* de *scikit-learn*. Esta librería está especialmente diseñada para el cálculo de estas métricas, y nos permite obtener los resultados de forma rápida y sencilla. A continuación, vamos a describir cómo se realiza el cálculo de cada una de ellas en el procedimiento de **entrenamiento y prueba**:

- *Accuracy*: a partir de las predicciones realizadas por el modelo para el conjunto de *test* (que supone el 20 % de las *builds*), se calcula a partir de los valores de las etiquetas reales de las *builds* y las predicciones realizadas por el modelo (1).
- *Precision*: a partir de las etiquetas reales del conjunto de *test* y las etiquetas predichas por el algoritmo, se calcula la precisión del modelo (2). Hemos tenido que indicar cuál es la clase positiva en nuestro problema (clase 0) e incluir el valor por defecto que asigna cuando se produce una división entre 0, en el caso de que la suma de verdaderos positivos y falsos positivos sea 0.
- *Recall*: a partir de las etiquetas reales del conjunto de *test* y las etiquetas predichas por el algoritmo, se calcula el *recall* del modelo (3). Al igual que en el caso de *precision*, hemos tenido que indicar cuál es la clase positiva en nuestro problema (clase 0) para que el cálculo sea correcto y, añadir el valor por defecto en caso de división entre 0, en este caso, cuando la suma de verdaderos positivos y falsos negativos sea 0.
- *F1-score*: esta métrica, podría calcularse directamente a partir de los valores de *precision* y *recall*, sin embargo, hemos decidido utilizar el método que nos proporciona la librería mencionada anteriormente, que usa al igual que los anteriores las etiquetas reales del conjunto de *test* y las etiquetas predichas por el algoritmo, además de indicar cuál es la clase positiva y el valor en caso de división entre 0.

En el caso de **key-fold cross-validation**, el cálculo en sí de estas métricas es similar al descrito anteriormente, con la característica de que por cada *fold* de *test* evaluado, se almacenan las predicciones, hasta que finalmente se tengan los resultados de $k - 1$ *folds*, en este caso 10 *folds*. Una vez obtenidos estos resultados, estos serían las etiquetas predichas por el algoritmo.

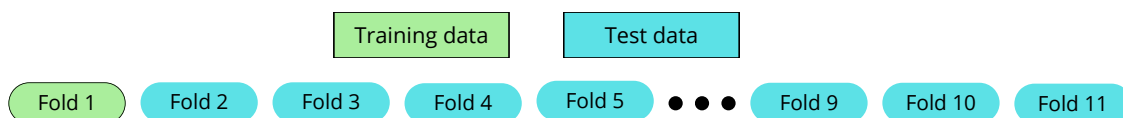


Figura 12. Acumulación de predicciones en *key-fold cross-validation*

Como vemos en la Figura 12, hemos ido acumulando las predicciones hechas en cada uno de los *folds* de *test*, para luego calcular las métricas en función de ellas. Podemos observar, además, que el primer *fold* no se usa para *test*, usándose únicamente para entrenamiento. Sucede algo similar con el último *fold*, que no es usado para entrenamiento y se usa únicamente para *test*.

6.2. Resultados

7. Amenazas a la validez

En cualquier trabajo de investigación, es crucial reconocer los factores que pueden comprometer la validez de los resultados obtenidos. Estos factores, comúnmente conocidos como amenazas a la validez, representan posibles fuentes de sesgo o error que pueden afectar a la precisión y generalización de las conclusiones del estudio. Al identificar y discutir estas amenazas, no solo hacemos más transparente nuestro estudio, sino que también proporcionamos una base crítica para que los lectores evalúen la fiabilidad de los hallazgos. En este apartado se examinan las principales amenazas que podrían afectar a la validez de constructo, validez interna y validez externa del presente estudio, con el fin de contextualizar los resultados y ofrecer una interpretación más sólida y matizada.

Concretamente, se consideran tres tipos de amenazas a la validez: validez de constructo, validez interna y validez externa. A continuación, explicamos en qué consiste cada una de ellas:

- **Validez de Constructo:** son aquellos factores que ponen en duda si el estudio está realmente midiendo lo que pretende medir, es decir, si los conceptos definidos en la investigación son evaluados correctamente. Estas amenazas pueden afectar la interpretación de los resultados en relación con los constructos teóricos.
- **Validez Interna:** se refiere a las amenazas que pueden afectar la relación causal entre las variables independientes y dependientes. En otras palabras, se trata de factores que pueden distorsionar la interpretación de la relación entre las variables manipuladas y las variables de respuesta.
- **Validez Externa:** son aquellas amenazas que pueden afectar la generalización de los resultados obtenidos en el estudio. Estas amenazas se refieren a la capacidad de generalizar los resultados a otros contextos, poblaciones o situaciones distintas a las evaluadas en el estudio.

A continuación, se detallan las amenazas a la validez identificadas en el presente estudio y se discuten las estrategias utilizadas para mitigar su impacto en los resultados obtenidos.

7.1. Validez de Constructo

En nuestro estudio, utilizamos métricas como indicadores para representar la cantidad de *build failures* detectados en CI. Algunas métricas, como el *accuracy*, puede ser engañosa en conjuntos de datos que están muy desbalanceados, ya que puede reflejar un alto valor incluso cuando el modelo falla en identificar los *build failures*. Para mitigar este problema, hemos utilizado métricas como *precision*, *recall* y *F1-score*, que proporcionan una visión más equilibrada del rendimiento del modelo. Además, debemos recordar que nuestro enfoque persigue reducir el costo computacional asociado a la CI mediante la detección de *build failures*, sin embargo, no hemos considerado valores económicos específicos asociadas a la ejecución de CI o a la reparación de errores en el conjunto específico que hemos usado para la experimentación.

7.2. Validez Interna

Para salvaguardar la validez interna, hemos realizado pruebas exhaustivas de nuestros procedimientos de evaluación en subconjuntos del *dataset* empleado durante el desarrollo. Nuestro análisis puede estar influenciado por información incorrecta en nuestro dataset. Por esto, siempre hemos considerado ramas principales de los proyectos, filtrando así valores atípicos más comunes en ramas secundarias o de desarrollo. Además, nuestros resultados podrían verse afectados por pruebas inestables (*flaky tests*) que causan fallos de manera errática o falsa.

Por último, la validación cruzada cronológica ha sido empleada para preservar el orden temporal de los datos y evitar que futuras *builds* se incluyan en el conjunto de entrenamiento. Esto garantiza que los resultados

sean más representativos de cómo el modelo funcionaría en condiciones reales, donde el entrenamiento se realiza con datos pasados y la prueba se realiza con datos futuros. Aunque la validación cruzada cronológica es una alternativa adecuada a la validación cruzada estándar, la decisión de utilizar este enfoque ha sido alineada con el objetivo de evaluar las técnicas en un escenario que respete la secuencia temporal.

7.3. Validez Externa

Para aumentar la validez externa, hemos seleccionado 20 proyectos ampliamente conocidos de *GitHub*. Todos son proyectos de código abierto y que tienen un número considerable de *forks* y estrellas. Como dato objetivo, el proyecto que menos *forks* y estrellas tiene es *coherence*, de *Oracle*, con 70 *forks* y 427 estrellas, y el que más, *spring-boot*, con 40,500 *forks* y 74,400 estrellas. Los proyectos seleccionados son todos proyectos *Java*, ya que necesitamos hacer una evaluación justa con otras técnicas. A pesar de que este lenguaje de programación es uno de los más utilizados en la actualidad, lenguajes de programación diferentes pueden tener hábitos de Integración Continua distintos, pudiendo provocar resultados ligeramente diferentes a los obtenidos en este estudio.

Finalmente, haber elegido proyectos de empresas muy conocidas puede haber limitado la diversidad del conjunto de datos, ya que estos proyectos suelen seguir procesos de desarrollo de *software* altamente estructurados. Esto podría no reflejar la realidad de proyectos más pequeños o menos formales, pudiendo limitar la aplicabilidad de los resultados a repositorios menos formales o con menos recursos.

8. Conclusiones y trabajos futuros

Referencias

1. Github rest api documentation, 2008. [Online; accessed 4-Sep-2024].
2. Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. Buildfast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20*, page 42–53, New York, NY, USA, 2021. Association for Computing Machinery.
3. Omar Elazhary, Colin Werner, Ze Li, Derek Lowlind, Neil Ernst, and Margaret-Anne Storey. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*, PP:1–1, 03 2021.
4. Martin Fowler and Matt Foemmel. Continuous integration, 2006. [Online; accessed 2-Aug-2024].
5. Foyzul Hassan and Xiaoyin Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '17*, page 157–162. IEEE Press, 2017.
6. Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE '16*, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery.
7. Yang Hong, Chakkrit Tantithamthavorn, Jirat Pasuksmit, Patanamon Thongtanunam, Arik Friedman, Xing Zhao, and Anton Krasikov. Practitioners' challenges and perceptions of ci build failure predictions at atlassian. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024*, page 370–381, New York, NY, USA, 2024. Association for Computing Machinery.
8. Md Rakibul Islam and Minhaz F. Zibran. Insights into continuous integration build failures. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17*, page 467–470. IEEE Press, 2017.
9. Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 13–25, 2020.
10. Xianhao Jin and Francisco Servant. Cibench: A dataset and collection of techniques for build and test selection and prioritization in continuous integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 166–167, 2021.
11. Xianhao Jin and Francisco Servant. Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. *J. Syst. Softw.*, 188(C), jun 2022.
12. Xianhao Jin and Francisco Servant. Hybridcisave: A combined build and test selection approach in continuous integration. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023.
13. Eriks Klotins, Tony Gorschek, Katarina Sundelin, and Erik Falk. Towards cost-benefit evaluation for continuous software engineering activities. *Empirical Softw. Engg.*, 27(6), nov 2022.
14. Bohan Liu, He Zhang, Weigang Ma, Gongyuan Li, Shanshan Li, and Haifeng Shen. The why, when, what, and how about predictive continuous integration: A simulation-based investigation. *IEEE Transactions on Software Engineering*, 49(12):5223–5249, 2023.
15. Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17*, page 345–355. IEEE Press, 2017.
16. Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. *Empirical Softw. Engg.*, 28(2), mar 2023.
17. Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.
18. Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*, page 1530–1534, New York, NY, USA, 2021. Association for Computing Machinery.
19. Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engg.*, 29(1), may 2022.