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

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* [1]. Han surgido numerosos enfoques para reducir el costo asociado a la carga computacional evitando ejecutar construcciones que se espera que sean exitosas [2]. 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* [3]. 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 [2,4,5,6,7,8], y se ha propuesto una implementación, *JAES24*, que busca contribuir a las mismas. Este nuevo enfoque amplia 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 desarrollarán unas conclusiones sobre lo resultados obtenidos y se propondrán 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 [1]. Several approaches have emerged to reduce the cost associated with computational load by avoiding running builds that are expected to be successful [2]. 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 [3]. 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 [2,4,5,6,7,8], 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 will be drawn from the obtained results and possible future work lines will be 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* [8] 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 [9], 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 [9], 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 [10] 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 [11,12]. 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* [1,2,4,5,6,7]. 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* [2]. 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 creará una aplicación web sencilla con la que el usuario pueda 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 nuestra enfoque al problema, describiendo las tecnologías usadas y el desarrollo de la solución. Después se presentarán las pruebas y resultados obtenidos, comparando la solución con otras existentes, a modo de validar y verificar la aportación de nuestra solución. Seguidamente, se comentarán 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 darán unas conclusiones sobre los resultados obtenidos y se propondrán 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 [2]. 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 linea 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 [10]. Aunque pueda parecerlo, la Integración Continua no es un proceso trivial, en [11] 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 faclitar 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 el código 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 [8], 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* [9] (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 denominados 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*) [10]. 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 catacteríticas extraídas directamente de la base de datos de TravisTorrent [6], sin embargo, estas características no son las mejores para predecir *builds* que fallan, es decir, *builds failures*. Algunos enfoques [6,7] hacen uso de características basadas en la *build* actual, la *build* anterior y el histórico ligado a todas las ejecuciones de *builds* anteriores. Hassan et al. [7] fue el primer estudio en utilizar técnicas de *Machine Learning* para predecir el resultado de *CI*. 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_buildfilechangecount*: 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_methodbodychangecount*: número de cambios en el cuerpo del método.
- *cmt_methodchangecount*: número de cambios en la cabecera del método.
- *cmt_importchangecount*: número de cambios en los *imports*.
- *cmt_fieldchangecount*: número de cambios en los atributos de clase.
- *day_of_week*: día de la semana del primer *commit* de la *build*.
- *cmt_classchangecount*: número de clases cambiadas.
- *gh_files_added*: número de archivos añadidos.
- *gh_test_churn*: número de líneas de código de test cambiadas.

En [6] 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* [2], 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. *Rausch et al.* [12] muestra una categorización de las *build failures* según el tipo de error que las origina, identificándose un total de 14 categorías. 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 usan 16 métricas de la literatura y descubren lo siguiente:

- Respaldan 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 más frecuentemente 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 [13] 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. [10] 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. [14] 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 un 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 [15] se presenta un estudio en el que se preguntó a trabajadores de la empresa *Atlassian* sobre sus percepciones sobre los fallos de *CI*. Entre sus 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, comentan que los fallos de *CI* pueden afectar tanto al flujo de trabajo individual como a la empresa. Los trabajadores notan que estos fallos pueden 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 pueden 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* [7,5,2,16,6,17,4,1,18]. Hassan et al. [7] 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 [5] 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 condiciones 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.** [2]. 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, tenemos que si la primera es cierta, al saltarse todas aquellas *builds* que se predigan como exitosas, se reducirá el coste considerablemente. Si la segunda es 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 lineas 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:

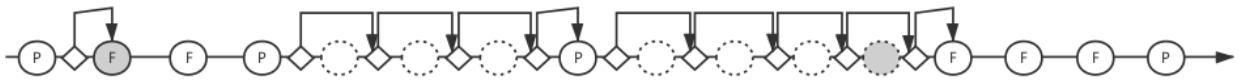

Figura 1. Línea temporal de SmartBuildSkip [2].

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 [2] 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. [16] 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 [6] 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 selecionan determinadas *features* en función del tipo de proyecto que se esté evaluando. En otro artículo, Ouni et al. [17] 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 [7] 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 poca duradera el enfoque, ya que está fuertemente acoplado a la estructura *HTML* de *GitHub* y sus cambios.

Jin et al. [4] 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. [1] 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. [18] 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 esado 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 qué tan bien están desempeñando 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.

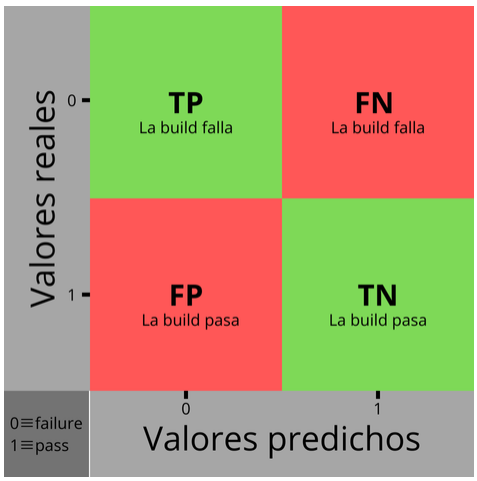

Figura 2. Matriz de confusión.

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 por el total de predicciones realizadas.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

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

$$P = \frac{TP}{TP + FP} \tag{2}$$

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

$$R = \frac{TP}{TP + FN} \tag{3}$$

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

$$F1 = 2 \times \frac{P \times R}{P + R} \tag{4}$$

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 estructutural, ya que las secciones y capítulos siempre irán horientados 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, 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

5. Detalles de la propuesta

6. Resultados

7. Amenazas a la validez

8. Conclusiones y trabajos futuros

Referencias

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. Martin Fowler and Matt Foemmel. Continuous integration, 2006. [Online; accessed 2-Aug-2024].
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.