Pontificia Universidad Católica del Ecuador

Facultad De Habitad, Infraestructura y Creatividad



Tema:

Diseño y evaluación de un modelo basado en aprendizaje automático para estimar calificaciones de tareas de programación mediante la integración de pruebas funcionales y métricas de calidad de código

Autor:

Adrián Estiven Balseca Olivo

Docente:

Sebastián Felipe Tamayo Proaño

Programa académico:

Maestría en Sistemas de la Información, Mención Data Science

Fecha de Entrega: 16 de agosto de 2026

Agradecimiento

Agradezco en primer lugar a Dios, por brindarme la fortaleza, la salud y la perseverancia necesarias para culminar esta etapa académica y profesional.

Agradezco a mis padres y a mi familia, quienes han sido un apoyo fundamental durante este proceso. Su confianza, paciencia y motivación me impulsaron a continuar incluso en los momentos de mayor exigencia.

Expreso también mi gratitud a los docentes de la maestría, por compartir sus conocimientos, experiencias y orientación a lo largo de mi formación. Sus enseñanzas contribuyeron significativamente a mi crecimiento profesional y al desarrollo de este trabajo de titulación.

Finalmente, agradezco a mis amigos y personas cercanas que me acompañaron durante este camino, por su apoyo, palabras de ánimo y comprensión. Este logro representa no solo un esfuerzo personal, sino también el respaldo de quienes estuvieron presentes durante esta etapa.

Índice

Tabla de contenido

Agradecimiento 2

Índice 3

Capítulo I: Introducción 5

1.1 Introducción 5

1.2 Problemática 5

1.3 Objetivos 6

1.3.1 Objetivo general 6

1.3.2 Objetivos específicos 6

1.4 Justificación 7

1.5 Pregunta de Investigación 8

Capítulo II: Desarrollo 9

2.1 Marco Teórico 9

2.2 Metodología 17

2.3 Resultados y Análisis 20

2.3.1 Enfoque conceptual aplicado al producto 20

2.3.2 Caracterización del dataset final 20

2.3.3 Preparación de variables de evaluación 21

2.3.4 Implementación del pipeline y de los modelos 22

2.3.5 Evaluación experimental de los modelos 23

2.3.6 Condiciones de uso del modelo 24

2.3.7 Resultados del modelo final 25

2.3.8 Análisis de los resultados 28

2.4 Discusión frente a la literatura y validez externa 30

Capítulo III: Conclusiones y Recomendaciones 31

3.1 Conclusiones 31

3.2 Limitaciones 32

3.3 Recomendaciones 32

3.4 Aportes 33

Anexos 35

Anexo A. Diccionario de datos del dataset final 35
Anexo B. Protocolo de reproducción 36
Anexo C. Matriz de trazabilidad metodológica 38
Anexo D. Control de privacidad y exclusiones 39
Anexo E. Evidencia experimental final 40
Anexo F. Evidencias del producto implementado 41

Capítulo I: Introducción

1.1 Introducción

La evaluación de tareas de programación constituye un proceso fundamental dentro de la formación académica en carreras relacionadas con la tecnología, la informática y el desarrollo de software. A través de estas actividades, los estudiantes demuestran competencias asociadas al pensamiento lógico, la resolución de problemas, el diseño algorítmico y la aplicación de buenas prácticas de programación.

No obstante, la calificación de este tipo de tareas puede convertirse en una actividad extensa y compleja para el docente, especialmente cuando se deben revisar múltiples entregas, considerar diferentes criterios de evaluación y mantener consistencia en las calificaciones asignadas. Además, la valoración de una tarea de programación no siempre debe limitarse al resultado final obtenido por el programa, sino que también puede incluir aspectos como la estructura del código, la legibilidad, la documentación, la complejidad, la mantenibilidad y el cumplimiento de criterios técnicos previamente definidos.

En este contexto, el uso de modelos de aprendizaje automático representa una alternativa para apoyar el proceso de evaluación, ya que permite analizar datos históricos, identificar patrones y estimar calificaciones a partir de variables disponibles en un conjunto de datos estructurado. Por ello, la presente investigación propone diseñar, implementar y evaluar un modelo basado en aprendizaje automático para estimar calificaciones de tareas de programación, considerando resultados funcionales, métricas de calidad de código y calificaciones de referencia contenidas en el dataset proporcionado para el estudio.

La investigación no busca reemplazar el criterio académico del docente, sino aportar una herramienta de apoyo que permita analizar de manera más sistemática, objetiva y replicable la información disponible sobre las tareas de programación. De esta manera, se espera contribuir al desarrollo de enfoques más eficientes para la evaluación asistida en contextos educativos vinculados con la enseñanza de la programación.

1.2 Problemática

La calificación de tareas de programación presenta una dificultad particular porque combina criterios objetivos, como la ejecución correcta del código, con criterios más interpretativos, como legibilidad, organización, documentación y complejidad. En cursos con varias

actividades y múltiples entregas, esta revisión puede demandar un tiempo considerable del docente y generar diferencias de criterio entre revisiones.

En el caso analizado, las entregas del Paralelo 4 se encontraban distribuidas en paquetes comprimidos, rúbricas, actividades y registros de calificación. Esta situación exigió construir primero una base de datos trazable, anonimizada y auditable, capaz de relacionar cada entrega con su actividad, resultado de aprendizaje, criterio de evaluación y calificación oficial normalizada.

El problema de investigación se centra, por tanto, en determinar si las variables extraídas del código fuente, junto con evidencia funcional parcial validada, permiten estimar las calificaciones de referencia con un error suficientemente bajo para apoyar la revisión docente. El alcance no corresponde a una calificación automática final, sino a una herramienta de apoyo que debe conservar revisión humana.

1.3 Objetivos

1.3.1 Objetivo general

Diseñar, implementar y evaluar un modelo basado en aprendizaje automático para estimar calificaciones de tareas de programación mediante la integración de resultados de pruebas funcionales y métricas de calidad de código.

1.3.2 Objetivos específicos

1.3.2.1 Analizar enfoques existentes de evaluación automática y evaluación asistida de tareas de programación mediante aprendizaje automático y técnicas de análisis de código.

1.3.2.2 Caracterizar el conjunto de datos disponible, identificando su fuente, estructura, tamaño, escala de calificación, variables predictoras, variable objetivo y posibles sesgos.

1.3.2.3 Definir y preparar las variables de evaluación derivadas de resultados funcionales, métricas de calidad de código y atributos estructurales disponibles en las entregas de programación.

1.3.2.4 Implementar modelos supervisados para estimar calificaciones de tareas de programación y compararlos con un modelo base o criterio de referencia.

1.3.2.5 Evaluar el desempeño predictivo de los modelos mediante métricas, análisis de errores e interpretación de variables relevantes.

1.3.2.6 Establecer las condiciones, limitaciones y recomendaciones para el uso del modelo como apoyo al proceso de calificación docente.

1.4 Justificación

Dentro de la formación en carreras afines a la tecnología, la evaluación de tareas de programación constituye un proceso fundamental para valorar competencias asociadas al pensamiento lógico, la resolución de problemas, el diseño algorítmico y la aplicación de buenas prácticas de desarrollo de software (Messer et al., 2024). Por esta razón, la revisión de estas actividades no solo implica verificar si el programa produce una salida correcta, sino también valorar una gran cantidad de criterios que convierten en muchos casos a la calificación manual en una actividad extensa, demandante e ineficiente, especialmente en cursos con alta cantidad de estudiantes o entregas frecuentes. En ese contexto, la evaluación manual de tareas de programación puede representar una carga considerable para los docentes, sobre todo cuando se requiere revisar corrección funcional, organización del código, documentación y criterios de calidad de manera consistente.

Además, debido al tiempo que demanda al docente la evaluación manual de tareas de programación, la demora en la entrega de retroalimentación también afecta el aprendizaje y el rendimiento académico de los estudiantes. Esto se debe a que, cuando la retroalimentación no se proporciona de manera oportuna, los estudiantes tienen menos posibilidades de identificar y corregir sus errores, reforzar conceptos y mejorar sus siguientes entregas, en este sentido y como consecuencia, continuarán repitiendo las mismas fallas en actividades posteriores, lo que limita su progreso académico y reduce la efectividad del proceso de enseñanza-aprendizaje. De hecho, la literatura sobre herramientas de evaluación automática en programación destaca que la retroalimentación rápida permite que los estudiantes reconozcan errores y realicen nuevos intentos de mejora, lo cual favorece su proceso de aprendizaje. Por estas razones, resulta necesario explorar mecanismos que apoyen al docente en la revisión de tareas, sin reemplazar su criterio académico, pero sí facilitando una valoración más ágil y consistente.

Por otra parte, en muchos casos, la calificación de tareas de programación se centra en el resultado final del programa o en el cumplimiento general de los requerimientos planteados, sin embargo, este enfoque puede dejar en segundo plano aspectos relevantes del proceso de desarrollo, como la organización del código, la lógica utilizada, la claridad de los nombres, el manejo de errores, la reutilización, la documentación y el cumplimiento de buenas prácticas. En este sentido, un estudiante puede presentar una solución parcialmente correcta o evidenciar

una aproximación adecuada al problema, pero recibir una calificación baja si la evaluación se enfoca únicamente en la salida final del programa.

De igual manera, investigaciones recientes señalan que los criterios parcialmente subjetivos, como legibilidad, mantenibilidad y documentación, suelen generar dificultades de consistencia en la evaluación de tareas de programación, lo que puede producir variaciones en la calificación entre evaluadores o entre entregas similares, afectando la equidad y la objetividad del proceso evaluativo.

Debido a esto, se plantea el diseño de un modelo basado en aprendizaje automático para estimar calificaciones de tareas de programación mediante la integración de resultados de pruebas funcionales y métricas de calidad de código, ya que, este enfoque permitirá representar cada entrega estudiantil a través de variables relevantes asociadas tanto al comportamiento funcional del programa como a características estructurales del código fuente, permitiendo que el modelo pueda considerar no solo si la solución produce el resultado esperado, sino también aspectos relacionados con la organización, complejidad, documentación, reutilización y calidad general del código.

Finalmente, la investigación busca aportar al conocimiento al evaluar la viabilidad de utilizar un modelo de aprendizaje automático para estimar calificaciones de tareas de programación mediante la integración de resultados de pruebas funcionales y métricas de calidad de código. Asimismo, permitirá analizar la relación entre estas variables y las calificaciones de referencia, con el propósito de identificar qué características contribuyen en mayor medida al proceso de estimación. En conclusión, el estudio busca aportar evidencia sobre el uso de enfoques más integrales, sistemáticos y replicables para la evaluación asistida de programación en contextos educativos.

1.5 Pregunta de Investigación

¿En qué medida la integración de resultados de pruebas funcionales y métricas de calidad de código permite estimar, mediante modelos de aprendizaje automático supervisado, las calificaciones de tareas de programación con un error predictivo aceptable y utilidad como apoyo al proceso de evaluación docente?

Capítulo II: Desarrollo

2.1 Marco Teórico

Evaluación automática y asistida de tareas de programación

La evaluación de tareas de programación es un proceso académico orientado a valorar el nivel de dominio que alcanza un estudiante al resolver problemas mediante código. En este tipo de evaluación no solo se analiza el resultado final de un programa, sino también la forma en que la solución fue construida, la organización del código, la claridad, la documentación y el cumplimiento de buenas prácticas.

La calificación manual de tareas de programación puede convertirse en una actividad compleja cuando existe una gran cantidad de entregas o cuando intervienen varios criterios de evaluación. Además, algunos criterios, como la legibilidad, la documentación o la elegancia del código, pueden presentar cierto grado de subjetividad, lo que puede afectar la consistencia de las calificaciones entre evaluadores.

En este contexto, la evaluación automática o asistida de tareas de programación ha surgido como una línea de investigación dentro de la educación en computación. Su propósito es utilizar herramientas computacionales para apoyar el proceso de revisión, calificación y retroalimentación de entregas de programación. Estos sistemas pueden emplear diferentes enfoques, como comparación de salidas, pruebas de ejecución, análisis estático, revisión de métricas de código, comparación con soluciones de referencia o modelos basados en datos.

Keuning, Jeuring y Heeren (2018) realizaron una revisión sistemática sobre generación automática de retroalimentación en ejercicios de programación. Su trabajo constituye uno de los antecedentes importantes en esta línea, ya que analiza cómo los sistemas automáticos pueden proporcionar información al estudiante sobre errores, calidad de solución y posibles mejoras. Esta revisión evidencia que las herramientas de retroalimentación automática buscan facilitar el aprendizaje y reducir parte de la carga del docente, aunque también muestran limitaciones respecto a la profundidad y adaptabilidad de los mensajes generados.

Posteriormente, Messer, Brown, Kölling y Shi (2024) desarrollaron una revisión sistemática sobre herramientas de calificación y retroalimentación automática en educación en programación. En este estudio se analizaron 121 trabajos publicados entre 2017 y 2021, identificando que muchas herramientas se enfocan principalmente en evaluar la corrección funcional de las soluciones. Sin embargo, los autores señalan que existe menor atención hacia

criterios como mantenibilidad, legibilidad y documentación, dimensiones que resultan importantes para una evaluación más completa del código.

A partir de estos antecedentes, el vacío que aborda esta tesis se ubica en la construcción de un dataset local, anonimizado y trazable que combine rúbricas oficiales, métricas estáticas de código, contexto curricular y evidencia funcional parcial. A diferencia de estudios centrados únicamente en corrección funcional o retroalimentación automática, este trabajo evalúa el desempeño mediante validación anidada agrupada por estudiante, reporta métricas de error como MAE, RMSE y R^2, analiza generalización por actividad, documenta amenazas de validez y delimita el uso del modelo como apoyo docente.

Estos antecedentes muestran que la evaluación automática o asistida de tareas de programación ha avanzado principalmente en la verificación de corrección, pero todavía presenta oportunidades de mejora cuando se busca considerar criterios más amplios relacionados con calidad del código. Por esta razón, resulta pertinente estudiar enfoques basados en aprendizaje automático que permitan estimar calificaciones a partir de diferentes variables asociadas al desempeño de una entrega de programación.

Criterios de calificación de tareas de programación

Los criterios de calificación de tareas de programación son los elementos que permiten valorar una solución desarrollada por un estudiante. Estos criterios pueden variar según el nivel de la asignatura, el tipo de ejercicio, los resultados de aprendizaje y la rúbrica definida por el docente. No obstante, en la literatura se identifican dimensiones comunes, como funcionalidad, corrección, legibilidad, documentación, organización, complejidad, mantenibilidad y uso de buenas prácticas.

La funcionalidad o corrección se relaciona con la capacidad del programa para cumplir los requerimientos planteados. Este criterio suele ser uno de los más utilizados en la evaluación de ejercicios de programación, debido a que permite verificar si la solución produce resultados adecuados. Sin embargo, evaluar únicamente la corrección puede generar una visión incompleta del desempeño del estudiante, ya que no necesariamente refleja la calidad interna del código.

La legibilidad se refiere a qué tan fácil resulta comprender el código. Este criterio está relacionado con el uso de nombres claros, una estructura ordenada, formato consistente, comentarios útiles y organización lógica. La documentación, por su parte, permite explicar partes importantes del programa y facilitar su comprensión. La mantenibilidad se relaciona con

la facilidad para modificar, corregir o ampliar una solución. La complejidad permite analizar el nivel de dificultad estructural o lógica del programa.

Messer, Brown, Kölling y Shi (2025) estudiaron la consistencia de evaluadores humanos al calificar tareas de programación. En su investigación, los participantes evaluaron trabajos de programación considerando criterios como correctness, code elegance, readability y documentation. Los resultados mostraron dificultades de consistencia entre evaluadores, especialmente en criterios más subjetivos. Este antecedente es relevante porque evidencia que la calificación de programación no depende únicamente de saber si una solución funciona, sino también de criterios que pueden interpretarse de distintas maneras.

Desde esta perspectiva, una evaluación más completa de tareas de programación requiere criterios definidos con claridad y, en la medida de lo posible, representados mediante variables medibles. Esto permite que los modelos computacionales puedan analizar información relacionada con las entregas y generar estimaciones más consistentes. Por ello, los criterios de calificación constituyen una base fundamental para diseñar modelos de aprendizaje automático orientados a estimar notas o niveles de desempeño.

Calidad de código y buenas prácticas de programación

La calidad de código es un concepto central en la ingeniería de software, ya que permite valorar si una solución ha sido desarrollada de forma clara, ordenada, comprensible y mantenible. Un programa puede producir una salida correcta, pero al mismo tiempo presentar baja calidad técnica si tiene una estructura confusa, alta complejidad, duplicación innecesaria, nombres poco claros o escasa documentación.

La norma ISO/IEC 25010:2023 se utiliza en esta investigación únicamente como marco conceptual para organizar dimensiones de calidad del producto software, no como instrumento de medición directa. En el dataset, algunas variables funcionan como aproximaciones o proxies: la adecuación funcional se observa parcialmente mediante resultados funcionales piloto y parse_success_rate; la mantenibilidad se aproxima mediante complejidad, profundidad de anidamiento, cantidad de funciones y reutilización; la legibilidad se relaciona con comentarios, nombres descriptivos y estructura; y la fiabilidad se observa de forma limitada mediante errores de sintaxis o excepciones amplias. Por tanto, los conteos de líneas, comentarios o complejidad no representan una evaluación integral de calidad ISO, sino indicadores observables usados para modelado predictivo.

En el ámbito de la programación, las buenas prácticas se relacionan con escribir código legible, simple, modular, documentado y fácil de mantener. En Python, por ejemplo, PEP 8 establece una guía de estilo cuyo propósito es mejorar la legibilidad y consistencia del código. Aunque una tarea de programación pueda desarrollarse en distintos lenguajes, este tipo de guías permite sustentar la importancia del estilo, la claridad y la organización como elementos relevantes en la calidad del código.

También se han propuesto métricas específicas para valorar aspectos internos del software. McCabe (1976) planteó la complejidad ciclomática como una medida para analizar la complejidad lógica de un programa a partir de sus caminos de ejecución. Esta métrica ha sido ampliamente utilizada como referencia para estimar la dificultad de comprensión, prueba y mantenimiento de un código. En tareas académicas, una complejidad excesiva puede indicar soluciones difíciles de entender o mantener, mientras que una complejidad adecuada puede reflejar una estructura más clara.

De esta manera, la calidad de código aporta una base teórica importante para la calificación de tareas de programación, ya que permite considerar aspectos que van más allá de la salida del programa. Incorporar criterios de calidad contribuye a una evaluación más integral y permite representar las entregas mediante características analizables por modelos de aprendizaje automático.

Programación de calidad en Python

La programación de calidad en Python se relaciona con la escritura de código claro, legible, simple y mantenible. Python es un lenguaje ampliamente utilizado en educación, ciencia de datos y desarrollo de software, por lo que sus principios de estilo y buenas prácticas resultan útiles para fundamentar criterios de calidad aplicables a tareas de programación.

PEP 8, la guía de estilo oficial de Python señala que las recomendaciones de estilo buscan mejorar la legibilidad del código y mantener consistencia dentro de la comunidad de desarrolladores. Entre sus recomendaciones se encuentran el uso adecuado de nombres, indentación, longitud de línea, organización de importaciones, espacios en blanco y estructura general del código. Estos elementos no se limitan a una cuestión estética, sino que facilitan la comprensión y mantenimiento de los programas.

En el contexto educativo, la programación de calidad implica que el estudiante no solo consiga resolver un problema, sino que lo haga mediante una solución comprensible, estructurada y

alineada con buenas prácticas. Esto es importante porque el aprendizaje de programación no se reduce a obtener una salida esperada; también implica desarrollar hábitos de escritura de código que puedan mantenerse en proyectos más grandes y complejos.

Autores como Ramalho, en Fluent Python, y Percival y Gregory, en Architecture Patterns with Python, resaltan la importancia de escribir código expresivo, modular y mantenible. Estos enfoques permiten sustentar que la calidad del código puede analizarse desde criterios como claridad, simplicidad, organización, reutilización y diseño. Por ello, los principios de programación de calidad pueden utilizarse como base conceptual para definir variables o dimensiones relacionadas con la evaluación de tareas de programación.

Minería de texto aplicada al análisis de programas

La minería de texto, o text mining, es el proceso mediante el cual se extrae información útil a partir de datos textuales. Su objetivo es transformar texto no estructurado o semiestructurado en representaciones que puedan ser analizadas mediante técnicas estadísticas, modelos predictivos o algoritmos de aprendizaje automático.

En el análisis de tareas de programación, la minería de texto puede ser relevante porque el código fuente posee una estructura textual. Un programa contiene palabras reservadas, nombres de variables, funciones, clases, comentarios, operadores y estructuras de control. Estos elementos pueden analizarse para obtener características relacionadas con la forma en que un estudiante construye una solución.

Por ejemplo, a partir de un programa se pueden identificar patrones de uso de estructuras condicionales, ciclos, comentarios, nombres de funciones, longitud de líneas, importaciones, retornos, parámetros o profundidad de anidamiento. En esta investigación se utilizaron métricas estructurales y estáticas del código, no representaciones textuales avanzadas ni modelos de lenguaje. Por ello, la minería de texto se presenta como antecedente conceptual para transformar código en variables, mientras que el experimento final se apoya en variables tabulares auditadas.

El análisis textual y estructural del código permite estudiar aspectos de legibilidad, documentación y estilo, especialmente cuando existen comentarios, identificadores descriptivos y organización observable dentro del programa. En combinación con métricas de calidad, estas técnicas contribuyen a representar una tarea de programación como un conjunto de variables analizables por modelos supervisados.

Por tanto, el análisis de código como dato constituye un fundamento importante para investigaciones que buscan estimar calificaciones de programación. En el presente estudio, dicho fundamento se materializa en métricas estáticas, atributos curriculares y evidencia funcional piloto, evitando afirmar el uso de técnicas que no fueron implementadas en el experimento final.

Machine Learning para estimar calificaciones

El Machine Learning, o aprendizaje automático, es una disciplina de la inteligencia artificial que permite a los sistemas aprender patrones a partir de datos y realizar predicciones o estimaciones. En lugar de depender únicamente de reglas programadas manualmente, los modelos de aprendizaje automático utilizan ejemplos históricos para aprender la relación entre variables de entrada y una variable de salida.

En el contexto educativo, el aprendizaje automático se ha utilizado para analizar rendimiento académico, predecir resultados, detectar riesgos y apoyar procesos de evaluación. En tareas de programación, su uso resulta pertinente porque las entregas pueden representarse mediante variables asociadas a criterios de calificación, resultados funcionales, métricas de calidad, características del código o indicadores de desempeño.

Cuando el objetivo es estimar una calificación numérica, el problema se puede formular como una tarea de regresión supervisada. En este enfoque, el modelo aprende a partir de registros previamente calificados y luego estima la calificación de nuevas entregas. La variable objetivo corresponde a la nota o puntuación, mientras que las variables predictoras corresponden a los atributos que representan cada tarea.

Si la calificación se transforma en categorías, como bajo, medio o alto desempeño, el problema puede abordarse como clasificación. Sin embargo, para una investigación orientada a estimar una nota, la regresión resulta más adecuada porque permite obtener una salida numérica comparable con la calificación real.

El uso de aprendizaje automático no debe entenderse como un reemplazo del docente, sino como una herramienta de apoyo. Su valor radica en analizar patrones, comparar modelos, identificar variables relevantes y generar estimaciones que puedan complementar el proceso de evaluación.

Modelos de Machine Learning para estimar calificaciones

Para estimar calificaciones de tareas de programación pueden utilizarse distintos modelos de aprendizaje automático. En esta investigación se compararon modelos tabulares adecuados para el tamaño y estructura del dataset: un baseline de mediana, Ridge Regression, Elastic Net, Random Forest e Hist Gradient Boosting. La comparación permitió evaluar si modelos más complejos superaban a un modelo lineal regularizado bajo validación agrupada por estudiante.

La Regresión Lineal es uno de los modelos base más utilizados en problemas de predicción numérica. Su propósito es modelar una relación lineal entre las variables predictoras y la variable objetivo. Aunque puede ser limitada cuando las relaciones son complejas, resulta útil como punto de comparación inicial. Una variante importante es Ridge Regression, propuesta por Hoerl y Kennard (1970), que incorpora regularización para reducir problemas asociados a multicolinealidad y sobreajuste.

Random Forest, propuesto por Breiman (2001), es un modelo de ensamble basado en múltiples árboles de decisión. Este enfoque combina varios árboles para obtener una predicción más robusta. En problemas de regresión, Random Forest puede capturar relaciones no lineales entre variables y suele ser útil cuando existen múltiples atributos con interacciones complejas.

Gradient Boosting, desarrollado por Friedman (2001), es otro enfoque de ensamble basado en árboles, pero construido de manera secuencial. Cada nuevo modelo intenta corregir los errores cometidos por los anteriores. Este método ha demostrado buen desempeño en problemas tabulares y puede ser una alternativa relevante para estimar calificaciones cuando existen múltiples variables predictoras.

La comparación entre Ridge, Elastic Net, Random Forest e Hist Gradient Boosting permitió evaluar distintos niveles de complejidad sin introducir modelos que requirieran mayor volumen de datos o una etapa adicional de ajuste. Finalmente, Ridge fue seleccionado como resultado principal por su desempeño, estabilidad e interpretación más sencilla frente a alternativas de mayor complejidad.

Evaluación de modelos predictivos

La evaluación de modelos predictivos permite determinar qué tan cercanas son las estimaciones generadas por un modelo respecto a los valores reales. En problemas de regresión, donde se busca estimar una calificación numérica, las métricas más utilizadas incluyen MAE, RMSE y R^2.

El MAE, o error absoluto medio, calcula el promedio de las diferencias absolutas entre las calificaciones reales y las calificaciones estimadas. Es una métrica fácil de interpretar porque expresa el error promedio en las mismas unidades de la calificación.

El RMSE, o raíz del error cuadrático medio, también mide la diferencia entre valores reales y predichos, pero penaliza con mayor fuerza los errores grandes. Esto resulta útil cuando se quiere evitar que el modelo genere estimaciones demasiado alejadas de la calificación real.

El coeficiente R^2 indica qué proporción de la variabilidad de la variable objetivo puede ser explicada por el modelo. Un valor más alto sugiere que el modelo representa mejor la relación entre las variables predictoras y la calificación.

Estas métricas permiten comparar modelos de manera objetiva y seleccionar aquel que presente mejor desempeño. Además, facilitan la interpretación de los resultados y la presentación de evidencia sobre la viabilidad del enfoque propuesto.

CRISP-DM como marco metodológico para proyectos de ciencia de datos

CRISP-DM, siglas de Cross-Industry Standard Process for Data Mining, es una metodología utilizada para organizar proyectos de minería de datos y ciencia de datos. Su estructura se compone de seis fases: comprensión del negocio o problema, comprensión de los datos, preparación de los datos, modelado, evaluación y despliegue o presentación de resultados.

La primera fase, comprensión del problema, consiste en definir qué necesidad se busca resolver, por qué es importante y qué resultado sería útil. En una investigación sobre estimación de calificaciones, esta fase permite establecer que el problema se relaciona con apoyar la evaluación de tareas de programación mediante modelos basados en datos.

La segunda fase, comprensión de los datos, se centra en conocer la información disponible, sus variables, calidad, estructura y posibles limitaciones. Esta fase es importante porque permite determinar si los datos son suficientes para responder a la pregunta de investigación.

La tercera fase, preparación de los datos, consiste en limpiar, transformar, seleccionar y organizar la información para que pueda ser utilizada por los modelos. Esta fase resulta fundamental porque la calidad de los datos influye directamente en la calidad de los resultados.

La cuarta fase, modelado, implica seleccionar, entrenar y ajustar algoritmos de aprendizaje automático. En este punto se comparan diferentes modelos con el propósito de identificar cuál ofrece mejores resultados.

La quinta fase, evaluación, permite analizar si los modelos cumplen con el objetivo del proyecto. No basta con obtener un resultado técnico; es necesario verificar si el modelo responde al problema planteado y si sus resultados son interpretables y útiles.

Finalmente, la sexta fase corresponde al despliegue o presentación de resultados. En una tesis, esta fase puede traducirse en la exposición clara de tablas, gráficos, métricas, análisis comparativos y conclusiones.

CRISP-DM es pertinente para esta investigación porque permite organizar de forma ordenada el proceso de análisis, desde la definición del problema hasta la evaluación de los modelos. Además, ayuda a justificar que la preparación de datos y la selección de modelos deben responder a una pregunta analítica clara y no realizarse de manera aislada.

Síntesis del marco teórico

La revisión teórica permite establecer que la calificación de tareas de programación es un proceso complejo que involucra tanto criterios funcionales como criterios relacionados con la calidad del código. Los estudios previos evidencian que la evaluación automática y asistida ha avanzado en la revisión de corrección funcional, pero todavía existen oportunidades para considerar dimensiones como legibilidad, documentación, mantenibilidad y calidad interna del código.

Los criterios de calificación, las métricas de calidad, el análisis estructural del código y los modelos de aprendizaje automático constituyen fundamentos importantes para analizar tareas de programación desde una perspectiva basada en datos. En coherencia con el experimento ejecutado, la comparación se centra en modelos tabulares como Ridge, Elastic Net, Random Forest e Hist Gradient Boosting, junto con un baseline de mediana.

Finalmente, CRISP-DM proporciona una guía metodológica adecuada para estructurar el desarrollo de la investigación, al organizar el proceso en fases que van desde la comprensión del problema hasta la evaluación y presentación de resultados. Con base en este marco teórico, la investigación se sustenta en antecedentes y conceptos que justifican el uso de aprendizaje automático para apoyar la estimación de calificaciones de tareas de programación.

2.2 Metodología

La presente investigación se desarrolló bajo la metodología CRISP-DM, debido a que esta permite organizar proyectos de ciencia de datos de manera progresiva, desde la comprensión del problema hasta la evaluación de resultados. En este estudio, CRISP-DM permitió estructurar

el trabajo en fases relacionadas con la comprensión del contexto educativo, la identificación de las fuentes de información, la preparación del dataset, la construcción de variables, el entrenamiento de modelos de aprendizaje automático y la interpretación de los resultados obtenidos.

En la fase de comprensión del problema se identificó que la calificación de tareas de programación es un proceso que demanda tiempo, criterio técnico y consistencia por parte del docente. A diferencia de otros tipos de evaluación, una tarea de programación puede analizarse desde varios componentes: cumplimiento funcional, estructura del código, legibilidad, documentación, complejidad, uso de instrucciones esperadas y relación con los criterios definidos en la rúbrica. Por esta razón, el problema se formuló como una tarea de regresión supervisada, cuyo propósito fue estimar una calificación numérica a partir de variables extraídas de las entregas de programación.

La población observada estuvo conformada por las entregas de programación del Paralelo 4 disponibles en Moodle. La unidad de análisis fue cada entrega individual calificada. El dataset final incluyó 349 entregas, correspondientes a 28 estudiantes anonimizados, 17 actividades, 3 resultados de aprendizaje y 8 criterios de evaluación. Se excluyeron trabajos grupales, rutas nominativas, nombres de estudiantes, mapas de identidad y claves sensibles como variables predictoras.

El dominio válido de interpretación corresponde al mismo curso, actividades similares, escala de rúbrica equivalente y uso con revisión humana. Los resultados no deben generalizarse automáticamente a otros cursos, cohortes o contextos sin una nueva validación.

Para interpretar la utilidad del modelo se utilizó una escala operativa post hoc del experimento estático. Un MAE menor o igual a 15 puntos se consideró evidencia de utilidad alta como apoyo docente; un MAE mayor a 15 y menor o igual a 20 puntos se consideró evidencia de utilidad moderada; y un MAE superior a 20 puntos se interpretó como utilidad limitada. Esta escala se complementó con la mejora frente al modelo base y con el porcentaje de predicciones dentro de un margen de ±10 puntos. Debido a que el criterio no fue preregistrado antes de todo el desarrollo, se reporta como criterio de interpretación y no como una condición temporal previa al experimento completo.

En la fase de comprensión y preparación de los datos se realizó un inventario de las fuentes originales, incluyendo archivos comprimidos de entregas, rúbricas, enunciados y documentos asociados al syllabus. Posteriormente, se extrajeron las entregas de los estudiantes sin ejecutar

su código, registrando metadatos técnicos como extensión, tamaño del archivo, profundidad de archivo comprimido, estado de lectura y hash de contenido. Este procedimiento permitió mantener trazabilidad sobre las fuentes usadas y evitar alteraciones manuales no controladas.

Para proteger la identidad de los estudiantes, se aplicó un proceso de anonimización. Los nombres, rutas nominativas, identificadores directos y archivos de correspondencia de identidad fueron excluidos del dataset de modelado. En su lugar, se utilizaron identificadores anonimizados como actor_id y submission_id, los cuales permiten agrupar entregas y controlar la validación sin exponer información personal. Esta decisión responde a criterios éticos y metodológicos, ya que los datos corresponden a actividades académicas reales.

La construcción del dataset incluyó la vinculación entre entregas, actividades, resultados de aprendizaje, criterios de evaluación y calificaciones registradas en rúbricas. La variable objetivo-utilizada fue target_technical_grade_pct, que representa la calificación de referencia normalizada en una escala de 0 a 100 puntos. En esta versión del estudio, dicha variable se interpreta como calificación de referencia derivada de la rúbrica oficial, no como una nota técnica independiente separada manualmente por componentes.

Las variables predictoras fueron construidas principalmente mediante análisis estático del código fuente. Entre las variables consideradas se incluyeron número de archivos analizados, líneas físicas, líneas de código, líneas en blanco, líneas de comentario, proporción de comentarios, número de funciones, importaciones, retornos, parámetros, asignaciones globales, profundidad de anidamiento, complejidad estructural y presencia de problemas de sintaxis. Estas variables no se interpretan como una medición absoluta de calidad del código, sino como variables proxy; es decir, indicadores observables que permiten aproximar aspectos como estructura, legibilidad, organización y complejidad.

En cuanto a las pruebas funcionales, se construyó un catálogo por actividad y se integró un piloto funcional parcial con oráculos validados. Este bloque permitió generar columnas functional_pilot_* para las entregas cubiertas; sin embargo, su cobertura todavía es limitada y no representa una evaluación funcional completa de todas las actividades. Por esta razón, sus resultados se interpretan como evidencia exploratoria y no como base suficiente para calificación automática.

Para el modelado se compararon modelos supervisados de regresión y un modelo base basado en la mediana. El protocolo final utilizó selección anidada con validación agrupada por estudiante: el ciclo interno seleccionó el modelo y sus hiperparámetros, mientras que el ciclo

externo estimó el desempeño sobre estudiantes no vistos. La imputación, la codificación one-hot y el escalado de variables numéricas se calcularon dentro de cada fold de entrenamiento y se aplicaron al fold de prueba correspondiente, con el fin de evitar fuga de información.

La evaluación consideró métricas como MAE, RMSE y R^2. El MAE permitió medir el error absoluto promedio en puntos sobre 100; el RMSE permitió penalizar errores de mayor magnitud; y el R^2 permitió estimar la proporción de variabilidad explicada por el modelo. Además, se analizó el porcentaje de predicciones ubicadas dentro de un margen de ±10 puntos, debido a que este criterio facilita interpretar la utilidad práctica del modelo como apoyo docente.

2.3 Resultados y Análisis

En esta sección se presentan los resultados obtenidos durante la construcción e implementación del producto. Primero se muestran los componentes generados: dataset final, variables de evaluación, pipeline reproducible, modelos implementados, auditorías y paquete público anonimizado. Luego se reportan las métricas finales y el análisis del desempeño predictivo. Esta organización permite evidenciar el cumplimiento de los objetivos específicos como parte de los resultados del trabajo.

2.3.1 Enfoque conceptual aplicado al producto

El producto desarrollado se fundamentó en los enfoques revisados en el marco teórico: evaluación automática, evaluación asistida, análisis estático de código, pruebas funcionales y modelos supervisados de regresión. Como resultado de esa revisión, el sistema se delimitó como una herramienta de apoyo docente con revisión humana, y no como un calificador automático.

2.3.2 Caracterización del dataset final

El conjunto de datos disponible fue caracterizado desde su fuente, estructura, tamaño, escala de calificación, variable objetivo, variables predictoras y restricciones de privacidad. La unidad de análisis definida fue una entrega individual calificada. El dataset final de modelado contiene 349 registros y 45 columnas, con 28 estudiantes anonimizados, 17 actividades, 3 resultados de aprendizaje y 8 criterios de evaluación.

Elemento caracterizado	Resultado en el producto	Evidencia generada
Fuente de datos	Entregas de programación, rúbricas y registros académicos del Paralelo 4.	Inventarios y auditorías en results/audits.
Unidad de análisis	Una fila por entrega individual calificada.	data/processed/dataset_model_final.csv.
Tamaño final	349 entregas y 45 variables persistidas.	dataset_model_final.csv y dataset_public_final.csv.
Escala de calificación	Variable objetivo normalizada de 0 a 100.	target_technical_grade_pct.
Privacidad	Identificadores anonimizados y exclusión de rutas nominativas.	privacy_audit_final.csv y paquete público.
Producto público	Release anonimizado con 39 archivos.	release/public/tesis_dataset_public_release.zip.

2.3.3 Preparación de variables de evaluación

Las variables de evaluación fueron preparadas en tres bloques principales: contexto académico de la entrega, métricas estáticas de código y variables funcionales piloto. Las métricas estáticas describen estructura, tamaño, legibilidad y complejidad sin ejecutar el código estudiantil. Las variables functional_pilot_* registran evidencia funcional materializada y auditada, pero no implican ejecución masiva de código sin entorno seguro.

Bloque de variables	Variables o ejemplos	Uso en el modelo
Contexto académico	activity_id, rda, criterion_id, attempt_type.	Permite ubicar la entrega dentro de actividad,

Bloque de variables	Variables o ejemplos	Uso en el modelo
		resultado de aprendizaje y criterio.
Metadatos de entrega	files_submitted, total_size_bytes, readable_files.	Describe cantidad, tamaño y legibilidad de los archivos enviados.
Métricas de código	physical_lines, code_lines, comment_lines, function_count, parse_success_rate.	Aporta señales observables de estructura y calidad del código.
Complejidad y organización	max_nesting_depth, descriptive_identifier_ratio, broad_exception_count.	Aproxima legibilidad, anidamiento y prácticas de implementación.
Pruebas funcionales piloto	functional_pilot_tests_assigned, executed, passed, pass_rate, coverage_status.	Registra cobertura funcional auditada para el análisis experimental.
Variable objetivo	target_technical_grade_pct.	Calificación técnica normalizada que el modelo intenta estimar.

2.3.4 Implementación del pipeline y de los modelos

La implementación se materializó en scripts reproducibles dentro de la carpeta src. El flujo principal se centralizó en run_final_pipeline.py, que ejecuta la integración de variables, auditoría de etiquetas, selección anidada, validación canónica, diagnósticos, remuestreo por estudiante, manifiesto de reproducibilidad y construcción del paquete público.

Componente implementado	Archivo o evidencia	Resultado
Pipeline final	src/run_final_pipeline.py.	Ejecuta el producto completo en orden reproducible.
Integración funcional	src/integrate_functional_pilot_features.py.	Agrega ocho variables functional_pilot_* al dataset final.

Componente implementado	Archivo o evidencia	Resultado
Auditoría de etiquetas	src/finalize_label_audit.py.	Verifica etiquetas oficiales anonimizadas y variable objetivo.
Modelado	src/run_nested_selection_final.py.	Compara modelos con validación anidada agrupada por estudiante.
Validación canónica	src/validate_canonical_results.py.	Detiene el cierre si cambian ganador o métricas esperadas.
Paquete público	src/build_public_release_final.py.	Genera release anonimizado con 39 archivos.

2.3.5 Evaluación experimental de los modelos

La evaluación comparó un modelo base de mediana con modelos supervisados. El protocolo final reportó predicciones externas sobre estudiantes no vistos mediante validación anidada agrupada por actor_id. El modelo canónico fue Ridge con solver="cholesky", MAE 14,78, RMSE 20,31, R^2 0,5233 y 44,41 % de predicciones dentro de ±10 puntos.

Evidencia de evaluación	Archivo	Contenido
Comparación final	results/metrics/results_master_final.csv.	Baseline y modelo ganador final.
Umbral de utilidad	results/metrics/utility_threshold_analysis.csv.	Mejora de 38,08 % frente al baseline y uso como apoyo docente.
Predicciones	results/predictions/oof_predictions_final.csv.	Predicciones fuera de muestra

Evidencia de evaluación	Archivo	Contenido
		para análisis de errores.
Importancia de variables	results/metrics/permutation_importance_final.csv.	Variables relevantes por permutación externa.
Errores por grupo	results/metrics/final_metrics_by_group.csv.	Desempeño por actividad, RDA, criterio y grupos de análisis.
Intervalos	results/metrics/actor_bootstrap_intervals_final.csv.	Intervalos por remuestreo agrupado por estudiante.

2.3.6 Condiciones de uso del modelo

El producto establece condiciones explícitas de uso: el modelo puede apoyar la revisión docente, pero no debe utilizarse como calificador automático. La decisión se sustenta en el umbral de utilidad alcanzado, las limitaciones de generalización, la cobertura parcial de pruebas funcionales y la necesidad de revisión humana sobre datos académicos sensibles.

Condición	Decisión documentada	Evidencia
Uso permitido	Apoyo docente con revisión humana.	winner_final.json y utility_threshold_analysis.csv.
Uso no permitido	Calificación automática final.	automatic_grading_threshold_passed = False.
Reproducibilidad	Resultado canónico bloqueado por validación automática.	canonical_results_final.json y validate_canonical_results.py.
Privacidad	Separación entre datos privados y release público.	privacy_audit_final.csv y public_release_inventory_final.csv.

Condición	Decisión documentada	Evidencia
Limitación funcional	No ejecutar masivamente código estudiantil sin sandbox.	functional_student_pilot_audit.csv.

2.3.7 Resultados del modelo final

2.3.7.1 Métricas finales del modelo

El proceso de preparación de datos permitió construir un dataset final de 349 entregas individuales calificadas. Estas entregas corresponden a 28 estudiantes anonimizados, 17 actividades académicas, 3 resultados de aprendizaje y 8 criterios de evaluación. La variable objetivo fue normalizada en una escala de 0 a 100 puntos, lo que permitió comparar actividades con diferentes puntajes máximos bajo una misma referencia.

Como modelo base se utilizó una predicción basada en la mediana. Este baseline obtuvo un MAE de 23,87 puntos, un RMSE de 32,00 y un R^2 de -0,18. Este resultado representa el error esperado cuando se utiliza una estrategia simple que no aprende patrones a partir de las características de las entregas.

Después de aplicar el protocolo de selección anidada por estudiante, el modelo ganador final fue Ridge con selección interna por fold. Para evitar variaciones entre entornos numéricos, Ridge se ejecutó con solver="cholesky" en lugar de solver="auto". El ganador mayoritario fue Ridge con alpha igual a 1, aunque Ridge con alpha igual a 3 fue seleccionado en parte de los pliegues externos. Este protocolo obtuvo un MAE de 14,78 puntos, un RMSE de 20,31 y un R^2 de 0,52. En comparación con el baseline, el modelo redujo el error absoluto medio en aproximadamente 38,08 %. Además, el 44,41 % de las predicciones se ubicó dentro de un margen de ±10 puntos respecto a la calificación real.

De acuerdo con la escala operativa definida, el modelo supera el umbral de apoyo docente, ya que obtuvo un MAE de 14,78 puntos, una mejora de 38,08 % frente al baseline de mediana y 44,41 % de predicciones dentro del margen de ±10 puntos. La estimación por remuestreo agrupado por estudiante ubicó el MAE medio en 14,79, con intervalo de confianza del 95 % entre 13,77 y 16,00. Por tanto, el resultado es útil para priorizar revisión docente, pero no alcanza el umbral definido para calificación automática.

Como análisis complementario, se ejecutó un experimento de mejora con variables derivadas. En ese escenario, la variante enhanced_features con Ridge alpha 3 obtuvo MAE 14,67, RMSE

20,20 y R^2 0,53, con una reducción de 0,076 puntos frente a la configuración base comparable. Este resultado sugiere que las variables derivadas pueden mejorar levemente el desempeño; sin embargo, se reporta como evidencia complementaria y no reemplaza el protocolo principal de selección anidada, cuyo resultado oficial es MAE 14,78.

Sin embargo, el modelo no cumple el criterio para ser usado como sistema de calificación automática, ya que no alcanza el 70 % de predicciones dentro del margen de ±10 puntos. Por esta razón, los resultados deben interpretarse como evidencia de utilidad para apoyar la revisión docente, no como sustituto de la evaluación humana.

También se generaron figuras finales a partir de las predicciones externas del protocolo nested_final. La figura de nota real versus nota predicha permite observar la cercanía entre predicción y calificación de referencia. La distribución de residuales muestra los errores de estimación del modelo ganador. El análisis de error por actividad evidencia en qué actividades el modelo presenta mayor dificultad predictiva.

Tabla 1. Comparación entre baseline y modelo ganador final

Indicador	Baseline mediana	Ridge nested_final
MAE	23,87	14,78
RMSE	32,00	20,31
R^2	-0,18	0,52
Predicciones dentro de ±10	No aplica	44,41 %
Mejora frente al baseline	Referencia	38,08 %

Figura 1. Nota real versus nota predicha - Ridge nested_final

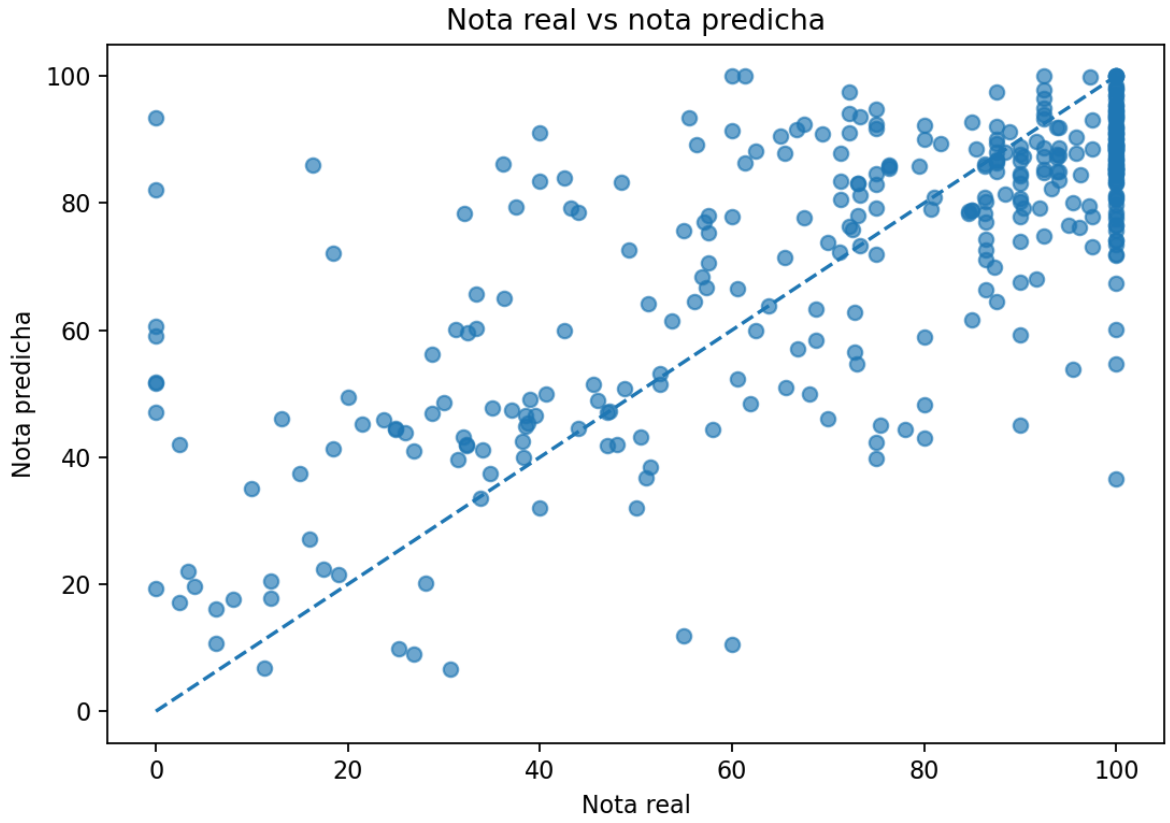


Figura 2. Distribución de residuales - Ridge nested_final

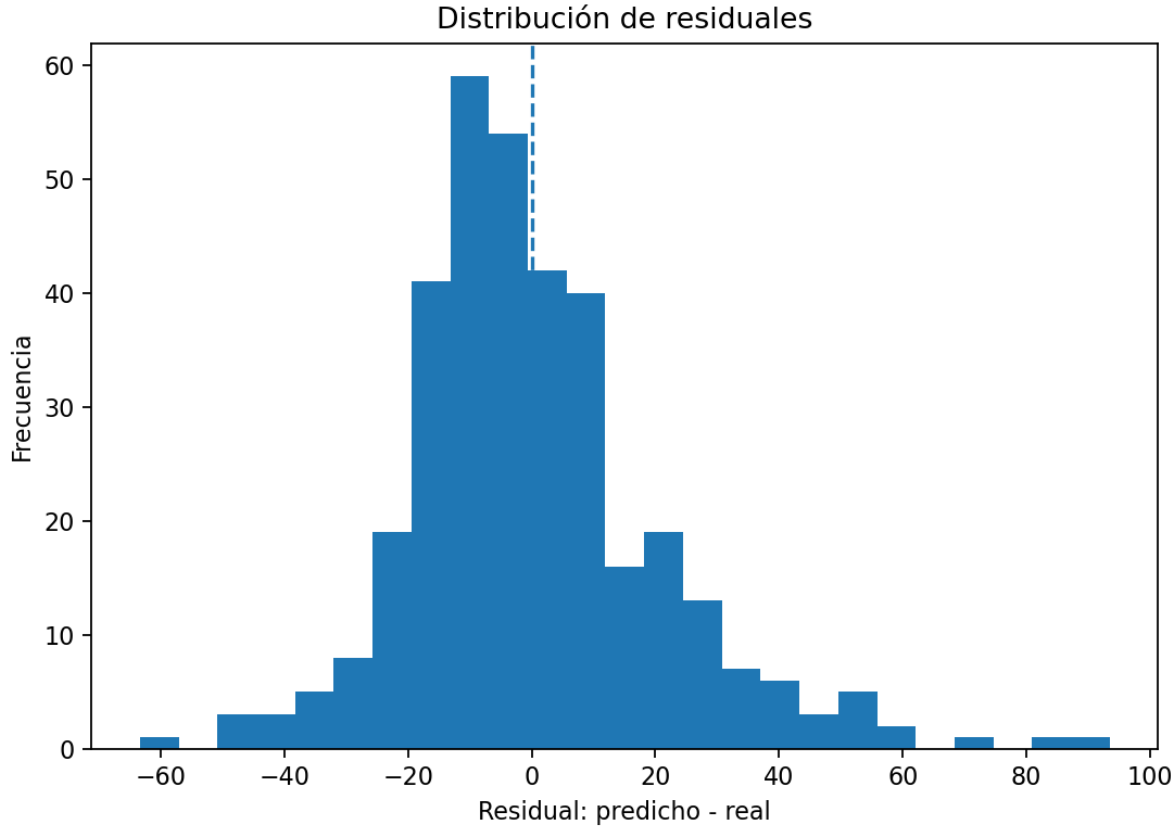
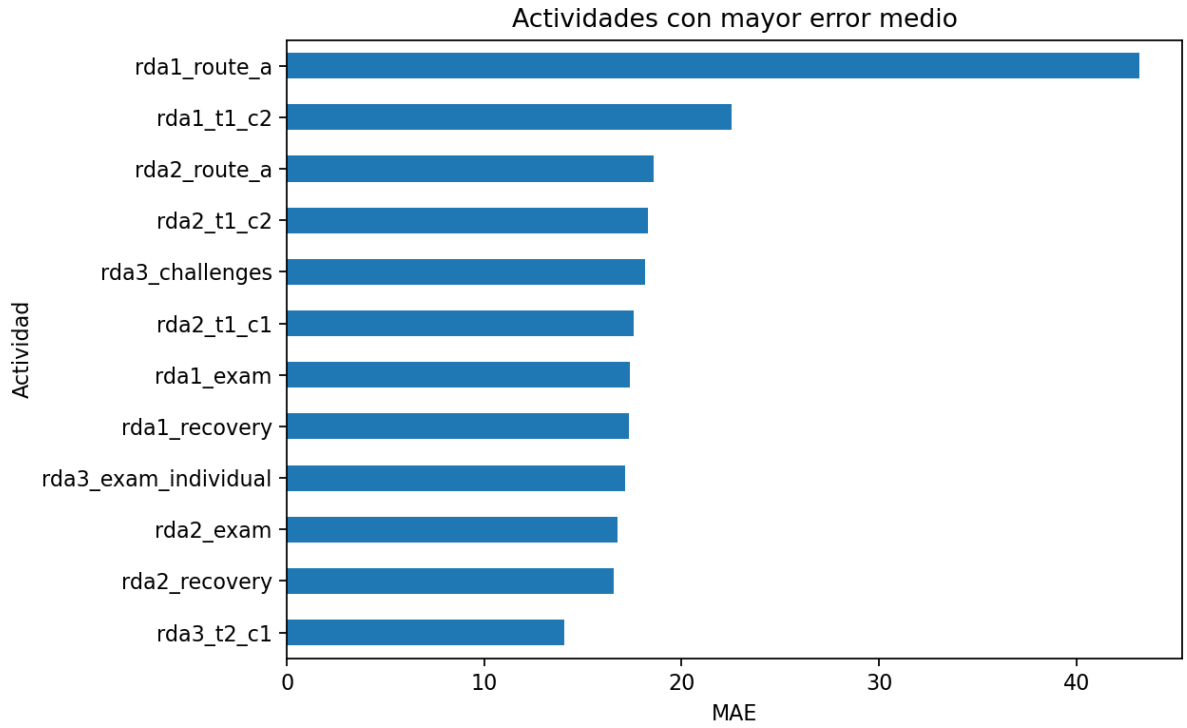


Figura 3. Error absoluto medio por actividad - Ridge nested_final



2.3.8 Análisis de los resultados

Tabla 2. Variables con mayor importancia por permutación externa

Variable	Importancia media	Interpretación
log_physical_lines	2,49	Permutación en folds externos; asociación no causal
import_count	2,31	Permutación en folds externos; asociación no causal
log_total_size_bytes	1,95	Permutación en folds externos; asociación no causal
comment_lines	1,92	Permutación en folds externos; asociación no causal

function_count	1,73	Permutación en folds externos; asociación no causal

Los resultados obtenidos evidencian que las variables extraídas del código fuente y las variables derivadas del flujo final contienen información útil para aproximar la calificación de las entregas de programación. La mejora del 38,08 % frente al modelo base indica que el modelo Ridge seleccionado mediante validación anidada logró capturar patrones que no son representados por una regla simple basada en la mediana. Esto sugiere que características como estructura del código, cantidad de líneas, comentarios, complejidad, funciones, importaciones, transformaciones de tamaño y criterios curriculares pueden relacionarse parcialmente con la calificación obtenida por los estudiantes.

El MAE de 14,78 puntos muestra que el modelo cumple el umbral operativo definido para apoyo docente. Sin embargo, el error sigue siendo considerable para reemplazar la revisión humana, porque una diferencia cercana a quince puntos puede cambiar la interpretación académica de una entrega. En consecuencia, el resultado debe defenderse como apoyo para priorizar revisión, detectar casos atípicos y complementar el análisis docente.

Durante el desarrollo se identificó que el dataset final debía conservar una sola fila por entrega individual calificada. Por ello, se incorporó una auditoría semántica de etiquetas en el pipeline final. Esta auditoría verificó 349 de 349 entregas con etiqueta oficial anonimizada, 0 valores nulos en target_technical_grade_pct, 0 submission_id duplicados, rango completo entre 0 y 100, coincidencia exacta entre el target del dataset y labels_final.csv, y unión uno a uno contra la rúbrica oficial. Con base en esta validación, el estado de etiqueta del dataset final quedó como auto_verified_official_rubric_join.

También se identificó que las pruebas funcionales debían incorporarse únicamente cuando existiera evidencia ejecutada o validada. El código final registra 6 oráculos validados, 84 filas funcionales utilizables, 52 ejecuciones materializadas, 28 revisiones por regla estática y 4 bloqueos por seguridad; además, mantiene 1075 filas como pendientes por definición de oráculo. La auditoría funcional declara explícitamente que el script final no reconstruye toda la ejecución estudiantil ni sustituye un sandbox institucional. Por ello, la ablación combinado-sin funcional, con una diferencia de MAE de 0,0245 puntos, confirma que no se debe atribuir la mejora global del 38,08 % al piloto funcional.

De igual manera, se excluyeron detectores y variables que no contaban con validación manual suficiente. Esta decisión redujo el riesgo de que el modelo aprendiera patrones desde evidencia incompleta o no auditada. Las métricas de código se interpretan como variables proxy de calidad, no como una medición integral de calidad de software en el sentido estricto de ISO/IEC 25010:2023.

El uso de validación anidada también fortaleció el diseño experimental, ya que la selección del modelo se realizó dentro de la validación interna y la evaluación se reportó sobre predicciones externas. Esto reduce el riesgo de presentar un desempeño optimista por seleccionar el mejor modelo sobre los mismos resultados usados para evaluarlo.

El análisis por actividad mostró una limitación adicional de validez externa. Cuando se dejó fuera una actividad completa, el MAE ponderado fue de 20,65 puntos y solo el 29,23 % de las predicciones quedó dentro de ±10 puntos; además, 13 de 16 valores de R^2 calculables fueron negativos. Por tanto, el dominio de uso recomendado se restringe a estudiantes no vistos dentro de actividades similares a las observadas, y no a actividades nuevas sin validación específica.

2.4 Discusión frente a la literatura y validez externa

Los resultados son coherentes con la literatura sobre evaluación automática y asistida de programación, la cual señala que las pruebas funcionales suelen ser útiles para verificar corrección, pero no cubren por completo dimensiones como legibilidad, documentación, mantenibilidad y diseño. Por esta razón, el uso de métricas estáticas y variables de contexto académico puede aportar información complementaria, aunque debe interpretarse como evidencia predictiva y no como medición absoluta de calidad.

La importancia por permutación mostró que variables como líneas físicas transformadas, importaciones, tamaño del código, comentarios, funciones, complejidad y criterios curriculares se relacionan con la estimación de la nota. Esto coincide con la idea de que la calificación de programación combina corrección, estructura y criterios de rúbrica. Sin embargo, estas asociaciones no deben interpretarse causalmente, ya que el modelo fue diseñado para predicción y no para explicar mecanismos causales.

La evaluación por actividad no vista evidenció una limitación relevante: el desempeño disminuye cuando se intenta generalizar a actividades completamente nuevas. En consecuencia, el uso recomendado del modelo corresponde a estudiantes no vistos dentro de actividades similares o ya representadas en el entrenamiento. Para aplicar el modelo a nuevas actividades

se requiere validación adicional, actualización de rúbricas y ampliación de oráculos funcionales.

Capítulo III: Conclusiones y Recomendaciones

3.1 Conclusiones

3.1.1 Respecto al objetivo general, la investigación permitió diseñar, implementar y evaluar un modelo basado en aprendizaje automático para estimar calificaciones de tareas de programación mediante la integración de resultados funcionales parciales y métricas de calidad de código. El objetivo se cumplió en alcance experimental controlado, con uso recomendado como apoyo docente y no como sustituto de la calificación humana.

3.1.2 El modelo ganador final fue Ridge con selección anidada por estudiante. El ganador mayoritario fue ridge_alpha_1 y el protocolo obtuvo un MAE de 14,78 puntos, un RMSE de 20,31 y un R^2 de 0,52. Estos resultados evidencian que las variables del código y del contexto académico aportan información relevante para la estimación de calificaciones, aunque no son suficientes para sustituir el criterio del docente.

3.1.3 Con relación a la pregunta de investigación, los resultados permiten afirmar que la combinación de métricas de código, contexto académico y evidencia funcional piloto permite estimar calificaciones con mejor desempeño que un baseline de mediana. No obstante, la ablación mostró que el aporte incremental del piloto funcional actual es mínimo; por ello, la evidencia más sólida del experimento corresponde al uso combinado de métricas estáticas y variables curriculares auditadas.

3.1.4 En relación con OE1 y OE2, se analizaron enfoques de evaluación automática y asistida, y se caracterizó un dataset de 349 entregas, 28 estudiantes anonimizados, 17 actividades, 3 resultados de aprendizaje y 8 criterios de evaluación. La variable objetivo correspondió a la calificación oficial normalizada de la rúbrica en escala de 0 a 100.

3.1.5 En relación con OE3, se prepararon métricas estáticas, atributos estructurales y variables funcionales piloto para las entregas cubiertas por oráculos validados. En relación con OE4, OE5 y OE6, se implementaron modelos supervisados, se evaluaron mediante validación anidada por estudiante y se compararon contra un baseline de mediana, manteniendo separación entre utilidad docente y calificación automática.

3.2 Limitaciones

3.2.1 La principal limitación del estudio es que los datos provienen de un único paralelo, por lo que los resultados no pueden generalizarse automáticamente a otros cursos, cohortes o contextos educativos. El dataset contiene 349 entregas de 28 estudiantes, lo cual permite una evaluación inicial dentro del curso analizado, pero no representa una población amplia.

3.2.2 Otra limitación es que las métricas estáticas del código son variables proxy. Estas permiten describir características observables, pero no garantizan que una solución sea funcionalmente correcta. Por tanto, deben interpretarse como aproximaciones y no como una medición definitiva de calidad.

3.2.3 Además, las pruebas funcionales fueron integradas solo como piloto parcial. El pipeline conserva resultados funcionales materializados y auditados, pero no reconstruye la ejecución completa de todos los códigos estudiantiles. La cobertura completa requiere ampliar oráculos, ejecutar todos los casos en un entorno aislado, controlar tiempo y recursos, auditar la unión entre resultados funcionales y entregas, y repetir la ablación para cuantificar el aporte real de ese bloque.

3.2.4 En relación con la privacidad, el proyecto generó un dataset público anonimizado y un paquete público separado. La auditoría del release público registró 39 archivos incluidos, 0 archivos esperados faltantes y 0 alertas de privacidad, por lo que el paquete compartible no incluye archivos raw, mapas de identidad, claves HMAC ni rutas nominativas. Los archivos crudos y restringidos deben conservarse únicamente en el entorno privado de investigación.

3.3 Recomendaciones

3.3.1 Se recomienda utilizar el modelo como herramienta de apoyo docente para priorizar revisiones, analizar patrones y detectar entregas que podrían requerir mayor atención. El MAE de 14,78 puntos cumple el umbral de apoyo docente definido, pero el 44,41 % de predicciones dentro de ±10 puntos y el desempeño más bajo en actividades no vistas impiden recomendarlo como calificador automático.

3.3.2 También se recomienda ampliar el dataset con nuevos paralelos, cohortes y actividades, con el fin de evaluar la capacidad de generalización del modelo. La evaluación por actividad evidenció debilidad frente a actividades no vistas; por tanto, cualquier uso en nuevas actividades debe acompañarse de validación adicional.

3.3.3 Finalmente, se recomienda mantener procesos de anonimización, auditoría y revisión humana cuando se trabaje con datos académicos de estudiantes, debido a la sensibilidad de la información y al impacto que una predicción puede tener en procesos de evaluación.

3.4 Aportes

3.4.1 El principal aporte de esta investigación es la construcción de un dataset estructurado, anonimizado y trazable a partir de entregas reales de programación. Este dataset integra información de actividades, resultados de aprendizaje, criterios de evaluación, métricas del código y calificaciones de referencia.

3.4.2 Otro aporte corresponde al diseño de un flujo reproducible para preparar datos educativos orientados al entrenamiento de modelos predictivos. Este flujo se centralizó en el comando python src/run_final_pipeline.py. En su versión final, el pipeline primero ejecuta integrate_functional_pilot_features.py para restaurar e integrar el dataset canónico con 349 entregas, 45 columnas y ocho variables functional_pilot_*. Después ejecuta finalize_label_audit.py para verificar la unión uno a uno entre entregas y etiquetas oficiales anonimizadas; luego realiza la selección anidada del modelo mediante run_nested_selection_final.py, actualiza evidencias, ejecuta validate_canonical_results.py para detener el cierre si cambia el ganador o las métricas esperadas, regenera diagnósticos y figuras, audita el piloto funcional, calcula intervalos bootstrap agrupados por estudiante, refresca el manifiesto de reproducibilidad y construye el paquete público anonimizado. De esta forma, la reproducción no depende de una copia manual ni de una celda aislada del notebook, sino de una secuencia explícita de scripts y una comprobación automática del resultado canónico.

3.4.3 La investigación también aporta una escala operativa de utilidad para interpretar modelos de estimación de calificaciones. Esta escala distingue el uso como apoyo docente del uso como calificación automática y se aplica de forma explícita sobre MAE, mejora frente al baseline y porcentaje de predicciones dentro de ±10 puntos.

3.4.4 Finalmente, el trabajo aporta una evaluación experimental con validación anidada por estudiante, importancia por permutación calculada en folds externos, intervalo por remuestreo agrupado por estudiante, figuras generadas desde las predicciones finales, auditoría semántica de etiquetas, auditoría de privacidad del paquete público y análisis de ablación para separar el aporte del contexto académico y del piloto funcional.

Referencias

Shearer, C. (2000). The CRISP-DM model: The new blueprint for data mining. Journal of Data Warehousing, 5(4), 13-22.

Python Software Foundation. (2001). PEP 8 - Style Guide for Python Code. https://peps.python.org/pep-0008/

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825-2830.

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2025). How consistent are humans when grading programming assignments? ACM Transactions on Computing Education, 25(4), Article 49. https://doi.org/10.1145/3759256

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2024). Automated grading and feedback tools for programming education: A systematic review. ACM Transactions on Computing Education, 24(1), Article 10. https://doi.org/10.1145/3636515

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education, 19(1), Article 3. https://doi.org/10.1145/3231711

ISO. (2023). ISO/IEC 25010:2023, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Product quality model. https://www.iso.org/standard/78176.html

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. Technometrics, 12(1), 55-67. https://doi.org/10.1080/00401706.1970.10488634

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. Annals of Statistics, 29(5), 1189-1232. https://doi.org/10.1214/aos/1013203451

Breiman, L. (2001). Random forests. Machine Learning, 45, 5-32. https://doi.org/10.1023/A:1010933404324

Kapoor, S., Cantrell, E. M., Peng, K., et al. (2024). REFORMS: Consensus-based recommendations for machine-learning-based science. Science Advances, 10(18), eadk3452. https://doi.org/10.1126/sciadv.adk3452

Anexos

Los anexos se incluyen como evidencia metodológica y técnica del estudio. No contienen nombres reales, rutas nominativas, mapas de identidad, claves HMAC ni datos crudos de estudiantes.

Anexo A. Diccionario de datos del dataset final

El dataset final utilizado para el modelado contiene 349 entregas individuales calificadas y 45 columnas persistidas. La variable objetivo es target_technical_grade_pct, interpretada como calificación oficial normalizada en escala de 0 a 100. El estado de etiqueta del dataset final es auto_verified_official_rubric_join, debido a que la auditoría verificó unión uno a uno, ausencia de nulos, ausencia de duplicados y coincidencia con labels_final.csv. Después de la depuración final de archivos, la carpeta data/processed conserva únicamente los CSV necesarios para el cierre: dataset_model_final.csv, dataset_public_final.csv, labels_final.csv, variant_assignment_final.csv, functional_results_final.csv y group_submissions.csv.

Grupo de variables	Ejemplos	Uso en el estudio
Identificadores anonimizados	submission_id, actor_id	Permiten trazabilidad y validación agrupada sin exponer identidad real.
Contexto académico	activity_id, rda, criterion_id, attempt_type	Relacionan cada entrega con actividad, resultado de aprendizaje y criterio de evaluación.
Métricas estáticas del código	physical_lines, code_lines, comment_lines, function_count, import_count	Describen estructura, tamaño, documentación y organización del código fuente.
Complejidad y sintaxis	parse_success_rate, max_function_complexity, mean_function_complexity	Aproximan dificultad estructural y posibilidad de analizar el código.

Variables funcionales piloto	functional_pilot_tests_assigned, functional_pilot_tests_evaluable, functional_pilot_tests_passed, functional_pilot_tests_executed...	Registran evidencia funcional parcial validada para las entregas cubiertas por oráculos.
Variable objetivo	target_technical_grade_pct	Calificación oficial normalizada usada para entrenar y evaluar el modelo.

Anexo B. Protocolo de reproducción

El flujo reproducible se ejecuta desde la raíz del proyecto TesisDatasetManual mediante el comando python src/run_final_pipeline.py. La reproducción debe realizarse con Python 3.13.9, las dependencias registradas en requirements.txt y el entorno descrito en configs/runtime_environment_final.txt. Para estabilizar la estimación numérica, los modelos Ridge del protocolo final usan solver="cholesky". El orden final del pipeline es el siguiente: integrate_functional_pilot_features.py integra las variables funcionales piloto; finalize_label_audit.py valida la consistencia de etiquetas y variable objetivo; run_nested_selection_final.py ejecuta la selección anidada por estudiante; update_final_evidence_after_nested.py actualiza resultados oficiales; validate_canonical_results.py verifica que el ganador y las métricas coincidan con configs/canonical_results_final.json; run_final_model_diagnostics.py regenera métricas y figuras; run_functional_student_pilot.py audita resultados funcionales ya materializados y no ejecuta todos los códigos estudiantiles; consolidate_functional_results_final.py resume la cobertura del piloto; compute_actor_bootstrap_intervals.py calcula la incertidumbre agrupada por estudiante; refresh_reproducibility_manifest.py actualiza los hashes de reproducibilidad; y build_public_release_final.py genera el paquete público anonimizado.

Después de la ejecución final, dataset_model_final.csv y dataset_public_final.csv conservan el mismo estado técnico: 349 filas, 45 columnas, ocho variables funcionales piloto y estado de etiqueta auto_verified_official_rubric_join. La validación canónica confirmó el modelo ridge_alpha_1, la selección ridge_alpha_1 en 209 registros y ridge_alpha_3 en 140 registros, MAE 14,7812, RMSE 20,3118, R^2 0,5233 y 44,41 % de predicciones dentro de ±10 puntos. El cierre de reproducibilidad verificó 38 de 38 hashes válidos en el manifiesto, incluyendo datasets, métricas, predicciones, figuras, configuración, entorno, requirements y scripts del pipeline. El paquete público release/public/tesis_dataset_public_release.zip contiene 39

archivos, sin rutas crudas, mapas de identidad ni claves sensibles. Las versiones intermedias v2/v3 y reconstrucciones auxiliares no se consideran entregables finales; se mantuvo una única versión de dataset para modelado y una única versión pública anonimizada.

Paso	Script	Funci?n dentro del pipeline final
1	src/integrate_functional_pilot_features.py	Integra las variables functional_pilot_* al dataset can?nico antes del modelado.
2	src/finalize_label_audit.py	Valida etiquetas, variable objetivo, rango 0 a 100 y unión con la rúbrica oficial anonimizada.
3	src/run_nested_selection_final.py	Ejecuta la selecci?n anidada por estudiante y genera predicciones fuera de muestra.
4	src/update_final_evidence_after_nested.py	Sincroniza resultados oficiales, umbral de utilidad, m?tricas maestras y evidencia final.
5	src/validate_canonical_results.py	Compara winner_final.json contra canonical_results_final.json y detiene el cierre si cambian ganador o m?tricas.
6	src/run_final_model_diagnostics.py	Regenera figuras, importancia por permutaci?n y evaluaci?n por actividad.
7	src/run_functional_student_pilot.py	Audita resultados funcionales ya materializados; no ejecuta

		masivamente las 1.159 filas de código estudiantil.
8	src/consolidate_functional_results_final.py	Resume la cobertura del piloto funcional y sus estados de ejecuci?n, revisi?n o bloqueo.
9	src/compute_actor_bootstrap_intervals.py	Calcula intervalos bootstrap agrupados por estudiante para cuantificar incertidumbre.
10	src/refresh_reproducibility_manifest.py	Actualiza el manifiesto de hashes de datasets, m?tricas, figuras, configuraci?n, entorno y scripts.
11	src/build_public_release_final.py	Reconstruye el paquete público anonimizado con los artefactos finales permitidos.

Anexo C. Matriz de trazabilidad metodológica

La trazabilidad del estudio conecta fuentes originales, entregas anonimizadas, rúbricas, criterios, etiquetas, variables predictoras y resultados del modelo. La matriz se resume sin incluir material sensible.

Elemento	Archivo o evidencia	Propósito
Fuentes originales	data/raw	Conservar entregas, syllabus y rúbricas en entorno privado.
Inventario	source_manifest.csv	Registrar fuentes, tamaños y hashes de control.
Entregas extraídas	submission_manifest.csv	Relacionar cada archivo con entrega y actividad anonimizada.

Etiquetas	labels_final.csv	Unir entrega con calificación oficial normalizada.
Dataset final	dataset_model_final.csv	Tabla usada para entrenamiento y evaluación final.
Predicciones externas	nested_oof_predictions_final.csv	Evaluar desempeño sobre estudiantes no vistos.
Auditorías	results/audits	Controlar uniones, privacidad, reproducibilidad y cumplimiento.

Anexo D. Control de privacidad y exclusiones

Debido a que las fuentes corresponden a entregas académicas reales, el proyecto separa los materiales privados del paquete público. Los identificadores reales, rutas nominativas y claves de anonimización no forman parte del dataset de modelado ni de la entrega pública.

Riesgo	Medida aplicada	Estado
Identidad del estudiante	Uso de actor_id y submission_id anonimizados.	Controlado
Rutas nominativas	Exclusión de rutas originales y nombres reales.	Controlado
Mapas de identidad	Conservación solo en data/restricted.	No público
Claves HMAC	Exclusión del paquete público.	No público
Datos crudos	No incluir data/raw en anexos públicos.	No público
Calificación automática	Uso restringido a apoyo docente con revisión humana.	Delimitado

Anexo E. Evidencia experimental final

La evidencia experimental final se calculó con validación anidada agrupada por estudiante. El ciclo interno selecciona el modelo y sus hiperparámetros; el ciclo externo estima el desempeño sobre estudiantes no vistos. El resultado final canónico fue Ridge con solver="cholesky", MAE 14,78, RMSE 20,31 y R^2 0,52, con mejora de 38,08 % frente al baseline de mediana y 44,41 % de predicciones dentro de ±10 puntos. Estos valores se controlan mediante validate_canonical_results.py, que compara winner_final.json contra configs/canonical_results_final.json antes de cerrar el manifiesto y el paquete público.

Métrica	Baseline mediana	Modelo final Ridge
MAE	23,87	14,78
RMSE	32,00	20,31
R^2	-0,18	0,52
Mejora MAE	Referencia	38,08 %
Predicciones dentro de ±10	No aplica	44,41 %
Umbral de apoyo docente	No aplica	Cumple
Calificación automática	No aplica	No cumple
Escenario de ablación	**MAE**	**Interpretación**
S0_full_dataset_final	14,75	Dataset final con variables estaticas, contexto academico y piloto funcional parcial.
S1_without_functional_pilot	14,77	Retira columnas functional_pilot_* para medir si el piloto funcional parcial aporta senal predictiva.
S2_without_activity_rda_criterion	17,21	Retira identificadores curriculares directos solicitados por el tutor.
S3_without_extended_activity_context	17,59	Retira contexto academico ampliado para observar

		dependencia de actividad/rubrica.
Variable	**Importancia media**	**Nota**
log_physical_lines	2,49	Permutación en folds externos; no causal
import_count	2,31	Permutación en folds externos; no causal
log_total_size_bytes	1,95	Permutación en folds externos; no causal
comment_lines	1,92	Permutación en folds externos; no causal
function_count	1,73	Permutación en folds externos; no causal

Anexo F. Evidencias del producto implementado

Como complemento a la matriz de trazabilidad, el producto técnico se evidencia mediante los archivos generados por el pipeline final. Estos archivos permiten verificar datos, modelos, métricas, predicciones, auditorías, reproducibilidad y paquete público sin exponer información nominativa de estudiantes.

Tipo de evidencia	Archivo principal	Propósito
Dataset de modelado	data/processed/dataset_model_final.csv	Tabla final utilizada para entrenar y evaluar modelos.
Dataset público	data/processed/dataset_public_final.csv	Versión anonimizada y

Tipo de evidencia	Archivo principal	Propósito
		publicable del dataset final.
Etiquetas	data/processed/labels_final.csv	Unión validada entre entregas y calificaciones normalizadas.
Resultados funcionales	data/processed/functional_results_final.csv	Evidencia funcional piloto materializada y auditada.
Resultados de modelos	results/metrics/results_master_final.csv	Comparación entre baseline y modelo final.
Ganador canónico	results/metrics/winner_final.json	Modelo ganador y métricas oficiales reproducibles.
Predicciones	results/predictions/oof_predictions_final.csv	Predicciones externas usadas para análisis de error.
Manifiesto	results/audits/reproducibility_manifest_final.csv	Hashes y presencia de 38 archivos de reproducibilidad.
Release público	release/public/tesis_dataset_public_release.zip	Paquete anonimizado con 39 archivos.