UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

Facultad de Ingeniería

MAESTRÍA Y DOCTORADO EN CIENCIAS E INGENIERÍA

EVALUACIÓN COMPARATIVA DE ALGORITMOS DE APRENDIZAJE AUTOMÁTICO PARA LA PREDICCIÓN DE DEFECTOS UTILIZANDO MÉTRICAS ORIENTADAS A OBJETOS Y DE COMPLEJIDAD DEL SOFTWARE

TESIS PARA OBTENER EL GRADO DE
MAESTRÍA EN CIENCIAS
QUE PRESENTA

Yaretxy Arely Pelayo Lomelí

DIRECTOR
Dr. Juan Pablo García Vázquez

CODIRECTOR
Dr. Jesús Eduardo Soto Vega

Mexicali, Baja California, a 16 diciembre de 2025

Tesis aprobada por el siguiente comité de tesis:

_____	_____
Dr. Juan Pablo García Vázquez	Dr. Jesús Eduardo Soto Vega
Director de tesis	**Codirector de tesis**
_____	_____
Dra. María Angélica Astorga Vargas	Dr. Jorge Eduardo Ibarra Esquer
Sinodal	**Sinodal**

_____
Dra. Araceli Celina Justo López
Sinodal

_____
Dr. Juan Pablo García Vázquez
Coordinador de Investigación y Posgrado
Facultad de Ingeniería

Dedicatoria

A mi prometido Daniel Cuevas González que me apoyó en todo momento, inspirándome a crecer y creer en mí, a mis padres, padrinos, amigos y compañeros que confiaron en mí en este proceso de superación profesional.

Agradecimientos

A Dios, por darme fuerza, sabiduría y perseverancia para cumplir con esta etapa de formación.

A mi prometido, por su paciencia, apoyo y ánimo durante este proceso.

A mi familia, por su apoyo incondicional, amor y comprensión.

A mis asesores el Dr. Juan Pablo García Vázquez y al Dr. Jesús Eduardo Soto Vega por la paciencia, apoyo, orientación y consejos dados en todo momento.

A mi comité de tesis, a la Dra. María Angelica Astorga Vargas, al Dr. Jorge Eduardo Ibarra Esquer y a la Dra. Araceli Celina Justo López, por sus recomendaciones y enseñanzas que me permitieron mejorar.

A la Dra. Yessica Espinosa Díaz y la Dra. Eilen Oviedo González, por su apoyo y comprensión durante el desarrollo de mis estudios.

A la Universidad Autónoma de Baja California, Facultad de ingeniería, por la oportunidad de ingreso a su programa de posgrado.

A la Secretaría de Ciencia, Humanidades, Tecnología e Innovación de México (SECIHTI) por el apoyo económico para mis estudios profesional.

Resumen

La predicción de defectos de software representa un desafío constante para los desarrolladores, debido a que se realizan manualmente, consumen mucho tiempo y suelen ser costosas. Las métricas de software juegan un papel clave en la predicción de defectos de software. El objetivo de esta tesis es evaluar y comparar el desempeño de diversos algoritmos de aprendizaje automático para predecir defectos de software utilizando métricas orientadas a objetos y métricas de complejidad. Se utilizó el conjunto de datos MJ12A, compuesto por 19,148 registros y 20 métricas. Se aplicaron técnicas de preprocesamiento de datos como balanceo de clases a través de SMOTE *(Synthetic Minority Over-Sampling Technique)*, normalización y se implementaron diversos algoritmos como *K-Nearest Neighbors* (KNN), *Naïve Bayes* (NB), *Random Forest* (RF), *Support Vector Machine* (SVM) y *Perceptron Multicapa* (MLP). Se realizaron tres experimentos utilizando: 1) métricas de complejidad, 2) métricas orientadas a objetos y 3) la combinación de métricas de complejidad y orientadas a objetos. Para determinar el desempeño de los algoritmos se utilizaron métricas de evaluación como exactitud, precisión, *recall*, sensibilidad, especificidad, F1-score, kappa y G-media lo que permitió conocer su capacidad predictiva. El algoritmo con mejor rendimiento fue RF con una exactitud del 87.23% y un *F1-score* de 86.67%. Para cada algoritmo se utilizaron múltiples configuraciones de hiperparámetros con el objetivo aplicar pruebas *t-student* con *p-valor* < 0.05 de significancia. Se confirmó que RF supera de forma significativa a los demás algoritmos. Otro hallazgo interesante es que la combinación de métricas orientadas a objetos y de complejidad permiten predecir defectos con mayor precisión que un solo un tipo de métrica.

Palabras clave: Aprendizaje automático, Predicción de defectos, Pruebas de software, Métricas de software.

Abstract

Predicting software defects is a constant challenge for developers, as it is done manually, it's time-consuming, and it's often expensive. Software metrics play a key role in predicting software defects. The objective of this thesis is to evaluate and compare the performance of multiple machine learning algorithms for predicting software defects using object-oriented metrics and complexity metrics. The MJ12A dataset, consisting of 19,148 records and 20 metrics, was used. Data preprocessing techniques such as class balancing through SMOTE (Synthetic Minority Over-Sampling Technique) and normalization were applied, and various algorithms such as K-Nearest Neighbors (KNN), Naïve Bayes (NB), Random Forest (RF), Support Vector Machine (SVM), and Multilayer Perceptron (MLP) were implemented. Three experiments were conducted using: 1) complexity metrics, 2) object-oriented metrics, and 3) a combination of complexity and object-oriented metrics. To determine the performance of the algorithms, evaluation metrics such as accuracy, precision, recall, sensitivity, specificity, F1-score, kappa, and G-mean were used, which allowed their predictive capacity to be determined. The best-performing algorithm was RF with an accuracy of 87.23% and an F1-score of 86.67%. Multiple hyperparameter configurations were used for each algorithm to apply t-student tests with a p-value < 0.05 significance. The results of the t-student tests confirmed that RF significantly outperforms the other algorithms. Another interesting finding is that the combination of object-oriented and complexity metrics allows defects to be predicted with greater accuracy than a single type of metric.

Keywords: Machine learning, defect prediction, software testing, software metrics.

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Capítulo 1: Introducción

1.1 Contexto del Problema

Actualmente, el paradigma de programación orientada a objetos (*POO*) es el más utilizado en el desarrollo de software (Deshpande et al., 2020). Según el informe anual del IEEE *The top programming languages*, los lenguajes de programación más utilizados en el desarrollo de software son los lenguajes orientados a objetos en los que destacan *Python*, *Java*, *C++*, *C* y *PHP* (Cass, 2025). A pesar de la existencia de metodologías para el desarrollo de software bajo el paradigma *POO*, el software sigue presentando defectos (Durelli et al., 2019).

Para asegurar la calidad del software, uno de los aspectos clave es prevenir defectos, esto a través de pruebas de software que permitan identificar problemas en un programa que desencadenan resultados no deseados, tales como fallos o pérdida de datos (S. K. Singh & Singh, 2012). Las pruebas de software son una actividad fundamental dentro del ciclo de vida de desarrollo de software (Durelli et al., 2019; *IEEE Std 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology* | *SE Goldmine*, n.d.).

Para el diseño y la ejecución de las pruebas, la norma *ISO/IEEE 29119:2022* sugiere tres técnicas: 1) Pruebas basadas en especificaciones asociadas a requisitos funcionales, estas pruebas también se conocen como pruebas de caja negra porque se realizan a nivel de interfaces de usuario sin evaluar la estructura interna del software; 2) Pruebas basadas en la estructura interna en la que se explora el código, estas pruebas se conocen como pruebas de caja blanca; y 3) Pruebas basadas en la experiencia del evaluador humano.

Las pruebas de caja negra son las más utilizadas, pero son menos efectivas en la detección de defectos, en comparación con las pruebas de caja blanca (Dustin, 2003). Esto se debe a que el diseño de caja blanca es más complejo ya que utiliza rutas, instrucciones y ramas durante la ejecución de las pruebas. Esto hace que las pruebas de software sean una actividad compleja y costosa, especialmente cuando el software se vuelve más complejo

(Canaparo et al., 2022; Durelli et al., 2019). Por lo tanto, no es posible probar, identificar y corregir completamente los defectos del software (Durelli et al., 2019; *IEEE Std 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology* | *SE Goldmine*, n.d.). En este sentido, se han establecido métricas de software para la predicción de defectos, basadas en diferentes atributos del software como tamaño, complejidad, acoplamiento, entre otros (Canaparo et al., 2022; Deshpande et al., 2020). En particular, las métricas de software orientadas a objetos juegan un papel clave en la predicción de defectos (Deshpande et al., 2020). Por otro lado, para optimizar las pruebas de *software*, se han desarrollado herramientas automatizadas para la detección de defectos. En el trabajo presentado por (Ricca et al., 2025) se presentan avances significativos en las herramientas automatizadas impulsadas con la IA donde normalmente se enfocan en tareas de detección de defectos a partir de la interfaz de usuario y la generación automática de *scripts*, mayormente en pruebas de caja negra. Sin embargo, de acuerdo con Karhu, Kasurinen, and Smolander (2025), las herramientas automatizadas basadas en IA aún no han logrado un avance significativo en las pruebas de *software* (Baqar & Khanda, 2024; Karhu et al., 2025).

En la Tabla 1.1 se encuentran algunas herramientas que utilizan Inteligencia Artificial (IA) para detectar defectos de *software*.

Tabla 1.1. Herramientas Automatizadas

Herramienta	¿Utiliza IA?	Licencia	Técnicas de pruebas
(*Unified Functional Testing*, 2025)	sí	Comercial	Caja Negra
(*Selenium*, 2025)	no	Código abierto	Caja Negra
(Keysight, 2025)	sí	Comercial	Caja Negra
(*Ranorex*, 2021)	sí	Comercial	Caja Negra

(*TestComplete*, 2025)	sí	Comercial	Caja Negra y Blanca
(*Functionize*, 2025)	sí	Comercial	Caja Negra
(*Applitools*, 2025)	sí	Comercial	Caja Negra
(*Mabl*, 2025)	sí	Comercial	Caja Negra
(Testim, 2025)	sí	Comercial	Caja Negra
(Appvence, 2025)	sí	Comercial	Caja Negra
(Microsoft Visual Tests, 2025*)*	sí	Comercial	Caja Blanca

El objetivo principal de estas herramientas es automatizar la ejecución de las pruebas, mientras que este estudio hace uso del aprendizaje automático que permite la detección temprana de defectos y la toma de decisiones con un enfoque en pruebas de caja blanca. Asimismo, se han desarrollado soluciones automatizadas para la predicción de defectos con la aplicación del aprendizaje automático debido al potencial que posee para procesar grandes volúmenes de datos (Mehmood et al., 2023).

Esto ha motivado el interés tanto de profesionales como de investigadores por mejorar la efectividad y eficiencia de las pruebas mediante la aplicación de diferentes algoritmos y técnicas para la predicción de defectos (Durelli et al., 2019; Saeed & Saleem, 2023; M. Singh & Chhabra, 2024). El aprendizaje automático es definido como una subdisciplina de la Inteligencia Artificial, que busca imitar la inteligencia humana. El aprendizaje automático permite entrenar algoritmos, a partir de datos ya existentes con el fin de descubrir patrones, para predecir y tomar decisiones, su principal ventaja es que puede realizar su trabajo automáticamente (Bishop & Nasrabadi, 2006; Dash et al.,2021).

El aprendizaje automático ofrece diversos métodos de predicción que nos permiten anticipar los defectos del código rediciendo el costo total, aumentando la confiabilidad y la calidad del producto final (Albattah & Alzahrani, 2024).

Los investigadores se han interesado en estudiar diferentes enfoques para la predicción de defectos, entre los cuales el aprendizaje automático ha demostrado ser el más exitoso(Catal & Diri, 2009; Ponnala & Reddy, 2021). Los algoritmos de clasificación más utilizados son *K-Nearest Neighbors* (K-vecinos más cercanos, KNN), *Random Forest* (Bosque aleatorio, RF), *Decision Tree* (Árboles de decisión, DT), *Support Vector Machine* (Máquinas de Vectores de Soporte, SVM), *Multilayer Perceptron* (Perceptrón Multicapa, MLP), y *Logistic Regression* (Regresión Logística, RL). Por otro lado, en el aprendizaje no supervisado, los algoritmos se encargan de descubrir patrones a partir del conjunto de datos no etiquetados, entre los algoritmos más típicos son *K-means* (K-medias) (Ali et al., 2023; Lu et al., 2022). Adicionalmente, en la revisión sistemática de la literatura se menciona que es posible predecir defectos en el software mediante el entrenamiento de algoritmos de aprendizaje automático. En el trabajo relacionado se encontró que los estudios utilizan en su mayoría métricas tradicionales, por lo que proponemos la exploración de métricas orientadas a objetos combinadas con métricas de complejidad (Ali et al., 2023; Kanth et al., 2024). Es por esto por lo que se ha considerado al aprendizaje automático una alternativa para entrenar modelos con características asociadas con la calidad del software (por ejemplo, métricas orientadas a objetos) con el propósito de identificar patrones o indicadores que puedan ayudar a predecir la ocurrencia de defectos de software. Teniendo en cuenta lo anterior, en esta tesis se presenta un análisis comparativo de varios algoritmos clásicos de aprendizaje automático, incluidos *KNN, NB, RF, SVM y MLP*. Estos algoritmos se entrenaron utilizando varias métricas de software para predecir defectos de software. Se realizaron tres experimentos: 1) los algoritmos se entrenaron con métricas de complejidad del software; 2) los algoritmos fueron entrenados con métricas orientadas a objetos; y finalmente, 3) los algoritmos se entrenaron con una combinación de métricas de complejidad de software y métricas orientadas a objetos.

1.2 Objetivo

1.2.1 Objetivo General

Evaluar el desempeño de algoritmos de aprendizaje automático entrenados para detectar defectos de software en pruebas estructurales basado en las métricas orientadas a objetos y métricas de complejidad.

1.2.2 Objetivos Específicos

- Diseñar y ejecutar una serie de experimentos para evaluar la eficacia de aprendizaje automático entrenados con métricas orientadas a objetos y de complejidad.
- Analizar y discutir los resultados obtenidos para proporcionar recomendaciones para la implementación exitosa en técnicas de algoritmos de aprendizaje automático en pruebas estructurales.

1.3 Metodología

Para alcanzar los objetivos de esta investigación se utilizó la metodología *CRISP-DM* (del inglés *Cross-Industry Standard Process for Data Mining*) que es el Proceso Estándar Inter-Industria para la Minería de Datos, es una metodología propuesta por Wirth y Hipp (Schröer et al., 2021; Wirth & Hipp, 2000) para el desarrollo de proyectos de ciencia de datos. Los principales objetivos de la metodología *CRISP-DM* es optimizar los proyectos de minería o ciencia de datos reduciendo el costo, obtener mayor confiabilidad, facilidad de repetición y mayor velocidad (Schröer et al., 2021). *CRISP-DM* consta de seis fases: comprensión del negocio, comprensión de los datos, preparación de los datos, modelado, evaluación y despliegue como se muestra en la Figura 1.1.

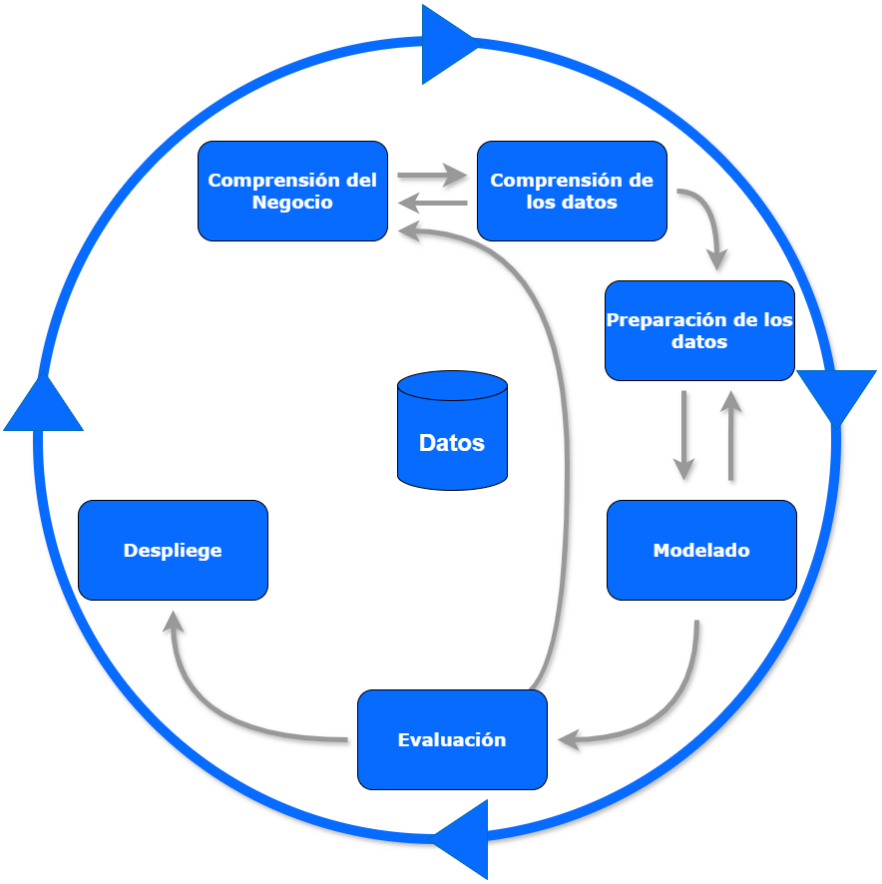

Figura 1.1. Fases de la metodología CRISP-DM

1.3.1 Comprensión del Negocio

Esta fase tiene como propósito comprender los objetivos y requerimientos del proyecto en un contexto empresarial. Llevando a cabo tareas para el cumplimiento de la fase como el establecimiento de objetivos del negocio, evaluación del problema central, la definición de objetivo del proyecto y la elaboración del plan de proyecto seleccionando tecnologías y herramientas para cada fase. Esto a su vez el equipo obtiene una visión clara del proyecto lo cual permite llevar a cabo la ruta del trabajo (Clark, 2018; García, 2019). En el presente trabajo se definió como objetivo general la evaluación de algoritmos para la detección temprana de defecto en el software, con métricas orientadas a objetos, métricas de complejidad, utilizando RStudio como entorno de desarrollo y pruebas estadísticas (*RStudio*, 2024), para el establecimiento de la ruta de trabajo en las siguientes fases de la metodología.

1.3.2 Comprensión de los Datos

Esta fase tiene como propósito el entendimiento y la recopilación de los datos con el fin de familiarizarse con su contenido para evaluar la calidad, la estructura, o encontrar hipótesis ocultas acerca de los datos. Esto se realiza mediante las tareas de recopilación, exploración, descripción y verificación de calidad. Como resultado se obtiene un informe inicial de los datos con sus características principales y su calidad (Clark, 2018; García, 2019). Durante esta fase se utilizó el entorno *RStudio* para llevar a cabo el análisis descriptivo, para descubrir datos atípicos y patrones relevantes en el conjunto de datos.

1.3.3 Preprocesamiento de Datos

El propósito de esta fase es preparar los datos para obtener una versión final adecuada para que sea utilizada en la fase del modelado. Para ello se realizan las tareas de la selección, limpieza, construcción, integración y transformación; como resultado se obtiene un informe sobre la preparación de los atributos (Clark, 2018; García, 2019). En esta fase se aplicó la transformación de atributos categóricos numéricos. La normalización para asegurar que todas las variables cuenten con una misma escala numérica, se aplicó el balanceo de las clases para mejorar la capacidad predictiva de los modelos y evitar el sesgo.

1.3.4 Modelado

Esta fase consiste en la selección y aplicación de técnicas de modelado, las actividades que se realizan son la selección de técnicas de modelado, generación de pruebas de diseño, construcción y evaluación del modelo (Clark, 2018; García, 2019). Se implementaron algoritmos de aprendizaje automático como KNN, NB, RF, SVM, MLP, con el objetivo de predecir la presencia de defectos en el código. Para el cumplimiento de esta fase se utilizó el entorno RStudio con los paquetes Caret, e1071, randomForest, y Class para la construcción y evaluación de los modelos. Para cada modelo se llevó a cabo la validación cruzada de con 10 k folds con el fin de validar del modelo (Cuevas et al., 2022) y la realización del ajuste de parámetros en los modelos para abordar los diferentes escenarios que se pudieran presentar.

1.3.5 Evaluación

Consiste en la evaluación de los resultados obtenidos de los modelos, esta fase nos permite hacer una revisión de proceso que se ha llevado a cabo desde la primera fase. Se analizaron los resultados obtenidos de cada modelo utilizando métricas que nos ayudan a evaluar el rendimiento de los algoritmos tales como la Precisión, Exactitud, F1-Score, Recall, Sensibilidad, Especificidad, Media Geométrica, Kappa Estadística, y seleccionar aquellos con la mejor capacidad predictiva.

1.3.6 Despliegue

En el proyecto la etapa de despliegue se llevó de manera parcial realizando los modelos, llevando a cabo la investigación. El propósito de esta fase es presentar los conocimientos adquiridos del proyecto, aquí se identifican la complejidad de la implementación. Las actividades que se deben realizar son el desarrollo del plan de implementación despliegue y el plan de mantenimiento para planes previos (Clark, 2018; García, 2019).

Capítulo 2: Pruebas de Software

Las pruebas de software es una etapa fundamental en el ciclo de vida del desarrollo del software que nos ayuda a evaluar la calidad, la funcionalidad, el rendimiento del software y prevenir defectos, reduciendo el tiempo y costo, esto a consecuencia de la complejidad en el software de hoy en día (Canaparo et al., 2022; Tadapaneni et al., 2022). Las pruebas de software se realizan a partir de una serie de procesos que determinan si el proyecto cumple con los requerimientos establecidos al inicio del proyecto. El objetivo de las pruebas no es demostrar que el software no contenga errores si no que el software sea confiable (Umar, 2020). Tomando en cuenta que las actividades realizadas por humanos son propensas a defectos, algunos estudios determinan que las pruebas en el desarrollo de software representan el 50% del costo, lo que conlleva que la pruebas son complejas y costosas (Durelli et al., 2019; S. K. Singh & Singh, 2012).

2.1 Técnicas de pruebas

Las técnicas de pruebas son el conjunto de procedimientos utilizados para el desarrollo de casos de pruebas, la realización de pruebas y el análisis de resultados a su vez nos ayudan a identificar las condiciones de pruebas (Umar, 2020). Existen diversas técnicas de pruebas como las pruebas de caja negra, pruebas de caja blanca y pruebas de caja gris, que nos ayudan a la detección de defectos y garantizar que el software funcione como se espera (Umar, 2020). Sin embargo, las pruebas de caja blanca son las mejores para la detección de defectos y la optimización del código, de acuerdo con (Capote, 2024).

2.1.1 Pruebas de Caja Negra

En las pruebas de caja negra, se evalúan los aspectos fundamentales a través de casos de prueba. Se evalúa la funcionalidad sin tener conocimiento alguno de la estructura interna del software, siendo efectivas en pruebas funcionales y pruebas de aceptación de usuario. Los casos de prueba son construidos en

torno a los requisitos establecidos; para cada caso de prueba, el evaluador verifica que las entradas y las salidas sean correctas (Capote, 2024; Umar, 2020). Cómo se interpreta en la Figura 2.1.

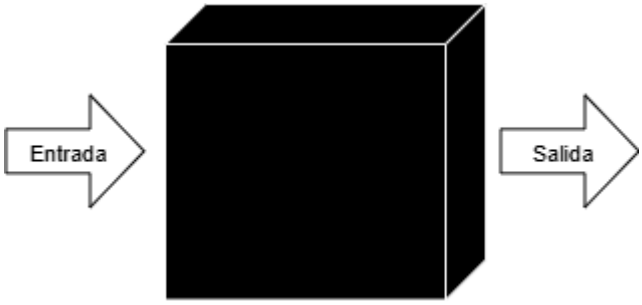

Figura 2.1. Pruebas de caja negra

2.1.2 Pruebas de Caja Blanca

Las pruebas de caja blanca, también conocidas como pruebas estructurales o pruebas de caja de vidrio, son significativamente efectivas, ya que no solo prueban la funcionalidad del software, sino también prueban la estructura interna del código. En esta estructura podemos encontrar sentencias de decisiones como *if, case, while*, entre otros caminos distintos en el código, llevando a cabo un proceso de prueba complejo. Para esto se requiere un conocimiento completo del código fuente y habilidades de programación para el diseño de casos de prueba. El evaluador selecciona entradas para verificar los caminos del programa y realizar una comparación de la salida con respecto a la salida esperada (Capote, 2024; Cristiá, 2017; Umar, 2020). Cómo se interpreta en la Figura 2.2.

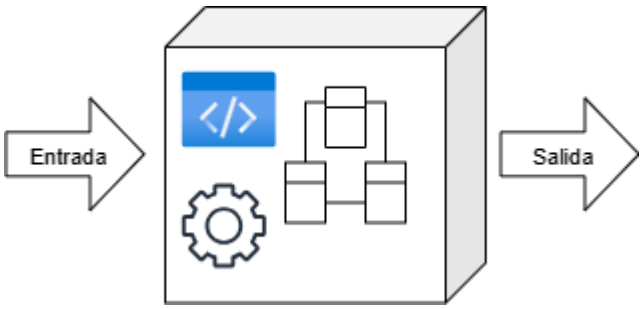

Figura 2.2. Pruebas de caja blanca

2.1.3 Pruebas de Caja Gris

Las pruebas de caja gris combinan elementos de las pruebas de caja negra y caja blanca. A partir de este enfoque se crean casos de pruebas donde se considera la funcionalidad externa del sistema parcialmente y el conocimiento parcial de la estructura interna (Umar, 2020). Como se interpreta en la Figura 2.3.

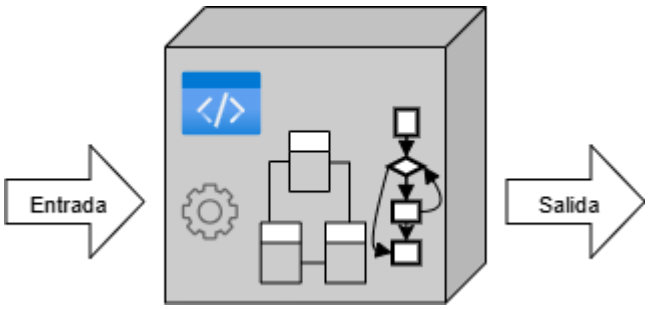

Figura 2.3. Pruebas de caja gris

2.2 Métricas de Software

Una métrica de software es una medida cuantitativa del grado en que un componente o sistema de proceso tiene un determinado atributo, según la definición del vocabulario de Ingeniería de Sistemas y Software en la norma *ISO/IEC/IEEE 24765:2010* ("IEEE Standard Glossary of Software Engineering Terminology," 1990). Estas métricas se han convertido en un elemento fundamental para la predicción y estimación de defectos (Deshpande et al., 2020; Ouellet & Badri, 2024). La finalidad de las métricas es evaluar el software para describir diferentes características del software, como el tamaño, la complejidad y calidad (Deshpande et al., 2020). Existen varias clasificaciones de métricas; entre las más utilizadas, métricas de complejidad, métricas de complejidad ciclomática y métricas orientadas a objetos propuestas por los autores Chidamber y Kemerer (CK) (Chidamber & Kemerer, 1994). Las métricas de software son ampliamente utilizadas en pruebas estructurales donde el conocimiento interno del código es necesario para estas pruebas. En este estudio se utilizaron métricas orientadas a objetos ya que el paradigma de

los lenguajes sigue esta tendencia considerada esencial para la predicción de defectos ya que estas métricas permiten medir, la cohesión, herencia, acoplamiento que son fundamentales para la predicción de módulos defectuosos, también se utilizaron métricas de complejidad ya que permiten medir las rutas lógicas (McCabe, 1976; Rebro et al., 2023). En la Tabla 2.1 se presentan las métricas utilizadas.

Tabla 2.1. Descripción de las métricas de software

Autor	Métrica	Descripción
(Deshpande et al., 2020)	*Average Method Complexity (AMC)* - Complejidad Promedio de Métodos	Calcula el tamaño promedio de los métodos en una clase, medido en número de instrucciones de *Java bytecode** que contiene cada método.
	Depth of Inheritance Tree (DIT) - Profundidad del Árbol de Herencia	Mide cuántos niveles de herencia tiene una clase desde ella hasta la clase raíz.
	Number of Children (NOC) - Número de Hijos	Mide el número de descendientes inmediatos de la clase.
	Measure of Functional Abstraction (MFA) - Medida de Abstracción Funcional	Es la relación entre el número de métodos heredados por una clase y el número total de métodos accesibles.
	Inheritance Coupling (IC) – Acoplamiento por Herencia	Mide el número de clases padre a las que esta acoplada una clase.
	Lack of Cohesion of Methods 3 (LCOM3) - Falta de Cohesión de Métodos 3	Variante que calcula la cohesión en los métodos.
	Cohesion Among Methods (CAM) - Cohesión entre Métodos	Calcula la relación entre métodos de una clase basado en la lista de parámetros.
	Coupling Between Methods (CBM) - Acoplamiento entre Métodos	Mide el acoplamiento que hay entre los métodos.
	Data Access Metric (DAM) - Métrica de Acceso a Datos	Es la relación entre el número de atributos privados y protegidos y el número total de atributos declarados en la clase.
	Measure of Aggregation (MOA) - Medida de Agregación	Mide el grado de relación parte-todo en una clase según la cantidad de campos de clase que son de tipos definidos por el usuario.
	Number of Public Methods (NPM) – Número de Métodos Públicos	Cuenta todos los métodos de una clase que están declarados como públicos.
	Maximum Cyclomatic Complexity (Max(CC)) - Complejidad Ciclomática Máxima	Valor CC más alto entre los métodos de la clase.

	Average Cyclomatic Complexity (Avg(CC)) - Complejidad Ciclomática Promedio	Media aritmética de CC en los métodos de la clase.
(Oullet & Badri, 2024)	*Weighted Methods per Class (WMC)* - Métodos Ponderados por Clase	El valor de WMC es igual al número de métodos en la clase (asumiendo un único peso para todos los métodos).
	Lines of Code (LOC) – Líneas de Código	Número total de líneas de código que tiene el programa/clase.
	Average Cyclomatic Complexity (ACC) - Complejidad Ciclomática Promedio	El método general de complejidad de McCabe promedio de una clase.
	Class Interface Size (CIS) – Tamaño de la Interfaz de la Clase	El número de métodos públicos de una clase.
	Afferent Coupling (Ca) – Acoplamiento Aferente	Mide el número de clases que dependen en la clase medida.
	Efferent Coupling (Ce) – Acoplamiento Eferente	Mide el número de clases en las que la clase medida depende.
	Response for a Class (RFC) – Respuesta para una Clase	Mide el número de métodos diferentes que se puede ejecutar cuando un objeto de esa clase recibe un mensaje. Incluye métodos internos y los llamados dentro del cuerpo de los métodos de la clase.
	Number of Methods (NOM) – Número de Métodos	Mide el número total de métodos de una clase, incluyendo métodos públicos, protegidos y estáticos.
(Mahamed et al., 2020)	*Depth of Inheritance Tree (DIT)* – Profundidad del Árbol de Herencia	Mide cuántos niveles de herencia tiene una clase desde ella hasta la clase raíz.
	Number of Children (NOC) - Número de Hijos	Mide el número de descendientes inmediato de la clase.
	Lack of Cohesion of Methods (LCOM) – Falta de Cohesión de Métodos	Cuenta los pares de métodos en una clase que no comparte el acceso a al menos un campo común. Esto se calcula restando el número de pares que comparten el acceso de aquellos que no lo hacen.
	Coupling Between Objects (CBO) – Acoplamiento entre Objetos	Representa el número de clases acopladas a una clase específica (incluye tanto referentes como acoplamientos aferentes). Estos acoplamientos ocurren a través de llamadas a métodos, acceso a campos, herencia, argumentos del método, tipos de retorno y excepciones.

*Bytecode: Conjunto de instrucciones generadas por java

Capítulo 3: Aprendizaje Automático

El aprendizaje automático, es una rama de la inteligencia artificial que permite a las máquinas aprender a partir de datos sin la necesidad de ser programadas con reglas explícitas. Se basa en métodos estadisticos y matemáticos que permiten detectar patrones en los datos y, con ello, realizar predicciones e incluso tomar decisiones (Olivas et al., 2023). Se reconocen tres paradigmas en el aprendizaje, aprendizaje supervisado, no supervisado, y por refuerzo. El aprendizaje supervisado es aquel en el que los datos tienen una salida esperada, a la cual se le suele llamar etiqueta. Los algoritmos de aprendizaje automático utilizan los datos y las etiquetas para reconocer patrones con el fin de predecir nuevos datos. El aprendizaje no supervisado, los datos no tienen una salida asociada busca descubrir patrones de datos no etiquetados. Por otro lado, el aprendizaje por refuerzo no dispone de un conjunto de datos como tal si no es conocido por imitar el aprendizaje de los humanos recibiendo retroalimentación basadas en acciones realizadas (Borrero & Arias, 2021; Lu et al., 2022). En este trabajo utilizamos el tipo de aprendizaje supervisado con los algoritmos KNN, NB, RF, SVM y MLP que serán descritos a continuación.

3.1 Algoritmos de Aprendizaje Supervisado

3.1.1 K-Nearest Neighbors

El algoritmo *K-Nearest Neighbors* (KNN) pertenece al grupo de los modelos de aprendizaje máquina y su tipo de aprendizaje es supervisado, su principal utilidad se observa en resolver tareas de clasificación. El algoritmo *KNN* funciona buscando los vecinos más cercanos a partir de un punto de búsqueda determinado con *K*, a través de distancias (Dash et al., 2021). En la Figura 3.1 se presenta el pseudocódigo de una *KNN*.

```
COMIENZO
    ENTRADA: D ={(X1,C1),...,(Xn,Cx)}
           X = (X1,...,Xn) Nuevo caso a clasificar
    PARA todo objeto ya clasificado (Xi,Ci)
           CALCULAR di = d(Xi,X)
    ORDENAR di(i = 1,...,N) en orden ascendente
    QUEDARNOS con los K casos D_X^K ya clasificador más cercanos a X
    ASIGNAR a X la clase más frecuente D_X^K
FIN
```

Figura 3.1. Pseudocódigo KNN

Consiste en tener un conjunto de datos de entrada representados por D donde cada X_i representa los atributos y C_i su clase. El nuevo caso representado por $X = (X_1 \ldots, X_n)$. Luego se calcula la distancia $d_i = d(X_i, X)$ del nuevo caso a clasificar representado por X. Se puede utilizar varias medidas de similitud como euclidiana, manhattan, minkowski por mencionar algunas. Una vez calculadas las distancias, se ordenan de menor a menor. Posteriormente, se define un valor de *K,* para seleccionar los *K* vecinos más cercanos al nuevo elemento que se va a clasificar. Hay varias heurísticas que se han propuesto para calcular el valor de *K*, tales como: (1) usar combinaciones de clase co-prima y *K*, (2) elegir una *K* que sea mayor o igual al número de clases más uno, y (3) elegir una *K* que sea lo suficientemente baja como para evitar el ruido (Kirk, 2017; Nelson, 2020).

3.1.2 Naïve Bayes

El clasificador *Naïve Bayes* (NB) es uno de los algoritmos más simples, basado en el teorema de bayes, destacando por su buen desempeño en tareas de clasificación. El algoritmo asume que un predictor (x) en una clase (c) es independiente, llamada dependencia condicional. La probabilidad subsecuente se calcula con la siguiente ecuación 3.1.

$$P(C|X) = \frac{P(X|C)\,P(C)}{P(C)} \quad (3.1)$$

Donde: $P(C|X)$ es la probabilidad posterior de la clase objetivo (C), $P(C)$ es la probabilidad apriori de la clase, $P(X|C)$ es la probabilidad del predictor dada la clase, y $P(X)$ es la probabilidad previa del predictor (Sayad, 2010; Zacharski, 2015).

3.1.3 Decision Tree

El algoritmo de *Decision Tree (DT)* es una técnica de aprendizaje supervisado que permite realizar tareas de clasificación y regresión. Se representa con una estructura similar a la de un diagrama de flujo, el nodo superior corresponde al nodo raíz, los nodos internos como atributos o características, las hojas se representan como las etiquetas o clases, y las ramas como subconjunto de atributos. Como se aprecia en la Figura 3.2.

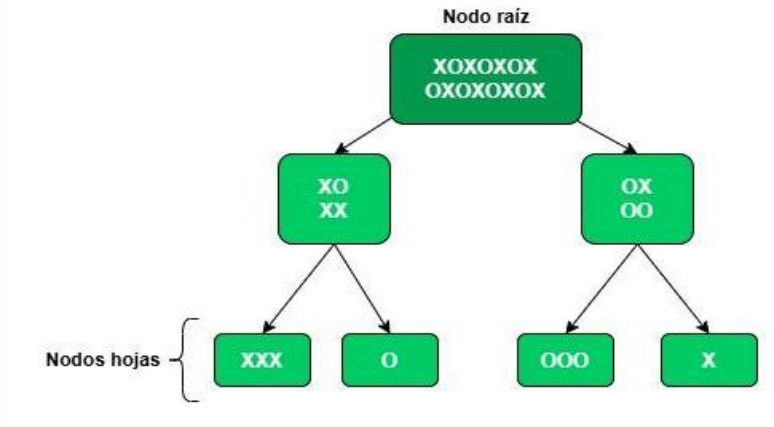

Figura 3.2. Estructura de DT

La selección de características se determina mediante el cálculo de métricas que evalúan qué atributo divide mejor los datos, un ejemplo es la entropía. Se calcula con la siguiente fórmula 3.2.

$$Entropía\ (S) = -\Sigma p_i log_2(p_i) \quad (3.2)$$

A partir de la entropía se calcula la ganancia de información que indica que atributo es el mejor para dividir (Ahmad Imran, 2024; Cuevas et al., 2022).

3.1.4 Random Forest

El algoritmo de *Random Forest (RF)* está basado en árboles de decisión y en la técnica de remuestreo (*bootstrap*). El *bootstrap* consiste en seleccionar aleatoriamente individuos de un conjunto de datos utilizando el muestreo con reemplazo para crear diferentes conjuntos de datos llamados muestras de *bootstrap*. A partir de cada uno de estos ejemplos, se genera un árbol de decisión. Cada árbol depende de valores aleatorios probados de forma independiente con la misma distribución para cada uno de ellos. Por último, los nuevos datos se predicen utilizando el voto por mayoría. En la Figura 3.3 se aprecia la estructura de RF.

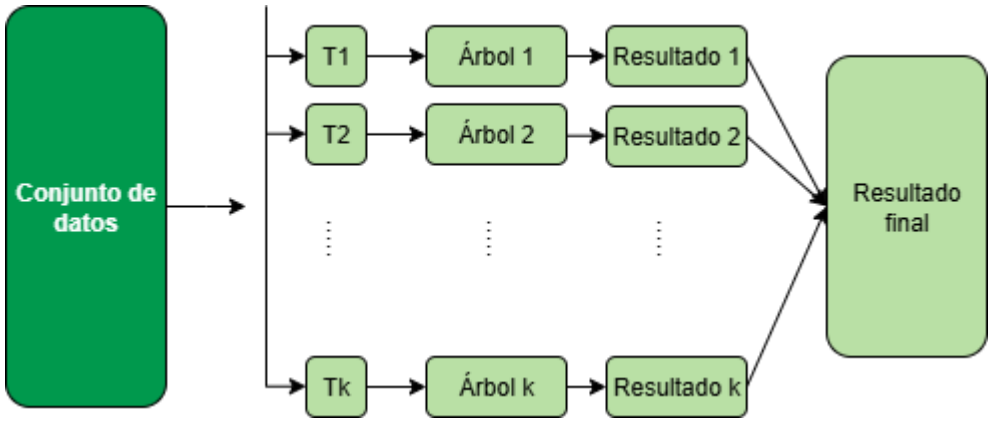

Figura 3.3. Estructura de RF

3.1.5 Support-Vector Machine

El algoritmo *Support-Vector Machine* (SVM) es un algoritmo de aprendizaje automático, donde el objetivo principal es la optimización. El algoritmo SVM realiza la clasificación mediante la definición del hiperplano óptimo que maximiza el margen entre dos clases como se observa en la Figura 3.4.

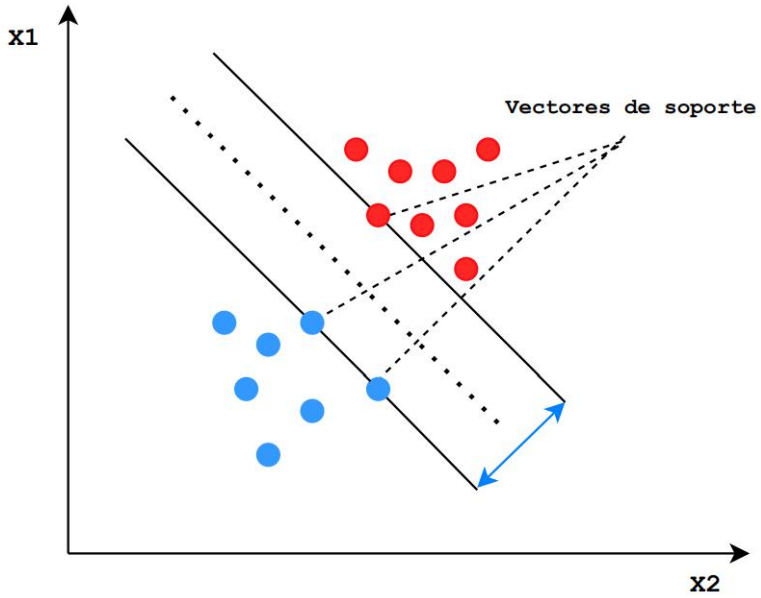

Figura 3.4. Ejemplo de funcionalidad de SVM

Los vectores que definen el hiperplano son llamados vectores de soporte (Pisner & Schnyer, 2020). Primero mapea los puntos de entrada a un espacio de características de una dimensión mayor y encuentra un hiperplano que los separa y maximiza el margen entre las clases (Pisner & Schnyer, 2020). Sin ningún conocimiento de mapeo, el algoritmo encuentra el hiperplano óptimo utilizando el producto puntual con funciones de espacio de características llamadas *kernels*. Las funciones *kernel* son funciones matemáticas, estas funciones son lineales, polinómicas, tangenciales y radiales (Gholami & Fakhari, 2017). Estas funciones son las que permiten convertir lo que sería un problema de clasificación no lineal en el espacio dimensional original en un problema de clasificación lineal simple en un espacio dimensional más grande (Betancourt, 2005).

3.1.6 Multilayer Perceptron

El algoritmo *Multilayer Perceptron* (MLP) está basado en el perceptrón simple propuesto por (Rosenblatt, 1958) que se caracteriza por ser una red monocapa es decir que solo cuenta con una capa de procesamiento que es la salida. La neurona de salida del perceptrón realiza la suma ponderada de las entradas, y

finalmente pasa el resultado a una función de activación (Caicedo B & López S, 2009). La estructura del perceptrón se observa en la Figura 3.5.

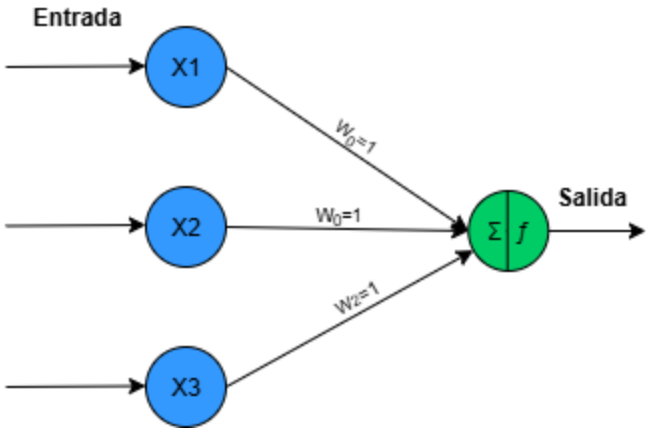

Figura 3.5. Arquitectura de un perceptrón simple

Sin embargo el perceptrón simple sólo es capaz de resolver problemas lineales de esta manera surge en la arquitectura *Multilayer perceptron*, compuesto por múltiples unidades llamadas perceptrones simples (Caicedo B & López S, 2009; Goodfellow et al., 2016). Su arquitectura está organizada en varias capas jerárquicas: una capa de entrada, una o más capas ocultas y una capa de salida. La estructura se observa en la Figura 3.6.

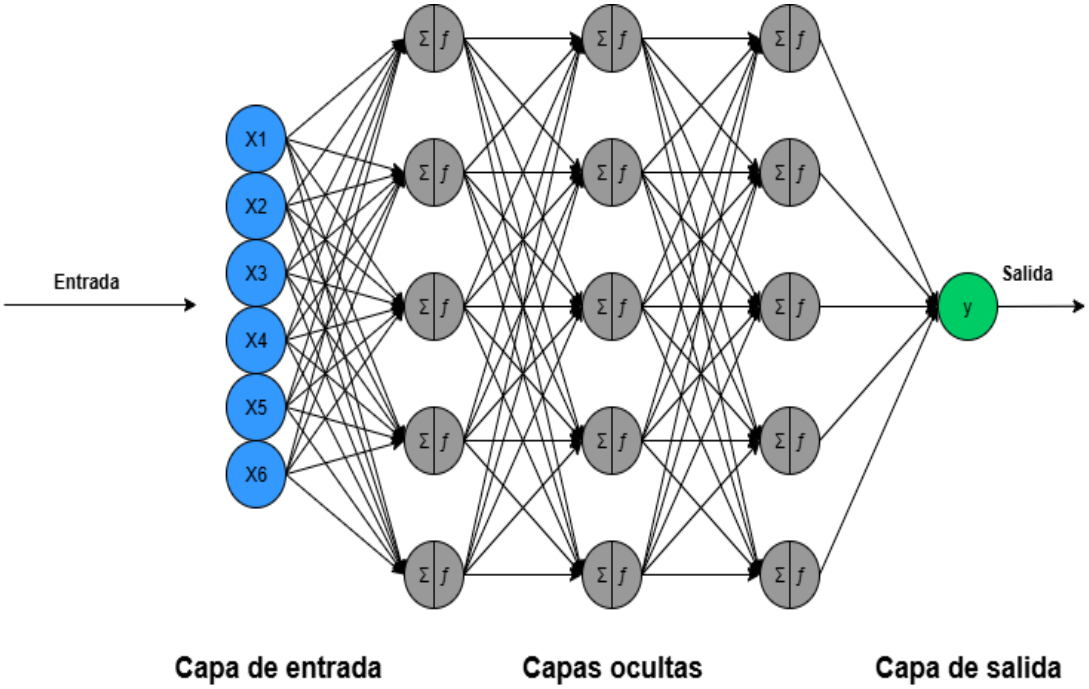

Figura 3.6. Arquitectura de un perceptrón multicapa

En la capa de entrada, los datos se reciben en forma de vector $x\epsilon R^d$, donde d representa el número de entidades en el conjunto de datos. Cada neurona de una capa realiza dos operaciones clave: ponderación y suma $(z = w^T\ x + b)$ y aplicación de una función de activación $(a = f(x))$. Donde w son los pesos asociados con las conexiones, b es un sesgo, y f es una función no lineal como el sigmoide, ReLU (Unidad Lineal Rectificada) o tanh, que introduce no linealidad en el modelo (Kim, 2016).

3.2 Técnicas de Preparación y Validación de Datos

Para el uso de algoritmos de aprendizaje automático es necesario la preparación de los datos con el principal objetivo de obtener una estructura adecuada de datos para ser utilizada en los algoritmos, ya que la calidad de los modelos está altamente relacionada con la calidad de los datos (Ndung'u, 2022). En este trabajo se explicarán tres técnicas utilizadas.

3.2.1 Normalización

La normalización es una técnica utilizada para la transformación de los datos aplicada cuando los datos se encuentran en diferentes escalas numéricas sin perder la información (Ndung'u, 2022)**.**

3.2.1.1 Z-Score

La normalización *Z-Score* consiste en transformar las unidades de medida en nuevos datos que tienen una media igual a 0 y una desviación estándar igual a 1.

Dada por la ecuación 3.3

$$Z = \frac{x - \mu}{\sigma} \tag{3.3}$$

Donde *Z-Score* es la puntuación estandarizada, x son los datos por analizar, μ es el promedio de la distribución de los datos y σ es la dispersión de los datos con respecto a la media (Kirch, 2008).

3.2.1.2 Percentil Rank

De acuerdo con (Beurs et al., 2022) el *Percentil Rank* indica el porcentaje de casos que se encuentran por debajo del valor analizado (x), lo que permite expresar la distribución en una escala de 0 a 1. Se encuentran dada por la siguiente ecuación 3.4.

$$PR = (trunk(\frac{rank(x)}{length(x)})) * 100 \quad (3.4)$$

Donde la función $trunk$ elimina la parte decimal de un número, dejando solo la parte entera (sin redondear), la función $rank$ asigna un rango a cada elemento de un vector, x representa los datos estandarizados y *length* calcula el tamaño de la muestra x.

3.2.2 Balanceo de datos

El balanceo de datos es fundamental en la etapa del preprocesamiento con el fin de evitar sesgos en los modelos de aprendizaje automático (Ndung'u, 2022).

3.2.2.1 Balanceo de Datos con SMOTE

SMOTE (Synthetic Minority Over-Sampling Technique) es una técnica de sobre-muestreo utilizada para balancear conjuntos de datos desbalanceados. SMOTE genera nuevos ejemplos sintéticos interpolando entre las instancias existentes de la clase minoritaria y sus vecinos más cercanos. Este enfoque permite aumentar la representación de la clase menos frecuente sin duplicar exactamente las observaciones existentes ni eliminar ejemplos de la clase mayoritaria, lo que ayuda a mejorar la capacidad de generalización del modelo (Chawla et al., 2002) como se puede observar en la Figura 3.7.

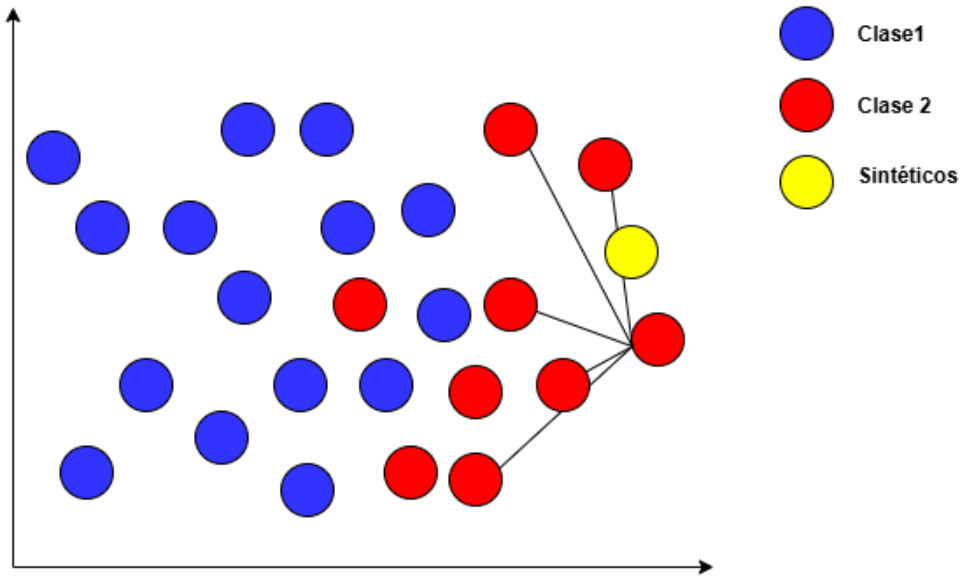

Figura 3.7 Ejemplo representativo de la técnica de SMOTE

Donde cada nueva instancia sintética se genera considerando una determinada cantidad de vecinos más cercanos de cada observación de la clase minoritaria, calculados mediante la distancia euclidiana.

3.2.3 Técnicas de Validación

Existen diferentes técnicas para validación para determinar el rendimiento de los algoritmos de acuerdo con sus predicciones. Para esto se divide el conjunto de datos para utilizar un subconjunto para entrenamiento y otro para pruebas. La selección de conjunto de prueba y entrenamiento es de mucha importancia para reducir el riesgo del sobre-entrenamiento (Cuevas et al., 2022). A continuación, se explica una de las técnicas más comunes.

3.2.3.1 Validación Cruzada (Cross-Validation)

La técnica de validación cruzada (del inglés Cross-Validation) está basada en dividir el conjunto de datos con el fin de evaluar el rendimiento de la predicción de los modelos (Yates et al., 2023).

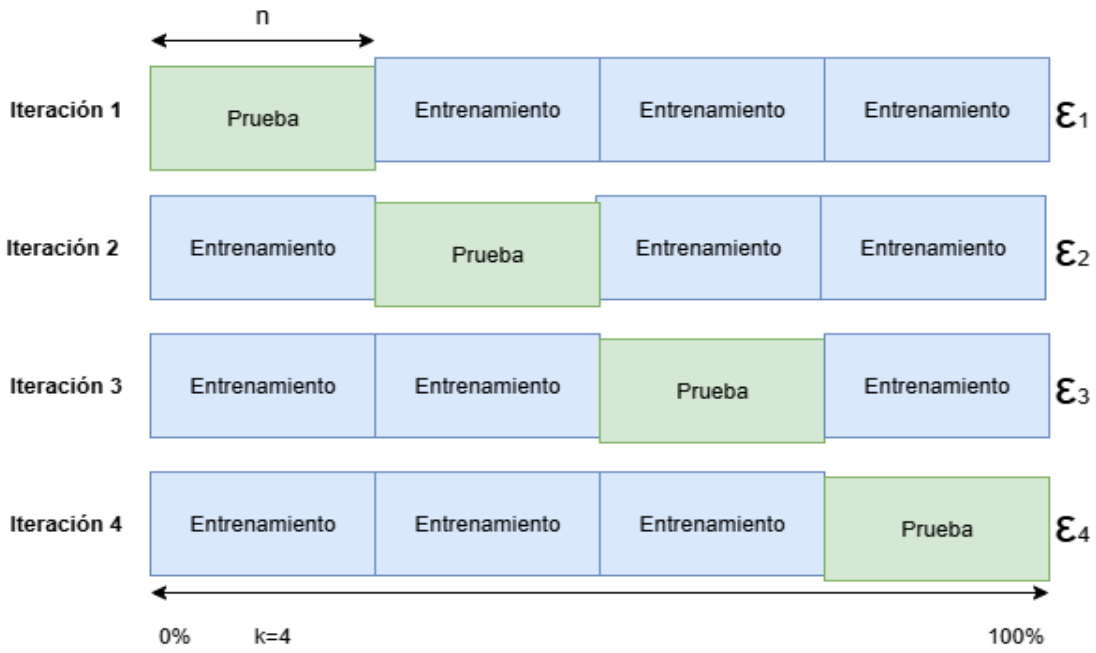

Figura 3.8. Ejemplo representativo de k-folds

De acuerdo con la Figura 3.8, *n* representa el número total de instancias el cual se divide en *k* = 4 subconjunto de datos del mismo tamaño, en cada iteración uno de cada conjunto de datos será tomado como prueba, mientras que los 3 subconjuntos resultantes serán para entrenamiento. El procedimiento se repite hasta que cada iteración se haya cumplido (Cuevas et al., 2022).

3.3 Métricas para la Evaluación del Desempeño de los Algoritmos

Las métricas de evaluación de algoritmos son un conjunto de indicadores estadísticos que nos ayudan a medir la eficacia y la eficiencia de los algoritmos (Duda et al., 2012). Los resultados de las métricas de evaluación permiten valorar si el modelo ha tenido un rendimiento óptimo o necesita refinamiento para la predicción del algoritmo en el contexto de clasificación, para esto existen diversas métricas que evalúan el rendimiento de los algoritmos (Naidu et al., 2023; Ponce, n.d.). A continuación, se explican las métricas de evaluación de algoritmos más comunes.

3.3.1 Matriz de Confusión

La matriz de confusión propuesta por Karl Pearson (Swaminathan & Tantri, 2024) es una tabla bidimensional que representa las instancias que fueron clasificadas positivas y negativas. Una de las dimensiones de la tabla muestra los valores predichos y la otra parte los valores reales (Lantz, 2013). La matriz de confusión es utilizada para encontrar la precisión del modelo de clasificación (Naidu et al., 2023). A continuación, se describen los elementos de la matriz de confusión, se representa en la Figura 3.9.

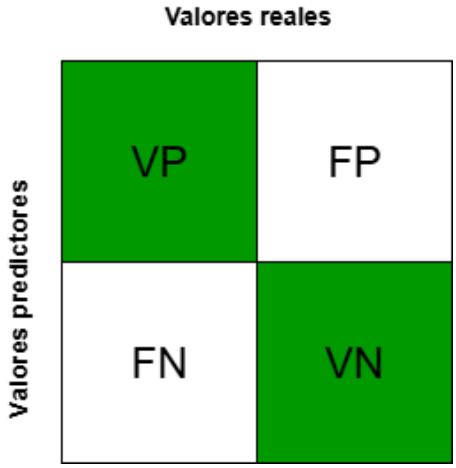

Figura 3.9. Matriz de confusión bidimensional

Verdaderos positivos (VP): Son los casos que realmente pertenecen a la clase de interés y que el modelo ha clasificado correctamente como tal.

Verdadero negativo (VN): Son los casos que no pertenecen a la clase de interés y que el modelo también ha identificado correctamente.

Falsos positivos (FP): Son los casos que no pertenecen a la clase de interés, pero que el modelo ha clasificado incorrectamente como si pertenecieran.

Falsos negativos (FN): Son los casos que sí pertenecen a la clase de interés, pero que el modelo ha clasificado incorrectamente como no pertenecientes.

3.3.2 Precisión

Las métricas de precisión se definen como la proporción de instancias positivas que realmente son positivas. Es decir, mide cuántas de las instancias que el modelo ha clasificado como positivas sean efectivamente positivas (Lantz, 2013). La precisión se calcula con la siguiente ecuación 3.5.

$$Pr = \frac{VP}{VP + FP} \tag{3.5}$$

Esta métrica es buena cuando el valor de Falsos positivos (FP) es mayor que los Falsos Negativos (FN). Un valor alto en esta métrica nos dice que tiene una tasa baja en FN lo que representa que da predicciones positivas precisas (Swaminathan & Tantri, 2024).

3.3.3 Exactitud

La métrica de exactitud calcula la frecuencia en la que las predicciones son correctas. Donde Exactitud = (número de instancias clasificadas correctamente) / (número total de instancias). Se calcula con la siguiente ecuación (3.6).

$$Acc = \frac{VP + VN}{VP + VN + FP + VN} \tag{3.6}$$

Se considera una métrica útil cuando tenemos clases equilibradas (Swaminathan & Tantri, 2024)

3.3.4 Recall

La métrica Recall se define como el número de instancias de verdaderos positivos sobre el número total de positivos (Lantz, 2013). Se calcula con la siguiente Ecuación (3.7).

$$R = \frac{VP}{VP + FN} \tag{3.7}$$

Un valor alto de esta métrica tiene indica que tiene una tasa baja en FN, lo que significa que captura eficazmente los casos positivos (Lantz, 2013).

3.3.5 F1-score

La métrica F1-score es una métrica derivada de la matriz de confusión que combina a dos métricas de evaluación precisión y *Recall*. Se utiliza para evaluar la precisión y recuperación de un algoritmo utilizando un único valor (Lantz, 2013). Se calcula con la siguiente ecuación (3.8).

$$F1 = 2\left(\frac{Pr \cdot R}{Pr + R}\right) \tag{3.8}$$

Es una métrica útil para escenarios donde los falsos positivos (FP) y falsos negativos (FN) son iguales. Los valores de F1-score oscilan entre 0 y 1 más cercano a uno este la puntuación, mejor será el clasificador (Swaminathan & Tantri, 2024).

3.3.6 Sensibilidad

La sensibilidad (SE) mide la proporción de instancias positivas que se clasificaron correctamente. Mide la capacidad del algoritmo para identificar correctamente los casos positivos (Lantz, 2013). Se calcula con la siguiente ecuación (3.9).

$$SE = \frac{VP}{VP + FN} \tag{3.9}$$

Es igual a Recall llamada tasa positiva total (TPR) (Swaminathan & Tantri, 2024).

3.3.7 Especificidad

La especificidad (SP) mide la proporción de instancias negativas que se clasificaron correctamente. La sensibilidad y la especificidad oscilan entre 0 y 1, siendo más deseables valores cercanos a 1 (Lantz, 2013). Se calcula con la siguiente Ecuación (3.10).

$$SP = \frac{VN}{VN + FP} \tag{3.10}$$

Esta métrica llamada también tasa total de negativos, es complementaria de la métrica de sensibilidad. Es útil cuando se requiere minimizar los falsos positivos (FP). Un valor alto en esta métrica sugiere que el modelo cuenta con una tasa de falsos positivos, lo que se entiende por identificar la mayoría de los casos negativos (Swaminathan & Tantri, 2024).

3.3.8 Geometric Mean

Una de las métricas recomendada para los clasificadores que cuentan con clases desequilibradas es la G-media se encarga de medir el rendimiento a partir de las medidas de sensibilidad y especificidad (He & Garcia, 2009; Sun et al., 2011). Se calcula con la siguiente Ecuación (3.11).

$$G - Mean = \sqrt{SE \cdot SP} \tag{3.11}$$

3.3.9 Kappa Statistic

Es una métrica utilizada para comparar el rendimiento de un clasificador con el de un clasificador que hace predicciones basadas únicamente en el azar. Los valores de Kappa oscilan entre un valor máximo de 1, lo que indica una concordancia perfecta entre las predicciones del modelo y los valores verdaderos, algo poco frecuente (Zacharski, 2015). En la Tabla 3.1 se pueden ver los valores de Kappa.

Tabla 3.1. Interpretación de valores de Kappa

Valor k	Descripción
< 0	Menos que el rendimiento por azar
0.01-0.20	Ligeramente bueno
0.21-0.40	Rendimiento justo
0.41-0.60	Rendimiento moderado
0.61-0.80	Desempeño sustancialmente bueno
0.81-1.00	Desempeño casi perfecto

Se calcula con la siguiente ecuación (3.12).

$$K = \frac{Pc - Pr}{1 - Pr} \tag{3.12}$$

Donde *Pc* es la proporción de aciertos observados, y *Pr* es el resultado de aciertos esperados al azar (Naidu et al., 2023).

Capítulo 4: Trabajo Relacionado

La predicción de defectos ha sido de gran relevancia en el área de investigación, con el principal objetivo de mejorar la calidad del software. Para esto se han presentado varias métricas orientadas a objetos para medir las propiedades estructurales del software orientado a objetos (Selvadurai et al., 2007).

La combinación de métricas orientadas a objetos con otras métricas de software, como las medidas de complejidad y centralidad de software de McCabe y Halstead (McCabe, 1976), por nombrar algunas, se han considerado para general modelos para predecir defectos de software.

Estudios como los de Ouellet y Badri (Ouellet & Badri, 2024) presentan la combinación de métricas orientadas a objetos y medidas de centralidad. Esto es con el fin de investigar si las medidas de centralidad mejoran la calidad desde la perspectiva de la predicción de la propensión a fallos, la gravedad de los fallos y el número de fallos. Para ello, clasificaron métricas orientadas a objetos en tres categorías: 1) Complejidad: Complejidad Ciclomática Media (*Average Cyclomatic Complexity* ACC), Respuesta para una Clase (*Response for a Class* RFC) y Métodos ponderados por clase (*Weighted Methods per Class* WMC); 2) Acoplamiento: acoplamiento aferente (*Afferent Coupling* CA), acoplamiento entre objetos (*Coupling Between Objects* CBO) y acoplamiento eferente (*Efferent Coupling* CE); y 3) Tamaño: Tamaño de la interfaz de clase (*Class Interface Size* CIS), líneas de código de origen (*Lines of Code* LOC) y número de métodos (*Number of Methods* NOM). Mientras que las medidas de centralidad utilizadas fueron la centralidad de intermediación (*Branch Count* BC), la centralidad de cercanía (*Class Length* CL), la densidad del componente de vecindad máxima (*Design Metric for Method Nesting Complexity* DMNC), la centralidad del vector propio (*Eigenvector Centrality* EV), la centralidad de Katz (*Katz Centrality* KATZ), la centralidad de apalancamiento (*Leverage Centrality* LE), la centralidad del vestíbulo (*Lobby Index* LO), la centralidad semi-local *(Semi-Local Centrality SLC)* y la centralidad de entropía de vértices (Vertex

Entropy Centrality VEC). Las metrcias se calcularon a partir de 20 versiones diferentes de cinco sistemas de software Java de código abierto. Las métricas se dividieron en tres subconjuntos: métricas orientadas a objetos, medidas de centralidad y la combinación de ambas. En sus experimentos se utilizaron un total de 9 algoritmos de aprendizaje automático; estos fueron: Regresión Logística Binaria (BLR), Árbol de Clasificación C5.0, KNN, MLP, Árbol de Partición y Regresión, RF, SVM, RL y Boosting con C5.0. En sus resultados, argumentan que los algoritmos basados en árboles de decisión (RF y C5.0) y técnicas como SVM tienen el mejor rendimiento en clasificación utilizando la combinación de ambas métricas para determinar clases propensas al fallo, obteniendo un valor de AUC superior a 0,90, un valor de G-mean superior al 85% y una medida F superior al 75%.

Por otro lado, Singh y Chhabra (M. Singh & Chhabra, 2024) proponen un modelo para la predicción de defectos de software entre proyectos utilizando algoritmos de aprendizaje automático y métricas de código estructural. Las métricas estructurales propuestas fueron las siguientes: 1) Complejidad total extendida de la clase (*Extended Total Class Complexity* ETCC), que se utiliza para medir la complejidad de la clase, 2) Acoplamiento extendido entre clases (*Extended Class Average Complexity* ECAC), que ayuda a evaluar el acoplamiento entre clases, 3) Falta extendida de cohesión de los métodos de clase (*Extended Lines of Code of Class Complexity* ELCOCC), que determina la coherencia de los métodos de clase, 4) Relación de acoplamiento de método heredado extendido (*Extended Inherited Method Coupling Ratio* EIMCR), que cuantifica la información heredada por la clase, 5) Relación de acoplamiento de método sobrecargado extendida (*Extended Overridden Method Coupling Ratio* EOMCR), que ayuda a medir la sobrecarga del método, y 6) Categoría de lógica (*Logic Category* COC). También propusieron la métrica de procedimiento del código de complejidad de ruta ponderada (Weighted Path Complexity WPC), para manejar las deficiencias de la complejidad ciclomática. Las métricas se calcularon en 18 proyectos escritos en lenguaje Java. Los algoritmos de aprendizaje automático utilizados fueron: DT, LR, SVM, KNN, RF, Impulso

Adaptativo (AdaBoost), Impulso de Gradiente Extremo (XGBoost) y Red Neuronal Artificial (ANN). Las categorías para la clasificación se asignaron como defectuosas o libres de defectos. El algoritmo de aprendizaje automático que logró el AUC promedio más alto en el estudio es XGBoost, con un AUC promedio de 0.76717.

El estudio de Canaparo et al. (Canaparo et al., 2022) tiene como objetivo construir modelos de predicción de defectos mediante la combinación de técnicas de selección de características y algoritmos de aprendizaje automático utilizando varias métricas de varios proyectos. Para ello, utilizaron 2 conjuntos de datos, el primero el conjunto de datos de predicción de defectos Eclipse con 5 proyectos de software codificados en Java que contienen amortiguador de Chi y métricas de Kemerer (CK) (Chidamber & Kemerer, 1994) como CBO, DIT, Falta de Cohesión en Métodos (LCOM), Número de Hijos (NOC), RFC, WMC, 11 métricas orientadas a objetos como número de líneas de código, número de métodos, número de atributos, entre otros, y una etiqueta binaria. El segundo conjunto de datos corresponde al repositorio *PROMISE* de la NASA relacionado con 12 proyectos codificados en *C* y *C++*. Las métricas que utilizaron fueron líneas de código, métricas calculadas a partir de la complejidad ciclomática de McCabe, métricas de complejidad basadas en el número de operandos y operadores, entre otras. Se utilizaron diferentes técnicas de selección de rasgos, como el Análisis de Componentes Principales (PCA), la Eliminación Hacia Atrás, la Selección Hacia Adelante y LASSO. Para la predicción de defectos se implementaron algoritmos de aprendizaje automático supervisado como NB, RF y aprendizaje automático no supervisado como *Neural Gas Clustering* (GNG) y *Spectral Clustering*. Los resultados mostraron que NB en combinación con *Backward Elimination* logró el mayor rendimiento en AUC con un promedio de 0.65, lo que indica que tiene una buena capacidad para identificar entre clases defectuosas y libres de defectos, mientras que *Spectral Clustering* logró un promedio de 0.66 en AUC utilizando todas las características del conjunto de datos y RF en combinación con *Backward* un promedio de 0.61.

Khleel y Nehéz (Khleel & Nehéz, 2021) presentan el entrenamiento de diferentes algoritmos de aprendizaje automático, como DT, NB, RF y LR. Los algoritmos se entrenaron con 4 conjuntos de datos del repositorio *PROMISE*. Los conjuntos de datos utilizados fueron JM1 y PC1, que contienen métricas de códigos escritos en *C*. Mientras que KC1 y KC2 contienen métricas de código en *C++*. Estos conjuntos de datos contienen métricas para medir la complejidad de un programa propuesto por Halstead, tales como: número total de operadores y operandos (*Total Number of Operators and Operands* N), Volumen (*Volume* V), Duración del programa (*Length* L), Medir la dificultad (*Difficulty* D), Medir la inteligencia (*Intelligence Content* I), Medir el esfuerzo (*Effort* E), Estimación del errores (*Estimated Bugs* B), Estimador de tiempo (*Time Estimator* T); y métricas de complejidad de McCabe, tales como: el LOC, la complejidad ciclomática (*Cyclomatic Complexity* v(g)), la complejidad esencial (*Essential Complexity* ev(g)) y la complejidad del diseño (*Design Complexity* iv(g)). Los resultados de sus experimentos mostraron que los clasificadores DT y RF obtuvieron los mejores resultados con una precisión del 99%, una precisión del 99% y un 100% de recuperación para los conjuntos de datos JM1 y KC1, mientras que KC2 mostró una precisión del 98% y del 98% de precisión.

En Mohamed (Mohamed et al., 2020) el objetivo es construir un modelo universal basado principalmente en la estructura de la *Red Neuronal Feedforward* (FFNN) para predecir defectos en una clasificación binaria en clases defectuosas y no defectuosas, utilizando diferentes conjuntos de datos. Estos se recogieron de 12 proyectos donde se analizaron 38 versiones de software orientado a objetos de código abierto codificado en *Java* desarrollado por la *Apache Software Foundation*. El conjunto de datos analizó las métricas de complejidad del software Chidamber y Kemerer, las métricas orientadas a objetos como WMC, DIT, NOC, CBO, RFC, LCOM y los métodos orientados a objetos de la complejidad cognitiva como la complejidad del método (*Method Complexity* MC), el peso del acoplamiento para la clase (*Coupling Weight for Class* CWC), Complejidad de Atributos (*Attribute Complexity* AC), Complejidad

de Clase (*Class Complexity* CLC), Complejidad de Código (*Code Complexity* CC) donde el objetivo es validar la asociación en la predicción de defectos. Se utilizaron técnicas de procesamiento, en el modelo universal 1 (UM1) se aplicó la técnica de normalización Min-Max, dejando los valores entre 0 y 1, en el modelo universal 2 (UM2) se utilizó la técnica de *decile ranking*, que consiste en mapear los valores del 1 al 10 y tomar estos como umbrales con el fin de unificar aproximaciones, y un modelo universal final (UM0) donde no se aplicó ningún procesamiento. Además, se presentaron varios experimentos de FFNN utilizando diferentes capas ocultas y neuronas, para la evaluación de los resultados utilizaron 4 métricas de rendimiento donde los valores van de 0 a 1 y los más altos indican un mejor rendimiento, UM1, UM2 y UM0 registran el mejor valor máximo de 0.69452 en *Recall* en las métricas CC, mientras que las métricas CK registran un mejor resultado en todas las métricas de rendimiento.

Otro estudio es el presentado por Deshpande et al. (Deshpande et al., 2020) donde propusieron un modelo para evaluar la fiabilidad del software a través de la predicción temprana de defectos utilizando algoritmos de aprendizaje automático. Las métricas utilizadas fueron las de Chidamber y Kemerer (6): WMC, DIT, NOC, CBO, RFC y LCOM; Métricas de Martins (2): CA, CE; Métricas QMOOD (5): Número de Métodos públicos (*Number of Public Methods* NPM), Métrica de Acceso a Datos (*Data Access Metric* DAM), Medida de Agregación (*Measure of Aggregation* MOA), Medida de Abstracción Funcional (*Measure of Functional Abstraction* MFA), Cohesión entre Métodos (*Cohesion Among Methods* CAM); métricas de Chidamber y Kemerer extendidas (4): Falta de cohesión en el método 3 (*Cohesion Among Methods* LCOM3), Acoplamiento de herencia (*Inheritance Coupling* IC), Número total de métodos nuevos o redefinidos a los cuales están acoplados todos los métodos heredados (*Coupling Between Methods* CBM), Método promedio por clase (*Average Method Complexity* AMC); métricas de tamaño (1): LOC, MAX CC, AVG CC. Las métricas se calcularon sobre un total de 10 proyectos de *software*, y todos estos proyectos se desarrollaron en el lenguaje de programación *Java*. Los algoritmos de aprendizaje automático utilizados fueron LR y DT.

El algoritmo que obtuvo los mejores resultados fue LR, con una precisión del 75%. En comparación, el DT mostró una precisión del 74,5%.

En la Tabla 4.1 se resumen las métricas de software utilizadas para predecir defectos en cada uno de los estudios previos, las métricas se clasificaron según su propósito y enfoque orientado a objetos y tradicional.
En el enfoque orientado a objetos, las métricas se identificaron en relación con: La complejidad y el tamaño (WMC, CC, LOC, AMC, CIS) (Canaparo et al., 2022; Deshpande et al., 2020; Khleel & Nehéz, 2021; Ouellet & Badri, 2024); la herencia (DIT, NOC, MFA, IC) (Canaparo et al., 2022; Deshpande et al., 2020; Khleel & Nehéz, 2021); la cohesión (LCOM, LCOM4, CAM); el acoplamiento (CBO, CA, CE, CBM, RFC) (Canaparo et al., 2022; Deshpande et al., 2020; Khleel & Nehéz, 2021; Ouellet & Badri, 2024); encapsulación y abstracción (DAM) (Deshpande et al., 2020), visibilidad (NPM) (Deshpande et al., 2020); métricas de complejidad cognitiva (MC, CWC, AC, CLC) (Khleel & Nehéz, 2021); así como la propuesta de nuevas métricas estructurales orientadas a objetos que buscan mejorar las limitaciones de las actuales (NOM, ETCC, ECAC, ELCOCC, EIMCR, EOMCR, COC, WPC) (Singh & Chhabra, 2024). Por otro lado, en el enfoque tradicional, se utilizaron las métricas de complejidad de Halsted (N, V, L, D, I, E, B, T) y McCabe (EV G, MAX CC, AVG CC,V G, IV G) (Deshpande et al., 2020; Mohamed et al., 2020)

Las métricas más utilizadas son WMC y CBO (Deshpande et al., 2020; Canaparo et al., 2022; Ouellet & Badri, 2024; Khleel & Nehéz, 2021), junto con LOC, DIT, NOC, IC y RFC. En estos estudios, se ha propuesto el uso de métricas para generar modelos que ayuden a predecir si una clase contiene defectos o incluso a indicar el grado de gravedad del defecto. Sin embargo, como se puede observar en la Tabla 4.1, solo (Deshpande et al., 2020) ha propuesto la combinación de métricas orientadas a objetos con otros tipos de métricas. Por lo tanto, nuestro trabajo, a diferencia de estudios anteriores, propone la combinación de varias métricas orientadas a objetos con métricas de complejidad de software para predecir defectos, así como la aplicación de

varios algoritmos para generar modelos que tengan como objetivo detectar defectos de software.

Tabla 4.1. Comparación de estudios que utilizan métricas de software

Autores		(Ouellet & Badri, 2024)	(Khleel & Nehéz, 2021)	(M. Singh & Chhabra, 2024)	Deshpande et al. (2020)	Mohamed (2020)	Canaparo (2022)	Esta tesis
Métricas de complejidad y tamaño	WMC	x			x	x	x	x
	CC					x		
	LOC	x	x		x			x
	AMC				x			x
	EV(G)		x					
	ACC	x						
	CIS	x						
	MAX (CC)				x			x
	AVG(CC)				x			x
Métricas de herencia	DIT				x	x	x	x
	NOC				x	x	x	x
	MFA				x			x
	IC				x			x
Métricas de Cohesión	LCOM				x	x	x	x
	LCOM3				x			x
	CAM				x			x
Métrica de acoplamiento	CBO	x			x	x	x	x
	CA	x			x			x
	CE	x			x			x
	CBM				x			x
	RFC	x			x	x	x	x
Métrica de encapsulación y Abstracción	DAM				x			x
	MOA				x			x
Métrica de visibilidad	NPM				x			x
Métricas de complejidad de Halstead	N		x					
	V		x					
	L		x					

	D		**x**					
	I		**x**					
	E		**x**					
	B		**x**					
	T		**x**					
Métricas de complejidad McCabe	V(G)		**x**					
	IV(G)		**x**					
Métricas estructurales	NOM	**x**						
	ETCC			**x**				
	ECAC			**x**				
	ELCOCC			**x**				
	EIMCR			**x**				
	EOMCR			**x**				
	COC			**x**				
	WPC			**x**				
Métricas de complejidad cognitiva	MC					**x**		
	CWC					**x**		
	AC					**x**		
	CLC					**x**		

Capítulo 5: Diseño Experimental

En este capítulo se describe el diseño experimental que se propuso para llevar a cabo los experimentos de la investigación con el objetivo de predecir defectos mediante técnicas de aprendizaje automático. La estructura experimental se desarrolló con base a las fases de la metodología *CRISP-DM*.

5.1 Conjunto de Datos

El conjunto de datos utilizado en esta investigación es MJ12A (Madeyski & Jureczko, 2015). Está disponible de forma gratuita en línea en http://purl.org/MarianJureczko/MetricsRepo. El conjunto de datos MJ12A se compone de tres conjuntos de métricas: 1) Las métricas utilizadas para medir la comprensión, el mantenimiento y la reutilización del código, propuestas por Chidamber y Kemerer (Kanth et al., 2024); 2) Las métricas del Modelo de Calidad para el Diseño Orientado a Objetos (QMOOD) propuesto por (Bansiya & Davis, 2002) https://www.zotero.org/google-docs/?8W6m5i , que se utilizan para medir y evaluar aspectos de la calidad del software y la satisfacción del usuario; y finalmente 3) las métricas de McCabe, que se utilizan para medir la complejidad de un programa (McCabe, 1976). Los valores de las métricas provienen de 12 proyectos de software escritos en *Java*.

Los cálculos de las métricas se realizaron utilizando dos programas informáticos: *BugInfo*, una herramienta que proporciona información sobre defectos y problemas comunes en el software, lo que ayuda a desarrolladores y usuarios a identificar y resolver defectos eficazmente que se encuentra disponible en https://github.com/search?q=BugInfo&type=repositories y *Ckjm*, un programa que calcula las métricas orientadas a objetos de Chidamber y Kemerer procesando el código de *bytes* de archivos *Java* compilados. Es de código abierto y está disponible en: https://github.com/dspinellis/ckjm (Deshpande et al., 2020). La lista y la descripción de las métricas se ilustran en las Tablas 5.1 - 5.6.

Tabla 5.1. Métricas de complejidad y tamaño

Nombre de métricas	Descripción
Métodos Ponderados por Clase (WMC)	El valor de WMC es igual al número de métodos en la clase (suponiendo pesos unitarios para todos los métodos).
Líneas de Código (LOC)	Basado en código binario de Java. Es la suma del número de campos, métodos y sentencias en cada método de la clase.
Complejidad del Método Promedio (AMC)	Calcula el tamaño promedio de los métodos en una clase, medido en número de códigos binarios de Java en el método.
Complejidad Ciclomática de McCabe (CC)	Número de rutas diferentes en un método más uno
Avg (CC)	Media aritmética de CC en los métodos de la clase
Max (CC)	El valor CC más alto entre los métodos de la clase

Tabla 5.2. Métricas de herencia

Nombre de métricas	Descripción
Profundidad del Árbol de Herencia (DIT)	Niveles desde la raíz hasta la clase
Número de Hijos (NOC)	Mide el número de descendientes inmediatos de la clase.
Medida de Abstracción Funcional (MFA)	Es la relación entre el número de métodos heredados por una clase y el número total de métodos accesibles.
Acoplamiento de Herencia (IC)	Mide el número de clases principales a las que está acoplada una clase. Una clase está acoplada a su principal y no se ajusta a los métodos heredados. Depende funcionalmente de los métodos redefinidos a los que están acoplados todos los métodos heredados.

Tabla 5.3. Métricas de cohesión

Nombre de métricas	Descripción
Falta de Cohesión en los Métodos (LCOM)	Cuenta los pares de métodos en una clase que no comparten acceso a al menos un campo común. Esto se calcula restando el número de pares que comparten acceso de los que no lo comparten.
Falta de Cohesión en los Métodos (LCOM3)	Variante que calcula la cohesión en los métodos
Cohesión de Métodos de Clase (CAM)	Calcular la relación entre los métodos de una clase según la lista de parámetros

Tabla 5.4. Métricas de acoplamiento

Nombre de métricas	Descripción
Acoplamiento entre Clases de Objetos (CBO)	Representa el número de clases acopladas a una clase específica (incluye los acoplamientos diferentes). Estos acoplamientos se producen mediante llamadas a métodos, acceso a campos, herencia, argumentos de métodos, tipos de retorno y excepciones.
Acoplamiento Aferente (Ca)	Mide el número de clases que dependen de la clase medida
Acoplamiento Eferente (Ce)	Mide el número de clases de las que depende la clase medida
Acoplamiento entre Métodos (CBM)	Acoplamiento entre métodos
Respuesta para una Clase (RFC)	Mide la cantidad de métodos diferentes que se pueden ejecutar cuando un objeto de esa clase recibe un mensaje. Incluye los métodos internos y los llamados dentro del cuerpo de los métodos de la clase.

Tabla 5.5. Métricas de encapsulación y abstracción

Nombre de métricas	Descripción
Métricas de Acceso a Datos (DAM)	Es la relación entre el número de atributos privados y protegidos y el número total de atributos declarados en la clase.
Medida de Agregación (MOA)	Mide el grado de relación parte-todo en una clase según la cantidad de campos de clase que son de tipos definidos por el usuario.

Tabla 5.6. Métricas de visibilidad

Nombre de métricas	Descripción
Número de Métodos Públicos (NPM)	Cuenta todos los métodos de una clase que se declaran como públicos

Estos fueron recopilados a partir de la publicación de Jureczko y Spinellis (2010) sobre el conjunto de datos MJ12A y se clasificaron según su tipo en: métricas de complejidad y tamaño, métricas de herencia, métricas de cohesión, métricas de acoplamiento, métricas de encapsulación y abstracción, métricas de visibilidad. El conjunto de datos contiene 98,148 registros con 29 atributos, donde 20 corresponden a métricas de software de complejidad y orientadas a objetos, que pertenecen a varias categorías. El restante de los atributos es llamado bugs, que especifican el número de defectos identificados en el software; se ha transformado en un atributo de tipo de clase. Si el número de errores es igual a cero, se le asigna la clase "Sin Defectos", mientras que, si el número de errores es mayor o igual a 1, se le asigna la clase "defectuoso". Otros atributos contenidos en el conjunto de datos son el nombre del proyecto, la versión y las clases Java, que no se consideraron para realizar los experimentos. El conjunto de datos utilizado en nuestro experimento consta de 20 métricas de software y la clase (Con defectos y Sin defectos). El conjunto de datos se distribuye de la siguiente manera: 79,849 registros correspondientes a la clase "Sin defectos" (81.35%) y 14,299 a la clase "Con Defectos" (18.64%).

5.2 Preparación de datos

5.2.1 Normalización

Con el objetivo de homogeneizar los datos y evitar sesgos en los algoritmos de aprendizaje automático, se utilizó la normalización $Z - Score$, permitiendo aplicar la misma escala a los atributos. La operación se llevó a cabo en *Rstudio* con la función *scale*. Así mismo, se aplicó *percentile rank* con el objetivo de estandarizar la escala. En la figura 5.1 se visualiza la representación normalización del conjunto de datos.

Conjunto de datos **MJ12A**

loc	dam	moa	mfa	cam	ic
404	1.0000000	1	0.7400000	0.5952381	2
125	0.0000000	0	0.0000000	0.4444444	0
271	1.0000000	0	0.8409091	0.4062500	1
68	1.0000000	1	0.6250000	0.5000000	1
527	0.0000000	2	0.0000000	0.3796296	0
300	0.8333333	0	0.8604651	0.3571429	1
4	0.0000000	0	0.0000000	0.5000000	0
42	0.0000000	0	0.0000000	0.5000000	0
26	1.0000000	1	0.0000000	0.5555556	0
0	0.0000000	0	0.0000000	0.0000000	0
437	0.7500000	1	0.4545455	0.3111111	1

Conjunto de datos **MJ12A NORMALIZADO**

loc	dam	moa	mfa	cam	ic
0.904782895	0.9117347	0.8998439	0.4332646	0.49345180	0.8316480
0.679281557	0.3244944	0.4231529	0.1722554	0.29033012	0.2217094
0.849571951	0.9117347	0.4231529	0.5271381	0.25524706	0.5974636
0.485326295	0.9117347	0.8998439	0.3976824	0.35315673	0.5974636
0.931942261	0.3244944	0.9656551	0.1722554	0.22342482	0.2217094
0.865190976	0.8087692	0.4231529	0.6407465	0.20026448	0.5974636
0.044286655	0.3244944	0.4231529	0.1722554	0.35315673	0.2217094
0.321589412	0.3244944	0.4231529	0.1722554	0.35315673	0.2217094
0.209993839	0.9117347	0.8998439	0.1722554	0.43267515	0.2217094
0.002453584	0.3244944	0.4231529	0.1722554	0.02441900	0.2217094
0.913635977	0.7944566	0.8998439	0.3637411	0.15091664	0.5974636

Figura 5.1. Representación de normalización en conjunto de datos MJ12A

5.2.2 Balanceo de Datos

Mientras que el conjunto de datos MJ12A no está distribuido uniformemente, ya que se distribuye de la siguiente manera: 14,299 registros que pertenecen a la clase "defectuosa" y 79,849 a la clase "libre de defectos". Era necesario realizar un balance de datos, es decir, equilibrar la clase "defectuosa", penalizando la clase "libre de defectos" durante el entrenamiento. El equilibrio de clases se llevó a cabo con la función SMOTE *(Synthetic Minority Over-Sampling Technique)* en lenguaje R.

La configuración utilizada por SMOTE en nuestro experimento fue la siguiente. El número de vecinos utilizados para la generación de nuevas instancias fue K = 5. Se utilizó la métrica de distancia Euclidiana para calcular los vecinos más cercanos. El conjunto de datos se equilibró aumentando la clase minoritaria en un 450%, por lo tanto, el parámetro de tamaño de duplicado fue 4.5.

5.3 Algoritmos de Aprendizaje Automático

Se emplearon diversos algoritmos de aprendizaje automático para construir modelos con el objetivo de predecir defectos de software, utilizando métricas de complejidad de software y orientadas a objetos. Los algoritmos se implementaron en *RStudio* 2024.04.2 con la versión 4.4.1 del lenguaje R. Se llevó a cabo en una computadora de marca *ASUS*, modelo *ASUS TUF Gaming* F15FX506LHFX506LH, con *Intel(R) Core (TM)i5 − 10300H CPU*@2.50GHz2.50GHz, 32.0 *GB RAM.* En la tabla 5.7 se presentan los parámetros utilizados para cada modelo y las librerías utilizadas en R, con el propósito de permitir la replicación de los modelos. Cada configuración de hiperparámetros se ejecutó 10 veces de manera independiente para obtener muestras, utilizando la métrica exactitud, estos resultados fueron necesarios para llevas acabo las pruebas estadísticas del Capítulo 7.

Tabla 5.7. Configuraciones de modelos

Algoritmo	Parámetros	Librerías
K-Nearest Neighbors	Número de vecinos K: 3,5,7,9,11 Distancias: *Euclidiana, Manhattan, minkowski*	*Class*
Naïve Bayes	Variante gaussiana	*E1071*
Random Forest	Número de árboles: 100, 200 y 500 Profundidad (mtry): 3	*Randomforest*
Support-Vector Machines	Kernel: *radial, linear, polynomial* Costo: 1	*e1071*
Multilayer Perceptron	Capas ocultas: 2 Capa oculta 1: 128 neuronas Capa oculta 2: 64 neuronas Función de activación: RELU Tasa de abandono: 0.3 Lote: 32 Tasa de aprendizaje: 0.001 Momento: 0.2 optimizador: Adam	*Neuralnet*

5.4 Evaluación del modelo

Se utilizaron métricas para evaluar el rendimiento de los algoritmos de aprendizaje automático las cuales de describieron en la sección 3.3. El conjunto de datos se dividió en 80% para entrenamiento y 20% para pruebas. Se aplicó la técnica de validación cruzada con un k = 10, que consiste en dividir el conjunto de datos en 10 partes iguales. Se utilizan nueve partes para entrenar el modelo y la décima para la validación, alternando esta última en cada iteración hasta que se cubran todas las partes (Cuevas et al., 2022). Por otro lado, se aplicaron pruebas estadísticas para esto se ejecutaron 10 veces los modelos con el objetivo de validar la fiabilidad de los resultados.

5.5 Configuración experimental

Se llevaron a cabo tres experimentos para predecir defectos de software:

1. Los algoritmos se entrenaron con métricas de complejidad de software propuestas por McCabe (McCabe, 1976) y Chidamber y Kemerer (Chidamber & Kemerer, 1994). El conjunto de métricas resultante fue: WMC, DIT, NOC, CBO, RFC, LCOM, MAX(CC), AVG(CC);

2. Los algoritmos fueron entrenados con las métricas del modelo de calidad para el diseño orientado a objetos (QMOOD) propuesto por Bansiya y Davis [28]. Estas métricas fueron: CA, CE, NPM, LOC, DAM, MOA, MFA, CAM, IC, CBM, AMC, por último;

3. Los algoritmos se entrenaron con una combinación de complejidad de *software* y métricas orientadas a objetos, con un total de 20 métricas. La lista final fue: WMC, DIT, NOC, CBO, RFC, LCOM, LCOM3, MAX. CC, AVG, CC, CA, CE, NPM, LOC, DAM, MOA, MFA, CAM, IC, CBM, AMC.

Capítulo 6: Resultados

En esta sección se presentan los mejores resultados de rendimiento de cada algoritmo utilizado en los experimentos realizados y descritos en la sección 5.5.

6.1 Resultados de los Algoritmos de Aprendizaje Automático Implementando Métricas de Complejidad

En la Tabla 6.1 se presenta el rendimiento de los algoritmos de aprendizaje automático utilizando sus mejores configuraciones de hiperparámetros: KNN (K=3, distancia *Euclidiana*); NB (modelo base); RF (200 árboles); SVM (*Kernel Radial*); MLP (función de activación *RELU*), entrenados con métricas de complejidad.

Tabla 6.1. Rendimiento de algoritmos mediante métricas de complejidad

Algoritmos	Acc.	Pr.	R	F1	SE	SP	k	G-Mean
KNN	**83.55%**	**80.76%**	**85.57%**	**83.09%**	**75.05%**	**87.00%**	**67.10%**	**80.13%**
NB	62.11%	60.96%	55.05%	57.85%	68.44%	55.05%	23.61%	61.38%
RF	75.60%	75.30%	71.90%	73.60%	71.90%	21.10%	59.40%	71.90%
SVM.	72.67%	72.60%	72.70%	72.60%	72.70%	27.70%	45.06%	72.70%
MLP	72.70%	72.80%	72.70%	72.60%	72.70%	28.00%	44.95%	72.70%

Nota: Acc. =*Accuracy*; Pr.=*Precision*; R.=*Recall*; F1=F1-*Score*; SE=*Sensibilidad*; SP=*Especificidad*; k=*Kappa*; G-Mean=*Geometric Mean*.

El algoritmo con el mejor rendimiento es KNN, teniendo un rendimiento con una precisión de 80.76%, una sensibilidad de 85.57%, una puntuación F1 de 83.09% y una media G de 80.13%. El algoritmo con el rendimiento más bajo fue Naïve Bayes con una precisión de 60.96%, una sensibilidad de 55.05% y una media G de 61.38%.

6.2 Resultados de los Algoritmos de Aprendizaje Automático Implementando Métricas Orientadas a Objetos

En la Tabla 6.2 se presenta el rendimiento de los algoritmos de aprendizaje automático utilizando sus mejores configuraciones de hiperparámetros: KNN (K=3, distancia *Euclidiana*); NB (modelo base); RF (200 árboles); SVM (*Kernel Radial*); MLP (función de activación *RELU*), entrenados con métricas orientadas a objetos.

Tabla 6.2. Rendimiento de algoritmos que utilizan métricas orientadas a objetos

Algoritmos	Acc.	Pr.	R	F1	SE	SP	k	G-Mean
KNN	**83.62%**	**80.26%**	**86.63%**	**83.32%**	**73.77%**	**88.42%**	**61.62%**	**79.94%**
NB	62.28%	62.46%	52.16%	56.84%	71.93%	52.16%	24.30%	61.25%
RF	80.95%	81.64%	76.98%	79.24%	84.51%	7.698%	61.67%	80.65%
SVM	67.50%	67.60%	67.22%	67.22%	67.50%	33.30%	34.41%	68.35%
MLP	69.87%	69.90%	69.90%	69.70%	69.90%	30.80%	39.24%	69.90%

Muestra que el algoritmo KNN obtuvo el mejor rendimiento con una exactitud de 83.62%, una precisión de 80.26%, una puntuación F1 de 83.32%, un Recall de 86.63% y una media G de 79.94%. Estos resultados se obtuvieron utilizando KNN con una K igual a 3 quien obtuvo el mejor resultado y la distancia Euclidiana. Además, el algoritmo con el rendimiento más bajo fue NB con una exactitud de 62.28% y una sensibilidad de 52.16% para identificar la clase defectuosa.

6.3 Resultados de los Algoritmos de Aprendizaje Automático Implementando Métricas de Complejidad y Orientadas a Objetos

En la Tabla 6.3 se presenta el rendimiento de los algoritmos de aprendizaje automático utilizando sus mejores configuraciones de hiperparámetros: KNN (K=3, distancia *Euclidiana*); NB (modelo base); RF (100 árboles); SVM (*Kernel Radial*); MLP (función de activación *RELU*), entrenados con métricas de complejidad y orientadas a objetos.

Tabla 6.3. Rendimiento mediante métricas de complejidad de software y orientadas a objetos

Algoritmos	Acc.	Pr.	R	F1	SE	SP	k	G-Mean
KNN	83.71%	80.09%	87.18%	83.49%	74.12%	88.75%	61.62%	80.38%
NB	63.07%	62.00%	56.40%	59.06%	69.05%	56.40%	25.57%	62.40%
RF	**87.23%**	**85.52%**	**87.85%**	**86.67%**	**86.68%**	**87.86%**	**74.43%**	**87.26%**
SVM.	75.28%	75.30%	75.30%	75.20%	75.30%	25.20%	52.25%	75.30%
MLP	72.75%	72.80%	72.80%	72.70%	72.80%	27.70%	47.15%	72.80%

El algoritmo con el mejor rendimiento es RF con una exactitud de 87.23%, una precisión de 85.52%, una *Recall* de 87.85%, una puntuación F1 de 86.67% y una media G de 87.26%. El algoritmo con el rendimiento más bajo fue NB con una precisión de 62.00%, un *Recall* de 56.40% y una *G-Media* de 62.40%.

Capítulo 7: Pruebas Estadísticas

En este capítulo se describen y se presentan las pruebas estadísticas realizadas para comparar el desempeño de los modelos implementados, tomando como referencias los valores obtenidos en los experimentos que se encuentran en los anexos A1, A2 y A3.

Se aplicó la prueba estadística *t-student* con el objetivo de analizar el desempeño obtenido de los modelos utilizados, así determinar qué modelo brinda mejor desempeño en la clasificación de detección de defectos, la prueba *t-student* es aplicada para muestras pequeñas de tamaño *n* menor o igual a 30 datos (Mendenhall et al., 2009), por lo que la muestra se obtuvo de los resultados generados por los modelos aplicados partiendo de la técnica de validación cruzada *(cross-validation)* con una *k-fold = 10* lo que conlleva obtener una muestra pequeña de 10 resultados.

El valor estadístico de la prueba *t-student* se puede calcular mediante la siguiente expresión:

$$t = \frac{(\bar{x_1} - \bar{x_2}) - (\mu_1 - \mu_2)}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}} \tag{7.1}$$

Dónde $\bar{x_1}$, $\bar{x_2}$ representan las medias muestrales, μ_1 , μ_2 representan las medias poblacionales; S_1^2, S_2^2 representan las varianzas muestrales y n_1 , n_2 representan los tamaños de las muestras de cada modelo.

Es importante identificar los planteamientos de las dos hipótesis para demostrar si existe evidencia significativa o no. La primera es la **hipótesis nula** $(\boldsymbol{H_0})$, es una declaración sobre el valor de un parámetro de la población; la segunda es la **hipótesis alternativa** $(\boldsymbol{H_1})$, es una declaración que se acepta si los datos de

la muestra proporcionan evidencia de que la hipótesis nula es falsa (Mendenhall et al., 2009).

En nuestra investigación queremos demostrar que existe diferencia en los desempeños entre los modelos aplicados para clasificar la detección de defectos. En ese sentido, para su demostración se utilizó un nivel de significancia de $\alpha = 0.05$, así demostrar si existe evidencia significativa en los desempeños obtenidos de los modelos implementados.

Para ello se plantearon las siguientes hipótesis:

$$H_0 : \mu_1 = \mu_2 \tag{7.2}$$

$$H_1 : \mu_1 \neq \mu_2 \tag{7.3}$$

Donde μ_1 y μ_2 representan las medias poblacionales de los modelos implementados.

7.1 Pruebas Estadísticas en Métricas de Complejidad

En esta sección se presentan las pruebas estadísticas de los experimentos aplicadas a las Métricas de Complejidad (MC).

7.1.1 Prueba Estadística Aplicada al Desempeño de KNN

De los resultados obtenidos del modelo KNN en sus diferentes escenarios de configuración paramétricas para calcular las distancias: *Euclidean, Manhattan, y Minkowski*. Es posible concluir que el desempeño obtenido de *KNN-Euclidean* es diferente a *KNN-Manhattan* y *KNN-Minkowski* como se puede observar en el siguiente planteamiento de hipótesis:

H_0: No existen diferencias significativas entre el desempeño *KNN-Euclidean* (μ_1) y *KNN-Manhattan* o *KNN-Minkowski* (μ_2), según la expresión (7.2).

H_1: Existen diferencias significativas entre el desempeño de *KNN-Euclidean* (μ_1) y *KNN-Manhattan* o *KNN-Minkowski* (μ_2), según la expresión (7.3).

En la Tabla 7.1 Resume los resultados de las pruebas *t-student* aplicadas al modelo de *KNN-Euclidean, KNN-Manhattan y KNN-Minkowski*.

Tabla 7.1. Resultados de las pruebas t-student del modelo KNN - {Euclidean, Manhattan, Minkowski} para las MC

Comparación	Estadístico	p-valor (<0.05)	Evidencia
KNN-Euclidean vs KNN-Manhattan	4.41	0.0001697	Significativa
KNN-Euclidean vs KNN-Minkowski	-1	0.3434	No significativa

Como se puede observar la comparación entre el modelo *KNN-Euclidean* vs *KNN-Manhattan* se demuestra que sí existe diferencia significativa en el desempeño entre ambos, por lo que p-valor es inferior a 0.05. Por el contrario, al comparar las distancias *KNN-Euclidean* vs *KNN-Mink*owski no existen diferencias significativas entre el desempeño entre los modelos, por lo que el p-valor es superior a 0.05.

En la Figura 7.1 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño de *KNN-Euclidean* vs *KNN-Manhattan*.

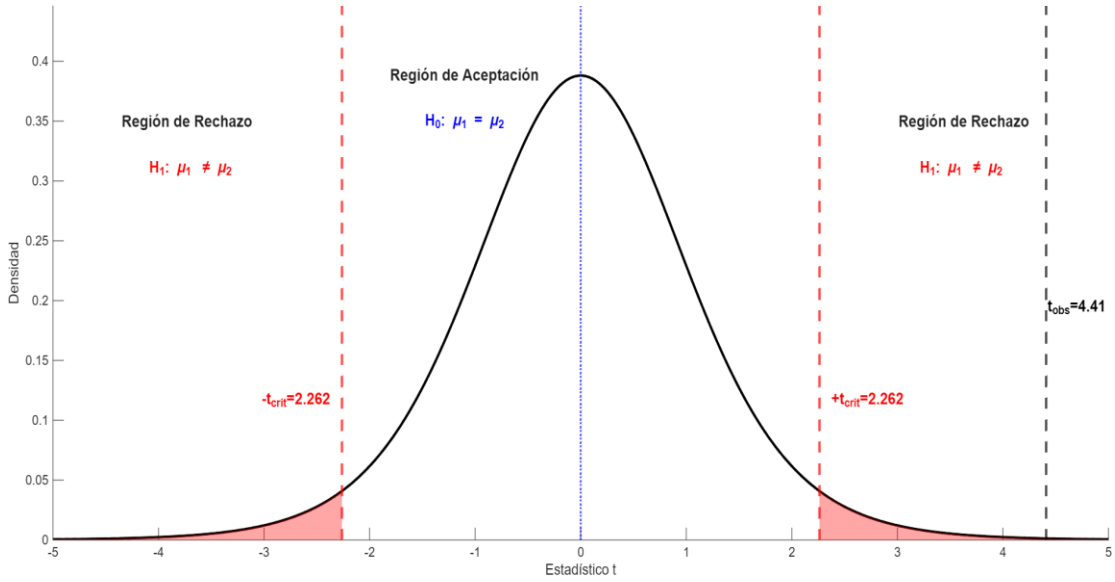

Figura 7.1. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre KNN-Euclidean vs KNN-Manhattan para las MC.

El valor estadístico obtenido en la prueba fue *t* = 4.41 (Ecuación. 7.1), el cual se encuentra en la región de rechazó; por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño de *KNN-Euclidean* vs *KNN-Manhattan* para la detección de defectos de software.

En la Figura 7.2 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia significativa entre el desempeño de *KNN-Euclidean* y *KNN-Minkowski*.

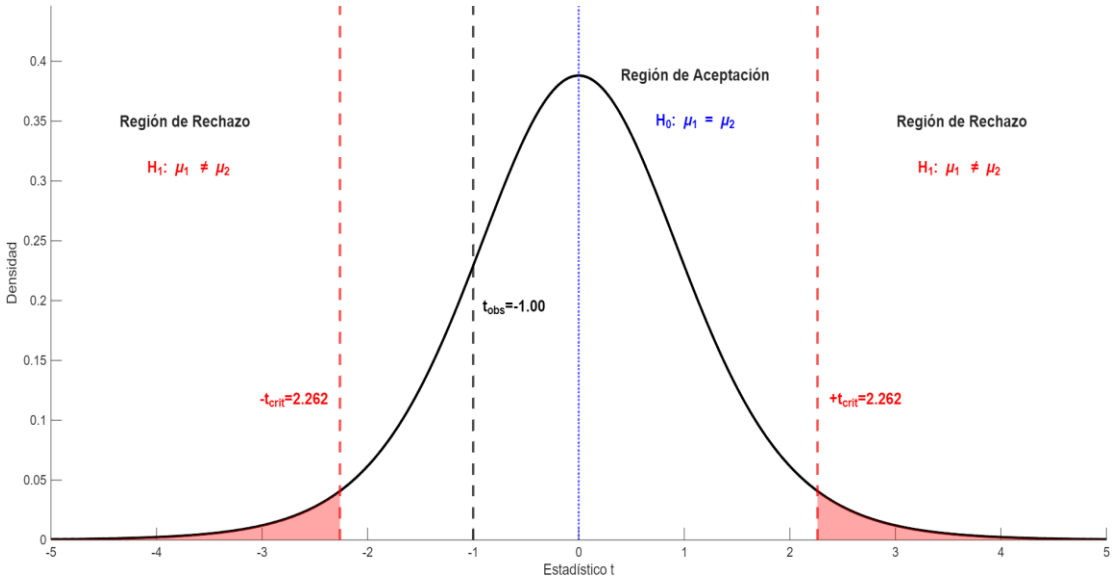

Figura 7.2. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre KNN-Euclidean vs KNN-Minkowski para las MC.

Donde el valor estadístico fue *t* = -1 (Ecuación. 7.1), el cual se encuentra dentro de la región de aceptación. Por lo tanto, se concluye que no existen diferencias significativas entre el desempeño de *KNN-Euclidean* y *KNN-Minkowski*.

7.1.2 Pruebas Estadísticas Aplicadas al Desempeño de RF

De los resultados obtenidos del modelo RF en sus diferentes escenarios de configuraciones, específicamente al manipular la cantidad de árboles: 100, 200, 500. Es posible concluir que el desempeño obtenido en RF-100 es diferente que RF-200 y RF-500 como se puede observar en el siguiente planteamiento de hipótesis:

H_0: No existen diferencias significativas entre el desempeño de RF-100 (μ_1) y RF-200 o RF-500 árboles (μ_2). Según la expresión (7.2).

H_1: Existen diferencias significativas entre el desempeño de RF-100 (μ_1) y RF-200, RF-500 árboles (μ_2). Según la expresión (7.3)

La Tabla 7.2. Resume los resultados de las pruebas *t-student* aplicadas al modelo de RF-100, RF-200 y RF-500.

Tabla 7.2. Resultados de pruebas t-student del modelo RF- {100, 200, 500} para las MC.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
RF-100 vs RF-200	1.07	0.3109	No significativa
RF-100 vs RF-500	0.47	0.6172	No significativa
RF-200 vs RF-500	-0.61	0.556	No significativa

Como se puede observar la comparación entre los parámetros se demuestra que no existe diferencia significativa, con un p-valor inferior a 0.05 en todos los casos.

En la Figura 7.3 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño del modelo RF-100 y RF-200.

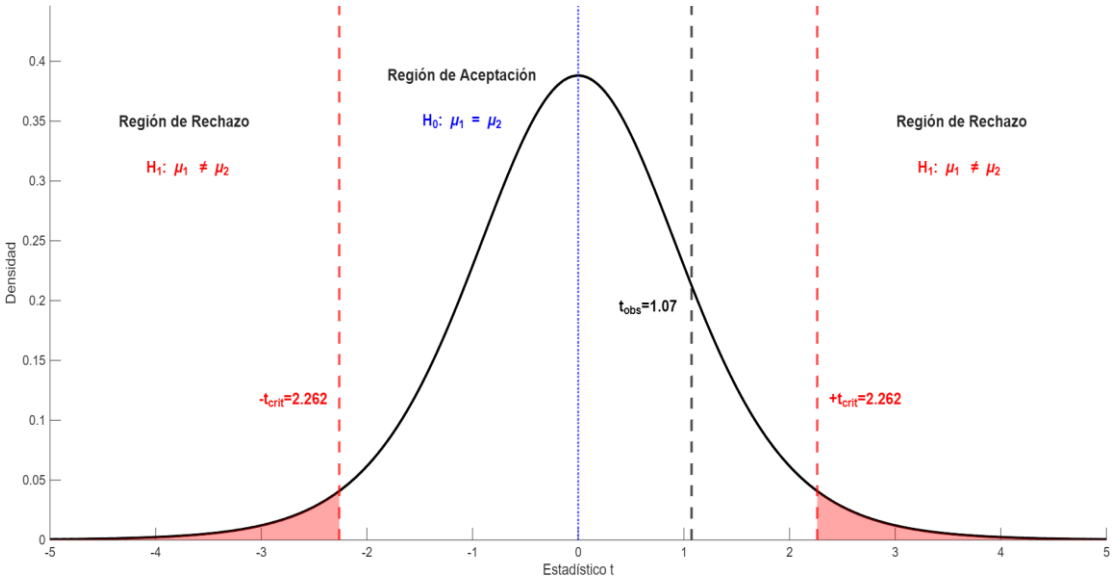

Figura 7.3. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-100 vs RF-200 para las MC.

El valor estadístico obtenido en la prueba fue *t* = 1.07 (Ecuación. 7.1), el cual se encuentra dentro de la región de aceptación; por lo tanto, se acepta la hipótesis nula (H_0), y se concluye que no existen diferencias significativas entre el desempeño de RF-100 vs RF-200.

En la Figura 7.4 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia significativa entre el desempeño de RF-100 vs RF-500.

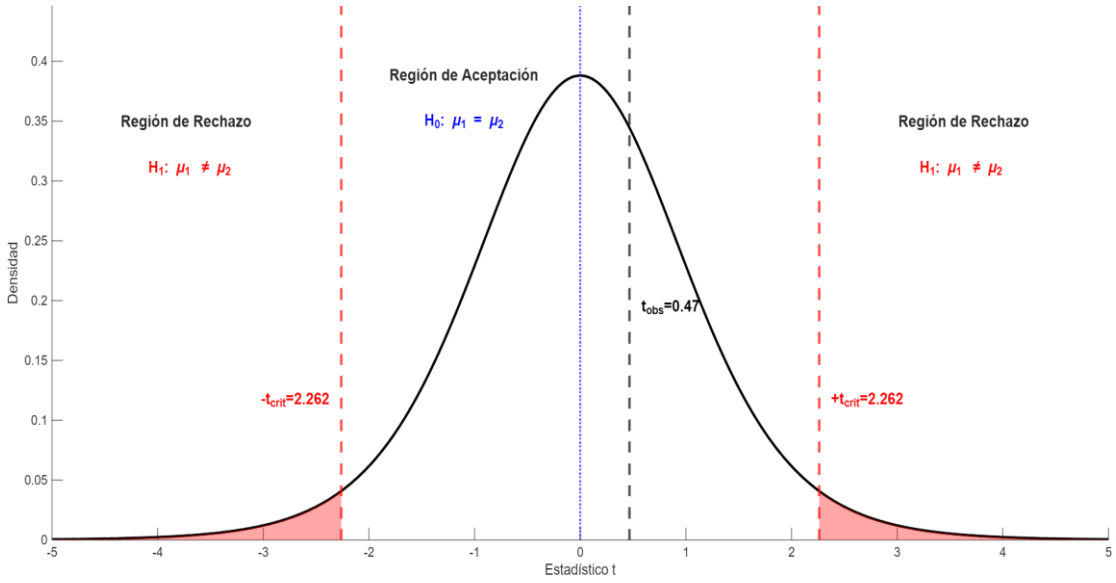

Figura 7.4. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-100 vs RF-500 para las MC.

El valor estadístico obtenido en la prueba fue *t* = 0.47 (Ecuación. 7.1), el cual se encuentra en la región de aceptación. Por lo tanto, se acepta la hipótesis nula (H_0), y se concluye que no existe diferencia significativa entre el desempeño de RF-100 y RF-500.

En la Figura 7.5 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia significativa entre el desempeño de RF-200 vs RF-500.

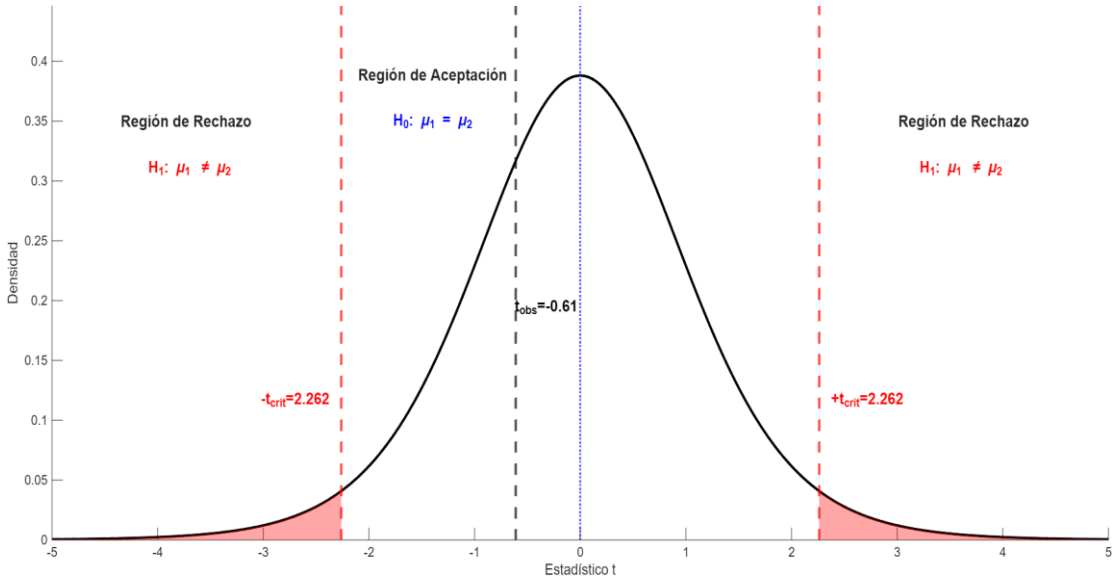

Figura 7.5. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-200 vs RF-500 para las MC.

El valor estadístico obtenido en la prueba fue *t* = -0.61 (Ecuación. 7.1), lo cual se encuentra dentro de la región de aceptación. Por lo tanto, se acepta la hipótesis nula (H_0), y se concluye que no existen diferencias significativas entre el desempeño de RF-200 vs RF-500.

7.1.3 Pruebas estadísticas aplicadas al desempeño de SVM

De los resultados obtenidos del modelo SVM en sus diferentes escenarios de configuración del parámetro *kernel*: *Radial*, *Linear* y *Polynomial*. Es posible concluir que el desempeño obtenido de *SVM-Radial* es diferente a *SVM-Linear* y *SVM-Polynomial* como se puede observar en el siguiente planteamiento de las hipótesis.

H_0: No existen diferencias significativas entre el desempeño *SVM-Radial* (μ_1) y *SVM-Linear* o *SVM-Polynomial* (μ_2), según la expresión (7.2)

H_1: Existen diferencias significativas entre el desempeño de *SVM-Radial* (μ_1) y *SVM-Linear* o *SVM-Polynomial* (μ_2), según la expresión (7.3).

La Tabla 7.3. Resume los resultados de las pruebas *t-student* aplicadas al modelo de *SVM-Radial, SVM-Linear, SVM-Polynomial.*

Tabla 7.3. Resultados de las pruebas t-student del modelo SVM- {Radial, Linear, Polynomial} para las MC.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
SVM-Radial vs *SVM-Linear*	-5.0262	0.000713	Significativa
SVM-Radial vs *SVM-Polynomial*	2.399	0.03996	Significativa

Como se puede observar la comparación entre *SVM-Radial, SVM-Linear y SVM-Polynomial* se demuestra que sí existe diferencia significativa en el desempeño en ambos casos, con un p-valor inferior a 0.05.

En la Figura 7.6 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de *SVM-Radial* y *SVM-Linear*.

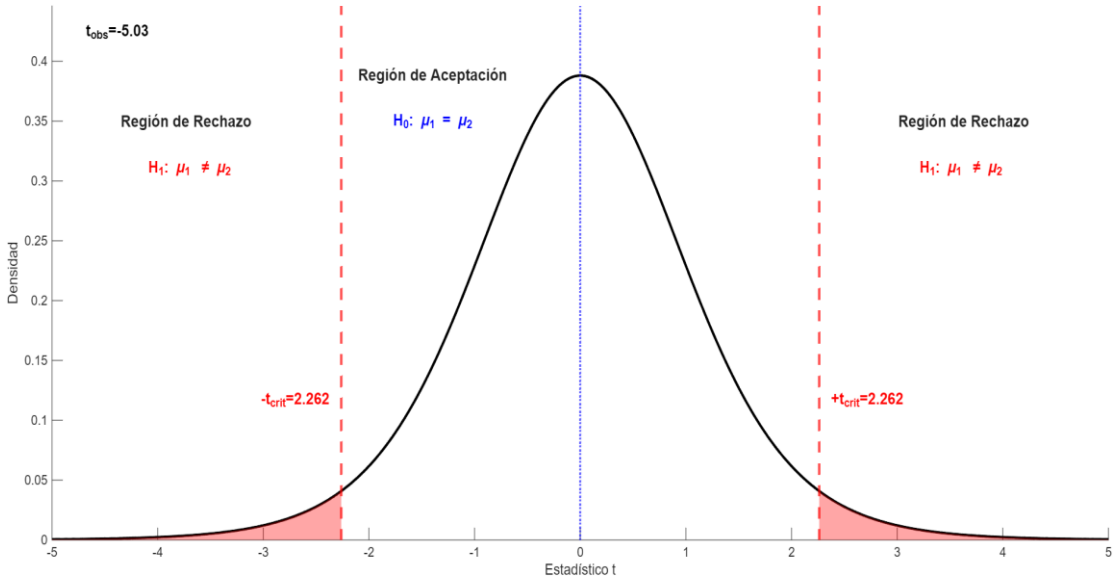

Figura 7.6. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre SVM-Radial vs SVM-Linear para las MC.

El valor estadístico obtenido en la prueba fue *t* = -5.03 (Ecuación. 7.1), el cual se encuentra en la región de rechazo. Por lo tanto, se rechaza la hipótesis nula

(H_0), y se concluye que sí existe diferencia significativa entre el desempeño de *SVM-Radial* y *SVM-Linear*.

En la Figura 7.7 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño de *SVM-Radial* y *SVM-Polynomia*l.

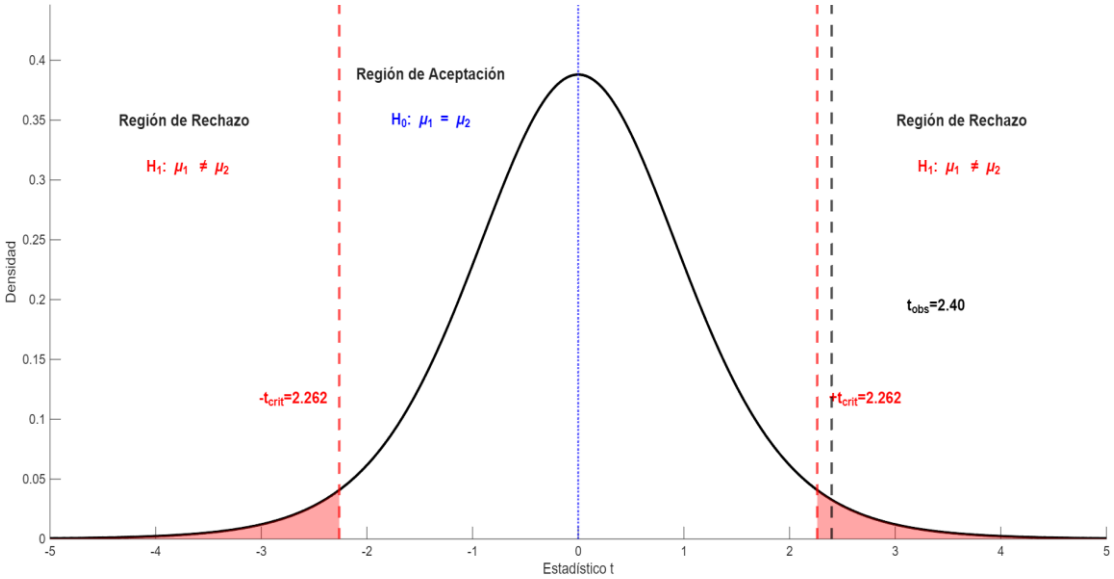

Figura 7.7. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre *SVM-Radial* vs *SVM-Polynomial* para MC.

El valor estadístico obtenido en la prueba fue *t* = 2.40 (Ecuación. 7.1), lo cual se encuentra en la región de rechazo. Por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño del modelo *SVM-Radial* y *SVM-Lineal*.

7.1.4 Pruebas Estadística para Demostrar el Desempeño de los Modelos KNN, RF, SVM, NB y MLP

En esta sección se presentan los resultados de las pruebas *t-student* aplicadas para demostrar que modelo tiene el mejor desempeño para la detección de defectos con base a los resultados presentados en la sección 7.1.1 a 7.1.3.

De acuerdo con los resultados obtenidos en el modelo *KNN-Euclidean* no existe diferencias significativas con *KNN-Manhattan*, sin embargo, la comparación entre *KNN-Euclidean* y *KNN-Minkowski* si existe con una diferencia significativa, por lo tanto, se concluye que *KNN-Euclidean* brinda mejor desempeño en la detección de defectos de software.

Los resultados obtenidos al manipular la cantidad de árboles del modelo RF-100, RF-200 y RF-500, demuestra que no existen diferencias significativas en el desempeño entre las configuraciones evaluadas.

En el modelo SVM se configuró el hiperparámetro *kernel*: *Radial*, *Linear*, y *Polynomial,* los resultados de las pruebas estadísticas demuestran que sí existen diferencias significativas en los desempeños de los modelos *SVM-Radial* en comparación con *SVM-Lineal* y *SVM-Polynomial*.

Partiendo de lo antes mencionado se desea demostrar que el modelo KNN brinda mejor desempeño que los modelos NB, RF, SVM, MLP, para ello se plantearon las siguientes hipótesis:

H_0 = Modelo KNN no representa un desempeño superior a los modelos NB, RF, SVM y MLP, representado por,

$$H_0 : \mu_1 <= \mu_2 \tag{7.4}$$

H_1 = Modelo KNN representa un mayor desempeño que los modelos NB, RF, SVM, MLP, representado por,

$$H_1 : \mu_1 > \mu_2 \tag{7.5}$$

Dónde μ_1 representa la media poblacional del modelo KNN, y μ_2 representan a las medias poblacionales para cada uno de los modelos NB, RF, SVM, MLP.

En la primera comparación, se desea evidenciar que KNN brinda mejor desempeño que NB por lo que en la Figura 7.8 se ilustra la representación de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, se quiere demostrar que el modelo KNN tiene mejor desempeño que NB.

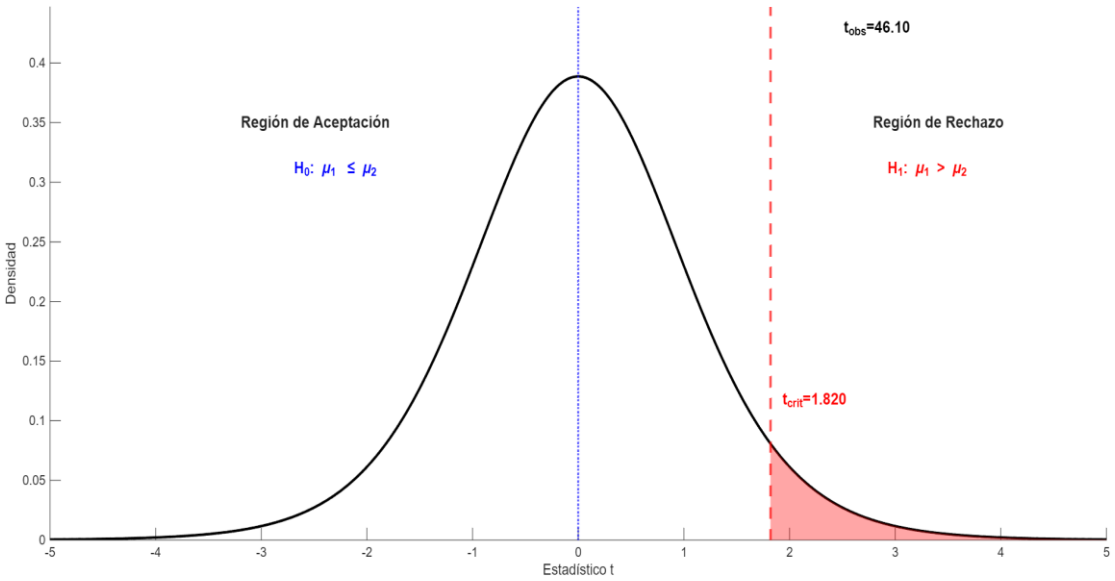

Figura 7.8. Prueba hipótesis de una cola a la derecha para demostrar el mejor desempeño entre KNN vs NB para las MC.

El valor estadístico obtenido en la prueba fue *t* = 46.10 (Ecuación. 7.1), el cual se encuentra en la región de rechazo, por lo tanto, se rechaza la hipótesis nula (H_0) y se concluye que el modelo KNN sí tiene un mejor desempeño en la detección de defectos de software que el modelo NB.

En la segunda comparación, se desea evidenciar que KNN brinda mejor desempeño que RF, por lo que en la Figura 7.9 se ilustra la representación de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, que se quiere demostrar que el modelo KNN tiene un mejor desempeño que RF.

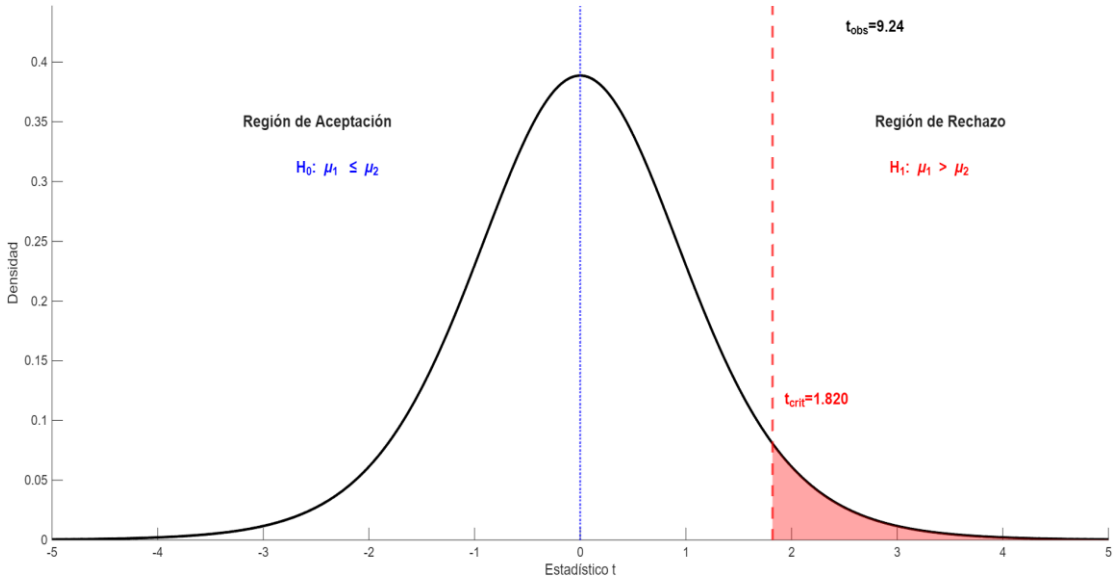

Figura 7.9. Prueba hipótesis de una cola para demostrar el desempeño entre KNN vs RF para las MC.

El valor estadístico obtenido en la prueba fue *t* = 9.24 (Ecuación. 7.1), la cual se encuentra dentro de la región de rechazo, y se concluye que sí existe diferencia significativa entre el desempeño de KNN y RF.

En la tercera comparación, se desea evidenciar que KNN brinda mejor desempeño que SVM por lo que en la Figura 7.10 se ilustra la representación de la hipótesis alternativa ($H_1 : \mu_1 > \mu_2$), es decir, que se quiere demostrar que el modelo KNN tiene mejor desempeño que SVM.

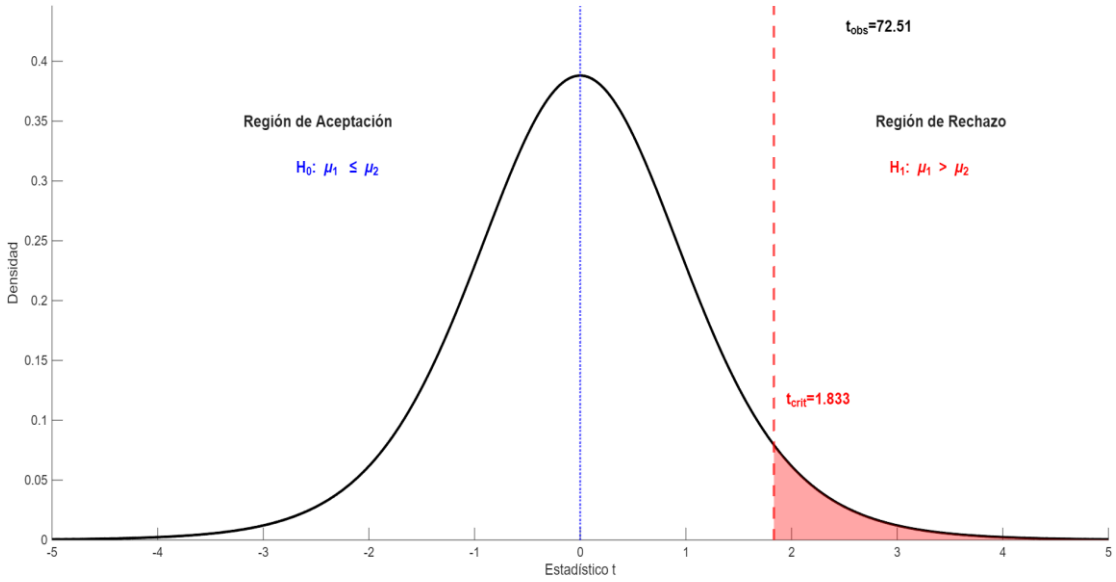

Figura 7.10. Prueba hipótesis de una cola para demostrar el mejor desempeño entre KNN vs SVM para las MC.

El valor estadístico en la prueba fue t = 72.5 (Ecuación. 7.1), el cual se encuentra en la región de rechazo. Por lo tanto, se concluye que KNN tiene un mejor desempeño en comparación con SVM.

En la cuarta comparación, se desea evidencias que KNN brinda mejor desempeño que MLP por lo que en la Figura 7.11 se ilustra la representación de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, se quiere demostrar que el modelo KNN tiene mejor desempeño que MLP.

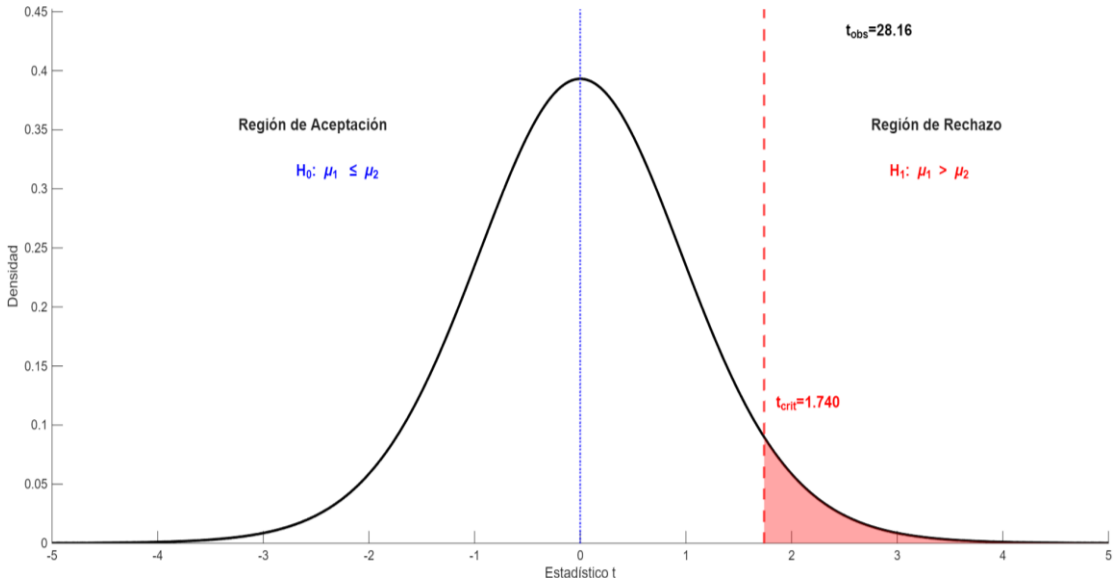

Figura 7.11. Prueba hipótesis de una cola para demostrar el mejor desempeño entre KNN vs MLP para las MC.

El valor estadístico obtenido en la prueba fue *t* = 28.16 (Ecuación. 7.1), el cual se encuentra en la región de rechazo. Por lo tanto, se concluye que sí existe evidencia significativa entre el desempeño de KNN y MLP.

En la tabla 7.4. Resume los resultados de las pruebas *t-student* aplicadas a los modelos KNN vs NB, RF, SVM, MLP.

Tabla 7.4. Resultados de las pruebas *t-student* para comparar el mejor desempeño de los modelos de las MC.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
KNN vs NB	46.103	6.776e-13	Significativa
KNN vs RF	9.2426	2.178e-06	Significativa
KNN vs SVM	41.163	3.7e-15	Significativa
KNN vs MLP	28.157	6.421e-16	Significativa

Como se puede observar al comparar el desempeño de los modelos KNN vs NB, RF, SVM y MLP el p-valor es inferior a 0.05 por lo que se demuestra que el modelo KNN sí tiene mayor desempeño en la detección de defectos de software que los otros modelos.

7.2 Pruebas Estadísticas en Métricas Orientadas a Objetos

En esta sección se presentan las pruebas estadísticas de los experimentos aplicadas a las métricas orientadas a objetos (MOO).

7.2.1 Prueba Estadística Aplicada al Desempeño de KNN

De los resultados obtenidos del modelo KNN en sus diferentes escenarios de configuración paramétricas para calcular las distancias: *Euclidean*, *Manhattan*, y *Minkowski*. Es posible concluir que el desempeño obtenido de *KNN-Euclidean* es diferente a *KNN-Manhattan* y *KNN-Minkowski* como se puede observar en el siguiente planteamiento de hipótesis:

(H_0): No existen diferencias significativas entre el desempeño *KNN-Euclidean* (μ_1) y *KNN-Manhattan* o *KNN-Minkowski* (μ_2), según la expresión (7.2).

(H_1): Existen diferencias significativas entre el desempeño de *KNN-Euclidean* (μ_1) y *KNN-Manhattan* o *KNN-Minkowski* (μ_2), según la expresión (7.3).

En la Tabla 7.5 Resume los resultados de las pruebas *t-student* aplicadas al modelo de *KNN-Euclidean, KNN-Manhattan y KNN-Minkowski*

Tabla 7.5. Resultados de las pruebas t-student del modelo KNN - {Euclidean, Manhattan, Minkowski} para MOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
KNN-Euclidean vs KNN-Manhattan	4.68	0.001148	No significativa
KNN-Euclidean vs KNN-Minkowski	7.7185	2.943e-05	No significativa

Como se puede observar la comparación entre el modelo *Euclidean* vs *Manhattan* y *Euclidean* vs *Manhattan* se demuestra que no existe diferencia significativa, con un p-valor inferior a 0.05 para ambos casos.

En la Figura 7.12 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño de *KNN-Euclidean* vs *KNN-Manhattan*.

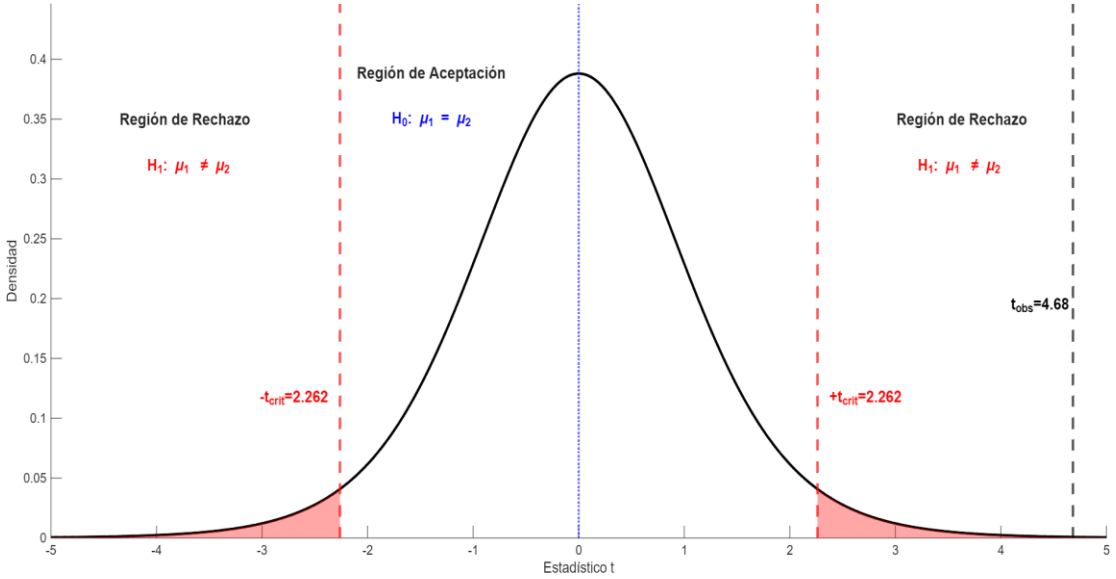

Figura 7.12. Prueba de hipótesis de dos colas para demostrar la diferencia en el desempeño entre KNN-Euclidean vs KNN-Manhattan para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 4.68 (Ecuación. 7.1), el cual se encuentra en la región rechazó; por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño de *KNN-Euclidean* y *KNN-Manhattan* para la detección de defectos de software.

En la Figura 7.13 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño de *KNN-Euclidean* vs *KNN-Minkowski*.

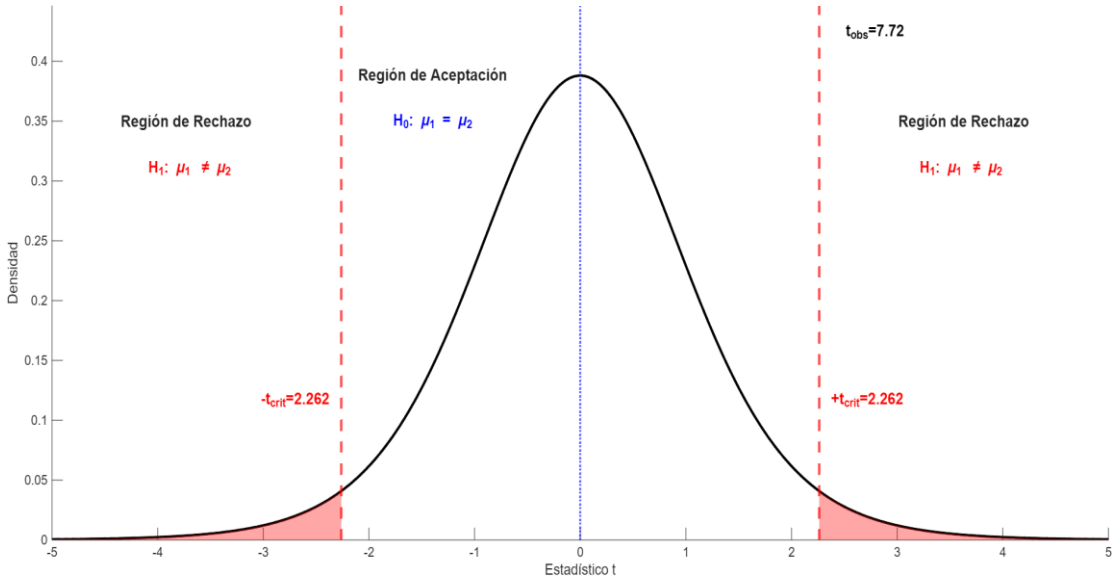

Figura 7.13. Prueba de hipótesis de dos colas para demostrar la diferencia en KNN-Euclidean vs KNN-Minkowski para MOO.

Donde el valor estadístico es de *t = 7.72* (Ecuación. 7.1), lo cual se encuentra dentro del región de rechazo; por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño de *KNN-Euclidean* y *KNN-Minkowski*.

7.2.2 Prueba Estadística Aplicada al Desempeño de RF

De los resultados obtenidos del modelo RF en sus diferentes escenarios de configuraciones, específicamente al manipular la cantidad de árboles: 100, 200, 500. Es posible concluir que el desempeño en RF-100 es diferentes que RF-200 y RF-500 como se puede observar en el siguiente planteamiento de hipótesis:

H_0: No existen diferencias significativas entre el desempeño de RF-100 (μ_1) y RF-200 o RF-500 árboles (μ_2). Según la expresión (7.2).

H_1: Existen diferencias significativas entre el desempeño de RF-100 (μ_1) y RF-200, RF-500 árboles (μ_2). Según la expresión (7.3)

La Tabla 7.6. Resume los resultados de las pruebas *t-student* aplicadas al modelo de RF-100, RF-200 y RF-500.

Tabla 7.6. Resultados de pruebas *t-student* del modelo RF - {100, 200, 500} para MOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
100 vs 200	0.71511	0.4922	No significativa
100 vs 500	0.791912	0.4481	No significativa
200 vs 500	0.45866	0.6521	No significativa

Como se puede observar la comparación entre los parámetros se demuestra que no existe diferencia significativa, con un p-valor inferior a 0.05 en todos los casos.

En la Figura 7.14 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño del modelo RF-100 y RF-200.

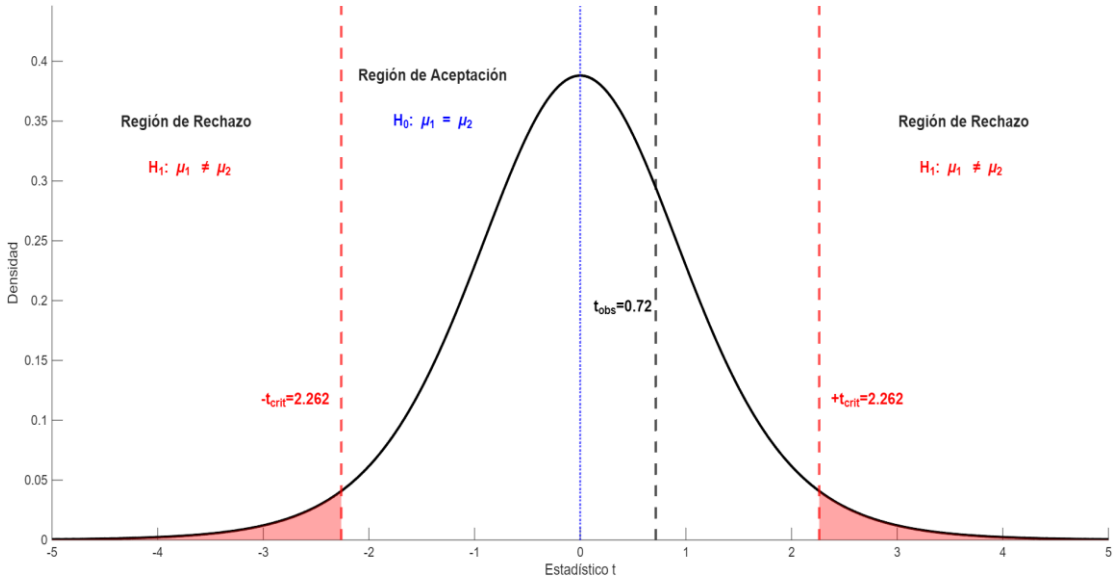

Figura 7.14. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre R-100 vs R-200 para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 0.72 (Ecuación. 7.1), el cual se encuentra dentro de la región de aceptación. Por lo tanto, se acepta la hipótesis nula (H_0), se concluye que no existen diferencias significativas entre el desempeño de RF-100 y RF-200.

En la Figura 7.15 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño de RF-100 y RF-500.

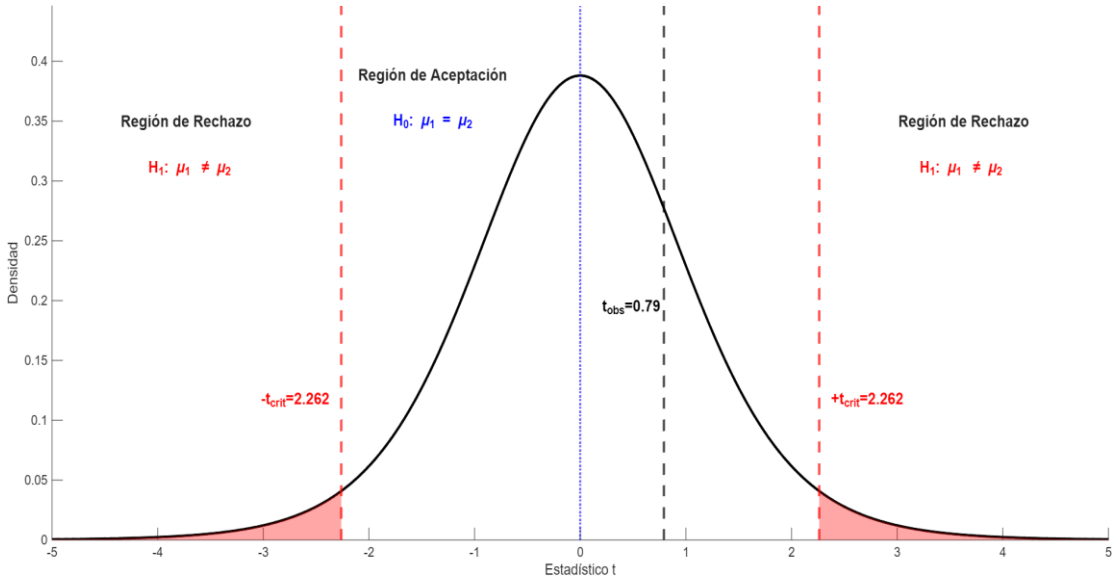

Figura 7.15. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre R-100 vs R-500 para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 0.79 (Ecuación. 7.1), el cual se encuentra en la región de aceptación. Por lo que se acepta la hipótesis nula (H_0), y se concluye que no existe diferencias significativas entre el desempeño de RF-100 y RF-500.

En la Figura 7.16 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia significativa entre el desempeño de RF-200 y RF-500.

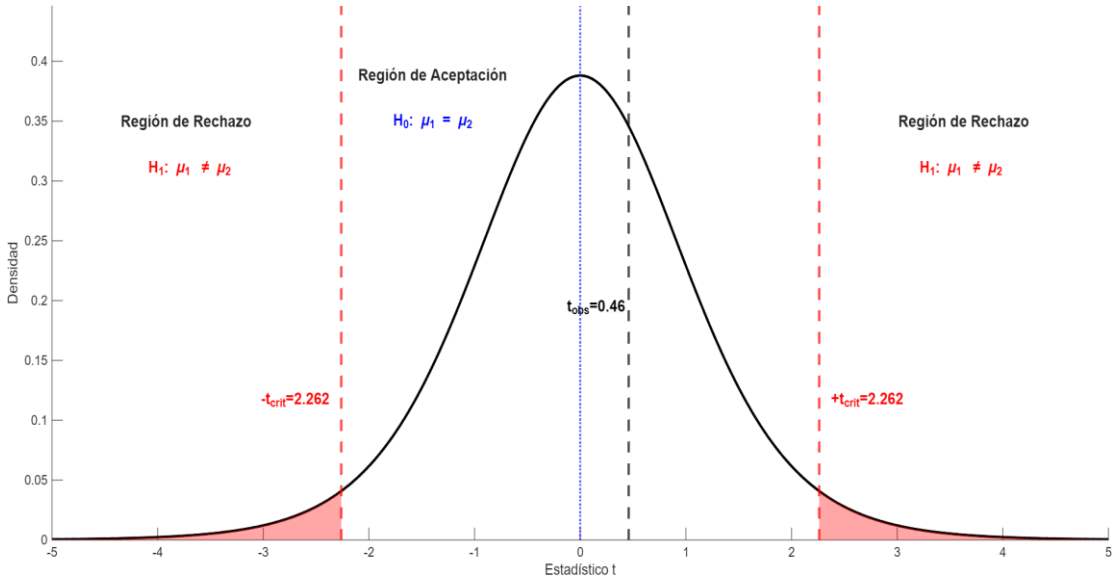

Figura 7.16. Prueba hipótesis de dos colas para demostrar la diferencia en el desempleo entre 200 vs RF-500 para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 0.46 (Ecuación. 7.1), que se encuentra dentro de la región de aceptación. Por lo tanto, se acepta la hipótesis nula (H_0), y confirma que no existen diferencias significativas entre el desempeño de RF-200 y RF-500.

7.2.3 Pruebas Estadísticas Aplicadas al Desempeño de SVM

De los resultados obtenidos del modelo SVM en sus diferentes escenarios de configuración del parámetro *kernel*: *Radial*, *Linear* y *Polynomial*. Es posible concluir que el desempeño obtenido de *SVM-Radial* es diferente a *SVM-Linear* y *SVM-Polynomial* como se puede observar en el siguiente planteamiento de hipótesis.

H_0: No existen diferencias significativas entre el desempeño *SVM-Radial* (μ_1) y *SVM-Linear* o *SVM-Polynomial* (μ_2), según la expresión (7.2)

H_1: Existen diferencias significativas entre el desempeño de *SVM-Radial* (μ_1) y *SVM-Linear* o *SVM-Polynomial* (μ_2), según la expresión (7.3).

La Tabla 7.7. Resume los resultados de las pruebas *t-student* aplicadas al modelo de *SVM-Radial, SVM-Linear* y *SVM-Polynomial*.

Tabla 7.7. Resultados de las pruebas t-student del modelo SVM - {Radial, Linear, Polynomial} para las MOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
SVM-Radial vs *SVM-Linear*	8825.5	2.2e-16	Significativa
SVM-Radial vs *SVM-Polynomial*	37.275	3.57e-11	Significativa

Como se puede observar la comparación entre SVM-*Radial*, *SVM-Linear y kernel-Polynomial* se demuestra que sí existe diferencia significativa en ambos casos, con un p-valor inferior a 0.05.

En la Figura 7.17 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño entre el desempeño de *SVM-Radial* y *SVM-Linear*.

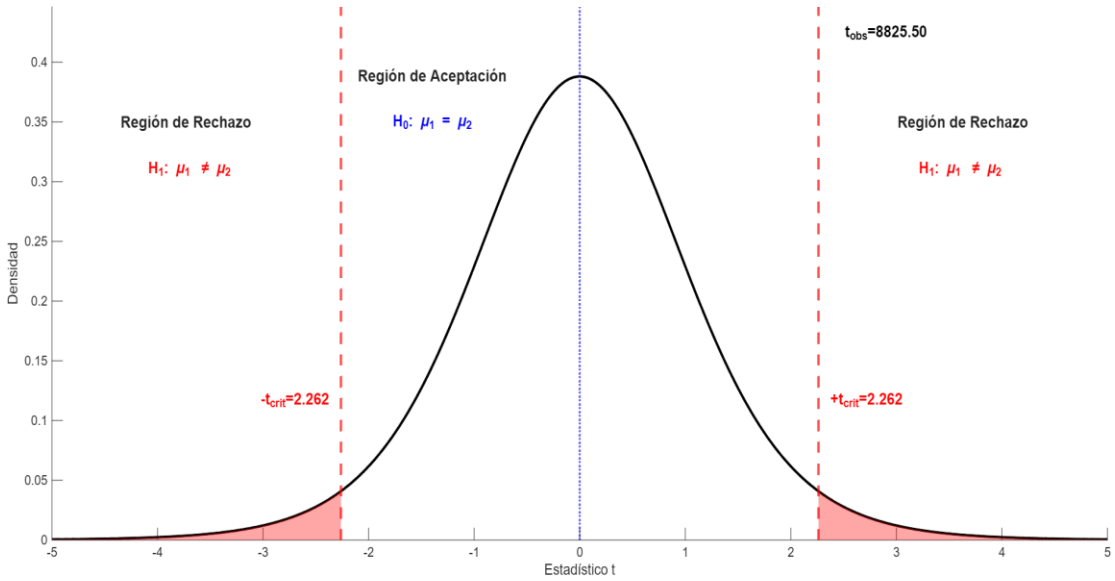

Figura 7.17. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre SVM-Radial vs SVM-Linear para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 8825.50 (Ecuación. 7.1), el cual se encuentra en la región de rechazo. Por lo tanto, se acepta la hipótesis

alternativa (H_1), y se concluye que sí existe diferencia significativa entre el desempeño de *SVM-Radial* y *SVM-Linear*.

En la Figura 7.18 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existe diferencia entre el desempeño de *SVM-Radial* y *SVM-Polynomia*l.

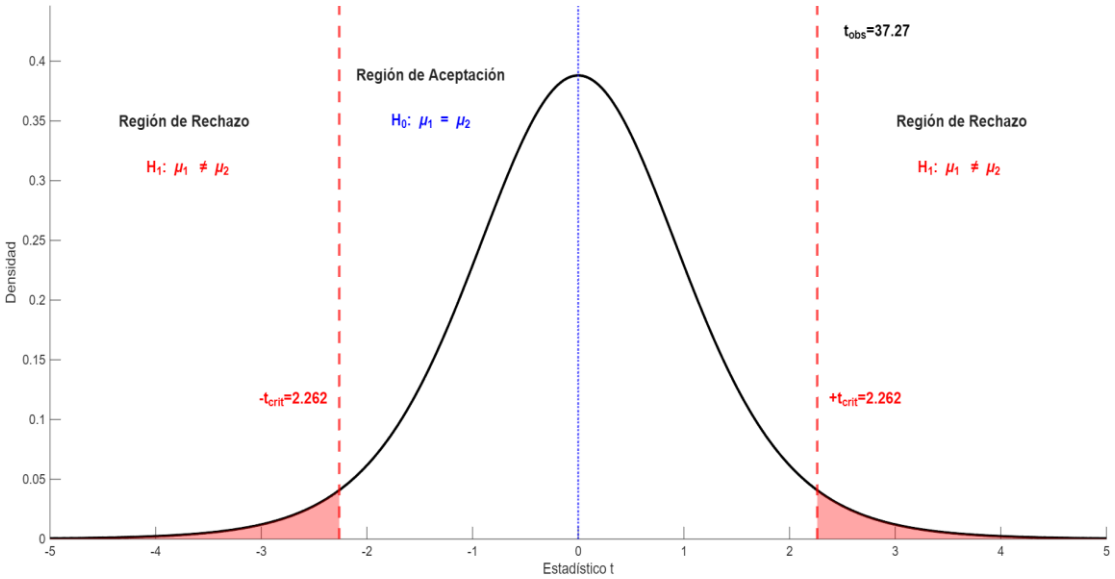

Figura 7.18. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre SVM-Radial vs SVM-Polynomial para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 37.27 (Ecuación. 7.1), lo cual se encuentra en la región de rechazo, por lo que se acepta la hipótesis nula (H_1), y se concluye que sí existe diferencia significativa entre el desempeño de *SVM-Radial* y *SVM-Polynomial*.

7.2.4 Pruebas Estadística para Demostrar el Desempeño de los Modelos KNN, RF, SVM, NB y MLP

En esta sección se presentan los resultados de las pruebas *t-student* para demostrar que modelo tiene el mejor desempeño para la detección de defectos con base a los resultados presentados en la sección 7.2.1 a 7.2.3.

De acuerdo con los resultados obtenidos en el modelo *KNN-Euclidean* no existe diferencias significativas con *KNN-Manhattan* y *KNN-Minkowski*, por lo tanto, se concluye que la distancias con mejor desempeño es *KNN-Euclidean*.

Los resultados obtenidos al manipular la cantidad de árboles del modelo RF-100, RF-200 y RF-500 árboles, se demostró que no existe diferencia significativa en el desempeño de las configuraciones.

En el modelo SVM se configuró el hiperparámetro *kernel*: *Radial*, *Linear*, y *Polynomial,* los resultados de las pruebas estadísticas demuestran que si existen diferencias significativas y se concluyó que el SVM-Radial cuenta con mejor desempeño.

Partiendo de lo antes mencionado se desea demostrar que el desempeño obtenido del algoritmo KNN es mejor que los otros modelos NB, RF, SVM, MLP, para ello se plantearon las hipótesis siguientes:

H_0 = Modelo KNN no representa un desempeño superior a los modelos NB, RF, SVM y MLP, representado por,

$$H_0 : \mu_1 <= \mu_2 \quad (7.4)$$

H_1 = Modelo KNN tiene un mayor desempeño que los modelos NB, RF, SVM, MLP, representado por,

$$H_1 : \mu_1 > \mu_2 \quad (7.5)$$

Dónde μ_1 representa la media poblacional del modelo KNN, y μ_2 representan a las medias poblacionales para cada uno de los modelos NB, RF, SVM, MLP.

En la primera comparación, se desea evidenciar que KNN brinda mejor desempeño que NB, por lo que en la Figura 7.19 se ilustra la representación de la hipótesis alternativa ($H_1 : \mu_1 > \mu_2$), es decir, que se quiere demostrar que el modelo KNN tiene mejor desempeño que NB.

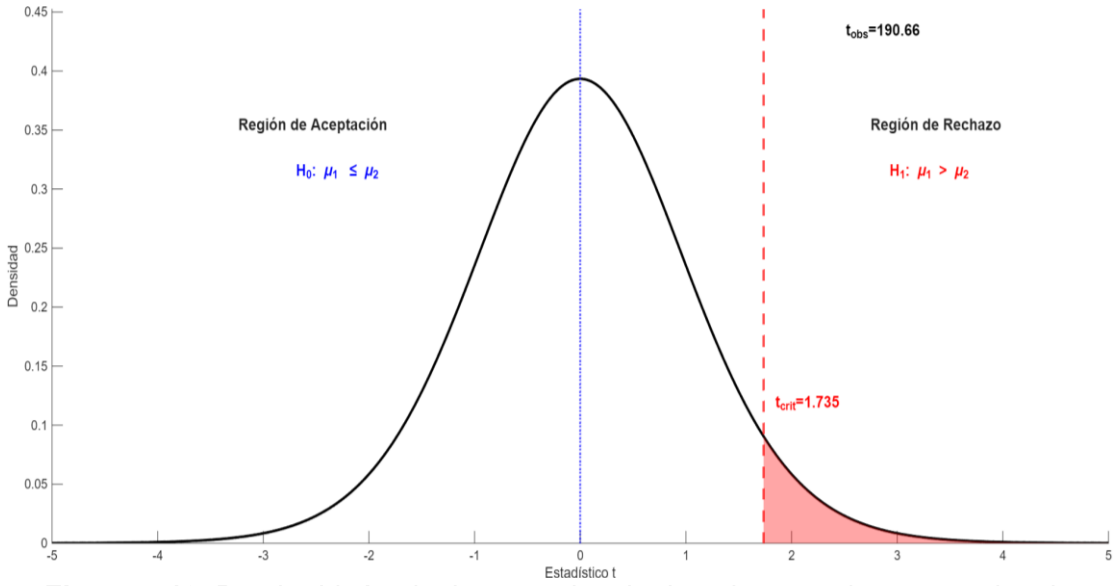

Figura 7.19. Prueba hipótesis de una cola a la derecha para demostrar el mejor desempeño entre KNN vs NB para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 190.66 (Ecuación. 7.1), el cual se encuentra en la región de rechazo, por lo tanto, se rechaza la hipótesis nula (H_0) y se concluye que sí existe diferencia significativa entre los algoritmos KNN y NB.

En la segunda comparación, se desea evidenciar que KNN brinda mejor desempeño que NB, por lo que en la Figura 7.20 se ilustra la representación de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, que se quiere demostrar que el modelo KNN tiene mejor desempeño que RF.

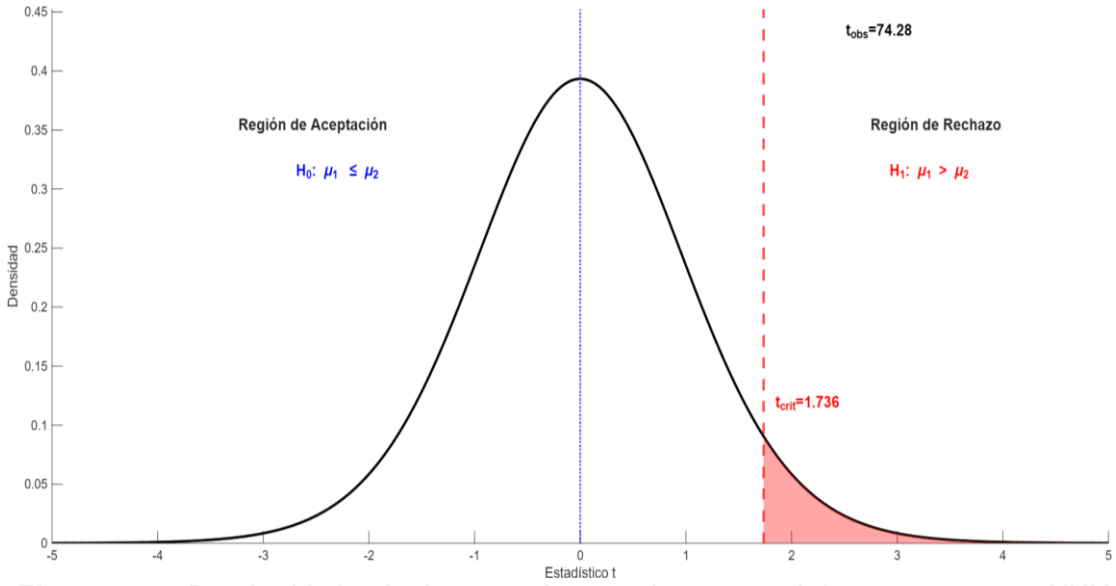

Figura 7.20. Prueba hipótesis de una cola para demostrar el desempeño entre KNN vs RF para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 74.28 (Ecuación. 7.1), la cual se encuentra dentro de la región de rechazo, y se concluye que sí existe evidencia significativa para el desempeño de KNN y RF.

En la tercera comparación, se desea evidenciar que KNN brinda mejor desempeño que SVM, por lo que en la Figura 7.21 se ilustra la representación de la hipótesis alternativa ($H_1 : \mu_1 > \mu_2$), es decir, que se quiere demostrar que el modelo KNN tiene mejor desempeño que SVM.

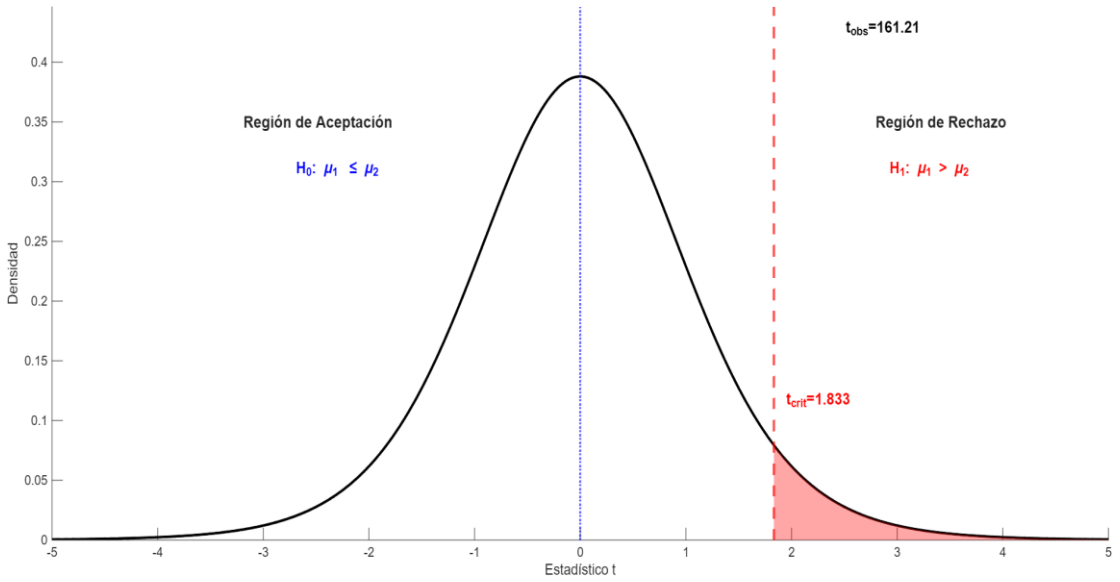

Figura 7.21. Prueba hipótesis de una cola para demostrar el mejor desempeño entre KNN vs SVM para las MOO.

El valor estadístico en la prueba fue t = 161.21 (Ecuación. 7.1), el cual se encuentra en la región de rechazo. Por lo tanto, se concluye que KNN tiene un mejor desempeño en comparación con SVM.

En la cuarta comparación, se desea evidenciar que KNN brinda mejor desempeño que MLP, por lo que en la Figura 7.22 se ilustra la representación de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, que se quiere demostrar que el modelo KNN tiene mejor desempeño que MLP.

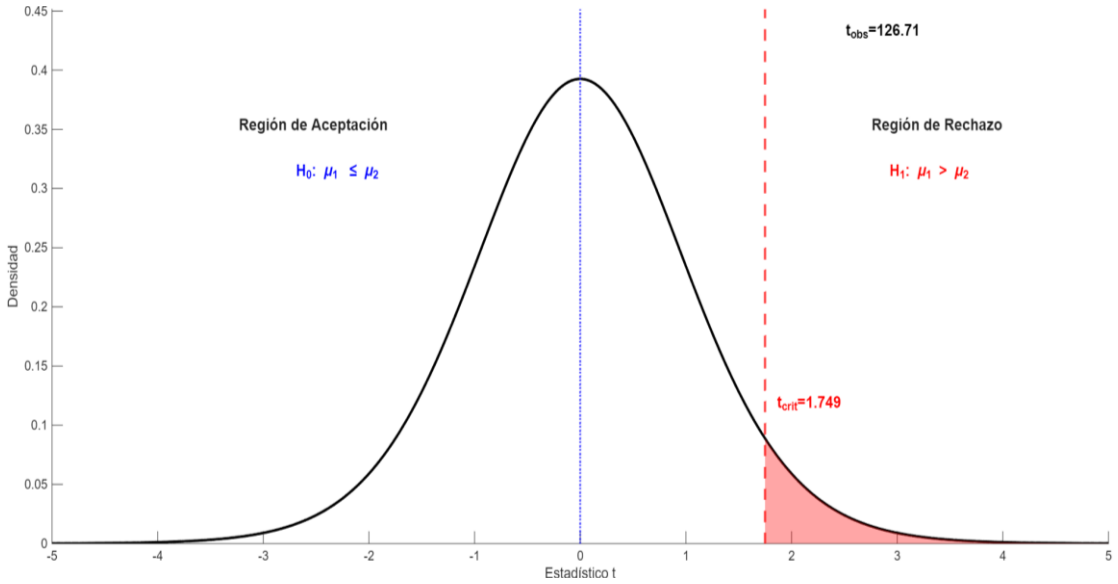

Figura 7.22. Prueba hipótesis de una cola para demostrar el mejor desempeño entre KNN vs MLP para las MOO.

El valor estadístico obtenido en la prueba fue *t* = 126.71 (Ecuación. 7.1), la cual se encuentra en la región de rechazo. Por lo tanto, se concluye que KNN tiene un mejor desempeño en comparación con MLP.

En la Tabla 7.8. Resume los resultados de las pruebas *t-student* aplicadas de los modelos KNN vs NB, RF, SVM, MLP.

Tabla 7.8. Resultados de las pruebas *t-student* aplicadas para comparar el mejor desempeño de las MOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
KNN vs Naïve Bayes	190.66	2.2e-16	Significativa
KNN vs RF	74.277	2.2e-16	Significativa
KNN vs SVM	161.21	2.2e-16	Significativa
KNN vs MLP	126.71	2.2e-16	Significativa

Como se puede observar al comparar el desempeño de los modelos KNN vs NB, RF, SVM y MLP el p-valor es inferior a 0.05 por lo que se puede se demuestra que el modelo KNN sí tiene mayor desempeño en la detección de defectos de software que los otros modelos.

7.3 Pruebas estadísticas en Métricas de Complejidad y Métricas de Software Orientada a Objetos

En esta sección se presentan las pruebas estadísticas de los experimentos aplicadas a las Métricas de Complejidad y Métricas Orientadas a Objetos (MCyMOO).

7.3.1 Prueba Estadística Aplicada al Desempeño de KNN

De los resultados obtenidos del modelo KNN en sus diferentes escenarios de configuraciones paramétricas para calcular las distancias: *Euclidean, Manhattan, y Minkowski*. Es posible concluir que el desempeño obtenido de *KNN-Euclidiana* es diferente a *KNN-Manhattan* y *KNN-Minkowski* como se puede observar en el siguiente planteamiento de hipótesis:

H_0: No existen diferencias significativas entre el desempeño de *KNN-Euclidean* (μ_1) y *KNN-Manhattan* o *KNN-Minkowski* (μ_2), según la expresión (7.2).

H_1: Existen diferencias significativas entre el desempeño de *KNN-Euclidean* (μ_1) y *KNN-Manhattan* o *KNN-Minkowski* (μ_2), según la expresión (7.3).

La Tabla 7.9. Resume los resultados de las pruebas *t-student* aplicadas al modelo de KNN-*Euclidean*, KNN-*Manhattan* y KNN-*Minkowski*.

Tabla 7.9. Resultados de las pruebas t-student del modelo KNN - {Euclidean, Manhattan, Minkowski} para las MCyMOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
KNN-Euclidean vs KNN-Manhattan	-7.78	2.755e-05	Significativa
KNN-Euclidean vs KNN-Minkowski	2.0567	0.06985	No significativa

Como se puede observar la comparación entre el modelo KNN-*Euclidean* vs KNN-*Manhattan* se demuestra que sí existe diferencia significativa, con un p-valor = 2.755e-05 lo cual es inferior a 0.05. Por el contrario, las KNN-*Euclidean*

vs KNN-*Minkowski* no mostraron diferencias significativas, ya que su p-valor fue superior a 0.05.

En la Figura 7.23 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de *KNN-Euclidean* vs *KNN-Manhattan*.

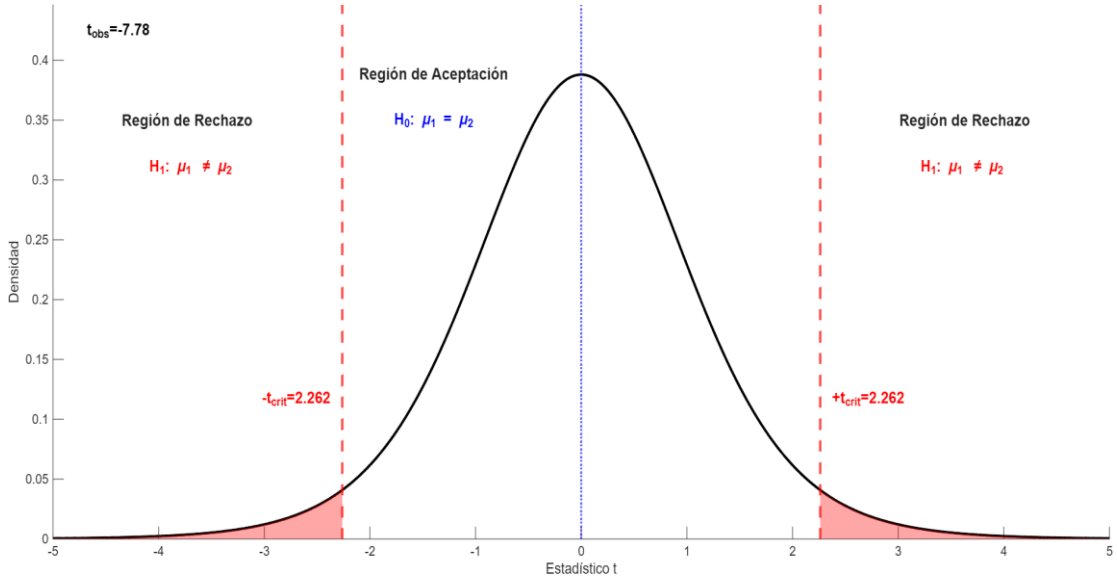

Figura 7.23. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-100 vs RF-500 para las MC.

El valor estadístico obtenido en la prueba fue *t* = -7.78 (Ecuación. 7.1), el cual se encuentra en la región de rechazo; por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño de *KNN-Euclidean* y *KNN-Manhattan* para la detección de defectos de software.

En la Figura 7.24 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de *KNN-Euclidean* y *KNN-Minkowski*.

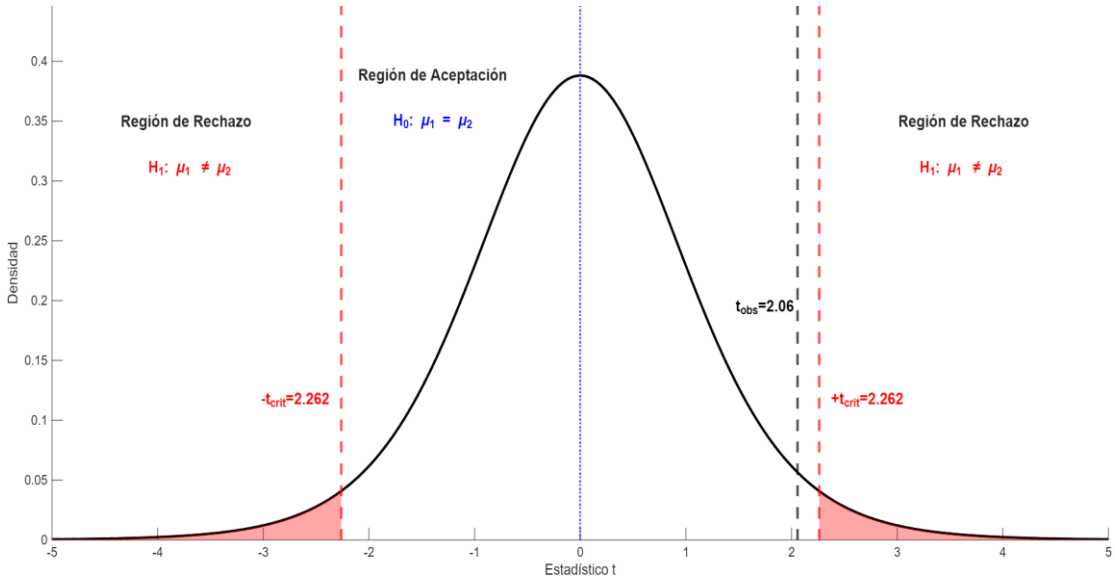

Figura 7.24. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre *KNN-Euclidean* vs *KNN-Minkowski* para las MCyMOO

Donde el valor estadístico fue *t = 2.06* (Ecuación. 7.1), el cual se encuentra dentro de la región de aceptación. Por lo tanto, se concluye que no existe diferencia significativa entre el desempeño de *KNN-Euclidean* y *KNN-Minkowski*.

7.3.2 Prueba Estadística Aplicada al Desempeño de RF

De los resultados obtenidos del modelo RF en sus diferentes escenarios de configuraciones, específicamente al manipular la cantidad de árboles: 100, 200, 500. Es posible concluir que el desempeño obtenido en RF-100 es diferente a RF-200 y RF-500 como se puede observar en el siguiente planteamiento de hipótesis:

H_0: No existen diferencias significativas en el desempeño de RF-100 (μ_1) y RF-200 o RF-500 árboles (μ_2). Según la expresión (7.2).

H_1: Existen diferencias significativas entre el desempeño de RF-100 (μ_1) y RF-200, RF-500 árboles (μ_2). Según la expresión (7.3)

La Tabla 7.10. Resume los resultados de las pruebas *t-student* aplicadas al modelo de RF-100, RF-200 y RF-500.

Tabla 7.10. Resultados de pruebas t-student del modelo RF- {100, 200, 500} para las MCyMOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
RF-100 vs RF-200	-2.1466	0.6037	No Significativa
RF-100 vs RF-500	-3.1969	0.01089	Significativa
RF-200 vs RF-500	-1.0338	0.3282	No significativa

Como se puede observar la comparación entre los modelos RF-100 vs RF-500 se demuestra que, si existe diferencia significativa, con un p-valor = 0.01089 lo cual es inferior a 0.05. Por el contrario, en los modelos RF-100 vs RF-200 y RF-200 vs RF-500 no existen diferencias significativas por lo que el p-valor fue superior a 0.05.

En la Figura 7.25 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de RF-100 y RF-200.

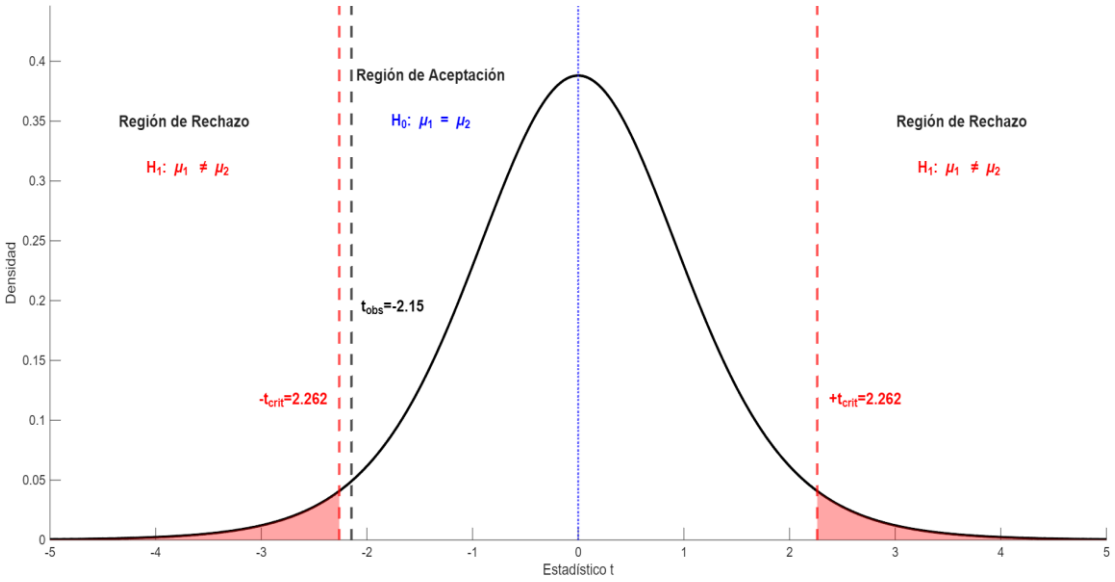

Figura 7.25. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-100 vs RF-200 para las MCyMOO.

El valor estadístico obtenido en la prueba fue *t* = -2.15 (Ecuación. 7.1) el cual se encuentra dentro la región de aceptación; por lo tanto, se acepta la hipótesis nula (H_0), y se concluye que no existen diferencias significativas entre el desempeño de RF-100 vs RF-200.

En la Figura 7.26 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas en el desempeño de RF-100 vs RF-500.

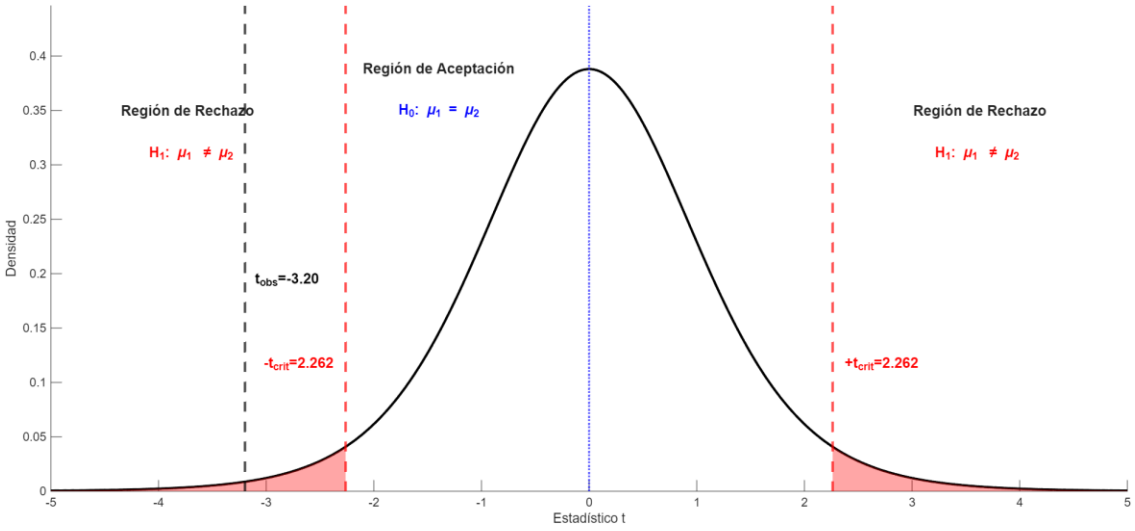

Figura 7.26. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-100 vs RF-500 para las MCyMOO.

El valor estadístico obtenido en la prueba fue *t* = -3.20 (Ecuación. 7.1), el cual se encuentra en la región de rechazo. Por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño de RF-100 vs RF-500.

En la Figura 7.27 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de RF-200 vs RF-500.

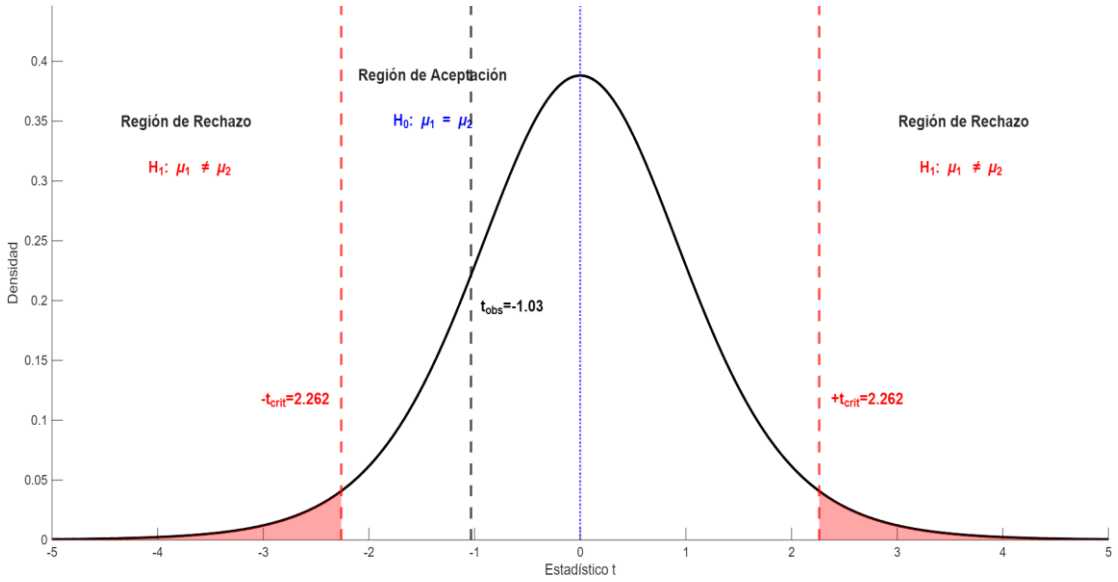

Figura 7.27. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre RF-200 vs RF-500 para las MCyMOO.

Donde el valor estadístico fue *t* = -1.03 (Ecuación. 7.1), lo cual se encuentra dentro de la región de aceptación. Por lo tanto, se concluye que no existen diferencias significativas en el desempeño entre RF-200 y RF-500.

7.3.3 Pruebas Estadísticas Aplicadas al Desempeño de SVM

De los resultados obtenidos del modelo SVM en sus diferentes escenarios de configuración del parámetro *Kernel*: *Radial*, *Linear* y *Polynomial*. Es posible concluir que el desempeño obtenido de *SVM-Radial* es diferentes a *SVM-Linear* y *SVM-Polynomial* como se puede observar en el siguiente planteamiento de las hipótesis.

H_0: No existen diferencias significativas entre el desempeño *SVM-Radial* (μ_1) y *SVM-Linear* o *SVM-Polynomial* (μ_2), según la expresión (7.2)

H_1: Existen diferencias significativas entre el desempeño de *SVM-Radial* (μ_1) y *SVM-Linear* o *SVM-Polynomial* (μ_2), según la expresión (7.3).

La Tabla 7.11. Resume los resultados de las pruebas *t-student* aplicadas al modelo de *SVM-Radial, SVM-Linear* y *SVM-Polynomial*.

Tabla 7.11. Resultados de las pruebas t-student del modelo SVM- {Radial, Linear, Polynomial} para las MCyMOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
SVM-Radial vs *SVM-Linear*	-34.192	7.725e-11	Significativa
SVM-Radial vs *SVM-Polynomial*	223.93	2.2e-16	Significativa

Como se puede observar la comparación entre *SVM-Radial, SVM-Linear y SVM-Polynomial* se demuestra que sí existe diferencia significativa en ambos casos, con un p-valor inferior a 0.05.

En la Figura 7.28 se ilustra la representación de la hipótesis alternativa ($H_1: \mu_1 \neq \mu_2$), es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de *SVM-Radial* y *SVM-Linear*.

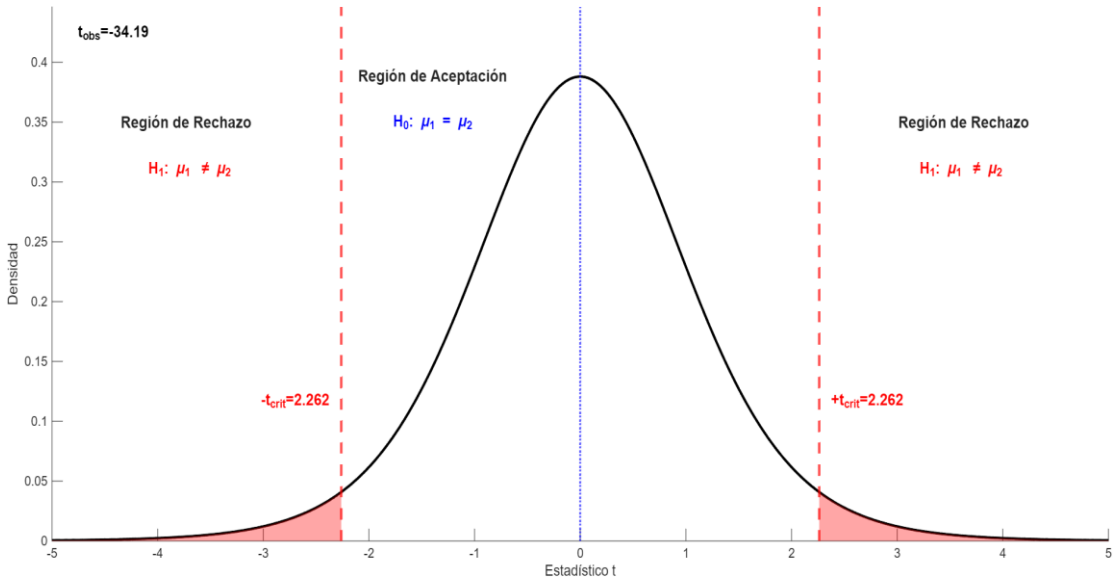

Figura 7.28. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre SVM-Radial vs SVM-Linear para las MCyMOO

El valor estadístico obtenido en la prueba fue *t* = -34.19 (Ecuación. 7.1) el cual se encuentra en la región de rechazo. Por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que si existe diferencia significativa entre el desempeño de *SVM-Radial* y *SVM-Linear*.

En la Figura 7.29 se ilustra la representación de la hipótesis alternativa $(H_1: \mu_1 \neq \mu_2)$, es decir, se quiere demostrar que existen diferencias significativas entre el desempeño de *SVM-Radial* y *SVM-Polynomial*.

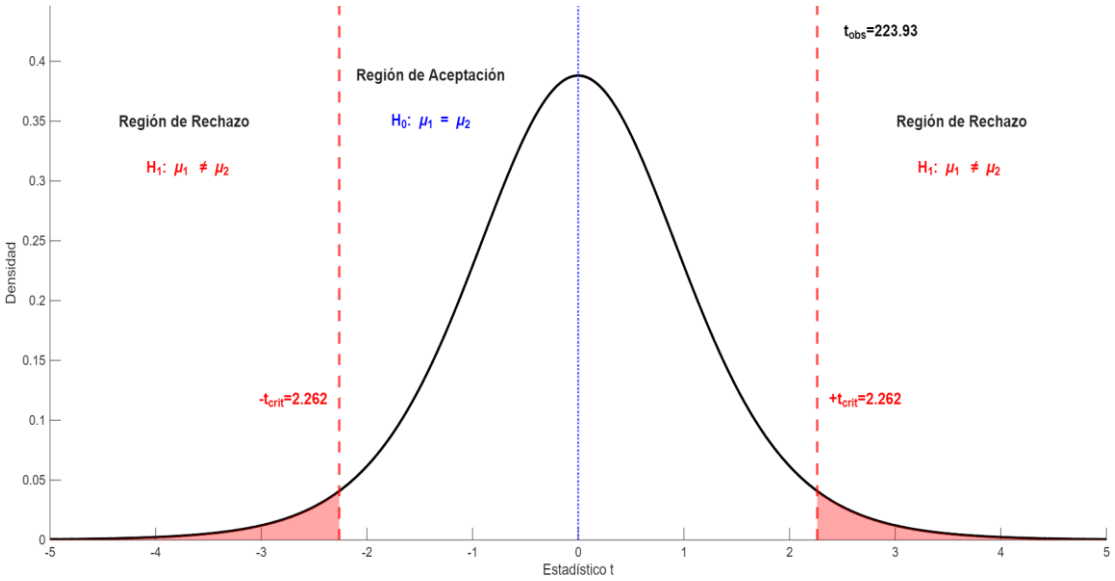

Figura 7.29. Prueba hipótesis de dos colas para demostrar la diferencia en el desempeño entre SVM-Radial vs SVM-Polynomial para las MCyMOO.

Donde el valor estadístico fue t = 223.93 (Ecuación. 7.1), lo cual se encuentra dentro la región de rechazo. Por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa en el desempeño entre *SVM-Radial* y *SVM-Polynomial*.

7.3.4 Pruebas Estadísticas para Demostrar el Desempeño de los Modelos KNN, RF, SVM, NB y MLP

En esta sección se presentan los resultados de las pruebas *t-student* aplicadas para demostrar qué modelo tiene el mejor desempeño para la detección de defectos con base a los resultados presentados en la sección 7.3.1 a 7.3.3.

De acuerdo con los resultados obtenidos en el modelo *KNN-Euclidean* no existe diferencias significativas con *KNN-Manhattan*, sin embargo, la comparación de entre *KNN-Euclidean* y *KNN-Minkowski* si existe diferencias significativas, por lo tanto, se concluye que *KNN-Euclidean* brinda mejor desempeño en la detección de defectos de software.

Los resultados obtenidos al manipular la cantidad árboles del modelo RF-100, RF-200 y RF-500, demuestra que no existe diferencias significativas en el desempeño al comparar RF-100 vs RF-200 y RF-200 vs RF-500, mientras que al comparar RF-100 vs RF-500, se muestra que si existe diferencias significativas en el desempeño de la clasificación. Por tanto, se concluye que RF-100 brinda mejor desempeño en la detección de defectos de software que RF-200 y RF-500.

En el modelo SVM se configuró el hiperparámetro *kernel*: *Radial*, *Linear*, y *Polynomial,* los resultados de las pruebas estadísticas demuestran que sí existen diferencias significativas en los desempeños de los modelos, pero, *SVM-Radial* brinda mejor desempeño que *SVM-Linear* y *SVM-Polynomial*.

Partiendo de lo antes mencionado se desea demostrar que el modelo RF brinda mejor desempeño que los modelos KNN, NB, SVM, MLP, para ello se plantearon las hipótesis siguientes:

H_0 = Modelo RF no representa un desempeño superior a los modelos KNN, SVM, NB y MLP representando por,

$$H_0 : \mu_1 <= \mu_2 \quad (7.4)$$

H_1 = Modelo RF representa un mayor desempeño que los modelos KNN, SVM, NB y MLP, representado por,

$$H_1 : \mu_1 > \mu_2 \quad (7.5)$$

Dónde μ_1 representa la media poblacional del modelo RF, y μ_2 representan a las medidas poblacionales para cada uno de los modelos KNN, NB, SVM y MLP.

En la primera comparación, se desea evidenciar que RF brinda mejor desempeño que KNN por lo que en la Figura 7.30 se ilustra la representación de la hipótesis alternativa ($H_1 : \mu_1 > \mu_2$), es decir, se quiere demostrar que el modelo RF tiene mayor desempeño que KNN.

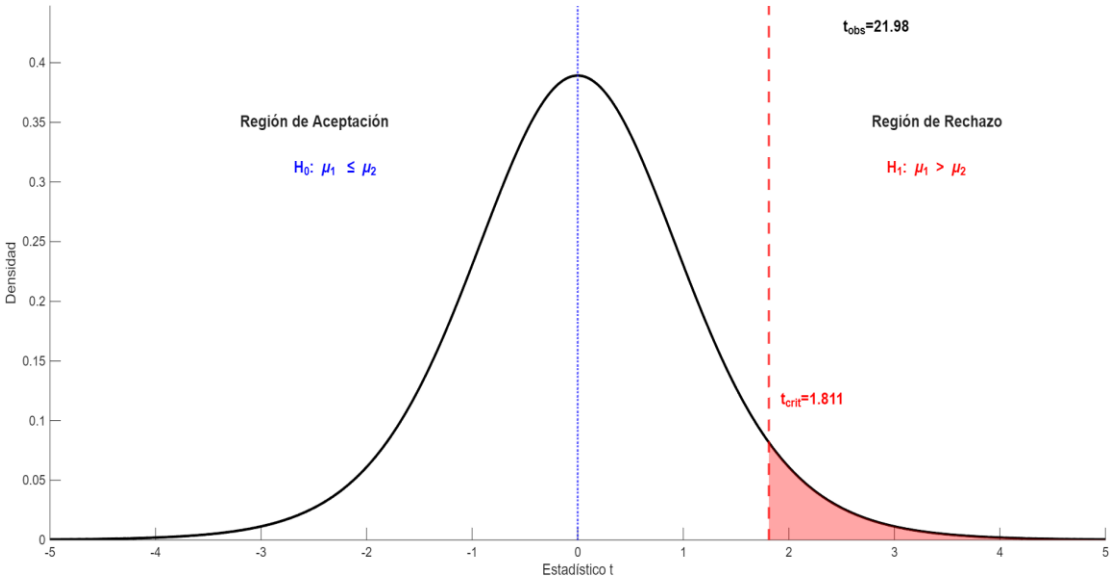

Figura 7.30. Prueba hipótesis de una cola a la derecha para demostrar el mejor desempeño entre RF vs KNN para las MCyMOO.

El valor estadístico obtenido en la prueba fue *t* = 21.98 (Ecuación. 7.1), el cual se encuentra en la región de rechazo; por lo tanto, se rechaza la hipótesis nula (H_0) y se concluye que el modelo RF sí tiene un mejor desempeño en la dirección de defectos de software que el modelo KNN.

En la segunda comparación, se desea evidenciar que RF brinda mejor desempeño que NB por lo que en la Figura 7.31 se ilustra la representación de la hipótesis alternativa ($H_1 : \mu_1 > \mu_2$), es decir, se quiere demostrar que el modelo RF tiene un mejor desempeño que NB.

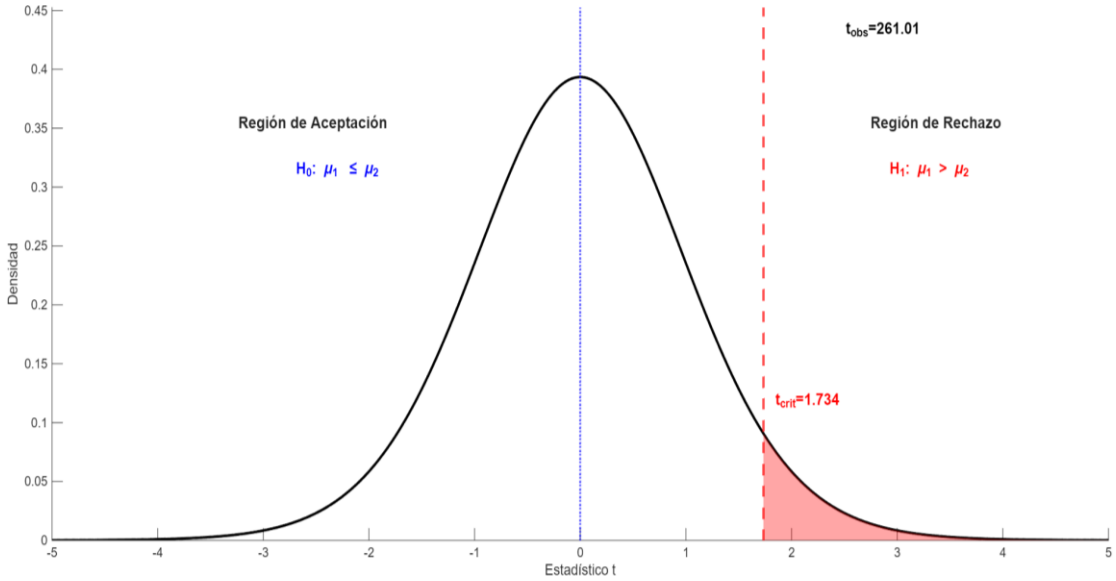

Figura 7.31. Prueba de hipótesis de una cola para demostrar el mejor desempeño entre RF vs NB para las MCyMOO.

El valor estadístico obtenido en la prueba fue *t* = 261.01 (Ecuación. 7.1) la cual se encuentra dentro de la región de rechazo, y se concluye que sí existe diferencia significativa entre el desempeño de RF y NB.

En la tercera comparación, se desea evidenciar que RF brinda mejor desempeño que SVM por lo que en la Figura 7.32 se ilustra la representación

de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, se quiere demostrar que el modelo RF tiene mejor desempeño que SVM.

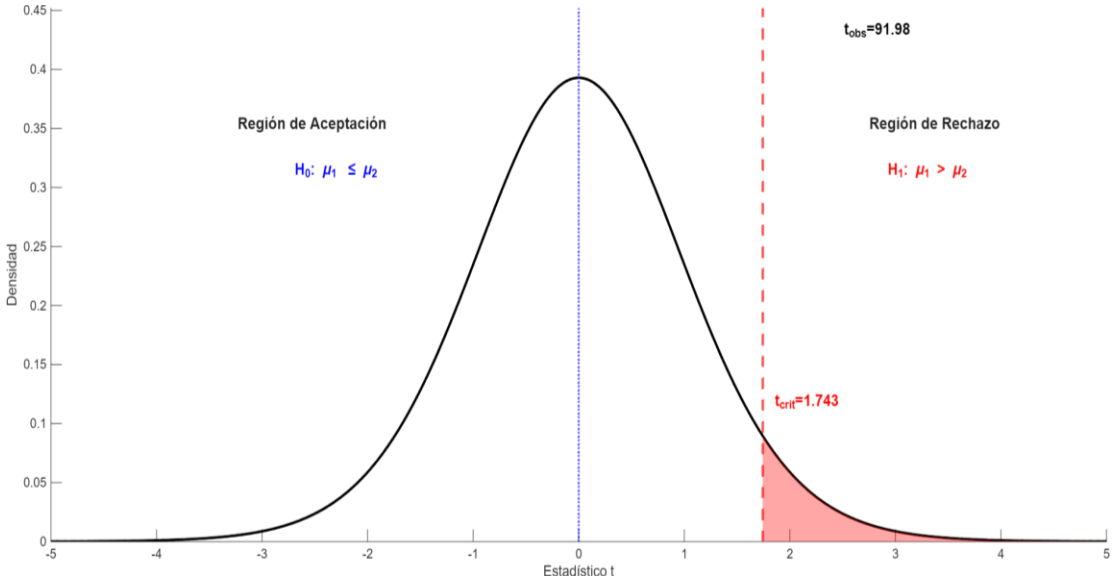

Figura 7.32. Prueba hipótesis de una cola para demostrar el mejor desempeño entre RF y SVM para MCyMOO.

El valor estadístico obtenido en la prueba fue *t* = 91.68 (Ecuación. 7.1), lo cual se encuentra en la región rechazó; por lo tanto, se rechaza la hipótesis nula (H_0), y se concluye que sí existe diferencia significativa entre el desempeño de RF y SVM.

En la cuarta comparación, se desea evidenciar que RF brinda mejor desempeño que MLP por lo que en la Figura 7.33 se ilustra la representación de la hipótesis alternativa $(H_1 : \mu_1 > \mu_2)$, es decir, se quiere demostrar que el modelo RF tiene mejor desempeño que MLP.

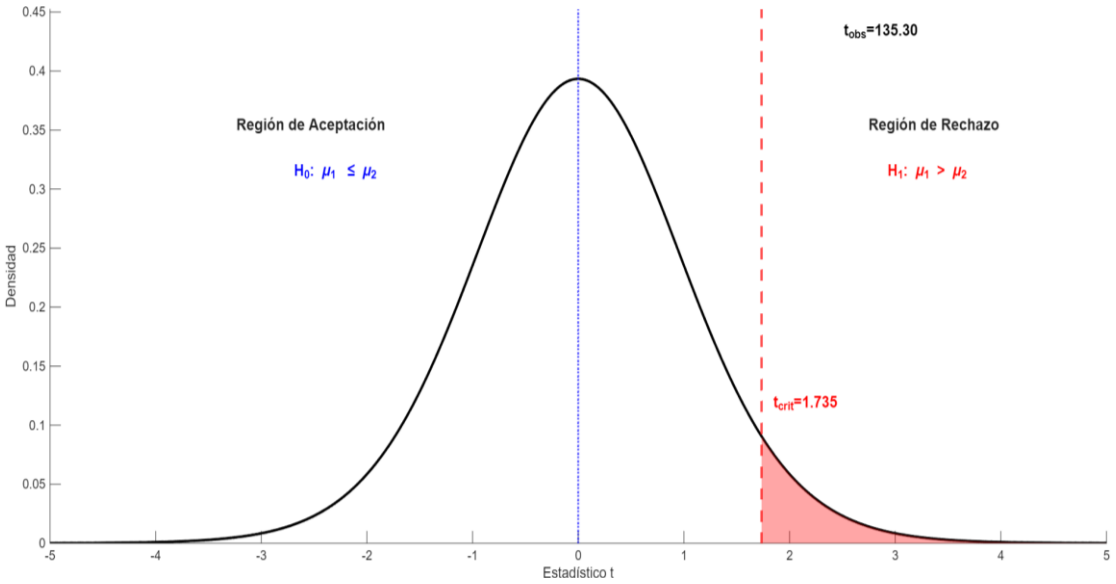

Figura 7.33. Prueba de hipótesis de dos una cola para demostrar el mejor desempeño entre RF y MLP para MCyMOO.

El valor estadístico obtenido en la prueba fue t = 135.30 (Ecuación. 7.1), el cual se encuentra en región de rechazo; por lo tanto, se concluye que sí existen diferencia significativa entre el desempeño de RF y MLP.

En la Tabla 7.12 Resume los resultados de las pruebas *t-student* ampliadas a los modelos RF vs KNN, SVM, NB y MLP.

Tabla 7.12. Resultados de las pruebas t-student para comparar el mejor desempeño de los modelos de las MCyMOO.

Comparación	Estadístico	p-valor (<0.05)	Evidencia
RF vs KNN	21.977	3.848e-10	Significativa
RFvs NB	261.01	2.2e-16	Significativa
RFvs SVM	91.981	2.2e-16	Significativa
RFvs MLP	135.3	2.2e-16	Significativa

Como se puede observar al comparar el desempeño del modelo RF vs KNN, SVM, NB y MLP el p-valor es inferior a 0.05 por lo que se demuestra que el modelo RF sí tiene mejor desempeño en la detección de defectos de software que los otros modelos.

Capítulo 8: Conclusiones, Contribuciones y trabajo futuro

8.1 Conclusiones

Durante el desarrollo de esta tesis se obtuvieron aprendizajes relevantes como:

- Se identifica que las métricas utilizadas para la predicción de defectos son diferentes por cada lenguaje de programación, por lo tanto, la selección de métricas debe alinearse al paradigma de programación.

- El preprocesamiento de los datos es una fase de suma importancia ya que la calidad influye directamente con el desempleo de los modelos.

- Las métricas de evaluación de modelos describen únicamente un parte del desempeño del modelo, por lo que es imposible interpretar los resultadas a partir de una sola métrica, por lo que se comprende que estas métricas son complementarias para una buena interpretación de resultados.

- La comparación entre distintos algoritmos mostró que dependen del contexto del dataset, la distribución de las clases y los parámetros de ajuste de cada modelo.

- Existen diferentes tipos de métricas orientadas a objetos, diseñadas para aspectos específicos, enfocadas en medir la estructura interna del código como la complejidad, cohesión, herencias, tamaño entre otras. Por lo que la selección deber ser de acuerdo con el objetivo del análisis.

- Las pruebas estadísticas aplicadas permitieron determinar si las diferencias observadas entre los modelos eran significativas. Los resultados observados respaldan que el desempeño reportado de acuerdo con los

hiperparámetros utilizados, confirman que los resultados del capítulo 6 no solo son los mejores valores si no que son validados estadísticamente.

Esta tesis presenta un análisis comparativo de diversos algoritmos tradicionales de aprendizaje automático entrenados con métricas de software. Se llevaron a cabo tres experimentos distintos, entrenando los algoritmos con métricas de complejidad del software, con métricas orientadas a objetos y con la combinación de ambas.

Este estudio permitió confirmar que la aplicación de algoritmos de aprendizaje automático es de mucha utilidad, para la predicción de defectos de software.

Los resultados demostraron que el modelo RF presenta un mejor desempeño en la predicción de defectos, por otro lado, el análisis estadístico validó que los resultados fueran comprobados significativamente entre los modelos.

- El experimento con mejor desempeño es la combinación de métricas de complejidad y métricas orientadas a objetos, el con las siguientes métricas: WMC, DIT, NOC, CBO, RFC, LCOM, LCOM3, MAX. CC, AVG, CC, CA, CE, NPM, LOC, DAM, MOA, MFA, CAM, IC, CBM, AMC. El mejor resultado presenta con RF con una exactitud de 87.23%, precisión de 85.52%, un *Recall* de 87.85%, F1 de 86.67%.

- El modelo con peor desempeño fue NB, presentando valores con exactitud de 62%, precisión de 62%, *Recall* 87%, F1, 86% en los tres experimentos.

- En los tres experimentos realizados se aplicaron pruebas estadísticas para comparar el desempeño de los modelos.

- El experimento llevado a cabo ambos grupos de métricas (Métricas de complejidad y métricas orientadas a objetos) el modelo RF obtuvo los

mejores resultados; en cambio en el segundo y tercer experimento, el modelo KNN fue el que mostró un desempeño superior.

- Al comparar globalmente los resultados de los tres experimentos, se observó que el experimento llevado a cabo con la combinación de métricas alcanzó un mejor desempeño con el modelo RF.

8.2 Contribuciones

En esta sección se presentan las contribuciones de esta investigación.

Yaretxy Arely Pelayo Lomeli, Juan Pablo García Vázquez, María Angélica Astorga Vargas, Jorge Eduardo Ibarra Esquer, Jesús Eduardo Soto Vega, Araceli Celina Justo López (2025) Springer Nature, ***Artificial Intelligence and Quantum Computing: Early Innovations***, Capítulo: ***Comparative Evaluation of Machine Learning Algorithms for Predicting Defects Using Object-Oriented and Software Complexity Metrics*.**

Este capítulo de libro (Artificial Intelligence and Quantum Computing: Early Innovations. Volume 1 | SpringerLink, 2025) presenta un análisis comparativo de varios algoritmos tradicionales de aprendizaje automático, tales como KNN, SVM, RF y MLP. Estos fueron entrenados con diversas métricas de software tales como métricas de complejidad, métricas orientadas a objetos y la combinación de ambas. Los resultados afirman que la combinación de estas métricas permite predecir defectos de software. Donde el mejor desempeño fue KNN con precisión de 80.26%.

Yaretxy Arely Pelayo Lomelí, Juan Pablo García Vázquez, María Angélica Astorga Vargas, Jesús Eduardo Soto Vega (2024). ***Detección de errores en pruebas estructurales de software mediante aprendizaje automático***, Escuela-Congreso Bayes Plurinacional 2024.

En esta contribución se presentaron las métricas orientadas a objetos y métricas de complejidad con técnicas de aprendizaje automático, donde se aplicó la metodología *CRISP-DM*, y se incorporó la detección de datos atípicos,

se analizó el desempeño de los algoritmos SVM, KNN y NB demostrando el mejor desempeño con una precisión de 86% en KNN.

8.3 Trabajo Futuro

Con el objetivo de generar modelos más robustos, se propone:

- Investigar si las métricas utilizadas para la predicción de defectos en Java son aplicables y efectivas en software desarrollado en otros lenguajes orientados a objetos, como Python, C++ y C, entre otros.
- Ampliar el conjunto de métricas y realizar un proceso de selección de características con el fin de crear modelos generalizables.
- Investigar las limitaciones de las métricas de software para determinar la complejidad del software.
- Utilizar técnicas de aprendizaje profundo tales como redes neuronales profundas, redes recurrentes para análisis de código.
- Probar modelos de ensamble (p. ej., stacking de clasificadores) para combinar fortalezas de varios algoritmos.
- Crear un dataset propio con proyectos de diferentes lenguajes de programación.

Referencias

Ahmad Imran. (2024). *50 algoritmos que todo programador debe conocer*. Marcombo. ISBN 978-84-267-3839-4

Albattah, W., & Alzahrani, M. (2024). Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach. *AI*, *5*(4), Article 4. https://doi.org/10.3390/ai5040086

Ali, E. K., Eissa, M. M., & Omara, A. F. (2023). A Review for Software Defect Prediction Using Machine Learning Algorithms. In D. Magdi, A. A. El-Fetouh, M. Mamdouh, & A. Joshi (Eds.), *Green Sustainability: Towards Innovative Digital Transformation* (pp. 219–231). Springer Nature. https://doi.org/10.1007/978-981-99-4764-5_14

Applitools. (2025). AI-Powered End-to-End Testing | Applitools. https://applitools.com/

Appvance. (2025). *AI-Driven Autonomous Software Testing Tools* [Sitio web].

Artificial Intelligence and Quantum Computing: Early Innovations. Volume 1 | SpringerLink. (2025). https://link.springer.com/book/10.1007/978-3-031-85614-3

Appvance de https://appvance.ai/

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17. https://doi.org/10.1109/32.979986

Baqar, M., & Khanda, R. (2024). The Future of Software Testing: AI-Powered Test Case Generation and Validation. arXiv. https://arxiv.org/abs/2409.05808

Betancourt, G. (2005). Las máquinas de soporte vectorial (SVMs). *Scientia Et Technica*.

Beurs, E., Boehnke, J. R., & Fried, E. I. (2022). *Common measures or common metrics? A plea to harmonize measurement results*. *29*(5), 1755–1767.

Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4). Springer. https://link.springer.com/book/9780387310732

Borrero, I. P., & Arias, M. E. G. (2021). *DEEP LEARNING*. Servicio de Publicaciones de la Universidad de Huelva.

Caicedo B, E. F., & López S, J. A. (2009). *Una aproximación práctica a las redes neuronales artificiales* (1st ed.). Programa Editorial Universidad del Valle. https://doi.org/10.25100/peu.64

Canaparo, M., Ronchieri, E., & Bertaccini, G. (2022). Software defect prediction: A study on software metrics using statistical and machine learning methods.

International Symposium on Grids & Clouds 2022, 20. https://inspirehep.net/files/45a01d78235645e57b8131d00f77fd52

Capote, T. D. (2024). A comparative study of black box and white box testing techniques in modern software development. *Frontiers in Engineering and Technology*, *5*. https://iaeme.com/Home/issue/FET?Volume=5&Issue=1

Cass, S. (2020). The top programming languages: Our latest rankings put Python on top-again-[Careers]. *IEEE Spectrum*, *57*(8), 22–22.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, *36*(4), 7346–7354 https://doi.org/10.1016/j.eswa.2008.10.027

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, *16*, 321–357. https://doi.org/10.1613/jair.953

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493 https://doi.org/10.1109/32.295895

Clark, A. (2018). The Machine Learning Audit-CRISP-DM Framework. *ISACA*. https://www.isaca.org/resources/isaca-journal/issues/2018/volume-1/the-machine-learning-auditcrisp-dm-framework

Cristiá, M. (2017). Introducción al Testing de Software. *ResearchGate*. https://www.researchgate.net/publication/316595764_Introduccion_al_Testing_de_Software

Cuevas, E., Avalos, O., Díaz, P., Valdivia, A., & Pérez, M. (2022). *Introducción al machine Learning con Matlab*. https://libroweb.alfaomega.com.mx/book/50_algoritmos_que_todo_programador_debe_conocer

Dash, S., Nayak, S., & Mishra, D. (2021). *A Review on Machine Learning Algorithms* (pp. 495–507). https://doi.org/10.1007/978-981-15-6202-0_51

Deshpande, B., Kumar, B., & Kumar, A. (2020). Object oriented design metrics for software defect prediction: An empirical study. *TEST Engineering and Management*, *83*(1), 10501–10518.

Duda, R. O., Hart, P. E., & Stork, D. G. (2012). *Pattern Classification*. John Wiley & Sons.

Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, *68*(3), 1189–1212.

Functionize. (2025). https://www.functionize.com

García, J. F. M. (2019). *Una metodología de proyectos de analítica de datos para las PYME.* [Master's thesis, Universidad de Medellín]. https://www.studocu.com/pe/document/universidad-tecnologica-del-peru/tecnologia-del-concreto/metodologia-de-proyectos-de-analitica-de-datos-para-pyme-mis-517/126956443

Gholami, R., & Fakhari, N. (2017). Chapter 27 - Support Vector Machine: Principles, Parameters, and Applications. In P. Samui, S. Sekhar, & V. E. Balas (Eds.), *Handbook of Neural Computation* (pp. 515–535). Academic Press. https://doi.org/10.1016/B978-0-12-811318-9.00027-2

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. https://www.deeplearningbook.org/

He, H., & Garcia, E. A. (2009). Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering*, *21*(9), 1263–1284. https://doi.org/10.1109/TKDE.2008.239

IEEE Standard Glossary of Software Engineering Terminology. (1990). *IEEE Std 610.12-1990*, 1–84. IEEE Std 610.12-1990. https://doi.org/10.1109/IEEESTD.1990.101064

IEEE Std 610.12-1990—IEEE Standard Glossary of Software Engineering Terminology | SE Goldmine. (n.d.). Retrieved March 25, 2025, from https://segoldmine.ppi-int.com/node/45375

Kanth, R., Guru, R., K, M. B., & Akshaya, D. V. S. (2024). AI Vs. Conventional Testing: A Comprehensive Comparison Of Effectiveness &Efficiency. *Educational Administration: Theory and Practice*, *30*(1), Article 1. https://doi.org/10.53555/kuey.v30i1.7495

Karhu, K., Kasurinen, J., & Smolander, K. (2025). *Expectations vs Reality—A Secondary Study on AI Adoption in Software Testing* (No. arXiv:2504.04921). arXiv. https://doi.org/10.48550/arXiv.2504.04921

Keysight. (2025). *Eggplant Test*. Keysight. https://www.keysight.com/us/en/products/software/software-testing/eggplant-test.html

Khleel, N., & Nehéz, K. (2021). Comprehensive Study on Machine Learning Techniques for Software Bug Prediction. *International Journal of Advanced Computer Science and Applications*, *12*, 2021. https://doi.org/10.14569/IJACSA.2021.0120884

Kim, K. G. (2016). Book Review: Deep Learning. *Healthcare Informatics Research*, *22*(4), 351. https://doi.org/10.4258/hir.2016.22.4.351

Kirch, W. (2008). *Z-Score*. Springer. https://link.springer.com/rwe/10.1007/978-1-4020-5614-7_3826

Kirk, M. (2017). *Thoughtful machine learning with Python: A test-driven approach*. O'Reilly Media, Inc. https://books.google.es/books?id=nG3vDQAAQBAJhttps://books.google.es/books?id=nG3vDQAAQBAJ

Lantz, B. (2013). *Machine Learning with R*. Packt Publishing.

https://www.researchgate.net

Lu, Y., Ye, T., & Zheng, J. (2022). Decision tree algorithm in machine learning. *2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*, 1014–1017. https://ieeexplore.ieee.org/abstract/document/9918857/

Mabl. (2025). *AI-Driven test automation platform for web, mobile, and APIs* [Sitioweb]. https://www.mabl.com/

Madeyski, L., & Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, *23*(3), 393–422. https://doi.org/10.1007/s11219-014-9241-7

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

Microsoft. (2025). *Testing tools in Visual Studio* [Documentación]. Recuperado de https://learn.microsoft.com/en-us/visualstudio/test/ Microsoft Learn+1

Mehmood, I., Shahid, S., Hussain, H., Khan, I., Ahmad, S., Rahman, S., Ullah, N., & Huda, S. (2023). A Novel Approach to Improve Software Defect Prediction Accuracy Using Machine Learning. *IEEE Access*, *11*, 63579–63597. https://doi.org/10.1109/ACCESS.2023.3287326

Mendenhall, W., Beaver, R. J., & Beaver, B. M. (2009). *Introduction to probability and statistics* (13. ed., international student ed). Brooks/Cole, Cengage Learning.

Mohamed, F. A., Salama, C. R., Yousef, A. H., & Salem, A. M. (2020). A universal model for defective classes prediction using different object-oriented metrics suites. *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, 65–70. https://ieeexplore.ieee.org/abstract/document/9257892/

Naidu, G., Zuva, T., & Sibanda, E. M. (2023). A Review of Evaluation Metrics in Machine Learning Algorithms. In R. Silhavy & P. Silhavy (Eds.), *Artificial Intelligence Application in Networks and Systems* (pp. 15–25). Springer International Publishing. https://doi.org/10.1007/978-3-031-35314-7_2

Ndung'u, R. (2022). Data Preparation For Machine Learning Modelling. *International Journal of Computer Applications Technology and Research*, *11*, 231–235. https://doi.org/10.7753/IJCATR1106.1008

Nelson, P. C. J. (2020). *Introducción a la ciencia de datos en R: Un enfoque práctico*. Editorial Universidad Distrital Francisco José de Caldas.

Olivas, E. S., Isla, M. A. S.-M., Cruz, R. G., & Caballer, B. C. (2023). *Sistemas de Aprendizaje Automático*. Ra-Ma Editorial.

Ouellet, A., & Badri, M. (2024). Combining object-oriented metrics and centrality measures to predict faults in object-oriented software: An empirical validation. *Journal of Software: Evolution and Process*, *36*(4), e2548. https://doi.org/10.1002/smr.2548

Pisner, D., & Schnyer, D. M. (2020). Support vector machine. In *Machine Learning*. Elsevier. https://doi.org/10.1016/B978-0-12-815739-8.00006-7

Ponce, O. E. R. (n.d.). *Tema 4: Métricas de Evaluación*.

Ponnala, R., & Reddy, D. (2021). Software Defect Prediction using Machine Learning Algorithms: Current State of the Art. *Solid State Technology*, *64*, 6541–6556.

Ranorex. (2021, November 11). https://www.ranorex.com/

Rebro, D. A., Rossi, B., & Chren, S. (2023). *Source Code Metrics for Software Defects Prediction* (No. arXiv:2301.08022). arXiv https://doi.org/10.48550/arXiv.2301.08022

Ricca, F., García, B., Nass, M., & Harman, M. (2025). Next-Generation Software Testing: AI-Powered Test Automation. *IEEE Software*, *42*(4), 25–33. https://doi.org/10.1109/MS.2025.3559194

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

RStudio. (2024). [Computer software]. Posit Software, PBC (o RStudio Team). https://www.posit.co/

Saeed, M. S., & Saleem, M. (2023). Cross project software defect prediction using machine learning: A review. *International Journal of Computational and Innovative Sciences*, *2*(3), 35–52.

Sayad, S. (2010). *Naive Bayesian*. http://saedsayad.com/naive_bayesian.htm

Schröer, C., Kruse, F., & Marx Gómez, J. (2021). A Systematic Literature Review on Applying CRISP-DM Process Model. *Procedia Computer Science*, *181*, 526–534. https://doi.org/10.1016/j.procs.2021.01.199

Selenium. (2025). *Selenium – Browser automation* [Sitio web]. Recuperado de https://www.selenium.dev/

Selvadurai, K., Uthariaraj, V., Sankaranarayanan, V., & Thambidurai, P. (2007). Object-oriented software prediction using neural networks. *Information and Software Technology*, *49*, 483–492. https://doi.org/10.1016/j.infsof.2006.07.005

Singh, M., & Chhabra, J. K. (2024). Machine learning based improved cross-project software defect prediction using new structural features in object oriented software. *Applied Soft Computing*, *165*, 112082.

Singh, S. K., & Singh, A. (2012). *Software testing* (1st ed.). Vandana Publications. https://books.google.es/books?hl=es&lr=&id=HdKwDwAAQBAJ&oi=fnd&pg=PT3&dq=Singh,+S.K.,+Singh,+A.:+Software+Testing,+1st+edn.+Vandana+Publications,+Delhi,+India+(2012)&ots=7agFj_sDa-&sig=nOYJ1_DK7oN0CPG-ceTA_JOEh0w

Sun, Y., Wong, A., & Kamel, M. S. (2011). Classification of imbalanced data: A review. *International Journal of Pattern Recognition and Artificial Intelligence*, *23*(04). https://doi.org/10.1142/S0218001409007326

Swaminathan, S., & Tantri, B. R. (2024). Confusion Matrix-Based Performance Evaluation Metrics. *African Journal of Biomedical Research*, *27*. https://doi.org/10.53555/AJBR.v27i4S.4345

Tadapaneni, P., Nadella, N. C., Divyanjali, M., & Sangeetha, Y. (2022). Software Defect Prediction based on Machine Learning and Deep Learning. *2022 International Conference on Inventive Computation Technologies (ICICT)*, 116–122. https://doi.org/10.1109/ICICT54344.2022.9850643

SmartBear Software. (2025). *TestComplete – Automated UI testing for desktop, web, and mobile applications* [Sitio web]. Recuperado de https://smartbear.com/product/testcomplete/

Testim. (2025). *AI-driven testing for Salesforce, web, and mobile* [Sitio web]. Recuperado de https://www.testim.io/

Umar, M. A. (2020). *A Study of Software Testing: Categories, Levels, Techniques, and Types*. https://doi.org/10.36227/techrxiv.12578714.v1

Unified Functional Testing. (2025). https://www.microfocus.com/documentation/silk-central/200/en/silkcentral-help-en/GUID-531809BA-688F-41D5-BDB2-FCE786A284CE.html

Wirth, R., & Hipp, J. (2000). CRISP-DM: Towards a standard process model for data mining. *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining*.

Yates, L. A., Aandahl, Z., Richards, S. A., & Brook, B. W. (2023). Cross validation for model selection: A review with examples from ecology. *Ecological Monographs*, *93*(1), e1557. https://doi.org/10.1002/ecm.1557

Zacharski, R. (2015). *A Programmer's Guide to Data Mining*. http://guidetodatamining.com/

Anexos

Anexo A1. Resultados de métricas de complejidad

KNN Distancia *Euclidiana*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.8377825	0.7827739	0.7735893	0.7728294
2	0.8850932	0.7861107	0.7793049	0.7693604
3	0.8351725	0.7878287	0.7810229	0.7653297
4	0.8389388	0.786375	0.7762984	0.7728624
5	0.8389388	0.7868376	0.779371	0.7719043
6	0.8362958	0.7882582	0.7770252	0.7684683
7	0.8417801	0.7824435	0.7750099	0.7715079
8	0.8413176	0.7840954	0.7776199	0.7729285
9	0.8403264	0.7824435	0.7751421	0.7729615
10	0.8388727	0.7857804	0.7767609	0.7666512
promedio	**0.84345184**	0.78529469	0.77711445	0.77048036
varianza	0.000218302479	0.000004856030361	0.000005218492081	0.000008102161423
ds	0.01477506274	0.002140205847	0.002035896722	0.002889384753

KNN Distancia *Manhattan*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.7963856	0.7835998	0.7720034	0.7686005
2	0.7985992	0.7835998	0.7713757	0.7665852
3	0.7955597	0.7835998	0.7749438	0.7686005
4	0.7968482	0.7835998	0.7724329	0.7721356
5	0.796749	0.7835998	0.7738205	0.7686335
6	0.7948989	0.7879278	0.7724329	0.7721356
7	0.7942712	0.7825757	0.7755055	0.7672129
8	0.7953614	0.7826748	0.7762984	0.7699220
9	0.7946676	0.7848883	0.7765627	0.7697899
10	0.7953945	0.786342	0.7726972	0.7674442
promedio	**0.79587353**	0.78424076	0.7738073	0.76910599
varianza	0.000001663310038	0.000002856398409	0.000003572278289	0.000003658298197
ds	0.001354549721	0.001776622787	0.001888613503	0.00201992489

KNN Distancia *Minkowski*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.7729285	0.7859456	0.7754064	0.7694265
2	0.7982358	0.7874983	0.7728294	0.7771904
3	0.7666512	0.7865072	0.7736223	0.7693273
4	0.7987974	0.7813532	0.7754064	0.7697568
5	0.7965839	0.7796022	0.7741509	0.7656931
6	0.7926523	0.7839302	0.7771904	0.7694265
7	0.7965508	0.7805934	0.7730937	0.7685344
8	0.7961213	0.7807586	0.7719704	0.7686005
9	0.797542	0.7815515	0.7771904	0.7701533
10	0.7966169	0.7869037	0.7719704	0.7701533
promedio	**0.79126801**	0.7834643900	0.77428307	0.76982621
varianza	0.0001330197319	0.000009147780439	0.000003789733829	0.000008363585728
ds	0.01014562167	0.003071839353	0.002021927495	0.003063795674

Naïve Bayes	
k-folds	**Accuracy**
1	0.6212171
2	0.6242897
3	0.6248183
4	0.6214484
5	0.6311286
6	0.6235959
7	0.6238271
8	0.6247192
9	0.6250496
10	0.624587
promedio	**0.62446809**
varianza	**7.31014E-06**
ds	**0.002599230613**

MLP	
k-folds	**Accuracy**
1	0.654687
2	0.664588
3	0.678954
4	0.6722565
5	0.6756216
6	0.6815451
7	0.6848486
8	0.6859876
9	0.692511
10	0.6857122
promedio	**0.63172893**
varianza	**0.00004625664608**
ds	**0.006308743382**

Random Forest			
k-folds	RF - ntree 500	RF - ntree 200	RF - ntree 100
	Accuracy	**Accuracy**	**Accuracy**
1	0.8008458	0.8417801	0.8398969
2	0.8012753	0.8371878	0.8815911
3	0.8017708	0.8428704	0.8832761
4	0.7994912	0.8392031	0.8394013
5	0.8006145	0.8428704	0.8392692
6	0.7973768	0.8401943	0.8365601
7	0.801077	0.8386084	0.8358332
8	0.7931148	0.840789	0.8357341
9	0.7982027	0.8397978	0.8384432
10	0.8017378	0.8424409	0.8396987
promedio	**0.79955067**	**0.84057422**	**0.84697039**
varianza	**7.31026E-06**	**3.72101E-06**	**3.51954E-04**
ds	**0.002826848969**	**0.001996038701**	**0.01972306399**

Support Vector Machine			
k-folds	**Accuracy**	**Accuracy**	**Accuracy**
1	0.641116	0.641400	0.608750
2	0.634589	0.643780	0.602284
3	0.634848	0.643150	0.609450
4	0.634952	0.643330	0.602040
5	0.625889	0.643780	0.600246
6	0.6424589	0.643140	0.607314
7	0.624893	0.643960	0.602548
8	0.624789	0.643300	0.675781
9	0.6289614	0.643790	0.608245
10	0.624793	0.643450	0.607854
promedio	**0.63172893**	**0.643308**	**0.612451**
varianza	**0.00004625664608**	**0.0000005352622222**	**0.0005063769066**
ds	**0.006308743382**	**0.0003107249588**	**0.02382795412**

Anexo A2. Resultados de métricas orientadas a objetos

KNN Distancia *Euclidiana*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.8377825	0.7827739	0.7735893	0.7728294
2	0.8850932	0.7861107	0.7793049	0.7693604
3	0.8351725	0.7878287	0.7810229	0.7653297
4	0.8389388	0.786375	0.7762984	0.7728624
5	0.8389388	0.7868376	0.779371	0.7719043
6	0.8362958	0.7882582	0.7770252	0.7684683
7	0.8417801	0.7824435	0.7750099	0.7715079
8	0.8413176	0.7840954	0.7776199	0.7729285
9	0.8403264	0.7824435	0.7751421	0.7729615
10	0.8388727	0.7857804	0.7767609	0.7666512
promedio	**0.84345184**	0.78529469	0.77711445	0.77048036
varianza	0.000218302479	0.00000485603061	0.000005218492081	0.000008102161423
ds	0.01477506274	0.002140205847	0.002035896722	0.002889384753

KNN Distancia *Manhattan*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.7963856	0.7835998	0.7720034	0.7686005
2	0.7985992	0.7835998	0.7713757	0.7665852
3	0.7955597	0.7835998	0.7749438	0.7686005
4	0.7968482	0.7835998	0.7724329	0.7721356
5	0.796749	0.7835998	0.7738205	0.7686335
6	0.7948989	0.7879278	0.7724329	0.7721356
7	0.7942712	0.7825757	0.7755055	0.7672129
8	0.7953614	0.7826748	0.7762984	0.7699220
9	0.7946676	0.7848883	0.7765627	0.7697899
10	0.7953945	0.786342	0.7726972	0.7674442
promedio	**0.79587353**	0.78424076	0.7738073	0.76910599
varianza	0.000001663310038	0.000002856398409	0.000003572278289	0.000003658298197
ds	0.001354549721	0.001776622787	0.001888613503	0.00201992489

KNN Distancia *Minkowski*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.7729285	0.7859456	0.7754064	0.7694265
2	0.7982358	0.7874983	0.7728294	0.7771904
3	0.7666512	0.7865072	0.7736223	0.7693273
4	0.7987974	0.7813532	0.7754064	0.7697568
5	0.7965839	0.7796022	0.7741509	0.7656931
6	0.7926523	0.7839302	0.7771904	0.7694265
7	0.7965508	0.7805934	0.7730937	0.7685344
8	0.7961213	0.7807586	0.7719704	0.7686005
9	0.797542	0.7815515	0.7771904	0.7701533
10	0.7966169	0.7869037	0.7719704	0.7701533
promedio	**0.79126801**	0.7834643900	0.77428307	0.76982621
varianza	0.0001330197319	0.000009147780439	0.000003789733829	0.000008363585728
ds	0.01014562167	0.003071839353	0.002021927495	0.003063795674

Naïve Bayes	
k-folds	**Accuracy**
1	0.6212171
2	0.6242897
3	0.6248183
4	0.6214484
5	0.6311286
6	0.6235959
7	0.6238271
8	0.6247192
9	0.6250496
10	0.624587
promedio	**0.62446809**
varianza	**7.31014E-06**
ds	**0.002599230613**

MLP	
k-folds	**Accuracy**
1	0.654687
2	0.664588
3	0.678954
4	0.6722565
5	0.6756216
6	0.6815451
7	0.6848486
8	0.6859876
9	0.692511
10	0.6857122
promedio	**0.63172893**
varianza	**0.00004625664608**
ds	**0.006308743382**

Support Vector Machine			
K-folds	**Accuracy**	**Accuracy**	**Accuracy**
1	0.641116	0.641400	0.608750
2	0.634589	0.643780	0.602284
3	0.634848	0.643150	0.609450
4	0.634952	0.643330	0.602040
5	0.625889	0.643780	0.600246
6	0.6424589	0.643140	0.607314
7	0.624893	0.643960	0.602548
8	0.624789	0.643300	0.675781
9	0.6289614	0.643790	0.608245
10	0.624793	0.643450	0.607854
promedio	**0.63172893**	**0.643308**	**0.612451**
varianza	**0.00004625664608**	**0.0000005352622222**	**0.0005063769066**
ds	**0.006308743382**	**0.0003107249588**	**0.02382795412**

Anexo A3. Resultados de métricas de complejidad y métricas orientadas a objetos.

KNN Distancia *Euclidiana*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.8340492	0.822684	0.8140611	0.8081803
2	0.8338509	0.822684	0.8143254	0.8078829
3	0.8343795	0.8222876	0.8143584	0.8082133
4	0.8340492	0.8228162	0.8145566	0.8082794
5	0.8340822	0.8229483	0.8144245	0.8076516
6	0.8345117	0.8221884	0.8143914	0.8082463
7	0.8350403	0.8225849	0.8141932	0.8081472
8	0.8333554	0.8228823	0.8146557	0.8076847
9	0.8340822	0.8224197	0.8145897	0.8086428
10	0.8345447	0.8227831	0.8143254	0.8085767
promedio	**0.83419453**	0.82262785	0.81438814	0.80815052
varianza	0.000000206939449	0.000000065263825	0.00000003284889822	0.0000001102357729
ds	0.0004794511805	0.0002701553463	0.0001486611144	0.0003519830551

KNN Distancia *Manhattan*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.8374521	0.8267808	0.8206026	0.8115832
2	0.8379146	0.8249967	0.8250958	0.8139
3	0.8422096	0.8230474	0.8194793	0.8081142
4	0.839071	0.8250628	0.8199749	0.8141271
5	0.839137	0.8206357	0.8187525	0.8108233
6	0.8433329	0.8287961	0.820966	0.8129047
7	0.8374191	0.8343795	0.8220233	0.8160103
8	0.8367913	0.8272433	0.8191489	0.8123431
9	0.838212	0.8284987	0.8201731	0.8089401
10	0.840855	0.8257235	0.8194463	0.8144245
promedio	**0.83923946**	0.82651645	0.82056627	0.81232655
varianza	0.000004837734525	0.00001369121143	0.000003448835491	0.000006261767912
ds	0.002235788262	0.003923379561	0.001969709779	0.002639648305

KNN Distancia *Minkowski*				
k-folds	**k3**	**k5**	**k7**	**k9**
1	0.8335205	0.8216268	0.8136976	0.810592
2	0.8310096	0.8223206	0.8128386	0.8090062
3	0.8324964	0.8212634	0.810526	0.8031254
4	0.8307453	0.8184221	0.8074865	0.8054381
5	0.83395	0.820933	0.807949	0.8020682
6	0.8350073	0.815713	0.8090723	0.80187
7	0.833025	0.8175301	0.8118475	0.8043478
8	0.8331241	0.816638	0.8094027	0.8048764
9	0.8358993	0.8194793	0.8090062	0.8002841
10	0.8328267	0.816638	0.810592	0.7997886
promedio	**0.83316042**	0.81905643	0.81024184	0.8034227556
varianza	0.000002521811575	0.000005715649736	0.000004212749154	0.000008199723835
ds	0.001678996731	0.002347870221	0.001755190357	0.002863515992

Naïve Bayes	
K-folds	**Accuracy**
1	0.6330448
2	0.6290056
3	0.6330448
4	0.6315581
5	0.6317563

MLP	
K-Folds	**Accuracy**
1	0.7296783
2	0.729852
3	0.7299457
4	0.7298456
5	0.7298478

6	0.6283864
7	0.6305009
8	0.6288489
9	0.6287498
10	0.6287498
promedio	**0.63036454**
varianza	**3.44661E-06**
ds	**0.001696962754**

6	0.729647
7	0.7258756
8	0.72598
9	0.729846
10	0.724889
promedio	**0.7285407**
varianza	**0.000006688453725**
ds	**0.002381611643**

Random Forest			
k-folds	**RF - ntree 500**	**RF - ntree 200**	**RF - ntree 100**
	Accuracy	**Accuracy**	**Accuracy**
1	0.873728	0.873695	0.8723735
2	0.8725717	0.8723074	0.8710663
3	0.8727038	0.8729351	0.8733646
4	0.8735298	0.8740914	0.8726047
5	0.8734968	0.8733646	0.8720761
6	0.8731664	0.872836	0.8718779
7	0.8737611	0.872869	0.8723735
8	0.8730342	0.8724726	0.8724395
9	0.8734307	0.8726047	0.8739263
10	0.8729682	0.8735298	0.8722743
promedio	**0.87323907**	**0.87307056**	**0.87243767**
varianza	**1.72418E-07**	**3.33377E-07**	**6.06975E-07**
ds	**0.0004009589384**	**0.0005664736639**	**0.0008259995369**

Support Vector Machine			
k-folds	**Lineal**	**Radial**	**Polynomial**
	Accuracy	**Accuracy**	**Accuracy**
1	0.650459	0.6330448	0.605874
2	0.650465	0.6290056	0.607589
3	0.65046	0.6330448	0.606589
4	0.650464	0.6315581	0.601286
5	0.650484	0.6317563	0.604895
6	0.650487	0.6283864	0.605548
7	0.650469	0.6305009	0.605781
8	0.650475	0.6288489	0.605478
9	0.650465	0.6287498	0.605904
10	0.650455	0.6287498	0.605752

promedio	**0.6504683**	**0.63036454**	**0.6054696**
varianza	**1.12678E-10**	**3.44661E-06**	**2.67920E-06**
ds	**0.00001061497893**	**0.001696962754**	**0.001636825057**

Anexo A4. conjunto de datos MJ12A

```
> head(MJ12A)
  Project Version                                                Class wmc dit noc cbo rfc lcom ca ce npm     lcom3 loc
1     ant     1.3                    org.apache.tools.ant.taskdefs.GenerateKey  14   3   0   6  33   65  0  6  14 0.9102564 404
2     ant     1.3 org.apache.tools.ant.taskdefs.compilers.CompilerAdapterFactory   3   1   0   8  16    3  1  7   1 2.0000000 125
3     ant     1.3                       org.apache.tools.ant.taskdefs.KeySubst   8   3   0   3  40   10  0  3   8 0.7142857 271
4     ant     1.3                org.apache.tools.ant.taskdefs.JavacOutputStream   4   2   0   1  11    0  0  1   1 0.2222222  68
5     ant     1.3                          org.apache.tools.mail.MailMessage  27   1   0   3  63  297  1  2  10 0.7788461 527
6     ant     1.3                        org.apache.tools.ant.taskdefs.Touch   8   3   0   7  32    8  3  4   5 0.7380952 300
> |
```

Consultar dataset aquí https://madeyski.e-informatyka.pl/tools/software-defect-prediction/

Anexo A5. Código para preprocesamiento de datos

Preprocesamiento de MJ12A.R

```
# creando un dataset temporal
# 24-11-2024

temp=MJ12A;

# dejando solo las métricas
temp.m=temp[,4:29]

#remover de temp.m 19 al 22

#metricas orientadas a objetos y métricas de complejidad
temp.m=subset(temp.m,
select=c(wmc,dit,noc,cbo,rfc,lcom,ca,ce,npm,lcom3,loc,dam,moa,mfa,cam,ic,cbm,amc,max.cc.,avg.cc
.,bugs))

#metricas complejidad
temp.m=subset(temp.m,
select=c(wmc,dit,noc,rfc,lcom,lcom3,loc,dam,mfa,max.cc.,avg.cc.,bugs))

#metricas POO
temp.m=subset(temp.m, select=c(cbo,ca,ce,npm,moa,cam,ic,cbm,amc,bugs))

# convirtiendo bus a factor
temp.m$bugs <- ifelse(temp.m$bugs == 0, "sin errores", "con errores")

write.csv(temp.m,file="temp_MJ12A_metricas.csv")
```

Anexo A6. Código de preprocesamiento: normalización

Normalizacion de MJ12A.R

```
# normalizando el dataset y particionando

# Suponiendo que la última columna es un factor
# Identificamos las columnas numéricas
numeric_columns=sapply(temp.m, is.numeric)

# Separamos las columnas numéricas y la columna de factor (si es la última)
numeric_data=temp.m[, numeric_columns]
factor_data=temp.m[, !numeric_columns]

# Normalizar solo las columnas numéricas con Z-score
numeric_data_normalized=scale(numeric_data)

# Aplicar Percentile Rank a las columnas numéricas normalizadas
percentile_rank=apply(numeric_data_normalized, 2, function(x) rank(x) / length(x))

# Convertir el resultado en un dataframe
percentile_rank_df=as.data.frame(percentile_rank)

# Recombinar las columnas numéricas normalizadas con la columna de factor
temp.m_normalized=cbind(percentile_rank_df, factor_data)

# Ver el resultado final
print(temp.m_normalized)

write.csv(temp.m_normalized, file="temp.m_normalized.csv")
```

Anexo A7. Entrenamiento del modelo **KNN**

Modelo KNN de MJ12A.R

```r
# Instalar y cargar las librerías necesarias
if (!require(class)) install.packages("class", dependencies = TRUE)
if (!require(caret)) install.packages("caret", dependencies = TRUE)

library(class)
library(caret)

# Función para verificar si dos números son coprimos
gcd <- function(a, b) {
  while (b != 0) {
    temp <- b
    b <- a %% b
    a <- temp
  }
  return(abs(a))
}
are_coprime <- function(a, b) {
  return(gcd(a, b) == 1)
}

# Definir los valores de k coprimos entre 3 y 11 con respecto a un número (ej., 12)
k_values <- 1:11
coprime_k <- k_values[sapply(k_values, function(k) are_coprime(k, 12))]

# Dividir los datos en X (predictoras) e y (variable objetivo)
X_train <- train_oo[, -ncol(train_oo)]  # Todas las columnas excepto la última
y_train <- train_oo[, ncol(train_oo)]   # La última columna
X_test <- test_oo[, -ncol(test_oo)]
y_test <- test_oo[, ncol(test_oo)]

# Asegurarse de que y_train y y_test son factores con los mismos niveles
y_train <- factor(y_train)
y_test <- factor(y_test, levels = levels(y_train))

# Configurar validación cruzada con caret
folds <- createFolds(y_train, k = 10, list = TRUE, returnTrain = TRUE)

# Inicializar lista para almacenar resultados
results <- list()
```

continuación Modelo KNN de MJ12A.R

```
# Probar cada valor de k
for (k in coprime_k) {
  accuracies <- c()

  for (fold in folds) {
    # Crear conjuntos de entrenamiento y validación
    fold_train_X <- X_train[fold, ]
    fold_train_y <- y_train[fold]
    fold_test_X <- X_train[-fold, ]
    fold_test_y <- y_train[-fold]

    # Aplicar KNN
    fold_predictions <- knn(train = fold_train_X, test = fold_test_X, cl = fold_train_y, k = k)
    fold_predictions <- factor(fold_predictions, levels = levels(fold_test_y))

    # Evaluar accuracy en la validación
    conf_matrix <- confusionMatrix(fold_predictions, fold_test_y)
    accuracies <- c(accuracies, conf_matrix$overall["Accuracy"])
  }

  # Promediar accuracy y guardar resultados
  mean_accuracy <- mean(accuracies)
  results[[paste("k =", k)]] <- list(CV_Accuracy = mean_accuracy)
}

# Evaluar el modelo con el conjunto de prueba para cada k
for (k in coprime_k) {
  predictions <- knn(train = X_train, test = X_test, cl = y_train, k = k)
  predictions <- factor(predictions, levels = levels(y_test))

  conf_matrix <- confusionMatrix(predictions, y_test)

    results[[paste("k =", k)]] <- c(
    results[[paste("k =", k)]],
    Test_Accuracy = conf_matrix$overall["Accuracy"],
    Precision = conf_matrix$byClass["Precision"],
    Recall = conf_matrix$byClass["Recall"],
    F1 = conf_matrix$byClass["F1"],
    Sensitivity = conf_matrix$byClass["Sensitivity"],
    Specificity = conf_matrix$byClass["Specificity"]
  )
}

# Mostrar resultados
print(results)
```

Anexo A8. Entrenamiento del modelo **NB**

Modelo NB de MJ12A.R

```
 # Cargar librerías necesarias
install.packages(c("e1071", "caret", "smotefamily"))  # Instalar si no están ya instaladas
library(e1071)          # Naive Bayes
library(caret)          # Validación cruzada y partición de datos
library(smotefamily) # SMOTE para balancear clases

# Asegurarse de que la clase 'y' es un factor
temp.m_normalized$factor_data <- as.factor(temp.m_normalized$factor_data)

# Verificar distribución original de las clases
cat("Distribución inicial de clases:\n")
print(prop.table(table(temp.m_normalized$factor_data)))

# Aplicar SMOTE para balancear las clases
smote_result <- SMOTE(
  X = temp.m_normalized[, -ncol(temp.m_normalized)],  # Todas las columnas excepto la última (la clase)
  target = temp.m_normalized$factor_data,                # Columna con la clase
  K = 5,                                                 # Vecinos más cercanos
  dup_size = 4.5                                         # Factor de aumento para balancear
)

# Extraer datos balanceados
balanced_data <- smote_result$data
balanced_data$class <- as.factor(balanced_data$class)  # Convertir clase a factor

# Verificar distribución después de SMOTE
cat("Distribución después de SMOTE:\n")
print(prop.table(table(balanced_data$class)))

# Dividir los datos balanceados en conjuntos de entrenamiento y prueba
set.seed(123)
train_index <- createDataPartition(balanced_data$class, p = 0.8, list = FALSE)
train_b <- balanced_data[train_index, ]
test_b <- balanced_data[-train_index, ]

# Dividir en X (predictoras) e y (variable objetivo)
X_train <- train_b[, -ncol(train_b)]  # Todas las columnas excepto la última
y_train <- train_b[, ncol(train_b)]   # La última columna (clase)
```

continuación Modelo NB de MJ12A.R

```
X_test <- test_b[, -ncol(test_b)]
y_test <- test_b[, ncol(test_b)]

# Asegurarse de que las etiquetas de entrenamiento y prueba tienen los mismos niveles
y_train <- factor(y_train)
y_test <- factor(y_test, levels = levels(y_train))

# Configurar validación cruzada estratificada
folds <- createFolds(y_train, k = 10, list = TRUE, returnTrain = TRUE)

# Inicializar lista para almacenar resultados
results <- list()

# Validación cruzada con Naive Bayes
for (fold in folds) {
  # Crear conjuntos de entrenamiento y validación
  fold_train_X <- X_train[fold, ]
  fold_train_y <- y_train[fold]
  fold_test_X <- X_train[-fold, ]
  fold_test_y <- y_train[-fold]

  # Entrenar modelo de Naive Bayes
  nb_model <- naiveBayes(fold_train_X, fold_train_y)

  # Realizar predicciones
  fold_predictions <- predict(nb_model, fold_test_X)

  # Evaluar accuracy y kappa en la validación
  conf_matrix <- confusionMatrix(fold_predictions, fold_test_y)

  # Almacenar resultados
  results <- append(results, list(
    CV_Accuracy = conf_matrix$overall["Accuracy"],
    CV_Kappa = conf_matrix$overall["Kappa"]
  ))
}

# Evaluar el modelo con el conjunto de prueba
nb_model_final <- naiveBayes(X_train, y_train)
predictions <- predict(nb_model_final, X_test)

# Evaluar las métricas en el conjunto de prueba
conf_matrix <- confusionMatrix(predictions, y_test)
conf_matrix
```

Anexo A9. Entrenamiento del modelo **SVM**

```
  # Cargar librerías necesarias
install.packages(c("e1071", "caret", "smotefamily"))  # Instalar si no están ya instaladas
library(e1071)          # SVM
library(caret)          # Validación cruzada y búsqueda de hiperparámetros
library(smotefamily)   # SMOTE para balancear clases

# Asegurarse de que la clase 'y' es un factor
temp.m_normalized$factor_data <- as.factor(temp.m_normalized$factor_data)

# Verificar distribución original de las clases
cat("Distribución inicial de clases:\n")
print(prop.table(table(temp.m_normalized$factor_data)))

# Aplicar SMOTE para balancear las clases
smote_result <- SMOTE(
  X = temp.m_normalized[, -ncol(temp.m_normalized)], # Todas las columnas excepto la última (la clase)
  target = temp.m_normalized$factor_data,              # Columna con la clase
  K = 5,                                               # Vecinos más cercanos
  dup_size = 4.5                                       # Factor de aumento para balancear
)

# Extraer datos balanceados
balanced_data <- smote_result$data
balanced_data$class <- as.factor(balanced_data$class)  # Convertir clase a factor
```

continuación Modelo SVM de MJ12A.R

```
# Asegurarse de que los niveles de la clase sean válidos nombres de variables en R
levels(balanced_data$class) <- make.names(levels(balanced_data$class))

# Verificar niveles después de la conversión
cat("Niveles de la clase después de make.names:\n")
print(levels(balanced_data$class))

# Verificar distribución después de SMOTE
cat("Distribución después de SMOTE:\n")
print(prop.table(table(balanced_data$class)))

# Dividir los datos balanceados en conjuntos de entrenamiento y prueba
set.seed(123)
train_index <- createDataPartition(balanced_data$class, p = 0.8, list = FALSE)
train_b <- balanced_data[train_index, ]
test_b <- balanced_data[-train_index, ]

# Dividir en X (predictoras) e y (variable objetivo)
X_train <- train_b[, -ncol(train_b)]  # Todas las columnas excepto la última
y_train <- train_b[, ncol(train_b)]   # La última columna (clase)
X_test <- test_b[, -ncol(test_b)]
y_test <- test_b[, ncol(test_b)]

# Asegurarse de que las etiquetas de entrenamiento y prueba tienen los mismos niveles
y_train <- factor(y_train)
y_test <- factor(y_test, levels = levels(y_train))

# Configurar control de entrenamiento para búsqueda de hiperparámetros
train_control <- trainControl(
  method = "cv",           # Validación cruzada
  number = 10,             # Número de folds
  classProbs = TRUE,    # Habilitar probabilidades para métricas como ROC
  summaryFunction = multiClassSummary # Métricas de evaluación
)

# Definir rango de hiperparámetros
grid <- expand.grid(
  C = 2^(-5:5),          # Valores de costo
  sigma = 2^(-5:5)       # Valores de sigma
)
```

continuación Modelo SVM de MJ12A.R

```
# Entrenar el modelo SVM con kernel radial
set.seed(123)
svm_model <- train(
  x = X_train,
  y = y_train,
  method = "svmRadial",     # SVM con kernel gaussiano
  trControl = train_control,
  tuneGrid = grid               # Rango de hiperparámetros
)

# Mostrar mejores parámetros
cat("Mejores hiperparámetros:\n")
print(svm_model$bestTune)

# Realizar predicciones en el conjunto de prueba
predictions <- predict(svm_model, X_test)

# Evaluar las métricas en el conjunto de prueba
conf_matrix <- confusionMatrix(predictions, y_test)

# Almacenar y mostrar resultados
results <- list(
  Best_Params = svm_model$bestTune,
  Test_Accuracy = conf_matrix$overall["Accuracy"],
  Test_Kappa = conf_matrix$overall["Kappa"],
  Precision = conf_matrix$byClass["Precision"],
  Recall = conf_matrix$byClass["Recall"],
  F1 = conf_matrix$byClass["F1"]
)

print(results)
```

Anexo A10. Entrenamiento del modelo **RF**

Modelo RF de MJ12A.R

```
# Cargar librerías necesarias
install.packages(c("randomForest", "e1071", "caret", "smotefamily"))  # Instalar si no están ya instaladas
library(randomForest)    # Random Forest
library(caret)           # Validación cruzada y partición de datos
library(smotefamily)     # SMOTE para balancear clases

# Asegurarse de que la clase 'y' es un factor
temp.m_normalized$factor_data <- as.factor(temp.m_normalized$factor_data)

# Verificar distribución original de las clases
cat("Distribución inicial de clases:\n")
print(prop.table(table(temp.m_normalized$factor_data)))

# Aplicar SMOTE para balancear las clases
smote_result <- SMOTE(
  X = temp.m_normalized[, -ncol(temp.m_normalized)],  # Todas las columnas excepto la última (la clase)
  target = temp.m_normalized$factor_data,          # Columna con la clase
  K = 5,   # Asegurarse de que las etiquetas de entrenamiento y prueba tienen los mismos niveles
  y_train <- factor(y_train)
  y_test <- factor(y_test, levels = levels(y_train))

# Configurar validación cruzada estratificada
folds <- createFolds(y_train, k = 10, list = TRUE, returnTrain = TRUE)

# Inicializar lista para almacenar resultados
results <- list()

# Validación cruzada con Random Forest
for (fold in folds) {
  # Crear conjuntos de entrenamiento y validación
  fold_train_X <- X_train[fold, ]
  fold_train_y <- y_train[fold]
  fold_test_X <- X_train[-fold, ]
  fold_test_y <- y_train[-fold]

  # Entrenar modelo de Random Forest
  rf_model <- randomForest(
    x = fold_train_X,
    y = fold_train_y,
    ntree = 500,                          # Número de árboles
    mtry = sqrt(ncol(fold_train_X))       # Variables a considerar en cada división
  )
```

continuación Modelo RF de MJ12A.R

```
# Asegurarse de que las etiquetas de entrenamiento y prueba tienen los mismos niveles
y_train <- factor(y_train)
y_test <- factor(y_test, levels = levels(y_train))

# Configurar validación cruzada estratificada
folds <- createFolds(y_train, k = 10, list = TRUE, returnTrain = TRUE)

# Inicializar lista para almacenar resultados
results <- list()

# Validación cruzada con Random Forest
for (fold in folds) {
  # Crear conjuntos de entrenamiento y validación
  fold_train_X <- X_train[fold, ]
  fold_train_y <- y_train[fold]
  fold_test_X <- X_train[-fold, ]
  fold_test_y <- y_train[-fold]

  # Entrenar modelo de Random Forest
  rf_model <- randomForest(
    x = fold_train_X,
    y = fold_train_y,
    ntree = 500,                          # Número de árboles
    mtry = sqrt(ncol(fold_train_X))       # Variables a considerar en cada división
  )

  # Realizar predicciones
  fold_predictions <- predict(rf_model, fold_test_X)

  # Evaluar accuracy y kappa en la validación
  conf_matrix <- confusionMatrix(fold_predictions, fold_test_y)

  # Almacenar resultados
  results <- append(results, list(
    CV_Accuracy = conf_matrix$overall["Accuracy"],
    CV_Kappa = conf_matrix$overall["Kappa"]
  ))
}

# Evaluar el modelo con el conjunto de prueba
rf_model_final <- randomForest(
  x = X_train,
  y = y_train,
  ntree = 500,                    # Número de árboles
  mtry = sqrt(ncol(X_train))      # Variables a considerar
)
```

continuación Modelo RF de MJ12A.R

```
predictions <- predict(rf_model_final, X_test)

# Evaluar las métricas en el conjunto de prueba
conf_matrix <- confusionMatrix(predictions, y_test)

# Almacenar resultados finales
results <- append(results, list(
  Test_Accuracy = conf_matrix$overall["Accuracy"],
  Test_Kappa = conf_matrix$overall["Kappa"],
  Precision = conf_matrix$byClass["Precision"],
  Recall = conf_matrix$byClass["Recall"],
  F1 = conf_matrix$byClass["F1"]
))

# Mostrar resultados
print(results)
```