



Casa abierta al tiempo

Universidad Autónoma Metropolitana Unidad Iztapalapa
División de Ciencias Básicas e Ingeniería
Posgrado en Ciencias (Ciencias y Tecnologías de la Información)

“Hacia un método para la estimación automática de esfuerzo en el desarrollo de software a partir de historias de usuario”

Tesis para obtener el grado de:
Maestro en Ciencias (Ciencias y Tecnologías de la Información)

Eduardo Rodríguez Sánchez

2202800353

erodsmx@gmail.com

ORCID: 0000-0002-7116-0568

Directores de tesis:

Dr. Humberto Gustavo Cervantes Maceda

Dr. Eduardo Filemón Vázquez Santacruz

Jurado:

Presidente: Dr. Rene Mac Kinney Romero

Secretario: Dr. Eduardo Filemón Vázquez Santacruz

Vocal: Dra. Sonia Guadalupe Mendoza Chapa

Vocal: Dr. Humberto Gustavo Cervantes Maceda

Iztapalapa, Ciudad de México, 10 de marzo de 2023

Plantilla de documento creada bajo la licencia [Creative Commons Attribution - Non-Commercial 3.0 Unported License](#). Adaptación correspondiente para cumplir con los requisitos del documento de ICR para el proyecto de investigación del Posgrado en Ciencias y Tecnologías de la Información (PCyTI). Parte del contenido presentado como algunas imágenes son de uso gratuito y se descargaron de los sitios web <https://publicdomainvectors.org> y <https://commons.wikimedia.org>.

Ciudad de México

Primera versión, Diciembre 2022

Hacia un método para la estimación automática de esfuerzo en el desarrollo de software a partir de historias de usuario

Resumen

La estimación de esfuerzo es importante para planificar eficientemente el uso de recursos en un proyecto de TI. En las fases de planeación de un proyecto, durante la elaboración de los artefactos y la pila de producto que define la funcionalidad total del proyecto, el equipo involucrado realiza una estimación inicial aproximada de tiempo y costo. Para mejorar la precisión de la estimación de esfuerzo en desarrollo de software existen varias técnicas de estimación: function points, object points, use case points, story points, etc. Los puntos de historia de usuario o story points, son la base de marcos de trabajo ágil que actualmente están tomando fuerza en el desarrollo de software.

Una de las principales lagunas de conocimiento se encuentra en la aplicación de aprendizaje automático en estimación de esfuerzo en desarrollo de software ágil. Este proyecto de investigación, contribuye a fortalecer el uso de modelos de estimación de esfuerzo con técnicas de aprendizaje automático como método de estimación de esfuerzo a nivel de proyecto en marcos de trabajo ágil que usan un enfoque de puntos de historia de usuario como Scrum.

Este trabajo presenta un estudio sobre el análisis de la precisión de la predicción del proceso de estimación ejecutado con diversas técnicas de aprendizaje automático supervisado como árboles de decisiones, redes neuronales, el algoritmo de los k vecinos más cercanos o KNN por sus siglas en inglés y un ensamble que combina los estimadores. Partiendo de un modelo de estimación de esfuerzo para el desarrollo de software ágil, se aplican las técnicas para estimar el tiempo de finalización y costo total de un proyecto tomando la pila de producto (esfuerzo), la velocidad del equipo de desarrollo y categorías de clasificación que agrupan los proyectos por tamaño de esfuerzo, tiempo y costo.

Los algoritmos de aprendizaje supervisado son evaluados a través del error cuadrático medio, error relativo medio, el coeficiente de determinación R^2 , la varianza y desviación estándar para determinar cuáles de las técnicas aplicadas ofrecen el mejor tiempo y el mejor costo. La capacidad de generalización de los algoritmos se evalúa aplicando validación cruzada de diez mangas. Con la aplicación de categorías de clasificación de tamaño de esfuerzo, tiempo y costo se prueba que es posible mejorar la precisión de las estimaciones al reducir los errores de evaluación. Los resultados se comparan con las propuestas establecidas en el estado del arte para determinar la fiabilidad de los modelos del presente proyecto.

Palabras Clave— Estimación, tiempo, costo, desarrollo ágil, aprendizaje automático.

Índice

1	Glosario	I
2	Introducción	1
2.1	Objetivos	2
2.1.1	Objetivo General	2
2.1.2	Objetivos particulares	2
2.2	Preguntas de investigación	3
2.3	Estructura del documento	3
3	Marco Teórico	5
3.1	Estimación y planificación	5
3.2	Modelos de estimación	6
3.3	Desarrollo de software ágil	7
3.4	Historias de usuario y puntos de historia	9
3.5	Aprendizaje automático y técnicas de aprendizaje supervisado	10
3.6	K vecinos más cercanos (KNN)	12
3.7	Árboles de Decisión	14
3.7.1	Bosque Aleatorio	16
3.7.2	AdaBoost	18
3.8	Redes Neuronales Artificiales	19
3.8.1	Función de activación	20
3.8.2	Neurona	20
3.8.3	Propagación hacia adelante (Forward Propagation)	22
3.8.4	Función de pérdida (loss)	22
3.8.5	El algoritmo de retropropagación	22
3.8.6	Tasa de aprendizaje	23
3.8.7	Red Neuronal Recurrente	23
3.8.8	Autoencoders	24
3.8.9	Diseño de un modelo de red neuronal	25
3.8.10	Regularización	25
3.9	Análisis de componentes principales	26
3.10	Discretización de variables continuas	27
3.11	Conjuntos de entrenamiento, validación y prueba	28
3.12	Validación Cruzada (k-Fold Cross Validation)	29
3.13	Otras técnicas de Inteligencia Artificial	30
4	Trabajo relacionado	32
4.1	Antecedentes	32
4.2	Desarrollo Tradicional	33
4.3	Desarrollo Ágil	36
5	Propuesta	40
5.1	Modelo de estimación	43
5.2	Clasificación de proyectos por tamaño	47
5.3	Modelo de aprendizaje automático	49
5.3.1	Estrategia de diseño de los experimentos	50
5.3.2	Integración de datos a los algoritmos	51
5.3.3	Resumen de técnicas aplicadas	52

6	Metodología	54
6.1	Recursos utilizados	55
6.2	Preprocesamiento y selección de características	56
6.2.1	Extracción de características	57
6.3	Diseño de algoritmos	57
6.3.1	Diseño empírico del algoritmo KNN	58
6.3.2	Diseño empírico de árboles de decisión	59
6.3.3	Diseño empírico de red neuronal MLP	60
6.3.4	Diseño empírico de Red Neuronal Recurrente	63
6.3.5	Diseño empírico de Autoencoder	66
6.4	Validación	67
7	Resultados	69
7.1	Regresión lineal simple y polinomial	69
7.2	Algoritmo KNN	70
7.3	Árboles de Decisión	73
7.3.1	Random Forest	76
7.3.2	AdaBoost	79
7.4	Redes Neuronales	82
7.4.1	Perceptrón Multicapa	82
7.4.2	Red Neuronal Recurrente	85
7.4.3	Autoencoder	89
7.5	Comparativa	94
7.6	Amenazas a la validez	97
8	Conclusiones	99
8.1	Trabajo a futuro	102
9	Apendice	103
9.1	Discretización de datos	103
9.2	Preprocesamiento de datos	103
9.3	Configuración de algoritmos	106
9.3.1	K-Nearest Neighbors	106
9.3.2	Árboles de decisión	106
9.3.3	Perceptrón multicapa	106
9.3.4	Red Neuronal Recurrente	107
9.3.5	Autoencoder	107
9.4	Entrenamiento de algoritmos	108
9.5	Evaluación de resultados	108
9.6	Ejecución de Red Neuronal Recurrente	109
10	Bibliografía	114

1. Glosario

ABE	Analogy Based Estimation
Adam	Adaptive Moment Estimation
ADD	Attribute-Driven Design
AE	Autoencoder
AID	Automatic Interaction Detector
ANN	Artificial Neural Network
API	Application Programming Interface
ASD	Agile Software Development
AWGN	Additive White Gaussian Noise
CBR	Case-Based Reasoning
CHAID	Chi-square Automatic Interaction Detector
CLS	Concept Learning Systems
COBRA	Cost estimation, Benchmarking, and Risk Assessment
COCOMO	Constructive Cost Model
DOI	Declaration of Interdependence
DSDM	Dynamic System Development Method
DT	Decision Tree
EWS	Early Warning Signs
FDD	Feature-Driven Development
FLANN	Functional Link Artificial Neural Networks
FP	Function Point
FPA	Function Point Analysis
GMDH	Group Method of Data Handling Polynomial Neural Network
GRNN	General Regression Neural Networks
IDEA	Interactive Data Exploration and Analysis
ISBSG	International Software Benchmarking Standards Group

KDSI	Kilo Delivered Source Instructions
KNN	K-Nearest Neighbor
LDA	Linear Discriminant Analysis
LSTM	Long Short-Term Memory
MAID	Multivariate AID
MLP	Multilayer Perceptron
MRE	Mean Relative Error
MSE	Mean Squared Error
OLS	Ordinary Least Squares
PCyTI	Posgrado en Ciencias y Tecnologías de la Información
PNN	Probabilistic Neural Network
RBNN	Radial Basis Neural Network
RF	Random Forest
RFECV	Recursive Feature Elimination with Cross-Validation
RMSProp	Root Mean Square Propagation
RNN	Recurrent Neural Network
SEE	Software Effort Estimation
SGB	Stochastic Gradient Boosting
SGD	Stochastic Gradient Descent
SLIM	Software Lifecycle Management
SLOC	Source Lines of Code
SNN	Spiking Neural Network
SVM	Support Vector Machine
SVR	Support Vector Regressor
THAID	THeta Automatic Interaction Detection
WBS	Work Breakdown Structure
WNN	Wavelet Neural Network
XP	Extreme Programming

2. Introducción

El presente documento es el resultado del proyecto de investigación llevado a cabo en el Posgrado en Ciencias y Tecnologías de la Información de la Universidad Autónoma Metropolitana Unidad Iztapalapa. El trabajo de investigación es uno de los veintiséis proyectos publicados en 2021 para los alumnos de la Maestría en Ciencias (CyTI). Hacia un método para la estimación automática de esfuerzo en el desarrollo de software a partir de historias de usuario, es una propuesta cuyo interés se encuentra en estudiar estrategias basadas en aprendizaje automático para proponer estimaciones de tiempo y costo usando la información de las historias de usuario.

La propuesta nace como una serie de experimentos estructurados basados en inteligencia artificial, en particular en el campo del aprendizaje automático que se enfoca en el uso de datos y algoritmos para imitar el proceso de aprendizaje humano. Los experimentos diseñan y prueban modelos de aprendizaje automático supervisado que generan estimaciones de tiempo de finalización y costo total de un proyecto de software a partir del total de puntos de historia de usuario de la pila de producto y la velocidad del equipo, es decir, las estimaciones se calculan considerando el total de la cantidad de trabajo a implementar y el número de elementos que un equipo de desarrollo completa durante un ciclo de trabajo (esfuerzo y velocidad respectivamente).

El objetivo de los experimentos consiste en la sintonización de sistemas que implica la selección de los hiperparámetros entrenables que permiten a los algoritmos emitir predicciones con alto porcentaje de precisión, coeficiente de determinación, y bajos valores de error cuadrático medio y error relativo. El resultado de la propuesta genera un modelo híbrido que combina un modelo de estimación basado en regresión lineal, técnicas de aprendizaje automático y categorías de clasificación, lo que permite obtener estimaciones de tiempo y costo de proyectos desarrollados con un enfoque de puntos de historia.

En la administración de proyectos de IT los riesgos siempre están presentes, la tasa de cancelación de proyectos antes de su finalización llega a ser de hasta el 20 %, y menos de un tercio de los proyectos son completados a tiempo y con el presupuesto esperado [111]. Existen señales de alerta temprana (EWS) que indican los posibles riesgos y dificultades que pueden surgir como la falta de entregables documentados, ausencia de fechas de vencimiento, una mala conciliación entre el cronograma del proyecto y la fecha límite, poco compromiso del equipo con el alcance y el cronograma del desarrollo, entre otras alertas [111].

Uno de los fundamentos de la ingeniería de software consiste en la gestión de proyectos. Toda organización que desarrolle sistemas de software debe entregar los proyectos en tiempo y forma, manteniendo los costos generales dentro del presupuesto y cumpliendo con las expectativas del cliente. Si se cumplen estos objetivos, la calidad está asegurada [55] y por lo tanto la satisfacción del cliente. Cada proyecto tiene características únicas, pero todos comparten factores comunes que afectan el desarrollo, incluido el tamaño de la empresa, los clientes y partes interesadas (*stakeholders*), el tipo y tamaño del software desarrollado, la cultura organizacional y el paradigma de desarrollo de software que se aplique [56].

Además, es importante considerar el hecho de que no se pueden pasar por alto algunas actividades como la planificación de proyectos, la gestión de riesgos, la gestión de personas, la presentación de informes y la redacción de propuestas. El propósito de la planificación de proyectos establece y mantiene planes que definen las actividades del proyecto. El plan del proyecto proporciona la base para realizar y controlar las actividades del proyecto que abordan

los compromisos con el cliente [60].

Podemos definir tres tipos de procesos [63]:

1. Estimación: el acto de juzgar y aproximar el valor o significado de algo.
2. Predicción: el acto de declarar o indicar con anticipación; predecir sobre la base de la observación, la experiencia o la razón científica.
3. Planificación: el acto de llevar a cabo planes, establecer metas, políticas y procedimientos.

En nuestra investigación nos estamos enfocando en la programación de fechas de finalización y estimación de costo total de un proyecto de software. La planificación de proyectos es fundamental durante la fase de inicio porque el equipo debe atender las necesidades y los requisitos relacionados con los procesos y actividades que satisfacen los objetivos comerciales [56]. Aquí es donde el equipo debe tener una propuesta clara, una estimación de los recursos para completar el trabajo y el tiempo necesario para entregar el producto terminado.

Bajo este contexto, la planificación de un proyecto contempla tres elementos básicos: recursos, alcance y fecha de entrega. La estimación requiere de una aproximación a los costos (cantidad de dinero gastado en una actividad o recurso específico), tiempos (horas necesarias para completar el sistema), alcance (compromiso de entregar cierta funcionalidad del sistema), y otros factores que pueden afectar el desarrollo del proyecto [7], también conocidos como riesgos. Si no se tienen claras las fechas de finalización de las actividades y posibles contratiempos, el líder puede tomar malas decisiones que ponen en riesgo al proyecto, generan entregas tardías, deuda técnica, aumento de costos, y a largo plazo posible pérdida de clientes. Por ello, es indispensable contar con mecanismos más confiables que apoyen el proceso de estimación en el desarrollo de proyectos de software.

2.1. Objetivos

Una vez ubicado el contexto de investigación se plantean los objetivos que busca el proyecto con la finalidad de ofrecer una propuesta que mejor se adapte a un enfoque de desarrollo de software ágil, generando un modelo híbrido que combina un modelo de estimación basado en regresión lineal, técnicas de aprendizaje automático y categorías de clasificación para emitir estimaciones de tiempo de finalización y costo total de un proyecto de software.

2.1.1. Objetivo General

Estudiar diversas técnicas de aprendizaje automático para construir un modelo capaz de realizar estimaciones de tiempo de finalización y costo total de un proyecto utilizando un enfoque de puntos de historia de usuario.

2.1.2. Objetivos particulares

1. Diseñar e implementar estrategias basadas en aprendizaje automático para generar estimaciones de tiempo y costo

2. Evaluar y comunicar los resultados.

2.2. Preguntas de investigación

Las etapas del proceso de investigación se muestran en la Figura 1. La primera etapa consiste en definir las preguntas de investigación adecuadas para delimitar el alcance del proyecto, para este estudio se definieron las siguientes preguntas:

- **RQ-1:** ¿Cuáles son los métodos de aprendizaje automático que se han utilizado en desarrollo de software ágil para estimar esfuerzo y costo de un proyecto?
- **RQ-2:** ¿Cómo se puede mejorar la precisión de las estimaciones de tiempo de finalización y costo total de un proyecto de software?

RQ-1 tiene el propósito de identificar los distintos métodos aplicados a estimación de esfuerzo en desarrollo ágil y ubicar aquellas técnicas que aún no han sido probadas, mientras que **RQ-2** identifica los elementos de los algoritmos ya existentes para realizar ajustes o mejoras e incrementar la precisión de las estimaciones.



Figura 1: Etapas que componen al proyecto de investigación.

2.3. Estructura del documento

La estructura del documento continúa con el capítulo 3, donde se presenta una introducción a la estimación de esfuerzo, la planificación, el desarrollo ágil, el enfoque de puntos de historia de usuario, la teoría relacionada a los algoritmos de aprendizaje automático que forman parte del marco técnico de este proyecto de investigación; incluyendo las particularidades

de cada algoritmo y sus hiperparámetros configurables que permiten su implementación, y también se incluyen secciones que abordan la validación aplicada para evaluar la capacidad de generalización de los algoritmos.

En el capítulo 4 se presenta el trabajo relacionado a la estimación de esfuerzo y a la estimación utilizando aprendizaje automático. El estado del arte representa una síntesis de los inicios de la estimación de esfuerzo mencionando las principales investigaciones que evolucionaron con el tiempo hasta llegar a la estimación con métodos ágiles y aprendizaje automático. De un pequeño conjunto de estudios, se seleccionan aquellas investigaciones que comparten el enfoque de puntos de historia de usuario aplicando un mismo modelo de estimación y conjunto de datos para realizar las estimaciones.

La propuesta que refiere este documento se desarrolla en el capítulo 5 con la intención de contribuir a fortalecer los modelos existentes de aprendizaje automático aplicados a métodos ágiles para realizar estimaciones de tiempo y costo de un proyecto de software.

La estrategia de los experimentos se desglosa y explica en el capítulo 6 presentando el proceso de entrenamiento de los algoritmos aplicados, los requisitos técnicos, así como el preprocesamiento del conjunto de muestras y diseño de cada algoritmo.

Los resultados de la evaluación empírica se muestran en el capítulo 7 mediante tablas que agrupan las métricas de evaluación, gráficas comparativas de las estimaciones hechas en el estado del arte contra la propuesta actual. También se hace evidente la diferencia que hay entre los valores reales y las estimaciones hechas por los algoritmos aplicados. Por último se presentan las amenazas a la validez, conclusiones y trabajo a futuro.

3. Marco Teórico

En esta capítulo se presentan aquellos aspectos fundamentales que forman parte del proceso de estimación del esfuerzo aplicado a la planificación y seguimiento de los recursos de un proyecto de software, ya que las fechas estimadas y el cálculo del presupuesto se ven afectados por la predicción del esfuerzo y el cronograma del desarrollo de software. Por eso resulta indispensable que toda organización sea capaz de identificar las herramientas, modelos y técnicas necesarias que les permita construir una base solida de conjuntos de datos de proyectos de software, con los cuales se pueda predecir con precisión el tiempo de finalización y costo total de un nuevo desarrollo que comparta características con los proyectos anteriores.

Las herramientas de estimación y predicción como los modelos de regresión o las técnicas de aprendizaje automático son importantes porque la evolución de los sistemas de software es cada vez más acelerada, las tecnologías de desarrollo y los paradigmas cambian rápidamente, y las organizaciones deben responder a estos cambios, pues estos sistemas son indispensables para el funcionamiento de la industria y la sociedad [59], [63].

La gestión eficaz de los activos de una organización es fundamental para el éxito empresarial. Estas organizaciones son desarrolladores de productos y servicios que necesitan una forma de gestionar sus actividades de desarrollo para poder lograr la consecución de sus objetivos de negocio [60]. En el caso de las organizaciones de tecnologías de la información (TI), el desarrollo de sistemas forma parte de los servicios ofertados y es la manera en que crean valor, por eso es esencial que en el desarrollo de software se gestionen de manera adecuada cada una de las actividades relacionadas a un proyecto.

3.1. Estimación y planificación

Al ofrecer un servicio como el desarrollo de software, el término *estimación de esfuerzo* se usa con frecuencia sin aclarar su significado, y la precisión de la estimación a menudo se evalúa sin garantizar que el esfuerzo estimado y el real sean comparables. Además, la estimación confiable se encuentra entre los diez factores de éxito más importantes en los proyectos de software [61]. Por lo tanto, es importante no utilizar indiscriminadamente el término estimación de esfuerzo para referirse a la planificación o la fijación de presupuesto, siempre es recomendable referirlos de manera independiente [63].

El proceso de estimación de las actividades relacionadas con ingeniería software consiste en calcular el número de horas/persona o días de trabajo necesarios para completar las actividades y tareas con los recursos estimados, es decir, se da una aproximación a la cantidad de tiempo que llevará completar cada actividad que forma parte de las características del sistema. Estimar los recursos de una actividad contempla los suministros, equipos, tecnología y conocimientos técnicos necesarios para realizar el trabajo del proyecto. Al desarrollar un calendario se analiza la secuencia de actividades, su prioridad, duración y complejidad técnica. Los equipos de trabajo pueden utilizar distintos métodos para realizar dichas aproximaciones, uno de los más conocidos son las técnicas de juicio de expertos.

Planificar la gestión de costos es un proceso en donde se define la manera en que se presupuestarán, gestionarán, supervisarán y controlarán los costos del proyecto [52]. La planificación está orientada a una correcta administración de los costos del proyecto a lo largo del mismo. La estimación de costos aproxima los recursos monetarios necesarios para completar el trabajo

relacionado con el proyecto, y mantener el presupuesto dentro del rango acordado. Una buena planificación refleja un plan de administración de costos que determina un presupuesto con sus correspondientes estimaciones y fija un control que monitorea el avance del proyecto y ajusta el presupuesto basado en las actualizaciones. Como lo define Steve McConnell [63]: *"Una buena estimación es una estimación que proporciona una visión lo suficientemente clara de la realidad del proyecto para permitir que el liderazgo del proyecto tome buenas decisiones sobre cómo controlar el proyecto para alcanzar sus objetivos."*

3.2. Modelos de estimación

En la literatura existen varias propuestas de modelos de estimación de esfuerzo utilizadas por los investigadores de acuerdo a los diferentes modelos de procesos de software [67], [68]. Estimación de esfuerzo de software o SEE por sus siglas en inglés apareció por primera vez en la década de 1950 y desde entonces ha atraído la atención de los especialistas con el objetivo de desarrollar modelos útiles que expliquen el ciclo de vida y la predicción de costos de un proyecto de software [58], [66].

En los años 70's, diversos esquemas y modelos se plantearon para estimar el tiempo y costo de un proyecto de software de acuerdo a su complejidad como son los modelos de estimación tradicionales SLIM y COCOMO. El campo de investigación ha resaltado de manera general que las técnicas de estimación se pueden clasificar en no algorítmicas y algorítmicas [15], [64]. Otras clasificaciones más detalladas se dividen en técnicas empíricas basadas en la práctica y experiencia previa (basado en analogía), técnicas basadas en aprendizaje automático, técnicas algorítmicas basadas en modelos, técnicas de *expertise* basadas en razonamiento no explícito (juicio de experto), técnicas basadas en modelos de regresión y técnicas compuestas [58].

La estimación de juicio de experto se basa en las experiencias acumuladas de equipos de individuos con una larga trayectoria. Los resultados han demostrado que esta es la estrategia dominante en la estimación del desarrollo de software. La técnica *Delphi* y la estructura de descomposición del trabajo (WBS), la estimación *top-down* y *bottom-up*, el razonamiento por analogía, el razonamiento formal e informal por analogía, y las reglas empíricas corresponden a técnicas no algorítmicas de juicio de experto [43], [58]. El método de punto de control es un ejemplo de un enfoque basado en analogía para la estimación de software. El modelo basado en la estimación de costos, la evaluación comparativa y el análisis de riesgos (COBRA) es un ejemplo de un modelo de estimación compuesto.

Existen otros modelos denominados modelos orientados al aprendizaje [69], estos modelos intentan automatizar el proceso de estimación entrenándose a sí mismos a partir de la experiencia previa para construir modelos computarizados. Los modelos orientados al aprendizaje cubren un área amplia e incluyen técnicas de inteligencia artificial, redes neuronales artificiales, razonamiento basado en casos, modelos de aprendizaje automático, modelos de lógica difusa, adquisición de conocimiento e inducción de reglas.

El aprendizaje automático es uno de los enfoques basados en analogía que se distingue por su bajo costo, simplicidad y precisión relativa cuando existe una base de datos confiable [65]. Esta estrategia permite aplicar distintos tipos de métricas y no dependen de la subjetividad con la que el equipo pueda estimar el trabajo. Al ser una técnica intensiva en datos tiene la desventaja de no poder manejar datos faltantes o no cuantitativos y su precisión depende de la disponibilidad de proyectos históricos [15].

El proceso de planificación basado en datos históricos es fundamental al aplicar técnicas basadas en aprendizaje automático, pues facilita la planeación y gestión de las actividades (esfuerzo) y los recursos relacionados. Apoyados por un conjunto de datos históricos de proyectos finalizados, los modelos son capaces de identificar patrones y realizar aproximaciones de tiempo y costo tomando como datos de entrada características esenciales del proyecto. Con esto es posible obtener predicciones que sirven como un punto de apoyo durante la elaboración del plan de administración del proyecto.

Una buena parte de los modelos de estimación están basados en técnicas de regresión [62]. Con los métodos algorítmicos, además de estimaciones basadas en la fórmula original, se han estudiado modelos de regresión para optimizar la precisión de la estimación del tamaño o la estimación del esfuerzo en función de los datos históricos [43]. Los modelos de regresión estadística estiman el esfuerzo de desarrollo de software como la variable dependiente. El tamaño del software se utiliza como variable independiente que según el caso pueden ser líneas de código, puntos de función, puntos de casos de uso o puntos de historia de usuario. Los modelos de regresión tienen la ventaja de una base matemática sólida, así como medidas que permiten conocer qué tan bien coincide la curva con el conjunto de datos dado utilizando, por ejemplo, el coeficiente de determinación R^2 [62].

Así, la estimación de esfuerzo en ingeniería de software estará definida por el modelo que un equipo decida aplicar en un contexto específico, el paradigma con el que se construya el sistema y las herramientas y metodologías definidas que permitan visualizar y controlar el desarrollo del proyecto. De manera similar, en el caso de este proyecto de investigación, es indispensable definir el modelo de estimación aplicado, las técnicas de aprendizaje automático utilizadas y la manera en que el modelo y las técnicas se combinan para realizar las estimaciones de tiempo de finalización y costo total de un proyecto desarrollado con metodologías ágiles, aplicando puntos de historia de usuario.

3.3. Desarrollo de software ágil

La administración de proyectos ágiles en desarrollo de software surge con la consolidación del manifiesto ágil y la declaración de interdependencia (DOI). En febrero de 2001, en una reunión en Utah, diecisiete expertos representantes de las principales metodologías ágiles como XP, *Scrum*, DSDM, *Crystal* y FDD, coincidían con la necesidad de una alternativa a los procesos de desarrollo de software de gran peso impulsados por la documentación. Dicho encuentro generó un texto breve y conciso llamado el Manifiesto de desarrollo de software ágil¹:

Descubrimos mejores formas de desarrollar software haciéndolo y ayudando a otros a hacerlo. A través de este trabajo hemos llegado a valorar:

- *Individuos e interacciones sobre los procesos y herramientas*
- *Software funcional por encima de la documentación exhaustiva*
- *Colaboración con el cliente sobre la negociación del contrato*
- *Responder al cambio por encima de seguir un plan.*

¹<https://agilemanifesto.org>

Esto es, mientras que hay valor en los elementos de la derecha, nosotros valoramos más los elementos de la izquierda.

Es decir, un enfoque ágil da prioridad a los individuos e interacciones, al software funcional, a la colaboración con el cliente, y a responder y adaptarse al cambio.

Doce son los principios del manifiesto que se enfocan en la satisfacción del cliente a través de entregas tempranas y continuas de software funcional, una adaptabilidad al cambio en requerimientos, relaciones entre las personas interesadas y el equipo de desarrollo, arquitecturas simples de equipos auto-organizados y procesos ágiles que promueven el desarrollo sostenible.

Mientras que el manifiesto ágil se enfocó en el desarrollo de software, DOI se centró más en la administración de proyectos. En 2005 los expertos aseguraron que para entregar resultados satisfactorios hay que:

- Aumentar el retorno de la inversión haciendo que el flujo continuo de valor sea el principal enfoque
- Entregar resultados confiables
- Administrar la incertidumbre a través de la anticipación y adaptabilidad
- Dar rienda suelta a la creatividad e innovación
- Impulsar el desempeño a través de la responsabilidad grupal
- Mejorar la eficacia y la fiabilidad a través de estrategias, procesos y prácticas.

Dos de los autores que participaron en la creación del manifiesto ágil son Jeff Sutherland y Ken Schwaber. Ambos autores colaboraron en 1995 recopilando la experiencia de años de trabajo, y creando una nueva metodología de desarrollo de software que describieron en el artículo de Schwaber llamado "*Scrum and the Perfect Storm*". El término *Scrum* proviene de un artículo publicado por Hirotaka Takeuchi e Ikujiro Nonaka en *Harvard Business Review* en 1986 [49]. Los autores describieron un enfoque holístico en donde los proyectos se componen de equipos pequeños multifuncionales (*cross-functional*) que trabajan hacia un objetivo común, que los autores compararon con la formación *Scrum* en el *rugby*.

Scrum es uno de los marcos de trabajo más utilizados, anunciado en 1996 en la conferencia *Object Oriented Programming, Systems, Languages, and Applications* (OOPSLA) [49], la metodología asume que el proceso de desarrollo de sistemas es un proceso complicado e impredecible. *Scrum* define el proceso de desarrollo de sistemas como un conjunto flexible de actividades que combina técnicas y herramientas conocidas, con lo mejor que un equipo de desarrollo puede diseñar para construir sistemas [50]. Como lo describen sus autores, *Scrum* es una mejora del ciclo de desarrollo iterativo/incremental orientado a objetos inicialmente propuesto por Matthew Pittman y mejorado por Grady Booch.

Como lo indica Schwaber, los lanzamientos de productos de software se planifican en función de los requerimientos del cliente, el tiempo requerido, la competencia, la calidad, la visión y los recursos. La estimación de proyectos en un inicio aplicaba *Function Points* que reflejan la clasificación y peso de los componentes funcionales del sistema, y se recomendaba estimar la productividad en aproximadamente el doble de la métrica actual.

3.4. Historias de usuario y puntos de historia

Los creadores de la metodología XP Ron Jeffries, Kent Beck y Ward Cunningham introdujeron una serie de conceptos que posteriormente se retomarían en *Scrum*. El término historia de usuario surge en los 90's con Kent Beck². Las historias de usuario surgen como una forma de expresión alternativa a los casos de uso, y es Mike Cohn quien propone la estructura básica de la historia de usuario como hoy la conocemos.

Una historia de usuario es un enunciado que describe cierta funcionalidad del sistema que permite al usuario alcanzar un objetivo que le aporta valor a la organización. Toda historia de usuario tiene un identificador, un nombre, una descripción, estimación inicial, criterios de aceptación y una posible lista de tareas o actividades a realizar para completar dicha historia.

A Ron Jeffries³ se le atribuye el concepto de punto de historia. Un *story point* o punto de historia es una unidad de medida que utiliza una escala numérica para proporcionar estimaciones acerca de la dificultad (entiéndase como esfuerzo necesario) para implementar funcionalidad a un sistema. La complejidad de una historia de usuario está relacionada con los requerimientos funcionales asociados, atributos de calidad y requerimientos técnicos.

El conjunto de historias de usuario que componen un proyecto se conoce como pila de producto o *product backlog*. La pila de producto es la suma de puntos de todas las historias de usuario, y la velocidad del proyecto refleja la cantidad de esfuerzo que un equipo realiza en una iteración o *sprint*.

La velocidad representa la cantidad de trabajo completado en una iteración y permite al equipo planificar su capacidad para el siguiente *sprint* de manera más precisa al analizar su rendimiento histórico [53]. La velocidad se puede estimar con técnicas que se basan en datos históricos, ejecutando iteraciones y mediante pronósticos [54].

El consumo de tiempo refiere la cantidad de horas o días necesarios para implementar una historia de usuario, no especifica la cantidad de trabajo, sino el tiempo necesario para completar las tareas de la historia. El esfuerzo implica el volumen de trabajo requerido para ejecutar una historia de usuario [71], y se expresa en varias unidades pero una de las más conocidas son los puntos de historia. De esta manera, la velocidad es igual al volumen de trabajo completado en una cantidad específica de tiempo (puntos de historia / duración del *sprint*).

En este proyecto de investigación se realizan estimaciones con base en información relacionada con un soporte estadístico de otros proyectos de desarrollo de software ya realizados con anterioridad. Con este enfoque, los modelos de aprendizaje automático solo se pueden usar precisamente en las reuniones iniciales a punto de comenzar el *sprint*, es decir, cuando el proyecto se está ejecutando y ya conocemos la velocidad dada la información del proyecto disponible en esta etapa de desarrollo, o cuando se conoce la velocidad de un equipo que ha desarrollado proyectos similares, es decir conocemos la velocidad ejecutando iteraciones o mediante datos históricos.

Las técnicas de aprendizaje automático toman características de diversos proyectos como puede ser el esfuerzo total, la velocidad del equipo, el tamaño del *sprint*, el valor de la desaceleración que refleja las fuerzas y los factores que ralentizan el proyecto, el salario del equipo o

²<https://www.youtube.com/watch?v=X7e4qgjZzbE>

³<https://ronjeffries.com/articles/019-01ff/story-points/Index.html>

los días laborales, y con estas características realizan estimaciones de tiempo de finalización y costo total. Con ayuda de distintas técnicas de aprendizaje supervisado es posible crear un modelo que mejore la precisión de las estimaciones y reduzca el error que hay entre los valores predichos y los valores reales. En la siguiente sección se define de manera general a un algoritmo de aprendizaje supervisado, también conocido como función de aprendizaje de entrada-salida que en este trabajo toma las características de los proyectos de software ágiles para realizar las estimaciones de los proyectos.

3.5. Aprendizaje automático y técnicas de aprendizaje supervisado

Para que un sistema informático o una computadora resuelva un problema se requiere de un algoritmo, es decir, necesitamos una serie de pasos o instrucciones que se deben ejecutar para transformar un conjunto de entradas en salidas. En ocasiones, para algunas tareas no se cuenta con un algoritmo, se sabe cuáles son las entradas y cuáles deberían ser las salidas pero no se conoce la forma en que dichas entradas se procesan para producir un resultado. Al no contar con un algoritmo específico, necesitamos de un procedimiento útil que nos permita hacer una aproximación, esta aproximación tal vez no explica todo el contenido, pero puede ser capaz de procesar y coincidir algunos de los resultados esperados e identificar patrones en los datos de entrada.

Esta labor forma parte del aprendizaje automático que comenzó en la década de los años 40's con trabajos como el primer modelo matemático de red neuronal de Walter Pitts y Warren McCulloch [72], el libro *The Organization of Behavior* de Donald O. Hebb [75] cuyo concepto se centra en la organización neuronal de la conducta, el juego de damas desarrollado a nivel competitivo por Arthur Samuel [73], [74], empleado de la compañía IBM y un pionero en el campo de los juegos informáticos y la inteligencia artificial quien popularizó el término de *machine learning*.

El campo de aprendizaje automático tiene distintas aplicaciones como el aprendizaje de asociaciones (encontrar reglas de asociación), problemas de clasificación y regresión, reconocimiento de patrones (reconocimiento de rostros, caracteres, etc), diagnóstico médico, reconocimiento de voz, procesamiento de lenguaje natural, autenticación biométrica, extracción de conocimiento, etc [17]. El aprendizaje automático normalmente se refiere a cambios en los sistemas que realizan tareas asociadas con la inteligencia artificial [70]. El aprendizaje automático como rama de la inteligencia artificial provee sistemas con la habilidad de aprender de la experiencia sin ser programados explícitamente [20]. El objetivo principal consiste en permitir a las computadoras aprender sin la intervención de humanos y ajustarse en consecuencia.

Este enfoque de aprendizaje consiste en extraer conocimiento de datos a través de una función de aprendizaje, es un campo de investigación en la intersección de la estadística, la inteligencia artificial y la computación, también se lo conoce como análisis predictivo o aprendizaje estadístico [19]. En el aprendizaje automático existen varias categorías para clasificar las distintas técnicas que lo componen [21], las áreas más conocidas son:

- **Aprendizaje supervisado:** requiere de un entrenador que proporciona las instancias de entrenamiento de entrada-salida. El sistema de aprendizaje adapta sus parámetros mediante algunos algoritmos para generar los patrones de salida deseados a partir de un patrón de entrada dado.

- **Aprendizaje no supervisado:** en este caso no se conoce el resultado deseado para una instancia de entrada determinada, por lo tanto el agente debe adaptar sus parámetros de forma autónoma. El agente tiene que construir conceptos experimentando con el entorno. Este tipo de aprendizaje utiliza algoritmos para analizar y agrupar conjuntos de datos sin etiquetar.
- **Aprendizaje por refuerzo:** al no conocer explícitamente las instancias de entrada-salida, el agente recopila información a través de un curso de acciones al interactuar con el entorno. En respuesta a una acción, el agente recibe dos tipos de información: su estado actual en el entorno y una recompensa de valor real que es específica para una tarea. El objetivo es maximizar la recompensa y determinar el mejor curso de la acción [112].

La naturaleza del problema permite aplicar técnicas de aprendizaje supervisado; conocemos gracias al conjunto de datos históricos de proyectos concluidos, cuáles son las características que nos permiten realizar las estimaciones de tiempo y costo total. En el aprendizaje supervisado existen funciones de aprendizaje de entrada-salida, las cuales describen el problema de una función de aprendizaje. N. J. Nilsson [70] explica de manera sencilla la terminología de estas funciones que se retoma a continuación:

Existe una función f , un agente (*learner*) tiene la tarea de aproximar cuál es esa función. La hipótesis acerca de la función a aprender se denota por la hipótesis h . Tanto f como h son funciones de una entrada con valor vectorial $\mathbf{X} = (x_1, x_2, \dots, x_n)$ que tiene n componentes, cada componente puede ser un valor real, discreto o categórico. Pensamos en la función h como una función que es implementada por un dispositivo que tiene $\mathbf{X}$ como entrada y $h(\mathbf{X})$ como salida, la función h es conocida como estimador de funciones si produce valores numéricos, y clasificador si la salida produce valores categóricos. Ambas funciones f y h pueden ser vectoriales. Asumimos *a priori* que la función hipotética h se selecciona de una clase de funciones $\mathcal{H}$, algunas veces se sabe que la función f también pertenece a esta clase o a un subconjunto de la misma clase. La función h se selecciona basada en un conjunto de entrenamiento Ξ de m muestras vectoriales de entrada.

Cuando deseamos aprender una función con el uso de aprendizaje supervisado, se aproximan los valores de f para las m muestras del conjunto de entrenamiento Ξ . Suponemos que si podemos encontrar una función h , que concuerde estrechamente con la función f para los miembros del conjunto Ξ , entonces esta función de hipótesis será una buena suposición de f , especialmente si Ξ es grande.

Al proceso de encontrar una función h basado en un conjunto Ξ se conoce como régimen de entrenamiento [70], y puede ser de tres tipos, el método por lotes donde todo el conjunto de entrenamiento está disponible y se usa todo a la vez para calcular la función h , el método incremental donde se selecciona cada vez un elemento del conjunto Ξ para obtener h , y el método en línea donde se usan los miembros del conjunto de entrenamiento a medida que estén disponibles, por ejemplo, cuando la siguiente instancia de entrenamiento es alguna función de la hipótesis actual y la instancia anterior.

Los algoritmos clásicos que forman parte del aprendizaje supervisado incluyen los modelos de regresión lineal, modelos basados en árboles de decisiones, redes neuronales artificiales, redes de aprendizaje profundo, máquinas de soporte vectorial, el algoritmo de los k vecinos más cercanos, la estimación bayesiana y Bayes ingenuo (*Naïve Bayes*) [76], [77], [17], [20], [21].

Una regla básica de todo algoritmo de aprendizaje supervisado es identificar cuáles son las entradas y salidas, además de conocer la terminología básica de los algoritmos a aplicar, pues de

ellos depende una correcta configuración y entrenamiento que permita realizar aproximaciones de una manera precisa y evitar que la información sea sobreajustada y los modelos no sean capaces de encontrar la función hipotética que produzca los resultados esperados. En la siguiente sección se detalla cada una de las técnicas aplicadas en este proyecto de investigación, presentando los conceptos básicos y resaltando los hiperparámetros configurables que llevan a su implementación.

Los distintos modelos aplicados en este proyecto de investigación son diversas variantes de redes neuronales dado que son consideradas aproximadores universales de funciones diferenciables [36] y tienen la capacidad de adaptarse a problemas no lineales, el algoritmo KNN por su simplicidad, los árboles de decisión por su facilidad de interpretación, y diversas técnicas de ensamble que permiten robustecer el modelo de estimación como *Random Forest*, *Bagging* y *AdaBoost*.

3.6. K vecinos más cercanos (KNN)

El primer método de solución que se ha estudiado en este proyecto es uno de los más simples y más utilizados en el aprendizaje supervisado [14], el algoritmo KNN [17]. Bajo este contexto el algoritmo logra emular el proceso de estimación por analogía, es decir, un proyecto nuevo puede ser clasificado en tamaño de esfuerzo, tiempo y costo comparándolo con proyectos de esfuerzo y/o velocidad similar. El algoritmo de los k vecinos cercanos calcula las distancias entre observaciones, a menudo utilizando la distancia euclidiana [18]. Técnicamente, KNN no entrena un modelo para hacer predicciones [34]. En cambio, utiliza las observaciones de los puntos más cercanos para predecir que la clase de un punto será la clase de la mayor proporción de las k observaciones más cercanas.

A Evelyn Fix y Joseph Hodges se les atribuyen las ideas iniciales del algoritmo KNN publicado en 1951 [24] y considerado como un método no paramétrico (no hace suposiciones sobre la forma paramétrica de la distribución) [26]. Posteriormente en 1967 [25], se detalla el método presentando el problema de clasificación, la regla del vecino más cercano, la versión más simple del procedimiento de decisión de Bayes para minimizar la probabilidad de error al clasificar una observación, la convergencia de los vecinos más cercanos y su probabilidad de error.

El algoritmo KNN se ha desarrollado al punto en que existe una justificación matemática para la distribución uniforme de elementos que forman un espacio vectorial $\mathbb{R}^d$ de todas las d -tuplas $\mathbf{x} = x_1, \dots, x_d$ con norma euclidiana $(\|\mathbf{x}\| = x_1^2 + \dots + x_d^2)^{\frac{1}{2}}$ [78], también hay desarrollo teórico para una consistencia en la distancia de los elementos x_i, x_j con $i \neq j$, una estimación de densidad de probabilidad no paramétrica, útil como una función que revela la concentración local de masa de probabilidad que puede usarse para visualizar distribuciones de variables aleatorias. También se ha escrito sobre las propiedades de la estimación de la densidad del vecino más cercano k , las diferentes formas de ponderar la estimación de la densidad del vecino más cercano k , entre otros conceptos [78]. La investigación se ha enfocado a diseñar algoritmos para la búsqueda del vecino más cercano que sean eficientes incluso cuando la cantidad de puntos o dimensionalidad de los datos crezcan, por lo que existen estructuras de datos populares y algoritmos desarrollados para diversos problemas ya sean de regresión o clasificación.

El problema de la búsqueda del vecino más cercano en un espacio euclidiano se puede definir como [79]:

Definición. Dado un conjunto P de puntos en un espacio d -dimensional $\mathbb{R}^d$, construya una estructura de datos que, dado cualquier punto de consulta X encuentre el punto en P con la menor distancia a X .

La solución requiere especificar el cálculo de la distancia entre dos puntos arbitrarios X, Y , utilizando la ecuación de distancia de minkowski [80]:

$$\mathcal{D}(X, Y) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}} \quad (1)$$

Donde $X = (x_1, x_2, \dots, x_n)$, $Y = (y_1, \dots, y_n) \in \mathbb{R}^d$, y el valor de p determina la distancia, cuando $p = 1$ se obtiene la distancia manhattan, $p = 2$ equivale a la distancia euclidiana. Si los puntos se encuentran en el plano, el problema del vecino más cercano se puede resolver con un tiempo $O(\log n)$ por consulta, utilizando el almacenamiento $O(n)$. Desafortunadamente, a medida que crece la dimensión, el algoritmo se vuelve menos eficiente, sus requerimientos de espacio o tiempo crecen exponencialmente, teniendo una solución con un tiempo de consulta $O(d^{O(1)} \log n)$, pero usando aproximadamente $n^{O(d)}$ de espacio [79].

Al no poder eliminar la dependencia exponencial de la dimensión, los investigadores llegaron a la conjetura de que no existen soluciones eficientes para este problema cuando la dimensión es lo suficientemente grande. Afortunadamente, en el contexto de este proyecto de investigación la dimensión de los datos de entrada no es lo suficientemente grande como para caer en este dilema, lo que permite aplicar este método de estimación. Al implementar el algoritmo KNN solo una serie de conceptos relevantes configuran al algoritmo y permiten realizar las estimaciones sobre un conjunto dado, estos conceptos se enlistan a continuación:

- **Número de vecinos cercanos:** cantidad de elementos a considerar para realizar las estimaciones
- **Método para ponderar las muestras:** ponderaciones uniformes o ponderación basada en la distancia que existe con los vecinos
- **Estructura de datos utilizadas** para mejorar el cálculo de las distancias: *k-d Tree*
- La **métrica de distancia** utilizada por los árboles al realizar los cálculos: distancia euclidiana, *manhattan* o la ecuación de *Minkowski* que es una generalización de la distancia.

K-d Tree es una estructura de tipo árbol creada por Jon Bentley en 1975 utilizada para realizar búsquedas en espacios multidimensionales (en la memoria principal) [81]. Esta estructura se construye de manera recursiva encontrando la mediana M de los valores de las i -ésimas coordenadas de los puntos, es decir, se calcula un valor de M de modo que al menos el 50 % de los puntos tengan su coordenada i -ésima mayor o igual a M y el otro 50 % tenga su coordenada i -ésima menor a M [79].

Por ejemplo, si ubicáramos un conjunto de puntos aleatorios en un plano coordenado como en la Figura 2, la construcción del árbol particiona los elementos tomando el punto medio de cada una de las coordenadas. En un espacio de n dimensiones, cada dimensión se particiona a la mitad, al construir el árbol, la primera dimensión corresponde al primer nivel del árbol, el segundo nivel corresponde a la segunda dimensión, continuando así hasta n . Al llegar al nivel n , se repiten las dimensiones, la primera, la segunda, y así sucesivamente.

En este caso los puntos medios se muestran en color negro. Iniciando con el eje de las abscisas,

A es el punto medio y divide al conjunto por medio de una línea punteada, generando a la izquierda el grupo de puntos con abscisa menor a A, y a la derecha el grupo de puntos con abscisa mayor a A. El siguiente paso es tomar la segunda dimensión (ordenadas) y dividir con el punto medio, las muestras a la izquierda de A se dividen por medio del punto B en el eje de las ordenadas, mientras que las muestras a la derecha del punto A se dividen en el eje de las ordenadas con el punto C. El siguiente nivel del árbol cambia de nuevo al eje de las abscisas y divide los nuevos subconjuntos mediante los puntos D, E, F y G.

El procedimiento continua hasta que ya no hay puntos que dividir, en negro se van marcando los nodos del árbol y siguiendo el orden se generaría una estructura parecida al árbol encerrado en el recuadro rojo. La búsqueda de vecino más cercano parte de la raíz del árbol, y en cualquier momento, el algoritmo mantiene la distancia $\mathcal{D}$ al punto más cercano a Y encontrado hasta el momento, actualizando su valor cuando encuentra un candidato con una distancia menor.

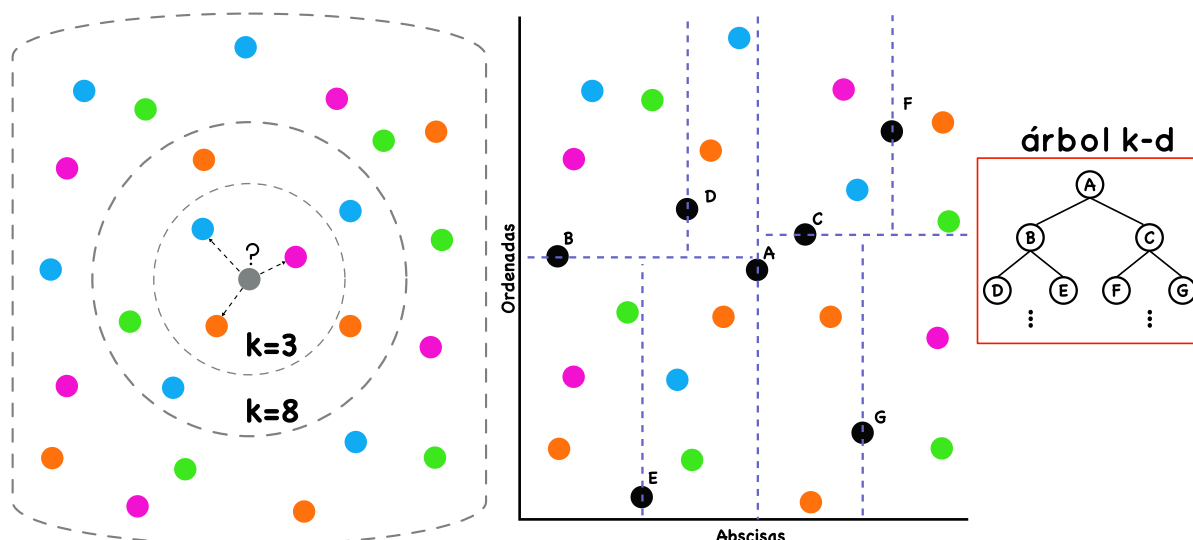


Figura 2: La búsqueda del vecino más cercano mejora utilizando una estructura de tipo k-d Tree.

3.7. Árboles de Decisión

Los árboles de decisión son un tipo de método estadístico que representa de manera gráfica el proceso de toma de decisiones bajo varias condiciones específicas [91]. La toma de decisiones requiere un análisis de la situación con la ayuda de los datos disponibles antes de dar un resultado final. Frank P. Ramsey [87] fue el primero en desarrollar una metodología de análisis de decisiones basada en la probabilidad y la utilidad en el año de 1931, y Quinlan el autor que introdujo la inducción de los árboles de decisión en 1986 [88]. Estos métodos evolucionarían en el dominio del análisis y fueron desarrollados principalmente por estadísticos [90, 82, 84, 85, 83, 89].

Las áreas de aplicación y los trabajos orientados a la construcción de árboles es amplia [92], estratégicamente el algoritmo aplicado en este proyecto de investigación es uno de los más simples en interpretación, los árboles de regresión. Al referirnos a los árboles de decisión o DT durante el documento, asumimos que estamos hablando de árboles de regresión.

De manera general definimos a un árbol de decisión [17] como un algoritmo capaz de mapear datos de entrada a un valor de salida. La estructura de un árbol de decisión se muestra

en la Figura 3 y la decisión se alcanza realizando alguna prueba comenzando en la parte superior o la raíz del árbol y luego siguiendo una rama hasta llegar a una hoja [23]. Cada rama del árbol proviene de un valor de prueba o de una condición, dependiendo de la respuesta, los datos evaluados pueden ir a una rama u otra.

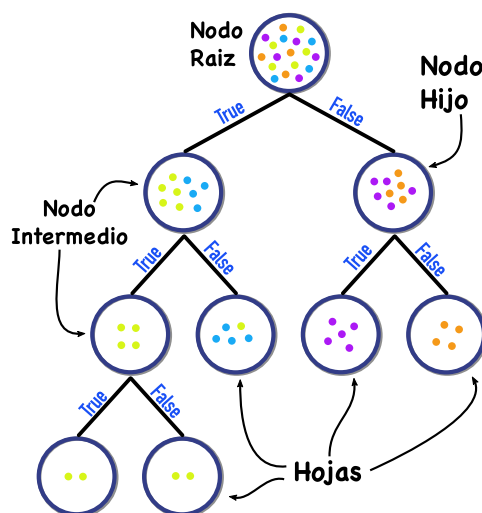


Figura 3: Estructura básica de un árbol de decisión, las condiciones generan bifurcaciones que trazan las ramas del mismo.

Un árbol de decisión se construye a partir de un conjunto de entrenamiento que consiste de objetos, y cada objeto se describe por un conjunto de atributos o tipo de clase. Los atributos pueden ser valores reales, valores booleanos o categorías. Un árbol puede contener cero o más nodos internos y uno o más nodos hoja [92]. Todos los nodos internos contienen divisiones, que prueban el valor de una expresión de los atributos y generan nuevas ramas. Así, un árbol de decisión es una estructura de datos jerárquica que implementa la estrategia divide y vencerás [17]. El proceso de estimación que comienza en la raíz se repite recursivamente hasta que se alcanza un nodo de hoja cuyo valor constituye la salida y es una medida apropiada para la regresión.

Construir un árbol y continuar hasta que todas las hojas sean puras conduce a modelos complejos que se ajustan mucho a los datos de entrenamiento. La presencia de hojas puras significa que un árbol es 100 % preciso en el conjunto de entrenamiento [19]. Hay dos estrategias comunes para evitar el sobreentrenamiento, es decir, para evitar la memorización de la información y que el algoritmo solo funcione con ese conjunto particular: detener la creación del árbol en cierto punto (*pre-prunning*), o construir el árbol y luego eliminar los nodos que contienen poca información (*prunning*).

El nodo raíz del árbol comprende todo el espacio predictor. Los nodos que no se dividen se denominan nodos terminales (hojas) y forman la partición final del espacio predictor. Cada nodo intermedio se divide en dos nodos descendientes según el valor de una de las variables predictoras. Para una variable predictora continua (de valores reales), una división está determinada por un valor c de división; los puntos para los que el predictor es más pequeño que el punto de división c van a la izquierda, y los puntos para los que el predictor es mayor al punto de división c van a la derecha.

Para dividir un árbol en sus dos descendientes se considera cada división posible en cada variable predictora y se elige la mejor de acuerdo con algún criterio, esto es, al tener distintas dimensiones de entrada se consideran las divisiones que puede generar cada una de ellas. En

el contexto de regresión, si los valores de respuesta en el nodo son $y_1, \dots, y_n$, el criterio de división típico es el residuo cuadrático medio en el nodo

$$Q = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2 \quad (2)$$

donde $\bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$, es el valor predicho en el nodo (el promedio de los valores de respuesta). El criterio de división da una medida de la buena calidad del ajuste del algoritmo (*fit*), y a mayor valor el ajuste será deficiente.

La regresión estructurada en árboles posee tres ventajas significativas sobre la regresión paramétrica y no paramétrica estándar como lo indica [93]:

1. Al permitir que la estructura de árbol maneje gran parte de la complejidad general del modelo, los modelos en cada partición se pueden mantener en un orden bajo y, por lo tanto, se pueden interpretar más fácilmente.
2. Las interacciones entre las covariables se transmiten directamente por la estructura del árbol de decisión. Como resultado, las interacciones pueden entenderse e interpretarse más fácilmente en términos cualitativos.
3. La forma simple de la función ajustada en cada nodo terminal permite estudiar analíticamente las propiedades estadísticas del método.

De manera similar al algoritmo KNN, al implementar un árbol de decisión solo es necesario detectar aquellos valores de configuración básica que permiten su programación mediante bibliotecas. Las características más importantes a considerar son:

- **Profundidad máxima**, al emplear la poda anticipada (*pre-prunning*) el algoritmo detendrá su ejecución una vez alcanzado un valor de profundidad fijo, es decir dado un valor m , el árbol detendrá su creación al tener un máximo de m niveles
- **Muestras mínimas para dividir**, es el número mínimo de muestras requeridas para dividir un nodo interno
- **Criterio de regresión**, es una función para medir la calidad de una división (*split*). El criterio utilizado es el error cuadrático medio, que es equivalente a la reducción de la varianza como criterio de selección de características.

El auge de los métodos de ensamble ha dificultado que los algoritmos compuestos de un solo árbol compitan en términos de precisión de predicción [84]. Otras propuestas que combinan más de un árbol han surgido para mejorar las predicciones hechas por los estimadores, y entre ellas se encuentran los bosques aleatorios.

3.7.1. Bosque Aleatorio

Los bosques aleatorios o *Random Forest* (RF) son una combinación de predictores de tipo árbol tal que cada árbol depende de los valores de un vector aleatorio muestreado de forma independiente y con la misma distribución para todos los árboles del bosque [94]. El error de generalización de un bosque aleatorio depende de la efectividad de los árboles individuales

que lo componen y la correlación que hay entre ellos. El bosque más simple con características aleatorias se forma seleccionando al azar, en cada nodo, un pequeño grupo de variables de entrada para realizar la división (*split*). Los bosques aleatorios son una buena opción porque naturalmente manejan problemas de regresión y clasificación, son relativamente rápidos para entrenar y predecir, dependen solo de uno o dos parámetros de ajuste, se pueden usar en problemas de alta dimensión y fácilmente se pueden implementar en paralelo [95].

Retomando [95], para un vector aleatorio p -dimensional $\mathbf{X} = (X_1, \dots, X_p)^T$ que representa los valores reales de entrada y una variable aleatoria Y que representa la respuesta de valores reales, asumimos una distribución conjunta desconocida $P_{XY}(X, Y)$. El objetivo es encontrar una función de predicción $f(X)$ para predecir Y . La función de predicción es determinada por una función de pérdida o *función loss* $L(Y, f(X))$ definida para minimizar el valor esperado de la pérdida

$$E_{XY}(L(Y, f(X))) \quad (3)$$

donde los subíndices denotan la expectativa con respecto a la distribución conjunta de X e Y . $L(Y, f(X))$ es una medición de qué tan cerca $f(X)$ es a Y , L penaliza los valores de $f(X)$ que están distantes de Y . En este caso L es la función de error cuadrático medio. Minimizar (3) para la pérdida de error cuadrático da la expectativa condicional también conocida como la función de regresión:

$$f(x) = E(Y | X = x) \quad (4)$$

Los ensambles construyen la función de regresión f en términos de una colección que se conoce como aprendices base o *base learners* $h_1(x), \dots, h_J(x)$. Estos elementos individuales son combinados para dar el ensamble predictor $f(x)$. En el caso de los problemas de regresión, el resultado de cada árbol es promediado, teniendo que:

$$f(x) = \frac{1}{J} \sum_{j=1}^J h_j(x) \quad (5)$$

En los bosques aleatorios, el j -ésimo estimador es un árbol denotado por $h_j(X, \Theta_j)$, donde Θ_j es una colección de variables aleatorias y todos los Θ_j son independientes para $j = 1, \dots, J$. Este proceso de realizar una estimación tomando el promedio de los árboles individuales también se conoce como el método *Bagging* y se muestra en la Figura 4.

Los parámetros básicos de configuración de un bosque aleatorio son:

- **Número de estimadores**, cantidad de árboles en el bosque
- **Número de características**, aleatorias para construir cada árbol
- **Criterio de división**, comúnmente utilizado el error cuadrático medio
- **Bootstrap**, si el muestreo de datos se realiza o no con reemplazo

En realidad los parámetros de configuración de un bosque son los mismos que los de un árbol, con la diferencia de que al construir un bosque aleatorio es indispensable definir el

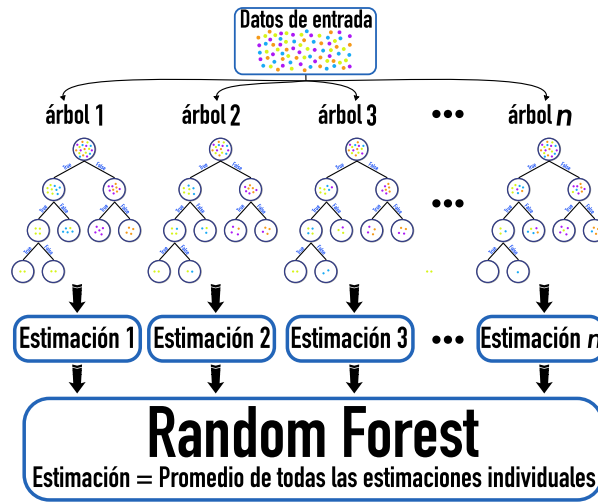


Figura 4: El resultado final de un bosque aleatorio depende directamente de los árboles que lo componen.

número de árboles a utilizar y el tipo de muestreo realizado para la construcción de los mismos.

3.7.2. AdaBoost

Uno de los enfoques más conocidos para mejorar la capacidad de aprendizaje (generalización) de los algoritmos son los métodos de ensamble [100], [97]. De manera general podemos decir que para construir un ensamble necesitamos de un conjunto de datos dado de n muestras y m características $\mathbf{D} = \{(x_i, y_i)\} (|\mathbf{D}| = n, x_i \in \mathbb{R}^m, y_i \in \mathbb{R})$, un modelo de ensamble ψ que utiliza una función de agregación G la cual agrega K inductores $\{f_1, \dots, f_k\}$ para predecir una sola salida de la siguiente manera:

$$\hat{y}_i = \psi(x_i) = G\{f_1, \dots, f_k\} \quad (6)$$

donde $\hat{y}_i \in \mathbb{R}$ para un problema de regresión [100]. *Boosting* es un método de ensamble introducido por Schapire [98] como un método para potenciar estimadores débiles con bajo rendimiento. Su evolución llevó al método conocido como *AdaBoost* (*Adaptive Boosting*) [99].

AdaBoost es una técnica que se aplica a aquellos algoritmos que tienen un sesgo bajo, en particular a árboles de decisiones y redes neuronales [97]. *AdaBoost* forma parte de un grupo llamado *Boosting methods*, en donde el conjunto de entrenamiento se filtra por un primer estimador que discrimina aquellos puntos del conjunto con baja precisión, un segundo estimador toma estos datos y realiza nuevas predicciones, los datos de entrada se van pasando de un estimador a otro hasta mejorar la precisión de los resultados. En otras palabras, se generan estimadores complementarios entrenando cada uno con los errores cometidos por el estimador anterior [17], se les asigna una ponderación a los datos estimados incorrectamente y se ejecuta el entrenamiento nuevamente. En las próximas iteraciones la ponderación de las observaciones estimadas incorrectamente da una mayor probabilidad de estimación correcta porque el modelo le dará más importancia a estos datos [101]. El diagrama que representa el funcionamiento del algoritmo se muestra en la Figura 5.

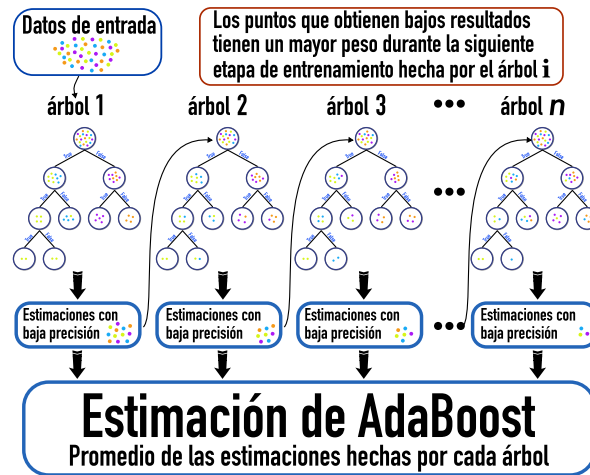


Figura 5: A diferencia de RF donde cada árbol tiene su propio conjunto de datos, en AdaBoost los árboles toman los puntos de la etapa de entrenamiento del árbol anterior cuyas estimaciones tuvieron baja precisión.

3.8. Redes Neuronales Artificiales

En este trabajo de investigación también se usaron algoritmos de redes neuronales artificiales (ANN) en el proceso de estimación de tiempo de finalización y costo total de un proyecto de desarrollo de software. Las ANNs pueden ser consideradas como un intento de modelar la capacidad del sistema nervioso para procesar información [27]. En 1943 Warren McCulloch y Walter Pitts presentaron el primer modelo de neuronas artificiales [72]. Durante la segunda mitad del siglo veinte surgen las redes neuronales con el concepto de perceptrón inventado por Frank Rosenblatt en 1957 y construido en 1958 [18].

El perceptrón representa una suma ponderada y las ponderaciones se representan con la letra w como se muestra en la Figura 6. La respuesta generada por el perceptrón es y_1 y su ecuación está dada por:

$$y_1 = w_0 + w_1x_1 + w_2x_2 \quad (7)$$

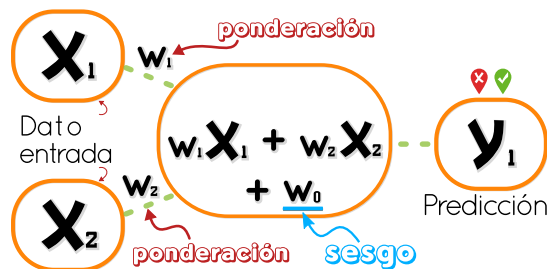


Figura 6: El perceptrón es un elemento que recibe datos de entrada y genera un resultado llamado predicción.

y_1 contiene dos operaciones básicas, la multiplicación escalar y la suma de productos. El procedimiento es muy similar a lo que en álgebra de vectores se conoce como combinación lineal, es decir, y_1 es una combinación lineal de x_1 y x_2 ya que existen escalares w_0, w_1, w_2 tales que se cumple (7). La predicción y_1 no necesariamente es unidimensional, la salida o resultado

también puede ser un vector con más de una dimensión, aunque el caso más sencillo arroja como resultado un número real.

Este esquema es parecido a una regresión lineal, pues y_1 representa rectas ancladas al origen. Luego, para completar el concepto general de recta falta agregar la ordenada al origen de la ecuación de la recta, en este caso w_0 representa a $\mathbf{b}$ u ordenada al origen que en la Figura 6 se marca como sesgo.

3.8.1. Función de activación

La evolución del modelo que resuelve problemas más complejos que la clasificación binaria agrega una perturbación que modifica la red, de tal manera que permita simular procesos no lineales llamada función de activación [76], es decir, la red ya no está limitada a valores de salida binarios (0 y 1), pues una vez calculada una suma ponderada en cada perceptrón, se aplica la función no lineal al resultado obtenido [19]. La función de activación, en general, intenta mejorar la contribución de la señal recibida y limita la amplitud de la señal procesada en el cuerpo celular [21]. Una de las funciones más utilizadas en redes de predicción y clasificación es la *sigmoide* definida por la siguiente ecuación:

$$f(\sum) = \frac{1}{1 + e^{-\sum}} \quad (8)$$

donde $\sum$ es la suma ponderada de entradas. La gráfica de la *sigmoide* es una curva suave como se muestra en la Figura 7a. En la literatura existen distintas funciones de activación y según el contexto se puede utilizar una u otra. Algunas de las funciones más conocidas son la tangente hiperbólica, la función *signum* (escalón) y la ReLU. En este trabajo, durante la etapa de experimentación se probarán distintas funciones para elegir aquella que arroje mejores resultados.

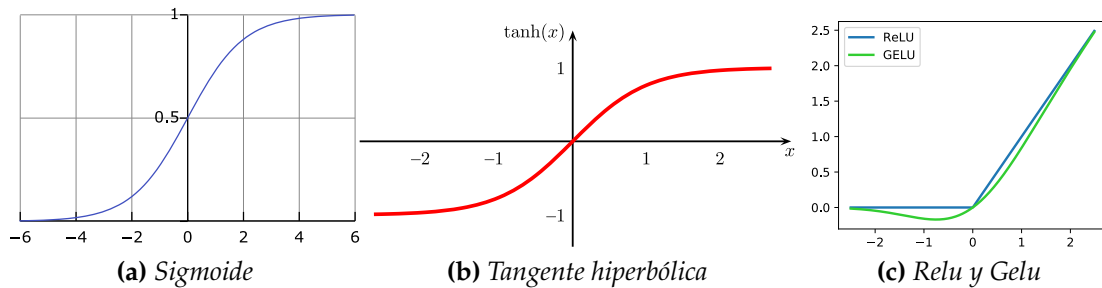


Figura 7: Imagen de sigmoide creada por Qef (talk), imagen de tanh por Geek3, CC BY-SA 3.0, imagen de Relu por Ringdongdang, CC BY-SA 4.0 wikimedia.org.

3.8.2. Neurona

La neurona es la unidad básica de procesamiento en una red neuronal. De manera simplificada, en una neurona ocurre una suma ponderada de señales que llegan a la entrada y se somete el resultado a una función de transferencia [14], como ya se explicó con el perceptrón. Este proceso se muestra en la Figura 8. Los datos de entrada se someten a una función de transferencia que se encarga de sumar el producto de las ponderaciones con los datos de entrada, después se agrega el sesgo para someter este resultado a la función de activación.

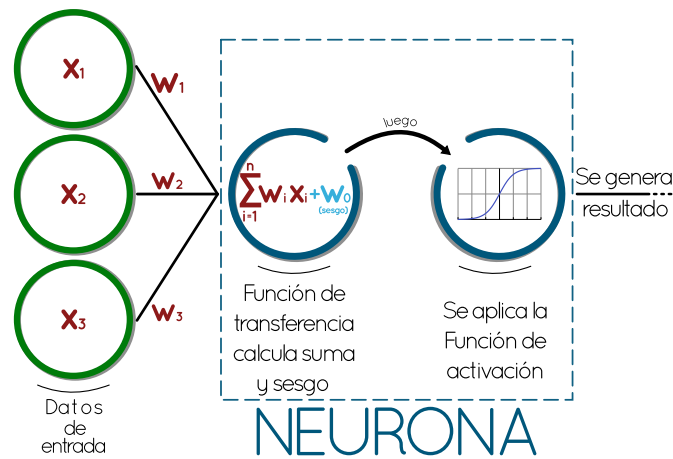


Figura 8: Operaciones que se realizan en una neurona.

El conjunto de perceptrones multicapa crea la estructura de red neuronal en donde las neuronas se organizan en capas y las conexiones van de una capa a otra como se muestra en la Figura 9. El número de neuronas y capas lo determina el usuario al crear la estructura de la red.

Los datos conectan con la capa de entrada, las capas ocultas realizan una función matemática creando un predictor, y la capa de salida recopila las predicciones y produce el resultado final [18].

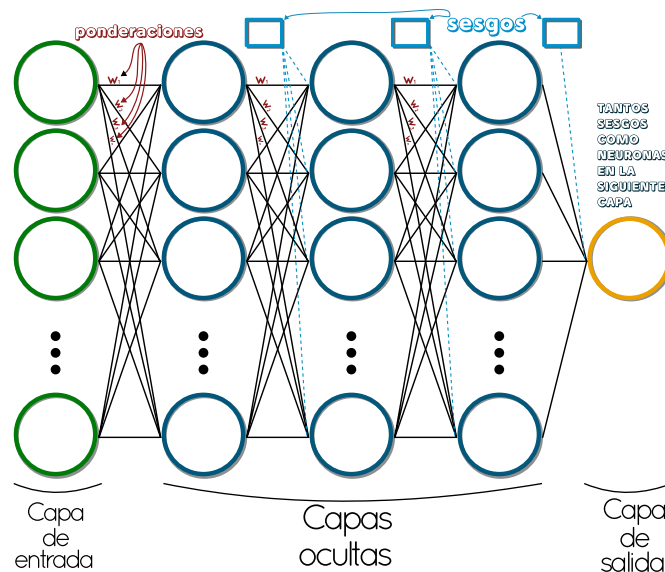


Figura 9: Esquema general de una red neuronal.

La información viaja de la capa de entrada a las capas ocultas, hasta llegar a la capa de salida y producir un resultado. Esto es importante para explicar el concepto de propagación hacia adelante o *forward propagation*.

3.8.3. Propagación hacia adelante (Forward Propagation)

Los datos de entrada de la red se transmiten a lo largo de los enlaces que establecen las neuronas. Un dato de entrada se multiplica por las ponderaciones asociadas de los enlaces, la i -ésima neurona oculta recibe como entrada la suma ponderada de la neurona de la capa anterior y somete esta suma a la función de activación. Los datos pasan de una capa a otra hasta llegar a la capa de salida, así, la i -ésima neurona de salida recibe la suma ponderada de los valores procedentes de las neuronas ocultas y, de nuevo, lo somete a la función de transferencia [14]. El proceso de propagar de esta manera los valores de los atributos de la entrada de la red a su salida se llama propagación hacia adelante.

3.8.4. Función de pérdida (loss)

Es necesario definir una función para medir el error cometido por la red durante el entrenamiento. Cuando se trabaja con problemas de regresión, la función *loss* o función de pérdida representa la distancia entre un dato predicho y su valor real. Algunas de las funciones más conocidas son el error cuadrático medio, el error absoluto medio y el error porcentual absoluto medio. Durante el proceso de entrenamiento, las ponderaciones y sesgos se actualizan hasta que se alcanza la diferencia mínima entre los resultados predichos por el modelo y los resultados reales. Esto se conoce como proceso de optimización [22] y en el caso de las redes neuronales usamos retropropagación o *backpropagation* para calcular las derivadas de la función de pérdida con respecto a las ponderaciones.

3.8.5. El algoritmo de retropropagación

El algoritmo de retropropagación es un método numérico que logró conectar con la inteligencia artificial en los años de 1985. En [27] se presenta el algoritmo de retropropagación basado en un enfoque gráfico, el cual se retoma en esta sección.

El algoritmo de retropropagación busca el mínimo de la función de error en el espacio de la ponderación (*weight space*) utilizando el método de descenso de gradiente. La combinación de ponderaciones que minimiza la función de error (*loss*) se considera una solución del problema de aprendizaje, es decir, obtener los valores mínimos posibles para realizar una buena predicción.

Los valores de las ponderaciones se eligen de manera aleatoria al iniciar el entrenamiento y el algoritmo de retropropagación se encarga de realizar las correcciones necesarias mediante los siguientes pasos:

1. Se hace el cálculo de tipo *feedforward*
2. Se aplica retropropagación a la capa de salida
3. Se aplica retropropagación a las capas ocultas
4. Se actualiza el valor de las ponderaciones

El cálculo de las derivadas de las funciones de activación se guardan en cada unidad

(neurona). El algoritmo inicia desde la capa de salida hacia la capa de entrada, y el algoritmo se detiene cuando el valor de la función *loss* se ha vuelto lo suficientemente pequeño. Así, el algoritmo de descenso del gradiente estándar calcula la pérdida y las derivadas utilizando todo el conjunto de datos para actualizar los parámetros de la red [22]. Existen distintas versiones del algoritmo descenso del gradiente también llamadas optimizadores, como el descenso del gradiente estocástico (SGD), el algoritmo *Adam*, *Adadelata*, *Adagrad*, *RMSprop*, entre otros. *Adam* suele superar a SGD al entrenar modelos de aprendizaje profundo y forma parte del estado de arte [138].

3.8.6. Tasa de aprendizaje

La tasa de aprendizaje se define en el contexto del algoritmo de optimización y define la longitud de cada paso o en términos simples, qué tan grandes pueden realizarse las actualizaciones de las ponderaciones en cada iteración [28]. Un valor por defecto que se utiliza en varios escenarios es una tasa de 0.001. De acuerdo al tipo de problema este valor puede aumentar o disminuir ligeramente.

3.8.7. Red Neuronal Recurrente

El siguiente algoritmo estudiado en este trabajo de investigación es la red neuronal que mejora la arquitectura del perceptrón multicapa incorporando el concepto de memoria: las redes recurrentes o RNN (*Recurrent Neural Network*). Matemáticamente una red recurrente simple se formula como:

$$\begin{aligned} h(t) &= f_H (W_{IH} x(t) + W_{HH} h(t-1)) \\ y(t) &= f_O (W_{HO} h(t)) \end{aligned} \quad (9)$$

donde $x(t)$ es el vector de entrada, $y(t)$ el vector de salida. W_{IH} , W_{HH} , W_{HO} , son las matrices de las ponderaciones y f_h , f_o son las funciones de activación ocultas y la de salida respectivamente. En la Figura 10 se muestra una celda recurrente simple y una celda LSTM (*Long Short Term Memory*).

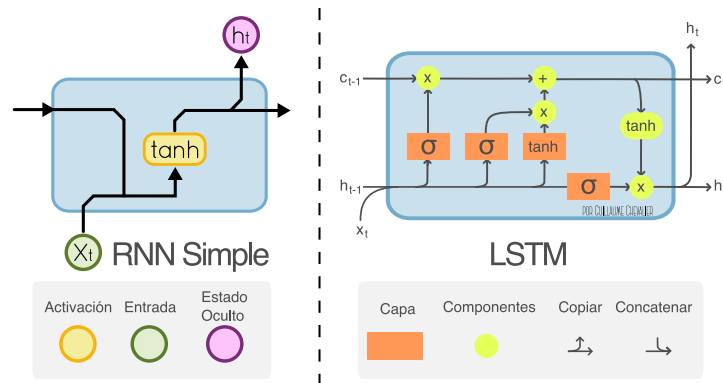


Figura 10: Una red neuronal recurrente incorpora el concepto de memoria implementando el estado oculto y una capa de activación a la entrada.

3.8.8. Autoencoders

En un perceptrón multicapa, si el número de unidades ocultas es menor que el número de entradas, la primera capa realiza una reducción de dimensionalidad [17]. Tradicionalmente un *Autoencoder* tiene una arquitectura tipo MLP donde hay tantas salidas como entradas, pues el objetivo es reconstruir la información de entrada al propagarse por las capas de la red.

Un codificador automático es una red neuronal que está entrenada para intentar copiar los datos de entrada entrada y recrearlos a la salida, por lo general se restringen a realizar copias aproximadas y esto permite que el algoritmo pueda priorizar qué aspectos de la entrada deben copiarse, aprendiendo propiedades útiles de los datos de entrenamiento [114].

En la Figura 11 se muestra la estructura de un AE, la red consiste de dos partes: una función codificadora $h = f(x)$ y una función decodificadora que produce una reconstrucción de la información $r = g(h)$. El objetivo es mapear una entrada x a una salida (reconstrucción) r a través de una representación interna h . En otras palabras, el codificador automático tiene dos componentes: una función f que asigna x a h , y el decodificador g que asigna h a r . Cuando un codificador automático cuya dimensión en las capas ocultas es menor que la dimensión de entrada se denomina subcompleto [114]. El aprendizaje de una representación incompleta obliga al codificador automático a capturar las características más destacadas de los datos de entrenamiento.

En la Figura 11, f representa las capas ocultas en donde la dimensión (número de neuronas) disminuye, y g representa las capas en donde la dimensión aumenta. La red neuronal es entrenada para encontrar la mejor codificación de las entradas en las unidades ocultas, realizando una reducción de dimensionalidad. Se ha probado que un *Autoencoder* con una capa oculta implementa el análisis de componentes principales (PCA) [115], esta ventaja con respecto a una arquitectura MLP clásica permite implementar la técnica de componentes principales y la regresión en un solo modelo, agregando una capa extra a la salida. Esta capa adicional se compone de una sola neurona debido a que la salida esperada no es una reconstrucción de los datos, lo que se espera es un número real que representa el tiempo de finalización o el costo total de un proyecto de software.

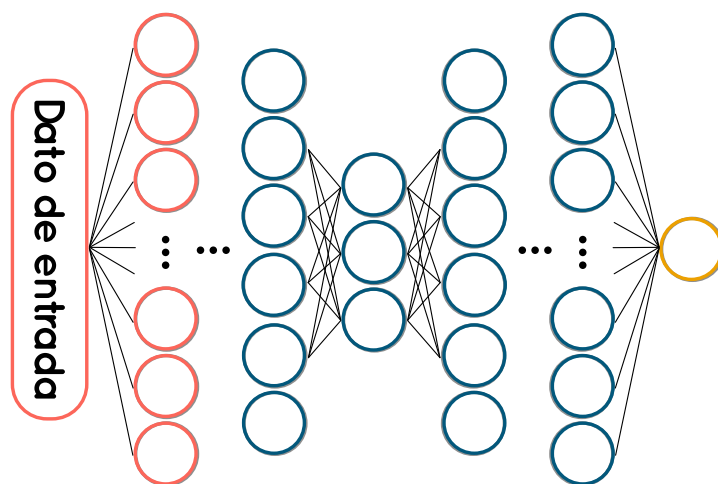


Figura 11: Un Autoencoder refleja la mitad de su arquitectura, teniendo un primer bloque (Encoder) que comprime la información y un segundo bloque (Decoder) que descomprime la información.

3.8.9. Diseño de un modelo de red neuronal

Para diseñar una red hay que considerar que la capacidad de cómputo es limitada, por lo que el diseño de la red debe ser simple. Moolayil J. [28] recomienda un conjunto de reglas para el diseño de una arquitectura de red, entre las que destacan:

- Comenzar con una red de una sola capa oculta y un número de neuronas aleatorio que puede ir desde 10 hasta 300. Entrenar y evaluar el modelo, si los resultados no son convincentes se agrega una nueva capa oculta con el mismo número de neuronas.
- Si una red pequeña no da buenos resultados es posible duplicar o triplicar el número de neuronas, hasta 1000 por capa y aumentar el valor de la regularización.
- Si el incremento en número de neuronas no mejora los resultados entonces hay que incrementar la profundidad de la red agregando más capas (manteniendo el *dropout*).
- Aplicar distintos valores de *dropout* o ruido gaussiano con el fin de mejorar la capacidad de generalización de la red.

En caso de que ninguna configuración arroje buenos resultados y como lo recomienda [28], entonces es necesario revisar que el conjunto de datos es apropiado para las redes neuronales o que estos se encuentren normalizados y codificados, de lo contrario será necesario aplicar otras técnicas de aprendizaje automático.

Además de la arquitectura de red, es indispensable conocer el rendimiento de la misma con distintas funciones de pérdida (*loss*), optimizadores y tasas de aprendizaje. Para evaluar la precisión y capacidad de generalización de la red se aplica validación cruzada o *k-fold cross validation*. La etapa de entrenamiento se mejora reduciendo el número de épocas a través de la inicialización de las ponderaciones y los sesgos. La atención no solo se fija en el diseño del modelo de red, también es crucial preparar el conjunto de datos aplicando normalización y organizando la información en lotes. Al escalar los datos entre cero y uno se logra reducir el número de épocas necesarias para disminuir la métrica aplicada a un valor cercano a cero.

3.8.10. Regularización

La regularización es un proceso para reducir el *overfitting* o sobreentrenamiento. Es un método para penalizar las ponderaciones del modelo en caso de sobreajuste [28].

Regularización mediante la técnica de *Dropout* En este método el modelo desactiva de manera arbitraria algunas de las neuronas de las capas ocultas durante cada iteración del entrenamiento, lo que provoca que en cada época el modelo tenga una estructura ligeramente diferente impactando en la retropropagación y en la asignación de ponderaciones (a la salida de una neurona desactivada se tiene un valor de cero). Si bien la regularización por *dropout* no ayuda a disminuir la función *loss*, es útil para mejorar la capacidad de generalización de la red [28].

En la Figura 12 se marcan en rojo las neuronas que se desactivan durante una iteración. La cantidad de neuronas a desactivar en cada capa se determina por el valor de configuración que está entre cero y uno. Un valor de 0.2 indica que el 20 % de las neuronas se desactivan, un

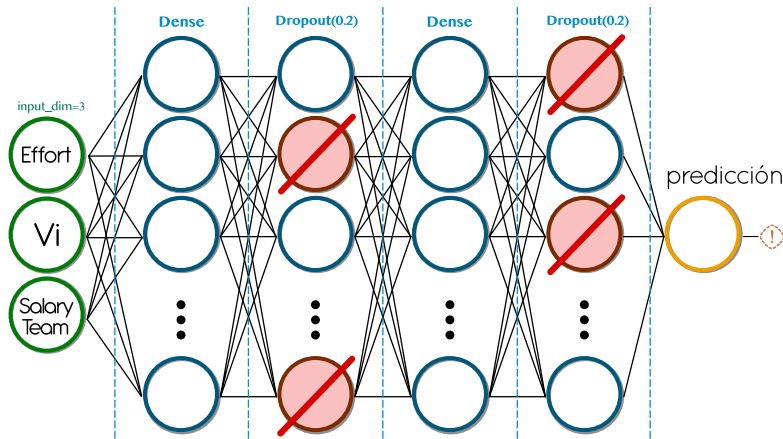


Figura 12: Ejemplo de red neuronal con capas de dropout.

valor de uno implica que el 100 % de las neuronas se desactivan. Es común el hecho de que el porcentaje de configuración de *dropout* no supere el 35 %, su valor se determina empíricamente y conviene utilizarlo cuando la arquitectura de red neuronal no da buenos resultados durante la validación.

El caso particular de la implementación de una capa de tipo *dropout* mediante el API de *Keras* conserva la misma estructura que una capa de tipo densa, es decir, todas las neuronas están conectadas con los elementos de la capa siguiente. Una capa de *dropout* es idéntica a una capa densa, con la diferencia que es posible controlar mediante el parámetro *rate* la cantidad de neuronas a desactivar.

Ruido Gaussiano En [22] se explica que una de las formas de mejorar la precisión del modelo consiste en agregar un factor de aleatoriedad a los valores de cada neurona que componen la red. Una capa de ruido gaussiano aplica ruido aditivo centrado en cero o también conocido como *Additive White Gaussian Noise* (AWGN) . Esto es útil para mitigar el sobreentrenamiento y es una forma de aumento de datos aleatorio. Al ser ésta una capa de regularización, solo está activa en tiempo de entrenamiento.

3.9. Análisis de componentes principales

La técnica *Principal Component Analysis* (PCA) [76] permite aplicar reducción de dimensionalidad a un conjunto de datos basándose en la descomposición de valores singulares. Este procedimiento es equivalente a encontrar un conjunto de matrices ortogonales que transforman a una matriz A , esto es:

$$A = U\Sigma V^T \quad (10)$$

donde U y V son matrices ortogonales y Σ es una matriz diagonal con los valores singulares de A . Con esta técnica se obtiene que el esfuerzo es la característica más importante para realizar las estimaciones de tiempo y costo. Este procedimiento se resume en el diagrama de la Figura 13.

Reducción de dimensionalidad

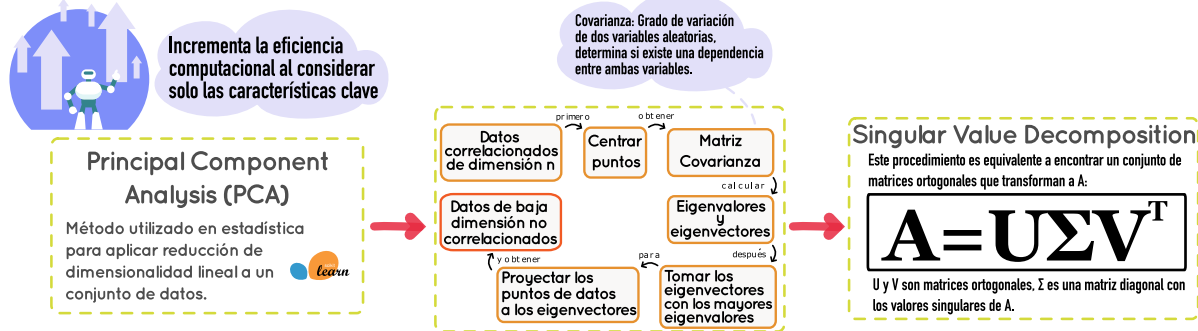


Figura 13: Con la técnica PCA se pueden conocer las características más importantes para realizar las estimaciones de los proyectos de software.

Otra técnica utilizada para reducir la dimensionalidad de un conjunto de datos es la eliminación recursiva de características con ajuste automático del número de características aplicando validación cruzada (RFECV) [137]. Dado un estimador (modelo de aprendizaje automático), el objetivo consiste en considerar recursivamente conjuntos de características cada vez más pequeños. Primero, el estimador se entrena con el conjunto inicial de características y la importancia de cada característica se obtiene a través del atributo de la implementación que pondera cada característica denominado como *feature importances*, a este atributo se le asigna un valor, cuanto más alto más importante es la característica. La importancia de una característica se calcula como la reducción total (normalizada) del criterio aportado por esa característica, conocido como la importancia de Gini. Con este resultado las características menos importantes se eliminan del conjunto actual de características. Ese procedimiento se repite recursivamente en el conjunto podado hasta que finalmente se alcanza el número deseado de características para seleccionar.

3.10. Discretización de variables continuas

La discretización es una forma de tratar con las probabilidades de una variable continua de valor real. Este procedimiento crea grupos en un conjunto de datos y la manera de discretizar puede influenciar la interpretación de los mismos, el no aplicar ninguna discretización puede llevar a una interpretación incorrecta de la información [18, 14].

La discretización puede ser una estrategia útil en escenarios donde una característica numérica debería comportarse más como una característica categórica [34]. Cuando los datos sin procesar están en forma numérica, generamos intervalos discretos y asignamos cada valor a uno de estos intervalos, generando una cuantificación de datos continuos [146]. Consideramos tres estrategias distintas para realizar este procedimiento:

- Uniforme: la estrategia uniforme utiliza intervalos de ancho constante
- Cuantiles: la estrategia de cuantiles utiliza los valores de los cuantiles para tener intervalos igualmente poblados en cada característica
- K-medias: la estrategia k-medias define intervalos basados en un procedimiento de agrupamiento (*k-means*) realizado en cada función de forma independiente. Los valores en cada intervalo tienen el mismo centro más cercano de un grupo de k-medias.

3.11. Conjuntos de entrenamiento, validación y prueba

La razón para no evaluar los modelos de aprendizaje automático con los mismos datos con los que fueron entrenados es que durante la ejecución, después de unas pocas épocas, los modelos presentan sobreajuste. El desempeño con los datos de entrenamiento siempre mejora a medida que avanzan las iteraciones, al evaluar con los mismos datos el algoritmo ya conoce la información y producirá resultados muy similares, pero cuando se presentan nuevos datos, el algoritmo empeora o se estanca [117], ya que memorizó la información y no fue capaz de extraer patrones presentes del conjunto.

El objetivo del aprendizaje automático consiste en encontrar modelos capaces de aproximar una función, parte de las estrategias para mitigar el sobreajuste y maximizar la generalización consiste en la división de los datos en tres conjuntos básicos: entrenamiento, validación y prueba. Durante la selección de los algoritmos, estos se entrenan con los datos de entrenamiento y se verifican con los datos de validación. Una vez que termina el proceso de ajuste de parámetros del algoritmo utilizando los conjuntos de entrenamiento y validación, el modelo se somete a prueba utilizando el conjunto de datos no procesados previamente (*test set*) [117]. Así, podemos definir tres tipos de muestras [112]:

- Muestra de entrenamiento: dato utilizado para entrenar un algoritmo de aprendizaje
- Muestra de validación: dato utilizado para verificar el desempeño de los parámetros de un algoritmo de aprendizaje (hiperparámetros)
- Muestra de prueba: dato utilizado para evaluar el desempeño de un algoritmo de aprendizaje. La muestra de prueba está separada de los datos de entrenamiento y validación y no está disponible en la etapa de aprendizaje.

El entrenamiento permite construir el modelo de aprendizaje automático. A través de la validación es posible determinar si la configuración de los hiperparámetros lleva al algoritmo a una etapa de estancamiento. Si al monitorear las métricas de evaluación (errores) estas aumentan su valor, el algoritmo no está siendo capaz de realizar estimaciones cercanas a los valores reales y su capacidad de generalización no es la adecuada, por lo tanto la brecha entre los errores del conjunto de entrenamiento y conjunto de validación será grande.

El conjunto de datos de prueba se utiliza una vez finalizado el ajuste de parámetros para ver el desempeño real del algoritmo aplicado a nuevos datos. El conjunto de prueba contiene información desconocida por el algoritmo, es decir, son valores que el modelo nunca ha visto o ha utilizado durante la etapa de entrenamiento ni de validación.

Un ejemplo de dividir el conjunto de datos para evaluar un algoritmo consiste en utilizar un conjunto de muestras de las cuales el 70 % se reserva para entrenamiento y el 30 % para prueba [70] como se muestra en la Figura 14. Si utilizamos el conjunto de prueba para seleccionar los parámetros del modelo, agotamos los datos de prueba, y usar los mismos datos para evaluar qué tan bien funcionará nuestro modelo en el futuro conducirá a estimaciones demasiado optimistas [119]. Así, de todo nuestro conjunto de datos, reservaremos dos tercios para realizar el entrenamiento utilizando validación cruzada y un tercio como datos de prueba.

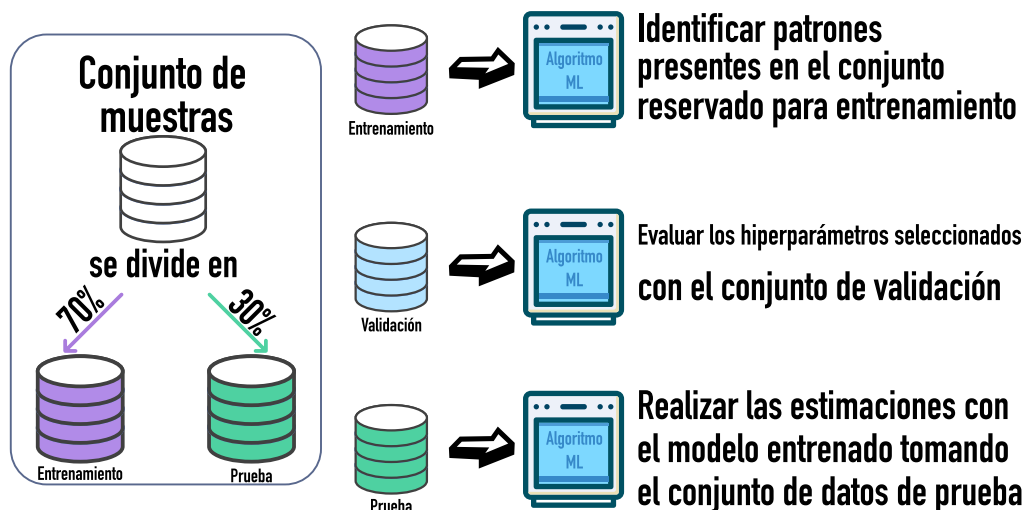


Figura 14: Conjuntos utilizados durante el desarrollo de un algoritmo de aprendizaje automático.

3.12. Validación Cruzada (k-Fold Cross Validation)

Un enfoque de selección de modelos es la validación cruzada o *cross-validation* (CV), método general formalizado en la década de los 70's por M. Stone y S. Geisser [116]. La idea del método consiste en dividir los datos en dos conjuntos, la mayoría de los datos se utilizan para entrenar el modelo y desarrollar el algoritmo predictivo, la segunda parte de los datos son observaciones que se aplican para realizar las estimaciones del modelo entrenado [18].

En la validación cruzada *k-Fold* el conjunto de datos se divide aleatoriamente en k subconjuntos de igual tamaño. Uno de estos subconjuntos es utilizado como el conjunto de validación, mientras que el resto $k - 1$ forman el conjunto de entrenamiento. Un modelo es entrenado con cada partición, si $k = 10$ entonces se entrenarán 10 instancias de un mismo modelo y cada subconjunto que se utiliza como conjunto de validación se utiliza una sola vez al entrenar una instancia del modelo.

Un ejemplo de la partición de datos se muestra en la Figura 15, en verde se representan los elementos que forman parte del conjunto de entrenamiento y en amarillo los datos reservados para validación. La idea representada en la Figura 15 implica que si $k = 10$, de las 10 particiones, una de ellas se utiliza como validación, y a medida que se ejecutan todas las particiones la ventana de elementos de validación se va recorriendo hasta utilizar todos los elementos del conjunto de datos.

CV evita el sobreajuste porque la muestra de entrenamiento es independiente de la muestra de validación [113]. Los puntajes de validación pueden tener una variación alta con respecto a la división de validación si el conjunto de datos es pequeño, por eso una de las mejores prácticas en estas situaciones es utilizar la validación *k-Fold*, ya que el puntaje de validación para el modelo utilizado es el promedio de los k puntajes de validación obtenidos [117]. Las proporciones de las diferentes particiones deben ser consistentes en cada muestra de entrenamiento y validación. En el caso de *k-Fold*, se ha estudiado que $k = 10$ proporciona resultados cercanos a los óptimos [113].



Figura 15: Partición de datos al aplicar validación cruzada de tipo 10-Fold.

3.13. Otras técnicas de Inteligencia Artificial

De acuerdo al contexto del problema cada autor elige los modelos y heurísticas que aplicará para realizar las estimaciones de esfuerzo en desarrollo de software. existe una amplia gama de algoritmos de inteligencia artificial que han permitido el estudio de la precisión de los algoritmos para predecir patrones en la información. Las principales técnicas se describen a continuación [21, 14, 20, 146, 38, 147, 148, 40, 28]:

Maquina de Soporte Vectorial: o *Support Vector Machine* (SVM) es un algoritmo de aprendizaje automático aplicado a problemas de clasificación y regresión que busca encontrar un hiperplano que separe mejor los datos en diferentes clases en un espacio de alta dimensión. En regresión se busca predecir un valor de salida continuo a través de líneas o hiperplanos que mejor se ajusten a los datos y predice la salida en función de la ubicación de una nueva muestra en relación con la línea o hiperplano.

Sistemas Expertos Inteligentes: sistemas que simulan la experiencia humana en un dominio específico. Compuestos por una base de conocimiento y un motor de inferencia, el sistema aplica reglas para inferir el conocimiento y automatizar la toma de decisiones a nivel de experto.

Soft Computing: rama de la inteligencia artificial capaz de trabajar con información incompleta, compleja e imprecisa. Es una colección de técnicas que incluye lógica difusa, redes neuronales, algoritmos evolutivos y métodos probabilísticos.

Lógica Difusa: es un tipo de lógica que estudia el razonamiento lógico con vaguedad o imprecisión. Un sistema difuso recibe la información y convierte a conjuntos difusos utilizando una función matemática μ , a través de inferencia difusa produce un valor de las variables estimadas, teniendo una capa de entrada, una capa de procesamiento y una capa de salida que produce los resultados esperados.

Programación Genética: es un tipo de algoritmo evolutivo que aplica el concepto de selección natural para evolucionar una población y seleccionar a los mejores individuos mediante una función de evaluación conocida como *fitness*. Los principios básicos son la reproducción, mutación y selección. El algoritmo genera distintas generaciones seleccionando aquellos individuos con las mejores características, este proceso se repite hasta alcanzar un criterio de paro definido.

Redes Neuronales: de base radial son redes de tres capas (entrada, oculta y de salida) que realizan un mapeo no lineal a través de una función de base radial definida por un centro y un ancho que se utiliza para modelar la relación entre las entradas y salidas. Redes neuronales de tipo *Wavelet* utilizan funciones matemáticas que se pueden usar para representar señales e imágenes de manera compacta. Este tipo de redes usan *wavelets* para preprocesar los datos de entrada y extraer características que posteriormente se utilizan como entrada para una red neuronal tradicional de tipo *feedforward*. Una red neuronal de picos (*spiking*) es una red neuronal que utiliza pulsos discretos de información para comunicarse entre neuronas, consideradas como la tercera generación de redes neuronales. Estas redes no satisfacen las condiciones de continuidad y diferenciabilidad, por lo que no es posible aplicar directamente un algoritmo de retropropagación. Las redes de tipo *Functional Link* usan una sola capa oculta, y cada nodo está conectado a un enlace funcional. Estas redes aumentan las dimensiones del espacio de entrada utilizando una combinación no lineal de entrada para manejar problemas linealmente no separables.

Aprendizaje profundo: en el contexto de las redes neuronales multicapas, una red profunda es aquella red cuya arquitectura contiene varias capas ocultas dándole profundidad a la red. El aprendizaje profundo o *Deep Learning* logra resolver las limitaciones del aprendizaje automático como por ejemplo al trabajar con imágenes y otros tipos de datos no estructurados. La ventaja del aprendizaje profundo es que con grandes conjuntos de entrenamiento el rendimiento del modelo mejora

4. Trabajo relacionado

4.1. Antecedentes

La estimación de esfuerzo en desarrollo de software existe desde la década de 1950 [58]. Entre los años de 1950 y 1970 esta actividad se realizaba de manera manual basada en el juicio de expertos [69]. A medida que el software iba creciendo y se volvía más complejo, las prácticas de la ingeniería de software fueron evolucionando dando lugar a modelos que satisficieran los requerimientos asociados al desarrollo de ese entonces. A partir de mediados de la década de 1970 y durante los años 80's se introdujo una oleada de métodos de estimación de software.

Algunos de los modelos de estimación que surgieron en ese periodo incluyen a los modelos *Doty*, *SLIM*, *Checkpoint*, *Price-S*, *SEER*, *IBM-FSDmodel* y *COCOMO* [139] [121]. Esta fue una respuesta a uno de los problemas naturales de la estimación; cuando los proyectos son pequeños es relativamente fácil su estimación pero a medida que el proyecto crece también lo hace la dificultad de la estimación, por lo que es indispensable la aplicación de mejores métodos y técnicas para administrar la creciente complejidad de la estimación.

El modelo *Doty* fue desarrollado para la Fuerza Aérea de Estados Unidos en 1977 e incluye una serie de factores relacionados al desarrollo como las restricciones de consumo de recursos (CPU), la operación en tiempo real, el cambio en los requerimientos operativos, si el desarrollo se realiza en el lugar de operaciones o en un lugar distinto, entre otros factores que contribuyen a la estimación de costos [123].

El modelo *SLIM* fue creado por L.H. Putnam de manera empírica en 1978 examinando datos e intentando determinar el comportamiento funcional por análisis empírico [120]. La propuesta muestra que los sistemas de software siguen un patrón de ciclo de vida con dos parámetros fundamentales: el esfuerzo del ciclo de vida (K), el tiempo de desarrollo (t_d), y una función de estos, la dificultad ($\frac{K}{t_d^2}$) [120]. *SLIM* depende de una estimación basada en *Source Lines of Code* (SLOC) para el tamaño general del proyecto, luego modifica esto mediante el uso del modelo de curva de Rayleigh para producir sus estimaciones de esfuerzo [121].

En 1979 Albrecht [15] propone un modelo presentando los puntos de función o *Function Points* (FP) con el objetivo de desarrollar una medida relativa del valor de la función entregada al usuario que sea independiente de la tecnología o enfoque particular utilizado [122]. Esta estrategia procesa una estimación del tamaño práctico de la programación y el gasto se determina a partir de actividades pasadas [139].

Otro modelo popular fue presentado por Boehm en 1981 [123] y se denominó Modelo de Costo Constructivo (COCOMO), con un nivel básico que calcula el trabajo en meses/hombre y los costos en función del tamaño del programa descrito en miles de instrucciones fuente entregadas estimadas o *Kilo Delivered Source Instructions* (KDSI). El nivel intermedio del modelo COCOMO considera otros factores de costo y atributos mediante una evaluación subjetiva de los productos, el personal, el hardware y los proyectos. El tercer nivel es similar al intermedio con excepción de que el proyecto se divide en cuatro fases: diseño de producto, diseño detallado, codificación y prueba unitaria, y fase de integración y pruebas [139] [121].

A lo largo de los años, estos modelos proporcionaron a los investigadores un valioso conjunto de datos para crear diferentes enfoques basados en técnicas de aprendizaje automático que generaban estimaciones usando dichos conjuntos. También durante la década de los 90's,

Shepperd, así como otros investigadores, estudiaron técnicas basadas en analogías y Hugges propuso un modelo de juicio experto basado en la experiencia [15].

4.2. Desarrollo Tradicional

Con la evolución de los modelos de estimación, a mediados de los años 80's el estado del arte resaltó los problemas pendientes que necesitaban más investigación [123]: estimación del tamaño del software, métricas de tamaño y complejidad del software, atributos de costos del software (*drivers*), refinamiento del modelo de costos de software, modelos cuantitativos de dinámica de proyectos de software y de evolución del ciclo de vida, así como la recopilación de datos de software.

Con el paso de los años distintos modelos de estimación de esfuerzo fueron mejorando como el enfoque FP [125] y durante la década de los años 90's más trabajos se incorporaron al estado del arte. En el año de 1992 Bock y Klepper [126] presentaron un método simplificado denominado FP-S derivado de los puntos de función de Albrecht con la intención de reducir el tiempo requerido para la tarea de conteo sin una reducción significativa en la precisión. El método FP-S elimina la clasificación de complejidad de funciones; los autores proponen obtener un conteo total de funciones por tipo de función para aplicar un solo factor de ajuste multiplicativo a cada conteo de tipo de función y obtener, después de sumar todos los tipos de función, un conteo total de FP sin ajustar.

Los puntos de casos de uso o *Use Case Points* (UCP) fueron propuestos en 1993 por G. Karner [131]; el modelo mide la funcionalidad del sistema con un contador llamado punto de caso de uso no ajustado (UUCP) que se relaciona con otros dos valores, el factor de complejidad técnica (TCF) y los factores del entorno (EF). El resultado del análisis genera los puntos de caso de uso como una combinación del producto de UUCP, TCF y EF. UCP evolucionó como una alternativa a FP y su estudio continuó por casi dos décadas con contribuciones en la mejora y validación del modelo existente, desarrollando nuevos modelos en el método de dimensionamiento UCP original o contribuyendo a un mejor clasificado de las técnicas para estimar el esfuerzo a partir de métricas de tamaño de UCP [132].

El razonamiento basado en casos (CBR) tiene sus antecedentes en la filosofía y la psicología; en el año de 1994 F. Marir e I. Watson [128] realizaron una revisión categorizada presentando los orígenes del método, las distintas técnicas de CBR como representación de casos, indexación, memoria de casos, recuperación y adaptación de casos, así como las aplicaciones implementadas en herramientas de software y la relación que hay entre CBR y otros modelos de razonamiento. En el mismo año, los autores publicaron otro trabajo de investigación [129] enfocándose en el desarrollo de CBR en los Estados Unidos durante la década de los 80's, además de presentar las técnicas fundamentales de CBR como los vecinos más cercanos, los algoritmos de inducción, un método con estructura jerárquica conocido como el modelo de memoria dinámica, entre otras técnicas. Su investigación concluye con una actualización de las distintas aplicaciones y sistemas basados en CBR.

K. Srinivasan y D. Fisher [130] describen en 1995 dos métodos de aprendizaje automático que utilizaron para crear estimadores del esfuerzo de desarrollo de software a partir de datos históricos. Los experimentos indican que los árboles de decisión y las redes neuronales son técnicas competitivas comparadas con los estimadores tradicionales en el conjunto de datos de SLIM, COCOMO y el enfoque FP.

La contribución hecha por G.R. Finnie et al. en 1997 [62] comparó tres técnicas de estimación utilizando puntos de función como una estimación del tamaño del sistema; el análisis de regresión, las redes neuronales y el razonamiento basado en casos (CBR). Sus hallazgos resaltan que los modelos de regresión no se desempeñan de manera muy efectiva en el modelado de las complejidades del desarrollo de software, CBR permite el desarrollo de una base de casos dinámica con nuevos datos del proyecto que se incorporan automáticamente a la base de casos a medida que están disponibles, mientras que las redes neuronales requieren reentrenamiento para incorporar nuevos datos al modelo aunque el modelo ANN es capaz de capturar de manera efectiva los parámetros que influyen en la productividad. La única desventaja de la red neuronal es que no es posible obtener ninguna explicación o justificación de una estimación mediante su arquitectura.

En 1998 G. Horgan et al. [127] plantearon un nuevo modelo para producir estimaciones de puntos de función al inicio del ciclo de vida de desarrollo. El modelo llamado *Early E* consta de dos pasos, el primer paso calcula el valor único requerido (conocido como la constante de ponderación) en función de los datos históricos del proyecto. El segundo paso usa este valor junto con la cuenta de puntos de función sin procesar (RFP) para producir la estimación de FPA.

En 2019 H. T. Hoc et. al [43] realizaron una revisión de los modelos y técnicas existentes, encontrando que de las técnicas algorítmicas que se aplicaban estaban *Source lines of code* (SLOC), FPA, *Object Point*, COCOMO, *Use Case Points* y *Story Points*. Los autores hallaron que dentro de las técnicas no algorítmicas se encontraban el juicio de experto, la estimación por analogía, las técnicas *Price-to-Win*, *Bottom-up*, *Top-Down*, *Wideband Delphi* y *Planning Poker*. Además, en la referida revisión se describen los distintos modelos de regresión aplicables como es la regresión lineal de una sola variable independiente (regresión simple), regresión de variables múltiples independientes, regresión multivariante (*multi-variate regression*) y enfoques como el método *Whale-Crow Optimisation*.

La clasificación de todos estos métodos y modelos ha cambiado a lo largo de los años y distintos autores han propuesto diversas clasificaciones según la técnica que se aplica, o de acuerdo al conjunto de datos disponible, o por los modelos específicos aplicados, ya sean modelos propietarios, no propietarios, manejados por datos o modelos compuestos [69]. También se han clasificado métodos que son semiparamétricos, los que están basados en juicio de expertos, aquéllos que combinan diferentes técnicas, y por supuesto los que están basados en aprendizaje automático.

Todas las clasificaciones mencionadas en la literatura tienen un conjunto común de técnicas, cuyo nombre puede variar de una clasificación a otra pero el significado se mantiene igual, algunas categorías son particularizaciones de las clasificaciones más genéricas. Es conveniente considerar la clasificación de acuerdo a las técnicas más populares como lo propone O. Fedotova et. al [58] en su tabla de categorías, mostrada en la Tabla 1, incluyendo una columna extra con algunos ejemplos de técnicas de cada grupo.

Manju Vyas et al. [15], por ejemplo, concluyen que la mayoría de las técnicas algorítmicas son intensivas en datos y el principal inconveniente de estas técnicas es que se requieren grandes conjuntos de datos. A diferencia de los métodos algorítmicos o centrados en datos, las técnicas no algorítmicas o la estimación basada en expertos, no requieren ningún dato de medición del proyecto, la precisión y fiabilidad de la técnica depende de la experiencia del experto.

Los modelos algorítmicos se basan en el análisis estadístico de datos históricos (proyectos

Categorías		Técnica equivalente de distintos autores	Ejemplo de técnicas
Técnicas no algorítmicas	Juicio de expertos	Técnica de <i>expertise</i>	Juicio de experto (WBS, ABE)
		Opinión de experto	
		Técnica basada en <i>expertise</i>	
		Juicio de Experto	
	Basadas en analogía o aprendizaje automático	Técnicas empíricas y de aprendizaje automático	Basado en aprendizaje (Redes neuronales, árboles de decisión, lógica difusa)
		Analogy-based o de aprendizaje automático	
		Analogía	
		Técnicas orientadas al aprendizaje	
Técnicas algorítmicas	Técnicas compuestas	Técnicas compuestas	Soft computing (Algoritmos de optimización)
	Modelo Algorítmico	Estimación de esfuerzo algorítmica	Modelos lineales y no lineales (Modelo de Bailey & Basili, Modelo de Farr & Zagorski, Modelo de Nelson)
		Modelo algorítmico	Modelos discretos (Modelo de Boeing, Modelo de Aron, Modelo de Wolverton)
		Técnicas basadas en dinámicas	Modelos multiplicativos (Walston & Felix, Modelo Doty)
		Técnicas de regresión	
		Técnicas de modelo/teoría	Power Function Model (SLIM, COCOMO, Modelo Jensen)
		Técnicas basadas en modelo	

Tabla 1: Las técnicas pueden ser nombradas de diferente manera pero todas atienden el mismo problema o tienen el mismo enfoque [58].

anteriores), algunos de estos modelos son SLIM, COCOMO y el Punto de Función de Albrecht [13]. Las limitaciones de los modelos algorítmicos han llevado a la exploración de modelos no algorítmicos basados en técnicas de *soft computing*, aprendizaje automático, y sistemas expertos. En el año de 2012, Wen J y otros investigadores [15] sugirieron el aprendizaje automático como otra categoría para la técnica de estimación del esfuerzo.

Esta última categoría ha ido creciendo y varios trabajos proponen modelos de aprendizaje automático como una alternativa a los modelos tradicionales [33], [38], [42], [39], [41], [40], [37] aprovechando los conjuntos de datos públicos y aplicando técnicas como árboles de regresión lineal, *Random Forest*, *Bagging Regressor*, *Adaboost Regressor*, *Gradient Boost Regressor*, *k-Nearest Neighbors*, *Support Vector Machines*, sistemas expertos inteligentes, lógica difusa, programación genética, redes neuronales artificiales y *soft computing*.

Recientemente los trabajos están enfocados a utilizar bases de datos públicas sobre las cuales se puedan construir propuestas de aprendizaje maquina. Por ejemplo, O. Hidmi y B. E. Sakar [33] proponen ensambles de algoritmos basados en técnicas de aprendizaje automático como alternativa a los métodos de estimación subjetivos que consumen tiempo como el juicio de expertos y estimación por analogía. [33] utiliza dos *dataset* (*Desharnais* y *Maxwell*) para construir un modelo de estimación de esfuerzo. El conjunto de datos de *Desharnais* incluye 81 proyectos desarrollados a finales de los 80's, y el conjunto de *Maxwell* consiste de 62 proyectos desarrollados entre 1985 y 1993.

En el año 2012, se desarrolló un sistema experto para la estimación de costo [38]. La propuesta se compone de una interfaz de usuario, el procesador de lenguaje natural, el motor de inferencia, la base de conocimiento y una base de datos adicional. El trabajo compara el modelo propuesto con COCOMO II y *Function Point Analysis* (FPA). De los tres modelos, el sistema experto propuesto tuvo el menor error relativo.

En 2014 M Waseem Khan & Imran Qureshi [42] presentaron un *survey* acerca del uso de técnicas de aprendizaje automático. Este trabajo destaca distintos métodos para mejorar la

estimación de esfuerzo como lógica difusa, árboles de regresión lineal, programación genética, *soft computing*, minería de datos, redes neuronales de tipo *feedforward*, entre otros.

El algoritmo *K-Nearest Neighbors* (KNN) se ha probado en investigaciones con un enfoque distinto a puntos de historia de usuario. En 2017 O. Hidmi y B. E. Sakar [33] combinaron un ensamble de algoritmos KNN con máquinas de soporte vectorial. Dos años más tarde, en 2019 Ibtissam Abnane et al. [39] realizaron estimación de esfuerzo basado en analogía usando un ensamble de algoritmos KNN que se aplica a tres conjuntos: Albrecht, ISBSG y COCOMO.

Los ensambles de aprendizaje automático más conocidos son bosques aleatorios, *Bagging Regressor*, *AdaBoost Regressor* y *Gradient Boosting Regressor*. En 2020 P. Suresh Kumar et al. [41] emplearon *function points* y los ensambles previamente mencionados más otras tres técnicas (KNN, DT, SGD), dando un total de siete algoritmos de aprendizaje automático. Los datos de entrada corresponden a 63 proyectos de COCOMO'81 y 499 proyectos del *dataset* China. De las siete técnicas la mejor es *Gradient Boosting Regressor* obteniendo una precisión de 81 % con COCOMO'81 y 93 % con el *dataset* China.

Por otra parte, en el campo de las redes neuronales, P. Suresh Kumar et al [40] presentaron en 2020 una revisión de la literatura de dos décadas con un total de 80 trabajos revisados. En este análisis se encontró que los *datasets* más utilizados son COCOMO con 41.25 % (33 trabajos), seguido de NASA con 23.75 % (19 trabajos). En esta referida revisión de la literatura, los autores encontraron que las redes neuronales artificiales aplicadas a estimación de esfuerzos en proyectos de software son las de tipo *Radial Basis*, *Wavelet*, *Spiking*, *Functional Link*, las redes de tipo *feedforward* como son MLP, Elman, LSTM, además de las redes de aprendizaje profundo.

Un ejemplo de implementación de red neuronal recurrente de tipo *feedforward* es el trabajo de Lujain A. Hussein et al [37] donde realizan estimaciones utilizando una red con tres capas: capa de entrada, capa oculta y capa de salida. Este modelo resultante recibe como datos de entrada 12 parámetros como por ejemplo el tiempo de finalización, tamaño, complejidad, tipo de aplicación, obsolescencia del hardware, equipo de desarrollo, lenguaje de programación, entre otros. La idea de aplicar recurrencia consiste en tomar parámetros de proyectos similares para que la red aprenda con estos parámetros. En la red recurrente propuesta la salida en las etapas N-1, N-2, ...1 se ingresan como entradas de la etapa N, esto es, las etapas previas se utilizan como entrada durante la siguiente iteración.

4.3. Desarrollo Ágil

El desarrollo de software ágil surgió en la primera década del siglo XXI [135]. En 2002 Grenning propuso el uso de *Planning Poker* como técnica de estimación y más tarde en 2005 M. Cohn sugirió que este método era útil en estimación de proyectos ágiles [15]. Durante los últimos diez años también se han trabajado propuestas que toman como eje principal los métodos ágiles. Marta Fernández-Diego en conjunto con otros investigadores [44] realizaron una revisión sistemática de la literatura de los años 2014 a 2020 encontrando 73 nuevas investigaciones sobre métodos ágiles como *Scrum*, *Extreme Programming*, *Lean Software Development*, *Adaptive Software Development* y *Feature Driven Development*. Los trabajos se clasifican en aquellas investigaciones basadas en el juicio de experto (*Planning Poker*, *Wideband Delphi*) y las investigaciones basadas en datos (técnicas de aprendizaje automático y redes neuronales).

Dentro de los modelos de estimación de esfuerzo se encuentran el trabajo de Zia [7] orientado a producir estimaciones de tiempo y costo de un proyecto utilizando un modelo de regresión

lineal. Coelho & Basu [8] reafirman que los puntos de historia son estimaciones del tamaño del trabajo a realizar, y que la velocidad es una medición del progreso por iteración del proyecto. En 2014, Rashmi Popli & Naresh Chauhan [35] calcularon el costo, esfuerzo y duración de un proyecto pequeño y mediano usando un enfoque de historias de usuario.

Sakshi Garg et al. [45] plantearon en 2015 un modelo basado en *Principal Component Analysis* que identifica los factores que afectan los costos del desarrollo y además aplicaron programación con restricciones. En el mismo año Atef Tayh Raslan et al. [3] propusieron mejorar la precisión de la estimación aplicando un modelo de estimación basado en puntos de historia y lógica difusa denominado *Fuzzy-SPEM (Story Point Estimation Model)*. E. Scott, & D. Pfahl [4] proponen la estimación de puntos de historia a través de las características de desarrollador. El modelo asigna puntos de historia a los reportes de incidencia tomando en cuenta el perfil del desarrollador que toma la incidencia, los autores aplican máquinas de soporte vectorial para realizar las estimaciones.

El esfuerzo en la fase de mantenimiento es estudiado por Jitender Choudhari y Ugrasen Suman [16]. Los autores proponen un modelo llamado *Software Maintenance Effort Estimation Model (SMEEM)* utilizando puntos de historia para calcular el volumen de mantenimiento y los factores de ajuste que afectan los puntos de la historia al estimar esfuerzo. En este caso, el modelo SMEEM está acotado a un entorno de mantenimiento basado en programación ágil y *Extreme Programming (XP)*.

O. Malgonde & K. Chari [46] propusieron en 2019 un modelo para la predicción de esfuerzo de la historia de usuario mediante tres componentes: un modelo predictivo de esfuerzo de la historia de usuario, un modelo basado en ensambles aplicando siete algoritmos (SVM, KNN, redes neuronales, árboles de decisión, *Ridge Regression*, OLS, redes bayesianas), y un modelo de optimización para mejorar la estimación del esfuerzo.

Morakot Choetkiertikul et al.[6] proponen en el mismo año un modelo de predicción para estimar puntos de historia basado en redes neuronales de tipo LSTM y *Recurrent Highway*. El enfoque está orientado a estimar esfuerzo a nivel de historia de usuario y no a nivel de proyecto (esfuerzo total).

Derivado de [6], surge el trabajo de K. A. Alsubhi [118] cuya propuesta consiste en la aplicación de una red LSTM recurrente y una capa de regresión para realizar estimaciones de puntos de historia. Como conjunto de datos se toman reportes tomados de los repositorios de JIRA [136]. El modelo propuesto predice el tamaño de los nuevos reportes en función de las estimaciones previas del punto de la historia del equipo mediante el empleo de arquitecturas de aprendizaje profundo. La propuesta denominada *Deep-ASDEE* lee el título y la descripción del reporte como texto y genera una estimación de punto de historia para ese reporte. Los dos bloques principales son la representación del documento mediante LSTM y una capa de salida de regresión simple, es decir, una red de tipo *feedforward* con activación lineal.

El modelo *Deep-ASDEE* se compara con las técnicas BOW y Doc2vec, demostrando a través de las métricas de evaluación que se mejora la estimación, pues de los 16 proyectos evaluados, la arquitectura de aprendizaje profundo mejoró la precisión estándar en 12 proyectos. Una vez construido el modelo *Deep-ASDEE*, los autores lo compararon con una red LSTM simple combinada con bloques de regresión como el modelo lineal de transformación automática, máquinas de soporte vectorial y regresión lineal simple. De las cuatro técnicas, *Deep-ASDEE* ofreció los mejores resultados de estimación.

Otro ejemplo de estimación a nivel de historia de usuario es el trabajo realizado en 2020

por M. Durán y otros investigadores [47] en donde proponen un método para estimar la complejidad de la historia a través de su descomposición utilizando redes bayesianas.

M. Gultekin & O. Kalipsiz [48] aplicaron el algoritmo *Support Vector Regressor* (SVR) con *Radial Basis Function* como *kernel*, *Random Forest Regression* y *Multi-Layer Perceptron*. Los autores utilizan dos notaciones de puntos de historia, aritmética y Fibonacci. Los resultados arrojan que el uso de la secuencia Fibonacci para realizar las estimaciones de esfuerzo ofrece mejor precisión. El conjunto de datos consiste de 56,687 incidencias de cinco *datasets* (JIRA, MongoDB, JBoss, Spring, Apache).

Otras propuestas híbridas han surgido, las cuales combinan un modelo de estimación de esfuerzo y un modelo de aprendizaje automático. Es decir, el modelo de estimación sienta las bases de los datos que utilizará el algoritmo de aprendizaje supervisado, y este algoritmo se encarga de producir los resultados de estimación esperados. Dependiendo del modelo aplicado, el resultado de la estimación puede ser en puntos de historia, tiempo en días, o costo total.

El presente trabajo de investigación tiene particular interés en el modelo propuesto por Zia et al. [7], con enfoque a nivel de proyecto. Este modelo forma parte del conjunto de modelos de estimación de esfuerzo en desarrollo ágil, y propone utilizar características como la pila de producto, la velocidad inicial del equipo, tamaño del *sprint*, número de días laborales en un mes, el salario del equipo y factores de desaceleración que impactan al proyecto. Los resultados del modelo basado en regresión lineal [7] generan un porcentaje de precisión del 57.14 % para tiempo y 61.40 % para costo.

Tres investigaciones retomaron este modelo para realizar estimaciones utilizando aprendizaje automático, Aditi Panda et al [1] aplicando cuatro tipos de redes neuronales, Ch. Prasada Rao et al. [2] con tres tipos de redes neuronales, y S. M. Satapathy, S. K. Rath [5] con árboles de decisión, bosques aleatorios y *Stochastic Gradient Boosting*. En la Tabla 2 se agrupan estas tres investigaciones que toman como datos de entrada los 21 proyectos del trabajo de Zia et al. [7].

No	Autor	Título	Técnica de aprendizaje automático reportada	Precisión(%) Tiempo	Precisión(%) Costo	Entradas	Salidas	Criterios de evaluación	Dataset
1	Aditi Panda et al.	Empirical Validation of Neural Network Models for Agile Software Effort Estimation based on Story Points	General Regression Neural Network	85.92	No	Total de puntos de historia y velocidad inicial del proyecto	Predicción de tiempo	Error cuadrático medio	21 proyectos desarrollados por seis compañías
			Probabilistic Neural Network	87.66				Coeficiente de determinación (R²)	
			GMDH Polynomial Neural Network	89.67				Magnitud media de error relativo	
			Cascade-Correlation Neural Network	94.76				Porcentaje de precisión	
2	Ch. Prasada Rao et al.	An Agile Effort Estimation Based on Story Points Using Machine Learning Techniques	Adaptive Neuro-Fuzzy Interface System	76.19	57.14	Total de puntos de historia y velocidad inicial del proyecto	Predicción de tiempo y costo	Magnitud media de error relativo	
			Generalized Regression Neural Networks	76.19	76.19			Porcentaje de precisión	
			Radial Basis Function Networks	76.19	76.19			Porcentaje de precisión	
3	S. M. Satapathy, S. K. Rath	Empirical assessment of machine learning models for agile software development effort estimation using story points	Decision Tree	90.48	No	Total de puntos de historia y velocidad inicial del proyecto	Predicción de tiempo	Error absoluto medio	
			Random Forest	95.24				Magnitud media de error relativo	
			Stochastic Gradient Boosting (SGB)	95.24				Porcentaje de precisión	

Tabla 2: Micro estado del arte, selección de tres trabajos de investigación que comparten el mismo modelo de estimación y conjunto de datos.

En los tres casos, los autores evalúan los modelos de aprendizaje maquina con el error relativo y generan un porcentaje de precisión de la predicción. La comparativa de los tres estudios de la Tabla 2 muestra que los árboles de decisión ofrecen los mejores resultados. El trabajo de Satapathy [5] reconoce que la investigación para proporcionar un procedimiento sistemático con el fin de estimar el esfuerzo de los proyectos desarrollados con metodología

ágil es escaso.

En los artículos revisados podemos ver que en general las investigaciones están orientadas a estimación de esfuerzo a nivel de puntos de historia de usuario y a nivel de proyecto (esfuerzo total de todas las historias que lo componen). El aprendizaje profundo se aplica en su mayoría a bases de datos que contienen las historias de usuario en lenguaje natural para poder predecir a partir del texto los puntos de historia de usuario. El aprendizaje automático supervisado se ha aplicado utilizando una sola técnica para emitir estimaciones, y también se ha aplicado la combinación de distintas técnicas en un ensamble para calcular el esfuerzo de la historia de usuario, combinando los resultados de técnicas como SVM, KNN, ANN, DT, entre otras [45]. La complejidad de una historia de usuario se ha descompuesto a través de redes bayesianas [47], y mediante lógica difusa se pueden robustecer modelos ya existentes aplicando inferencia difusa para producir valores de esfuerzo y velocidad de un proyecto de software [3].

Una de las revisiones de la literatura [44] hecha en 2020 destaca que existen pocos proyectos enfocados al aprendizaje automático y en particular las investigaciones enfocadas a estimación de tiempo y costo son escasas debido a la falta de conjuntos de datos que permitan el análisis y experimentación con distintos modelos de aprendizaje supervisado.

Parte de las investigaciones que realizan estimaciones del esfuerzo total a nivel de proyecto destaca principalmente el uso de redes neuronales y árboles de decisiones, aunque es necesario continuar el estudio de otras técnicas con la finalidad de contribuir en la frontera de conocimiento, en particular el uso del algoritmo KNN, arquitecturas de aprendizaje profundo como los *Autoencoders*, redes de tipo MLP y técnicas de ensamble como *AdaBoost* y *Bagging*, todas ellas con el propósito de mejorar la precisión de las estimaciones a nivel de proyecto.

Como se observa en la Figura 16, diversos autores identifican algunos trabajos en donde se aplica una o más técnicas de inteligencia artificial en estimación de esfuerzo. El desarrollo tradicional marcado en azul que aplica marcos de trabajo y métricas no ágiles, de 1997 a 2019 cuenta con mas de 113 investigaciones, mientras que el desarrollo ágil marcado en verde, que abarca de 2004 a 2020 [140, 141, 142, 143, 144, 40, 66, 44, 67]. Un nicho muy reducido de investigaciones realiza estimaciones a nivel de historia de usuario o a nivel de proyecto, identificando tan solo 14 publicaciones de 2012 a 2020.

El estado del arte compacto con los trabajos más significativos se presentan como mapa conceptual en la Figura 17.

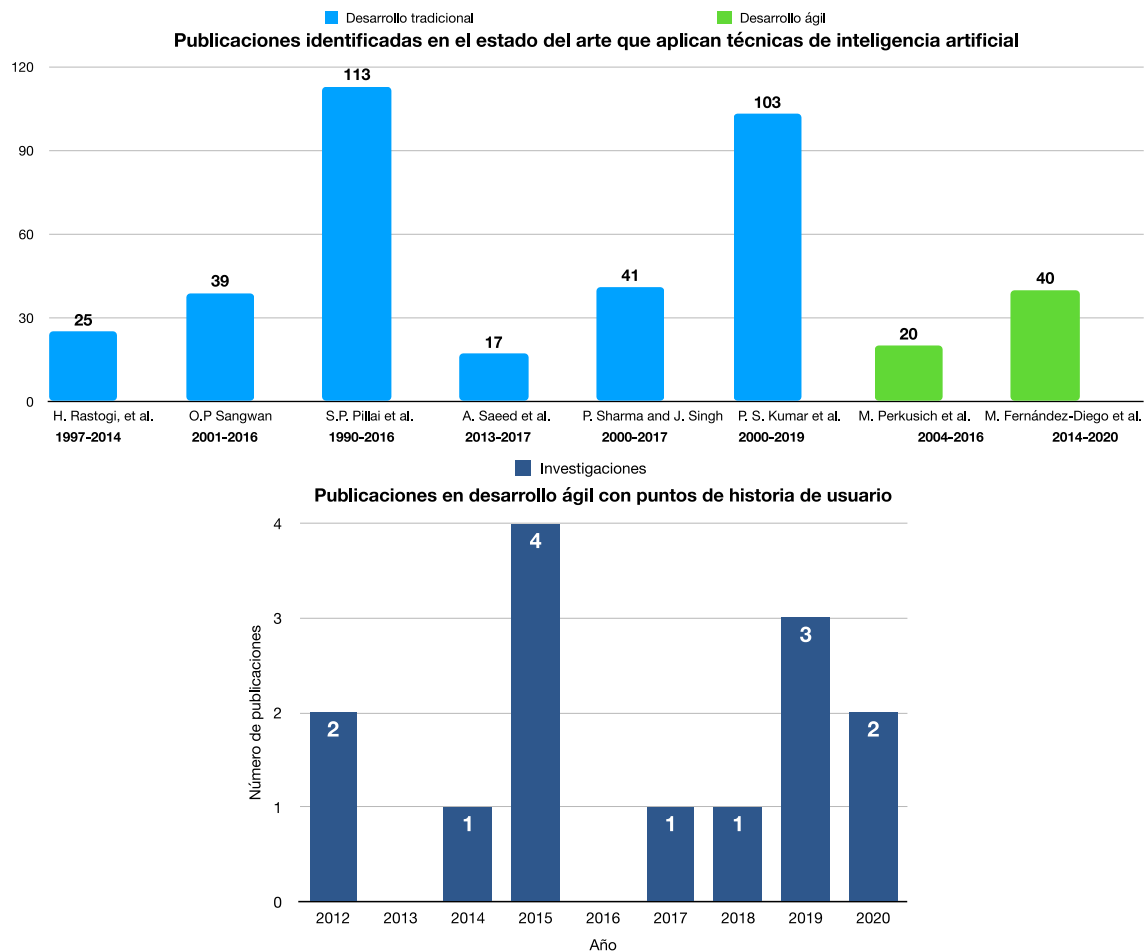


Figura 16: Publicaciones en desarrollo tradicional y desarrollo ágil, las lagunas de conocimiento se encuentran principalmente en un enfoque ágil que combine aprendizaje automático.

5. Propuesta

Las distintas líneas de investigación orientadas a estimación de esfuerzo son vastas y aplican una diversidad de técnicas, desde modelos de estimación algorítmicos y no algorítmicos, hasta métodos de juicio de expertos y técnicas basadas en aprendizaje automático. Nuestro trabajo tiene la finalidad de estudiar diversas estrategias de aprendizaje automático, en particular algoritmos de aprendizaje supervisado aplicados a la estimación de tiempo de finalización y costo total de un proyecto de software utilizando un enfoque basado en puntos de historia de usuario. Hay un aumento constante en el número de organizaciones que adoptan prácticas y procesos ágiles durante el desarrollo de sistemas de software, en parte por las nuevas condiciones de trabajo impuestas por la pandemia [32]. Las principales áreas de las organizaciones que adoptan principios y prácticas ágiles son el desarrollo de software y tecnologías de la información (TI), por lo que es importante realizar estudios que permitan la elaboración de herramientas de apoyo en el proceso de estimación de proyectos desarrollados con métodos ágiles.

La propuesta de este proyecto de investigación consiste en un modelo (Figura 18), construido a partir de un modelo de estimación algorítmico propuesto por Zia et al. en el estado del arte [7], al cual se le agregan categorías de clasificación de tamaño de esfuerzo, tiempo y costo para organizar los proyectos. Además se agrega un modelo de aprendizaje automático, y ambos

ESTADO DEL ARTE

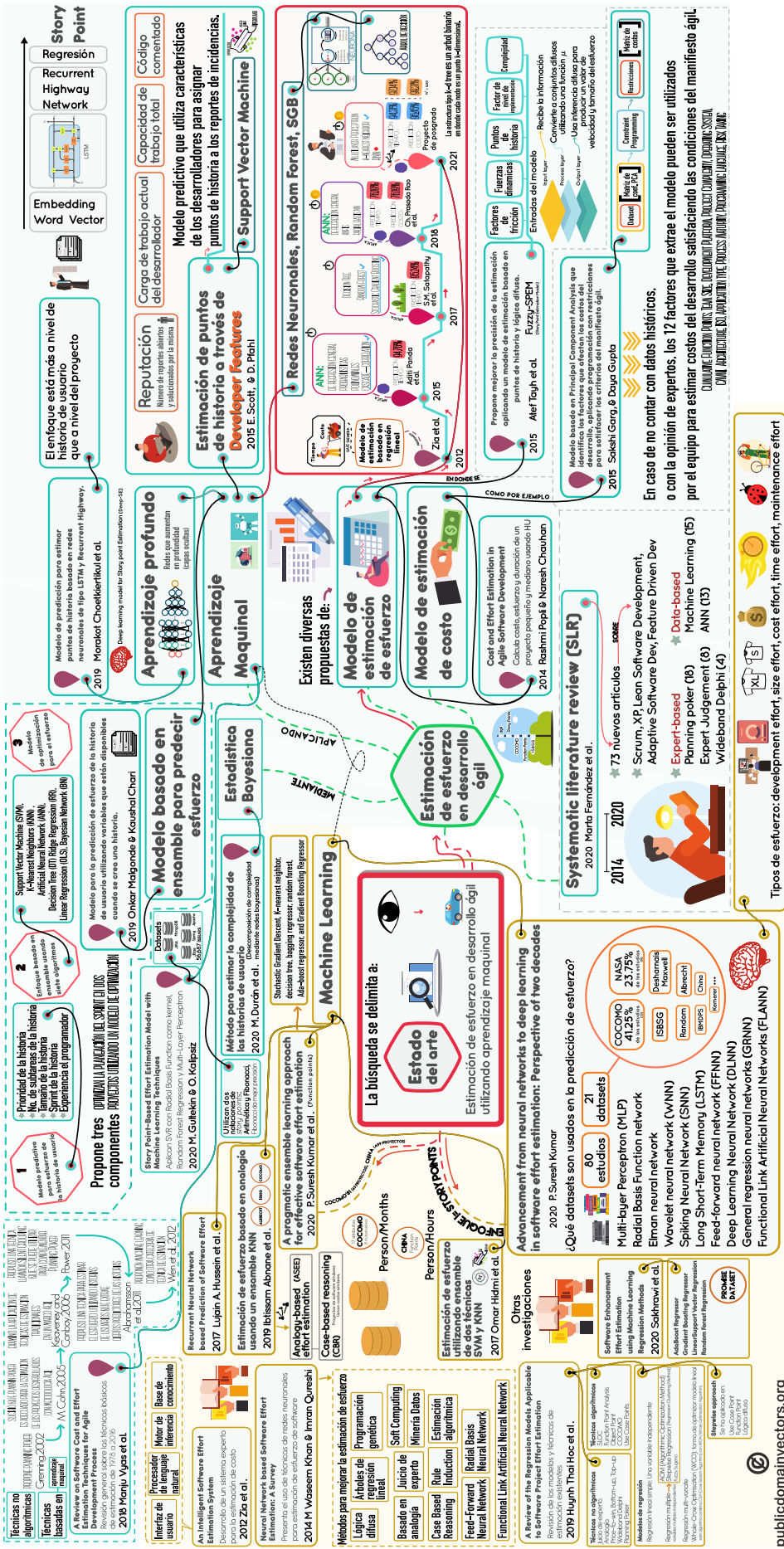


Figura 17: Mapa conceptual compacto del estado del arte.

modelos se combinan para realizar estimaciones a nivel de proyecto. Al revisar el trabajo relacionado se ha identificado que hay una área de oportunidad en la aplicación de diversos algoritmos orientados a la estimación de tiempo y costo tales como las redes neuronales de tipo *Multilayer Perceptron* (MLP), redes recurrentes y el algoritmo *k-Nearest Neighbors* (KNN).

Como se ha mencionado en el Capítulo 3 de Marco Teórico, la técnica de los vecinos más cercanos se basa en la distancia entre muestras y permite comparar la simplicidad de un algoritmo contra técnicas mucho más complejas como las redes neuronales. Por otro lado, los árboles de decisión han probado su capacidad de realizar estimaciones con precisión [5], y además proporcionan una vía simple para interpretar las estructuras del conjunto de datos que resuelven el problema.

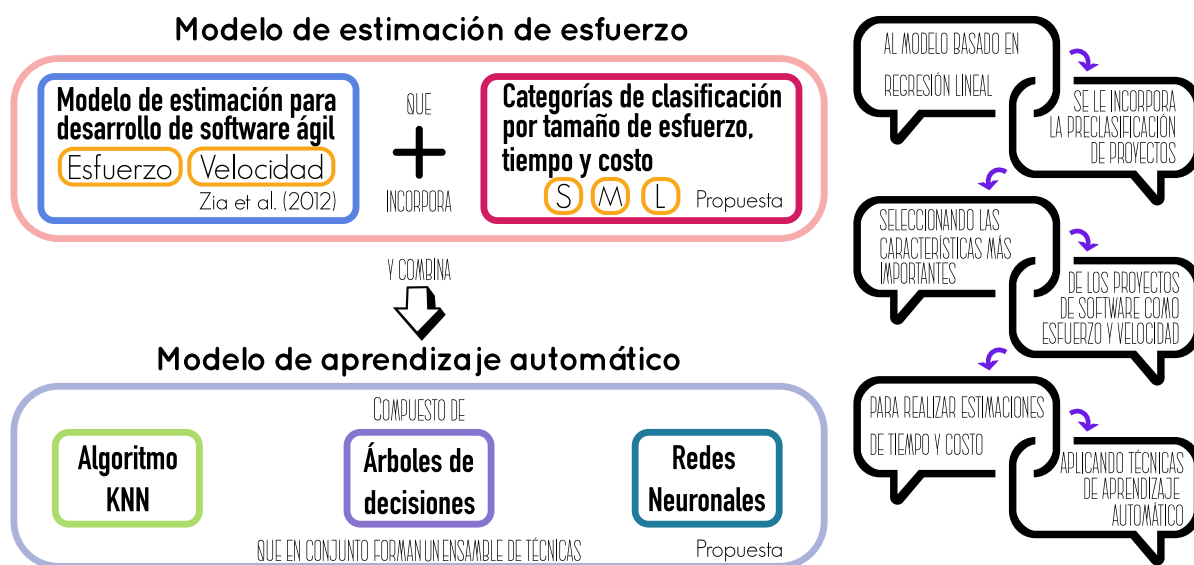


Figura 18: La propuesta del proyecto consiste en generar estimaciones mediante un modelo de estimación de esfuerzo y un modelo de aprendizaje automático.

Este estudio desglosa un análisis de la precisión de la predicción del proceso de estimación ejecutado con técnicas de aprendizaje supervisado. Los algoritmos seleccionados tras una revisión de la literatura incluyen redes neuronales de tipo MLP, redes neuronales recurrentes, *Autoencoders*, ensamble de tipo *AdaBoost* de redes neuronales, árboles de decisión, bosques aleatorios, ensamble de tipo *AdaBoost* para árboles de decisión y el algoritmo KNN. Algunas de estas técnicas aún no han sido probadas en el desarrollo de software ágil, es importante su estudio para contribuir a la frontera de conocimiento de los algoritmos que permiten realizar estimaciones de esfuerzo en proyectos de software.

El enfoque aplicado utiliza puntos de historia de usuario y marcos de trabajo ágil, por lo que el modelo de estimación empleado utiliza el tamaño total del proyecto equivalente a la pila de producto (*product backlog*), la métrica de velocidad del equipo y las categorías de clasificación. La hipótesis del proyecto establece que el uso de categorías de clasificación por tamaño mejora la precisión de la predicción de tiempo y costo de los proyectos, ya que al etiquetar cada proyecto de acuerdo a su tamaño, se crean grupos que permiten discriminar de mejor manera las estimaciones de tiempo y costo. Este particionamiento permitiría en un trabajo a futuro la aplicación de técnicas de aprendizaje no supervisado [70] para la clasificación automática de proyectos de software. Al etiquetar los datos se crean grupos de objetos de modo que en un grupo, las muestras comparten características similares y difieren de las muestras en diferentes grupos [26].

5.1. Modelo de estimación

La primera parte de la propuesta consiste en un modelo basado en regresión lineal [7], cuyo conjunto de muestras forma parte de los pocos conjuntos de datos abiertos disponibles para realizar investigación a nivel de proyecto con métodos ágiles [133], cuyas características más importantes son el esfuerzo total del proyecto y la velocidad del equipo de desarrollo, ambas medidos en puntos de historia de usuario. Este modelo es consistente con la forma en que se desarrolla la clasificación de las historias de usuario utilizando una escala numérica, los defectos y riesgos calculados a partir de valores fijos, y además el modelo es construido sobre una base de 21 proyectos desarrollados por seis compañías, lo que permitió calibrar algunas de las características asociadas a los riesgos.

Un punto a favor de la aplicación de este modelo de estimación y su conjunto de datos es la manera en que se presenta el detalle de las fórmulas matemáticas que componen las características del modelo, los valores obtenidos para cada proyecto y los resultados esperados que consisten en el tiempo de finalización y costo total. Al contar con las entradas y salidas esperadas es posible realizar la reproducibilidad de la implementación de la solución propuesta mediante la presentación del código de los modelos de aprendizaje. Estas cualidades del trabajo de investigación [7] permiten realizar estudios empíricos comparativos al evaluar los resultados del modelo original con propuestas que aplican un enfoque de solución distinto, esto es, comparamos los resultados de un modelo basado en regresión lineal con modelos de aprendizaje automático.

Además, al ser un conjunto de datos pequeño, la naturaleza del conjunto permite no solo estudiar la aplicación de técnicas de aprendizaje, sino evaluar un enfoque de estimación basado en la técnica *data augmentation* [107] abordada en la sección 5.3.2, en donde el principal problema a resolver es la falta de información o muestras, pues uno de los principales problemas documentados es la falta de disponibilidad de conjuntos de datos y herramientas por parte de los investigadores [66].

El modelo de estimación que se toma como base para este proyecto [7] está desarrollado para acoplarse a la mayoría de las características de los métodos ágiles, especialmente la adaptación e iteración, ya que con cambios en los requerimientos se pueden ajustar o crear nuevas historias de usuario, además la velocidad del equipo de desarrollo está pensada para desarrollos iterativos (*sprints*). El modelo es prácticamente implementable en cualquier organización y no está restringido a contextos específicos. Sus principales características se describen a continuación.

Tamaño de la historia de usuario El tamaño de la historia es una estimación de la escala relativa del trabajo en términos de esfuerzo del desarrollo real. La escala propuesta es de uno a cinco, siendo uno el tamaño más pequeño y cinco una historia épica:

$$S \in \mathbb{N} : S[1, 5]$$

Complejidad Complejidad de los requisitos de la historia o complejidad técnica. La complejidad introduce incertidumbre en la estimación, más complejidad significa más incertidumbre. De manera similar al tamaño de la historia se asigna una escala de uno a cinco:

$$C \in \mathbb{N} : C[1, 5]$$

Esfuerzo Usando el tamaño y complejidad de la historia se determina el esfuerzo de una historia particular con la siguiente ecuación:

$$ES = C \times S$$

El esfuerzo para todo el proyecto es la suma individual de los esfuerzos de cada historia que lo compone:

$$E = \sum_{i=1}^n (ES)_i$$

Determinación de la velocidad ágil En desarrollo de software ágil como Scrum consideramos que la distancia es las unidades de esfuerzo y el tiempo es la duración del *sprint*:

$$V_i = \frac{\text{Unidades de esfuerzo completado en un sprint}}{\text{Duración del sprint}}$$

La velocidad se puede definir como la cantidad de esfuerzo que un equipo puede manejar en una unidad de tiempo, es decir cuántas historias de la pila de producto se entregan en un *sprint*.

Optimización de la velocidad La optimización de la velocidad considera las fuerzas de fricción y las fuerzas dinámicas, factores que reducen la velocidad del proyecto y causan una irregularidad desacelerando al equipo.

Fricción La fricción (FR) se calcula como el producto de los cuatro factores de fricción (FF) propuestos [7]. Estos factores son la composición del equipo, cambios en los procesos, factores del entorno y la dinámica en la que trabaja el equipo. Los valores van de factor estable hasta altamente volátil:

$$FR = \prod_{i=1}^4 (FF)_i$$

Fuerza Dinámica El cálculo de la fuerza dinámica (DF) es el producto de los nueve factores variables (VF) indicados en [7]. Los factores variables incluyen cambios en el equipo de trabajo, introducción de nuevas herramientas o defectos en las mismas, responsabilidades del personal no relacionadas con el proyecto, problemas personales del equipo de trabajo, problemas con las partes interesadas (*stakeholders*) y requerimientos mal definidos o cambiantes. Los valores para calcular la DF se pueden consultar en el trabajo original [7]:

$$DF = \prod_{i=1}^9 (VF)_i$$

Desaceleración Con la nueva información obtenida de **FR** y **DF** se calcula la desaceleración (D) en términos de la fricción y fuerza dinámica como:

$$D = FR \times DF$$

Velocidad final Para ajustar la velocidad a un rango más predecible, se calcula la velocidad final como:

$$V = (V_i)^D$$

Tiempo de finalización Para estimar la duración necesaria para completar el proyecto se aplica la ecuación:

$$T = \frac{E}{V}$$

T se mide en días, para convertir a meses se divide por el número de días laborales en el mes (WD) teniendo la siguiente expresión:

$$T = \frac{E}{V} \times \frac{1}{(WD)} = \frac{\sum_{i=1}^n (ES)_i}{(WD) (V_i)^D}$$

Costo de desarrollo El costo se calcula de acuerdo a los distintos gastos relacionados con el proyecto. Zia et al. [7] proporcionan una relación salarial del equipo con otros gastos como el salario, equipos tecnológicos, licencias, *marketing*, rentas, mobiliario, etc. Todos los gastos se han calculado para un proyecto por mes de 14 compañías de Pakistán con CMMI nivel 3. Al tomar la relación neta, el costo de desarrollo es calculado como:

$$\text{Costo} = 1.681 \times TS \times T$$

donde TS es el salario mensual del equipo y T es el tiempo calculado en meses. El valor neto de la relación de gastos asociados corresponde al coeficiente 1.681.

Aplicando cada una de las ecuaciones, los autores [7] trabajan con distintas compañías para generar las estimaciones de tiempo de finalización y costo total de proyectos de software, esto les permitió calibrar y ajustar el modelo acorde a los resultados y la información empírica recolectada. El conjunto de muestras resultante permite la aplicación de técnicas de aprendizaje automático para comparar los modelos de estimación algorítmicos contra los modelos de aprendizaje. Así, partimos de un conjunto de 21 proyectos desarrollados por seis compañías de software de Pakistán como se muestra en la Tabla 3. Los autores en [7] obtuvieron información

No	Effort	V_i	D	V	SprintSize	WorkDays	TeamSalary	ActualTime	ActualCost
1	156	4.2	0.687	2.7	10	22	230,000	63	1200000
2	202	3.7	0.701	2.5	10	21	260,000	92	1600000
3	173	4	0.878	3.2	10	22	250,000	56	1000000
4	331	4.5	0.886	3.8	10	22	300,000	86	2100000
5	124	4.9	0.903	4.2	10	22	300,000	32	750000
6	339	4.1	0.903	3.6	10	22	400,000	91	3200000
7	97	4.2	0.859	3.4	10	22	250,000	35	600000
8	257	3.8	0.833	3	10	22	250,000	93	1800000
9	84	3.9	0.646	2.4	10	22	190,000	36	500000
10	211	4.6	0.758	3.2	10	22	250,000	62	1200000
11	131	4.6	0.758	3.2	10	22	250,000	45	800000
12	112	3.9	0.773	2.9	10	22	200,000	37	650000
13	101	3.9	0.773	2.9	10	22	200,000	32	600000
14	74	3.9	0.773	2.9	10	22	200,000	30	400000
15	62	3.9	0.773	2.9	10	22	200,000	21	350000
16	289	4	0.742	2.8	10	22	250,000	112	2000000
17	113	4	0.742	2.8	10	22	250,000	39	800000
18	141	4	0.742	2.8	10	22	250,000	52	1000000
19	213	4	0.742	2.8	10	22	250,000	80	1500000
20	137	3.7	0.758	2.7	10	22	220,000	56	800000
21	91	3.7	0.758	2.7	10	22	220,000	35	550000

Tabla 3: El modelo original basado en regresión lineal [7] obtiene estimaciones de tiempo y costo utilizando la información de las columnas que componen el conjunto de muestras.

relacionada con los proyectos agrupándola en 21 registros que corresponden a cada uno de los proyectos,. La información de las columnas de la Tabla 3 representa características presentadas previamente en el modelo como el esfuerzo necesario para completar el proyecto (*effort*), la velocidad del equipo de desarrollo (V_i), la relación de las fuerzas y los factores que pueden ralentizar el proyecto (*D*), el tamaño del *sprint* en días (*SprintSize*), los días laborables en un mes (*WorkDays*) y el salario del equipo en rupias pakistaníes (*TeamSalary*).

Considerando las características más importantes del modelo [7], el esfuerzo de una sola historia se obtiene aplicando el producto del tamaño de la historia y su complejidad. El esfuerzo del proyecto completo es la suma de los esfuerzos de todas las historias individuales. La velocidad es la relación entre la distancia y el tiempo, es decir, cuántas unidades de esfuerzo completa un equipo en un *sprint* típico.

Los conceptos de esfuerzo y velocidad referidos en la Sección 3.4 son importantes durante la implementación del proyecto, ya que cada punto de la historia representa una distribución normal del tiempo necesario que requerirá el equipo para implementar completamente un elemento de la pila de producto, a partir de esto, está claro que estamos hablando de un rango

de probabilidad estadística. Por ejemplo, un punto de historia podría representar entre 4 y 12 horas, 2 puntos de historia entre 10 y 20 horas y así sucesivamente. Además, para conocer el valor de la velocidad, el proyecto debe estar corriendo para que el líder sepa cuantos puntos hace el equipo por *sprint*, sin embargo como estamos trabajando con datos históricos, podemos conocer la velocidad si un equipo no cambia y ha hecho proyectos similares anteriormente.

Habiendo dicho eso, ahora podemos considerar el esfuerzo y la velocidad como datos de entrada de los algoritmos porque nuestras estimaciones se basan en información de proyectos finalizados, y estas dos características combinadas con etiquetas de tamaño se proporcionan a las técnicas de aprendizaje automático para generar predicciones. Así, una vez obtenido el conjunto de muestras, se procede a su clasificación con el propósito de someter a prueba la mejora de precisión de las estimaciones como establece la hipótesis.

5.2. Clasificación de proyectos por tamaño

Una vez identificadas las características principales de un proyecto de software notamos que estos proyectos pueden ser agrupados de acuerdo al tamaño de esfuerzo total, la cantidad de tiempo para completarlo y su costo asociado. La clasificación permite agrupar los proyectos en tres conjuntos lo cuál facilita su discriminación de acuerdo al tamaño.

En un problema de regresión, el valor obtenido a la salida es un valor numérico. Los valores numéricos de este proyecto son el tiempo de finalización medido en días y el costo total del desarrollo dado en rupias pakistaníes debido a que el modelo de esfuerzo se desarrolló en Pakistán. La propuesta consiste en generar valores de tiempo y costo a partir de las características del proyecto y las categorías de clasificación que agrupan a los proyectos. A este conjunto de datos se le considera como un conjunto de muestras de entrenamiento preclasificadas [14].

Clasificar los proyectos es importante porque permite proyectar diferentes escenarios para un mismo proyecto, es decir, dada una cantidad fija de esfuerzo, el modelo propuesto con etiquetas de clasificación puede predecir la cantidad de días estimados para un proyecto de tamaño de tiempo pequeño, mediano o grande. Lo mismo se aplica al costo total, se pueden generar tres casos para un mismo proyecto ofreciendo un mejor panorama de cuánto puede crecer el proyecto de software en términos de costo y tiempo.

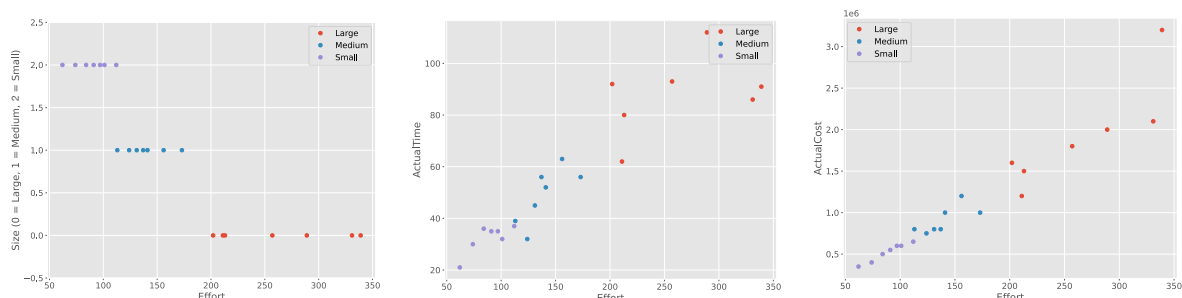
Al agrupar los proyectos por esfuerzo se generan tres rectas, una para cada tamaño, los proyectos grandes son puntos que caen en la recta $y = 0$, los proyectos medianos en $y = 1$ y los proyectos pequeños en $y = 2$ (Figura 19a).

La convención utilizada es simple, los proyectos pequeños se representan con la letra S (*small*), los medianos con la letra M (*medium*) y los grandes con la letra L (*large*). Un proyecto nuevo puede ser clasificado de manera empírica en tamaño de esfuerzo, tiempo y costo comparándolo con proyectos de esfuerzo similar y asignando la etiqueta correspondiente. Una vez que todo el conjunto de datos es clasificado se procede a realizar su codificación a enteros para que los algoritmos de aprendizaje supervisado puedan realizar las estimaciones, ya que debemos recordar que algunos algoritmos solo pueden recibir como datos de entrada valores numéricos. Por lo tanto la convención utilizada relaciona las categorías de la siguiente manera:

$$0 = \text{Grande} = L, \quad 1 = \text{Mediano} = M, \quad 2 = \text{Pequeño} = S$$

En la Figura 19b se muestran los proyectos agrupados por tiempo, en 19c aparecen los

proyectos agrupados por costo. En las tres gráficas se mantienen las categorías (S,M,L), la codificación a enteros (0,1,2), así como los colores de los tres grupos.



(a) Proyectos catalogados por esfuerzo (b) Tiempo en función del esfuerzo (c) Costo en función del esfuerzo

Figura 19: Los proyectos se clasifican en pequeños (morado), medianos (azul), y grandes (rojos).

La discretización de los proyectos se puede realizar aplicando distintas técnicas como se presentó en el Capítulo 3 de Marco Teórico. Una vez clasificados los 21 proyectos de manera empírica se aplica la discretización uniforme, por cuantiles y kmedias para comparar los distintos grupos resultantes. En la Figura 20 se observa que el procedimiento empírico dió el mismo resultado que los cuantiles, la única diferencia reside en el grupo de costo, en donde los proyectos de costo 120,000 PKR se consideran medianos en la discretización empírica y grandes por cuantiles. La discretización uniforme y con kmedias tuvieron resultados similares pero generan conjuntos desbalanceados

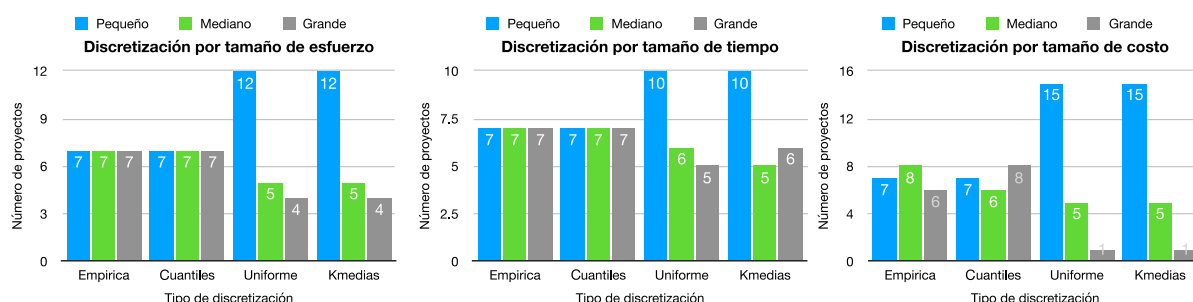


Figura 20: La discretización mediante diversas técnicas genera diferentes conjuntos de proyectos según el tamaño de esfuerzo, tiempo y costo.

La extracción y selección de características es un paso importante en el preprocesamiento de datos [96]. Las dos primeras columnas del conjunto de muestras (Tabla 3) son las características extraídas de los proyectos, adicionalmente tenemos las dos últimas columnas que corresponden al tiempo de finalización y el costo total real de cada proyecto. Incorporando las categorías de clasificación, se genera un nuevo conjunto de datos como se muestra en la Tabla 4. Del total de características originales, solo se toman como entradas el esfuerzo y la velocidad del equipo, ya que en el estado del arte, como en el análisis de componentes principales, se muestra que estas dos características son las más importantes para entrenar un modelo de aprendizaje automático. Más adelante, en la sección de metodología se explica la selección de características que permite simplificar el conjunto de datos de entrada.

No	Effort	Vi	ActualTime	ActualCost	SizeEffort	SizeTime	SizeCost
1	156	4.2	63	1200000	M	L	M
2	202	3.7	92	1600000	L	L	L
3	173	4	56	1000000	M	M	M
4	331	4.5	86	2100000	L	L	L
5	124	4.9	32	750000	M	S	M
6	339	4.1	91	3200000	L	L	L
7	97	4.2	35	600000	S	S	S
8	257	3.8	93	1800000	L	L	L
9	84	3.9	36	500000	S	S	S
10	211	4.6	62	1200000	L	M	M
11	131	4.6	45	800000	M	M	M
12	112	3.9	37	650000	S	M	S
13	101	3.9	32	600000	S	S	S
14	74	3.9	30	400000	S	S	S
15	62	3.9	21	350000	S	S	S
16	289	4	112	2000000	L	L	L
17	113	4	39	800000	M	M	M
18	141	4	52	1000000	M	M	M
19	213	4	80	1500000	L	L	L
20	137	3.7	56	800000	M	M	M
21	91	3.7	35	550000	S	S	S

Tabla 4: Reduciendo el número de características el conjunto de datos tiene una dimensión de entrada de $n=5$.

5.3. Modelo de aprendizaje automático

Durante el periodo 2004-2015 se realizaron diversos trabajos de investigación de los cuáles muy pocos aplican técnicas de aprendizaje automático en el área de administración de ingeniería de software [66]. Mientras que en otro estudio realizado de 2014 a enero de 2020, se encontró que la mayoría de las investigaciones se enfocan a planificación a nivel de *sprint*, y un número reducido lo hace a nivel de proyecto aplicando técnicas de aprendizaje automático y redes neuronales [44], esto se observa en la Figura 16 del Capítulo de trabajo relacionado.

La propuesta de este trabajo de investigación se enfoca en la estimación de costos y duración total de un proyecto de software mediante la aplicación de un modelo de aprendizaje automático que contenga tres elementos: un método de razonamiento basado en casos como el

algoritmo *KNN*, que por sus reglas simples permite realizar estimaciones a través de comparaciones con otros elementos del conjunto de datos disponibles, un algoritmo como los árboles de decisión que permiten interpretar y comprender el conocimiento codificado a través de su estructura, y un algoritmo capaz de adaptarse a problemas complejos de naturaleza no lineal como son las redes neuronales.

En el aprendizaje supervisado [20], se espera que al modelo se le proporcionen las entradas y las salidas deseadas. Después de completar el entrenamiento, el algoritmo aplica el mismo procedimiento con los datos de prueba para emitir una estimación y evaluar su desempeño. En este proyecto los algoritmos se construyen considerando dos casos como en la Figura 21, en un primer escenario los modelos de aprendizaje automático reciben como entrada el esfuerzo medido como el total de puntos de historia de usuario requeridos (*Effort*), y la velocidad inicial del equipo (V_i).

Una segunda estrategia implica recibir como datos de entrada el esfuerzo, la velocidad y las categorías de clasificación por tamaño para comprobar que ocurre una modificación en los resultados al integrar las categorías. La finalidad de ejecutar dos estrategias consiste en probar que la clasificación mejora la precisión de las estimaciones hechas por los algoritmos.

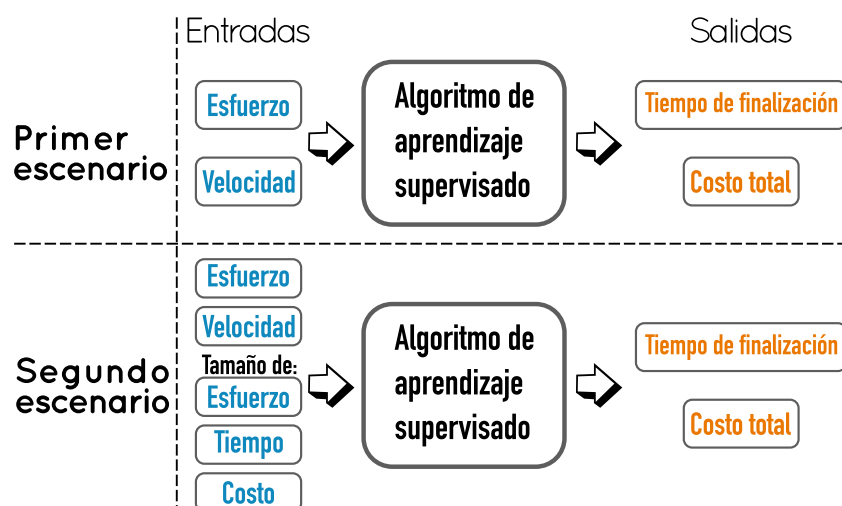


Figura 21: Dos casos a ejecutar para probar la hipótesis de mejora en las estimaciones.

5.3.1. Estrategia de diseño de los experimentos

A continuación se presenta una estrategia universal [117] para atacar y resolver cualquier problema de aprendizaje automático supervisado mediante los siguientes pasos:

1. **Definir el problema:** respondiendo las preguntas ¿Cuáles son los datos de entrada?, ¿Qué se está tratando de predecir? y ¿A qué tipo de problema se enfrenta, es clasificación o regresión?.
2. **Elegir métricas de evaluación:** la métrica guiará la elección de una función de pérdida la cuál será optimizada por el modelo.
3. **Decidir sobre un protocolo de evaluación:** establecer cómo se medirá el progreso actual del algoritmo, considerando el tamaño del conjunto de datos con el que se trabaja, ¿Se aplicará *Cross-Validation*, *Leave-One-Out* u otro tipo de protocolo?.

4. **Preparar la información:** formatear los datos de manera adecuada para que el algoritmo pueda ser entrenado.
5. **Desarrollar un modelo que funcione mejor que una línea de base:** se espera que los modelos de aprendizaje automático ofrezcan mejores resultados que el modelo de regresión lineal [7].
6. **Regularizar y ajustar el modelo a través de sus hiperparámetros:** ajustar repetidamente los algoritmos mediante el entrenamiento, validación y prueba para mejorar el desempeño del proceso de estimación.

Identificar el tipo de problema permite seleccionar el tipo y la arquitectura del modelo. En este caso trabajamos con un problema de regresión, y los algoritmos que permiten resolver este tipo de problemas son KNN, DT, y ANN, útiles por sus características presentadas en el marco teórico. El modelo híbrido de aprendizaje se muestra en la Figura 22.

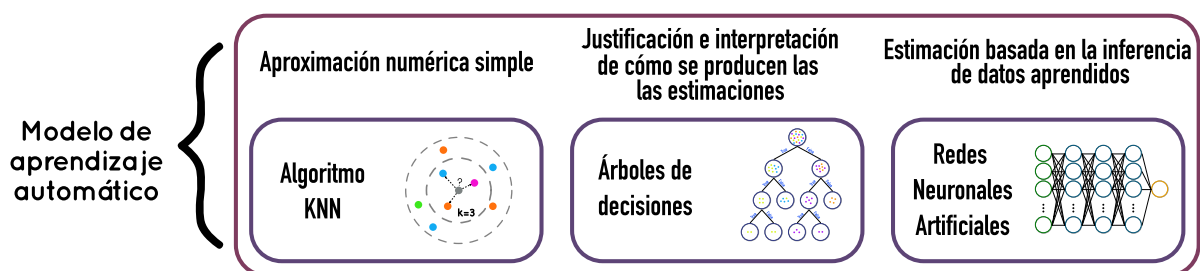


Figura 22: Modelo de aprendizaje supervisado compuesto por tres técnicas distintas.

Al seleccionar los datos y algoritmos a aplicar, la función de pérdida asociada a los modelos es el error cuadrático medio aplicado en el método de mínimos cuadrados ordinarios. Adicionalmente se aplican otras métricas como el error relativo medio, la raíz del error cuadrático medio, la varianza y el coeficiente de determinación. Todas estas métricas permiten evaluar los hiperparámetros de los algoritmos y determinar si un modelo entrenado realiza buenas o malas estimaciones, ya que tenemos dos puntos de comparación, los valores reales de los proyectos y las estimaciones hechas por el modelo de regresión [7].

5.3.2. Integración de datos a los algoritmos

Un problema en la estimación del esfuerzo del desarrollo de software es la falta de información. Debido al proceso largo y costoso de recopilación de datos, las organizaciones suelen tener muestras de entrenamiento limitadas. La creación de proyectos sintéticos basados en proyectos completados es una técnica útil en donde la alteración debe ser lo suficientemente pequeña para modificar los valores reales del proyecto y mantener sus características, pero lo suficientemente grande para ser útil y crear nuevas entradas [105]. La generación de proyectos sintéticos es una forma económica y útil de mejorar el desempeño de la predicción al agregar pequeñas cantidades de ruido gaussiano a los valores de las características de las muestras de entrenamiento [105]. Dado que el conjunto de datos es pequeño, se utiliza la técnica llamada aumento de datos (*Data Augmentation*) [104] para duplicar el número total de los proyectos, dando como resultado 42 registros de proyectos útiles para el entrenamiento.

El aumento de datos es un método de regularización efectivo para entrenar modelos de aprendizaje automático donde los datos disponibles se amplían inyectando ruido para evitar

memorizar el conjunto pequeño y reducir el sobreajuste [22]. La técnica ajusta el número de ejemplos de entrenamiento y su distribución, es útil en enfoques de aprendizaje profundo para la técnica de *word embedding*, para la clasificación de imágenes, el reconocimiento facial y series temporales [106], [107], [108], [109], [110], así como en problemas de regresión con datos pequeños [105].

En el caso de los proyectos de software las características a modificar son esfuerzo y velocidad, ambas de valores numéricos, la adición de ruido consiste en agregar de manera aleatoria hasta un $\pm 5\%$ del valor original, por ejemplo aplicando una regla de tres:

$$E_m = \frac{E \times (100 + R)}{100} \quad (11)$$

donde E_m es el esfuerzo modificado, E es el esfuerzo total del proyecto y R es un valor aleatorio entre $(-5, 5)$ sin incluir a $R = 0$.

Solo es posible aprender a predecir si contamos con información suficiente para entrenar. La extensión de datos permite mejorar el rendimiento de los algoritmos ya sea en problemas de clasificación o regresión [107]. Así, el conjunto de datos se extiende mediante la adición de ruido al conjunto original para generar muestras independientes de entrenamiento, validación y prueba. Tener claras cuáles son las entradas y salidas delimita el conjunto de muestras y selecciona las características de los proyectos que corresponden a los datos de entrada.

Como se muestra en la Figura 23, el conjunto de datos inicial corresponde a los proyectos reportados en [7], cada uno de los proyectos se duplica, modificando sus características de esfuerzo y velocidad para generar un total de 42 proyectos. No hay una regla general sobre cómo elegir el número de observaciones para entrenamiento, validación y prueba ya que esto depende de los datos de entrada [134]. Una técnica común para dividir los datos consiste en reservar dos tercios para el entrenamiento y un tercio para verificar el rendimiento de la generalización del algoritmo [70]. Al aplicar validación cruzada de 10 mangas el conjunto de datos se divide en $k - 1 = 9$ mangas para entrenar on $k = 10$ y una manga para validar, lo que resulta en una partición de 90 % entrenamiento y 10 % para validación.

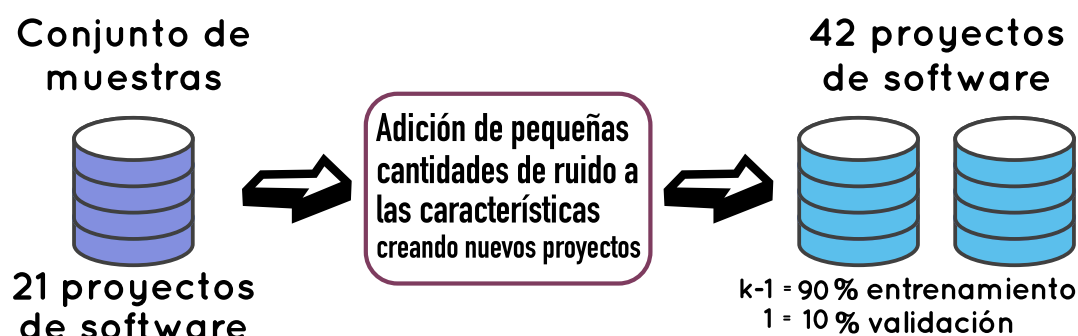


Figura 23: Creación de nuevas muestras mediante la adición de ruido a las características.

5.3.3. Resumen de técnicas aplicadas

El presente trabajo mezcla dos áreas de las ciencias de la computación, la ingeniería de software y la inteligencia artificial. Los algoritmos seleccionados forman parte del aprendizaje

automático o *machine learning*, campo de la inteligencia artificial, y el problema se resuelve a través de técnicas de aprendizaje supervisado.

Los resultados del modelo basado en regresión lineal [7] permiten comparar el desempeño de los algoritmos de aprendizaje supervisado seleccionados. Cada técnica se diseñó de manera empírica probando diferentes configuraciones de los algoritmos y tomando aquellas que ofrecieron los mejores resultados. Los árboles de decisión y las redes neuronales se probaron de manera individual y en ensamble. Las técnicas estudiadas en este proyecto de investigación se representan en un árbol como se muestra en la Figura 24.

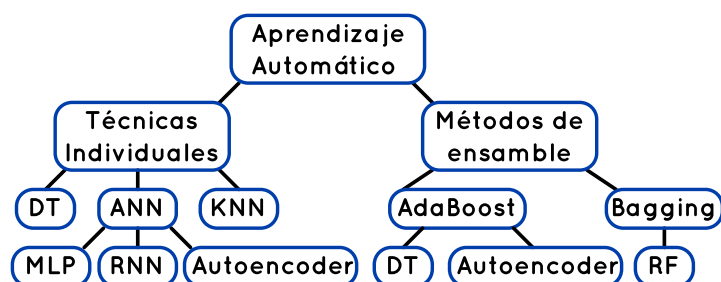


Figura 24: Técnicas aplicadas en el modelo de aprendizaje automático.

6. Metodología

La evaluación empírica se divide en distintas fases; la primera consiste en el análisis y preprocesamiento de datos, es decir, elegir las mejores características y formatear los datos. Posteriormente en la segunda fase se diseñan los modelos de aprendizaje automático, que son depurados hasta simplificarlos y obtener las configuraciones que ofrecen los mejores resultados; el proceso se repite hasta obtener las configuraciones finales para realizar las estimaciones de los 21 proyectos de software. En la Figura 25 se muestra el diagrama de flujo de la estrategia general aplicada a cada uno de los algoritmos de aprendizaje supervisado. Este proceso se repite para cada uno de los algoritmos debido a que cada técnica tiene su propia estrategia de programación.

La estrategia consiste de tres partes; la primera etapa marcada en color verde corresponde al preprocesamiento de la información. Los datos se cargan de un archivo csv a una tabla tipo *dataframe* de pandas. Los datos se etiquetan de acuerdo a su tamaño de esfuerzo, tiempo y costo. Posteriormente estos datos se convierten en arreglos tipo *numpy* para normalizar el contenido de los mismos.



Figura 25: Estrategia general del proyecto.

Dado que la estandarización de datos es imprescindible en muchas técnicas de aprendizaje automático [96], cada entrada del conjunto de datos se normaliza, lo que transforma al conjunto en datos estándar distribuidos normalmente con media cero y varianza unitaria. Cuando la información no se escala al rango entre $[0,1]$ puede ocurrir que si una característica tiene una varianza mucho mayor que el resto, la función objetivo solo puede buscar esta característica e ignorar las que son mucho más pequeñas, lo que hace que el estimador no pueda aprender de todo el conjunto de características.

Una vez que los datos están listos, en la segunda etapa marcada en amarillo de la Figura 25 se configuran los algoritmos de aprendizaje automático seleccionados para realizar las estimaciones y la validación cruzada aplicada durante el entrenamiento. La capacidad de aprendizaje y desempeño de los algoritmos se evalúa con una validación cruzada *10-Fold* [102].

Como se presentó en la Sección 3.12, la validación *k-Fold* permite la selección de modelos basada en la predicción del error promedio [103]. En este caso, se ejecuta una función llamada *RepeatedKFold* que repite *k-Fold* n veces y se usa cuando se requiere ejecutar la validación cruzada *k-Fold* n veces, produciendo diferentes mangas con distintos elementos en cada repetición; lo que permiten mejorar el rendimiento del algoritmo al contar con más elementos de evaluación. Al elegir $n=2$, *k-Fold* con $k = 10$ se ejecuta dos veces para obtener un total de 20 particiones. El algoritmo se entrena con cada partición y con ello se evalúa la capacidad de generalización ya que se evita entrenar con todo el conjunto a la vez, lo que puede provocar sobreentrenamiento.

Finalmente, la última etapa de la Figura 25 marcada en rojo utiliza los modelos entrenados para generar las estimaciones de tiempo de finalización y costo total de los proyectos de software. Las estimaciones generadas por cada partición se promedian para obtener los valores de cada uno de los 21 proyectos aplicando la técnica de ensamble tipo *Bagging* en donde las estimaciones individuales de cada modelo entrenado son promediadas para generar una estimación final. Promediar diferentes hipótesis reduce el riesgo de elegir una hipótesis incorrecta y, por lo tanto, mejora el rendimiento predictivo general [100]. La ejecución de la estrategia algorítmica referida se muestra en la Figura 26. Al obtener las métricas de evaluación de las estimaciones promediadas es cuando podemos analizar el desempeño real del algoritmo y la calidad de las estimaciones, esto es, qué tan cercanas son las predicciones a los valores reales.

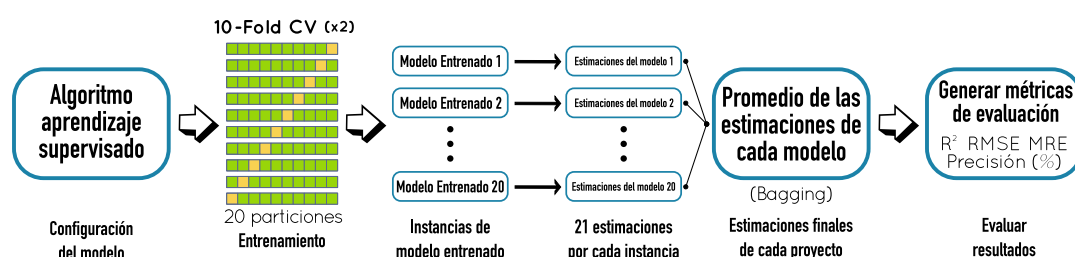


Figura 26: Proceso de entrenamiento del algoritmo de aprendizaje supervisado, cada modelo es entrenado 20 veces con subconjuntos de datos distintos.

6.1. Recursos utilizados

La evaluación experimental requiere de un conjunto de bibliotecas que permitan implementar los algoritmos de aprendizaje automático, así como aquellos procedimientos para manejar la información, cargar los datos en arreglos y realizar operaciones matemáticas para calcular las funciones de error. A continuación se enlistan las herramientas utilizadas en este proyecto de investigación:

- **scikit-learn**⁴, es una herramienta utilizada para el análisis predictivo de datos, construido con *NumPy*, *SciPy*, y *matplotlib*, ofrece una licencia de tipo *Open Source* cuya biblioteca (API) brinda clases y funciones de aprendizaje supervisado, no supervisado, selección de modelos y evaluación, inspección, visualización y transformación de conjuntos de datos.
- **NumPy**⁵, es un proyecto *Open Source* que tiene como objetivo habilitar el cómputo

⁴<https://scikit-learn.org/stable/index.html>

⁵<https://numpy.org>

numérico con *Python*; su principal uso es convertir un conjunto de datos a potentes arreglos multidimensionales que permiten realizar eficientemente diversos cálculos numéricos.

- **Pandas**⁶, es una herramienta de análisis y manipulación de datos de código abierto rápida, potente, flexible y fácil de usar. Construido sobre el lenguaje de programación *Python*., facilita la manipulación de archivos .csv para su posterior procesamiento con *NumPy*.
- **Keras**⁷, es una API de aprendizaje profundo escrita en *Python*, que se ejecuta sobre la plataforma de aprendizaje automático *TensorFlow*, útil para la creación de redes neuronales en un entorno de alto nivel, minimiza la cantidad de acciones requeridas por el usuario para casos de uso común y diseño de redes.
- **SciKeras**⁸, encapsulador o *Wrapper* compatible con *scikit-learn* para poder ejecutar validaciones cruzadas en modelos creados con *Keras*.
- **Google Colab**, es una plataforma que permite escribir y ejecutar *Python* en un navegador web sin la necesidad de instalarlo localmente. Su versión gratuita ofrece acceso a una tarjeta gráfica (GPU) para realizar entrenamientos de redes neuronales.

Los códigos de los programas en el contexto de este experimento se ejecutan en la versión gratuita de *Google Colab* que brinda dos procesadores Intel(R) Xeon(R) CPU @ 2.20 GHz, 13 GB de memoria RAM, 106 GB de espacio en disco y una GPU Tesla K80 con (x2) 2496 cores a una velocidad de 560 MHz y (x2) 12GB de memoria GDDR5 @ 2500 MHz. Una ventaja de utilizar esta plataforma es que toda la información se guarda en la nube y la ejecución de los algoritmos no requiere de la instalación de los recursos de manera local.

6.2. Preprocesamiento y selección de características

La preparación de la información para la etapa de entrenamiento no forma parte del proceso para configurar un modelo de aprendizaje automático, pero es un paso indispensable para asegurarse que los datos solo contienen valores numéricos escalados entre cero y uno, y que las categorías tienen su correspondiente codificación a enteros. La normalización de datos es simple, aplicando una instancia de *MinMaxScaler* de la API de *scikit-learn*, definimos el rango del intervalo $\min = 0$, $\max = 1$, X es el valor a escalar y el resultado se almacena en X_{scaled} , aplicando la siguientes ecuaciones:

$$\begin{aligned} X_{\text{std}} &= (X - X.\min(\text{axis} = 0)) / (X.\max(\text{axis} = 0) - X.\min(\text{axis} = 0)) \\ X_{\text{scaled}} &= X_{\text{std}} * (\max - \min) + \min \end{aligned} \quad (12)$$

$\text{axis} = 0$ es una instrucción de *Python* utilizada para hacer referencia a los renglones de un arreglo, esto significa que el recorrido se va a hacer renglón a renglón, cuando $\text{axis} = 1$ se interpreta que el recorrido va a ser columna a columna. En el ejemplo de la ecuación 12 se

⁶<https://pandas.pydata.org>

⁷<https://keras.io>

⁸<https://pypi.org/project/scikeras/>

indica que en X_{std} se tomará el mínimo valor de todos los renglones del conjunto X y el máximo valor de todos los renglones del conjunto X .

Parte del preprocesamiento de la información incluye la selección de características que formarán parte del conjunto de entrada, pues no todas las características del conjunto de entrada son relevantes para la estimación de esfuerzo de los proyectos de software, por lo que necesitamos de una técnica que seleccione solo aquellas que son más importantes.

6.2.1. Extracción de características

El modelo de estimación aplicado en este proyecto, basado en [7] utiliza siete características que corresponden a los datos de: *Effort*, *Vi*, *D*, *V*, *SprintSize*, *WorkDays*, *TeamSalary*. La ejecución de la técnica RFECV ordena las características de acuerdo a su importancia, concluyendo que el esfuerzo y la velocidad son las dos características más importantes para estimar el tiempo de finalización, mientras que el esfuerzo y el salario del equipo son las dos características más importantes para estimar el costo total. El resultado de la importancia de cada característica se muestra en la Tabla 5, cada una de las entradas se ordena del uno al seis, siendo uno la característica más importante y seis la menos importante para las estimaciones de tiempo y costo.

RFECV	Salidas	
Entradas	Tiempo	Costo
Effort	1	1
Vi	2	5
D	3	3
V	5	4
SprintSize	7	7
WorkDays	6	6
TeamSalary	4	2

Tabla 5: Las características se enumeran de acuerdo a su prioridad, siendo la primera la más importante y al sexta la menos importante, este ordenamiento es posterior al calculo de la importancia de Gini.

Con la información obtenida acerca de las prioridades de las características, el *dataset* inicial [7] se reduce a la Tabla 4 presentada anteriormente. Como el salario del equipo no es una métrica de los métodos ágiles, esta columna no se incluye como parte de los datos de entrada. El modelo de estimación en este proyecto utiliza las características del proyecto (esfuerzo) y las métricas disponibles (velocidad), así como las categorías de clasificación por tamaño como se muestra en la Figura 27.

6.3. Diseño de algoritmos

El desarrollo de los experimentos tiene la finalidad de conocer el funcionamiento de los algoritmos y la configuración de sus hiperparámetros. Mediante evaluación empírica se seleccionan las características que ofrecen los mejores porcentajes de precisión de las estimaciones de los 21 proyectos y reduzcan lo más posible los errores al comparar la estimación con el valor real. Cada algoritmo se programa de acuerdo a sus características elementales.

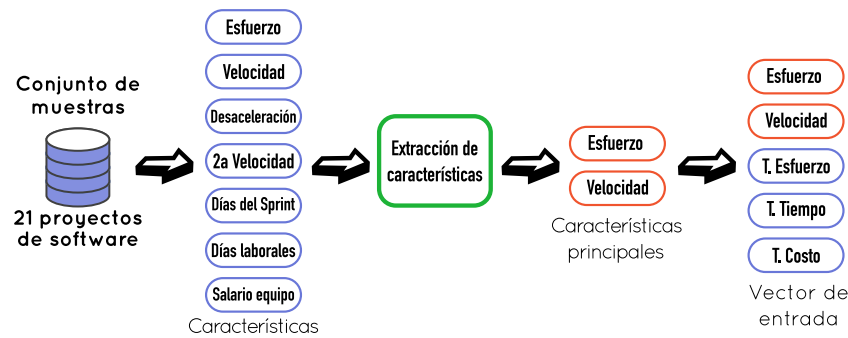


Figura 27: El vector de entrada se compone de las características principales de un proyecto y las categorías de clasificación por tamaño.

6.3.1. Diseño empírico del algoritmo KNN

Los parámetros de configuración del algoritmo KNN son:

- **Número de vecinos cercanos**
- **Método para ponderar las muestras:** ponderaciones uniformes para todas las muestras o ponderaciones basadas en la distancia que hay con los vecinos
- **Estructura de datos utilizadas** para mejorar el cálculo de las distancias: *k-d Tree*
- La **métrica de distancia** utilizada por los árboles al realizar los cálculos: distancia euclidiana, manhattan o ecuación de minkowski.

Código 1: Creación de modelo KNN

```

1  def createKNNmodel(k):
2  KNN = KNeighborsRegressor(n_neighbors=k, weights='distance',
3  algorithm='kd_tree', metric='minkowski')
4  classifier_pipeline = make_pipeline(MinMaxScaler(), KNN)
5  return classifier_pipeline
6  KNN = createKNNmodel(2)

```

La implementación del Código 1 define una función *createKNNmodel()* que recibe como parámetro el número de vecinos cercanos a considerar. *KNeighborsRegressor* implementa el aprendizaje basado en los k vecinos más cercanos de cada punto de consulta, donde k es un valor entero especificado por el usuario. El método de búsqueda se realiza con la estructura *k-d Tree*, *make_pipeline* construye un objeto que aplica secuencialmente una lista de transformaciones y un estimador final; en este caso se le indica que primero ejecuta el procedimiento para normalizar los datos (*MinMaxScaler*) y posteriormente se aplica el estimador previamente definido *KNN*.

Las pruebas para estimación de tiempo y costo se realizan en un solo modelo; cuando $k \in [1, 9]$ el número de vecinos que da los mejores coeficientes de determinación R^2 es $k = 2$. Las ponderaciones basadas en la distancia aplican la ecuación de Minkowski (1). La primera ejecución del modelo se realiza cuando los datos de entrada son el esfuerzo y la velocidad, una segunda ejecución utiliza el esfuerzo, la velocidad y las tres categorías de clasificación de tamaño. En la Tabla 6 se muestran cuatro casos resumidos en los que se cambian los datos de entrada.

Parámetro		Modelo KNN			
Configuración	Vecinos (k)	2			
	Pesos	Distancia			
	Algoritmo	KD-Tree			
	Métrica de distancia	Minkowski			
	Validación Cruzada k-Fold	RepeatedKFold(n_splits=10, n_repeats=2), se generan 20 estimadores			
Datos	Entradas	Effort, Vi	Effort, Vi, SizeEffort, SizeTime, SizeCost	Effort, Vi	Effort, Vi, SizeEffort, SizeTime, SizeCost
	Salidas	Tiempo		Costo	

Tabla 6: Configuración a implementar del algoritmo KNN.

6.3.2. Diseño empírico de árboles de decisión

Tomando en cuenta los parámetros de configuración básicos de los árboles de decisión mencionados en la Sección 3.7 del marco teórico definimos la profundidad máxima, el número de muestras mínimas para dividir y el criterio de regresión. La implementación del algoritmo a través del API de *scikit-learn* toma valores de configuración propuestos en la literatura y aquellos parámetros que no se indican explícitamente en el código toman los valores por defecto definidos en el API. En la Tabla 7 se muestran las tres configuraciones de los distintos algoritmos y su codificación se muestra en el Código 2.

Parámetro		Modelo Decision Tree			
Configuración	Max profundidad	5			
	min_samples_split	2			
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	

(a) Configuración a implementar del algoritmo DT

Parámetro		Modelo Random Forest			
Configuración	Num estimadores	10			
	Max profundidad	6			
	min_samples_split	2			
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	

(b) Configuración a implementar del algoritmo RF

Parámetro		Modelo Adaboost			
Configuración	Num estimadores	10			
	Max profundidad	5			
	min_samples_split	2			
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	

(c) Configuración a implementar del algoritmo AdaBoost

Tabla 7: Configuraciones utilizadas para realizar las estimaciones de los proyectos con árboles de decisión.

Código 2: Creación de variantes de árboles de decisión.

```
1 regrDT = DecisionTreeRegressor(max_depth= 5, min_samples_split=2, random_state=0)
2 regrRF = RandomForestRegressor(n_estimators=10, max_depth=6, random_state=0)
3 regrAda = AdaBoostRegressor(regrDT, random_state=0, n_estimators=10)
```

El objeto *DecisionTreeRegressor* representa un árbol de decisión con una profundidad de nivel cinco y dos muestras mínimas para dividir un nodo interno. El parámetro *random.state* controla la aleatoriedad del estimador, pues las características siempre se permutan aleatoriamente en cada división. El algoritmo *Random Forest* y *AdaBoost* se configuran en *regrRF* y *regrAda* respectivamente, el número de árboles que componen ambos ensambles se indica con el parámetro *n_estimators*. Por defecto la métrica es el error cuadrático medio. El incremento en el número de estimadores se detiene cuando la precisión de los resultados no tiene una mejora significativa.

6.3.3. Diseño empírico de red neuronal MLP

El modelo de red neuronal de tipo *feedforward* se diseña tomando en cuenta las siguientes características:

- Número de neuronas
- Número de capas
- Tipo de optimizador
- Función de activación
- Tasa de aprendizaje
- Ruido Gaussiano o *dropout* aplicado
- Inicialización de ponderaciones y sesgos con *Glorot_uniform* (también llamada Normalización Xavier) [51]

Las primeras cinco características se determinan empíricamente, esto es, el modelo se prueba con distinto número de neuronas y de capas ocultas para seleccionar aquella configuración que arroje las mejores predicciones minimizando el error y utilizando el menor número de neuronas y capas. El número mínimo de neuronas por capa está dado por la dimensión de los datos de entrada y el número máximo en todas las pruebas no supera las 20 neuronas. De manera similar la red más básica contiene una sola capa oculta y las pruebas se realizan con dos, tres y hasta 5 capas, seleccionando el número de capas que ofrezca los mejores resultados, siguiendo los pasos de diseño sugeridos en la Sección 3.8.9.

Respecto a los optimizadores, se comparan *RMSProp*, *Adadelata* y *Adam*, estos son los algoritmos de entrenamiento de una red neuronal que aplican la retropropagación. Por ejemplo, *Adam* es un método para la optimización estocástica eficiente que solo requiere gradientes de primer orden y poca necesidad de memoria [138]. El criterio de selección del optimizador se basa en el tiempo de ejecución y el número de épocas

necesarias para entrenar el modelo. En otras palabras, se selecciona el optimizador que reduzca al máximo el error y lo haga en el menor número de épocas, ya que al realizar la validación cruzada *10-Fold*, en cada partición se entrena un modelo y al repetirla n veces se entrenan $10 * n$ modelos. La tasa de entrenamiento conserva su valor estándar de 0.001 y no se modifica al comparar los distintos optimizadores.

Otra manera de reducir las épocas de entrenamiento consiste en inicializar las ponderaciones de la red mediante distintas técnicas. El estado del arte aplica la inicialización uniforme Xavier [51]. El método de inicialización Xavier normalizado se calcula como un número aleatorio con una distribución de probabilidad uniforme dentro del límite $[-l, l]$ con

$$l = \sqrt{\frac{6}{(a + b)}}$$

donde a es el número de unidades de entradas en una neurona y b es el número de unidades de salida. Esto significa que la inicialización toma en cuenta las unidades que se encuentran en la capa anterior y posterior.

Para determinar los mecanismos de regularización que se mencionan en el marco teórico, es necesario entrenar la red sin capas de *dropout* y sin ruido gaussiano, posteriormente se agregan capas de *dropout* si la brecha del error entre entrenamiento y validación es grande. El ruido gaussiano a agregar debe tener una desviación estándar muy pequeña para evitar destruir los datos a medida que se propagan por las capas de la red. El ruido agregado se aplica en la salida de cada neurona y en las capas ocultas. En la Tabla 8 se muestran las configuraciones a ejecutar del modelo basado en una red neuronal de tipo perceptrón multicapa. La implementación de la red se hace a través del modelo secuencial del API de *Keras*:

Parámetro				
Configuración	Optimizador	Adam		
	Activación	Tanh		
	Capas Ocultas	2		
	Neuronas p/Capa	8		
	Tasa de aprendizaje	0.001		
	Función de pérdida (Loss)	Error Cuadrático Médio (MSE)		
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2		
	Normalización de datos	MinMaxScaler con valores entre (0,1)		
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo
Regularización	Dropout	No		
	Ruido Gaussiano	stddev=0.00001		
	No. Capas de ruido	2		
Máximo de épocas en entrenamiento		2000		

Tabla 8: Configuraciones de un MLP para realizar estimaciones de tiempo y costo.

Como se observa en la Tabla 8 la arquitectura base de la red neuronal consiste en cuatro capas, el ruido gaussiano se distribuye a la entrada de las dos capas ocultas,

haciendo que la información recibida a la entrada de las capas ocultas esté modificada por el ruido aplicado. La métrica utilizada durante el entrenamiento es el error cuadrático medio. Aquellos parámetros que no se indican explícitamente en el código toman los valores predeterminados, como por ejemplo la tasa de aprendizaje y la inicialización de las ponderaciones. Una ventaja de utilizar el API de *scikit-learn* es que todos los algoritmos se implementan con técnicas del estado del arte, por lo que sus parámetros de configuración por defecto serán aquellos indicados en la literatura.

La función *get_reg()* del Código 3 crea el modelo de tipo secuencial al que se le van agregando capas mediante el método *add*. La capa de entrada siempre debe tener una dimensión de entrada que es igual al tamaño de los datos de entrada, la función de activación *tanh* se aplica a todas las neuronas de la capa. A la salida de la capa de entrada y de la primera capa oculta se agrega una capa de ruido gaussiano. Una capa de tipo *GaussianNoise* no contiene neuronas, solamente modifica los valores de salida de la capa anterior. El optimizador es utilizado durante la retropropagación, y la función de pérdida especificada *mse* se calcula al finalizar cada época.

Código 3: Creación de modelo de red MLP

```
1  def get_reg():
2  model = Sequential()
3  i=8
4  model.add(Dense(i, input_dim=inputDim, activation='tanh', name="CapaEntrada"))
5  model.add(GaussianNoise(stddev=0.00001))
6  model.add(Dense(i, activation='tanh', name="PrimeraOculta"))#
7  model.add(GaussianNoise(stddev=0.00001))
8  model.add(Dense(i, activation='tanh', name="SegundaOculta"))#
9  model.add(Dense(1, activation='linear', name="CapaSalida"))
10 opt = keras.optimizers.Adam() #Adam y tanh
11 model.compile(loss='mse', optimizer=opt)
12 return model
```

La configuración adicional del modelo de la red MLP se muestra en el Código 4. Un procedimiento de tipo *callback* puede realizar distintas acciones en varias etapas del entrenamiento, al inicio o al final de cada época, o después de cada lote de información. *EarlyStopping* deja de entrenar cuando una métrica monitoreada haya dejado de mejorar, esto es, cuando el valor de error ya no disminuye pasado cierto umbral.

El valor *patience = 80* indica que al alcanzar un valor mínimo de error se entrenará solo 80 épocas extras, si el valor ya no disminuye durante las 80 iteraciones el entrenamiento termina y la red restaura el valor de sus ponderaciones a la época donde se alcanzó el valor mínimo. La ventana debe ser lo suficientemente grande en caso de que la red caiga en un mínimo local, si *patience* tiene un valor pequeño es posible que no se pueda alcanzar un valor menor de la función de error.

KerasRegressor es una herramienta que permite crear un objeto que contiene una red neuronal creada con Keras para poder aplicar la validación cruzada de *scikit-learn*. El objeto establece que el entrenamiento se detiene si se alcanza el límite de 2000 épocas, también puede terminar si la función de pérdida no disminuye en 80 épocas consecutivas. Cuando el contenedor recibe el conjunto de entrenamiento reserva el 30 % para validación y las métricas que aplica son el error cuadrático medio y el coeficiente de determinación.

Código 4: Configuración adicional de la red MLP

```

1  cb = [EarlyStopping(monitor="val_loss", patience=80,mode='min',\
2  restore_best_weights=True)]
3  reg = KerasRegressor(model=get_reg, loss="mse",
4  metrics=[KerasRegressor.r_squared],
5  epochs=2000,
6  callbacks=[cb],
7  validation_split=0.3,
8  verbose =0)

```

6.3.4. Diseño empírico de Red Neuronal Recurrente

El caso de la red neuronal recurrente atiende los mismos pasos de diseño referidos en el marco teórico, teniendo como principal diferencia el uso de capas recurrentes en lugar de capas densas. El modelo puede utilizar dos tipos de capas, recurrencia simple y una capa de tipo LSTM como en la Figura 10. Debido a la escasez de datos, una red LSTM no puede explotar todo su potencial por lo que una red recurrente simple se ajusta mejor a conjuntos de datos pequeños. Recordemos que una red recurrente aplicada a problemas de regresión trabaja con series temporales por lo que se necesita una cantidad robusta de datos para su buen funcionamiento.

La configuración del tercer algoritmo de aprendizaje automático se muestra en la Tabla 9. La arquitectura contiene cuatro capas, las dos primeras capas son de tipo recurrente y las últimas dos son capas densas. En este caso solo a la entrada de la primer capa oculta se agrega ruido gaussiano además de aplicarle *dropout* recurrente de 0.1 o 10 % a cada una de las capas de tipo *SimpleRNN*, esto significa que durante el entrenamiento, el 10 % de las neuronas apagan el enlace de recurrencia, teniendo como valor de salida cero.

Este tipo de red recurrente simple requiere un número mayor de neuronas por capa, ya que al realizar el entrenamiento con $n < 40$ la magnitud de la función de pérdida no disminuye lo suficiente provocando que las estimaciones de la red tengan baja precisión. De acuerdo a los pasos de diseño de la red, la arquitectura se incrementa en tamaño hasta que haya una mejora en las estimaciones, o en su caso una reducción de la función de pérdida.

El modelo de red neuronal recurrente del Código 5 tiene como capa de entrada y primera capa oculta una capa de tipo RNN en donde la salida de la capa se realimenta a la entrada de la misma capa y todas sus neuronas están interconectadas con las siguiente capa como en las capas tipo *Dense* del MLP. Una capa RNN usa un ciclo *for* para iterar sobre los pasos de tiempo de una secuencia (*timesteps*), mientras mantiene un estado interno que codifica información sobre los pasos de tiempo que ha visto hasta el momento⁹.

Código 5: Creación de un modelo de red recurrente

```

1  model = Sequential()
2  model.add(SimpleRNN(units=i,input_shape=in_dim, \
3  activation="relu",return_sequences=True))
4  model.add(GaussianNoise(stddev=0.00001))
5  model.add(SimpleRNN(units=i, activation="relu",\

```

⁹https://keras.io/guides/working_with_rnns/

Parámetro				
Configuración	Optimizador	Adam		
	Activación	Relu		
	Activación recurrente	Tanh		
	Capas Ocultas	2		
	Tipo de celda recurrente	Simple RNN		
	Neuronas p/Capa RNN	40		
	Neuronas p/Capa Dense	10		
	Tasa de aprendizaje	0.001		
	Loss	Error Cuadrático Medio (MSE)		
	Normalización de datos	MinMaxScaler en el rango (0,1)		
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi
	Salida	Tiempo		Costo
Regularización	Dropout recurrente	0.1		
	Ruido Gaussiano	stddev=0.00001		
	No. Capas de ruido	1		
Máximo de épocas en entrenamiento		900		

Tabla 9: Configuraciones de una RNN para realizar estimaciones de tiempo y costo.

```

6 recurrent_dropout=0.1))
7 model.add(Dense(10))
8 model.add(Dense(out_dim))
9 model.compile(loss='mse', optimizer='adam')

```

Las redes recurrentes son capaces de reconocer secuencias a partir de un conjunto de datos, la información debe estar formateada como una serie temporal a modo de poder crear una ventana de tiempo de n observaciones e ir la recorriendo hasta cubrir todo el conjunto de entrenamiento. En la Figura 28 se observa a la izquierda un vector de dimensión 1×5 , que representa las cinco características de un proyecto de software. En una red MLP, los 42 proyectos generados para el conjunto de entrenamiento van ingresando uno a uno, es decir cinco características entran a la vez a la red neuronal MLP.

En cambio la información que ingresa a una red recurrente no puede ser recibida de la misma manera que una red MLP. En una red recurrente es necesario crear bloques de n muestras, y cada muestra tiene pasos de tiempo, donde cada paso es un proyecto. Los pasos de tiempo forman una ventana que abarca cuatro proyectos, el quinto proyecto es el valor a predecir. Como se observa a la derecha de la Figura 28 la información se presenta en tres dimensiones, cada muestra es un conjunto de cuatro proyectos, y cada proyecto contiene cinco características.

Una muestra de cuatro proyectos ingresan a la red, es decir, se analizan cuatro proyectos para predecir un quinto proyecto, conforme avanza el entrenamiento la ventana de tiempo se va recorriendo en una posición. En otras palabras, la red necesita un historial de cuatro proyectos (una muestra) para predecir el valor del siguiente proyecto. Si todos los proyectos forman una serie temporal, tenemos una lista de 42

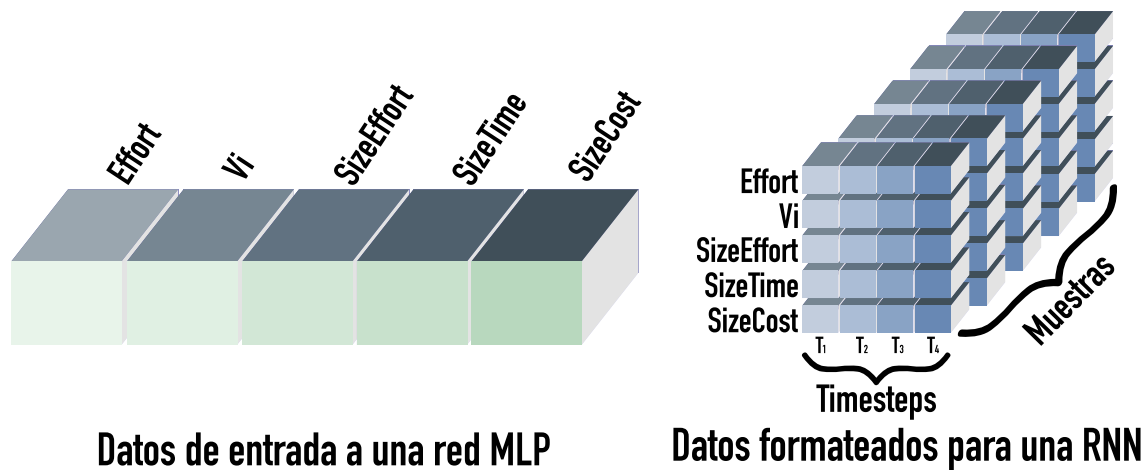


Figura 28: Una red recurrente recibe la información formateada en tres dimensiones (muestras, características y pasos de tiempo).

posiciones, la ventana comienza con los primeros cuatro proyectos y se predice el quinto, después la ventana se desplaza del segundo al quinto proyecto y se predice el sexto, el desplazamiento continúa hasta que se haya recorrido toda la serie temporal.

Una desventaja del conjunto de datos es que no se proporcionan las fechas de desarrollo de los proyectos y al ser un conjunto pequeño, aparentemente no forman una serie temporal. Sin embargo cada uno de los proyectos tiene un desarrollo en el tiempo, su organización como serie temporal es aleatoria y a pesar de no contar con una fecha precisa del desarrollo sí es posible aplicar las redes recurrentes a series temporales ya que todos los proyectos forman una colección de observaciones hechas secuencialmente a través del tiempo [145, 37].

Con el procedimiento del Código 6 podemos transformar el conjunto de datos de entrenamiento al formato especificado para la red recurrente. La función recibe como parámetros los datos de entrada, los valores a predecir y el número de pasos de tiempo. Mediante un ciclo *for* se recorre todo el conjunto ordenando cuatro proyectos para *X* y un proyecto para *Y*, es decir, cuatro proyectos para la ventana de tiempo (*timesteps*), y un proyecto para el valor a predecir. En cada iteración la ventana se va recorriendo hasta cubrir todos los elementos del conjunto de entrenamiento. La función regresa los valores ordenados para *X* y para *Y*.

Código 6: Formateo de datos de entrada para la red recurrente

```

1  def convertData(datax, datay, step):
2      X, Y = [], []
3      for i in range(len(datax)-step):
4          d = i+step
5          X.append(datax[i:d,])
6          Y.append(datay[d])
7      return array(X), array(Y)

```

6.3.5. Diseño empírico de Autoencoder

Basado en las propiedades del *autoencoder* para comprimir información, este algoritmo es entrenado con el conjunto de datos original ya que el bloque *Encoder* tiene la capacidad de realizar la técnica PCA y extraer las mejores características del conjunto de datos de la Tabla 3.

El algoritmo de aprendizaje profundo toma como entrada todo el conjunto de datos y agrega las etiquetas de tamaño agrupando los proyectos por tamaño de esfuerzo, tiempo y costo. Hay tres arquitecturas probadas en los experimentos, y la estructura general de la configuración se muestra en la Tabla 10.

Parámetro	Modelo Autoencoder			
Configuración	Optimizador	Adam		
	Activación	Tanh		
	Capas Ocultas	4		
	Dimensión de entrada	9		
	Neuronas p/Capa	reducción y aumento en 3 neuronas		
	Tasa de aprendizaje	0.001		
	Función de pérdida (Loss)	Error Cuadrático Medio (MSE)		
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2		
	Normalización de datos	MinMaxScaler con valores entre (0,1)		
	Entradas (columnas)	Dataset	Dataset + Categorías	Dataset + Categorías
Regularización	Dropout	No		
	Ruido Gaussiano	stddev=0.00001		
	No. Capas de ruido	2		
Máximo de épocas en entrenamiento		Tiempo		
		Costo		
		3000		

Tabla 10: Configuración general del Autoencoder para realizar las estimaciones de tiempo y costo.

Primero se construye un *Autoencoder* tradicional que tiene dos bloques, un codificador y un decodificador. La segunda arquitectura solo prueba el bloque del codificador, y la tercera usa el codificador como primer bloque y capas de tipo *feedforward* como segundo bloque, para probar si la red más profunda mejora la precisión de las estimaciones. Finalmente, el método de *AdaBoost* se usa con la arquitectura del codificador para probar el desempeño del algoritmo.

La primera arquitectura tiene un codificador cuya capa de entrada es igual a las nueve características, cada capa se reduce en tres neuronas por lo que el codificador tiene nueve neuronas en su primer capa, seis en la segunda y tres neuronas en la tercera. El decodificador aumenta en tres neuronas teniendo dos capas, de seis y nueve neuronas como se muestra en la Figura 29a. La segunda arquitectura usa solo el primer bloque, un codificador con tres capas que van de nueve a seis y tres neuronas como se muestra en la Figura 29b. Una tercera arquitectura agrega dos capas adicionales de dos neuronas al codificador para obtener una red más profunda, como se muestra en la Figura 29c. La configuración general para el *Autoencoder* se muestra en la Tabla 10.

La implementación de un *Autoencoder* es muy similar a un MLP, la única diferencia es que en cada capa oculta variamos el número de neuronas, pero las capas son las mismas, estructuras tipo *Dense* donde todas las unidades están conectadas con los elementos de la siguiente capa. El Código 7 incluye las capas para implementar las tres

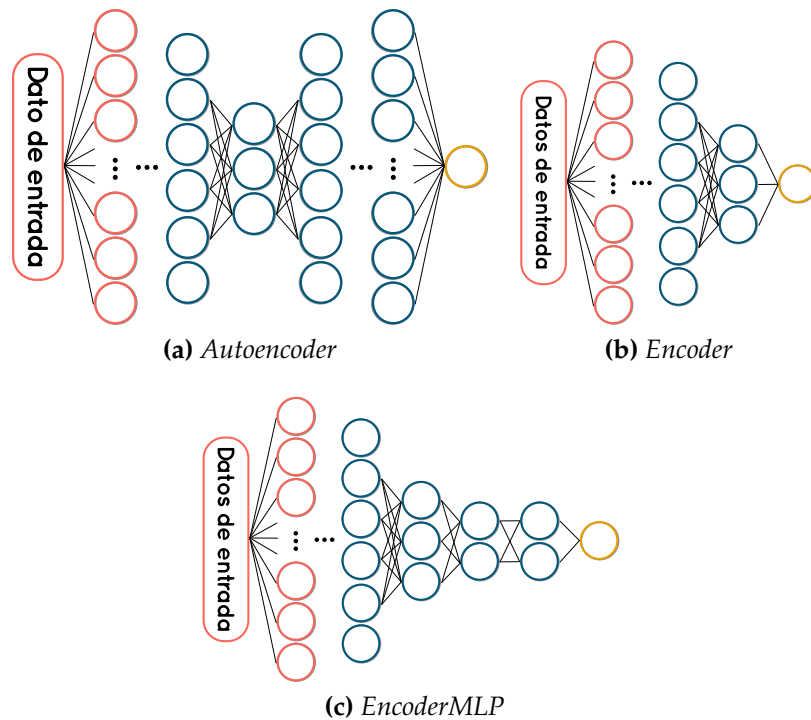


Figura 29: Tres arquitecturas utilizadas para realizar las estimaciones de los proyectos de software.

arquitecturas, comentadas están las líneas que agregan dos capas tipo MLP, cada capa tiene una etiqueta que permite ubicar a cuál bloque de la red neuronal representa.

Código 7: Creación de modelo de red tipo Autoencoder.

```

1  def get_reg():
2  model = Sequential()
3  i=9
4  model.add(Dense(i, input_dim=inputDim, activation='tanh', name="CapaEntrada"))
5  model.add(GaussianNoise(stddev=0.00001))
6  model.add(Dense(i-3, activation='tanh', name="PrimeraEncoder" ))#
7  model.add(GaussianNoise(stddev=0.00001))
8  model.add(Dense(i-6, activation='tanh', name="SegundaEncoder" ))#
9  model.add(Dense(i-3, activation='tanh', name="PrimeraDecoder" ))#
10 model.add(Dense(i, activation='tanh', name="SegundaDecoder" ))#
11 #MLP
12 #model.add(Dense(2, activation='tanh', name="MLP1"))#
13 #model.add(Dense(2, activation='tanh', name="MLP2"))#
14 model.add(Dense(1, activation='linear', name="CapaSalida"))
15 opt = keras.optimizers.Adam() #Adam y tanh
16 model.compile(loss='mse', optimizer=opt)
17 return model

```

6.4. Validación

Como ya se ha explicado en el Capítulo 3 de Marco Teórico, la validación cruzada es una estrategia ampliamente utilizada debido a su simplicidad y universalidad [113]. La validación cruzada permite evaluar el rendimiento de un algoritmo. Su implementación mediante *scikit-learn* facilita su ejecución, requiriendo la definición de un conjunto de parámetros: el número de particiones (*Folds*), las veces que se repetirá la validación *10-Fold* y un valor entero que controla la aleatoriedad de cada instancia

de validación cruzada repetida.

La validación cruzada se define en *rkf* y el procedimiento *cross_validate* es el encargado de ejecutar la validación especificada anteriormente; en sus parámetros se le indica que aplicará el procedimiento al estimador *reg* (algoritmo de aprendizaje supervisado), con los datos *x_train_scaled*, *y_train_scaled*, regresando los modelos entrenados una vez que termine el procedimiento y regresando las métricas de evaluación R^2 , MSE, RMSE y la varianza. Así, *cross_validate* regresa diccionarios que incluye los estimadores entrenados y sus métricas, de esta manera se puede recuperar la información y calcular la precisión obtenida en cada partición. La configuración de la validación cruzada se muestra en el Código 8.

Código 8: Configuración de la validación cruzada.

```
1  n_splits=10
2  n_repeats=2
3  rkf = RepeatedKFold(n_splits=n_splits, n_repeats=n_repeats, random_state=30)
4  cv_results = cross_validate(reg, x_train_scaled, y_train_scaled,
5  cv=rkf, return_estimator=True, return_train_score=True,
6  scoring=('r2', 'neg_mean_squared_error', 'explained_variance',
7  'neg_root_mean_squared_error'))
```


7. Resultados

Los resultados se agrupan en bloques de acuerdo a cada técnica; cada grupo contiene el desempeño del entrenamiento, los valores obtenidos de las funciones de error, el coeficiente de determinación R^2 y la comparativa del uso de categorías de clasificación. Las tablas con los resultados siguen el mismo formato presentado en la configuración de los algoritmos. El código completo que permite la obtención de los resultados se presenta en Apéndice de este documento.

7.1. Regresión lineal simple y polinomial

Una de las técnicas elementales de regresión y un primer punto de comparación es la regresión lineal simple o también conocida como el método de mínimos cuadrados ordinarios (OLS). Este algoritmo no requiere de programación ya que con facilidad es posible realizar la regresión en una hoja de cálculo. De manera similar podemos calcular la regresión de los polinomios de segundo y tercer grado. Al analizar los puntos de los 21 proyectos se obtienen tres curvas diferentes con sus respectivas ecuaciones y coeficientes de diferenciación.

En la Figura 30 se presenta la gráfica de los 21 proyectos, teniendo el tiempo de finalización en función del esfuerzo, es decir, al ubicar los proyectos en el plano representados como puntos rojos se pueden trazar tres curvas de regresión. La primera es una recta en color rojo cuya ecuación está dada por $y = 0.2891x + 9.1052$ y su coeficiente de determinación $R^2 = 0.8348$.

La curva de regresión de segundo grado está representada en color amarillo y su coeficiente de determinación es de $R^2 = 0.8812$, la curva de tercer grado se traza en color verde y su coeficiente $R^2 = 0.9144$. A medida que incrementamos el grado del polinomio la curva se sobreajusta a los datos, por lo que la precisión de las estimaciones aumenta pero se cae en el riesgo de que el algoritmo termine memorizando la información, y en un escenario real no tenga la capacidad de estimar con precisión nueva información. De las tres curvas, el polinomio de tercer grado se ajusta mejor a los datos y tiene el mayor valor de R^2 . El objetivo de todas las técnicas de aprendizaje aplicadas consiste en igualar o mejorar este valor para poder obtener estimaciones que se aproximen más a los datos reales. Un algoritmo tendrá un mal desempeño si su coeficiente de determinación es menor al coeficiente de la regresión lineal simple, por lo tanto es preferible utilizar el método OLS a programar técnicas de aprendizaje supervisado o aprendizaje profundo.

La gráfica correspondiente al costo total se muestra en la Figura 31 y tiene un formato similar a la gráfica de tiempo de finalización, presentando la regresión lineal simple en rojo, la curva de segundo grado en amarillo y la curva de tercer grado en verde. A diferencia del tiempo de finalización, el costo total tiene un buen coeficiente de determinación desde la recta de regresión simple. En los tres casos el valor $R^2 > 0.90$, esto ocurre porque la dispersión de los datos de costo es menor comparado con el tiempo de finalización.

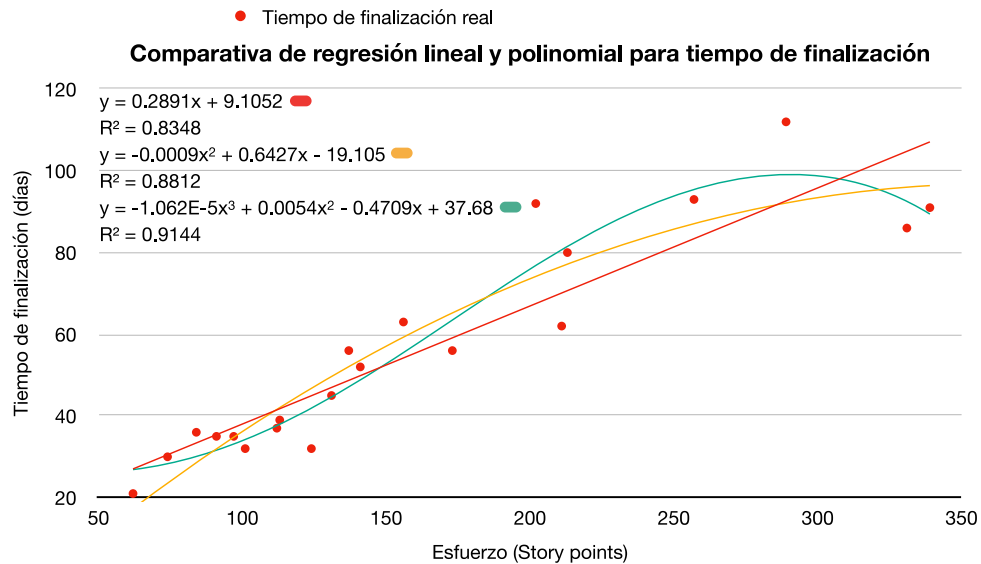


Figura 30: El modelo de regresión que mejor se adapta al tiempo de finalización real de los 21 proyectos es un polinomio de grado tres (curva en color verde), R^2 mayor es mejor.

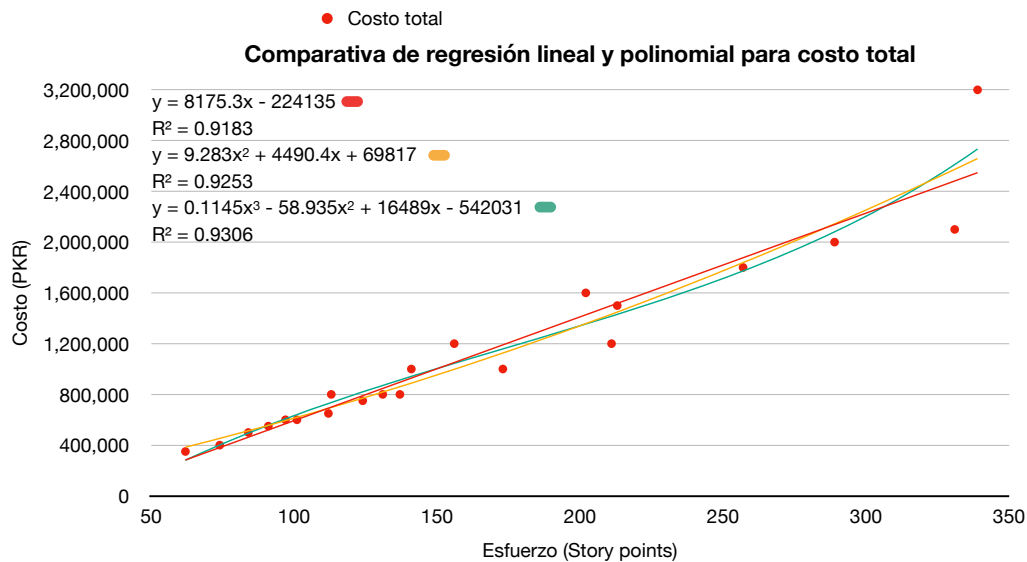


Figura 31: Regresión lineal en rojo, polinomio cuadrado en amarillo y cúbico en verde.

7.2. Algoritmo KNN

La ejecución de la validación cruzada implica el entrenamiento de un modelo con cada partición (*fold*). El algoritmo *KNN* se considera como modelo entrenado cuando se ha construido el árbol para calcular las distancias entre la muestra a predecir y el conjunto de datos seleccionados. En total se obtienen 20 modelos entrenados cuyas predicciones se promedian para obtener las estimaciones de tiempo de finalización y costo total. En la Tabla 11 se agrupan los resultados obtenidos por el algoritmo. Las cuatro columnas se dividen en grupos de dos, y cada una de las columnas de un grupo representa los resultados de las estimaciones con y sin categorías de clasificación.

Se observa que cuando los datos de entrada son el esfuerzo y la velocidad la precisión alcanza un 94.17 %, con un coeficiente de determinación $R^2 = 0.976$. Estos valores

	Parámetro	Modelo K-Nearest Neighbor (KNN)			
Configuración	Vecinos (k)	2			
	Pesos	Distancia			
	Algoritmo	KD-Tree			
	Métrica de distancia	Minkowski			
	Validación Cruzada k-Fold	RepeatedKfold(n_splits=10, n_repeats=2), se generan 20 estimadores			
Datos	Entradas	Effort, Vi	Effort, Vi, SizeEffort, SizeTime, SizeCost	Effort, Vi	Effort, Vi, SizeEffort, SizeTime, SizeCost
	Salidas	Tiempo		Costo	
Resultados	Predicción(%)	94.17	94.7	94.65	94.54
	R²	0.976	0.984	0.9833	0.986
	Error relativo MRE	0.0583	0.053	0.0535	0.0546
	RMSE	3.96	3.23	89,031	81,388
	Explained Variance	0.9772	0.9840	0.9839	0.9861

Tabla 11: Resultados de la ejecución del algoritmo KNN.

superan a la regresión lineal simple calculada con anterioridad cuyo valor de $R^2 = 0.8348$. El algoritmo solo necesita los valores de los dos vecinos más cercanos para realizar las aproximaciones.

La segunda columna del grupo son los valores obtenidos al integrar las categorías de clasificación, la dimensión de los datos de entrada aumenta de dos a cinco, y el resultado muestra que hay una mejora en las estimaciones de tiempo con una precisión del 94.7 %. Al aumentar la precisión y el coeficiente de determinación, el error relativo medio disminuyó en 0.0053.

Comparando las estimaciones de la propuesta original [7], el algoritmo KNN tuvo mejores resultados en 15 de los 21 proyectos en el caso de las estimaciones de tiempo como se muestra en la Tabla 12. El criterio consiste en que la diferencia entre el tiempo real y la estimación de KNN debe ser menor a la diferencia del tiempo real con la predicción hecha por [7].

La comparativa de costos en la Tabla 11 tiene la misma organización que el tiempo de finalización, y sus dos columnas corresponden a los datos de entrada con y sin categorías de clasificación. El porcentaje de precisión de las estimaciones disminuyó ligeramente de 94.65 % a 94.54 %, el coeficiente de determinación aumentó en 0.0027 pero el error relativo aumentó lo que indica que el desempeño del algoritmo se redujo.

Al comparar las estimaciones de costo total de [7] y las estimaciones del algoritmo KNN en la Tabla 12, en 12 de los 21 proyectos hubo una mejora. Para ello se toma la diferencia de la columna *ActualCost* y KNN, luego se compara con la diferencia entre *ActualCost* y la columna *Zia et al.*, si KNN tiene una diferencia menor entonces el

algoritmo mejoró la estimación.

No.	Effort	ActualTime	Zia et al.	KNN	ActualCost	Zia et al.	KNN	Mejora Tiempo	Mejora Costo
1	156	63	58	60	1,200,000	1,023,207	1,139,828	SI	SI
2	202	92	81	93	1,600,000	1,680,664	1,635,421	SI	SI
3	173	56	52	53	1,000,000	992,270	987,734	SI	NO
4	331	86	87	89	2,100,000	2,002,767	2,332,740	NO	NO
5	124	32	29	36	750,000	676,081	781,066	NO	SI
6	339	91	95	93	3,200,000	2,895,133	3,047,614	SI	SI
7	97	35	29	34	600,000	540,114	572,930	SI	SI
8	257	93	84	94	1,800,000	1,614,079	1,797,263	SI	SI
9	84	36	35	28	500,000	507,265	395,136	NO	NO
10	211	62	66	60	1,200,000	1,267,180	1,160,367	SI	SI
11	131	45	41	44	800,000	786,732	810,599	SI	SI
12	112	37	39	37	650,000	597,143	641,800	SI	SI
13	101	32	35	31	600,000	538,495	499,306	SI	NO
14	74	30	26	28	400,000	394,546	391,415	SI	NO
15	62	21	22	23	350,000	330,561	363,751	NO	SI
16	289	112	103	110	2,000,000	1,971,485	2,069,899	SI	NO
17	113	39	40	46	800,000	770,857	899,717	NO	NO
18	141	52	50	51	1,000,000	961,866	985,374	SI	SI
19	213	80	76	86	1,500,000	1,453,032	1,626,830	NO	NO
20	137	56	51	55	800,000	854,348	815,698	SI	SI
21	91	35	34	35	550,000	567,484	528,559	SI	NO

Tabla 12: Comparativa de estimaciones hechas por el algoritmo KNN y la propuesta original [7].

Finalmente al graficar todos los puntos en un mismo plano (Figura 32), se observa que las rectas de regresión presentan cierta similitud y las estimaciones en la mayoría de los casos no distan demasiado de los valores reales, ya que la desviación estándar máxima en las estimaciones de tiempo no supera los seis días, y en el caso del costo total la desviación estándar máxima no supera las 343,970 rupias como se muestra en la Tabla 13.

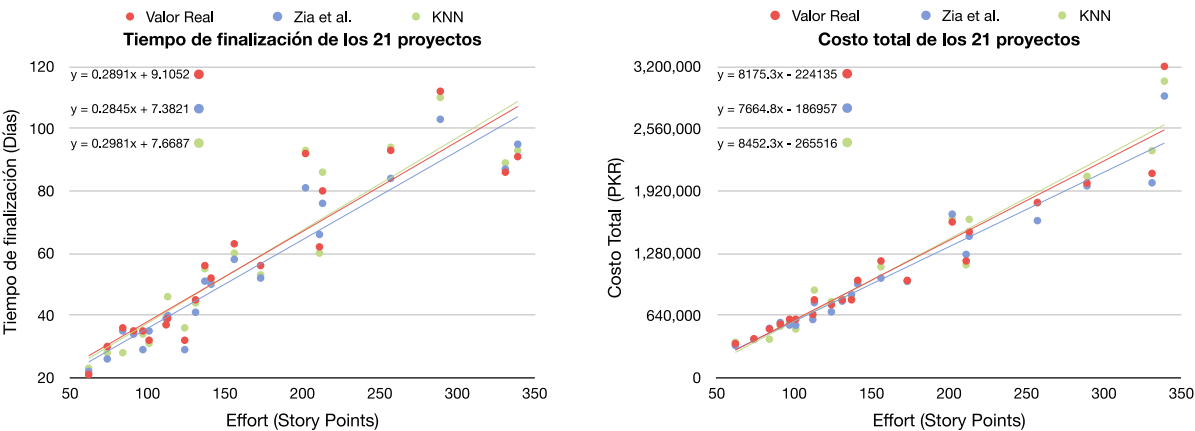


Figura 32: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y estimaciones de KNN en verde.

No.	ActualTime	KNN	ΔT (Días)	ActualCost	KNN	ΔC (PKR)
1	63	60 $\pm$ 4	3	1,200,000	1,139,828 $\pm$ 81,847	60,172
2	92	93 $\pm$ 2	1	1,600,000	1,635,421 $\pm$ 80,019	35,421
3	56	53 $\pm$ 2	3	1,000,000	987,734 $\pm$ 25,256	12,266
4	86	89 $\pm$ 4	3	2,100,000	2,332,740 $\pm$ 139,441	232,740
5	32	36 $\pm$ 5	4	750,000	781,066 $\pm$ 61,427	31,066
6	91	93 $\pm$ 3	2	3,200,000	3,047,614 $\pm$ 343,970	152,386
7	35	34 $\pm$ 3	1	600,000	572,930 $\pm$ 67,457	27,070
8	93	94 $\pm$ 3	1	1,800,000	1,797,263 $\pm$ 2,065	2,737
9	36	28 $\pm$ 1	8	500,000	395,136 $\pm$ 17,479	104,864
10	62	60 $\pm$ 5	2	1,200,000	1,160,367 $\pm$ 108,629	39,633
11	45	44 $\pm$ 1	1	800,000	810,599 $\pm$ 28,177	10,599
12	37	37 $\pm$ 1	0	650,000	641,800 $\pm$ 22,805	8,200
13	32	31 $\pm$ 1	1	600,000	499,306 $\pm$ 19,933	100,694
14	30	28 $\pm$ 2	2	400,000	391,415 $\pm$ 7,128	8,585
15	21	23 $\pm$ 3	2	350,000	363,751 $\pm$ 23,112	13,751
16	112	110 $\pm$ 6	2	2,000,000	2,069,899 $\pm$ 156,819	69,899
17	39	46 $\pm$ 3	7	800,000	899,717 $\pm$ 35,990	99,717
18	52	51 $\pm$ 1	1	1,000,000	985,374 $\pm$ 30,270	14,626
19	80	86 $\pm$ 2	6	1,500,000	1,626,830 $\pm$ 38,075	126,830
20	56	55 $\pm$ 3	1	800,000	815,698 $\pm$ 34,135	15,698
21	35	35 $\pm$ 2	0	550,000	528,559 $\pm$ 11,953	21,441

Tabla 13: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y el algoritmo KNN.

7.3. Árboles de Decisión

Las distintas variantes de los árboles de decisiones se ejecutan de manera independiente y una vez que se obtienen todos los resultados se incorporan al ensamble que calcula las estimaciones finales de tiempo y costo. Al igual que el resto de los algoritmos, la fase de entrenamiento, la validación y las pruebas se evalúan mediante la precisión de la predicción, el coeficiente de determinación, el error cuadrático medio, la raíz del error cuadrático medio, el error relativo medio y la varianza. Esto da como resultado tablas similares a KNN que agrupan las métricas de cada fase, el formato se mantiene teniendo dos grupos, uno para las estimaciones de tiempo y otro para las estimaciones de costo. Cada variable de salida presenta dos casos, cuando se utiliza el esfuerzo y la velocidad como datos de entrada, y cuando se incorporan las categorías de clasificación.

Como se observa en la Tabla 14, todas las ejecuciones tuvieron buen desempeño en la etapa de prueba al comparar los valores de precisión y R^2 . El algoritmo tiene la tendencia a caer en sobreajuste (*overfitting*) cuando la profundidad del árbol es grande, pues en esos casos la precisión de las estimaciones es de 99 % con coeficientes de 0.99. Una de las alertas que permiten identificar la memorización de la información consiste en la comparación de los errores de entrenamiento y validación, y la comparación de los coeficientes R^2 en entrenamiento, validación y prueba.

Parámetro		Modelo Decision Tree			
Configuración	Max profundidad	5			
	min_samples_split	2			
	k-Fold Cross Validation	RepeatedKfold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	
Entrenamiento	Error Cuadrático Medio (MSE)	0.0002	0.0002	2.9304E-05	2.0226E-05
	Raíz del MSE (RMSE)	0.0136	0.0122	0.0049	0.0039
	R^2	0.9971	0.9980	0.9995	0.9997
	Explained Variance	0.9971	0.9980	0.9995	0.9997
Validación	MSE	0.0027	0.0064	0.0057	0.0048
	RMSE	0.0417	0.0531	0.0392	0.0383
	R^2	0.9472	0.8599	0.9314	0.9156
	Explained Variance	0.9596	0.9038	0.9581	0.9441
Prueba	Precisión (%)	97.45	96.57	96.49	97.33
	R^2	0.9963	0.9910	0.9924	0.9944
	MRE	0.0255	0.0343	0.0351	0.0267
	MSE	2.43	5.86	3,612,415,756	2,637,510,055
	RMSE	1.56	2.42	60,103	51,357
	Explained Variance	0.9964	0.9914	0.9934	0.9953

Tabla 14: Métricas de evaluación de la ejecución del algoritmo DT.

En el caso de las estimaciones de tiempo, al utilizar categorías de clasificación tenemos un coeficiente $R^2 = 0.9980$ para entrenamiento, $R^2 = 0.8599$ para validación y $R^2 = 0.9910$ para prueba. Durante la etapa de validación los errores aumentan al realizar las estimaciones, lo que disminuye el coeficiente R^2 en 0.1381 pero las varianzas se mantienen por encima de 0.9.

En la Tabla 15 se comparan las estimaciones del trabajo [7] con los árboles de decisiones. Para el caso de las estimaciones de tiempo de finalización, el algoritmo DT mejoró en 15 de los 21 proyectos. En el caso de las estimaciones de costo hubo una mejora en 16 de los 21 proyectos.

La dispersión de las estimaciones hechas por la propuesta original [7] y el algoritmo DT para tiempo y costo se muestran en la Figura 33. Al realizar regresión simple con los datos obtenidos se observa que las rectas de regresión de los valores reales y el algoritmo DT son bastante cercanas para el tiempo de finalización, y en el caso del costo total están ligeramente separadas pero los puntos que representan a los proyectos son cercanos.

No.	Effort	ActualTime	Zia et al.	DT	ActualCost	Zia et al.	DT	Mejora Tiempo	Mejora Costo
1	156	63	58	62	1,200,000	1,023,207	1,155,125	SI	SI
2	202	92	81	92	1,600,000	1,680,664	1,596,500	SI	SI
3	173	56	52	54	1,000,000	992,270	1,011,792	SI	NO
4	331	86	87	87	2,100,000	2,002,767	2,103,250	NO	SI
5	124	32	29	34	750,000	676,081	759,500	SI	SI
6	339	91	95	92	3,200,000	2,895,133	3,145,000	SI	SI
7	97	35	29	34	600,000	540,114	606,065	SI	SI
8	257	93	84	93	1,800,000	1,614,079	1,800,000	SI	SI
9	84	36	35	34	500,000	507,265	508,333	NO	NO
10	211	62	66	60	1,200,000	1,267,180	1,171,875	SI	SI
11	131	45	41	45	800,000	786,732	797,500	SI	SI
12	112	37	39	37	650,000	597,143	637,986	SI	SI
13	101	32	35	34	600,000	538,495	606,065	SI	SI
14	74	30	26	31	400,000	394,546	398,750	SI	SI
15	62	21	22	20	350,000	330,561	354,750	NO	SI
16	289	112	103	111	2,000,000	1,971,485	1,961,000	SI	NO
17	113	39	40	38	800,000	770,857	681,340	NO	NO
18	141	52	50	53	1,000,000	961,866	818,750	SI	NO
19	213	80	76	83	1,500,000	1,453,032	1,522,000	SI	SI
20	137	56	51	47	800,000	854,348	798,625	NO	SI
21	91	35	34	34	550,000	567,484	543,333	NO	SI

Tabla 15: Comparativa de estimaciones hechas por el algoritmo DT y la propuesta [7].

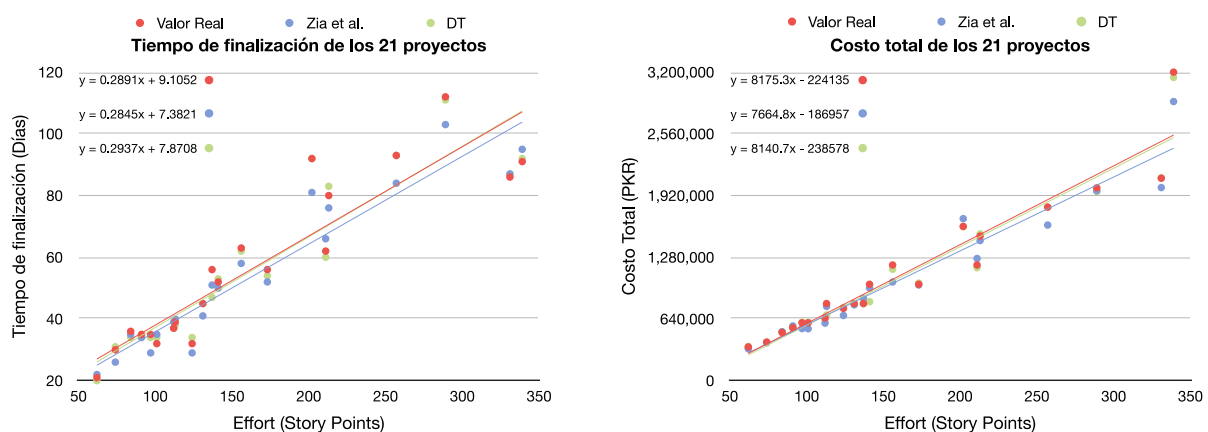


Figura 33: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y algoritmo DT en verde.

Las estimaciones hechas por los modelos entrenados en cada partición tuvieron valores muy cercanos, ya que la desviación estándar más grande fue de diez días o dos semanas laborales, y la desviación estándar más grande para las estimaciones de costo fue de 245,967 rupias.

Al comparar la diferencia de días de la columna (ΔT) de la Tabla 16, solo uno de los proyectos tuvo nueve días de diferencia respecto al valor real y el valor central de la estimación. El resto de los proyectos se mantuvo por debajo de la semana laboral. Comparando la diferencia de las estimaciones de costo de la columna (ΔC), solo dos proyectos tuvieron diferencias por encima de las cien mil rupias, el resto se mantuvo debajo incluso de las 60,000 rupias.

No.	ActualTime	DT	ΔT (Días)	ActualCost	DT	ΔC (PKR)
1	63	62 $\pm$ 3	1	1,200,000	1,155,125 $\pm$ 66,996	44,875
2	92	92 $\pm$ 1	0	1,600,000	1,596,500 $\pm$ 7,452	3,500
3	56	54 $\pm$ 2	2	1,000,000	1,011,792 $\pm$ 44,329	11,792
4	86	87 $\pm$ 0	1	2,100,000	2,103,250 $\pm$ 1,642	3,250
5	32	34 $\pm$ 1	2	750,000	759,500 $\pm$ 10,594	9,500
6	91	92 $\pm$ 5	1	3,200,000	3,145,000 $\pm$ 245,967	55,000
7	35	34 $\pm$ 1	1	600,000	606,065 $\pm$ 5,741	6,065
8	93	93 $\pm$ 0	0	1,800,000	1,800,000 $\pm$ 0	0
9	36	34 $\pm$ 1	2	500,000	508,333 $\pm$ 13,245	8,333
10	62	60 $\pm$ 1	2	1,200,000	1,171,875 $\pm$ 33,175	28,125
11	45	45 $\pm$ 0	0	800,000	797,500 $\pm$ 6,335	2,500
12	37	37 $\pm$ 1	0	650,000	637,986 $\pm$ 13,876	12,014
13	32	34 $\pm$ 1	2	600,000	606,065 $\pm$ 5,741	6,065
14	30	31 $\pm$ 2	1	400,000	398,750 $\pm$ 5,590	1,250
15	21	20 $\pm$ 1	1	350,000	354,750 $\pm$ 5,730	4,750
16	112	111 $\pm$ 0	1	2,000,000	1,961,000 $\pm$ 19,708	39,000
17	39	38 $\pm$ 1	1	800,000	681,340 $\pm$ 44,985	118,660
18	52	53 $\pm$ 2	1	1,000,000	818,750 $\pm$ 62,558	181,250
19	80	83 $\pm$ 10	3	1,500,000	1,522,000 $\pm$ 98,387	22,000
20	56	47 $\pm$ 4	9	800,000	798,625 $\pm$ 4,252	1,375
21	35	34 $\pm$ 1	1	550,000	543,333 $\pm$ 10,679	6,667

Tabla 16: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales del algoritmo DT.

7.3.1. Random Forest

La ejecución de los bosques aleatorios tiene un tamaño de diez estimadores; en relación con el algoritmo DT no hubo una mejora en las estimaciones. Las métricas de evaluación se agrupan en la Tabla 17, en este algoritmo el uso de categorías de clasificación también mejoró las estimaciones de tiempo de finalización y costo total. Observamos que la precisión y el coeficiente de determinación en las estimaciones de tiempo aumentaron de 95.44 % a 96.92 % y de 0.9794 a 0.9869 respectivamente.

Para el caso del costo total el aumento fue de 96.08 % a 97.03 % en precisión, y de 0.9858 a 0.9885 en coeficiente R^2 . La ventaja de RF sobre DT es que en la etapa de validación los errores fueron menores en las estimaciones de tiempo utilizando categorías, y los coeficientes de determinación de tiempo y costo fueron mayores a los obtenidos por DT.

Parámetro		Modelo Random Forest			
Configuración	Num estimadores	10			
	Max profundidad	6			
	min_samples_split	2			
	k-Fold Cross Validation	RepeatedKfold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	
Entrenamiento	Error Cuadrático Medio (MSE)	0.0011	0.0008	0.0007	0.0006
	Raíz del MSE (RMSE)	0.0330	0.0274	0.0252	0.0236
	R ²	0.9854	0.9900	0.9880	0.9896
	Explained Variance	0.9859	0.9908	0.9882	0.9898
Validación	MSE	0.0046	0.0035	0.0066	0.0061
	RMSE	0.0603	0.0504	0.0519	0.0486
	R ²	0.8980	0.9223	0.9211	0.9298
	Explained Variance	0.9262	0.9499	0.9444	0.9534
Prueba	Precisión (%)	95.44	96.92	96.08	97.03
	R ²	0.9794	0.9869	0.9858	0.9885
	MRE	0.0456	0.0308	0.0392	0.0297
	MSE	13.38	8.52	6,723,275,745	5,443,614,797
	RMSE	3.66	2.92	81,996	73,781
	Explained Variance	0.9795	0.9869	0.9870	0.9890

Tabla 17: Métricas de evaluación de la ejecución del algoritmo RF.

De los 21 proyectos, la técnica de RF mejoró las estimaciones de tiempo y costo en 14 proyectos como se indica en la Tabla 18. El algoritmo RF no logró mejorar el número total de proyectos estimados comparado con DT, teniendo un proyecto menos para las estimaciones de tiempo y dos proyectos menos para las estimaciones de costo. Sin embargo, ambas rectas de regresión (Figura 34) se aproximan bastante a las rectas rojas de los datos reales, y nuevamente, la dispersión de los datos es pequeña al graficarlos en el plano.

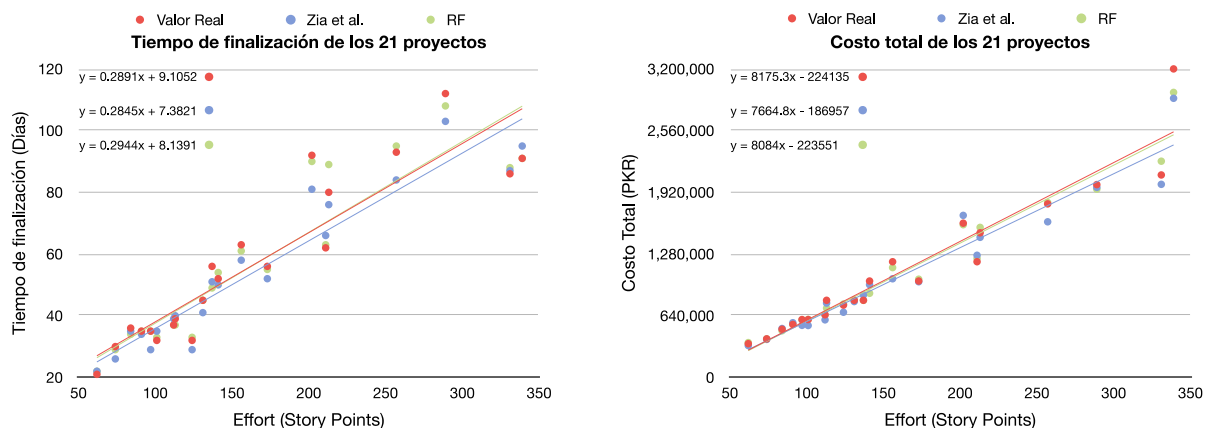


Figura 34: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y algoritmo RF en verde.

No.	Effort	ActualTime	Zia et al.	RF	ActualCost	Zia et al.	RF	Mejora Tiempo	Mejora Costo
1	156	63	58	61	1,200,000	1,023,207	1,139,050	SI	SI
2	202	92	81	90	1,600,000	1,680,664	1,583,750	SI	SI
3	173	56	52	55	1,000,000	992,270	1,018,964	SI	NO
4	331	86	87	88	2,100,000	2,002,767	2,242,725	NO	NO
5	124	32	29	33	750,000	676,081	761,358	SI	SI
6	339	91	95	91	3,200,000	2,895,133	2,956,300	SI	SI
7	97	35	29	35	600,000	540,114	597,867	SI	SI
8	257	93	84	95	1,800,000	1,614,079	1,813,600	SI	SI
9	84	36	35	34	500,000	507,265	483,824	NO	NO
10	211	62	66	63	1,200,000	1,267,180	1,234,562	SI	SI
11	131	45	41	45	800,000	786,732	804,908	SI	SI
12	112	37	39	37	650,000	597,143	647,647	SI	SI
13	101	32	35	33	600,000	538,495	602,123	SI	SI
14	74	30	26	29	400,000	394,546	399,830	SI	SI
15	62	21	22	22	350,000	330,561	361,330	NO	SI
16	289	112	103	108	2,000,000	1,971,485	1,952,125	SI	NO
17	113	39	40	37	800,000	770,857	717,856	NO	NO
18	141	52	50	54	1,000,000	961,866	872,764	NO	NO
19	213	80	76	89	1,500,000	1,453,032	1,556,350	NO	NO
20	137	56	51	49	800,000	854,348	799,625	NO	SI
21	91	35	34	35	550,000	567,484	551,563	SI	SI

Tabla 18: Comparativa de estimaciones hechas por el algoritmo RF y la propuesta [7].

Al observar la diferencia de días y de rupias del algoritmo RF en la Tabla 19, las estimaciones de tiempo se mantuvieron por debajo de las dos semanas laborales, y en tres proyectos la diferencia superó las cien mil rupias. La desviación estándar de los modelos mantuvieron sus estimaciones en no más de cuatro días con respecto al valor central y en costo la desviación estándar mayor fue de 256, 518 rupias para el proyecto más costoso de 3, 200, 000.

No.	ActualTime	RF	ΔT (Días)	ActualCost	RF	ΔC (PKR)
1	63	61 $\pm$ 2	2	1,200,000	1,139,050 $\pm$ 55,325	60,950
2	92	90 $\pm$ 2	2	1,600,000	1,583,750 $\pm$ 25,915	16,250
3	56	55 $\pm$ 1	1	1,000,000	1,018,964 $\pm$ 24,665	18,964
4	86	88 $\pm$ 1	2	2,100,000	2,242,725 $\pm$ 119,010	142,725
5	32	33 $\pm$ 1	1	750,000	761,358 $\pm$ 8,250	11,358
6	91	91 $\pm$ 2	0	3,200,000	2,956,300 $\pm$ 256,518	243,700
7	35	35 $\pm$ 1	0	600,000	597,867 $\pm$ 9,428	2,133
8	93	95 $\pm$ 2	2	1,800,000	1,813,600 $\pm$ 31,951	13,600
9	36	34 $\pm$ 1	2	500,000	483,824 $\pm$ 17,547	16,176
10	62	63 $\pm$ 3	1	1,200,000	1,234,562 $\pm$ 50,963	34,562
11	45	45 $\pm$ 1	0	800,000	804,908 $\pm$ 19,500	4,908
12	37	37 $\pm$ 1	0	650,000	647,647 $\pm$ 10,646	2,353
13	32	33 $\pm$ 0	1	600,000	602,123 $\pm$ 6,009	2,123
14	30	29 $\pm$ 1	1	400,000	399,830 $\pm$ 9,826	170
15	21	22 $\pm$ 2	1	350,000	361,330 $\pm$ 9,523	11,330
16	112	108 $\pm$ 3	4	2,000,000	1,952,125 $\pm$ 49,503	47,875
17	39	37 $\pm$ 1	2	800,000	717,856 $\pm$ 25,525	82,144
18	52	54 $\pm$ 1	2	1,000,000	872,764 $\pm$ 33,161	127,236
19	80	89 $\pm$ 4	9	1,500,000	1,556,350 $\pm$ 47,042	56,350
20	56	49 $\pm$ 2	7	800,000	799,625 $\pm$ 16,991	375
21	35	35 $\pm$ 0	0	550,000	551,563 $\pm$ 9,801	1,563

Tabla 19: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y el algoritmo RF.

7.3.2. AdaBoost

La última variante de los árboles de decisiones aplicada es la técnica de ensamble *AdaBoost*. Recordemos que a diferencia de RF, *AdaBoost* conecta todos sus estimadores asignando ponderaciones a aquellas estimaciones con menor precisión, así cada uno de los árboles va depurando los peores datos con cada iteración del entrenamiento.

Las métricas de evaluación del algoritmo se muestran en la Tabla 20. El uso de las categorías de clasificación mejoraron las estimaciones de tiempo y costo al aumentar los valores del coeficiente de diferenciación y reducir los errores en la etapa de entrenamiento, validación y prueba con respecto a los escenarios donde los datos de entrada corresponden al esfuerzo y la velocidad. La raíz del error cuadrático medio al aplicar categorías se mantiene por debajo de los tres días, y al realizar las estimaciones de costo, el error se reduce a 60,432 rupias.

Los puntos de estimación en el plano (Figura 35) se mantienen cercanos a los valores reales, y las rectas de regresión mantienen su proximidad tanto en tiempo de finalización como en costo total de los proyectos. La diferencia en el valor de la pendiente de la recta de las estimaciones de [7] es mayor que la pendiente de la recta de las estimaciones generadas con el algoritmo de ensamble.

Parámetro		Modelo Adaboost			
Configuración	Num estimadores	10			
	Max profundidad	6			
	min_samples_split	2			
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	
Entrenamiento	Error Cuadrático Medio (MSE)	0.0007	0.0005	0.0004	0.0003
	Raíz del MSE (RMSE)	0.0261	0.0220	0.0209	0.0180
	R ²	0.9910	0.9935	0.9922	0.9943
	Explained Variance	0.9910	0.9936	0.9924	0.9944
Validación	MSE	0.0027	0.0022	0.0061	0.0055
	RMSE	0.0471	0.0399	0.0492	0.0398
	R ²	0.9357	0.9408	0.8813	0.9246
	Explained Variance	0.9507	0.9584	0.9143	0.9516
Prueba	Precisión (%)	95.49	96.44	94.74	95.19
	R ²	0.9883	0.9926	0.9903	0.9923
	MRE	0.0451	0.0356	0.0526	0.0481
	MSE	7.62	4.81	4,611,014,418	3,651,980,200
	RMSE	2.76	2.19	67,904	60,432
	Explained Variance	0.9886	0.9930	0.9912	0.9930

Tabla 20: Métricas de evaluación del algoritmo *AdaBoost*.

Comparando los valores de las estimaciones de la Tabla 21, el algoritmo *AdaBoost*

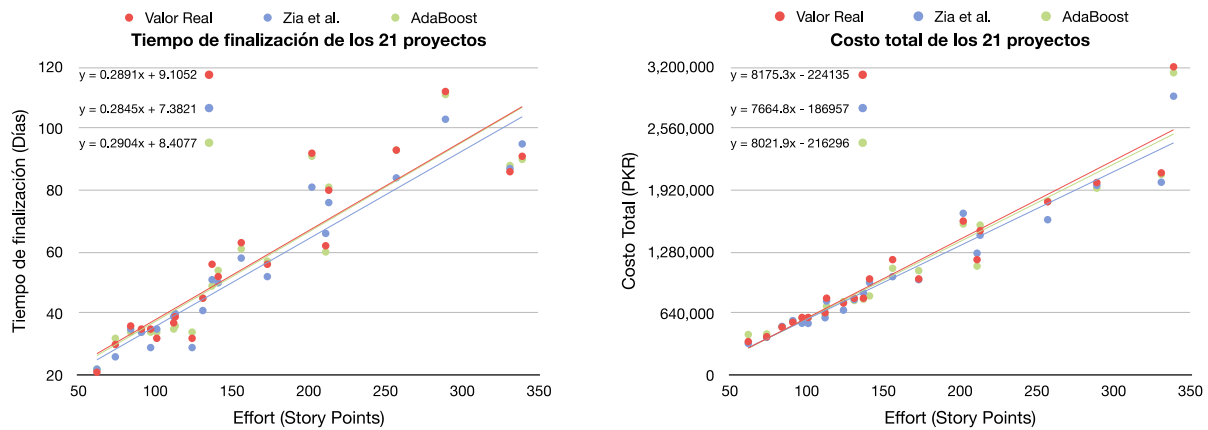


Figura 35: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y AdaBoost en verde.

No.	Effort	ActualTime	Zia et al.	AdaBoost	ActualCost	Zia et al.	AdaBoost	Mejora Tiempo	Mejora Costo
1	156	63	58	61	1,200,000	1,023,207	1,110,060	SI	SI
2	202	92	81	91	1,600,000	1,680,664	1,569,257	SI	SI
3	173	56	52	57	1,000,000	992,270	1,085,586	SI	NO
4	331	86	87	88	2,100,000	2,002,767	2,083,124	NO	SI
5	124	32	29	34	750,000	676,081	765,137	SI	SI
6	339	91	95	90	3,200,000	2,895,133	3,138,500	SI	SI
7	97	35	29	34	600,000	540,114	595,069	SI	SI
8	257	93	84	93	1,800,000	1,614,079	1,808,979	SI	SI
9	84	36	35	34	500,000	507,265	504,022	NO	SI
10	211	62	66	60	1,200,000	1,267,180	1,132,852	SI	SI
11	131	45	41	45	800,000	786,732	775,709	SI	NO
12	112	37	39	35	650,000	597,143	627,336	NO	SI
13	101	32	35	34	600,000	538,495	597,240	SI	SI
14	74	30	26	32	400,000	394,546	428,911	SI	NO
15	62	21	22	22	350,000	330,561	423,469	NO	NO
16	289	112	103	111	2,000,000	1,971,485	1,941,095	SI	NO
17	113	39	40	36	800,000	770,857	715,490	NO	NO
18	141	52	50	54	1,000,000	961,866	823,623	NO	NO
19	213	80	76	81	1,500,000	1,453,032	1,558,689	SI	NO
20	137	56	51	49	800,000	854,348	790,001	NO	SI
21	91	35	34	34	550,000	567,484	562,872	NO	SI

Tabla 21: Comparativa de estimaciones hechas por el algoritmo AdaBoost y la propuesta [7].

mejoró en 13 de los 21 proyectos con respecto a las predicciones de tiempo hechas por [7]. Al calcular el costo de los proyectos, las estimaciones también mejoraron en 13 de los 21 proyectos. *AdaBoost* no tuvo un mejor desempeño que el algoritmo DT.

Por último al observar en la Tabla 22 las diferencias entre el valor central de las estimaciones y el valor real del proyecto, hubo una diferencia máxima de 7 días para el tiempo de finalización y 176,377 rupias para el costo total. La mayoría de las diferencias en los proyectos se mantienen por debajo de la semana laboral y las cien mil rupias. Respecto a las desviaciones estándar, los modelos entrenados durante la validación cruzada presentaron una desviación máxima de cuatro días para tiempo y

No.	ActualTime	AdaBoost	ΔT (Días)	ActualCost	AdaBoost	ΔC (PKR)
1	63	61 $\pm$ 2	2	1,200,000	1,110,060 $\pm$ 48,657	89,940
2	92	91 $\pm$ 1	1	1,600,000	1,569,257 $\pm$ 33,452	30,743
3	56	57 $\pm$ 1	1	1,000,000	1,085,586 $\pm$ 40,064	85,586
4	86	88 $\pm$ 1	2	2,100,000	2,083,124 $\pm$ 32,971	16,876
5	32	34 $\pm$ 1	2	750,000	765,137 $\pm$ 27,292	15,137
6	91	90 $\pm$ 1	1	3,200,000	3,138,500 $\pm$ 275,036	61,500
7	35	34 $\pm$ 1	1	600,000	595,069 $\pm$ 33,960	4,931
8	93	93 $\pm$ 0	0	1,800,000	1,808,979 $\pm$ 39,268	8,979
9	36	34 $\pm$ 1	2	500,000	504,022 $\pm$ 50,317	4,022
10	62	60 $\pm$ 2	2	1,200,000	1,132,852 $\pm$ 34,986	67,148
11	45	45 $\pm$ 1	0	800,000	775,709 $\pm$ 23,482	24,291
12	37	35 $\pm$ 1	2	650,000	627,336 $\pm$ 35,346	22,664
13	32	34 $\pm$ 1	2	600,000	597,240 $\pm$ 33,628	2,760
14	30	32 $\pm$ 2	2	400,000	428,911 $\pm$ 41,470	28,911
15	21	22 $\pm$ 2	1	350,000	423,469 $\pm$ 35,288	73,469
16	112	111 $\pm$ 2	1	2,000,000	1,941,095 $\pm$ 31,296	58,905
17	39	36 $\pm$ 2	3	800,000	715,490 $\pm$ 46,874	84,510
18	52	54 $\pm$ 1	2	1,000,000	823,623 $\pm$ 56,509	176,377
19	80	81 $\pm$ 4	1	1,500,000	1,558,689 $\pm$ 53,474	58,689
20	56	49 $\pm$ 3	7	800,000	790,001 $\pm$ 32,265	9,999
21	35	34 $\pm$ 1	1	550,000	562,872 $\pm$ 21,090	12,872

Tabla 22: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y el algoritmo AdaBoost con árboles de decisiones.

275, 036 rupias para costo, de nuevo en esta ocasión la desviación estándar más grande la tiene el proyecto más costoso.

7.4. Redes Neuronales

El último algoritmo a considerar en este trabajo son las redes neuronales artificiales, la primera ejecución basada en MLP entrena el modelo con validación cruzada de 20 particiones. Las estimaciones de tiempo y costo corresponden al promedio de las estimaciones realizadas por todos los modelos entrenados al igual que las técnicas previamente evaluadas.

Se ejecutan dos modelos de redes neuronales, uno para predecir tiempo de finalización y otro para las estimaciones de costo, pues la implementación de la extensión (*wrapper*) que se utiliza para ejecutar la validación cruzada no permite múltiples salidas. La variación en número de neuronas indica que a mayor número de neuronas se reduce el número de épocas para entrenar los modelos, lo que reduce a su vez el tiempo de ejecución. El número mínimo de neuronas es igual a la dimensión de los datos de entrada y el número máximo de neuronas es de diez, al seguir incrementando el número de neuronas la red no mejora la precisión de la predicción.

7.4.1. Perceptrón Multicapa

Los resultados del entrenamiento de la red MLP se muestran en la Tabla 23 conservando la estructura y formato del resto de las tablas de los algoritmos. En la etapa de entrenamiento el modelo tuvo un promedio de 1196 épocas para estimación de tiempo y 700 para la estimación de costo. Al preclasificar los datos se observa en el grupo de las métricas de tiempo que el error cuadrático medio se disminuyó de 0.0048 a 0.0039, aunque en las métricas de costo no fue posible reducir el error ya que este aumentó de 0.0051 a 0.0054. De manera similar el coeficiente R^2 aumentó durante el entrenamiento de tiempo de finalización, y disminuyó en el entrenamiento de costos.

Recordemos que una partición de validación cruzada tiene 10 bloques, siete para entrenamiento y tres para validación; en nuestro caso la escasa información disponible provoca que la etapa de validación tenga un rendimiento menor al entrenamiento. Sin embargo, el desempeño general de la red neuronal es satisfactorio, ya que al comparar los coeficientes de determinación de las etapas de entrenamiento y prueba, los valores se mantienen por encima de 0.90. Al realizar las estimaciones con los valores de prueba, el algoritmo mejoró la precisión de las estimaciones usando categorías de clasificación aumentando de 91.61 % a 93.14 % para el caso de tiempo y de 90.47 % a 92.52 % para el caso de costo.

Tomando en cuenta los valores de los proyectos de la Tabla 24, la red MLP mejoró 12 de las estimaciones de tiempo de los proyectos, y solo 9 proyectos tuvieron mejor estimación de costo comparado con las predicciones hechas por el modelo original [7]. Desafortunadamente el algoritmo no logró superar el desempeño de la técnica de regresión aplicada por los autores en el trabajo original, ya que menos de la mitad de los proyectos tuvieron valores más cercanos al costo real.

Si bien las estimaciones generadas no superan al modelo original, su comparativa no presenta una gran dispersión de los datos. Como se observa en la Figura 36, las rectas

Parámetro		Modelo MultiLayer Perceptron (MLP)			
Configuración	Optimizador	Adam			
	Activación	Tanh			
	Capas Ocultas	2			
	Neuronas p/Capa	8			
	Tasa de aprendizaje	0.001			
	Función de pérdida (Loss)	Error Cuadrático Médio (MSE)			
	k-Fold Cross Validation	RepeatedKFold con 10-Fold y n_repeats=2			
	Normalización de datos	MinMaxScaler con valores entre (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	
Entrenamiento	Error Cuadrático Medio (MSE)	0.0048	0.0039	0.0051	0.0054
	Raíz del MSE (RMSE)	0.0689	0.0612	0.0696	0.0684
	R ²	0.9372	0.9494	0.9138	0.9104
	Explained Variance	0.9373	0.9495	0.9148	0.9109
Validación	MSE	0.0063	0.0052	0.0083	0.0084
	RMSE	0.0733	0.0650	0.0704	0.0764
	R ²	0.8644	0.8867	0.8354	0.7894
	Explained Variance	0.8979	0.9161	0.9041	0.8266
Prueba	Precisión (%)	91.61	93.14	90.47	92.52
	R ²	0.9399	0.9614	0.9201	0.9341
	MRE	0.0839	0.0686	0.0953	0.0748
	MSE	39.00	25.05	37,851,092,256	31,223,726,814
	RMSE	6.25	5.00	194,554	176,702
	Explained Variance	0.9402	0.9615	0.9201	0.9341
Regularización	Dropout	No			
	Ruido Gaussiano	stddev=0.00001			
	No. Capas de ruido	2			
Máximo de épocas en entrenamiento		2000			
Promedio de épocas en entrenamiento		784	1196	473	700

Tabla 23: Métricas de evaluación de la ejecución del algoritmo MLP.

de regresión generadas por las predicciones tienen valores de pendientes cercanos y la diferencia de días solo es significativa en uno de los proyectos. Para ello consideremos la Tabla 25, en donde la diferencia de días (ΔT) se mantiene por debajo de la semana laboral en la mayoría de los proyectos y solamente en uno de los proyectos la diferencia fue de 17 días. Este proyecto en particular presentó algún tipo de riesgo durante su desarrollo ya que es el proyecto que tomó más tiempo en finalizar pero no es el proyecto de mayor esfuerzo. El proyecto número seis con esfuerzo de 339 unidades tomó menos días que el proyecto número 16 con esfuerzo de 289 unidades. La red neuronal no es capaz de identificar estas variaciones de la información, por eso es que la estimación emitida está muy por debajo del valor real.

Al evaluar las estimaciones de costo hechas por el algoritmo MLP, solo en siete proyectos la diferencia fue mayor a cien mil rupias, y en uno de ellos superó el medio millón. La desviación estándar de los modelos entrenados refleja que sí existe una mayor dispersión de los datos al emitir el costo total, ya que en diez proyectos la desviación supera las cien mil rupias, y solo en cuatro proyectos la desviación se mantiene por debajo de las cincuenta mil rupias.

La técnica de la red neuronal de tipo MLP no logró superar el desempeño de los árboles de decisiones; tanto DT, RF y *AdaBoost* tuvieron un mejor desempeño presentando valores mayores de porcentaje de precisión, coeficiente R² y un mayor

No.	Effort	ActualTime	Zia et al.	MLP	ActualCost	Zia et al.	MLP	Mejora Tiempo	Mejora Costo
1	156	63	58	59	1,200,000	1,023,207	1,075,580	SI	SI
2	202	92	81	89	1,600,000	1,680,664	1,538,916	SI	SI
3	173	56	52	58	1,000,000	992,270	1,147,123	SI	NO
4	331	86	87	90	2,100,000	2,002,767	2,270,374	NO	NO
5	124	32	29	33	750,000	676,081	738,145	SI	SI
6	339	91	95	97	3,200,000	2,895,133	2,509,692	NO	NO
7	97	35	29	32	600,000	540,114	597,953	SI	SI
8	257	93	84	94	1,800,000	1,614,079	1,984,985	SI	SI
9	84	36	35	31	500,000	507,265	498,267	NO	SI
10	211	62	66	61	1,200,000	1,267,180	1,211,152	SI	SI
11	131	45	41	43	800,000	786,732	823,232	SI	NO
12	112	37	39	39	650,000	597,143	686,952	NO	SI
13	101	32	35	34	600,000	538,495	582,385	SI	SI
14	74	30	26	29	400,000	394,546	449,198	SI	NO
15	62	21	22	27	350,000	330,561	390,775	NO	NO
16	289	112	103	95	2,000,000	1,971,485	2,191,890	NO	NO
17	113	39	40	45	800,000	770,857	728,496	NO	NO
18	141	52	50	51	1,000,000	961,866	924,479	SI	NO
19	213	80	76	86	1,500,000	1,453,032	1,604,152	NO	NO
20	137	56	51	54	800,000	854,348	893,999	SI	NO
21	91	35	34	33	550,000	567,484	506,491	NO	NO

Tabla 24: Comparativa de estimaciones hechas por el algoritmo MLP y la propuesta [7].

número de proyectos mejor estimados con respecto al modelo de [7].

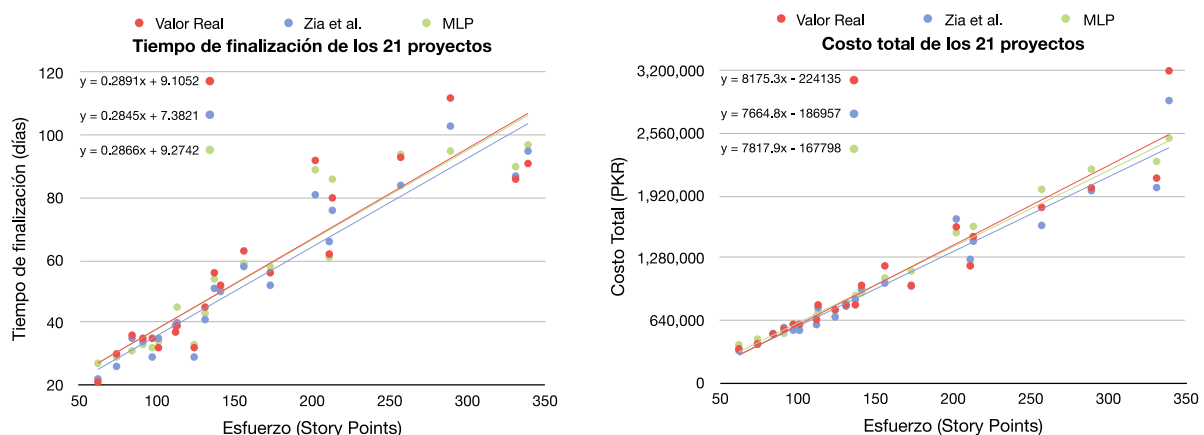


Figura 36: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y MLP en verde.

No.	ActualTime	MLP	ΔT (días)	ActualCost	MLP	ΔC (PKR)
1	63	59 $\pm$ 4	4	1,200,000	1,075,580 $\pm$ 179,308	124,420
2	92	89 $\pm$ 5	3	1,600,000	1,538,916 $\pm$ 111,615	61,084
3	56	58 $\pm$ 2	2	1,000,000	1,147,123 $\pm$ 73,832	147,123
4	86	90 $\pm$ 5	4	2,100,000	2,270,374 $\pm$ 201,146	170,374
5	32	33 $\pm$ 4	1	750,000	738,145 $\pm$ 99,578	11,855
6	91	97 $\pm$ 4	6	3,200,000	2,509,692 $\pm$ 293,731	690,308
7	35	32 $\pm$ 2	3	600,000	597,953 $\pm$ 45,569	2,047
8	93	94 $\pm$ 3	1	1,800,000	1,984,985 $\pm$ 111,671	184,985
9	36	31 $\pm$ 1	5	500,000	498,267 $\pm$ 31,296	1,733
10	62	61 $\pm$ 6	1	1,200,000	1,211,152 $\pm$ 168,359	11,152
11	45	43 $\pm$ 2	2	800,000	823,232 $\pm$ 113,289	23,232
12	37	39 $\pm$ 4	2	650,000	686,952 $\pm$ 140,407	36,952
13	32	34 $\pm$ 2	2	600,000	582,385 $\pm$ 44,350	17,615
14	30	29 $\pm$ 1	1	400,000	449,198 $\pm$ 39,727	49,198
15	21	27 $\pm$ 2	6	350,000	390,775 $\pm$ 58,769	40,775
16	112	95 $\pm$ 2	17	2,000,000	2,191,890 $\pm$ 164,173	191,890
17	39	45 $\pm$ 3	6	800,000	728,496 $\pm$ 111,648	71,504
18	52	51 $\pm$ 2	1	1,000,000	924,479 $\pm$ 59,210	75,521
19	80	86 $\pm$ 3	6	1,500,000	1,604,152 $\pm$ 86,458	104,152
20	56	54 $\pm$ 2	2	800,000	893,999 $\pm$ 67,829	93,999
21	35	33 $\pm$ 1	2	550,000	506,491 $\pm$ 52,225	43,509

Tabla 25: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y el algoritmo MLP.

7.4.2. Red Neuronal Recurrente

De todos los modelos de aprendizaje automático, la red neuronal recurrente tuvo la peor capacidad de generalización de datos al ejecutar la validación cruzada. Uno de los principales problemas es el reducido tamaño de los datos de entrada. El potencial de las redes recurrentes se explota cuando el conjunto de datos de entrada es grande, al no tener una serie temporal lo suficientemente grande, las ventanas de tiempo generadas durante la validación cruzada no contienen la suficiente información que permita a la red recurrente realizar una aproximación de la función esperada.

De las particiones creadas para la validación cruzada, en cinco de ellas el error no disminuye al orden de las milésimas y en menos de diez épocas la red comienza a caer en sobreentrenamiento. Como se observa en la Figura 37, es hasta la segunda mitad de las particiones de la validación que la red neuronal disminuye el error de manera significativa. El comportamiento del entrenamiento indica que a medida que la ventana de tiempo va incrementando, la red tiene mayor capacidad de asimilar los datos y mejorar las predicciones, la desventaja es que su capacidad de generalización no es buena durante las primeras particiones.

El bajo rendimiento de la red neuronal recurrente se refleja en el porcentaje de

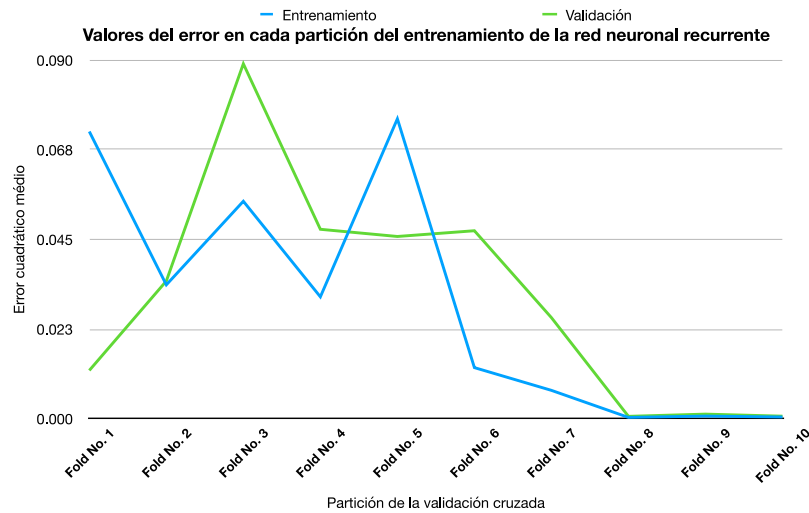


Figura 37: Desempeño de la red recurrente, en azul el error durante el entrenamiento y en verde el valor del error del conjunto seleccionado para validación.

precisión y el coeficiente de determinación de las estimaciones de tiempo y costo de la Figura 38. A la izquierda se muestra la gráfica del comportamiento del porcentaje de precisión, en azul los valores correspondientes a la precisión de tiempo y en verde la precisión para costo. Se observa que de las diez particiones de la validación cruzada, a partir de la sexta partición la red comienza a detectar un patrón en los datos y la precisión sube en ambos casos (tiempo y costo) a poco más del 80 %.

Con el coeficiente de determinación ocurre algo similar, durante las primeras particiones, cuando la ventana es pequeña la red presenta un bajo rendimiento, a partir de la sexta la red comienza a mejorar las estimaciones hasta alcanzar un coeficiente de determinación por encima de 0.90. En la gráfica de la derecha, en color verde se muestra el coeficiente de determinación de las estimaciones de costo y en amarillo los valores del coeficiente para las estimaciones de tiempo.

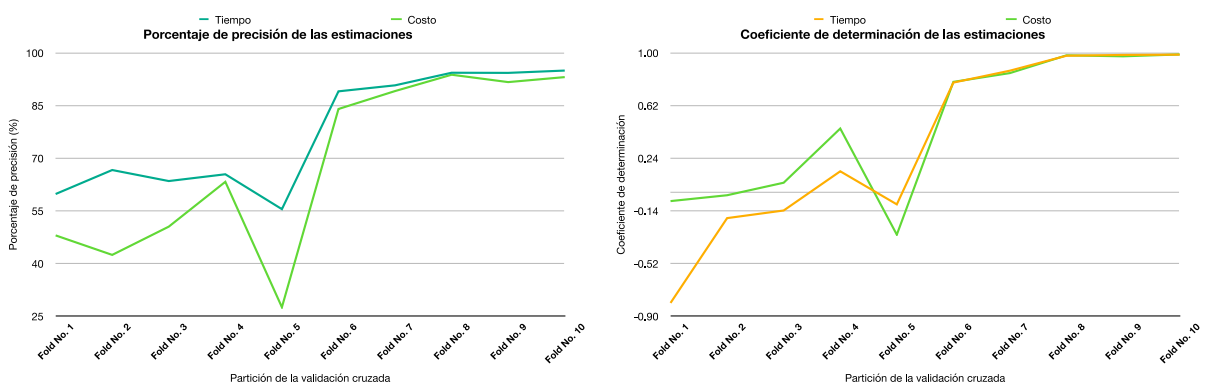


Figura 38: Resultados de la validación cruzada en las estimaciones de tiempo de la red recurrente.

Ninguno de los resultados de las primeras cinco particiones de la validación cruzada son útiles ya que el coeficiente de determinación es muy bajo. Para emitir los valores de las estimaciones finales se toma el promedio de los cinco mejores modelos entrenados, es decir a partir de la sexta partición, esto da como resultado las métricas presentadas en la Tabla 26.

Parámetro		Modelo Recurrent Neural Network (RNN)			
Configuración	Optimizador	Adam			
	Activación	Relu			
	Activación recurrente	Tanh			
	Capas Ocultas	2			
	Tipo de celda recurrente	Simple RNN			
	Neuronas p/Capa RNN	40			
	Neuronas p/Capa Dense	10			
	Tasa de aprendizaje	0.001			
	Loss	Error Cuadrático Medio (MSE)			
	Normalización de datos	MinMaxScaler en el rango (0,1)			
	Entradas (columnas)	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost	Effort, Vi	Eff, Vi SizeEffort, SizeTime, SizeCost
	Salida	Tiempo		Costo	
Prueba	Precisión (%)	91.47	95.27	89.49	93.44
	R ²	0.9441	0.9751	0.9547	0.9658
	MRE	0.0853	0.0473	0.1051	0.0656
	MSE	36.48	16.24	21,482,779,729	16,207,233,554
	RMSE	6.04	4.03	146,570	127,308
	Explained Variance	0.9477	0.9764	0.9584	0.9677
Regularización	Dropout recurrente	0.1			
	Ruido Gaussiano	stddev=0.00001			
	No. Capas de ruido	1			
Máximo de épocas en entrenamiento		900			

Tabla 26: Resultados de la evaluación de la red recurrente para las estimación de tiempo y costo.

A diferencia de la red MLP, la red RNN tiene una función de activación extra para el enlace recurrente, la función tanh se reserva para la recurrencia y la función *ReLU* para la activación tradicional. Cada capa recurrente se compone de 40 neuronas y la última capa oculta de tipo *Dense* se compone de diez neuronas. Los resultados comparan el uso de categorías, en ambos casos las estimaciones de tiempo y costo se beneficiaron de la clasificación de datos.

En el caso de las estimaciones de tiempo, los datos de prueba generaron una precisión del 95.27 % con un coeficiente $R^2 = 0.9751$. En las estimaciones de costo, la precisión aumentó a 93.44 % con un coeficiente de $R^2 = 0.9658$, en ambas estimaciones los errores se redujeron a casi la mitad de su valor.

Al comparar las estimaciones de la red RNN con la propuesta original [7] (Tabla 27), hubo una mejora en 15 de los 21 proyectos al estimar el tiempo de finalización. Al comparar las estimaciones de costo total, 12 de los 21 proyectos presentaron una mejora. A pesar de las dificultades del algoritmo para realizar las aproximaciones, si solo consideramos aquellos modelos con buen rendimiento, los resultados superan al algoritmo MLP. Sin embargo hay que recordar que si incluimos todos los modelos entrenados, el desempeño cae drásticamente, esto nos indica que la red recurrente no tiene una buena capacidad de generalización cuando las muestras son pequeñas.

En tres de los proyectos se tuvo una diferencia (ΔT) mayor a la semana laboral como se muestra en la Tabla 28. La desviación estándar de las estimaciones en tres proyectos es amplia, con 19, 14 y 13 días, el resto se mantiene por debajo de la semana laboral. Al comparar el costo actual con la estimaciones hechas por RNN, en 5 proyectos la

No.	Effort	ActualTime	Zia et al.	RNN	ActualCost	Zia et al.	RNN	Mejora Tiempo	Mejora Costo
1	156	63	58	66	1,200,000	1,023,207	1,308,695	SI	SI
2	202	92	81	87	1,600,000	1,680,664	1,490,040	SI	NO
3	173	56	52	47	1,000,000	992,270	870,208	NO	NO
4	331	86	87	74	2,100,000	2,002,767	1,591,492	NO	NO
5	124	32	29	34	750,000	676,081	781,927	SI	SI
6	339	91	95	92	3,200,000	2,895,133	3,222,765	SI	SI
7	97	35	29	35	600,000	540,114	604,019	SI	SI
8	257	93	84	95	1,800,000	1,614,079	1,852,733	SI	SI
9	84	36	35	35	500,000	507,265	494,307	NO	SI
10	211	62	66	64	1,200,000	1,267,180	1,236,064	SI	SI
11	131	45	41	43	800,000	786,732	737,623	SI	NO
12	112	37	39	36	650,000	597,143	599,896	SI	SI
13	101	32	35	32	600,000	538,495	573,062	SI	SI
14	74	30	26	30	400,000	394,546	443,372	SI	NO
15	62	21	22	22	350,000	330,561	353,935	NO	SI
16	289	112	103	110	2,000,000	1,971,485	1,946,445	SI	NO
17	113	39	40	37	800,000	770,857	747,143	NO	NO
18	141	52	50	52	1,000,000	961,866	948,179	SI	NO
19	213	80	76	79	1,500,000	1,453,032	1,498,412	SI	SI
20	137	56	51	54	800,000	854,348	790,688	SI	SI
21	91	35	34	42	550,000	567,484	683,761	NO	NO

Tabla 27: Comparativa de estimaciones hechas por el algoritmo RNN y la propuesta [7].

diferencia (ΔC) es mayor a cien mil rupias pero el resto se mantiene por debajo de las 65, 000 rupias.

Al representar los puntos en el plano (Figura 39), la dispersión de las estimaciones no dista demasiado de los valores reales, sin embargo a diferencia del resto de los algoritmos, en esta ocasión la recta de regresión de los valores reales en rojo y la recta de regresión del algoritmo en verde no son cercanas comparados con el resto de los algoritmos ya probados. La recta de RNN tiene una mayor similitud a la recta de regresión de las estimaciones de [7] que a la recta de los valores reales. En parte esto refleja los problemas presentados durante el entrenamiento. Una disminución en el valor de la pendiente implica que la tasa de crecimiento arrojará tiempos de finalización y costos menores a medida que crece el esfuerzo de un proyecto, pues la recta en color verde está por debajo de la recta en color rojo.

No.	ActualTime	RNN	ΔT (días)	ActualCost	RNN	ΔC (PKR)
1	63	66 $\pm$ 4	3	1,200,000	1,308,695 $\pm$ 182,395	108,695
2	92	87 $\pm$ 19	5	1,600,000	1,490,040 $\pm$ 466,853	109,960
3	56	47 $\pm$ 13	9	1,000,000	870,208 $\pm$ 180,025	129,792
4	86	74 $\pm$ 14	12	2,100,000	1,591,492 $\pm$ 463,821	508,508
5	32	34 $\pm$ 2	2	750,000	781,927 $\pm$ 42,933	31,927
6	91	92 $\pm$ 3	1	3,200,000	3,222,765 $\pm$ 71,400	22,765
7	35	35 $\pm$ 2	0	600,000	604,019 $\pm$ 46,281	4,019
8	93	95 $\pm$ 1	2	1,800,000	1,852,733 $\pm$ 27,998	52,733
9	36	35 $\pm$ 2	1	500,000	494,307 $\pm$ 72,035	5,693
10	62	64 $\pm$ 2	2	1,200,000	1,236,064 $\pm$ 12,487	36,064
11	45	43 $\pm$ 2	2	800,000	737,623 $\pm$ 40,989	62,377
12	37	36 $\pm$ 2	1	650,000	599,896 $\pm$ 58,472	50,104
13	32	32 $\pm$ 1	0	600,000	573,062 $\pm$ 71,353	26,938
14	30	30 $\pm$ 3	0	400,000	443,372 $\pm$ 59,798	43,372
15	21	22 $\pm$ 1	1	350,000	353,935 $\pm$ 48,741	3,935
16	112	110 $\pm$ 4	2	2,000,000	1,946,445 $\pm$ 84,065	53,555
17	39	37 $\pm$ 2	2	800,000	747,143 $\pm$ 52,482	52,857
18	52	52 $\pm$ 2	0	1,000,000	948,179 $\pm$ 65,350	51,821
19	80	79 $\pm$ 1	1	1,500,000	1,498,412 $\pm$ 23,717	1,588
20	56	54 $\pm$ 3	2	800,000	790,688 $\pm$ 79,973	9,312
21	35	42 $\pm$ 2	7	550,000	683,761 $\pm$ 24,908	133,761

Tabla 28: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y el algoritmo RNN.

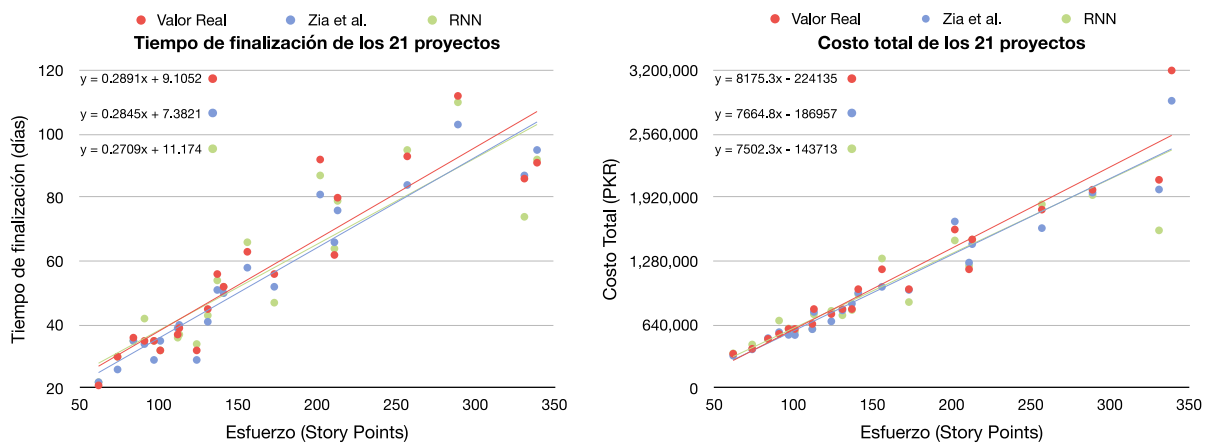


Figura 39: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y algoritmo RNN en verde.

7.4.3. Autoencoder

La última variante de redes neuronales a ejecutar en este trabajo es el *Autoencoder*, primero se ejecuta la arquitectura tradicional (Figura 40a), después se entrena un modelo compuesto solamente por el bloque de codificación (40b) y posteriormente a ese modelo se le agregan dos capas ocultas adicionales (40c).

Los resultados de los entrenamientos se agrupan en la Tabla 29. Cada bloque contiene dos estimadores, primero se muestra el modelo *Autoencoder* y el *Encoder*, debajo se agrupa el modelo *Encoder MLP* y el ensamble *AdaBoost*. Todos los casos respondieron

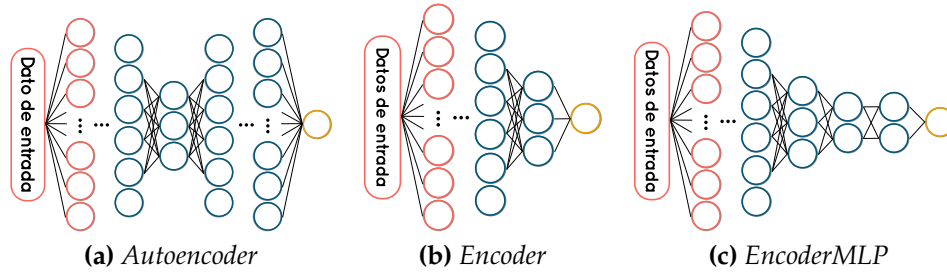


Figura 40: Aunque cambia el número de neuronas, las redes siguen siendo de tipo feedforward.

con una mejora en el coeficiente de determinación al aplicar categorías de clasificación, excepto el modelo *Encoder*, pues al realizar las estimaciones de costo, la precisión disminuyó de 91.93 % a 89.03 % y el coeficiente R^2 disminuyó de 0.9367 a 0.9143. La arquitectura de un *Encoder* con dos capas adicionales también presentó valores de coeficientes R^2 menores al resto de los algoritmos.

La diferencia que hay entre los árboles de decisión, las redes neuronales MLP y RNN, y las arquitecturas de *Autoencoder*, es que estos últimos se alimentan del conjunto de datos original de la Tabla 3 contemplando todas las características y no solo el esfuerzo y velocidad.

La Tabla comparativa 30 de mejora de proyectos agrupa las cuatro técnicas en una sola tabla, teniendo las columnas de esfuerzo, tiempo de finalización, estimación de [7], estimación de AE, *Encoder* y *Encoder MLP*, seguido del grupo de estimaciones de costo, y al final las condiciones que indican si hubo una mejora con respecto a las estimaciones de [7]. El resumen de las estimaciones de tiempo es el siguiente: *Autoencoder* mejoró en 14 de los 21 proyectos, *Encoder* en 16 de los 21 proyectos, y *Encoder MLP* mejoró en 15 de los 21 proyectos. Al revisar las estimaciones de costo *Autoencoder* mejoró en 10 de los 21 proyectos, *Encoder* en 6 de los 21 proyectos, y *Encoder MLP* mejoró en 9 de los 21 proyectos.

Los puntos representados en el plano de la Figura 41 correspondientes a las estimaciones de tiempo tiene una similitud con los valores reales y las rectas se aproximan con gran precisión a la recta de regresión lineal de los datos originales (en rojo), esto significa que hubo una buena aproximación a la función deseada. En cambio las rectas de regresión de las estimaciones de costo no tuvieron tal precisión, teniendo valores de pendiente menores a la pendiente de la recta obtenida por [7]. La única recta de regresión que se asemeja a la recta roja es la recta morada que corresponde al ensamble *AdaBoost* de codificadores.

Por último, la Tabla 31 agrupa los resultados del *Autoencoder*, *Encoder* y *Encoder MLP* con su desviación estándar y diferencia de días y rupias con respecto a los valores centrales de las estimaciones. Podemos destacar que la diferencia máxima de días es de siete para el *Autoencoder* y el *Encoder MLP*, y seis días para el *Encoder*.

Al revisar la diferencia de costos, las tres técnicas presentaron un proyecto en donde la diferencia rebasa el medio millón de rupias con desviaciones estándar por encima de las 400,000 rupias. Además, al contar el número de proyectos con diferencia de

Parámetro		Modelo Autoencoder				Encoder			
Datos	Entradas (columnas)	Dataset	Dataset + Categorías	Dataset	Dataset + Categorías	Dataset	Dataset + Categorías	Dataset	Dataset + Categorías
	Salida	Tiempo		Costo		Tiempo		Costo	
Entrenamiento	Error Cuadrático Medio (MSE)	0.0119	0.0018	0.0139	0.0055	0.0013	0.0010	0.0051	0.0110
	Raíz del MSE (RMSE)	0.0652	0.0349	0.0900	0.0629	0.0330	0.0293	0.0654	0.0865
	R ²	0.8362	0.9753	0.7656	0.9089	0.9828	0.9861	0.9127	0.8159
	Explained Variance	0.8571	0.9755	0.7929	0.9094	0.9830	0.9863	0.9130	0.8231
Validación	MSE	0.0230	0.0050	0.0176	0.0066	0.0030	0.0038	0.0083	0.0105
	RMSE	0.0869	0.0592	0.0911	0.0639	0.0459	0.0496	0.0712	0.0816
	R ²	0.6997	0.8875	0.4999	0.8095	0.9413	0.9109	0.7889	0.5684
	Explained Variance	0.8288	0.9238	0.7409	0.8409	0.9612	0.9208	0.8821	0.6718
Prueba	Precisión (%)	93.86	95.32	91.89	91.77	95.77	96.04	91.93	89.03
	R ²	0.9723	0.9888	0.9059	0.9487	0.9901	0.9915	0.9367	0.9143
	MRE	0.0614	0.0468	0.0811	0.0823	0.0423	0.0396	0.0807	0.1097
	MSE	18.00	7.29	44,565,746,289	24,329,148,553	6.43	5.52	29,982,176,142	40,618,680,183
	RMSE	4.24	2.70	211,106	155,978	2.54	2.35	173,154	201,541
	Explained Variance	0.9759	0.9890	0.9108	0.9490	0.9904	0.9917	0.9367	0.9164
Regularización	Dropout	No				No			
	Ruido Gaussiano	stddev=0.00001				stddev=0.00001			
	No. Capas de ruido	2				2			
Máximo de épocas en entrenamiento		3000				3000			
Promedio de épocas en entrenamiento		1204	1823	519	904	2014	2551	743	963

Parámetro		Encoder MLP				Encoder AdaBoost			
Datos	Entradas (columnas)	Dataset	Dataset + Categorías	Dataset	Dataset + Categorías	Dataset	Dataset + Categorías	Dataset	Dataset + Categorías
	Salida	Tiempo		Costo		Tiempo		Costo	
Entrenamiento	Error Cuadrático Medio (MSE)	0.0049	0.0013	0.0094	0.0100	0.0008	0.0006	0.0013	0.0010
	Raíz del MSE (RMSE)	0.0414	0.0320	0.0879	0.0870	0.0284	0.0244	0.0314	0.0304
	R ²	0.9364	0.9828	0.8308	0.8332	0.9891	0.9920	0.9777	0.9827
	Explained Variance	0.9394	0.9829	0.8366	0.8358	0.9892	0.9921	0.9779	0.9830
Validación	MSE	0.0056	0.0038	0.0171	0.0118	0.0014	0.0014	0.0053	0.0064
	RMSE	0.0542	0.0539	0.0871	0.0830	0.0368	0.0334	0.0487	0.0527
	R ²	0.9011	0.9038	0.7344	0.7424	0.9558	0.9514	0.8879	0.8824
	Explained Variance	0.9117	0.9237	0.8065	0.8038	0.9682	0.9615	0.9337	0.9179
Prueba	Precisión (%)	95.34	95.96	91.21	89.67	94.99	96.01	94.63	93.65
	R ²	0.9873	0.9901	0.8900	0.8973	0.9882	0.9928	0.9833	0.9856
	MRE	0.0466	0.0404	0.0879	0.1033	0.0501	0.0399	0.0537	0.0635
	MSE	8.24	6.43	52,142,108,589	48,671,189,953	7.67	4.67	7,910,094,406	6,802,189,692
	RMSE	2.87	2.54	228,346	220,615	2.77	2.16	88,939	82,475
	Explained Variance	0.9886	0.9904	0.8920	0.8992	0.9890	0.9937	0.9838	0.9859
Regularización	Dropout	No				No			
	Ruido Gaussiano	stddev=0.00001				stddev=0.00001			
	No. Capas de ruido	2				2			
Máximo de épocas en entrenamiento		3000				3000			
Promedio de épocas en entrenamiento		1694	2377	364	773	1518	1607	1623	1560

Tabla 29: Resultados obtenidos de la ejecución del autoencoder y sus variantes.

No.	Effort	ActualTime	Zia et al.	AE	Encoder	Encoder MLP	ActualCost	Zia et al.	AE	Encoder	Encoder MLP	AE Mejora Tiempo	Encoder Mejora Tiempo	Enc MLP Mejora Tiempo	AE Mejora Costo	Encoder Mejora Tiempo	Enc MLP Mejora Tiempo
1	156	63	58	61	61	62	1,200,000	1,023,207	1,133,868	1,054,427	1,119,214	SI	SI	SI	SI	SI	SI
2	202	92	81	89	90	90	1,600,000	1,680,664	1,586,210	1,387,014	1,534,462	SI	SI	SI	SI	SI	SI
3	173	56	52	56	56	56	1,000,000	992,270	1,074,890	1,140,797	1,098,829	SI	SI	SI	NO	NO	NO
4	331	86	87	87	86	87	2,100,000	2,002,767	2,167,721	2,081,459	2,031,269	NO	SI	NO	SI	SI	SI
5	124	32	29	34	32	33	750,000	676,081	836,084	885,034	908,089	SI	SI	SI	NO	NO	NO
6	339	91	95	92	92	92	3,200,000	2,895,133	2,563,786	2,402,292	2,264,046	SI	SI	SI	NO	NO	NO
7	97	35	29	34	34	34	600,000	540,114	596,071	582,851	605,289	SI	SI	SI	SI	SI	SI
8	257	93	84	92	92	92	1,800,000	1,614,079	1,953,939	1,955,553	1,900,456	SI	SI	SI	SI	SI	SI
9	84	36	35	33	34	33	500,000	507,265	464,804	401,279	431,378	NO	NO	NO	NO	NO	NO
10	211	62	66	61	64	62	1,200,000	1,267,180	1,223,387	1,239,449	1,252,054	SI	SI	SI	SI	SI	SI
11	131	45	41	43	42	42	800,000	786,732	881,315	879,655	921,950	SI	SI	SI	NO	NO	NO
12	112	37	39	38	38	38	650,000	597,143	672,834	641,172	654,052	SI	SI	SI	SI	SI	SI
13	101	32	35	34	34	34	600,000	538,495	544,127	533,093	539,805	SI	SI	SI	SI	NO	SI
14	74	30	26	28	27	27	400,000	394,546	477,031	466,312	484,262	SI	SI	SI	NO	NO	NO
15	62	21	22	25	24	24	350,000	330,561	447,635	436,837	459,754	NO	NO	NO	NO	NO	NO
16	289	112	103	107	108	107	2,000,000	1,971,485	1,921,431	1,861,980	1,884,426	SI	SI	SI	NO	NO	NO
17	113	39	40	42	41	40	800,000	770,857	794,364	849,369	818,139	NO	NO	NO	SI	NO	SI
18	141	52	50	50	51	51	1,000,000	961,866	884,599	945,467	901,653	NO	SI	SI	NO	NO	NO
19	213	80	76	87	86	87	1,500,000	1,453,032	1,595,441	1,557,343	1,608,756	NO	NO	NO	NO	NO	NO
20	137	56	51	53	53	53	800,000	854,348	847,412	953,783	867,810	SI	SI	SI	SI	NO	NO
21	91	35	34	33	33	33	550,000	567,484	488,937	492,499	496,194	NO	NO	NO	NO	NO	NO

Tabla 30: Comparativa de estimaciones hechas por el algoritmo AE y la propuesta [7].

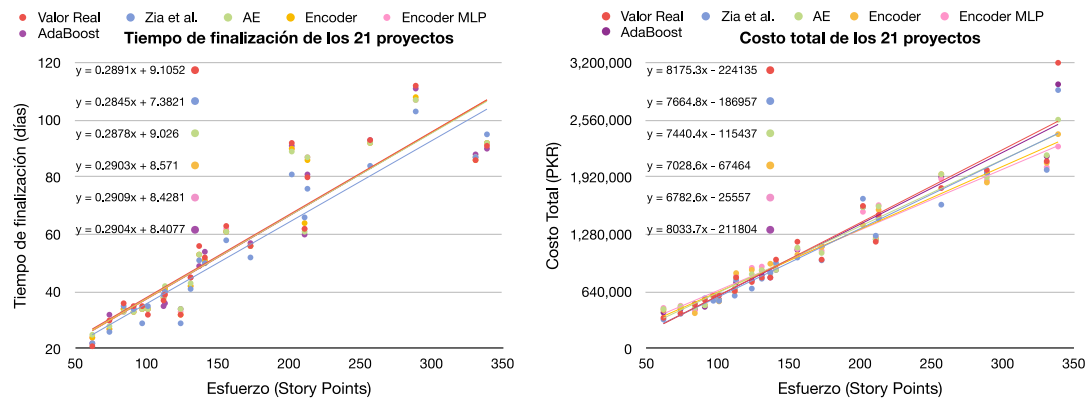


Figura 41: Tiempo de finalización real en rojo, estimaciones de [7] en azul, y AE en verde.

No.	ActualTime	AE	Encoder	Encoder MLP	AE ΔT (días)	Encoder ΔT (días)	Enc MLP ΔT (días)	ActualCost	AE	Encoder	Encoder MLP	AE ΔC (PKR)	Encoder ΔC (PKR)	Enc MLP ΔC (PKR)
1	63	61 ± 2	61 ± 1	62 ± 2	2	2	1	1,200,000	1,133,868 ± 112,980	1,054,427 ± 220,471	1,119,214 ± 157,094	66,132	145,573	80,786
2	92	89 ± 5	90 ± 2	90 ± 3	3	2	2	1,600,000	1,586,210 ± 180,814	1,387,014 ± 403,215	1,534,462 ± 270,202	13,790	212,986	65,538
3	56	56 ± 2	56 ± 2	56 ± 2	0	0	0	1,000,000	1,074,890 ± 111,488	1,140,797 ± 157,784	1,098,829 ± 102,966	74,890	140,797	98,829
4	86	87 ± 3	86 ± 2	87 ± 2	1	0	1	2,100,000	2,167,721 ± 154,210	2,081,459 ± 350,879	2,031,269 ± 275,543	67,721	18,541	68,731
5	32	34 ± 5	32 ± 1	33 ± 2	2	0	1	750,000	836,084 ± 211,412	885,034 ± 293,599	908,089 ± 260,235	86,084	135,034	158,089
6	91	92 ± 2	92 ± 2	92 ± 2	1	1	1	3,200,000	2,563,786 ± 426,091	2,402,292 ± 513,780	2,264,046 ± 499,645	636,214	797,708	935,954
7	35	34 ± 2	34 ± 2	34 ± 2	1	1	1	600,000	596,071 ± 48,936	582,851 ± 137,163	605,289 ± 76,158	3,929	17,149	5,289
8	93	92 ± 2	92 ± 1	92 ± 1	1	1	1	1,800,000	1,953,939 ± 154,954	1,955,553 ± 205,720	1,900,456 ± 206,574	153,939	155,553	100,456
9	36	33 ± 3	34 ± 2	33 ± 3	3	2	3	500,000	464,804 ± 94,346	401,279 ± 181,328	431,378 ± 109,572	35,196	98,721	68,622
10	62	61 ± 2	64 ± 5	62 ± 2	1	2	0	1,200,000	1,223,387 ± 151,917	1,239,449 ± 165,695	1,252,054 ± 171,859	23,387	39,449	52,054
11	45	43 ± 3	42 ± 2	42 ± 2	2	3	3	800,000	881,315 ± 104,705	879,655 ± 147,162	921,950 ± 120,770	81,315	79,655	121,950
12	37	38 ± 2	38 ± 1	38 ± 2	1	1	1	650,000	672,834 ± 76,327	641,172 ± 168,237	654,052 ± 156,297	22,834	8,828	4,052
13	32	34 ± 1	34 ± 2	34 ± 2	2	2	2	600,000	544,127 ± 33,851	533,093 ± 80,745	539,805 ± 61,025	55,873	66,907	60,195
14	30	28 ± 2	27 ± 2	27 ± 2	2	3	3	400,000	477,031 ± 53,506	466,312 ± 80,911	484,262 ± 70,509	77,031	66,312	84,262
15	21	25 ± 3	24 ± 3	24 ± 3	4	3	3	350,000	447,635 ± 79,169	436,837 ± 100,358	459,754 ± 83,552	97,635	86,837	109,754
16	112	107 ± 8	108 ± 6	107 ± 7	5	4	5	2,000,000	1,921,431 ± 128,331	1,861,980 ± 247,576	1,884,426 ± 178,642	78,569	138,020	115,574
17	39	42 ± 4	41 ± 3	40 ± 4	3	2	1	800,000	794,364 ± 91,426	849,369 ± 172,941	818,139 ± 118,861	5,636	49,369	18,139
18	52	50 ± 1	51 ± 1	51 ± 1	2	1	1	1,000,000	884,599 ± 61,591	945,467 ± 123,619	901,653 ± 92,751	115,401	54,533	98,347
19	80	87 ± 4	86 ± 3	87 ± 3	7	6	7	1,500,000	1,595,441 ± 170,531	1,557,343 ± 189,772	1,608,756 ± 156,203	95,441	57,343	108,756
20	56	53 ± 2	53 ± 1	53 ± 2	3	3	3	800,000	847,412 ± 85,900	953,783 ± 219,773	867,810 ± 125,563	47,412	153,783	67,810
21	35	33 ± 1	33 ± 1	33 ± 2	2	2	2	550,000	488,937 ± 45,119	492,499 ± 83,430	496,194 ± 69,210	61,063	57,501	53,806

Tabla 31: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y las diferentes implementaciones de Autoencoders.

costo mayor a cien mil rupias, observamos que *Autoencoder* solo tuvo tres proyectos, *Encoder* tuvo ocho proyectos, y *Encoder MLP* tuvo siete proyectos.

El ensamble de redes neuronales tiene la peculiaridad de que las estimaciones de tiempo tienen buena precisión y en 16 de los 21 proyectos hubo una mejoría como se observa en la Tabla 32. Solo en 12 de las estimaciones de costo hubo una mejora comparado con las predicciones hechas por [7]. Sin embargo, la desviación estándar de las estimaciones de tiempo de todos los proyectos se encuentra por debajo de la semana laboral como se observa en la Tabla 33, y la diferencia de días con respecto al valor real también se mantiene por debajo de la semana laboral.

Al comparar las estimaciones de costo, solo cuatro proyectos tuvieron una diferencia mayor a las cien mil rupias respecto del valor real, y la desviación estándar superó las cien mil rupias en dos de los proyectos. Esto nos indica que la red tuvo una capacidad de aproximación similar en la mayoría de los modelos entrenados en la validación cruzada.

Con este último análisis empleando un algoritmo basado en redes neuronales se

No.	Effort	ActualTime	Zia et al.	AE AdaBoost	ActualCost	Zia et al.	AE AdaBoost	AE Mejora Tiempo	AE Mejora Costo
1	156	63	58	61	1,200,000	1,023,207	1,108,169	SI	SI
2	202	92	81	90	1,600,000	1,680,664	1,587,128	SI	SI
3	173	56	52	55	1,000,000	992,270	1,146,366	SI	NO
4	331	86	87	86	2,100,000	2,002,767	2,154,676	SI	SI
5	124	32	29	32	750,000	676,081	770,366	SI	SI
6	339	91	95	92	3,200,000	2,895,133	2,959,692	SI	SI
7	97	35	29	33	600,000	540,114	602,755	SI	SI
8	257	93	84	92	1,800,000	1,614,079	1,924,081	SI	SI
9	84	36	35	34	500,000	507,265	444,365	NO	NO
10	211	62	66	62	1,200,000	1,267,180	1,225,340	SI	SI
11	131	45	41	43	800,000	786,732	828,979	SI	NO
12	112	37	39	36	650,000	597,143	671,808	SI	SI
13	101	32	35	34	600,000	538,495	552,714	SI	SI
14	74	30	26	27	400,000	394,546	449,644	SI	NO
15	62	21	22	24	350,000	330,561	407,837	NO	NO
16	289	112	103	109	2,000,000	1,971,485	1,964,096	SI	NO
17	113	39	40	42	800,000	770,857	744,123	NO	NO
18	141	52	50	51	1,000,000	961,866	885,697	SI	NO
19	213	80	76	83	1,500,000	1,453,032	1,472,542	SI	SI
20	137	56	51	51	800,000	854,348	798,986	NO	SI
21	91	35	34	33	550,000	567,484	472,570	NO	NO

Tabla 32: Comparativa de estimaciones hechas por el algoritmo AdaBoost (Encoder) y la propuesta [7].

No.	ActualTime	AE	AE ΔT (días)	ActualCost	AE	AE ΔC (PKR)
1	63	61 $\pm$ 1	2	1,200,000	1,108,169 $\pm$ 41,771	91,831
2	92	90 $\pm$ 1	2	1,600,000	1,587,128 $\pm$ 38,824	12,872
3	56	55 $\pm$ 1	1	1,000,000	1,146,366 $\pm$ 87,067	146,366
4	86	86 $\pm$ 1	0	2,100,000	2,154,676 $\pm$ 74,023	54,676
5	32	32 $\pm$ 1	0	750,000	770,366 $\pm$ 63,391	20,366
6	91	92 $\pm$ 1	1	3,200,000	2,959,692 $\pm$ 254,036	240,308
7	35	33 $\pm$ 2	2	600,000	602,755 $\pm$ 28,447	2,755
8	93	92 $\pm$ 1	1	1,800,000	1,924,081 $\pm$ 106,807	124,081
9	36	34 $\pm$ 1	2	500,000	444,365 $\pm$ 66,229	55,635
10	62	62 $\pm$ 1	0	1,200,000	1,225,340 $\pm$ 76,269	25,340
11	45	43 $\pm$ 1	2	800,000	828,979 $\pm$ 42,494	28,979
12	37	36 $\pm$ 1	1	650,000	671,808 $\pm$ 35,695	21,808
13	32	34 $\pm$ 1	2	600,000	552,714 $\pm$ 18,170	47,286
14	30	27 $\pm$ 1	3	400,000	449,644 $\pm$ 40,517	49,644
15	21	24 $\pm$ 2	3	350,000	407,837 $\pm$ 55,182	57,837
16	112	109 $\pm$ 2	3	2,000,000	1,964,096 $\pm$ 66,986	35,904
17	39	42 $\pm$ 2	3	800,000	744,123 $\pm$ 41,887	55,877
18	52	51 $\pm$ 1	1	1,000,000	885,697 $\pm$ 41,299	114,303
19	80	83 $\pm$ 2	3	1,500,000	1,472,542 $\pm$ 89,877	27,458
20	56	51 $\pm$ 2	5	800,000	798,986 $\pm$ 53,141	1,014
21	35	33 $\pm$ 1	2	550,000	472,570 $\pm$ 29,258	77,430

Tabla 33: Estimaciones de tiempo y costo con sus desviaciones estándar y diferencias con respecto a los valores reales y las estimaciones por el ensamble AdaBoost Encoder.

concluye la fase de evaluación experimental de este trabajo, y una vez que se ha recopilado la información de todas las técnicas aplicadas, se procede a emitir los valores definitivos de tiempo de finalización y costo total que se reportan como estimaciones finales.

7.5. Comparativa

El resumen de todas las técnicas aplicadas en este trabajo se muestra en la Tabla 34. Cada proyecto tuvo un total de diez estimaciones, una predicción por cada técnica, el promedio de todas las predicciones da como resultado un ensamble tipo *bagging* que da el valor final del proyecto. En el caso del tiempo de finalización, el promedio final tuvo una mejora en el porcentaje de precisión y el coeficiente de determinación. El error relativo se redujo a 0.0278 como se observa en la última columna, mientras que la raíz del error cuadrático medio disminuyó su valor a dos días.

En el caso de las estimaciones de costo, el ensamble final que agrupa todos los algoritmos se mantuvo por debajo de las técnicas RF y DT, con una precisión de la predicción de 95.06 %, y un coeficiente de 0.9813. Una de las ventajas del ensamble compuesto por las diez técnicas que se aplicaron en este trabajo es que se proporciona estabilidad en las predicciones y se logra reducir de manera significativa el error relativo y la raíz del error cuadrático medio como se muestra en la Tabla 34.

	Algoritmo	AE	Encoder	Encoder + MLP	AdaBoost Enc	KNN	MLP	RNN	RF	AdaBoost	DT	Ensamble
Datos	Entradas	Dataset + Categorías				Effort, Vi, SizeEffort, SizeTime, SizeCost						
	Salidas	Tiempo										
Resultados	Predicción(%)	95.32	96.04	95.96	96.01	94.7	93.14	95.27	96.92	96.44	96.57	97.22
	R²	0.9888	0.9915	0.9901	0.9928	0.9840	0.9614	0.9751	0.9869	0.9926	0.9910	0.9937
	Error relativo MRE	0.0468	0.0396	0.0404	0.0399	0.0530	0.0686	0.0473	0.0308	0.0356	0.0343	0.0278
	RMSE	2.70	2.35	2.54	2.16	3.23	5.00	4.03	2.92	2.19	2.42	2.02
	Explained Variance	0.9890	0.9917	0.9904	0.9937	0.9840	0.9615	0.9764	0.9869	0.9930	0.9914	0.9939

	Algoritmo	AE	Encoder	Encoder + MLP	AdaBoost Enc	KNN	MLP	RNN	RF	AdaBoost	DT	Ensamble
Datos	Entradas	Dataset + Categorías				Effort, Vi, SizeEffort, SizeTime, SizeCost						
	Salidas	Costo										
Resultados	Predicción(%)	91.77	89.03	89.67	93.65	94.54	92.52	93.44	97.03	95.19	97.33	95.06
	R²	0.9487	0.9143	0.8973	0.9856	0.9860	0.9341	0.9658	0.9885	0.9923	0.9944	0.9813
	Error relativo MRE	0.0823	0.1097	0.1033	0.0635	0.0546	0.0748	0.0656	0.0297	0.0481	0.0267	0.0494
	RMSE	155,978	201,541	220,615	82,475	81,388	176,702	127,308	73,781	60,432	51,357	94,195
	Explained Variance	0.9490	0.9164	0.8992	0.9859	0.9861	0.9341	0.9677	0.9890	0.9930	0.9953	0.9819

Tabla 34: Comparativa de los tres algoritmos de aprendizaje automático, sus variantes y el ensamble final.

Al tomar la diferencia de los valores predichos con los datos reales se observa que en términos de tiempo ninguno de los proyectos supera la diferencia de dos semanas, solo el proyecto con el tiempo de finalización más grande tuvo una diferencia de 7 días (una semana con dos días). Respecto al costo de los proyectos, solo el proyecto de mayor costo tuvo una diferencia de 273,310 rupias, el resto se encuentra por debajo de las 70,000 rupias.

En la Tabla 35 se muestran los valores reales, las estimaciones de [7] y los valores finales del ensamble compuesto por las diez técnicas. En este caso, de los 21 proyectos, 17 presentaron una mejora en las estimaciones de tiempo, y solo 11 mejoraron la estimación de costo. Las diferencias de tiempo y costo de los valores centrales de las estimaciones con respecto a los datos reales se muestran en la Tabla 36.

No.	Effort	ActualTime	Zia et al.	Ensamble	ActualCost	Zia et al.	Ensamble	Mejora Tiempo	Mejora Costo
1	156	63	58	61	1,200,000	1,023,207	1,134,402	SI	SI
2	202	92	81	90	1,600,000	1,680,664	1,550,870	SI	SI
3	173	56	52	55	1,000,000	992,270	1,058,229	SI	NO
4	331	86	87	86	2,100,000	2,002,767	2,105,883	SI	SI
5	124	32	29	34	750,000	676,081	798,671	SI	SI
6	339	91	95	92	3,200,000	2,895,133	2,820,969	SI	NO
7	97	35	29	34	600,000	540,114	596,087	SI	SI
8	257	93	84	93	1,800,000	1,614,079	1,879,159	SI	SI
9	84	36	35	33	500,000	507,265	462,572	NO	NO
10	211	62	66	62	1,200,000	1,267,180	1,208,710	SI	SI
11	131	45	41	44	800,000	786,732	826,147	SI	NO
12	112	37	39	37	650,000	597,143	648,148	SI	SI
13	101	32	35	33	600,000	538,495	562,992	SI	SI
14	74	30	26	29	400,000	394,546	438,873	SI	NO
15	62	21	22	23	350,000	330,561	400,007	NO	NO
16	289	112	103	108	2,000,000	1,971,485	1,969,439	SI	NO
17	113	39	40	40	800,000	770,857	769,604	NO	NO
18	141	52	50	52	1,000,000	961,866	899,059	SI	NO
19	213	80	76	85	1,500,000	1,453,032	1,560,052	NO	NO
20	137	56	51	52	800,000	854,348	835,663	SI	SI
21	91	35	34	35	550,000	567,484	532,678	SI	SI

Tabla 35: Comparación de las estimaciones finales de tiempo y costo generadas por el ensamble con las predicciones de [7].

Se observa que la desviación estándar del tiempo de finalización de la columna de ensamble no supera la semana laboral y la diferencia de días en un solo proyecto es de cinco días, el resto permanece por debajo de este valor. Al revisar las estimaciones de costo, las diferencias se mantienen por debajo de las cien mil rupias con excepción de dos proyectos. La desviación de las estimaciones de los modelos entrenados solo excedió las cien mil rupias en dos de los proyectos. Los resultados obtenidos son satisfactorios ya que al reducir las desviaciones, los modelos entrenados tienden a predecir los valores en un rango acotado lo cuál implica que hay una similitud en la aproximación de la función deseada.

Tomando en cuenta el número de proyectos donde mejoraron las estimaciones respecto a la propuesta original [7], en la Tabla 37 se muestran todas las técnicas y los casos donde mejoraron las estimaciones de tiempo y costo. El mejor desempeño lo obtuvo DT, seguido del ensamble *AdaBoost* compuesto por *Encoders*. Considerando

No.	ActualTime	Ensamble	ΔT (días)	ActualCost	Ensamble	ΔC (PKR)
1	63	61 $\pm$ 2	2	1,200,000	1,134,402 $\pm$ 64,982	65,598
2	92	90 $\pm$ 2	2	1,600,000	1,550,870 $\pm$ 66,464	49,130
3	56	55 $\pm$ 3	1	1,000,000	1,058,229 $\pm$ 83,176	58,229
4	86	86 $\pm$ 4	0	2,100,000	2,105,883 $\pm$ 193,582	5,883
5	32	34 $\pm$ 1	2	750,000	798,671 $\pm$ 54,714	48,671
6	91	92 $\pm$ 2	1	3,200,000	2,820,969 $\pm$ 332,367	379,031
7	35	34 $\pm$ 1	1	600,000	596,087 $\pm$ 10,054	3,913
8	93	93 $\pm$ 1	0	1,800,000	1,879,159 $\pm$ 69,277	79,159
9	36	33 $\pm$ 2	3	500,000	462,572 $\pm$ 40,198	37,429
10	62	62 $\pm$ 1	0	1,200,000	1,208,710 $\pm$ 37,665	8,710
11	45	44 $\pm$ 1	2	800,000	826,147 $\pm$ 51,957	26,147
12	37	37 $\pm$ 1	0	650,000	648,148 $\pm$ 23,798	1,852
13	32	33 $\pm$ 1	1	600,000	562,992 $\pm$ 33,145	37,008
14	30	29 $\pm$ 2	1	400,000	438,873 $\pm$ 31,619	38,873
15	21	23 $\pm$ 2	2	350,000	400,007 $\pm$ 38,565	50,007
16	112	108 $\pm$ 4	4	2,000,000	1,969,439 $\pm$ 90,768	30,561
17	39	40 $\pm$ 3	1	800,000	769,604 $\pm$ 65,193	30,396
18	52	52 $\pm$ 1	0	1,000,000	899,059 $\pm$ 50,881	100,942
19	80	85 $\pm$ 3	5	1,500,000	1,560,052 $\pm$ 47,913	60,052
20	56	52 $\pm$ 3	4	800,000	835,663 $\pm$ 52,009	35,663
21	35	35 $\pm$ 3	1	550,000	532,678 $\pm$ 57,777	17,322

Tabla 36: Diferencias de los valores predichos por el ensamble y los valores reales de cada proyecto.

que cada modelo tiene un tiempo de ejecución y una cantidad de recursos utilizada, el algoritmo *KNN* ofreció resultados competitivos al ser la técnica más sencilla y de ejecución inmediata. Después sigue la técnica *RF* con una mejora de 14 proyectos en las estimaciones de tiempo y costo. La cantidad de proyectos mejor estimados ocurrió en el tiempo de finalización, esto muestra que existe una mejor relación entre los datos de entrada y la variable dependiente de tiempo medida en días. Una de las desventajas respecto a las estimaciones de costo es que la información proporcionada a los algoritmos necesita de más características esenciales para aumentar la precisión de las estimaciones.

Las investigaciones del estado del arte no proporcionan mayor detalle respecto de las estimaciones emitidas como se hizo en este trabajo, por lo que solo un par de métricas son útiles para comparar los resultados de las distintas investigaciones. En la Tabla 38 se muestra la comparación del porcentaje de precisión, el coeficiente de determinación y el error relativo obtenido en esta investigación con los datos reportados en el estado del arte.

Técnica	¿En cuántos proyectos mejoró la estimación de tiempo?	¿En cuántos proyectos mejoró la estimación de costo?
KNN	15	12
DT	15	16
RF	14	14
AdaBoost DT	13	13
MLP	12	9
RNN	15	12
AE	14	10
Encoder	16	6
EncoderMLP	15	9
AdaBoost AE	16	12
Ensamble	17	11

Tabla 37: Comparativa de las técnicas aplicadas, el mejor desempeño lo obtuvo el algoritmo DT, seguido de AdaBoost AE, RF y KNN.

Autor	Estimación	Precisión (%)	R ²	MMRE
Zia et al.	Tiempo	57.14	No incluido	0.0719
	Costo	61.90	No incluido	0.0576
Aditi Panda et al.	Tiempo	94.76	0.9303	0.1486
SM Satapathy et al.	Tiempo	95.24	No incluido	0.1632
Ch Prasada Rao et al.	Tiempo	76.19	No incluido	0.0483
	Costo	76.19	No incluido	0.0276
Proyecto de Investigación (Ensamble)	Tiempo	97.22	0.9937	0.0278
	Costo	95.06	0.9813	0.0494
Proyecto de Investigación (DT)	Tiempo	96.57	0.9910	0.0343
	Costo	97.33	0.9944	0.0267

Tabla 38: Tabla comparativa de porcentaje de precisión del trabajo de investigación con los reportado en el estado del arte.

7.6. Amenazas a la validez

Como ya se ha reportado en investigaciones anteriores [1], una deficiencia por parte del conjunto de datos es que no hay información adicional sobre el tipo de proyectos que se toman para el estudio, la composición del equipo, o la clasificación de los

proyectos por compañía. Tampoco existe un mayor detalle de la pila de producto que permita organizar las historias que la componen. La ausencia de este tipo de detalles nos limita a utilizar solamente la información que se proporciona para realizar las estimaciones de los proyectos.

Debido al tamaño del conjunto de datos, no se cuenta con suficientes registros para obtener resultados confiables al aplicar validación cruzada, aunque esto se puede contrarrestar con la adición de ruido en los datos para generar conjuntos artificiales como se mencionó en secciones anteriores. En el caso de la red recurrente la serie de tiempo generada es muy pequeña para explotar el potencial de las redes recurrentes en donde es recomendable utilizar conjuntos de datos grandes.

La velocidad es una métrica que no siempre está disponible, por ejemplo en casos donde el equipo de trabajo es nuevo o hubo cambios en el personal de algún grupo ya existente, en estos casos es indispensable contar con más métricas u otros puntos de referencia para realizar las estimaciones. También es necesario contar con los riesgos asociados a los proyectos ya que la información utilizada no es suficiente para dar buenas estimaciones del costo relacionado al desarrollo de los proyectos. Al contar con una matriz de riesgos es posible clasificar cada proyecto de acuerdo al tipo y tamaño de riesgo lo que permitiría mejorar las estimaciones de los proyectos de software.

8. Conclusiones

Uno de los fundamentos de la ingeniería de software, consiste en la gestión y planificación de proyectos a través de métodos y paradigmas que permitan entregar resultados confiables dentro del presupuesto establecido y con los recursos disponibles. Hay tres elementos básicos a considerar cuando se planifica un proyecto de software: recursos, alcance y fecha de entrega. Al completar los proyectos en tiempo y forma, manteniendo los costos generales dentro del presupuesto y cumpliendo con las expectativas del cliente, se asegura la calidad del desarrollo [55].

El proceso de estimación de esfuerzo de un proyecto de software requiere calcular el tiempo necesario para completar las actividades y tareas con los recursos contemplados, así como presupuestar, gestionar y controlar los costos asociados atendiendo el nivel de calidad deseado.

La literatura clasifica las técnicas de estimación de esfuerzo en ingeniería de software de acuerdo a: el conjunto de datos disponible, los modelos de estimación específicos aplicados, o por la combinación de técnicas empleadas. En la Tabla 1 presentada en el estado del arte se agrupa una clasificación general de las distintas técnicas de estimación de esfuerzo existentes, cada autor puede nombrar una técnica con distintos nombres pero los fundamentos teóricos son los mismos. Así, resulta conveniente simplificar en cuatro categorías principales: Juicio de expertos, técnicas basadas en analogía o en aprendizaje, modelos algorítmicos y técnicas compuestas. Este proyecto se basó en un método compuesto por un modelo basado en regresión lineal, que incorpora un modelo de aprendizaje supervisado de tipo ensamble y categorías de clasificación que permitieron aumentar el porcentaje de precisión de las estimaciones de tiempo de finalización y costo total de los proyectos de software.

Así, podemos responder a la primera pregunta de investigación **RQ-1**: ¿Cuáles son los métodos de aprendizaje automático que se han utilizado en desarrollo de software ágil para estimar esfuerzo y costo de un proyecto? Dos de los métodos más utilizados en estimación de esfuerzo son los árboles de decisión y las redes neuronales, ambas técnicas tienen la característica de adaptarse a problemas en contextos no lineales, y a conjuntos de muestras de alta dimensión. En particular las técnicas que dan el mejor porcentaje de precisión son *Random Forest*, *Stochastic Gradient Boosting*, Redes neuronales de tipo *Cascade-Correlation*, *Multilayer Perceptron* con arquitectura de *Encoder* y el método *AdaBoost* aplicado a árboles de decisiones y redes neuronales. Otras técnicas aplicadas son las máquinas de soporte vectorial, lógica difusa, *Principal Component Analysis*, redes bayesianas, el algoritmo de los k vecinos más cercanos y redes neuronales de aprendizaje profundo como son las redes LSTM.

En este estudio el esfuerzo medido en puntos de historia de usuario y la velocidad inicial de los proyectos de software establecen dos características esenciales utilizadas para realizar estimaciones de tiempo de finalización y costo total de un proyecto desarrollado con métodos ágiles. Se construyó un ensamble de diez técnicas: el algoritmo de los k vecinos más cercanos (*KNN*), árboles de decisión (*DT*), bosques aleatorios (*RF*), ensamble *AdaBoost* de árboles de decisión, red neuronal de tipo perceptrón multicapa (*MLP*), red neuronal recurrente simple (*RNN*), red neuronal de tipo *Autoencoder* (*AE*), y

dos variaciones de la misma (*Encoder* y *Encoder MLP*), así como un ensamble *AdaBoost* de redes tipo *Encoder*.

Dentro de los principales hallazgos encontrados en esta investigación podemos mencionar que:

1. La clasificación de proyectos mejoró la precisión de la predicción y también aumentó el coeficiente de determinación en todos los algoritmos aplicados
2. La adición de ruido al conjunto de entrenamiento permitió a los algoritmos incrementar su capacidad de generalización y predicción.
3. La validación cruzada permitió evaluar la capacidad de generalización de los algoritmos mediante la comparación de los errores en entrenamiento, validación y prueba; el único algoritmo que no tuvo un buen desempeño en todas la particiones de la validación fue la red neuronal recurrente debido al tamaño reducido de la serie temporal.
4. Al aplicar un ensamble de técnicas de aprendizaje automático, se mejoró la precisión de las estimaciones, reduciendo la dispersión de los datos y las métricas de evaluación.
5. Como lo muestran las tablas comparativas de las predicciones emitidas, la desviación estándar de las estimaciones de tiempo no superan la semana laboral en la mayoría de los proyectos, y las estimaciones de costo presentan una desviación estándar por debajo de las 100 mil rupias.
6. El algoritmo KNN funciona satisfactoriamente cuando el conjunto de datos no es grande y la dimensión de los datos de entrada es pequeña. Al aumentar el número de características la precisión de la predicción y el coeficiente de determinación disminuyen. En este caso el algoritmo no presentó una baja en el rendimiento ya que la dimensión de los datos nunca fue mayor a cinco.
7. De las distintas propuestas los mejores algoritmos fueron los árboles de decisión, seguido del ensamble tipo *AdaBoost* de *Encoders*. El algoritmo KNN también fue una de las mejores opciones debido al corto tiempo de ejecución y consumo de recursos, así como altos valores de porcentaje de precisión y coeficiente de determinación.
8. El algoritmo DT tuvo el mejor desempeño al mejorar 15 de las estimaciones de tiempo y 16 de las estimaciones de costo.

Todos los algoritmos con excepción de la red RNN presentaron una adaptabilidad al contexto del problema, su modelado fue flexible, y ofrecieron mejores métricas que el método de regresión lineal simple. El problema con las redes recurrentes es que están diseñadas para grandes conjuntos de datos y en este caso el tamaño de los subconjuntos de la validación cruzada era pequeño, resultando en pequeñas ventanas de tiempo que no lograron brindar la suficiente información para las estimaciones de los proyectos.

En particular las redes neuronales son auto-organizadas, con tolerancia a fallas y operación en tiempo real una vez que los modelos han sido entrenados. Todas las redes neuronales de tipo perceptrón multicapa lograron adaptarse al contexto del problema emitiendo estimaciones de cada uno de los proyectos con desviaciones pequeñas que posicionaron a las estimaciones cercanas a los valores reales de tiempo de finalización y costo total de los proyectos.

Una vez concluida la ejecución de los experimentos podemos responder a la segunda pregunta de investigación **RQ-2**: ¿Cómo se puede mejorar la precisión de las estimaciones de tiempo de finalización y costo total de un proyecto de software? Como lo muestran los resultados, es posible reducir la desviación estándar de las estimaciones cuando se clasifica el conjunto de entrada. Adicionalmente, la aplicación de un ensamble de tipo *Bootstrap aggregating (Bagging)* contribuye a la reducción de los errores de las estimaciones al promediar las predicciones de cada una de las técnicas aplicadas, en este caso el ensamble promedia diez técnicas distintas. El ensamble tipo *Bagging* es más adecuado para problemas con conjuntos de datos de entrenamiento disponibles relativamente pequeños [95].

El ensamble final reduce la desviación estándar de los proyectos a no más de cuatro días laborales y la diferencia con el valor real de tiempo no supera la semana laboral. En el caso de las estimaciones de costo, solo dos proyectos presentaron una desviación estándar superior a las cien mil rupias, y una diferencia que supera esta misma cantidad con respecto a los valores reales de costo.

En el contexto del aprendizaje automático, la estimación del esfuerzo del software es un problema de regresión. La resolución mediante el método de ensamble aplica un conjunto de modelos formado por más de un algoritmo de aprendizaje automático, y al resolver un problema mediante distintas técnicas se incrementa el uso de recursos durante la etapa de entrenamiento. El ensamble es costoso en términos de tiempo y espacio ya que la salida del modelo ensamblado requiere la ejecución de la validación cruzada de los n modelos que lo componen, y es solo cuando todos los algoritmos han sido entrenados que se puede emitir un valor final de estimación.

La importancia de la aplicación de aprendizaje automático en un contexto real permite a los líderes robustecer sus herramientas al planificar el desarrollo de un proyecto de software. Los valores de estimación arrojados por los algoritmos no son una regla a seguir, en cambio permiten generar distintos escenarios a través de los cuales el equipo de desarrollo y los analistas pueden generar casos hipotéticos de tiempo y costo para un mismo proyecto. El utilizar un grupo de técnicas, crea un rango de tiempo y de costo para cada proyecto que permite al equipo involucrado ubicar fechas estimadas y costos aproximados, tomando como punto de partida la pila de producto y la velocidad del equipo de desarrollo asignado al proyecto.

El enfoque de puntos de historia de usuario permite el desarrollo de modelos matemáticos que pueden ser aplicados en estimación de esfuerzo, que en combinación con el juicio de expertos se logra robustecer la toma de decisiones en proyectos de desarrollo de software. Con los resultados obtenidos por los algoritmos aseguramos que la fiabilidad de las estimaciones tienen una validez en escenarios que cuenten con una base de datos sólida de proyectos previamente desarrollados con alguna

metodología ágil como Scrum.

8.1. Trabajo a futuro

Como parte del trabajo a futuro hay distintos algoritmos de aprendizaje supervisado y aprendizaje profundo que se pueden aplicar a la estimación de tiempo y costo de proyectos de software, como son las redes convolucionales, los *transformers* y las máquinas de soporte vectorial. El presente trabajo da pauta a continuar el estudio del campo de aprendizaje profundo para realizar estimaciones a nivel de puntos de historia de usuario tomando como entrada la información contenida en la historia en forma de lenguaje natural. Con un conjunto de datos apropiado, la propuesta actual se puede robustecer creando un sistema capaz de predecir a nivel de historia de usuario y a nivel de proyecto.

Además, al verificar el desempeño de las redes neuronales, un siguiente paso consiste en la aplicación de algoritmos de optimización y otros procedimientos que mejoren las ponderaciones de la red con la finalidad de mejorar la precisión de las estimaciones. Un ejemplo es la aplicación de algoritmos genéticos para la optimización de las ponderaciones de los modelos entrenados, lo que permitiría producir mejores estimaciones. Otra posible técnica a implementar es el análisis de sensibilidad que se basa en la detección de las ponderaciones más importantes de cada capa para su posterior modificación mediante la adición de ruido gaussiano. Esto también permite modificar las estimaciones emitidas por la red neuronal e incrementar su precisión.

Finalmente también es posible dar continuidad a la clasificación de proyectos a través técnicas de aprendizaje no supervisado con el fin de etiquetar automáticamente los proyectos por tamaño de esfuerzo, tiempo y costo, aplicando técnicas de *clustering* en lugar de hacer la clasificación manualmente en una hoja de cálculo.

9. Apendice

9.1. Discretización de datos

Los datos se clasifican de manera empírica durante la realización del proyecto, pero es posible realizarlo de manera automática utilizando la clase *KBinsDiscretizer* mediante cuantiles como se muestra en el Código Fuente 1. Existen tres estrategias distintas para generar los grupos y es necesario cambiar el parámetro *strategy* por los siguientes valores: *'quantile'*, *'uniform'*, *'kmeans'*.

Código Fuente 1: *Discretización de datos*

```
1 import pandas as pd #Cargar datos de CSV en un dataframe
2 import numpy as np #Arreglos tipo numpy
3 from sklearn.preprocessing import KBinsDiscretizer
4 dataset = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/SimpleANN/dataset.csv')
5 dataset.drop(["No", "D", "V", "SprintSize", "WorkDays", "TeamSalary"], axis=1, inplace=True)
6     ↪ #Eliminar columnas no utilizadas
7 effort=dataset.iloc[:,0:1].values #Columna Effort
8 time=dataset.iloc[:,2:3].values #(ActualTime = 2:3, ActualCost = 3:4)
9 cost=dataset.iloc[:,3:4].values
10 #Crear instancias para discretizar
11 est = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='quantile')
12 est.fit(effort)
13 est2 = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='quantile')
14 est2.fit(time)
15 est3 = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='quantile')
16 est3.fit(cost)
17 #Etiquetar datos por tamaño de esfuerzo, tiempo y costo
18 DiscEff = est.transform(effort)
19 DiscTime = est2.transform(time)
20 DiscCost = est3.transform(cost)
21 #Guardar vategorias en un solo arreglo
22 array = np.append(DiscEff, DiscTime, axis=1)
23 array = np.append(array, DiscCost, axis=1)
24 #Guardar en un archivo CSV
25 DF = pd.DataFrame(array)
26 DF.to_csv("/content/drive/MyDrive/Colab Notebooks/discretizerUniform.csv")
```

9.2. Preprocesamiento de datos

Todos los algoritmos reciben los mismos datos de entrenamiento y prueba por lo que la sección de Código Fuente 2 es común a KNN, DT y MLP. En el caso del autoencoder, el algoritmo recibe como datos de entrada el conjunto de datos original obtenido en el trabajo de Zia et al. [7] y las categorías de clasificación, ya que las características del algoritmo permiten reproducir la técnica de PCA al comprimir la información.

Los experimentos se ejecutaron en archivos por separado y la estimaciones de tiempo y costo también se hicieron de la misma manera, en la línea 87 del Código Fuente 2, el usuario debe cambiar la numeración de las columnas para estimar tiempo de finalización o costo total.

Cuando el conjunto de datos $y = \text{dataset.iloc[:, 2 : 3].values}$ se obtienen los datos de tiempo, y cuando $y = \text{dataset.iloc[:, 3 : 4].values}$ se copia la columna de costo

total del conjunto de datos.

Código Fuente 2: Preprocesamiento de datos

```
1  #Prueba si bibliotecas instaladas si no instala
2  try:
3  import scikeras
4  except ImportError:
5  !python -m pip install scikeras
6
7  #Eliminar advertencias de TensoFlow
8  import warnings
9  from tensorflow import get_logger
10 get_logger().setLevel('ERROR')
11 warnings.filterwarnings("ignore", message="Setting the random state for TF")
12 # importar bibliotecas
13 import pandas as pd #Leer el archivo CSV, usar dataframes para almacenar los datos.
14 import numpy as np #para usar arreglos tipo numpy
15 from tensorflow import random
16 #Para la red neuronal se necesita importar:
17 from keras.models import Sequential #Modelo secuencial para poder crear redes de tipo
18     ↳ feed-forward
19 from keras.layers import Dense, Activation #Las capas densas conectan todos sus elementos
20     ↳ de entrada y salida a sus capas adyacentes
21 from keras.layers import GaussianNoise #Mecanismo de regularizacion
22 from keras.layers import Dropout #Otro mecanismo de regularizacion
23 #from tf.keras.layers import GaussianDropout
24 from tensorflow.keras import layers
25 from tensorflow.keras import initializers
26 from keras import regularizers
27 from tensorflow.keras.constraints import MaxNorm, UnitNorm, MinMaxNorm, NonNeg
28     ↳ #Restricciones para que los pesos de las capas sean no negativos
29 from keras.callbacks import EarlyStopping, ModelCheckpoint #EarlyStopping evita
30     ↳ overfitting, ModelCheck salva el modelo
31 from scikeras.wrappers import KerasRegressor #Wrapper para poder insertar el modelo de
32     ↳ Keras como estimador en k-Fold CrossValidation (CV)
33 #Bibliotecas para validación cruzada:
34 from sklearn.model_selection import KFold, RepeatedKFold, StratifiedKFold #Tipos de
35     ↳ validacion cruzada
36 from sklearn.model_selection import cross_val_score, cross_val_predict, cross_validate
37     ↳ #Criterios de evaluacion para CV
38 from sklearn.metrics import SCORERS #metricas disponibles para la validacion cruzada
39 #Otros
40 from sklearn.ensemble import AdaBoostRegressor, BaggingRegressor #meta-estimador
41 from sklearn.pipeline import make_pipeline #Automatizar el proceso de escalar datos,
42     ↳ aplicar la red neuronal, entrenar, etc.
43 import tensorflow as tf
44 from keras import backend as K #
45 from tensorflow import keras #
46 #Metricas de evaluacion, encoding y escalador de datos
47 from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error #funciones
48     ↳ de error y r2
49 from math import sqrt
50 from sklearn.preprocessing import LabelEncoder, OneHotEncoder
51 from sklearn.preprocessing import MinMaxScaler, StandardScaler #Escalador de datos
52 from numpy import array #Para ejemplo de encoding
53 from numpy import argmax
54 import matplotlib as mpl #para graficar
55 import matplotlib.pyplot as plt #graficar
56 import seaborn as sns #Para ejemplo de histogramas
57 from numpy.random import normal
58 from sklearn.metrics import explained_variance_score
59 #####
60 #Inicio de preprocesamiento de datos
61 #####
62 dataset = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/SimpleANN/dataset.csv')
63     ↳ #Cargar archivo CSV
64 #Aqui es importante separar primero la tabla de costo y tiempo para hacer predicciones por
65     ↳ separado
66 dataset.drop(["No", "D", "V", "SprintSize", "WorkDays", "TeamSalary"], axis=1, inplace=True)
67     ↳ #Eliminar columnas no utilizadas
68 # Seccion de codigo para hacer encoding de datos:
```

```

57 data = ['M', 'L', 'M', 'L', 'M', 'L', 'S', 'L', 'S',
↳ 'L', 'M', 'S', 'S', 'S', 'L', 'M', 'M', 'L', 'M', 'S'] #Valores para la columna Effort
58 dataT = ['L', 'L', 'M', 'L', 'S', 'L', 'S', 'L', 'S',
↳ 'M', 'M', 'M', 'S', 'S', 'L', 'M', 'M', 'L', 'M', 'S'] #Time
59 dataC = ['M', 'L', 'M', 'L', 'M', 'L', 'S', 'L', 'S',
↳ 'M', 'M', 'S', 'S', 'S', 'L', 'M', 'M', 'L', 'M', 'S'] #Cost
60 values = array(data) # Guarda valores como array
61 label_encoder = LabelEncoder() # INTEGER ENCODE
62 integer_encoded = label_encoder.fit_transform(values)
63
64 dataset['Size'] = integer_encoded
65 #Tiempo
66 values = array(dataT)
67 integer_encoded = label_encoder.fit_transform(values)
68 integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
69 dataset['SizeT'] = integer_encoded
70 #Costo
71 values = array(dataC)
72 integer_encoded = label_encoder.fit_transform(values)
73 integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
74 dataset['SizeC'] = integer_encoded
75 print(dataset.head())
76 scr = MinMaxScaler(feature_range = (0, 1))
77
78 datasetRuido = pd.read_csv('/content/drive/MyDrive/Colab
↳ Notebooks/SimpleANN/datasetModificadoRuido2.csv') #Cargar archivo CSV
79 datasetRuido.drop(['No', 'D', 'V', 'SprintSize', 'WorkDays', 'TeamSalary'], axis=1,
↳ inplace=True) #Eliminar columnas no utilizadas
80 datasetRuido['SizeEffort'] = label_encoder.fit_transform(datasetRuido['SizeEffort'])
81 datasetRuido['SizeTime'] = label_encoder.fit_transform(datasetRuido['SizeTime'])
82 datasetRuido['SizeCosto'] = label_encoder.fit_transform(datasetRuido['SizeCosto'])
83 print(datasetRuido.head())
84
85 #ASIGNAR PROYECTOS REALES TEST SET
86 X=dataset.iloc[:,0:2].values #Columna Effort, Vi --
87 y=dataset.iloc[:,2:3].values #target (ActualTime = 2:3, ActualCost = 3:4)
88 X = np.append(X, dataset.iloc[:,4:7].values , axis=1) #Categorias
89
90 x_test = X #los valores de prueba son los valores reales de tiempo de los proyectos.
91 sc = MinMaxScaler(feature_range = (0, 1)) # normalizacion para X
92 x_test_scaled = sc.fit_transform(x_test) # escalamiento
93
94 Xt=datasetRuido.iloc[:,0:2].values #Columna Effort, Vi --
95 Xt = np.append(Xt, datasetRuido.iloc[:,4:7].values , axis=1) #Categorias
96 x_train = Xt #Se entrena con ruido para usar los valores reales como test.
97 x_train_scaled = scr.fit_transform(x_train) # escalamiento
98 y_train = datasetRuido.iloc[:,2:3].values #target (ActualTime = 2:3, ActualCost = 3:4)
99
100 sc2 = MinMaxScaler(feature_range = (0, 1)) # Un escalador con las dimensiones de y
101 y_train_scaled = sc2.fit_transform(y_train) # escalamiento
102
103 def calcularResultados(y,y_pred): #Calcula errores de la prediccion generada.
104 m = tf.keras.metrics.MeanRelativeError(normalizer=y)
105 m.update_state(y, y_pred)
106 mre = m.result().numpy()
107 accuracy = (1-mre)*100
108 print('PRED: ', np.around(accuracy, decimals=6) )
109 print('R2: ', np.around(r2_score(y,y_pred), decimals=8))
110 print('MRE (tf): ', mre)
111 print('MSE: ' + str(round(mean_squared_error(y, y_pred), 4)))
112 print("RMSE: %.4f" % (sqrt(mean_squared_error(y, y_pred))))
113 print()
114 return accuracy
115
116 inputDim = x_test.shape[1] # Toma como referencia el numero de columnas del conjunto de
↳ prueba
117
118 seed = 1
119 np.random.seed(seed)
120 random.set_seed(seed)

```

9.3. Configuración de algoritmos

Cada uno de los algoritmos se ejecuta por separado y al ejecutar el entrenamiento es necesario cambiar la instancia del modelo que se desee entrenar. Esto es, al ejecutar la validación cruzada, el primer parámetro que recibe el método es el estimador a entrenar, este valor debe ser cambiado según el algoritmo que se desee ejecutar.

9.3.1. K-Nearest Neighbors

En el Código Fuente 3 se muestra la configuración del modelo de los k vecinos más cercanos.

Código Fuente 3: Configuración de algoritmo KNN.

```
1 def createKNNmodel(k):
2     KNN = KNeighborsRegressor(n_neighbors=k, weights='distance',
3                               algorithm='kd_tree', metric='minkowski')
4     classifier_pipeline = make_pipeline(MinMaxScaler(), KNN)
5     return classifier_pipeline
6 KNN = createKNNmodel(2)
```

9.3.2. Árboles de decisión

En el Código Fuente 4 se muestra la configuración del modelo de árboles de decisión, bosque aleatorio y *AdaBoost*.

Código Fuente 4: Configuración de árboles de decisión.

```
1 regrDT = DecisionTreeRegressor(max_depth= 5, min_samples_split=2, random_state=0)
2 regrRF = RandomForestRegressor(n_estimators=10, max_depth=6, random_state=0)
3 regrAda = AdaBoostRegressor(regrDT, random_state=0, n_estimators=10)
```

9.3.3. Perceptrón multicapa

En el Código Fuente 5 se muestra la configuración del modelo de red neuronal de tipo perceptrón multicapa. Para ejecutar el entrenamiento aplicando la validación cruzada es necesario montar la red en una extensión que permita entrenar la red de *Keras* con el API de *scikit-learn*.

Código Fuente 5: Configuración de perceptrón multicapa.

```
1 #El siguiente bloque crea la arquitectura de red neuronal a través del método get_reg().
2 def get_reg():
3     # build the Keras model
4     model = Sequential()
5     i=8
6     model.add(Dense(i, input_dim=inputDim, activation='tanh', name="CapaEntrada" ))
7     model.add(GaussianNoise(stddev=0.00001))
8     model.add(Dense(i, activation='tanh', name="PrimeraOculta" )) #
```

```

9 model.add(GaussianNoise(stddev=0.00001))
10 model.add(Dense(i, activation='tanh', name="SegundaOculta"))#
11 model.add(Dense(1, activation= 'linear', name="CapaSalida"))
12 opt = keras.optimizers.Adam() #Adam y tanh
13 model.compile(loss='mse', optimizer=opt) #, metrics=['mean_absolute_error',
    ↳ 'RootMeanSquaredError']
14 return model
15
16 #El siguiente bloque utiliza una extension (KerasRegressor) para poder entrenar una red de
    ↳ Keras con el API de scikit-learn.
17 cb = [EarlyStopping(monitor="val_loss", patience=80,mode='min',restore_best_weights=True)]
18 reg = KerasRegressor(
19 model=get_reg,
20 loss="mse",
21 metrics=[KerasRegressor.r_squared],
22 epochs=2000,
23 callbacks=[cb],
24 validation_split=0.3,
25 verbose =0
26 )

```

9.3.4. Red Neuronal Recurrente

En el Código Fuente 6 se muestra la configuración del modelo de red neuronal recurrente.

Código Fuente 6: Configuración de red neuronal recurrente.

```

1 model = Sequential()
2 model.add(SimpleRNN(units=i,input_shape=in_dim, \
3 activation="relu",return_sequences=True))
4 model.add(GaussianNoise(stddev=0.00001))
5 model.add(SimpleRNN(units=i, activation="relu",\
6 recurrent_dropout=0.1))
7 model.add(Dense(10))
8 model.add(Dense(out_dim))
9 model.compile(loss='mse', optimizer='adam')

```

9.3.5. Autoencoder

En el Código Fuente 7 se muestra la configuración del modelo de red neuronal de tipo *Autoencoder*. Para probar las tres arquitecturas propuestas es necesario comentar o descomentar aquellas capas tipo *Dense* que se desean agregar o eliminar. Por defecto el código crea la arquitectura tradicional de AE.

Código Fuente 7: Configuración de autoencoder.

```

1 def get_reg():
2     model = Sequential()
3     i=9
4     model.add(Dense(i,input_dim=inputDim,activation='tanh',name="CapaEntrada"))
5     model.add(GaussianNoise(stddev=0.00001))
6     model.add(Dense(i-3, activation='tanh',name="PrimeraEncoder" ))#
7     model.add(GaussianNoise(stddev=0.00001))
8     model.add(Dense(i-6, activation='tanh', name="SegundaEncoder"))#
9     model.add(Dense(i-3, activation='tanh',name="PrimeraDecoder" ))#
10    model.add(Dense(i, activation='tanh',name="SegundaDecoder" ))#
11    #MLP

```

```

12     #model.add(Dense(2, activation='tanh', name="MLP1"))#
13     #model.add(Dense(2, activation='tanh', name="MLP2"))#
14     model.add(Dense(1, activation= 'linear', name="CapaSalida"))
15     opt = keras.optimizers.Adam() #Adam y tanh
16     model.compile(loss='mse', optimizer=opt)
17     return model

```

9.4. Entrenamiento de algoritmos

Todos los algoritmos tienen en común el entrenamiento aplicado, la validación cruzada de diez mangas. El Código Fuente 8 presenta la configuración y ejecución mediante el método *cross_validate*. En el primer parámetro es necesario reemplazar *reg* por cualquiera de los algoritmos previamente definidos como *KNN*, *regrDT*, *regrRF*, *regrAda*.

Código Fuente 8: *Entrenamiento*

```

1  #En el siguiente bloque se define el procedimiento de la validación cruzada, el método
   ↳ cross_validate recibe como parametro el estimador creado anteriormente reg.
2  n_splits=10
3  n_repeats=2
4  rkf = RepeatedKfold(n_splits=n_splits, n_repeats=n_repeats, random_state=30)
5  cv_results = cross_validate(reg, x_train_scaled, y_train_scaled,
   ↳ cv=rkf, return_estimator=True, return_train_score=True, scoring=('r2', 'neg_mean_squared_error',
   ↳ 'explained_variance', 'neg_root_mean_squared_error') )

```

9.5. Evaluación de resultados

Una vez que los modelos han sido entrenados se imprimen las métricas de la validación cruzada y se obtienen las predicciones de los proyectos con su desviación estandar como se muestra en el Código Fuente 9.

Código Fuente 9: *Evaluación de resultados*

```

1  #Las métricas de la validación cruzada se guardan en cv_results, se imprime el promedio de
   ↳ todos los estimadores.
2  print(cv_results.keys())
3  #print(cv_results.items()) #prints keys and values.items() #prints keys and values
4  #type(cv_results.values())
5  print('Promedio de train_MSE:
   ↳ ', np.abs(np.mean(cv_results.get('train_neg_mean_squared_error'))))
6  print('Promedio de train_RMSE:
   ↳ ', np.abs(np.mean(cv_results.get('train_neg_root_mean_squared_error'))))
7  print('Promedio de train_r2: ', np.abs(np.mean(cv_results.get('train_r2'))))
8  print('Promedio de train_explained_variance:
   ↳ ', np.abs(np.mean(cv_results.get('train_explained_variance'))))
9  print('Promedio de test_MSE:
   ↳ ', np.abs(np.mean(cv_results.get('test_neg_mean_squared_error'))))
10 print('Promedio de test_RMSE:
   ↳ ', np.abs(np.mean(cv_results.get('test_neg_root_mean_squared_error'))))
11 print('Promedio de test_r2 : ', np.abs(np.mean(cv_results.get('test_r2'))))
12 print('Promedio de test_explained_variance :
   ↳ ', np.abs(np.mean(cv_results.get('test_explained_variance'))))
13
14 #El siguiente bloque calcula el promedio de épocas de entrenamiento de todos los
   ↳ estimadores.
15 promedioEpocas = []

```



```

16 for i in range( len(cv_results.get('estimator')) ):
17     est = cv_results.get('estimator')[i]
18     #print(f"Trained {len(est.history_['loss'])} epochs")
19     promedioEpocas.append(len(est.history_['loss']))
20     np.around(np.mean(promedioEpocas), decimals=0)
21
22     #El siguiente bloque imprime las métricas individuales de cada estimador.
23     predTime = []
24     for i in range(n_splits*n_repeats):
25         print('estimador = ',i)
26         estimatorFold = cv_results.get('estimator')[i] # y 11
27         ypt_scaled = estimatorFold.predict(x_test_scaled)
28         ypt_scaled = ypt_scaled.reshape(-1,1)
29         ypt_unscaled = sc2.inverse_transform(ypt_scaled)
30         predTime.append(ypt_unscaled)
31         calcularResultados(y_train[0:21],ypt_unscaled)
32
33     #Una vez entrenada la red se obtienen los valores de las estimaciones con el siguiente
    ↪ bloque.
34     prediccionFinalTiempo = []
35     stddevTiempo = []
36     for j in range(21):
37         proyectos = []
38         for i in range(n_splits*n_repeats):
39             proyectos.append( predTime[i][j] )
40             print( np.around(np.mean(proyectos), decimals=0) )
41             prediccionFinalTiempo.append(np.around(np.mean(proyectos), decimals=0))
42             stddevTiempo.append(np.std(proyectos, ddof=1))
43
44     #Calcular las desviaciones estandar de cada estimación.
45     for i in range(21):
46         print(np.around(stddevTiempo[i], decimals=2))
47
48     #Calcular las métricas tomando los valores de estimación final.
49     calcularResultados(y_train[0:21],prediccionFinalTiempo)
50
51     print(explained_variance_score(y_train[0:21],prediccionFinalTiempo))

```

9.6. Ejecución de Red Neuronal Recurrente

Como se explicó en la Sección 6.3.4, la red neuronal recurrente necesita un formato de la información como serie temporal. En el Código Fuente 10 se muestra el procedimiento completo para poder entrenar la red y generar los resultados.

Código Fuente 10: *Ejecución de red neuronal recurrente*

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from math import sqrt
5 from sklearn.preprocessing import StandardScaler
6 from sklearn.pipeline import make_pipeline
7 from keras.callbacks import EarlyStopping, ModelCheckpoint #EarlyStopping evita
    ↪ overfitting, ModelCheck salva el modelo
8 from keras.wrappers.scikit_learn import KerasRegressor #Wrapper para poder insertar el
    ↪ modelo de Keras como estimador en k-Fold CrossValidation (CV)
9 #Bibliotecas para validación cruzada:
10 from sklearn.model_selection import KFold, RepeatedKFold, StratifiedKFold #Tipos de
    ↪ validacion cruzada
11 from sklearn.model_selection import cross_val_score, cross_val_predict, cross_validate
    ↪ #Criterios de evaluacion para CV
12 from sklearn.metrics import SCORERS #metricas disponibles para la validacion cruzada
13 from numpy import array
14 from numpy import argmax
15 from sklearn.metrics import mean_squared_error, r2_score

```

```

16 from keras.models import Sequential
17 from keras.layers import Dense, SimpleRNN, Dropout, LSTM, GaussianNoise
18 from numpy.random import uniform
19 from numpy import hstack
20 from sklearn.model_selection import train_test_split
21 from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler,
    ↳ MinMaxScaler
22 from tensorflow import random
23 import tensorflow as tf
24 import random as rand
25 from keras.layers import Flatten
26 from keras.layers.convolutional import Conv1D
27 from keras.layers.convolutional import MaxPooling1D
28 from sklearn.model_selection import TimeSeriesSplit
29 from numpy.random import normal
30 seed = 1
31 np.random.seed(seed)
32 random.set_seed(seed)
33
34 def imprimirResultados(y,y_pred):
35     print('MSE: ' + str(round(mean_squared_error(y, y_pred), 4)))
36     print("RMSE: %.4f" % (sqrt(mean_squared_error(y, y_pred))))
37     print('R2: ', r2_score(y,y_pred))
38     m = tf.keras.metrics.MeanRelativeError(normalizer=y)
39     m.update_state(y, y_pred)
40     mre = m.result().numpy()
41     print('MRE (tf): ', mre)
42     pred = (1-mre)*100
43     print('PRED: ', pred)
44     print('-----')
45     return pred
46
47 def crearLista(arreglo):
48     def extract_tuple_value(tup):
49         return tup[0]
50     my_list = [tup[0] for tup in arreglo]
51     return my_list
52
53 def printLoss(history):
54     # Get training and test loss histories
55     plt.close('all')
56     fig = plt.figure()
57     training_loss = history.history["loss"]
58     test_loss = history.history["val_loss"]
59     epoch_count = range(1, len(training_loss) + 1) # Create count of the number of
    ↳ epochs
60     # Visualize loss history
61     plt.plot(epoch_count, training_loss, "r--",linewidth=1)
62     plt.plot(epoch_count, test_loss, "b-",linewidth=1)
63     #plt.ylim(0.001,0.3)
64     plt.legend(["Training Loss", "Test Loss"])
65     plt.xlabel("Epoch")
66     plt.ylabel("Loss")
67     plt.show()
68     return fig
69
70 def convertData(datax,datay,step):
71     X, Y = [], []
72     for i in range(len(datax)-step):
73         d = i+step
74         X.append(datax[i:d,])
75         Y.append(datay[d])
76
77     return array(X), array(Y)
78
79 dataset = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/SimpleANN/datasetFinal.csv')
80 dataset.drop(["No", "D", "V", "SalaryTeam", "SprintSize", "WorkDays"], axis=1, inplace=True)
81
82 X=dataset.iloc[:,0:2].values #Effort
83 y=dataset.iloc[:,2:4].values #target (ActualTime = 2:3, ActualCost = 3:4)
84 #Columnas: 4:5 = Size, 5:6 = SizeT, 6:7 = SizeC = 6:
85 X = np.append(X, dataset.iloc[:,4:7].values , axis=1)
86 x_test = X

```

```

87 x_test = np.append(x_test, x_test[0:4 ,:] ,axis=0)
88 y_test = y
89 y_test = np.append(y_test, y_test[0:4 ,:] ,axis=0)
90 y_train = y
91 y_train = np.concatenate((y_train, y_train), axis=0) #Duplicar datos para el dataset con
    ↪ ruido de 42 registros
92 print('x_test set:', x_test.shape)
93 print(x_test[0:4,:])
94 print('y_train set:', y_train.shape)
95 print(y_train[0:4,:])
96 datasetRuido = pd.read_csv('/content/drive/MyDrive/Colab
    ↪ Notebooks/SimpleANN/datasetModificadoRuido2.csv') #Cargar archivo CSV
97 datasetRuido.drop(['No", "D", "V", "SprintSize", "WorkDays", "TeamSalary"], axis=1,
    ↪ inplace=True) #Eliminar columnas no utilizadas
98
99 ###Lineas de codigo para dataset con ruido de 42 registros
100 z = dataset.iloc[:,4:7].values
101 #print('z shape' , z.shape)
102 d = pd.DataFrame(np.concatenate((z, z), axis=0))
103 datasetRuido['SizeEffort'] = d[0]
104 datasetRuido['SizeTime'] = d[1]
105 datasetRuido['SizeCosto'] = d[2]
106
107 Xt = datasetRuido.iloc[:,0:2].values #Effort
108 Xt = np.append(Xt, datasetRuido.iloc[:,4:7].values , axis=1)
109 x_train = Xt
110
111
112 #Los datos se normalizan entre cero y uno
113 sc = MinMaxScaler(feature_range = (0, 1)) # normalizacion para X
114 x_train_scaled = sc.fit_transform(x_train) # escalamiento
115 sc2 = MinMaxScaler(feature_range = (0, 1)) # normalizacion para X
116 y_train_scaled = sc2.fit_transform(y_train) # escalamiento
117 sc3 = MinMaxScaler(feature_range = (0, 1))
118 x_test_scaled = sc3.fit_transform(x_test)
119 sct = MinMaxScaler(feature_range = (0, 1))
120 y_test_scaled = sct.fit_transform(y_test)
121
122 #Configuración de validación cruzada
123 step=4
124 i=40 #40 para SimpleRNN, 30 para LSTM
125 callbacks = [EarlyStopping(monitor="val_loss", patience=100, restore_best_weights=True)]
126 checkpoint = [ModelCheckpoint("keras_model.pt", monitor='val_loss', save_best_only=True)]
127 train_scores = []
128 modelosEntrenados = []
129 tscv = TimeSeriesSplit(n_splits=10)
130
131 for train_index, test_index in tscv.split(x_train_scaled):
132     print("TRAIN:", train_index, "TEST:", test_index)
133     X_train, X_test = x_train_scaled[train_index], x_train_scaled[test_index]
134     Y_train, Y_test = y_train_scaled[train_index], y_train_scaled[test_index]
135
136     testx,testy = convertData(X_test,Y_train, step)
137     trainx,trainy = convertData(X_train,Y_train , step)
138     print("test-x:", testx.shape, "test-y:", testy.shape)
139     print("train-x:", trainx.shape, "train-y:", trainy.shape)
140
141     in_dim = trainx.shape[1:3]
142     out_dim = trainy.shape[1]
143     model = Sequential()
144     model.add(SimpleRNN(units=i,input_shape=in_dim,
        ↪ activation="relu",return_sequences=True)) # ,return_sequences=True
145     model.add(GaussianNoise(stddev=0.00001))
146     model.add(SimpleRNN(units=i, activation="relu",recurrent_dropout=0.1)) #
        ↪ ,dropout=0.1
147     #model.add(SimpleRNN(units=i, activation="relu",dropout=0.2))#
        ↪ ,recurrent_dropout=0.2
148     model.add(Dense(10)) #10 para SimpleRNN, 5 para LSTM
149     model.add(Dense(out_dim))
150     model.compile(loss='mse', optimizer='adam')
151     #model.summary()

```

```

152     history = model.fit(trainx,trainy,epochs=1000,
    ↪ validation_data=(testx,testy),validation_split=0.25, batch_size=32,verbose=0,
    ↪ shuffle=False,callbacks=callbacks) #
153     trainScore = model.evaluate(trainx, trainy, verbose=0)
154
155     print('Train score: ', trainScore)
156     train_scores.append(trainScore)
157     print('minimo de loss:',np.min(history.history.get("loss")))
158     print('minimo de val_loss:',np.min(history.history.get("val_loss")))
159     printLoss(history)
160     modelosEntrenados.append(history.model)
161
162     step=4
163     testx,testy = convertData(x_test_scaled,y_train_scaled, step)
164     trainx,trainy = convertData(x_train_scaled,y_train_scaled , step)
165
166     prediccionesTiempo =[]
167     prediccionesCosto =[]
168
169     for i in range(len(modelosEntrenados)):
170         bestModel = modelosEntrenados[i]
171         predtest = bestModel.predict(testx)
172         predicciones = sc2.inverse_transform(predtest)
173         TiempoEstimado = predicciones[:,0]
174         CostoEstimado = predicciones[:,1]
175         TiempoReal = y_train[0:21,0]
176         CostoReal = y_train[0:21,1]
177         parte2 = TiempoEstimado[0:17]
178         parte1 = TiempoEstimado[17:]
179         TiempoReordenado = np.array([[]])
180         TiempoReordenado = np.append(TiempoReordenado,parte1,axis=None)
181         TiempoReordenado = np.append(TiempoReordenado,parte2,axis=None)
182         c2 = CostoEstimado[0:17]
183         c1 = CostoEstimado[17:]
184         CostoReordenado = np.array([[]])
185         CostoReordenado = np.append(CostoReordenado,c1,axis=None)
186         CostoReordenado = np.append(CostoReordenado,c2,axis=None)
187         print('Tiempo')
188         t = imprimirResultados(TiempoReal,TiempoReordenado)
189         print('Costo')
190         c = imprimirResultados(CostoReal,CostoReordenado)
191         print()
192         prediccionesTiempo.append(TiempoReordenado)
193
194     prediccionFinalTiempo = []
195     stddevTiempo = []
196     stddevCosto = []
197     for j in range(21):
198         proyectos = []
199         for i in range(5,10):
200             proyectos.append( prediccionesTiempo[i][j] )
201             print( np.around(np.mean(proyectos),decimals=0) )
202             prediccionFinalTiempo.append(np.around(np.mean(proyectos),decimals=0))
203             stddevTiempo.append(np.std(proyectos,ddof=1))
204     imprimirResultados(TiempoReal,prediccionFinalTiempo)
205
206     prediccionFinalCosto = []
207
208     for j in range(21):
209         proyectos = []
210         for i in range(5,10):
211             proyectos.append( prediccionesCosto[i][j] )
212             print( np.around(np.mean(proyectos),decimals=0) )
213             prediccionFinalCosto.append(np.around(np.mean(proyectos),decimals=0))
214             stddevCosto.append(np.std(proyectos,ddof=1))
215     imprimirResultados(CostoReal,prediccionFinalCosto)
216
217     for i in range(21):
218         print(np.around(stddevTiempo[i], decimals=2))
219
220     for i in range(21):
221         print(np.around(stddevCosto[i], decimals=0))
222

```

```
223 from sklearn.metrics import explained_variance_score
224 print(explained_variance_score(TiempoReal, prediccionFinalTiempo))
225 print(explained_variance_score(CostoReal, prediccionFinalCosto))
```

10. Bibliografía

Referencias

- [1] A. Panda, S. M. Satapathy, and S. Rath, "Empirical validation of neural network models for agile software effort estimation based on story points," vol. 57, 08 2015.
- [2] C. P. Rao, P. S. Kumar, S. R. Sree, and J. Devi, "An Agile Effort Estimation Based on Story Points Using Machine Learning Techniques," Proceedings of the Second International Conference on Computational Intelligence and Informatics Advances in Intelligent Systems and Computing, pp. 209–219, 2018.
- [3] A. T. Raslan, et al., "Effort Estimation in Agile Software Projects using Fuzzy Logic and Story Points," 50th Annual Conference on statistics, computer sciences, and operation research, 27-30 2015.
- [4] E. Scott and D. Pfahl, "Using developers features to estimate story points", Proceedings of the 2018 International Conference on Software and System Process, 2018.
- [5] S. M. Satapathy and S. K. Rath, "Empirical assessment of machine learning models for agile software development effort estimation using story points," Innovations in Systems and Software Engineering, vol. 13, no. 2-3, pp. 191–200, 2017.
- [6] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A Deep Learning Model for Estimating Story Points," IEEE Transactions on Software Engineering, vol. 45, no. 7, pp. 637–656, 2019.
- [7] Z. K. Zia, S. K. Tipu and S. K. Zia. "An Effort Estimation Model for Agile Software Development." Advances in Computer Science and its Applications,. p. 314-324, 2012.
- [8] E. Coelho and A. Basu, "Effort Estimation in Agile Software Development using Story Points," International Journal of Applied Information Systems, vol. 3, no. 7, pp. 7–10, 2012.
- [9] M. Adnan and M. Afzal, "Ontology Based Multiagent Effort Estimation System for Scrum Agile Method," IEEE Access, vol. 5, pp. 25993–26005, 2017.
- [10] S. Bilgaiyan, S. Mishra, and M. Das, "Effort estimation in agile software development using experimental validation of neural network models," International Journal of Information Technology, vol. 11, no. 3, pp. 569–573, 2018.
- [11] A. Sharma and N. Chaudhary, "Linear Regression Model for Agile Software Development Effort Estimation," 2020 5th IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE), 2020.
- [12] L.-D. Radu, "Effort Prediction in Agile Software Development with Bayesian Net-

- works," Proceedings of the 14th International Conference on Software Technologies, 2019.
- [13] S. Kad and V. Chopra, "Software development effort estimation using soft computing," *International Journal of Machine Learning and Computing*, pp. 548–551, 2011.
 - [14] M. Kubat, and Kubat. *An introduction to machine learning*. Vol. 2. Cham, Switzerland: Springer International Publishing, 2017.
 - [15] M Vyas, A Bohra, Dr. C.S. Lamba, A Vyas, "A Review on Software Cost and Effort Estimation Techniques for Agile Development Process," *International Journal of Recent Research Aspects* ISSN: 2349-7688, Vol. 5, Issue 1, March 2018, pp. 1-5.
 - [16] J. Choudhari and U. Suman, "Story Points Based Effort Estimation Model for Software Maintenance," *Procedia Technology*, vol. 4, pp. 761-765, 2012. Available: 10.1016/j.protcy.2012.05.124.
 - [17] E. Alpaydin, *Introduction to machine learning*, The MIT Press, 2014
 - [18] A. Faul, *A concise introduction to machine learning*. CRC Press, 2019.
 - [19] A. C. Müller and S. Guido, *Introduction to machine learning with Python: a guide for data scientists*. O'Reilly, 2017.
 - [20] R. Shirkin, *Artificial Intelligence : The Complete Beginners' Guide to Artificial Intelligence* Editor: Rodrigo Llagas/GlobaltechNTC. Amazon KDP Printing and Publishing, 2019.
 - [21] A. Konar, *Artificial intelligence and soft computing: behavioral and cognitive modeling of the human brain*. CRC Press, 2018.
 - [22] Moocarme, M., Abdolahnejad, M. and Bhagwat, R., 2020. *The Deep Learning with Keras Workshop*. Birmingham UK: PACKT Publishing.
 - [23] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Harlow: Pearson Education Limited, 2021.
 - [24] E. Fix and J. Hodges, "Discriminatory analysis: Nonparametric discrimination: Consistency properties", *PsycEXTRA Dataset*, 1951. Available: 10.1037/e471672008-001 [Accessed 27 July 2022].
 - [25] T. Cover and P. Hart, "Nearest neighbor Pattern Classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.
 - [26] S. Rogers and M. Girolami, *A first course in machine learning*. Boca Raton: Chapman & Hall/CRC, 2020.
 - [27] R. Rojas, *Neural networks*, Berlin: Springer, 1996.
 - [28] J. Moolayil, *Learn Keras for Deep Neural Networks*. Berkeley, CA: Apress, 2019.

- [29] O. Malgonde and K. Chari, "An ensemble-based model for predicting agile software development effort," *Empirical Software Engineering*, vol. 24, no. 2, pp. 1017–1055, 2018.
- [30] VersionOne. 2017. 11th Annual State of Agile Survey. (2017). <https://explore.versionone.com/state-of-agile>.
- [31] 14th Annual State of Agile Report. (2019).
- [32] 15th Annual State of Agile Report. (2021).
- [33] O. Hidmi and B. E. Sakar, "Software Development Effort Estimation Using Ensemble Machine Learning," *International Journal of Computing, Communication and Instrumentation Engineering*, vol. 4, no. 1, 2017.
- [34] C. Albon, *Machine learning with Python cookbook: practical solutions from preprocessing to deep learning*. Sebastopol: O'Reilly, 2018.
- [35] R. Popli & N. Chauhan, "Cost and effort estimation in agile software development," *ICROIT 2014 - Proceedings of the 2014 International Conference on Reliability, Optimization and Information Technology*. 57-61, 2014, 10.1109/ICROIT.2014.6798284.
- [36] K.M. Hornik, M. Stinchcomb, and H. White, "Multilayer feedforward networks are universal approximator," *IEEE Transactions on Neural Networks* 2, 1989.
- [37] L. A. Hussein et al., "Recurrent Neural Network based Prediction of Software Effort", *International Journal of Computer Applications* Volume 177 – No.1, November 2017
- [38] K. Z. K. Ziauddin, S. K. Tipu and S. Zia, "An Intelligent Software Effort Estimation System," *Journal of Expert Systems (JES)*, Vol. 1, No. 4, ISSN 2169-3064, 2012.
- [39] I. Abnane, M. Hosni, A. Idri and A. Abran, "Analogy Software Effort Estimation Using Ensemble KNN Imputation", *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019.
- [40] P. S. Kumar, H. Behera, A. K. K, J. Nayak, and B. Naik, "Advancement from neural networks to deep learning in software effort estimation: Perspective of two decades," *Computer Science Review*, vol. 38, p. 100288, 2020.
- [41] P. S. Kumar, H. S. Behera, J. Nayak, and B. Naik, "A pragmatic ensemble learning approach for effective software effort estimation," *Innovations in Systems and Software Engineering*, 2021.
- [42] M. W. Khan and I. Qureshi, "Neural Network based Software Effort Estimation: A Survey", *Int. J. Advanced Networking and Applications*, Volume: 05, Issue: 04, Pages:1990-1995, 2014.
- [43] H. T. Hoc, V. V. Hai, and H. L. T. K. Nhung, "A Review of the Regression Models Applicable to Software Project Effort Estimation," *Computational Statistics and*

Mathematical Modeling Methods in Intelligent Systems Advances in Intelligent Systems and Computing, pp. 399–407, 2019.

- [44] M. Fernandez-Diego, E. R. Mendez, F. Gonzalez-Ladron-De-Guevara, S. Abrahao, and E. Insfran, "An Update on Effort Estimation in Agile Software Development: A Systematic Literature Review," *IEEE Access*, vol. 8, pp. 166768–166800, 2020.
- [45] S. Garg and D. Gupta, "PCA based cost estimation model for agile software development projects," 2015 International Conference on Industrial Engineering and Operations Management (IEOM), 2015.
- [46] O. Malgonde and K. Chari, "An ensemble-based model for predicting agile software development effort," *Empirical Software Engineering*, vol. 24, no. 2, pp. 1017–1055, 2018.
- [47] M. Durán, R. Juárez-Ramírez, S. Jiménez, and C. Tona, "User Story Estimation Based on the Complexity Decomposition Using Bayesian Networks," *Programming and Computer Software*, vol. 46, no. 8, pp. 569–583, 2020.
- [48] M. Gultekin and O. Kalipsiz, "Story Point-Based Effort Estimation Model with Machine Learning Techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 30, no. 01, pp. 43–66, 2020.
- [49] A. Pham and P. V. Pham, *Scrum in action: agile software project management and development*. Boston: Course Technology PTR, 2012.
- [50] K. Schwaber, "SCRUM Development Process," *Business Object Design and Implementation*, pp. 117–134, 1997.
- [51] Glorot, X. & Bengio, Y., "Understanding the difficulty of training deep feedforward neural networks," *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, in *Proceedings of Machine Learning Research* 9:249-256, 2010.
- [52] *A guide to the Project Management Body of Knowledge: (PMBOK Guide)*. Project Management Institute, 2017.
- [53] *Agile Practice Guide*. Newton Square, PA: Project Management Institute, 2017.
- [54] M. Cohn, *Agile Estimating and Planning*, Prentice Hall 1st edition, 2005.
- [55] R. K. Wysocki, *Effective project management: Traditional, Agile, hybrid, extreme*. Wiley, 2019.
- [56] I. Sommerville, *Software engineering*, Tenth Edition. Pearson Education, 2019.
- [57] S. Cha, R. N. Taylor, and K.-C. Kang, *Handbook of Software Engineering*. Springer International Publishing, 2019.
- [58] O. Fedotova, L. Teixeira, and A.H. Alvelos, "Software effort estimation with multiple linear regression: review and practical application," *J. Inf. Sci. Eng.*, vol 29,

- [59] ITIL Foundation Itil 4 edition. TSO (The Stationery Office), 2019.
- [60] CMMI for development, Version 1.3. Hanscom AFB, MA: SEI, 2010.
- [61] S. Grimstad, M. Jørgensen, and K. Moløkken-Østvold, "Software effort estimation terminology: The tower of babel," *Information and Software Technology*, vol. 48, no. 4, pp. 302–310, 2006.
- [62] G. R. Finnie, G. E. Wittig, and J.-M. Desharnais, "A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and Regression Models," *Journal of Systems and Software*, vol. 39, no. 3, pp. 281–289, 1997.
- [63] A. Trendowicz and R. Jeffery, *Software project effort estimation: Foundations and best practice: Guidelines for Success*. Cham: Springer, 2014.
- [64] H. Leung and Z. Fan, "Software cost estimation," in *Handbook of Software Engineering and Knowledge Engineering*, Hong Kong Polytechnic University, 2002.
- [65] G. Coleman and R. Verbruggen, "A quality software process for rapid application development," *Software Quality Journal*, Vol. 7, 1998, pp. 107-122.
- [66] M. Perkusich et al., "Intelligent software engineering in the context of agile software development: A systematic literature review," *Information and Software Technology*, vol. 119, p. 106241, 2020. Available: 10.1016/j.infsof.2019.106241.
- [67] C. E. Carbonera, K. Farias and V. Bischoff, "Software development effort estimation: a systematic mapping study", *IET Software*, vol. 14, no. 4, pp. 328-344, 2020. Available: 10.1049/iet-sen.2018.5334.
- [68] A. K. Bardsiri et al., "Software Effort Estimation: A Survey of Well-known Approaches," *International Journal of Computer Science Engineering (IJCSE)*, 2014.
- [69] T. Vera, S. Ochoa, D. Perovich, *Survey of Software Development Effort Estimation Taxonomies*. 2018, Available: 10.13140/RG.2.2.14599.29601.
- [70] N.J. Nilsson, *Introduction to Machine Learning*, class notes for Machine Learning, Department of Computer Science, Stanford University, 1996.
- [71] M. Labedzki, P. Promiński, A. Rybicki, and M. Wolski, "Agile effort estimation in software development projects - case study," *The Central European Review of Economics and Management*, vol. 1, no. 3, 2017.
- [72] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.
- [73] G. Wiederhold and J. McCarthy, "Arthur Samuel: Pioneer in machine learning," *IBM Journal of Research and Development*, vol. 36, no. 3, pp. 329–331, 1992.

- [74] A. Samuel, "Some Studies in Machine Learning Using the Game of Checkers", IBM Journal of Research and Development, vol. 3, no. 3, pp. 210-229, 1959. Available: 10.1147/rd.33.0210.
- [75] R. E. Brown, "Donald O. Hebb and the organization of behavior: 17 years in the writing," Molecular Brain, vol. 13, no. 1, 2020.
- [76] N. Matloff, Statistical Regression and Classification From Linear Models to Machine Learning. California: Chapman & Hall CRC, 2017.
- [77] E. Learned-Miller, Supervised Learning and Bayesian Classification. Amherst: University of Massachusetts, 2022.
- [78] G. Biau and L. Devroye, Lectures on the nearest neighbor method. Cham: Springer International Publishing, 2015.
- [79] G. Shakhnarovich, T. Darrell, G. Shakhnarovich, and P. Indyk, Nearest-neighbor methods in learning and vision: Theory and practice. Cambridge, MA: MIT Press, 2005.
- [80] E. Schroger, R. Rauh, and W. Schubo, "Probability distributions of Minkowski distances between discrete random variables," Educational and Psychological Measurement, vol. 53, no. 2, pp. 379–398, 1993.
- [81] J.L. Bentley, "Multidimensional binary search trees used for associative searching," Communications of the ACM, 18:509–517, 1975.
- [82] R. A. Fisher, "The use of multiple measurements in taxonomic problems," Annals of Eugenics, vol. 7, no. 2, pp. 179–188, 1936.
- [83] G. Ritschard, "CHAID and earlier supervised tree methods," Contemporary Issues in Exploratory Data Mining in the Behavioral Sciences, New York, pages 48–74, 2013.
- [84] W.-Y. Loh, "Fifty years of classification and regression trees," International Statistical Review, vol. 82, no. 3, pp. 329–348, 2014.
- [85] M. Ali, P. Hickman and A. Clementson, "The Application of Automatic Interaction Detection (AID) in Operational Research", Operational Research Quarterly (1970-1977), vol. 26, no. 2, p. 243, 1975. Available: 10.2307/3008458.
- [86] W. Y. Loh, "Classification and regression trees," WIREs Data Mining and Knowledge Discovery, vol. 1, no. 1, pp. 14–23, 2011.
- [87] F. P. Ramsey, "Foundations of mathematics and other logical essays", Routledge, 1931.
- [88] J. R. Quinlan, "Induction of Decision Trees," Machine Learning, vol. 1, no. 1, pp. 81–106, 1986.
- [89] G. V. Kass, "An exploratory technique for investigating large quantities of catego-

- rical data," *Applied Statistics*, vol. 29, no. 2, p. 119, 1980.
- [90] W. A. Belson, "Matching and prediction on the principle of biological classification," *Applied Statistics*, vol. 8, no. 2, p. 65, 1959.
- [91] K. Mittal , D. Khanduja, P.C. Tewari, "An insight into 'Decision Tree Analysis'". *World Wide Journal of Multidisciplinary Research and Development*, 3(12):111-5, 2017.
- [92] S. K. Murthy, "Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey," *Data Mining and Knowledge Discovery*, 2, 345–389, 1998
- [93] P. Chaudhuri, W.-D. Lo, W.-Y. Loh, C.-C. Yang, "Generalized regression trees," *Statistica Sinica*, vol. 5, 641–666, 1995.
- [94] L. Breiman, Random forests,"*Machine learning* 45.1, 5-32,2001.
- [95] C. Zhang and Y. Ma, "Ensemble machine learning," New York: Springer, 2012.
- [96] S. K. Sehra, J. Kaur, and S. S. Sehra, "Effect of data preprocessing on software effort estimation," *International Journal of Computer Applications*, vol. 69, no. 25, pp. 29–32, 2013.
- [97] T. Windeatt and R. Ghaderi, "AdaBoost and neural networks," *ESANN*, 1999.
- [98] R. E. Schapire, "The strength of weak learnability," *Machine learning*, 5,2 1990: 197-227.
- [99] Y. Freund and R.E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, 55(1), pp.119-139, 1997.
- [100] O. Sagi and L. Rokach, "Ensemble learning: A survey," *WIREs Data Mining and Knowledge Discovery*, vol. 8, no. 4, 2018.
- [101] A. Kumar and M. Jain, *Ensemble learning for AI developers: Learn Bagging, stacking, and boosting methods with use cases*. Berkeley, California: Apress, 2020.
- [102] R. R. Picard and R. D. Cook, "Cross-validation of Regression Models," *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 575–583, 1984.
- [103] J. Shao, "Linear model selection by cross-validation," *Journal of the American Statistical Association*, vol. 88, no. 422, pp. 486–494, 1993.
- [104] M. A. Tanner and W. H. Wong, "The calculation of posterior distributions by data augmentation: Rejoinder," *Journal of the American Statistical Association*, vol. 82, no. 398, p. 548, 1987.
- [105] L. Song, "Learning to cope with small noisy data in software effort estimation," Ph.D. dissertation, School of computer science, The University of Birmingham, Birmingham, United Kingdom, 2019.

- [106] V. Tawosi, R. Moussa, F. Sarro, "Deep Learning for Agile Effort Estimation Have We Solved the Problem Yet?," 2022, arXiv preprint arXiv:2201.05401.
- [107] L. Song, L. L. Minku, and X. Yao, "A novel automated approach for software effort estimation based on data augmentation," Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018.
- [108] W.M. Brown, T.D. Gedeon and D.I. Groves, "Use of Noise to Augment Training Data: A Neural Network Method of Mineral–Potential Mapping in Regions of Limited Known Deposit Examples," Natural Resources Research 12, 141–152, 2003. <https://doi.org/10.1023/A:1024218913435>.
- [109] C. Figura and T. M. Khoshgoftaar, "A survey on image data augmentation for Deep Learning," Journal of Big Data, vol. 6, no. 1, 2019.
- [110] K. M. Rashid and J. Louis, "Times-series data augmentation and deep learning for construction equipment activity recognition," Advanced Engineering Informatics, vol. 42, p. 100944, 2019.
- [111] L. A. Kappelman, R. McKeeman, and L. Zhang, "Early warning signs of IT project failure: The dominant dozen," Information Systems Management, vol. 23, no. 4, pp. 31–36, 2006.
- [112] M. Mohri, A. Rostamizadeh, and A. Talwalkar, Foundations of Machine Learning. Cambridge, MA: The MIT Press, 2018.
- [113] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," Statistics Surveys, vol. 4, no. none, 2010.
- [114] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning. Cambridge, MA: The MIT Press, 2017.
- [115] G. Zhang, Y. Liu, and X. Jin, "A survey of autoencoder-based Recommender Systems," Frontiers of Computer Science, vol. 14, no. 2, pp. 430–450, 2019.
- [116] G. Claeskens and N. L. Hjort, Model selection and model averaging. Cambridge: Cambridge University Press, 2010.
- [117] F. Chollet, Deep learning with python. Shelter Island, NY: Manning, 2021.
- [118] K. A. Alsubhi, "Effort estimation in Agile Software Development using deep learning model," International Journal of Advances in Computer Science and Technology, vol. 8, no. 12, pp. 41–49, 2019.
- [119] M. W. Berry, A. Mohamed, and Y. B. Wah, Supervised and unsupervised learning for Data Science. Cham: Springer, 2020.
- [120] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," IEEE Transactions on Software Engineering, vol. SE-4, no. 4, pp. 345–361, 1978.

- [121] C. F. Kemerer, "An empirical validation of software cost estimation models," *Communications of the ACM*, vol. 30, no. 5, pp. 416–429, 1987.
- [122] C. A. Behrens, "Measuring the productivity of computer systems development activities with function points," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, pp. 648–652, 1983.
- [123] B. W. Boehm, "Software engineering economics," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4–21, 1984.
- [124] J. Rashid, M. W. Nisar, T. Mahmood, A. Rehman, and Y. A. Syed, "A study of software development cost estimation techniques and Models," *April 2020*, vol. 39, no. 2, pp. 413–431, 2020.
- [125] G. Liu and L. Lavazza, "Early and quick function points analysis: Evaluations and proposals," *Journal of Systems and Software*, vol. 174, p. 110888, 2021.
- [126] D. B. Bock and R. Klepper, "FP-S: A simplified function point counting method," *Journal of Systems and Software*, vol. 18, no. 3, pp. 245–254, 1992.
- [127] G. Horgan, S. Khaddaj, and P. Forte, "Construction of an FPA-type metric for early lifecycle estimation," *Information and Software Technology*, vol. 40, no. 8, pp. 409–415, 1998.
- [128] F. Marir and I. Watson, "Case-based reasoning: A categorized bibliography," *The Knowledge Engineering Review*, vol. 9, no. 4, pp. 355–381, 1994.
- [129] F. Marir and I. Watson, "Case-based reasoning: A review," *The Knowledge Engineering Review*, vol. 9, no. 4, pp. 327–354, 1994.
- [130] K. Srinivasan and D. Fisher, "Machine learning approaches to estimating software development effort," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 126–137, 1995.
- [131] G. Karner, "Resource estimation for objectory projects," *Objective Systems SF AB*, 17(1):9, 1993.
- [132] M. Azzeh, A. Bou Nassif, and I. B. Attili, "Predicting software effort from use case points: A systematic review," *Science of Computer Programming*, vol. 204, p. 102596, 2021.
- [133] M. Arora, S. Verma, Kavita, and S. Chopra, "A systematic literature review of machine learning estimation approaches in scrum projects," *Cognitive Informatics and Soft Computing*, pp. 573–586, 2020.
- [134] T. Hastie, J. Friedman, and R. Tibshirani, *The elements of Statistical Learning: Data Mining, Inference, and prediction*. New York: Springer, 2017.
- [135] P. Hohl, J. Klünder, A. van Bennekum, R. Lockard, J. Gifford, J. Münch, M. Stupperich, and K. Schneider, "Back to the future: Origins and directions of the 'Agile Manifesto' – views of the originators," *Journal of Software Engineering*

Research and Development, vol. 6, no. 1, 2018.

- [136] Atlassian, "Atlassian JIRA agile software," 2016. [Online]. Available: <https://www.atlassian.com/software/jira>
- [137] B. Akkaya, "The Effect of Recursive Feature Elimination with Cross-Validation Method on Classification Performance with Different Sizes of Datasets," IV International Conference on Data Science and Applications, 2021
- [138] D. P. Kingma, J. L. Ba, Kingma DP, Ba J., "Adam: A method for stochastic optimization," 2014, arXiv preprint arXiv:1412.6980.
- [139] J. Rashid, M. W. Nisar, T. Mahmood, A. Rehman, and Y. A. Syed, "A study of software development cost estimation techniques and models," Mehran University Research Journal of Engineering and Technology, April 2020, vol. 39, no. 2, pp. 413–431, 2020.
- [140] H. Rastogi, S. Dhankhar, and M. Kakkar, "A survey on software effort Estimation Techniques," 2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence), 2014.
- [141] Monika and O. P. Sangwan, "Software effort estimation using Machine Learning Techniques," 2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence, 2017.
- [142] S. P. Pillai, S. D. Madhukumar, and T. Radharamanan, "Consolidating evidence based studies in software cost/effort estimation — a tertiary study," TENCON 2017 - 2017 IEEE Region 10 Conference, 2017.
- [143] A. Saeed, W. H. Butt, F. Kazmi, and M. Arif, "Survey of software development effort Estimation Techniques," Proceedings of the 2018 7th International Conference on Software and Computer Applications, 2018.
- [144] P. Sharma and J. Singh, "Systematic literature review on software effort estimation using machine learning approaches," 2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS), 2017.
- [145] C. Chatfield, Time-series forecasting. Boca Raton, FL: Chapman & Hall/CRC, 2001.
- [146] T. Munakata, Fundamentals of the new artificial intelligence neural, evolutionary, Fuzzy and more. London: Springer, 2007.
- [147] J. Wang, B. Hafidh, H. Dong, and A. El Saddik, "Sitting posture recognition using a spiking neural network," IEEE Sensors Journal, vol. 21, no. 2, pp. 1779–1786, 2021.
- [148] M. Xiao, Q. Meng, Z. Zhang, Y. Wang, and Z. Lin, "Spide: A purely spike-based method for training feedback spiking neural networks," Neural Networks, vol. 161, pp. 9–24, 2023.



Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE EXAMEN DE GRADO

No. 00103

Matrícula: 2202800353

Hacia un método para la estimación automática de esfuerzo en el desarrollo de software a partir de historias de usuario.

En la Ciudad de México, se presentaron a las 17:00 horas del día 10 del mes de marzo del año 2023 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DR. RENE MAC KINNEY ROMERO
DR. HUMBERTO GUSTAVO CERVANTES MACEDA
DRA. SONIA GUADALUPE MENDOZA CHAPA
DR. EDUARDO FILEMON VAZQUEZ SANTACRUZ



Eduardo

EDUARDO RODRIGUEZ SANCHEZ
ALUMNO

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron para proceder al Examen de Grado cuya denominación aparece al margen, para la obtención del grado de:

MAESTRO EN CIENCIAS (CIENCIAS Y TECNOLOGIAS DE LA INFORMACION)

DE: EDUARDO RODRIGUEZ SANCHEZ

y de acuerdo con el artículo 78 fracción III del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:

aprobar

Acto continuo, el presidente del jurado comunicó al interesado el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

DIRECTOR DE LA DIVISIÓN DE CBI

Roman Linares Romero
DR. ROMAN LINARES ROMERO

PRESIDENTE

Rene Mac Kinney Romero
DR. RENE MAC KINNEY ROMERO

VOCAL

Humberto Gustavo Cervantes Maceda
DR. HUMBERTO GUSTAVO CERVANTES
MACEDA

VOCAL

Sonia Guadalupe Mendoza Chapa
DRA. SONIA GUADALUPE MENDOZA CHAPA

SECRETARIO

Eduardo Filemon Vazquez Santacruz
DR. EDUARDO FILEMON VAZQUEZ
SANTACRUZ