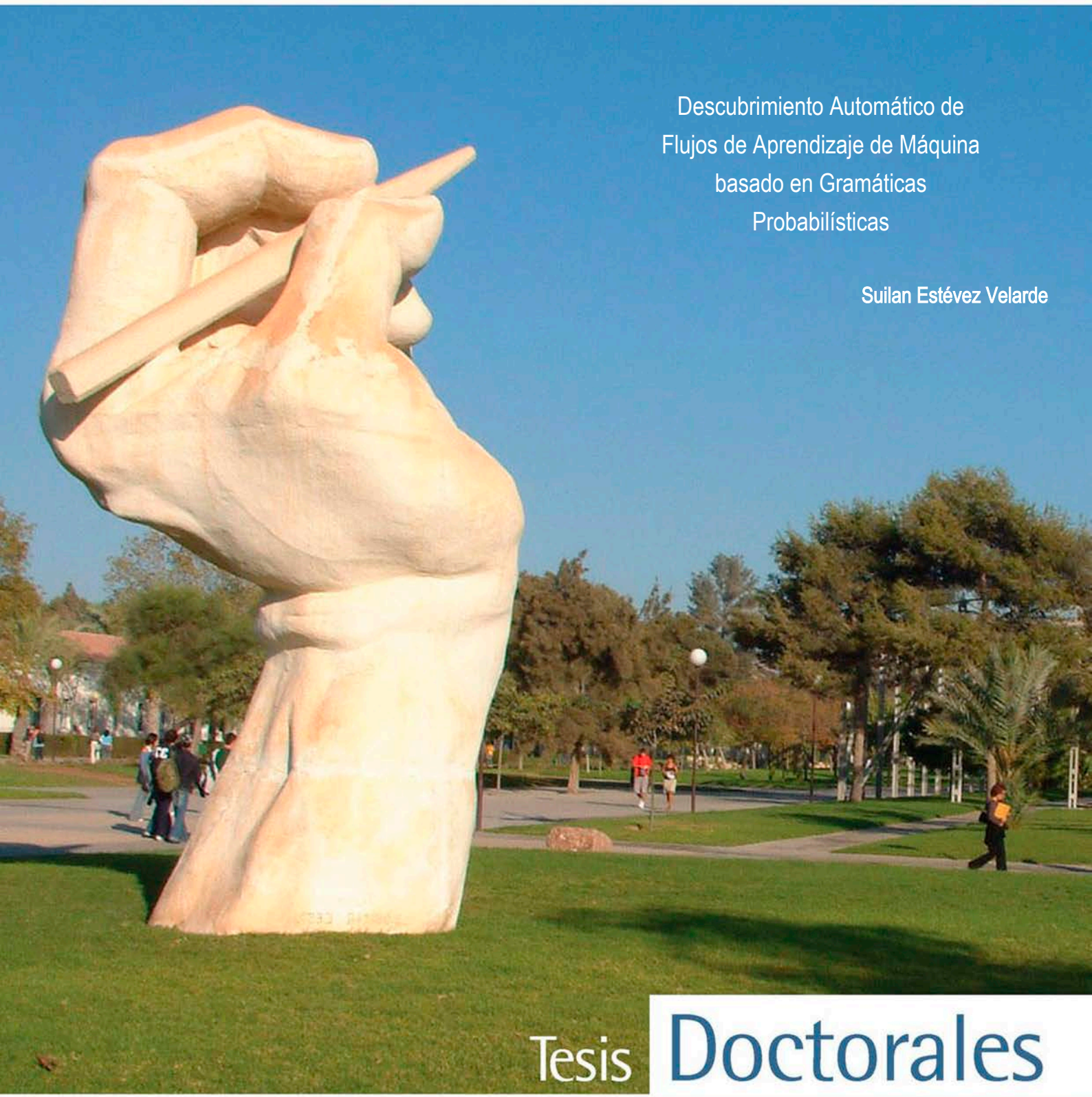
Universitat d'Alacant
Universidad de Alicante

Descubrimiento Automático de Flujos de Aprendizaje de Máquina basado en Gramáticas Probabilísticas

Suilan Estévez Velarde



Tesis Doctorales
UNIVERSIDAD de ALICANTE

Unitat de Digitalització UA
Unidad de Digitalización UA

Universitat d'Alacant
Universidad de Alicante

Departamento de Lenguajes y Sistemas Informáticos
Escuela Politécnica Superior

Descubrimiento Automático de Flujos de Aprendizaje de Máquina basado en Gramáticas Probabilísticas

Suilan Estévez Velarde

Tesis presentada para aspirar al grado de

DOCTOR POR LA UNIVERSIDAD DE ALICANTE
DOCTORADO EN INFORMÁTICA

Dirigida por

Dr. Yoan Gutierrez Vazquez
Dr. Juan Andrés Montoyo Guijarro
Dr. Yudivián Almeida Cruz

Esta tesis ha sido financiada por la Fundación Carolina, la Universidad de Alicante y la Universidad de La Habana.

TESIS DOCTORAL EN FORMA DE COMPENDIO DE PUBLICACIONES

Descubrimiento Automático de Flujos de Aprendizaje de Máquina basado en Gramáticas Probabilísticas

El presente documento contiene una síntesis del trabajo realizado por Suilan Estévez Velarde, bajo la dirección de Dr. Yoan Gutiérrez Vázquez, Dr. Juan Andrés Montoyo Guijarro y Dr. Yudivián Almeida Cruz, para optar por el grado de Doctor en Informática. Se presenta en la Universidad de Alicante y se estructura según la normativa establecida para la presentación de tesis doctorales en forma de compendio de publicaciones: una primera parte con una síntesis, una segunda parte que reproduce las publicaciones científicas realizadas y una tercera parte con las conclusiones.

"Todos los modelos están equivocados pero algunos son útiles."
— George E.P. Box

Espero que este sea uno de esos modelos útiles.
Dedicado a todos los que hicieron posible este sueño,
en especial a mi esposo que lo ha construido junto conmigo.

Agradecimientos

Ahora que llego al final de este recorrido parece fácil e incluso se siente divertido. Siento que crecí y que mejoré la persona que era antes de este doctorado. Además siento que tengo herramientas para enfrentarme al mundo y puertas abiertas que antes no estaban. Y me doy cuenta que que nada de esto sería posible sin todas las personas que me han ayudado y que han estado para mí a cada paso del camino. En estas líneas quiero agradecerles no solo por este trabajo, sino por cuidar y creer en mí, por volverme mejor, porque al compartir con ustedes parte del camino siento que dejaron una huella que no voy a olvidar.

A mi familia que siempre me ha apoyado, a los que están siempre que son muchos porque mi familia es muy grande y siempre está unida. Gracias a mi madre y mi hermana que siempre son un apoyo emocional y siempre están pendientes de los pequeños detalles de la vida, a mi padre por compartir conmigo numerosas charlas de Inteligencia Artificial y por desde pequeña fomentarme el gusto por adquirir nuevos conocimientos, por ser mi ejemplo desde bien pequeña y mostrarme que con esfuerzo se logran muchas cosas y que un trabajo es para disfrutarlo. A mi familia elegida Tania, Alejandro, Alenia, Sofía y Andy con los que siempre puedo contar y con los que tanto me divierto. A mi abuelito que no está pero que estuvo conmigo parte del camino, a mis abuelos que si están, Estela y Gregorio, que incluso ven documentales para saber de que estoy hablando. A mis tíos y mis primos que me apoyan y que me ayudan en los momentos más insospechados. A Tony y Bety que siempre me animan en mis decisiones, a Alain y Yahima que para mí son casi hermanitos. A Glen, Ernesto, Sonia, Chuchi, Ivon, Vivian, Eric, Magui, Jorge, Isora, Ricardo...a la familia grande que seguro comparte y celebra este logro conmigo.

A mis tutores... realmente no puedo agradecerles lo suficiente por el

trabajo, las horas, la confianza. Por dejarme ser yo misma y a la vez hacerme mejor, por inspirarme y lograr que quiera ser como ustedes. Les doy las gracias por darme la oportunidad de conocerlos y trabajar a su lado. A Yudivián que ha estado formándome desde pequeñita cuando no sabía ni que es la IA, posiblemente sin tu influencia no fuera la persona que soy hoy, ni tuviera la confianza que tengo. Realmente no sé que decirte porque sin tí ni siquiera hubiera empezado el doctorado. A Yoan que siempre tuvo el tiempo de revisar cada pequeño detalle de cada artículo y que se preocupó por mí y no solo por el resultado. A Andrés que me motiva con su entusiasmo y que siempre encuentra interesante las nuevas ideas y cómo mejorarlas y aplicarlas. A Rafa que aunque no es mi tutor he aprendido muchímo con él. Muchas gracias por darme esta oportunidad, espero que tengamos la oportunidad de trabajar juntos en el futuro porque les prometo seguir intentando ser mejor.

A mis compañeros de trabajo y amigos que me ayudaron en Alicante. A Bea, Lea, Marta e Isa con las que pasé buenísimos momentos en Cuba y en España. A Antonio, Estela, Patricio, Mario, Alba, Victor, María de los Ángeles, Sarai, Cristina por los momentos buenos que compartimos y el trabajo juntos. A Alejandro y su familia por ayudarme tanto y acogerme y malcriarme como uno más de su familia. A Tania por todas sus correcciones y todas las charlas en inglés. A Jose Manuel y Sonia por todas las discusiones interesantes y momentos entretenidos. A Robier, Aylin, Jose Ignacio, Yoan y su familia por hacerme sentir en Cuba incluso estando en España. A Gonzalo y Aurora por mostrarme Andalucía y sus encantos. A Carlos, Lorena y Andrés por enseñarme a vivir las fiestas, a respetarlas y admirarlas.

A mis compañeros de la Universidad de La Habana que sé que me protegen y me cuidan y se preocupan por mi futuro. Al Guino, por ser el mejor decano, preocuparse por la formación de los jóvenes y siempre tener tiempo para atenderme. A Carmen, Wilfre, Alberto, PQR, Dafne, Fernando, Marta, Danay, Genry, Idania, Aimee, Alina, Katrib, Ricardo ... y seguro que se me olvidan muchos a los que he molestado por infinidad de temas y me siguen apoyando. A Luciano, que lo admiro y lo quiero muchímo y he aprendido Filosofía e IA a partes iguales en nuestras conversaciones y espero aún aprender mucho más.

A mi equipo .... Ustedes son indispensables. A Hian que haces magia para que las cosas funcionen. A Daniel por entenderme y compartir conmigo la

forma de ver el mundo. A Sadan que incluso estando fuera de la UH sabemos que podemos contar contigo. A Juan Pablo, sé que serás doctor mucho más rápido, me alegro de haber compartido juntos tantas comidas. A Rocío por ayudar siempre y tener siempre una sonrisa. A Estevanell por todo lo que trabajaste en AutoGOAL y todas las consultas a media noche. Sin ustedes no me atrevería a querer cambiar el mundo. No les puedo explicar cuanta confianza tengo en ustedes y en su trabajo y cuanto me alegro por ustedes de cada cosa que les sale bien. Estoy muy orgullosa de formar parte de un equipo en el que están.

A mis estudiantes que les encanta la IA y hacen tantas preguntas y me obligan a aprender más para responderles. A los miembros del grupo, a los dedicaron tiempo a anotar corpus, a los que están ya aprendiendo a escribir artículos cintíficos que sepan que les deseo que se lo pasen tan bien como me lo he pasado en mi doctorado.

A todos los que he tenido la suerte de llamar amigos y que muchas veces me han dado su apoyo y comprensión cuando no les he podido dedicar todo el tiempo y la atención que merecen. A Mariana y su niña, Claupa, Borrero y Mirella, Heydi, David, Ezequiel, Thomas, Pablo, Silvio, Ian Pedro, y Charly. A los casi no veo, Hayden, Andy, Bolufé, Pepe, muchas gracias por todo lo que hemos compartido. A mis amigas que siempre están cuando las necesito, a cualquier hora, no importa la distancia, sé que siempre puedo contar con ustedes. A Amalia, Elizabeth y Gabriela por estar siempre conmigo y no molestarse nunca por el tiempo que le dedico al trabajo. Por toda su paciencia y los buenos consejos por celebrar conmigo cuando estoy alegre y soportarme cuando estoy triste.

Por encima de todo hay un agradecimiento que se me hace especial. A Ale que estuvo conmigo en cada línea de código, en cada paper, en cada discusión filosófica. A ti que entiendes mis ideas cuando todavía están enredadas y les tienes confianza cuando no son más que sueños. Tú eres más en este caso que un amigo, un compañero de trabajo o un esposo, a pocos les toca tanto trabajo junto. No solo me apoyaste sino que hoy puedo decir que me lo he pasado estupendamente, lograste que todo fuera divertido y por muchas horas de trabajo que hicieramos nunca fue aburrido o monótono. Gracias por hacer este doctorado conmigo.

Muchas gracias a todos ....

Los quiero mucho ....

Esta investigación ha sido desarrollada de forma conjunta en la Universidad de Alicante (España) y la Universidad de La Habana (Cuba), entre noviembre de 2017 y septiembre de 2020, en sucesivas estancias de investigación. La tesis ha sido co-financiada por la Fundación Carolina y ambas instituciones anteriormente nombradas. Por la Universidad de Alicante, el Departamento de Lenguajes y Sistemas Informáticos ha soportado esta investigación a través de los proyectos SIIA (`PROMETEO/2018/089`, `PROMETEU/2018/089`) y LIVING-LANG (`RTI2018-094653-B-C22`). Por la Universidad de La Habana, la Facultad de Matemática y Computación y el Departamento de Inteligencia Artificial y Sistemas Computacionales han soportado esta investigación.

Resumen

El aprendizaje de máquinas ha ganado terreno utilizándose en casi todas las áreas de la vida cotidiana, ayudando a tomar decisiones en las finanzas, la medicina, el comercio y el entretenimiento. El desarrollo continuo de nuevos algoritmos y técnicas de aprendizaje automático, y la amplia gama de herramientas y conjuntos de datos disponibles han traído nuevas oportunidades y desafíos para investigadores y profesionales tanto del mundo académico como de la industria. Seleccionar la mejor estrategia posible para resolver un problema de aprendizaje automático es cada vez más difícil, en parte porque requiere largos tiempos de experimentación y profundos conocimientos técnicos.

En este escenario, el campo de investigación *Automated Machine Learning* (AutoML) ha ganado protagonismo, proponiendo estrategias para automatizar progresivamente tareas usuales durante el desarrollo de aplicaciones de aprendizaje de máquina. Las herramientas de AutoML más comunes permiten seleccionar automáticamente dentro de un conjunto restringido de algoritmos y parámetros la mejor estrategia para cierto conjunto de datos. Sin embargo, los problemas prácticos a menudo requieren combinar y comparar algoritmos heterogéneos implementados con diferentes tecnologías subyacentes. Un ejemplo es el procesamiento del lenguaje natural, un escenario donde varía el espacio de posibles técnicas a aplicar ampliamente entre diferentes tareas, desde el preprocesamiento hasta la representación y clasificación de textos. Realizar AutoML en un escenario heterogéneo como este es complejo porque la solución necesaria podría incluir herramientas y bibliotecas no compatibles entre sí. Esto requeriría que todos los algoritmos acuerden un protocolo común que permita la salida de un algoritmo para ser compartida como entradas a cualquier otro.

En esta investigación se diseña un nuevo paradigma de AutoML que utiliza

técnicas heterogéneas. A diferencia de los enfoques de AutoML existentes, nuestra contribución puede combinar técnicas y algoritmos de diferentes bibliotecas y tecnologías, incluidos algoritmos de aprendizaje de máquina clásicos, extracción de características, herramientas de procesamiento de lenguaje natural y diversas arquitecturas de redes neuronales. Definimos el problema heterogéneo de optimización de AutoML como la búsqueda de la mejor secuencia de algoritmos que transforme datos de entrada específicos en la salida deseada. Esto proporciona un enfoque teórico y práctico novedoso para AutoML. Nuestra propuesta se evalúa experimentalmente en diversos problemas de aprendizaje automático y se compara con enfoques alternativos, lo que demuestra que es competitiva con otras alternativas de AutoML en los puntos de referencia estándar. Además, se puede aplicar a escenarios novedosos, como varias tareas de procesamiento de lenguaje natural, donde las alternativas existentes no se pueden implementar directamente. Una implementación concreta de este paradigma está disponible de forma gratuita e incluye compatibilidad incorporada con una gran cantidad de marcos de aprendizaje automático populares, lo que hace que nuestro enfoque sea útil para resolver problemas prácticos con relativa facilidad y esfuerzo.

El uso de la herramienta propuesta en esta investigación permite a los investigadores y profesionales desarrollar rápidamente algoritmos de referencia optimizados en diversos problemas de aprendizaje automático. En algunos escenarios, la solución proporcionada por nuestro sistema podría ser suficiente. Sin embargo, los sistemas AutoML no deben intentar reemplazar a los expertos humanos, sino servir como herramientas complementarias que permitan a los investigadores obtener rápidamente mejores prototipos y conocimientos sobre las estrategias más prometedoras en un problema concreto. Las técnicas de AutoML abren las puertas a revolucionar la forma en que se realiza la investigación y el desarrollo del aprendizaje automático en la academia y la industria.

Abstract

Machine learning has gained traction, being used in almost every area of everyday life, helping to make decisions in finance, medicine, commerce and entertainment. The continuous development of new algorithms and machine learning techniques, and the extensive range of tools and data sets available have brought new opportunities and challenges for researchers and professionals from both the academic and industrial world. Selecting the best possible strategy to solve a machine learning problem is increasingly difficult in part because it requires long experimentation times and deep technical knowledge.

In this scenario, the *Automated Machine Learning* (AutoML) research field has gained prominence, proposing strategies to progressively automate the usual tasks during the development of machine learning applications. The most common AutoML tools can automatically select from a restricted set of algorithms and parameters the best strategy for a certain data set. However, practical problems often require combining and comparing heterogeneous algorithms implemented with different underlying technologies. An example is natural language processing, a scenario where the range of possible techniques to be applied varies widely between different tasks, from preprocessing to the representation and classification of texts. Performing AutoML in a heterogeneous scenario like this is complex because the required solution might include tools and libraries that are not compatible with each other. This requires that all algorithms agree to a common protocol that allows the output of one algorithm to be shared as inputs to any other.

In this research, an novel AutoML paradigm is designed, that uses heterogeneous techniques. Unlike existing AutoML approaches, our contribution can combine techniques and algorithms from different libraries and technologies, including classical machine learning algorithms, feature extraction,

natural language processing tools, and various neural network architectures. We define the heterogeneous AutoML optimisation problem as finding the best sequence of algorithms that transforms specific input data into a desired output. This provides a novel theoretical and practical approach to AutoML. Our proposal is experimentally tested on various machine learning problems and compared to alternative approaches, demonstrating that it is competitive with other AutoML alternatives on standard benchmarks. Additionally, it can be applied to novel scenarios, such as various natural language processing tasks, where existing alternatives cannot be implemented directly. A practical implementation of this paradigm is available for free and includes built-in support for a host of popular machine learning frameworks, making our approach useful for solving practical problems with relative ease and effort.

Using the tool proposed in this research allows researchers and practitioners to rapidly develop optimised reference algorithms in various machine learning problems. In some scenarios, the solution provided by our system might be sufficient. However, AutoML systems should not attempt to replace human experts, but rather serve as complementary tools that researchers can quickly obtain better prototypes and knowledge about the most promising strategies in a concrete. AutoML techniques open the door to revolutionising the way machine learning research and development is done in academia and industry.

Índice general

I Síntesis de la Tesis **1**

1. Introducción **1**

1.1. Trabajos Relacionados 4

1.2. Motivación 6

1.3. Definición Formal de AutoML Heterogéneo 7

1.4. Problema Científico 9

1.5. Objetivos 10

1.6. Contribuciones 10

1.7. Estructura de la Tesis 11

2. Estado del Arte **15**

2.1. Definición del problema de AutoML 18

2.1.1. Optimización de parámetros 18

2.1.2. Selección de Modelos 19

2.1.3. AutoML como problema CASH 20

2.2. Componentes del Proceso de AutoML 20

2.2.1. Espacio de Búsqueda 21

2.2.2. Estrategia de Búsqueda 23

2.2.3. Estrategias de Estimación de Rendimiendo 27

2.3. Comparación entre Sistemas 29

2.3.1. Análisis de Características Internas 31
2.3.2. Análisis de Características Externas 36
2.4. Sistemas AutoML 47
2.4.1. Sistemas de AutoML genéricos 47
2.4.2. Sistemas de NAS 81
2.4.3. Sistemas Comerciales 93
2.5. Conclusiones 98

3. Paradigma de AutoML Heterogéneo **101**
3.1. Arquitectura General 103
3.1.1. Interfaz de Alto Nivel (*High-Level API*) 105
3.1.2. Interfaz de Bajo Nivel (*Low-Level API*) 106
3.2. Diseño Algorítmico 108
3.2.1. Construcción del espacio de búsqueda 109
3.2.2. Proceso de Optimización 110
3.3. Implementación Computacional 112
3.3.1. Tipos de Datos 112
3.3.2. Componentes 112
3.3.3. Meta-tipos 114
3.3.4. Biblioteca de Algoritmos 115
3.3.5. Adaptadores y Protocolos 115
3.4. AutoGOAL 116

4. Resultados Experimentales **117**
4.1. Comparación con Sistemas de AutoML 118
4.2. Evaluación en Problemas de Procesamiento de Lenguaje Natural 119
4.3. Evaluación en Problemas de Descubrimiento de Conocimiento 121
4.4. Análisis del Proceso de Optimización 122
4.5. Discusión 123

II Artículos Publicados **127**

5. Modelación de Flujos de Algoritmos. ***Optimizing Natural Language Processing Pipelines: Opinion Mining Case Study*** **129**

6. Optimización de Flujos Jerárquicos. ***AutoML strategy based on grammatical evolution: A case study about knowledge discovery from text*** **131**

7. Modelación con Gramáticas Probabilísticas. ***General-purpose Hierarchical Optimisation of Machine Learning Pipelines with Grammatical Evolution*** **133**

8. AutoML Heterógeneo. ***Automatic Discovery of Heterogeneous Machine Learning Pipelines: An Application to Natural Language Processing*** **135**

9. Aplicación AutoGOAL. ***Demo Application for the AutoGOAL Framework*** **137**

III Conclusiones y Recomendaciones **139**

10. Conclusiones **141**

10.1. Publicaciones 143

11. Trabajo Futuro **145**

Índice de figuras

2.1. Ejemplo ilustrativo de un flujo abstracto de algoritmos, en el cual se pueden observar diferentes opciones en cada uno de los pasos que producen resultados difrentes para una métrica determinada(e.g, precisión, recobrado, etc.) 16

3.1. Arquitectura general de AutoGOAL. 103

3.2. Ejemplo de código fuente para ejecutar AutoGOAL en un conjunto de datos específico, en este caso, un problema de PLN. 106

3.3. Ejemplo de definición de adaptadores para algoritmos de *scikit-learn*. 107

3.4. Ejemplo de definición de un componente para el cómputo de vectores de *embedding*. 108

3.5. Ejemplo de definición de un componente para extraer el resumen de un artículo de Wikipedia. 108

3.6. Representación visual del proceso de creación y muestreo del espacio de búsqueda. 110

3.7. Tipos semánticos definidos en AutoGOAL 113

4.1. Análisis del proceso de experimentación 123

Índice de tablas

1.1. Estado del arte en AutoML 5

2.1. Comparación de sistemas de AutoML genéricos en cuanto a sus características internas. 35
2.2. Comparación de sistemas de AutoML para NAS en cuanto a sus características internas. 37
2.3. Comparación de sistemas de AutoML comerciales en cuanto a sus características internas. 38
2.4. Comparación de sistemas de AutoML genéricos en cuanto a sus características externas. 42
2.5. Comparación de sistemas de AutoML para NAS en cuanto a sus características externas. 44
2.6. Comparación de sistemas de AutoML comerciales en cuanto a sus características externas. 46

4.1. Resultados experimentales de AutoGOAL 118
4.2. Comparación de AutoGOAL con los resultados oficiales del corpus HAHA. 120
4.3. Comparación de AutoGOAL con los resultados oficiales del corpus MEDDOCAN. 121
4.4. Comparación de enfoques en el desafío eHealth-KD. Solo se consideran los investigadores que participaron en el Escenario 1. 122

Parte I

Síntesis de la Tesis

Capítulo 1

Introducción

En la última década, la disponibilidad masiva de datos y el aumento del poder computacional de los ordenadores ha impulsado el desarrollo de la inteligencia artificial, y específicamente del aprendizaje automático, a tareas que hasta hace algunos años eran dominio absoluto de los seres humanos. Las técnicas de aprendizaje automático se están utilizando en la academia para expandir la frontera del conocimiento humano, para extraer conocimiento a partir del análisis de grandes volúmenes de datos. De igual forma, son varios los sectores de la industria que están aplicando técnicas de inteligencia artificial de forma intensiva, desde la optimización de grandes almacenes en la venta minorista (p.e., Amazon [1], Walmart [2]), hasta la clasificación y organización de contenido multimedia (p.e., Netflix [3], YouTube [4]). A escala personal, estas tecnologías tienen el potencial de mejorar la calidad de vida humana, por ejemplo, a través de tratamientos médicos más precisos y personalizados, a partir de datos suministrador continuamente por dispositivos electrónicos como teléfonos y relojes inteligentes. A escala global, pueden representar un punto de inflexión en la solución de problemas como el cambio climático, por ejemplo automatizando las flotas de transporte urbano.

La inteligencia artificial es uno de los pilares fundamentales de la cuarta revolución industrial [5]. Aplicarla al sector empresarial aportaría beneficios variados en múltiples escenarios, entre ellos: automatizar procesos; potenciar las tareas creativas; reducir el error humano; reducir los tiempos empleados en análisis de datos; hacer mantenimiento predictivo; mejorar la toma de

decisiones a nivel de producción y de negocios; controlar y optimizar los procesos productivos; y aumentar de la calidad de la producción [6]. Por este motivo la demanda de especialistas en esta área ha aumentado continuamente en los últimos años. Esto provoca un aumento de ofertas educativas en programas de pregrado y postgrado en áreas relacionadas, así como en el crecimiento masivo de las conferencias de inteligencia artificial, históricamente destinadas a una comunidad específica.

Una de las ramas de la inteligencia artificial de mayor utilidad es la denominada aprendizaje automático (en inglés *machine learning*) que consiste en el desarrollo de sistemas computacionales capaces de aprender automáticamente a partir de datos [7]. Justamente por su complejidad, la formación de un especialista en este campo requiere de años de entrenamiento en temas que varían desde la teoría matemática subyacente hasta la ingeniería de software necesaria para desplegar sistemas de aprendizaje automático eficientes y escalables en la práctica. Esto provoca por un lado una barrera de entrada para las empresas e instituciones que no cuentan con suficiente personal preparado. Por otro lado, se incrementa la motivación para acercar las técnicas más avanzadas de aprendizaje automático a usuarios con menos nivel de experticia.

Así, más allá de la posible carencia de expertos en aprendizaje automático, incluso en la academia y en las empresas donde estos expertos se desenvuelven, existen barreras importantes para aprovechar al máximo estas técnicas. En la mayoría de los problemas se pueden aplicar múltiples algoritmos de aprendizaje automático, que a su vez tienen parámetros que deben ser configurados por los especialistas para un funcionamiento óptimo. La diferencia entre un resultado de alta precisión y un resultado inservible puede ser el escoger incorrectamente un algoritmo, o el valor de un parámetro interno, como la cantidad de neuronas en una capa, una función de activación, o un coeficiente de regularización. La tarea de explorar todas las combinaciones posibles de algoritmos y parámetros para un problema dado puede ser impracticable, ya que el número de posibilidades es a menudo exponencial con respecto al número de pasos en una solución [8]. Además, cuando algunos algoritmos tienen parámetros numéricos (discretos o continuos) es matemáticamente imposible evaluar todas las combinaciones. Este problema se complica aún más por el hecho de que cada experimento puede tener una alta complejidad computacional (por ejemplo, entrenar desde cero una red neuronal en un

conjunto de datos extenso). Los investigadores en este campo a menudo se ven obligados a seleccionar un pequeño número de posibilidades basándose en la experiencia previa y el conocimiento del dominio sobre el problema en cuestión.

Ante estas problemáticas, en los últimos años ha comenzado a crecer significativamente el interés en una nueva área del aprendizaje automático denominada *Automated Machine Learning*, o AutoML. El AutoML engloba el diseño de técnicas para automatizar y facilitar todo el proceso de implementación, experimentación y despliegue de algoritmos de aprendizaje automático, desde la selección de qué algoritmos utilizar, hasta el ajuste de sus parámetros internos [9]. Este campo se enfoca, por un lado, en reducir considerablemente el tiempo necesario para la experimentación y el ajuste de parámetros a partir de procedimientos de optimización. Por otro lado, permite acercar a los no-expertos a estas tecnologías al brindar interfaces simplificadas a los algoritmos más complejos. El objetivo fundamental es intercambiar el tiempo del experto por el tiempo de cómputo, automatizando todas las tareas de bajo nivel posibles, y dejando al experto las tareas de más alto nivel, relacionadas con elegir los enfoques y modelos más apropiados para un problema.

La creciente importancia de este campo se evidencia, por una parte, en el aumento de las publicaciones relacionadas en los congresos más importantes de inteligencia artificial, y por otra parte en el marcado interés de la industria por aplicarla.[1] Un ejemplo reciente es la edición del *International Conference on Machine Learning* del año 2020[2], uno de los eventos más grandes de este campo que recogió más de 30 artículos en esta temática específica. Además, en la feria industrial adjunta a este evento[3] las compañías tecnológicas más importantes, incluyendo Amazon, IBM, Baidu y Huawei, presentaron nuevos productos específicamente orientados al AutoML. En estos momentos ya la mayoría de los proveedores de servicios en la nube, incluyendo Google [10], Amazon [11] y Microsoft [12], ofrecen servicios en demanda de inteligencia artificial basados en técnicas de AutoML. El desarrollo del AutoML en la inteligencia artificial promete traer un salto de productividad similar al que ocurrió con la llegada de los lenguajes de programación de alto nivel,

[1] https://www.stateof.ai/

[2] https://icml.cc/Conferences/2020

[3] https://icml.cc/Expo/Conferences/2020/Expo

permitiendo a toda una generación nueva de programadores hacer uso de herramientas hasta entonces demasiado complejas.

1.1. Trabajos Relacionados

A pesar de su corto tiempo de existencia, el campo de AutoML ya cuenta con herramientas establecidas para la automatización de varios sub-dominios de problemas dentro del aprendizaje automático. El Capítulo 2 presenta un análisis detallado del marco teórico y el estado del arte de este dominio. La presente sección, en cambio, se centra en las bibliotecas de código abierto que sirven de soporte a este tipo de sistemas y se usan tanto en la academia como en la industria. En el marco de la investigación realizada en esta Tesis se han desarrollado varios enfoques de este corte, que serán explicadas con detalle en los capítulos siguientes. En esta sección se presenta una comparativa a muy alto nivel entre estas propuestas y las principales alternativas existentes en el estado del arte. La comparación se realiza en términos de las mismas características en las que se enfoca esta investigación.

La Tabla 1.1 contrasta varios sistemas de AutoML existentes con los sistemas propuesto en esta investigación en términos de sus capacidades para tratar con escenarios heterogéneos. Esta evaluación no intenta comparar los sistemas AutoML en términos de su rendimiento general, capacidad o aplicabilidad, sino con respecto a la cuestión específica de tratar directamente con algoritmos de múltiples bibliotecas y orientados a distintos dominios. Varios sistemas AutoML populares se basan en bibliotecas de aprendizaje automático específicas —como Auto-Sklearn [13], Auto-Weka [14] y Auto-Keras [15]— que restringen su uso a los escenarios para los que están diseñadas las bibliotecas subyacentes (por ejemplo, aprendizaje supervisado en el caso de Auto-Sklearn). Otros enfoques, como RECIPE [16] e Hyperopt [17], tienen diseños extensibles que, en principio, les permiten incluir cualquier biblioteca de aprendizaje automático, pero sus implementaciones actuales se basan en tecnologías específicas. Por otro lado, los sistemas como TPOT [18] y ML-Plan [19] proporcionan implementaciones concretas de flujos de aprendizaje automático que abarcan más de una biblioteca de aprendizaje automático.

Además de admitir múltiples bibliotecas, se pueden ampliar varios sistemas de AutoML para tratar con más de un tipo de problema de aprendizaje

Sistemas de AutoML	Múltiples bibliotecas	Múltiples Problemas	Probabilístico	Extensible	Descubrimiento Automático	Año
Auto-Weka			✓	✓		2013
Hyperopt	≈	✓	✓	✓		2014
Auto-Sklearn			✓	✓		2015
Auto-Keras				✓		2016
RECIPE	≈	✓				2017
TPOT	✓					2018
ML-Plan	✓	✓				2018
Desarrollados en esta investigación						
NLP-Opt			✓	✓		2018
HML-Opt	✓	✓	✓	✓		2019
AutoGOAL	✓	✓	✓	✓	✓	2020

Tabla 1.1: Comparación de varios sistemas AutoML existentes con las propuestas de esta Tesis en términos de sus capacidades para tratar problemas heterogéneos de aprendizaje automático. Las entradas marcadas con ≈ indican que el diseño del sistema habilita la capacidad dada pero no está implementada.

automático, con las métricas de evaluación adecuadas. Como ejemplo, Hyperot es un sistema de ajuste de hiperparámetros de propósito general que proporciona bloques de construcción de bajo nivel que se pueden adaptar a cualquier flujo de aprendizaje automático. Sin embargo, la mayoría de los sistemas que proporcionan este nivel de flexibilidad requieren un alto grado de personalización y no admiten un descubrimiento automático de las flujos relevantes para un problema determinado. Esta flexibilidad también tiene un precio en términos de extensibilidad. Los sistemas como Auto-Sklearn y Auto-Keras se benefician de una API subyacente unificada. Esto permite a

los investigadores ampliar las capacidades del sistema AutoML simplemente ampliando la API subyacente, sin tener que tratar con los detalles de implementación del sistema AutoML. En contraste, los sistemas más flexibles como RECIPE y HML-Opt, ambos basados en gramáticas libres del contexto para describir el espacio de posibles flujos, requieren que los investigadores modifiquen sus gramáticas internas para agregar nuevos algoritmos.

Otra característica interesante para esta investigación es el uso de modelos probabilísticos para describir el espacio de posibles flujos. Los sistemas AutoML basados en la optimización bayesiana (por ejemplo, Auto-Sklearn, Auto-Weka) u optimización evolutiva probabilística (por ejemplo, HML-Opt) construyen una representación interna del espacio de los posibles flujos de algoritmos, que puede interpretarse como la asignación de una distribución de probabilidad a cada flujo particular. Estos sistemas no necesariamente permiten acceder fácilmente a sus modelos internos. Sin embargo, esta característica podría ser útil en sí misma como una descripción del espacio de los flujos de algoritmos para que los investigadores recopilen información adicional al analizar qué regiones de este espacio tienen mayores o menores probabilidades.

1.2. Motivación

Las técnicas actuales de AutoML se centran principalmente en un subconjunto específico de algoritmos, a menudo adaptados a una biblioteca o conjunto de herramientas. Resolver problemas complejos, por otro lado, requiere la combinación de diferentes herramientas que podrían no estar disponibles en una misma biblioteca. Además, los problemas de aprendizaje automático más interesantes no se limitan a encontrar la mejor arquitectura o conjunto de hiperparámetros para un algoritmo dado, sino que implican decisiones de más alto nivel. Por ejemplo, en el procesamiento del lenguaje natural, antes de decidir qué algoritmo de aprendizaje automático utilizar, el investigador debe decidir sobre la representación (p.e., usar *embeddings* y cuáles específicamente), características (p.e., incorporar conocimiento externo), preprocesamiento (p.e., eliminar *stopwords* o aplicar *stemming*), etc. Avanzar hacia la automatización de este proceso requiere herramientas más expresivas que permitan al investigador definir un espacio complejo de algoritmos que incluya desde decisiones de alto nivel hasta rangos de

hiperparámetros de bajo nivel.

En los sistemas de AutoML actuales, una solución se concibe como un flujo compuesto por algoritmos de aprendizaje automático. La cantidad y variedad de algoritmos a ser contemplados en el proceso de búsqueda influye normalmente en el rendimiento del sistema. Además, técnicas específicas de dominios diferentes son regularmente encontradas en bibliotecas de aprendizaje automático distintas, por lo que la compatibilidad con varias bibliotecas se hace crucial. Sin embargo, muchas propuestas de AutoML se centran en presentar un espacio de búsqueda de un tamaño predefinido, y no incluyen algoritmos de dominios específicos (e.g., AutoSklearn, AutoWeka), lo que repercute a la hora de tratar con tareas enfocadas en estos dominios. Estos sistemas son incapaces de tratar con diferentes problemas usando una misma formulación, por ejemplo, en tareas para múltiples dominios que incluyan problemas en imágenes, texto, vídeo y audio de forma simultánea [20]. Este problema se intensifica cuando es necesario considerar procesos que no son parte del flujo clásico de aprendizaje automático, por ejemplo, la adición de características semánticas basadas en conocimiento externo.

De manera similar, muchos de los sistemas actuales no pueden ser directamente desplegados para resolver determinados problemas y requieren modificaciones en su código. En la práctica, esto conlleva la necesidad de expertos para adecuar el sistema (si es posible) o resolver el problema en cuestión, dedicando tiempo para desarrollar una solución. Por estas razones, se hace cada vez más necesaria la creación de sistemas AutoML generales, escalables y que presenten un grado adecuado de flexibilidad. Resumiendo, lo que se pretende es incluir técnicas de la mayor cantidad de dominios posibles, maximizando el número de casos de uso, y esto es esencial para lograr una herramienta que minimice el esfuerzo humano al aplicar técnicas de aprendizaje automático.

1.3. Definición Formal de AutoML Heterogéneo

Gran parte de los sistemas que forman el estado del arte, interpretan el problema del AutoML como un problema de selección combinada de modelos y optimización de hiperparámetros. Este enfoque captura naturalmente sub-problemas del aprendizaje supervisado como la clasificación basada

en características vectoriales. Por otro lado, esta formulación excluye los problemas donde las características son más complejas y el objetivo no es directamente clasificación o regresión, sino por ejemplo, la extracción de entidades o el descubrimiento de conocimiento.

En esta Tesis se propone una extensión al problema clásico de AutoML, denominada *AutoML Heterogéneo*, generalizando los conjuntos de problemas que pueden ser resueltos y las estrategias de solución. A continuación se define formalmente el problema de AutoML Heterogéneo. Esta definición incluye no solo problemas de aprendizaje supervisado sobre datos vectoriales, sino escenarios más generales como la recuperación de información.

Se define A como el espacio de todos los algoritmos de aprendizaje automático "atómicos" —es decir, técnicas específicas como la regresión logística, k-means o representación tf-idf— que pueden intervenir en cualquier proceso de aprendizaje automático. Un algoritmo $a \in A$ se representa para este propósito como una función $a : T_{in} \rightarrow T_{out}$ que asigna a cada entrada $x \in T_{in}$ la salida correspondiente en $y \in T_{out}$. T_i son todos los tipos de datos posibles que manejan los algoritmos de aprendizaje automático, tales como texto en lenguaje natural, vectores de *embeddings*, categorías, palabras, etc. Por ejemplo, los algoritmos de tokenización pueden verse como cualquier $a : T_{in} \rightarrow T_{out} \in A$ tal que T_{in} es `Sentence` y T_{out} es `List[Word]`, donde estos nombres tienen una interpretación semántica que corresponde a su uso habitual en el aprendizaje automático.

Se define S como el espacio de todos los posibles "flujos de algoritmos", i.e., secuencias de algoritmos construidas a partir de A. Un flujo $p =< a^1, \ldots, a^n >\in S$ puede ser visto como un caso especial de algoritmo que aplica cada algoritmo $a^{(i)}$ de forma secuencial a la salida del algoritmo anterior en la secuencia. Formalmente, se puede ver como la composición de los algoritmos correspondientes, es decir, $p(x) = a^n(a^{n-1}(\ldots a^1(x)\ldots))$. Por lo tanto, p tiene como entrada el tipo T^1_{in} y como salida el tipo T^n_{out}. No todos los algoritmos pueden ser combinados de esta forma, sino solo aquellos algoritmos a^i y a^j tales que sus correspondientes tipos de entrada y salida sean compatibles, p.e., $T^i_{out} = T^j_{in}$. En el caso general, es posible definir una función de orden parcial $T_i \leq T_j$ que represente esta noción de compatiblidad entre los tipos, por ejemplo, definida a partir de una jerarquía de tipos. Bajo esta restricción, se define entonces S' como el espacio de todos flujos válidos,

o sea que cumplen las restricciones de compatibilidad entre las respectivas entradas y salidas de algoritmos consecutivos.

Para seleccionar el mejor flujo, se requiere una métrica $\varphi(p) : p \in S'$ que puede evaluar cada flujo y permite la comparación de dos flujos válidos. Finalmente, se define el problema de AutoML heterogéneo como el problema de optimización de encontrar el mejor flujo (dado un arbitrario φ) que transforma una entrada específica $x \in T^*_{in}$ a un deseado $y \in T^*_{out}$. Formalmente:

$$\begin{aligned} \text{argmax} \quad & \{\, \varphi(p) \,|\, p \in S' \,\} \\ \text{s.t:} \quad & p : T^*_{in} \rightarrow T^*_{out} \end{aligned}$$

Esta definición se aplica directamente a los escenarios de aprendizaje supervisado sobre datos vectoriales y tabulares más comunes en tareas de AutoML, a la vez que permite modelar problemas donde la entrada tenga un tipo semántico de más alto nivel, como texto o imágenes. En el aprendizaje supervisado, la entrada consistirá en algunos datos en el espacio de características más el objetivo de predicción, permitiendo una fase de entrenamiento sobre datos supervisados. Además, esta formulación permite modelar problemas donde la salida no es solamente una categoría o valor, sino un conjunto de datos semánticos, por ejemplo entidades y relaciones, que son evaluados por una métrica a optimizar. De esta manera el AutoML Heterogéneo incluye todos los problemas actualmente resueltos por otros sistemas de AutoML y amplía el campo de estudio a problemas de aprendizaje en nuevos dominios.

1.4. Problema Científico

La comunidad científica no dispone de enfoques de AutoML que puedan ser utilizados en escenarios heterogéneos, donde se requiera la combinación de técnicas de múltiples dominios. Los sistemas de AutoML existentes no son aplicables en estos escenarios sin realizar modificaciones significativas para adaptarlos a un problema en concreto. Esto limita la aplicabilidad del AutoML a problemas prácticos en la industria, así como su potencial para acelerar la investigación científica, especialmente en dominios complejos como el procesamiento de lenguaje natural.

1.5. Objetivos

El objetivo general de esta Tesis es el diseño de un paradigma integrador que permita resolver el problema de AutoML Heterogéneo, a partir de la combinación de técnicas de aprendizaje automático, optimización y teoría de lenguajes formales. El paradigma propuesto debe cumplir con los siguientes requerimientos:

- Ser competitivo con el estado del arte en los problemas de AutoML clásicos.
- Ser utilizable por usuarios con diferentes grados de experticia.
- Ser capaz de combinar herramientas de diferentes dominios de forma transparente.
- Ser adaptables a nuevos problemas sin hacer grandes cambios en su implementación computacional.
- Ser flexible para incluir nuevos tipos de algoritmos y técnicas sin necesidad de cambios en la formulación teórica o la implementación computacional.

1.6. Contribuciones

En la consecución del objetivo planteado se desarrolló la presente investigación, logrando las siguientes contribuciones concretas:

1. Se define el problema de AutoML Heterogéneo, que generaliza diferentes formulaciones de problemas de AutoML en un marco conceptual unificado, independiente de tecnologías, técnicas de optimización y dominio de los problemas de aprendizaje subyacentes.
2. Se diseña una estructura basada en gramáticas libres del contexto probabilísticas que representan el espacio de todos los flujos de algoritmos.
3. Se diseñan variantes de metaheurísticas evolutivas para la exploración de espacios complejos que contienen algoritmos e hiperparámetros.

4. Las técnicas desarrolladas en la investigación se agrupan en una herramienta computacional para AutoML Heterogéneo denominada *AutoGOAL*, que propone un diseño extensible y modular, basado en anotaciones de tipo, y permite descubrir automáticamente los flujos válidos para un problema dado, proporcionando solo los tipos de entrada y salida deseados.

5. Se ofrece una extensa experimentación en diferentes escenarios, dominios y espacios de algoritmos, incluyendo problemas estructurados y procesamiento de lenguaje natural, que demuestra la eficacia de las tecnologías desarrolladas en esta investigación.

1.7. Estructura de la Tesis

La presente Tesis está organizada por el sistema de compendio de artículos científicos. Se presentan un total de cinco artículos publicados que engloban todo el trabajo realizado en la consecución de los objetivos definidos en la Sección 1.5. A modo de resumen se presentan dos capítulos que recogen los principales resultados obtenidos, y luego los artículos científicos se organizan en la Parte II. Finalmente se presentan unas conclusiones generales y recomendaciones de trabajos futuros. A continuación se describe en mayor detalle el contenido de cada capítulo.

Parte I presenta el resumen de la Tesis en idioma Español.

Capítulo 2 presenta un análisis del estado de la cuestión en el campo de AutoML. Se definen los principales elementos teóricos que constituyen el objeto de estudio del AutoML y los paradigmas fundamentales que se han desarrollado en esta área. Se presenta además un resumen de las principales herramientas existentes, tanto de código abierto como comerciales, así como una comparación entre un subjconjunto de estas herramientas en términos cualitativos.

Capítulo 3 presenta una síntesis de la propuesta de paradigma para el problema de AutoML heterogéneo, concretada en una implementación computacional (la biblioteca AutoGOAL). Se describen las ideas téoricas fundamentales que aportan a esta solución así como los detalles de implementación.

Capítulo 4 resume los resultados experimentales que comparan la propuesta presentada con otros enfoques de AutoML y con soluciones diseñadas por expertos en 10 problemas de aprendizaje automático diferentes, incluyendo problemas tabulares, clasificación de texto y recuperación de información.

Parte II presenta cinco artículos publicados en el marco de la Tesis.

Capítulo 5 presenta el artículo *Optimizing Natural Language Processing Pipelines: Opinion Mining Case Study* que formula la solución a un problema de aprendizaje y la selección de hiperparámetros como la optimización de un flujo de algoritmos de aprendizaje automático. Este artículo introduce los problemas de procesamiento de lenguaje natural dentro del campo del AutoML, utilizando procesos que no son parte del flujo clásico de AutoML, como preprocesamiento de texto y extracción de características.

Capítulo 6 presenta el artículo *AutoML strategy based on grammatical evolution: A case study about knowledge discovery from text* que define un algoritmo de optimización basado en gramáticas probabilistas que permite optimizar flujos jerárquicos en problemas de aprendizaje automático complejos, donde la salida no es un clasificación o regresión. El algoritmo definido en este artículo permite optimizar en un espacio de flujos jerárquicos con la complejidad suficiente para resolver problemas de AutoML Heterogéneo.

Capítulo 7 presenta el artículo *General-purpose Hierarchical Optimisation of Machine Learning Pipelines with Grammatical Evolution* que propone un lenguaje de gramáticas probabilísticas para la definición de espacios de búsqueda sobre flujos de algoritmos complejos y lo evalúa en la solución de problemas de distinta naturaleza en comparación con otros sistemas de AutoML y soluciones diseñadas por expertos. El API de bajo nivel de AutoGOAL está basado en esta formulación de gramáticas probabilísticas.

Capítulo 8 presenta el artículo *Automatic Discovery of Heterogeneous Machine Learning Pipelines: An Application to Natural Language Processing* que describe la API de alto nivel de AutoGOAL, para usuarios no expertos, que automatiza todo el proceso de descubrimiento de

flujos en el problema de AutoML Heterogéneo. En este artículo se presentan las componentes de la biblioteca AutoGOAL y se describe su funcionamiento interno.

Capítulo 9 presenta el artículo *Demo Application for the AutoGOAL Framework* que describe una aplicación web interactiva para usar el sistema AutoGOAL. En este artículo se presentan detalles de la arquitectura de software de la biblioteca y una evaluación experimental en varios problemas de disímil naturaleza.

Parte III presenta las conclusiones y recomendaciones finales.

Capítulo 10 resume las conclusiones finales de la investigación.

Capítulo 11 presenta algunas líneas de trabajo futuro para dar continuidad a la investigación.

Capítulo 2

Estado del Arte

El proceso de investigación para resolver un problema de aprendizaje automático a menudo implica experimentar con varios enfoques diferentes (redes neuronales, clasificadores supervisados, algoritmos de agrupamiento, etc.). El diseño de una solución eficaz para cualquiera de estos problemas requiere una gran fase de experimentación en la que los investigadores utilizan diferentes conjuntos de datos, algoritmos y parámetros específicos. Como ejemplo, la Figura 2.1 muestra un flujo hipotético compuesto de varios pasos. En cada paso, hay diferentes opciones disponibles. Las combinaciones adecuadas de estas opciones producen valores diferentes para la métrica de rendimiento que se está evaluando. Estos pasos hipotéticos pueden ir desde la aplicación de algunas técnicas de preprocesamiento de datos hasta la selección de algoritmos específicos y el ajuste adicional de los valores de sus hiperparámetros.

Una solución eficaz a un problema de aprendizaje automático requiere equilibrar varias decisiones, más allá de las funciones, los modelos y las métricas de rendimiento a utilizar. Un flujo de trabajo para un problema suficientemente complejo puede requerir la combinación de diferentes técnicas en diferentes etapas, incluida la limpieza y el preprocesamiento de datos, el aumento de datos, la aplicación de diferentes representaciones, la integración del conocimiento del dominio como características adicionales, la reducción de dimensionalidad, el ajuste de modelos preentrenados, pasos de procesamiento, entre otros. La amplia gama de dominios de interés, desde imágenes hasta

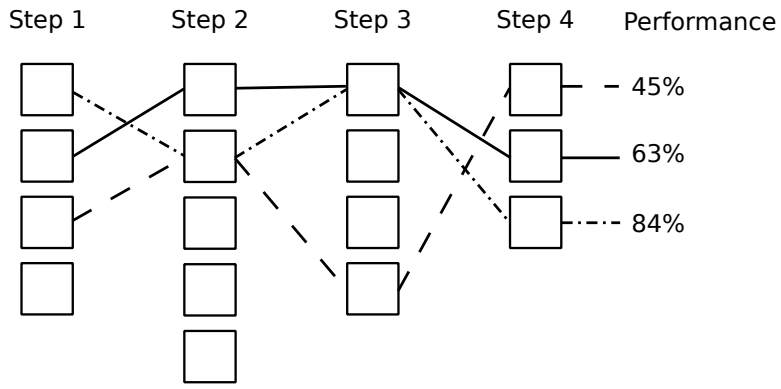


Figura 2.1: Ejemplo ilustrativo de un flujo abstracto de algoritmos, en el cual se pueden observar diferentes opciones en cada uno de los pasos que producen resultados difrentes para una métrica determinada(e.g, precisión, recobrado, etc.)

lenguaje natural, datos estructurados y series de tiempo, cada uno con sus características y mejores prácticas, agrega otra capa de complejidad que hace que la aplicación práctica del aprendizaje automático a problemas del mundo real sea aún más desafiante.

Además, en muchas ocasiones, el objetivo no es encontrar una solución óptima (o cercana a la óptima), sino analizar un gran espacio de soluciones candidatas (diferentes flujos de trabajo) para comprender el impacto de diferentes métricas involucradas. En un entorno de investigación, estas preguntas son cruciales para una comprensión más profunda del problema que se está estudiando, mientras que en un entorno de producción, podrían ser importantes para decidir sobre las compensaciones entre funcionalidad y consumo de recursos, por ejemplo.

Explorar todas las combinaciones posibles de algoritmos y parámetros para un problema dado puede ser inviable, ya que el número de posibilidades es a menudo exponencial con respecto al número de pasos en un flujo. Además, cuando algunos algoritmos tienen parámetros numéricos (discretos o continuos) (por ejemplo, tasa de regularización, número de neuronas en una capa de red neuronal), es imposible evaluar todas las combinaciones. Este problema se complica aún más por el hecho de que cada experimento puede tener una alta complejidad computacional (por ejemplo, entrenar desde cero una red neuronal en un gran conjunto de datos). Los investigadores a menudo

se ven obligados a seleccionar un pequeño número de posibilidades basándose en la experiencia previa y el conocimiento del dominio sobre el problema en cuestión.

La automatización de este proceso de experimentación se denomina Automatic Machine Learning (AutoML). AutoML es un campo cada vez más creciente cuyo objetivo es encontrar flujos de aprendizaje automático óptimos en una variedad de escenarios. Como ejemplo, en la visión por computadora, donde se han explorado ampliamente varias arquitecturas de redes neuronales, Zoph et al. [21] aplica aprendizaje por reforzamiento para aprender a construir la arquitectura de red neuronal óptima para cualquier conjunto de imágenes. Asimismo, han aparecido sistemas como Auto-Sklearn [13], Auto-Weka [14] o Auto-Keras [15], basados en bibliotecas de aprendizaje automático existentes, que exploran automáticamente diferentes combinaciones de algoritmos disponibles.

La comunidad de AutoML se ha centrado en resolver varias partes de un flujo de trabajo de aprendizaje automático estándar. Se ha prestado especial atención a la selección de modelos y el ajuste de hiperparámetros y, en menor medida, al preprocesamiento y limpieza de datos [9]. Por lo tanto, la mayoría de las soluciones de AutoML existentes se centran en una subtarea específica en este flujo de trabajo (p. ej., Búsqueda de modelo completo [18, 16] o selección de algoritmos combinados y optimización de hiperparámetros [22, 23]) o en Flujos de trabajo de extremo a extremo basados en un subdominio específico de modelos (por ejemplo, búsqueda de arquitectura neuronal [15, 24]).

En este capítulo se presenta un estudio del estado de la cuestión en el dominio de AutoML. La Sección 2.1 presenta las diferentes formulaciones téoricos del problema de AutoML. La Sección 2.2 describe los componentes fundamentales de un proceso de AutoML, así como ejemplos de sistemas para cada uno de los enfoques existentes. La Sección 2.3 presenta una comparación cualitativa entre los principales sistemas AutoML del estado del arte, con respecto a sus características tanto internas como externas. Finalmente, la Sección 2.4 describe en mayor detalle cada uno de los sistemas previamente comparados.

2.1. Definición del problema de AutoML

En esta sección se describe formalmente el problema del AutoML como un problema CASH introducido por Auto-Weka [14]. Revisamos la definición de las tarea de optimización automática de hiperparámetros (*Hyperparameter Optimization*, HPO) [25] y de selección de modelos (*Model Selection*, MS) [14], que conforman la base del AutoML.

El AutoML clásico, en esta investigación, refiere al problema del AutoML como un problema de selección combinada de modelos y optimización de hiperparámetros (*Combined Algorithm Selection and Hyperparameter Optimization*, CASH) que se describe en el presente capítulo. Siguiendo este esquema, hay dos tareas fundamentales que deben resolver eficientemente los sistemas para poder dar solución al sCASH: encontrar un modelo adecuado para una tarea de ML dada; determinar la configuración de un algoritmo que maximice su rendimiento.

2.1.1. Optimización de parámetros

Un sistema AutoML debe ser capaz de crear un flujo adecuado para un problema determinado. Un flujo de ML se compone de algoritmos que interactúan con los datos, y entre ellos. Todo algoritmo de *machine learning* tiene hiperparámetros, y una de las tareas fundamentales en el AutoML es ajustarlos para optimizar el rendimiento. La HPO permite reducir el esfuerzo humano necesario para aplicar algoritmos de ML así como mejorar el rendimiento de dichos métodos[25].

Definición 2.1. Sean A el conjunto de los algoritmos de *machine learning* y $a \in A$ un algoritmo con N parámetros. Denotamos el dominio del n-ésimo hiperparámetro como Λ_n y el espacio de configuraciones como $\Lambda = \Lambda_1$ x Λ_2 x $\ldots \Lambda_N$. Un vector de hiperparámetros es denotado por $\lambda \in \Lambda$, y a, con sus hiperparámetros instanciados en λ, por a_λ.

Un hiperparámetro Λ_i puede tener un dominio de valores continuos, discretos, booleanos o categóricos. Un parámetro de ritmo de aprendizaje sería continuo; la cantidad de capas de una red neuronal es discreta; la decisión de eliminar *stopwords* o no, es booleana.

El espacio de configuraciones $\Lambda = \Lambda_1$ x Λ_2 x $\ldots \Lambda_N$ puede contener

condicionalidad, por ejemplo, un Λ_2 puede solo ser relevante si Λ_1 tiene un valor determinado. Los espacios condicionados toman la forma de grafos dirigidos acíclicos.

Entonces, siguiendo la definición de Feurer and Hutter [25], dado un conjunto de datos D, el objetivo es encontrar

$$\lambda^* = \underset{\lambda \in \Lambda}{argmin}\ \mathbb{E}_{(D_{train}, D_{valid})} \mathbf{V}(\mathcal{L}, a_\lambda, D_{train}, D_{valid}) \tag{2.1}$$

donde $\mathbf{V}(\mathcal{L}, a_\lambda, D_{train}, D_{valid})$ mide la pérdida de un modelo generado por a_λ en el conjunto de entrenamiento D_{train} y evaluado sobre el conjunto D_{valid}. El protocolo de validación puede tomar distintas formas; una de las más populares opciones es el cálculo del error de validación cruzada con una función de pérdida definida por el usuario [25].

2.1.2. Selección de Modelos

De manera similar, un sistema AutoML realiza una búsqueda sobre un conjunto de algoritmos disponibles, para determinar cuáles pertenecen al flujo óptimo dada un problema determinado. Para alcanzar esto, se debe encontrar un algoritmo adecuado para un problema determinado. Esta tarea recibe el nombre de Selección de Modelos (*Model Selection*, MS) y es clave en el AutoML clásico.

Definición 2.2. Dado un conjunto A de algoritmos de *machine learning* y una cantidad limitada de datos $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, el objetivo de la selección de modelos es determinar el $a^* \in A$ con un rendimiento óptimo general. Para una evaluación general se divide el D en D_{train} y D_{valid} disjuntos para entrenamiento y validación respectivamente, se aprenden funciones f_i al aplicar a^* a D_{train}, y se evalúa el rendimiento de las funciones en D_{valid}. Más formalmente el problema de selección de modelos consiste en encontrar

$$a^* = \underset{a \in A}{argmin}\ \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(a, D_{train}, D_{valid}) \tag{2.2}$$

donde $\mathcal{L}(a, D_{train}, D_{valid})$ es la pérdida de a al ser evaluado en D_{valid} y entrenado con D_{train} [14]. De igual manera que en HPO, hay distintos

protocolos de validación que pueden producir diferentes resultados. Uno de los más populares es el cálculo del error por validación cruzada.

2.1.3. AutoML como problema CASH

HPO y MS representan retos importantes a día de hoy. Teniendo en cuenta que ningún algoritmo rinde mejor en todos los conjuntos de datos, y que existen algoritmos que dependen crucialmente de sus parámetros [13], es necesario desarrollar nuevas soluciones eficientes. Muchas investigaciones se dirigen a resolver estos problemas independientemente [26, 27]. Sin embargo, pueden ser formulados como un único problema estructurado que captura la intención del AutoML.

Definición 2.3. Sea $A = \{a^1, \ldots, a^k\}$ un conjunto de algoritmos con espacios de configuraciones asociados $\Lambda^1, \ldots, \Lambda^k$. Sea D un conjunto de datos que se divide en en $\{D^{(1)}_{train}, \ldots, D^{(k)}_{train}\}$ y $\{D^{(1)}_{valid}, \ldots, D^{(k)}_{valid}\}$ tal que $D^{(i)}_{train} = D \backslash D^{(i)}_{valid}$. Definimos el problema de selección combinada de modelos y optimización de hiperparámetros (CASH) [14, 13] como el computo de

$$a^*_{\lambda^*} \in \underset{a^{(j)} \in A, \lambda \in \Lambda^{(j)}}{argmin} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(a^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{valid}) \tag{2.3}$$

donde $\mathcal{L}(a, D_{train}, D_{valid})$ es la pérdida de $a^{(i)}$ al ser evaluado en D_{valid} y entrenado con D_{train}

Sin embargo, esta definición se ajusta a paradigmas específicos dentro del ML, asemejándose directamente al aprendizaje supervisado. Sistemas que resuelven esta tarea presentan dificultades o no pueden ser desplegados directamente para solucionar problemas no supervisados, dejando un amplio margen de aplicaciones del ML sin soporte para automatización.

2.2. Componentes del Proceso de AutoML

El proceso de AutoML consta de tres componentes fundamentales que definen el mecanismo de optimización [28]. El primer componente es un espacio de búsqueda que describe el conjunto de todas las soluciones posibles al problema de AutoML planteado. El segundo componente es una estrategia

de búsqueda que permite optimizar dentro del espacio definido, diseñada para encontrar buenas soluciones eficientemente. El tercer componente es una estrategia de estimación de rendimiento que permite comparar dos soluciones en función del rendimiento esperado en el problema planteado. En esta sección se describen estos tres componentes y se analizan diferentes alternativas utilizadas en sistemas del estado del arte.

2.2.1. Espacio de Búsqueda

El espacio de búsqueda define que algoritmos o combinaciones son representables en principio (soluciones posibles) para un problema de AutoML concreto. En este espacio se definen no solo los algoritmos de interés sino también los posibles rangos válidos para sus hiperparámetros. Si se optimizan flujos complejos de algoritmos, es posible que existan restricciones de compatibilidad entre algoritmos que también son modeladas.

Esta definición incorpora conocimiento *a priori* sobre las características típicas de aquellas soluciones que son buenas para una tarea en específico, lo que puede reducir el tamaño del espacio y simplificar la búsqueda. Por otro lado, incluye un sesgo humano que puede impedir que se descubran o exploren soluciones novedosas que están muy lejos del conocimiento humano actual [28]. De cierta forma se impide la creatividad en la exploración de nuevas soluciones muy distintas, aunque sí es posible que cobre relevancia la combinación de soluciones existentes que no han sido evaluadas aún.

El espacio de búsqueda más sencillo se compone de los hiperparámetros de un solo algoritmo o familia de algoritmos muy similares (p.e., factor de regularización y tipo de kernel para modelos SVM [29]). De manera general los hiperparámetros pueden ser de varios tipos: continuos, discretos, categóricos, o booleanos, entre otros. Por este motivo, aunque este tipo de espacio se puede ver como un vector de tamaño fijo, cada dimensión tiene rangos y dominios diferentes.

Un espacio de búsqueda más complejo considera múltiples algoritmos con diferentes hiperparámetros cada uno (p.e., regresión logística con regularización y árboles de decisión con profundidad máxima). A este tipo de espacio se le denomina condicional, pues una decisión respecto a qué algoritmo utilizar condiciona qué hiperparámetros son relevantes a explorar. En este tipo de espacio pueden considerarse además flujos de tamaño fijo

donde cada paso define un subconjunto de algoritmos (p.e., preprocesamiento, reducción de dimensiones y clasificación). Puede verse como un espacio a dos niveles, donde en el nivel superior se escoge para cada paso en el flujo un algoritmo dentro del conjunto válido, y en el segundo nivel se escogen los valores de sus hiperparámetros. La definición de AutoML como problema CASH es un ejemplo donde se utilizan espacios de búsqueda condicionales. Sistemas como Auto-Sklearn [22] y Auto-Weka [23] son ejemplos notables de este paradigma.

Cuando se permite que un algoritmo a su vez esté compuesto por sub-algoritmos (p.e., en los modelos de ensemble), se obtiene un espacio jerárquico. Este tipo de espacios también son condicionales, pero pueden tener una cantidad variable (potencialmente infinita) de niveles. La búsqueda de arquitecturas de redes neuronales (NAS) es un ejemplo donde se obtienen espacios jerárquicos de complejidad arbitraria, donde cada decisión sobre la arquitectura (p.e., adicionar una capa de un tipo específico) deriva recursivamente en nuevas decisiones. Este tipo de espacios permite construir flujos lineales de longitud variable o flujos en forma de grafo. Ejemplos de espacio de búsqueda jerárquicos son ML-Plan [19], RECIPE [16] y Auto-Keras [15], entre otros.

Además de la estructura, hay otras características del espacio de búsqueda que son explotadas por diferentes sistemas de AutoML:

Diferenciable: es un espacio de búsqueda donde el rendimiento de una solución es una función diferenciable. Cada solución es un punto en el espacio, y es posible calcular un gradiente que apunta en la dirección en la que mejora el rendimiento. Por lo tanto, en estos espacios es posible encontrar la mejor solución con algoritmos de optimización continuos basados en gradiente. Un ejemplo de este tipo de espacios es usado por DARTS [30] y SNAS [31].

Probabilístico: es un espacio de búsqueda donde cada solución tiene asociada un valor (que usualmente varía durante el proceso de búsqueda) que describe la probabilidad de que dicha solución tenga un rendimiento bueno. En este tipo de espacios es posible aplicar técnicas de optimización probabilísticas basadas en muestrear con mayor probabilidad ciertas regiones del espacio. Ejemplos de este tipo de espacios son utilizados por Auto-Sklearn [22] y Hyperopt [17], entre otros.

2.2.2. Estrategia de Búsqueda

La estrategia de búsqueda detalla como se explora el espacio de búsqueda, que puede ser de tamaño exponencial o ilimitado. Se ve afectado por el clásico problema de Exploración vs Explotación, ya que se quieren encontrar soluciones de alto rendimiento rápidamente pero se debe evitar converger prematuramente a regiones sub-óptimas del espacio de búsqueda [28]

A raíz del desarrollo de las investigaciones en las distintas áreas, han surgido un conjunto de propuestas que utilizan estarategias de búsqueda basadas en técnicas de diferentes dominios:

- Optimización Bayesiana
- Programación Evolutiva
- Búsqueda Aleatoria
- Aprendizaje de refuerzo
- Métodos basados en gradiente
- Métodos constructivos
- Monte Carlo

Optimización Bayesiana

Hay dos componentes principales en las propuestas de AutoML basadas en optimización bayesiana [32]: un modelo estadístico bayesiano para modelar la función objetivo, y una función de adquisición que dirige la búsqueda en el espacio. Estos sistemas, gracias al modelo estadístico, seleccionan el mejor candidato a evaluar. Luego de la evaluación, se actualiza el modelo y se itera la búsqueda.

Esta estrategia de búqueda es empleada por Auto-Sklearn [22] y Auto-Weka [23] que presentan dos optimizadores bayesianos: una propuesta de Configuración Secuencial de Algoritmos basada en Modelos (SMAC) y un Estimador de Parzen con Estructura de Árbol (TPE). Otro ejemplo es Hyperopt que utiliza SMBO (*Sequential Model-based Bayesian Optimization*), considerando este algoritmo como una componente intercambiable con el

fin de permitir que cualquier técnica de búsqueda pueda ser utilizado con cualquier problema de búsqueda [33]. Por otra parte Auto-Keras [34] extiende el paradigma de optimización bayesiana a un espacio jerárquico definiendo una función de kernel entre redes neuronales.

Programación Evolutiva

Para el problema del AutoML, los algoritmos evolutivos [35] producen una población de flujos. En cada paso evolutivo, al menos un miembro de la población actúa como "padre"para nuevos individuos. Luego, estos son evaluados y añadidos a la población. Finalmente, un proceso genera o escoge las soluciones del próximo paso evolutivo, reiniciando el ciclo. Las propuestas principalmente difieren en las estrategias utilizadas para representar el espacio de búsqueda, generar nuevos individuos y evaluar los flujos.

Se han utilizado varias metaheurísticas como estrategias de optimización. para buscar inteligentemente la mejor combinación de algoritmos para un problema dado [36, 37]. Entre las estrategias más utilizadas se encuentran los algoritmos genéticos [38] y los algoritmos evolutivos [39]. En este último grupo, la evolución gramatical [40] se ha empleado para optimizar la arquitectura de las redes neuronales [41] así como para encontrar directamente los pesos óptimos [42, 43, 44]. Esto aumenta la complejidad de la optimización, porque el espacio de búsqueda es mucho mayor. Además, existen muchas estrategias de optimización eficientes específicamente para ajustar los pesos de las redes neuronales usando variantes de descenso de gradiente, como SGD [45], RMSProp [46] y Adam [47].

A modo de ejemplo, TPOT [18] optimiza flujos de algoritmos en forma de grafos utilizando una versión de programación genética (GP) [48], una técnica bien conocida de programación evolutiva para la construcción automática de programas [49]. Además, TPOT considera una selección Pareto (NSGA-II) [50] para realizar una búsqueda multiobjetivo. Dos objetivos separados son considerados: maximizar el rendimiento final del flujo de trabajo y minimizar la complejidad total del mismo, dado un número de operadores, para evitar un sobreajuste. Por otro lado, RECIPE [16] utiliza un enfoque de programación genética basada en gramáticas. La gramática permite dirigir la búsqueda [51], restringiendo las operaciones de mutación y cruzamiento. La estructura de la gramática permite a RECIPE garantizar, a diferencia de TPOT, que

todo flujo generado es válido.

Búsqueda Aleatoria

La búsqueda aleatoria consiste en explorar el espacio de búsqueda ya sea de forma secuencial o en paralelo mediante la selección de soluciones aleatorias. Este método a menudo se usa como *baseline* en comparación con estrategias más avanzadas, aunque puede llegar a ser competitivo si el espacio de búsqueda contiene muchas soluciones buenas [52].

Un ejemplo de búsqueda aleatoria es la estrategia empleada por H2O AutoML [53], que construye múltiples modelos con diferentes juegos de hiperparámetros seleccionados aleatoriamente en base a ciertas heurísticas. En función de la disponibilidad de recursos de cómputo, la búsqueda se detiene y los mejores modelos se utilizan para construir un ensemble. Otro ejemplo de este tipo de estrategia es TransmogrifAI [54], donde la búsqueda se realiza sobre el espacio de las transformaciones de características que son posibles en función de los tipos de datos utilizados.

Aprendizaje por Refuerzo

El aprendizaje por refuerzo, como estrategia de búsqueda, consiste en entrenar un agente que realiza modificaciones sobre una solución con el objetivo de maximizar una recompensa que depende del rendimiento de dicha solución [55]. Esta estrategia es usual es búsqueda de arquitecturas en redes neuronales (NAS), donde el agente puede realizar acciones cómo añadir, quitar, o modificar una capa, o sus hiperparámetros. Las propuestas difieren en cómo representan la política del agente, y como la optimizan.

Un ejemplo de esta estrategia es utilizado por EAS [56], que utiliza un meta-controlador basado en aprendizaje por refuerzo y el algoritmo *REINFORCE* [57] para actualizarlo. En general, este sistema modela el proceso de diseño automático de arquitecturas como un proceso de toma de decisiones secuencial, donde el estado es la red actual y la acción es la operación de transformación correspondiente [56]. Después de una cantidad determinada de pasos, la arquitectura resultante es evaluada para obtener la señal de recompensa, que luego es utilizada para actualizar el meta-controlador, maximizando el rendimiento de validación esperado [56].

Métodos de Gradiente

Esta estrategia de búsqueda consiste en calcular o aproximar la dirección de gradiente en la cuál se mejora el rendimiento de una solución inicial y evaluar nuevas soluciones en una vecindad en dicha dirección. Cuando el espacio de búsqueda es diferenciable, es posible evaluar el gradiente de forma exacta y aplicar métodos de optimización continua. Un ejemplo de esta estrategia es DARTS [30], un método de optimización de dos niveles (*Bilevel Optimization*) para NAS. La propuesta modela el espacio de búsqueda de forma continua y diferenciable para poder optimizar la arquitectura por descenso por gradiente [30].

Cuando no existe información de gradiente explícito, es posible aproximarlo a partir de evaluar soluciones aleatorias en una vecindad y aplicar métodos de *hill-climbing*. Un ejemplo de este tipo de solución es NASH [58], que sigue una estrategia simple de *hill climbing* [59] para el problema NAS y realiza transformaciones sobre las redes para evitar el reentrenamiento desde cero [58]. De forma general, NASH consiste en la repetición de un ciclo de optimización que culmina al lograr una red con determinado rendimiento [60]. Un ciclo comienza con una pequeña red posiblemente pre-entrenada; luego, a partir de transformaciones sobre esta red, se construyen nuevas, que pueden rendir mejor con más entrenamiento. Estas redes "hijas" son vecinas de la red "padre" en el espacio de búsqueda. Entonces, se realiza un entrenamiento posterior para obtener la red hija con mejores resultados, que pasará a ser la red inicial en la próxima iteración del proceso [60].

Métodos Constructivos

Otra familia de estrategias de búsqueda consiste en explorar el espacio de forma estructurada, donde se definen de antemano los posibles modelos y las formas de combinarlos. Para ello se establece un orden de evaluación de las soluciones, y la búsqueda termina cuando se ha explorado todo el espacio. Esta estrategia es útil cuando el espacio de búsqueda es pequeño y consta de modelos bien definidos, variados, y con buen rendimiento. Es muy utilizada para construir meta-modelos, como ensembles.

El ejemplo más notable es AutoGluon [11], que construye un ensemble de varios niveles a partir de entrenar una instancia de cada uno de los tipos de

modelos predefinidos. Cada modelo se entrena de forma secuencial y se añade al ensemble. Al terminar se realiza un proceso de post-optimización para "destilar' el ensemble resultante en un modelo más pequeño con rendimiento similar.

Monte Carlo

Esta estrategia se emplea en espacios de búsqueda jerárquicos, para explorar eficientemente el árbol que describe todas las posibles soluciones. Cada ciclo de evaluación consiste en construir una solución completa, que se traduce en descender por una rama del árbol del espacio de búsqueda. En un algoritmo de Monte Carlo [61] es necesario definir cómo se escoge el siguiente nodo a evaluar, lo que conlleva un balance entre exploración y explotación. A medida que se explora el espacio de búsqueda, se descubre qué decisiones en niveles superiores tienen un mejor rendimiento y se sesga la búsqueda hacia esas regiones del espacio. Para que esta estrategia sea efectiva, el espacio de búsqueda debe estar estructurado de forma que las decisiones más importantes ocurran antes. Un ejemplo de esta estrategia se emplea en ML-Plan [19].

2.2.3. Estrategias de Estimación de Rendimiendo

El objetivo de AutoML es encontrar soluciones que logren predecir en datos nuevos, o sea, no aquellos con los que se entrenan los algoritmos. Las estrategias de estimación de rendimiento son mecanismos para estimar esta capacidad predictiva. La opción más simple es hacer un entrenamiento y validación estándar en los datos de entrenamiento, aunque esta solución puede llegar a ser muy costosa, teniendo en cuenta que dada la naturaleza estocástica de los algoritmos se deben realizar múltiples entrenamientos y validaciones. Este costo reduce la cantidad de soluciones distintas que pueden ser evaluadas en un tiempo definido [28].

Para estimar la capacidad predictiva, según el problema definido, se debe escoger una métrica de rendimiento, tal como precisión, recobrado, F_1, entre otras. Además, entre los sistemas de AutoML se pueden evaluar otras características, tales como la complejidad computacional de los modelos, la velocidad de convergencia del proceso de entrenamiento, la interpretabilidad

o la susceptibilidad a sesgos, entre otros. Para todas estas características, se deben definir métricas que dependen de la naturaleza del problema.

Una vez se deciden los criterios que definen el rendimiento de un sistema de AutoML, los métodos para estimarlo se pueden dividir en dos categorías [62]:

Frecuentistas: son métodos de estimación basados en medir directamente las métricas de interés en un conjunto de datos de validación. Para reducir el ruido inherente los procesos de entrenamiento estocásticos, la métrica a evaluar se estima como la media de varias observaciones en diferentes particiones de los datos. Estos métodos son costosos, pues cada estimación debe ser independiente, lo que requiere un proceso de entrenamiento y evaluación desde cero para cada modelo o flujo evaluado. La fortaleza fundamental de estos métodos es que realizando suficientes mediciones es posible estimar con precisión arbitraria el valor real de la métrica de interés, siempre que se cumpla el supuesto de que los datos de entrenamiento son una buena representación de los datos reales. Además, son simples de implementar, y pueden servir de *baseline* para comparar la eficacia de métodos más eficientes pero menos precisos.

Bayesianos: son métodos de estimación basados en aproximar mediante algún modelo las métricas de interés, evitando o disminuyendo el costo de evaluar completamente los modelos o flujos. Una manera de aproximar estas métricas puede consistir simplemente en evaluar en un conjunto reducido de datos, o realizar un conjunto reducido de épocas de entrenamiento en el caso de las redes neuronales. Los métodos más avanzados construyen modelos de estimación del rendimiento indirectos, que dependen de las características de la solución, tales como los algoritmos utilizados y el valor de los hiperparámetros. Aunque pueden ser aplicados con cualquier estrategia de búsqueda, este tipo de métodos a menudo se emplea junto a estrategias basadas en optimización bayesiana, ya que la propia naturaleza del espacio de búsqueda los hace factibles. Su principal fortaleza es la eficiencia de la estimación, lo que permite explorar más soluciones. Sin embargo, su naturaleza aproximada hace imposible garantizar que las soluciones encontradas correspondan realmente a las mejores soluciones. Por este motivo a menudo se combinan con evaluaciones frecuentistas cada cierto número

de iteraciones que permiten reajustar los modelos aproximados.

Las estrategias de estimación de rendimiento comparan dos soluciones en cuanto a una métrica de calidad. Estas estrategias permiten ordenar las soluciones evaluadas en el espacio de búsqueda en función de su efectividad para resolver el problema de AutoML definido. Mientras más exacta sea la estimación más probable es que la solución obtenida por un proceso de AutoML coincida con las mejores soluciones del espacio de búsqueda. Por otro lado mientras más eficiente sea la estimación mayor cantidad de soluciones podrán evaluarse con los mismos recursos. Por lo tanto mantener un balance entre estos dos factores contrapuestos: eficiencia y exactitud es uno de los retos más importantes del proceso de AutoML.

2.3. Comparación entre Sistemas

La idea de diseñar meta-algoritmos para seleccionar los mejores algoritmos para dominios de problemas específicos es una tendencia recurrente en la investigación de inteligencia artificial, que ha sido motivado por varios factores. Estos incluyen: la complejidad del ajuste de parámetros en problemas prácticos; la amplia variedad de algoritmos existentes con rendimiento similar; y la existencia de teoremas tipo *no-free-lunch* en optimización y aprendizaje automático [63]. En los dominios de la optimización continua y combinatoria, se han desarrollado estrategias híbridas para seleccionar entre diferentes técnicas de búsqueda (por ejemplo, diferentes metaheurísticas) para un problema específico [64]. Las hiperheurísticas [65] son metodologías de alto nivel para seleccionar o generar estrategias a partir de un espacio subyacente de componentes heurísticos para resolver un problema de optimización específico o una clase de problemas.

En esta sección se presenta una comparación entre los sistemas de AutoML más conocidos en la literatura en cuanto a características internas y externas. Se clasificaron los sistemas de AutoML en 3 categorías atendiendo a las características de su uso y el paradigma de aprendizaje automático que utilizan:

- Sistemas de AutoML genéricos.
- Sistemas de Selección de Arquitecturas de Redes Neuronales.

- Sistemas comerciales en la nube.

Los sistemas de AutoML genéricos usualmente definen un pipeline de tamaño fijo o variable que abarca varias etapas típicas de un proceso de machine learning. Por ejemplo Preprocesamiento, Selección de Características, Reducción de Dimensiones y Predicción. En general se basan en una biblioteca específica de Machine Learning para garantizar que los algoritmos que se pueden aplicar en cada uno de estos pasos sean compatibles entre sí.

Los sistemas de selección de arquitecturas (NAS) se concentran en modelos de redes neuronales que eligen la mejor arquitectura posible escogiendo parámetros como: número de neuronas, cantidad de capas o funciones de activación. A diferencia de los anteriores que configuran un pipeline estos se centran en devolver un único modelo de red neuronal que es entrenable. Lo más usual es que se especialicen según el tipo de dato de la entrada.

Los sistemas comerciales en la nube son sistemas de caja negra que brindan al usuario una interfaz programática (API) o una interfaz interactiva para subir los datos y definir el problema. Todo el proceso de entrenamiento se realiza en servidores en la nube, en los que generalmente el usuario tiene poco o ningún control sobre las configuración interna del proceso de AUtoML. El usuario generalmente no tiene acceso directo al modelo entrenado sino que interactúa con este a través de un servicio de predicción.

La Sección 2.3.1 presenta una comparación entre los sistemas de AutoML, agrupados atendiendo a la categorización anterior, en cuanto a características internas. Estas características resumen la estrategia de búsqueda, la estructura del espacio de búsqueda, y otros detalles internos de cada sistema. Por otro lado, La Sección 2.3.2 presenta un análisis de las características externas. En estas características se resumen los dominios, técnicas, y tipos de tareas de AutoML donde cada sistema es aplicable.

Las características utilizadas en esta comparación son un selección del conjunto de criterios que pueden ser relevantes para comparar diferentes sistemas de AutoML. Estos sistemas se aplican en dominios y tareas muy

variados, por lo que se han escogido características cualitativas orientadas fundamentalmente a mostrar las diferencias entre los paradigmas en los que se basa cada sistema. En dominios o tareas específicas puede ser interesante analizar otros criterios que permitan seleccionar un sistema sobre otro. Sin embargo, cuando se analiza todo el espectro del campo de AutoML, no es posible considerar un sistema objetivamente mejor que el resto, pues todos tienen ventajas y desventajas. Por este motivo, la comparación que se presenta intenta mostrar la variedad de enfoques existentes, y señalar aquellos criterios donde existe mayor espacio para la investigación.

En las tablas 2.1, 2.2, 2.3, 2.4, 2.5, y 2.6 se utiliza la siguiente leyenda:

- • representa que la característica indicada está presente en el sistema;
- ◦ representa que la característica indicada está presente, pero con ciertas restricciones o consideraciones especiales;
- **?** representa que la documentación disponible no ha permitido determinar si la característica está presente o no;
- ⊘ representa que la característica indicada no es aplicable al sistema en cuestión.

2.3.1. Análisis de Características Internas

Las características internas se refieren fundamentalmente a los sistemas de AutoML. Estas describen el fucionamiento fundamental del sistema a partir de tres conjuntos. Primero el tipo de optimización que utiliza el sistema, que corresponde a una de las estrategia de búsqueda. Segundo, características que describen el espacio de búsqueda, los tipos de parámetros representados y la complejidad de los flujos de algoritmos. Tercero, un conjunto de características desde el punto de vista de la arquitectura de software: su extensibilidad, portabilidad, entre otros.

A continuación se describen cada una de las características internas definidas para este análisis:

Optimización: Indica las estrategias de búsquedas que utiliza el sistema. Una explicación detallada de cada estrategia puede encontrarse en la sección 2.2.2.

Probabilístico: Si el espacio de búsqueda incluye un modelo probabilista de cada solución.

Jerárquico: Si el espacio de búsqueda tiene una estructura jerárquica en 2 o más niveles.

Parámetros Continuos: Si el espacio de búsqueda incluye hiperparámetros continuos.

Parámetros Discretos: Si el espacio de búsqueda incluye hiperparámetros discretos.

Construcción Automática: Si la estructura del espacio de búsqueda se infiere automáticamente a partir de los algoritmos disponibles en el sistema, en vez de ser definido explícitamente.

Garantiza flujos válidos: Si toda solución del espacio de búsqueda corresponde a un flujo o algoritmo válido, donde ningún paso es conceptualmente incoherente (p.ej., usar un algoritmo que requiere datos densos sobre una matriz esparcida).

Flujos de tamaño fijo: Si cada solución corresponde a un solo algoritmo o a un conjunto fijo de algoritmos asociados a pasos predefinidos, p.ej., preprocesamiento, reducción de dimensiones y clasificación. Este es el tipo más simple de flujo utilizado en AutoML.

Flujos lineales: Si cada solución corresponde a un flujo de longitud variable, donde cada algoritmo procesa la salida del algoritmo anterior.

Flujos de grafo: Si cada solución corresponde a un flujo con forma de grafo de tamaño variable donde cada algoritmo puede procesar la salida de uno o más de los algoritmos anteriores. Este es el tipo de flujo más complejo.

Extensible por el usuario: Si el sistema tiene la capacidad de ser extendido fácilmente, de forma que añadir un nuevo algoritmo o componente no requiera modificaciones al código del sistema AutoML.

Acceso al modelo: Si el sistema permite inspeccionar el o los modelos entrenados una vez termina el proceso de AutoML.

Portabilidad del modelo: Si el sistema permite exportar el o los modelos entrenados ya sea a un formato estándar o en alguna forma que permita usarlos en otras aplicaciones fuera del contexto de un proceso de AutoML.

Interfaz gráfica: Si el sistema brinda una aplicación con interfaz gráfica que permita usarlo sin necesidad de escribir código.

Soporta GPU: Si el sistema es capaz de aprovechar capacidades de procesamiento en GPU para acelerar el entrenamiento de los algoritmos de aprendizaje que sean compatibles con este tipo de *hardware*.

Múltiples bibliotecas: Si el sistema soporta o contiene implementaciones de más de una biblioteca de algoritmos de aprendizaje.

Sistemas AutoML genéricos

La tabla 2.1 muestra una comparación entre los sistemas de AutoML genéricos en cuanto a sus características internas. En estos sistemas predomina como estrategia de búsqueda la optimización bayesiana, aunque varios de los sistemas brindan también una estrategia aleatoria. En algunos casos, como H2O y TransmogrifAI, la búsqueda aleatoria es la estrategia principal, en vez de simplemente un *baseline*. Esto se debe a que los espacios de búsqueda en estos dos sistemas son suficientemente pequeños y bien estructurados, de forma que casi cualquier flujo encontrado tiene un alto rendimiento. Los enfoques evolutivos y por gradiente son menos comunes pero no inexistentes.

En general, los sistemas basados en optimización bayesiana soportan un espacio de búsqueda jerárquico y con una descripción probabilística, dada la propia naturaleza de la estrategia de búsqueda. Aún así, la mayoría de los espacios jerárquicos en estos sistemas están estructurados en dos niveles: selección de modelos, y ajuste de hiperparámetros [13]. Esto es especialmente cierto en los sistemas basados en el paradigma CASH como Auto-Sklearn y Auto-Weka, por lo que solamente son capaces de optimizar flujos de tamaño fijo, compuestos de un número predeterminado de pasos.

La mayoría de los sistemas de AutoML genérico son herramientas de código abierto que se usan desde una API programática. Por tal motivo, están diseñados para ser fácilmente extensibles por el usuario, generalmente

mediante la implementación de una interfaz o protocolo que permite adicionar nuevos modelos al sistema. Algunos sistemas, como Hyperpot-Sklearn [17], permiten también adicionar fácilmente otras estrategias de búsqueda. Debido a que estos sistemas generalmente se basan en una biblioteca de aprendizaje automático de código abierto, los modelos son fácilmente accesibles e inspeccionables. Sin embargo, solo en ocasiones son los modelos exportables fuera del marco del sistema AutoML, ya que a menudo incluyen una capa de software que depende del sistema AutoML para ser ejecutados.

Sistemas de Selección de Arquitecturas

La tabla 2.2 muestra la comparación entre los sistemas específicamente diseñados para la selección de arquitecturas de redes neuronales. En este caso, se incluyen dos sistemas AutoML propiamente, así como cuatro métodos de NAS que no están debidamente enpaquetados en una biblioteca de software. Los sistemas de NAS más populares, Auto-Keras [15] y Auto-Pytorch [66], se basan en optimización bayesiana, mientras que, de los otros métodos, tres realizan una relajación continua del espacio NAS y aplican optimización basada en descenso por gradientes, y el cuarto (EAS) [56] usa aprendizaje por refuerzo.

De manera general los sistemas NAS optimizan flujos en forma de grafo, que consisten en capas de redes neuronales conectadas en arquitecturas arbitrariamente complejas. Sin embargo, algunos sistemas tienen restricciones en las arquitecturas que permiten optimizar. Tal es el caso de Auto-Pytorch, que define un espacio restringido a tipos de capas densas y residuales, y solamente algunas formas de conexión basadas en conocimiento experto. Por su lado, métodos como SNAS [31] y DARTS [30] se basan en la optimización de celdas, que luego son conectadas en forma de grafos acíclicos. Por este motivo, estas arquitecturas no pueden expresar todo el espacio de redes neuronales posible, sino solamente un subconjunto que los autores consideran relevante en el dominio en cuestión.

Con respecto a las características como aplicación de software, los métodos NASH [58], EAS [56], y SNAS [31] no proporcionan una implementación en un paquete de software, por lo que estas características no son relevantes. Debido a la naturaleza del espacio de búsqueda (redes neuronales), es indispensable para estos sistemas aprovechar el cómputo en GPU para acelerar el proceso

	AutoWeka	AutoSklearn	TPOT	H2O	AutoGluon	TransmogrifAI	Hyperopt	RECIPE	Knime	ML-Plan
Optimización										
Evolutiva			●					●		
Bayesiana	●	●					●		●	
Aleatoria				●		●	●		●	
Gradiente							●		●	
Por Refuerzo										
Constructivo					●					
Montecarlo										●
Espacio de búsqueda										
Probabilístico	●	●					●			
Jerárquico	●	●			●	●	●	●	●	●
Parámetros continuos	●	●	●	○		●	●	●	●	●
Parámetros discretos	●	●	●	○		●	●	●	●	●
Construcción automática						●				
Garantiza flujos válidos	●	●		●	●	●	●	●	●	●
Flujos de tamaño fijo	●	●		○	●		●		●	●
Flujos lineales								●		
Flujos de grafo			●			●				
Arquitectura										
Extensible por el usuario		●	●		●	●	●			
Acceso al modelo	●	●	●	●	●	?	●	●	●	
Portabilidad del modelo			●	●					●	
Interfaz Gráfica				●					●	
Soporta GPU				●	●	●			●	
Multiples bibliotecas		●			●				●	●

Tabla 2.1: Comparación de sistemas de AutoML genéricos en cuanto a sus características internas.

de entrenamiento.

Sistemas Comerciales en la Nube

La tabla 2.3 muestra la comparación entre cuatro sistemas comerciales de AutoML. Debido a su naturaleza comercial, varios detalles internos de algunos sistemas son difíciles o imposibles de obtener, pues la documentación existente está orientada a usuarios no expertos y no a la comunidad científica. La mayor disponibilidad de información es, comprensiblemente, con respecto a las características de la arquitectura de software.

Los sistemas más abiertos en este sentido son Google Vertex AI y Microsoft Azure ML, que permiten entrenar los modelos en la nube y luego exportarlos en diferentes formatos, relativamente independientes de la plataforma. De manera general los sistemas comerciales incluyen una interfaz gráfica para usuarios no expertos, excepto AutoNLP, que solo es usable a través de una API REST o programática en el lenguaje Python. Todos los sistemas aprovechan al máximo las capacidades de cómputo de la plataforma correspondiente, incluyendo unidades GPU y TPU cuando están disponibles.

2.3.2. Análisis de Características Externas

Las características externas evalúan la utilidad de los sistemas de AutoML para resolver problemas concretos. Se dividen en tres conjuntos. En primer lugar, se analizan los dominios en que son aplicables los sistemas, entiéndose por dominio el tipo de datos de entrada aceptados por el sistema. En segundo lugar se analizan las técnicas que se incluyen en los sistemas, agrupadas en diferentes familias de algoritmos. Finalmente se analizan las tareas en las que son aplicables los sistemas, que consisten tanto en problemas típicos de aprendizaje automático (e.j., clasificación, regresión, o clustering) como tareas específicas de dominios concretos.

A continuación se describen cada una de las características internas definidas para este análisis:

Imágenes: Si el sistema es aplicable a problemas de imágenes, en el sentido de que tiene componentes específicamente diseñadas para este dominio, a diferencia de sistemas que pudieran aplicarse a imágenes

	Autokeras	**AutoPytorch**	**NASH**	**EAS**	**SNAS**	**DARTS**
Optimización						
Evolutiva						
Bayesiana	●	●				
Aleatoria						
Gradiente			●		●	●
Por Refuerzo				●		
Constructivo						
Montecarlo						
Espacio de búsqueda						
Probabilístico	●	●			●	●
Jerárquico	●	●	●	●		
Parámetros continuos	●	●			●	●
Parámetros discretos	●	●	●	●		
Construcción automática						
Garantiza flujos válidos	●	●	●	●	●	●
Flujos de tamaño fijo						
Flujos lineales				●		
Flujos de grafo	●	○	●		○	○
Arquitectura						
Extensible por el usuario	●		⊘	⊘	⊘	
Acceso al modelo	●	●	⊘	⊘	⊘	●
Portabilidad del modelo			⊘	⊘	⊘	
Interfaz Gráfica			⊘	⊘	⊘	
Soporta GPU	●	●	⊘	⊘	⊘	●
Multiples bibliotecas		●	⊘	⊘	⊘	

Tabla 2.2: Comparación de sistemas de AutoML para NAS en cuanto a sus características internas.

	Vertex AI	**Azure ML**	**AI Builder**	**AutoNLP**
Optimización				
Evolutiva			?	?
Bayesiana	•	•	?	?
Aleatoria	•	•	?	?
Gradiente			?	?
Por Refuerzo			?	?
Constructivo			?	?
Montecarlo			?	?
Espacio de búsqueda				
Probabilístico	•	•	?	?
Jerárquico	•		?	?
Parámetros continuos	•	•	?	?
Parámetros discretos	•	•	?	?
Construcción automática	?	?	?	?
Garantiza flujos válidos	?	?	?	?
Flujos de tamaño fijo	?	•	?	•
Flujos lineales	?		?	
Flujos de grafo	?		?	
Arquitectura				
Extensible por el usuario		?		○
Acceso al modelo	•	•		?
Portabilidad del modelo	•	•		
Interfaz Gráfica	•	•	•	
Soporta GPU	•	•	•	•
Multiples bibliotecas	•	•	?	

Tabla 2.3: Comparación de sistemas de AutoML comerciales en cuanto a sus características internas.

interpretándolas como vectores numéricos.

Lenguaje Natural: Si el sistema es aplicable a texto en lenguaje natural.

Datos Tabulares: Si el sistema es aplicable a datos numéricos o categóricos organizados de forma tabular. Este es el dominio más común.

Series de Tiempo: Si el sistema es aplicable a series de tiempo, en el sentido de que tiene componentes y algoritmos específicos para problemas en este dominio, a diferencia de sistemas que pudieran aplicarse a series de tiempo interpretándolas como vectores numéricos.

Multidominio: Si el sistema permite entradas compuestas por datos de varios dominios simultáneamente, p.ej., imágenes y texto en lenguaje natural.

Modelos Lineales: Si el sistema contiene algoritmos lineales como Regresión Logística o Regresión Lineal.

Modelos basados en árboles: Si el sistema contiene algoritmos basados en árboles de decisión, *random forest*, etc.

Modelos bayesianos: Si el sistema contiene algoritmos basados en modelos bayesianos como *naive bayes*.

Máquinas de Kernel: Si el sistema contiene algoritmos como máquinas de soporte vectorial.

Redes Neuronales: Si el sistema es capaz de optimizar arquitecturas de redes neuronales, ya sea de forma arbitraria o con restricciones.

Modelos preentrenados: Si el sistema contiene modelos preentrenados (p.ej., modelos de lenguaje como BERT o redes neuronales preentrenadas en dominios concretos).

Ensembles: Si el sistema es capaz de construir ensembles a partir de los algoritmos que contiene.

No ML : Si el sistema contiene otros algoritmos que no son de aprendizaje, p.ej., tokenizadores.

Clasificación: Si el sistema es aplicable a problemas de clasificación, ya sea binaria o en múltiples clases.

Regresión: Si el sistema es aplicable a problemas de regresión.

Agrupamiento: Si el sistema es aplicable a problemas de agrupamiento, no solamente como un paso interno, sino directamente como el objetivo a resolver.

Imputación: Si el sistema es capaz de aplicar técnicas de imputación para datos faltantes como parte de un preprocesamiento automático de los datos.

Selección de Características: Si el sistema es capaz de aplicar técnicas de selección de características como parte de un preprocesamiento automático de los datos.

Preprocesamiento de Características: Si el sistema es capaz de aplicar técnicas de preprocesamiento estándar tales como escalado y normalización de los datos.

Preprocesamiento de Dominio: Si el sistema es capaz de aplicar técnicas de preprocesamiento de dominio específico tales como *stemming* para lenguaje natural.

Predicción Estructurada: Si el sistema es aplicable a problemas de predicción estructurada tales como *postagging* o detección de entidades.

Segmentación: Si el sistema es aplicable a problemas de segmentación, tales como detección de objetos en imágenes.

Incremento de datos: Si el sistema es capaz de aplicar técnicas de incremento automático de datos.

Reducción de dimensiones: Si el sistema es capaz de aplicar técnicas de reducción de dimensiones tales como PCA o SVD.

Destilación: Si el sistema es capaz de aplicar técnicas de destilación de modelos como paso de post-optimización para disminuir la complejidad de los modelos entrenados en el proceso de AutoML.

Meta-aprendizaje: Si el sistema incluye técnicas de meta-aprendizaje para acelerar el proceso de AutoML a partir de ejecuciones anteriores.

Multitarea: Si el sistema es aplicable a múltiples tareas de forma simultánea, por ejemplo, clasificación y regresión.

Sistemas AutoML genéricos

La tabla 2.4 muestra la comparación entre los sistemas de AutoML genéricos en cuanto a sus características externas. Todos los sistemas contemplados en esta investigación son aplicables fundamentalmente a datos tabulares, aunque algunos, notablemente AutoGluon, se extienden también a otros dominios. Además, AutoGluon y TransmogrifAI son los únicos que admiten nativamente multi-modalidad, es decir, entradas combinadas de múltiples dominios como imágenes, texto, y datos tabulares, en el mismo problema.

En cuanto a las técnicas y modelos disponibles, todos los sistemas incluyen modelos lineales y basados en árboles, y la mayoría se extienden además a modelos bayesianos y máquinas de soporte vectorial. Esto se debe a que, en la mayoría de los casos, estos sistemas AutoML se basan en bibliotecas, como scikit-learn, donde estas familias de modelos están implementadas. Algunos sistemas incluyen también la búsqueda de arquitecturas neuronales, ya sea en forma restringida (e.j., KNIME [] y ML-Plan [19]) o en forma general (e.j., AutoGluon [24]). Una técnica interesante es la construcción automática de *ensembles*, que permite a algunos sistemas descubrir flujos combinados de muy alto rendimiento.

Los sistemas de AutoML genéricos se aplican fundamentalmente en tareas de clasificación, y en una menor medida, de regresión. Ninguno de los sistemas analizados es capaz de resolver problemas de agrupamiento como tarea final (a diferencia de usar técnicas de agrupamiento internamente como parte de la selección de características). Todos los sistemas, excepto Auto-Weka [23], incluyen preprocesamiento de características (e.j., normalización, escalado) y la mayoría incluyen imputación, selección de características, y reducción de dimensiones. Finalmente, es notable el caso de AutoGluon, que adiciona un paso de destilación automática de modelos, que permite obtener flujos mucho más eficientes con un rendimiento marginalmente inferior, especialmente útiles para el despliegue en dispositivos de bajo rendimiento, como sistemas IoT [67].

Sistemas de Selección de Arquitecturas

La tabla 2.5 muestra la comparación entre los sistemas de búsqueda de arquitecturas de redes neuronales en cuanto a las características externas.

	AutoWeka	**AutoSklearn**	**TPOT**	**H2O**	**AutoGluon**	**TransmogrifAI**	**Hyperopt**	**RECIPE**	**Knime**	**ML-Plan**
Dominio										
Imágenes					●					
Lenguaje Natural					●	●	●			
Datos Tabulares	●	●	●	●	●	●	●	●	●	●
Series de Tiempo						●				
Multidominio					●	●				
Técnicas										
Modelos Lineales	●	●	●	●	●	●	●	●	●	●
Modelos basados en árboles	●	●	●	●	●	●	●	●	●	●
Modelos bayesianos	●	●				●	●	●	●	**?**
Máquinas de Kernel	●	●				●	●	●		●
Redes Neuronales				●	●	○		○	○	○
Modelos preentrenados					●					
Ensembles	●	●		●	●					●
No ML						●				
Tareas										
Clasificación	●	●	●	●	●	●	●	●	●	●
Regresión	●	●	●	●	●	●	●			●
Agrupamiento										
Imputación		●		●		●		●	●	●
Selección de Características	●	●	●		●	●	●	●		●
Prep. de Características		●	●	●	●	●	●	●	●	●
Prep. de Dominio					●	●				
Predicción Estructurada						●				
Segmentación					●					
Incremento de los datos										
Reducción de dimensiones		●	●				●	●	●	●
Destilación					●					
Meta-aprendizaje		○								
Multitarea										

Tabla 2.4: Comparación de sistemas de AutoML genéricos en cuanto a sus características externas.

Todos los sistemas y métodos evaluados son aplicables fundamentalmente al dominio de imágenes, que es dónde mayor impacto ha tenido el campo de NAS [28]. Sin embargo, recientemente ha resurgido el interés por aplicar NAS a otros dominios, incluyendo texto en lenguaje natural y problemas tabulares. En este sentido, el sistema más flexible de esta familia es AutoKeras [15], que incluye componentes para todos los dominios analizados en esta comparación.

Como es de esperar, todos los sistemas de esta categoría se concentran en modelos de redes neuronales. Es notable el caso de Auto-Pytorch [66], que incluye además técnicas de *ensemble* [68]. Así mismo, todos los métodos de NAS son directamente aplicables a problemas de clasificación y, al menos potencialmente, a problemas de regresión sin grandes modificaciones. Más allá de estas tareas, Auto-Pytorch incluye también preprocesamiento y reducción de dimensiones, a través de una integración con algoritmos de la biblioteca *scikit-learn*. Por último, Auto-Keras es el único sistema de todos los analizados en este capítulo que incluye la funcionalidad multi-tarea, es decir, entrenar un mismo modelo para producir más de una predicción simultáneamente.

Sistemas Comerciales en la Nube

La tabla 2.6 muestra la comparación entre los cuatro sistemas comerciales de AutoML considerados en este capítulo en cuanto a sus características externas. En esta categoría el foco de atención está en los problemas de procesamiento de lenguaje natural, potencialmente motivados por la necesidad de los negocios de analizar las redes sociales, y procesar facturas y otros documentos de interés. En un segundo lugar, los datos tabulares siguen siendo de interés para los sistemas comerciales de AutoML, pues la mayoría de la información interna en las empresas modernas está almacenada aún en sistemas de bases de datos relacionales.

A pesar de la poca disponibilidad de información, se pudo constatar que todos los sistemas comerciales de AutoML utilizan modelos preentrenados. Esto que permite reducir notablemente el costo de desplegar un nuevo modelo, sobre todo en problemas bien estudiados como el reconocimiento de objetos o la detección de entidades. Adicionalmente, la mayoría de estos sistemas incluyen modelos de redes neuronales en una forma u otra.

En cuánto a las tareas, los sistemas comerciales son aplicables directamente a problemas de clasificación y regresión, y generalmente automatizan

	Autokeras	**AutoPytorch**	**NASH**	**EAS**	**SNAS**	**DARTS**
Dominio						
Imágenes	●	●	●	●	●	●
Lenguaje Natural	●					●
Datos Tabulares	●	●				
Series de Tiempo	●					
Multidominio	●					
Técnicas						
Modelos Lineales						
Modelos basados en árboles						
Modelos bayesianos						
Máquinas de Kernel						
Redes Neuronales	●	●	●	●	●	●
Modelos preentrenados						
Ensembles		●				
No ML						
Tareas						
Clasificación	●	●	●	●	●	●
Regresión	●	●	○	○	○	○
Agrupamiento						
Imputación						
Selección de Características						
Prep. de Características		●				
Prep. de Dominio		●				
Predicción Estructurada						
Segmentación						
Incremento de los datos	●	●				
Reducción de dimensiones		●				
Destilación						
Meta-aprendizaje						
Multitarea	●					

Tabla 2.5: Comparación de sistemas de AutoML para NAS en cuanto a sus características externas.

gran parte de la selección y preprocesamiento de características. A partir de la información disponible, no se pudo constatar ni descartar el uso de técnicas de meta-aprendizaje. Sin embargo, dada la naturaleza del cómputo en la nube, es altamente probable que estos sistemas incluyen algún tipo de retro-alimentación para mejorar su rendimiento a partir de los datos de los usuarios.

	Vertex AI	**Azure ML**	**AI Builder**	**AutoNLP**
Dominio				
Imágenes	•		•	
Lenguaje Natural	•	•	•	•
Datos Tabulares	•	•	•	
Series de Tiempo	•	•		
Multidominio				
Técnicas				
Modelos Lineales	?	•	?	
Modelos basados en árboles	?	•	?	
Modelos bayesianos	?	•	?	
Máquinas de Kernel	?	•	?	
Redes Neuronales	•	○	?	○
Modelos preentrenados	•	•	•	•
Ensembles		•	?	?
No ML			?	
Tareas				
Clasificación	•	•	•	•
Regresión	•	•	•	•
Agrupamiento		•		
Imputación	?	•	?	
Selección de Características	•	•	?	?
Prep. de Características	•	•	•	
Prep. de Dominio	•	•	•	•
Predicción Estructurada	•		•	•
Segmentación	•		•	•
Incremento de los datos	?		?	
Reducción de dimensiones				
Destilación			?	?
Meta-aprendizaje	?	?	?	?
Multitarea				

Tabla 2.6: Comparación de sistemas de AutoML comerciales en cuanto a sus características externas.

2.4. Sistemas AutoML

En esta sección se hace una descripción más detallada de cada uno de los sistemas incluidos en la comparación con el objetivo de poder evaluar mejor sus diferencias y conocer sus características. Las descripciones se centran en las caracrerísticas que describen los autores en sus artículos, aunque en algunos casos para poder extaer información de interés fue necesario consultar el código fuente o la página web del servicio. Aún de esta forma hay sistemas donde no están disponibles todos los detalles técnicos.

2.4.1. Sistemas de AutoML genéricos

Auto-WEKA

AutoWeka [23] es una herramienta que presenta un enfoque automático para resolver el problema de CASH [13]. Utiliza todos los algoritmos de clasificación y selectores o evaluadores de características implementados en el paquete estándar WEKA [69]. Aunque el enfoque es centrado en los algoritmos de clasificación en WEKA, no hay ningún obstáculo para extenderlo a otras configuraciones.

Contiene una lista de los 39 algoritmos de clasificación de WEKA. De estos modelos, 27 se consideran clasificadores base (que se pueden usar de forma independiente), 10 de los clasificadores restantes son meta-métodos (que toman un clasificador base único y sus parámetros como entrada), y los 2 clasificadores de ensemble finales pueden tomar cualquier número de clasificadores base como entrada.

Se permite que los metamétodos utilicen cualquier clasificador base con cualquier configuración de hiperparámetros, y que los 2 métodos de ensemble usaran hasta cinco de los 27 clasificadores base, nuevamente con cualquier configuración de hiperparámetros. No todos los clasificadores son aplicables a todos los conjuntos de datos (p. ej., Debido a la incapacidad de un clasificador para manejar los datos faltantes). Para un conjunto de datos determinado, Auto-WEKA solo considera automáticamente el subconjunto de clasificadores aplicables.

Contiene también los 3 métodos de búsqueda de características de WEKA,

así como sus 8 evaluadores de características y su respectivo número de subparámetros (hasta 5 para la búsqueda; hasta 4 para los evaluadores). Para realizar la selección de características, se combina un método de búsqueda con un evaluador de características, y es necesario crear una instancia de los subparámetros de ambos. La selección de características se ejecuta como una fase de preprocesamiento antes de construir cualquier clasificador. Los algoritmos mencionados tienen una amplia variedad de hiperparámetros, que toman valores de intervalos continuos, de rangos de números enteros y de otros conjuntos discretos. Se asocia una distribución uniforme o logarítmica uniforme *a priori* por cada parámetro numérico, dependiendo de su semántica. Por ejemplo, se establece una distribución *a priori* logarítmica uniforme para la penalización por regresión en *Ridge* [70] y una distribución uniforme para la profundidad máxima de un árbol en *Random Forest* [71]. Auto-WEKA trabaja con valores de hiperparámetros continuos directamente hasta la precisión de la máquina; sin embargo, para dar una idea del tamaño del espacio de búsqueda, nótese que la discretización de los dominios de hiperparámetros a un máximo de 10 valores cada uno da lugar a más de 10^{47} configuraciones de hiperparámetros. Este espacio es mucho más grande que una simple unión de los espacios hiperparámetros de los algortimos de aprendizaje base (cuyo tamaño es aproximadamente 10^8), ya que los métodos de ensemble [68] permiten hasta 5 algoritmos base independientes, dando lugar a un espacio con aproximadamente $(10^8)^5 = 10^{40}$ elementos. La selección de características da lugar a otra decisión independiente entre aproximadamente 10^6 opciones, y varios parámetros en el nivel de meta y de ensemble contribuyen en otro orden de magnitud al tamaño total del espacio de hiperparámetros de Auto-WEKA.

Auto-WEKA puede entenderse como un algoritmo de aprendizaje único con un espacio de hiperparámetros altamente condicional. Auto-WEKA tiene dos parámetros booleanos de nivel superior. El primero selecciona entre clasificadores base únicamente y clasificadores de ensemble o meta. Si este parámetro es verdadero, entonces otro parámetro determina cuál de los 27 clasificadores base se utilizará. Si es falso, entonces se utiliza un ensemble o un metaclasificador. En el segundo caso, se elige uno de los 27 clasificadores base. En el caso de que se escoja un clasificador de ensemble, un parámetro adicional es elegido entre $1, \ldots, 5$, que determina la cantidad de clasificadores base a usar. Para cada uno de los diferentes parámetros base, se adjuntan

hiperparámetros condicionales para cada modelo.

El segundo parámetro booleano de nivel superior determina si se aplica uno de los métodos de selección de características. Si es falso, Auto-WEKA pasa el conjunto de datos sin modificar al clasificador. Si es verdadero, se selecciona la opción de método de búsqueda de características, y la opción de evaluador de características. Esto da como resultado un árbol muy amplio que captura toda la naturaleza jerárquica de los hiperparámetros del modelo y permite la creación de un único problema de optimización de hiperparámetros con cuatro capas jerárquicas de un total de 786 parámetros.

Auto-WEKA es independiente de la elección del optimizador, por lo que implementa variantes que aprovechan SMAC [72] y TPE [73]. Se definen dos variantes de Auto-WEKA, basadas en SMAC y TPE, respectivamente. Ambas versiones de Auto-WEKA están disponibles para el público en `www.cs.ubc.ca/labs/beta/Projects/autoweka`. Tanto TPE como SMAC tienen sus propios parámetros que influyen en su rendimiento. En Auto-Weka, se utilizan los valores predeterminados para estos meta-hiperparámetros, según lo establecido por los autores. Finalmente, tanto TPE como SMAC son algoritmos aleatorios y, por lo tanto, producen resultados diferentes basados en la semilla aleatoria proporcionada. Esto permite una paralelización trivial pero efectiva del proceso de optimización simplemente realizando k ejecuciones independientes del método de optimización en paralelo y seleccionando el resultado de la ejecución con el menor error de validación cruzada.

Auto-sklearn

Auto-sklearn [22] elige como biblioteca de aprendizaje automático subyacente, scikit-learn [74], una de las bibliotecas de aprendizaje automático más conocidas y utilizadas. Ofrece una amplia gama de algoritmos de aprendizaje bien establecidos e implementados de manera eficiente y es fácil de usar tanto para expertos como para principiantes. El sistema AutoML se parece mucho a Auto-WEKA [23], pero, como HYPEROPT-SKLEARN [17], se basa en scikit-learn.

Este sistema comprende 15 algoritmos de clasificación, 14 métodos de preprocesamiento de características y 4 métodos de preprocesamiento de datos. Teniendo en cuenta sus parámetros, se define un espacio de 110

hiperparámetros. La mayoría de estos son hiperparámetros condicionales que solo están activos si se selecciona su componente respectivo, SMAC [72] puede manejar esta condicionalidad de forma nativa.

Los 15 algoritmos de clasificación de Auto-sklearn se dividen en diferentes categorías, como modelos lineales generales [75] (2 algoritmos), máquinas de soporte vectorial [29] (2), análisis discriminante [76] (2), vecinos más cercanos [77] (1), Bayesianos [78] (3), árboles de decisión [79] (1) y ensembles [68] (4). En contraste con Auto-WEKA, se enfoca el espacio de configuración en clasificadores base y se excluyen metamodelos y ensembles que son parametrizados por uno o más clasificadores base. Si bien estos conjuntos aumentan el número de hiperparámetros de Auto-WEKA en casi un factor de cinco (a 786), Auto-sklearn solo presenta 110 hiperparámetros. En su lugar, se construyen ensembles complejos utilizando un método post-hoc. Comparado con Auto-WEKA, esto es mucho más eficiente en cuanto a los datos: en Auto-WEKA, evaluar el desempeño de un ensemble con cinco componentes requiere la construcción y evaluación de cinco modelos; por el contrario, en Auto-sklearn, los ensembles vienen en gran parte de forma gratuita y es posible mezclar y combinar modelos evaluados en momentos arbitrarios durante la optimización.

Los métodos de preprocesamiento para conjuntos de datos en representación densa en Auto-sklearn comprenden preprocesadores de datos (que cambian los valores de las funciones y siempre se utilizan cuando se aplican) y preprocesadores de funciones (que cambian el conjunto real de funciones, y solo se utiliza una, o ninguna). El preprocesamiento de datos incluye el cambio de escala de las entradas, la imputación de los valores perdidos, la codificación one-hot y el equilibrio de las clases. Los 14 métodos posibles de preprocesamiento de características se pueden clasificar en selección de características (2), aproximación de kernel [80] (2), descomposición de matrices (3), *embeddings* [81] (1), agrupamiento de características (1), expansión de características polinomiales (1) y métodos que usan un clasificador para la selección de características (2). Por ejemplo, las SVM [29] lineales regularizadas ajustadas con L1 a los datos se pueden utilizar para la selección de características eliminando las características correspondientes a los coeficientes del modelo de valor cero.

Para aprovechar al máximo el poder de cómputo y no realizar una

ejecución muy lenta de una determinada combinación de algoritmo de preprocesamiento y aprendizaje automático, Autosklearn implementa varias medidas. Primero, se limita el tiempo para cada evaluación de una instanciación de un flujo así como la memoria de dichas evaluaciones para evitar que el sistema operativo se congele. Cuando una evaluación supera uno de esos límites, se cancela automáticamente y se asigna la peor puntuación posible para la métrica de evaluación dada. Para algunos de los modelos se emplea un procedimiento de entrenamiento iterativo; de forma que devuelvan su valor de rendimiento actual cuando se alcanzó un límite antes de que se cancelen. Para reducir aún más la cantidad de ejecuciones demasiado largas, se eliminan varias combinaciones de preprocesadores y métodos de clasificación: en particular, se elimina que la aproximación de kernel [80] estuviera activa junto con métodos no lineales y basados en árboles, así como en el algoritmo KNN [77]. (SMAC maneja tales combinaciones restringidas de forma nativa). Por la misma razón, tampoco incluyen los algoritmos de aprendizaje de funciones, como el aprendizaje de diccionario.

Otro problema en la optimización de hiperparámetros es el sobreajuste y el remuestreo de datos, ya que los datos de entrenamiento del sistema AutoML deben dividirse en un conjunto de datos para entrenar el flujo de algoritmos de aprendizaje (conjunto de entrenamiento) y un conjunto de datos utilizado para calcular la función de pérdida para la optimización bayesiana (conjunto de validación). Los autores de este sistema proponen un balance entre ejecutar una validación cruzada más robusta (que conlleva una pequeña sobrecarga adicional en SMAC) y evaluar modelos en todos los subconjuntos de validación cruzada para permitir la construcción de ensembles con estos modelos. No todas las tareas de aprendizaje supervisado (por ejemplo, la clasificación multiclase) pueden resolverse mediante todos los algoritmos disponibles en Auto-sklearn. Por lo tanto, dado un nuevo conjunto de datos, Auto-sklearn preselecciona los métodos que son adecuados según las propiedades del conjunto de datos. Dado que los métodos de scikit-learn están restringidos a valores de entrada numéricos, siempre se tranforma los datos aplicando una codificación *one-hot* a características categóricas. Para mantener bajo el número de características ficticias, se configura un umbral de porcentaje y cualquier valor que ocurre con menos frecuencia que este porcentaje se transforma en otro valor especial.

Auto-sklearn incluye un componente de meta-aprendizaje que funciona

de la siguiente manera. En una fase anterior a la optimización, para cada conjunto de datos de aprendizaje automático en un repositorio de conjuntos de datos (140 conjuntos de datos del repositorio OpenML [82]), se evalúan un conjunto de metacaracterísticas, usando optimización bayesiana para determinar y almacenar un flujo que presenta un rendimiento empírico para ese conjunto de datos. Luego, dado un nuevo conjunto de datos D, se calculan sus meta-características, se clasifican todos los conjuntos de datos por su distancia L_1 a D en el espacio de meta-características y se seleccionan los flujos almacenados para los $k = 25$ conjuntos de datos más cercanos para la evaluación antes de comenzar la optimización bayesiana [32] con sus resultados.

Para caracterizar los conjuntos de datos, se utilizan un total de 38 meta-características de la literatura, incluyendo meta-características simples, basadas en teoría de la información y estadística, tales como estadísticas sobre el número de puntos de datos, características y clases, como así como la asimetría de los datos y la entropía de los objetivos. Se observa que este enfoque de metaaprendizaje obtiene su poder de la disponibilidad de un repositorio de conjuntos de datos.

TPOT

TPOT [18] está basado en la biblioteca de aprendizaje automático scikit-learn [74]. Por lo tanto, cada operador del flujo de aprendizaje automático (es decir, un paso atómico del flujo) en TPOT corresponde a un algoritmo de aprendizaje automático, como un modelo de clasificación supervisado o un normalizador de características. Todas las implementaciones de los algoritmos de aprendizaje automático que utilizan son de scikit-learn (excepto XGBoost [83]). Operadores de clasificación supervisados: DecisionTree [79], RandomForest [71], eXtreme Gradient Boosting Classifier [83], LogisticRegression [84] y KNearestNeighborClassifier [77]. Los operadores de clasificación almacenan las predicciones del clasificador como una nueva característica. También contienen algoritmos de preprocesamiento de funciones como StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler, RandomizedPCA [85], Binarizer y PolynomialFeatures (de la biblioteca scikit-learn). Estos modifican el conjunto de datos de alguna manera y devuelven el conjunto de datos modificado.

Los algoritmos de selección de funciones como: VarianceThreshold, SelectKBest, SelectPercentile, SelectFwe y Eliminación de características recursivas (RFE) reducen el número de características en el conjunto de datos utilizando algunos criterios y devuelven el conjunto de datos modificado. También incluyen un operador que combina conjuntos de datos dispares que permite combinar múltiples copias modificadas del conjunto de datos en un solo conjunto de datos. Por último, proporcionan valores enteros y flotantes para parametrizar los distintos operadores, como el número de vecinos k en el clasificador k-Vecinos más cercanos [77].

Para combinar estos operadores en un flujo de aprendizaje automático, se tratan como elementos atómicos y se construyen árboles a partir de ellos. Debido a que todos los operadores reciben un conjunto de datos como entrada y devuelven el conjunto de datos modificado como salida, es posible construir flujos de aprendizaje automático de forma arbitraria que pueden actuar en múltiples copias del conjunto de datos. Por lo tanto, los árboles proporcionan una representación intrínsecamente flexible de los flujos de aprendizaje automático.

Para que funcionen estos flujos basados en árboles, se almacenan tres variables adicionales para cada registro en el conjunto de datos. La variable çlaseïndica la etiqueta verdadera de cada registro y se utiliza al evaluar la precisión de cada flujo. La variable çonjeturaïndica la última conjetura del flujo para cada registro, donde las clasificaciones del último operador de clasificación en este se almacenan como la çonjetura". Por último, la variable "grupoïndica si el registro se utilizará como parte del conjunto de pruebas o entrenamiento interno, de modo que los flujos basados en árboles solo se entrenan con los datos de entrenamiento y se evalúan con los datos de las pruebas. El conjunto de datos proporcionado a TPOT como datos de entrenamiento se divide en un conjunto de pruebas y entrenamiento estratificado interno, con proporción 25 % y 75 % respectivamente.

Para generar y optimizar automáticamente estos flujos basadas en árboles, se utiliza un algoritmo genético (*Genetic Programing GP*) basado en el paquete DEAP de Python [86]. El algoritmo TPOT GP sigue un proceso estándar: Para comenzar, el algoritmo genera 100 flujos aleatorios basados en árboles y evalúa su precisión con validación cruzada equilibrada en el conjunto de datos. Para cada generación del algoritmo GP, el algoritmo selecciona los

20 principales flujos de la población de acuerdo con el esquema de selección NSGA-II [87], donde los flujos se seleccionan para maximizar simultáneamente la precisión de clasificación en el conjunto de datos y minimizar el número de operadores. Cada una de los 20 principales flujos seleccionadas produce cinco copias (es decir, descendencia) en la población de la próxima generación, el 5 % de esa descendencia se cruza con otra descendencia mediante un cruzamiento de punto (*point crossover* [88]), luego el 90 % de la descendencia restante no afectada se cambian aleatoriamente por una mutación. Cada generación, el algoritmo actualiza un frente de Pareto de las soluciones no dominadas descubiertas en cualquier punto de la ejecución del algoritmo genético. El algoritmo repite este proceso de evaluación, selección, cruce y mutación durante 100 generaciones, agregando y ajustando operadores del fujo que mejoran la precisión de clasificación y eliminando los operadores que la degradan. Al finalizar se selecciona el flujo de mayor precisión del frente de Pareto como el "mejor"flujo representativo de la ejecución.

Los autores se evalúan en 150 conjuntos de datos de clasificación supervisada de una amplia variedad de fuentes, incluido el repositorio de aprendizaje automático UCI [89], un gran repositorio de referencia y conjuntos de datos de análisis genéticos simulados. Estos conjuntos de datos de referencia varían de 60 a 60.000 registros, de pocos a cientos de características, e incluyen problemas de clasificación supervisados de clases múltiples y binarios.

En general, TPOT descubre flujos que funcionan estadísticamente mejor que un *RandomForest* [71] con 500 árboles en 21 conjuntos de datos, significativamente peor en 4 y no tuvieron diferencias significativas en 125. En particular, la mayoría de las mejoras de TPOT en los conjuntos de datos evaluados son bastante grandes, con varias mejoras en la precisión de la mediana del 10 % al 60 % en comparación. Por el contrario, los 4 conjuntos en los que TPOT experimentó una degradación en la precisión media oscilaron entre solo un 2-5 % de degradación de la precisión. En algunos casos, las mejoras de TPOT se realizaron al descubrir preprocesadores de características útiles que permiten que los modelos clasifiquen mejor los datos, por ejemplo, TPOT descubrió que la aplicación de un preprocesador de características RandomizedPCA [85] antes de modelar los datos de "Hill Valley"permite a *RandomForests* clasificar el conjunto de datos con una precisión casi perfecta. En otros casos, las mejoras de TPOT se realizaron aplicando un modelo diferente, por ejemplo TPOT descubrió que un clasificador de knn [77] con k

= 10 puede clasificar el punto de referencia de "paridad5", mientras que un *RandomForest* logró consistentemente 0 % de precisión en el mismo.

Al comparar TPOT con una versión de TPOT que usa búsqueda aleatoria ("TPOT Random"), se decubren flujos que logran una precisión comparable a los descubiertos por TPOT, excepto en algunos conjuntos de datos. Sin embargo, la búsqueda aleatoria a menudo genera flujos innecesariamente complejos, incluso cuando un algoritmo simple con un modelo ajustado es suficiente para resolver el problema dado. Por lo tanto, incluso si la búsqueda aleatoria a veces puede funcionar tan bien como TPOT en términos de precisión, realizar una búsqueda guiada que logre una alta precisión con la menor cantidad posible de operaciones todavía ofrece ventajas considerableadverstisementilidad.

H2O AutoML

H2O AutoML [53] es un sistema de AutoML implementado en H2O [90], una biblioteca de aprendizaje automático distribuida, escalable y de código abierto. H2O AutoML está disponible en Python, R, Java y Scala, así como a través de una interfaz de usuario web. Aunque el proceso de AutoML es completamente automatizado, muchas de las configuraciones se exponen como parámetros al usuario, de modo que ciertos aspectos de los pasos de modelado se pueden personalizar.

H2O AutoML proporciona actualmente el mismo tipo de preprocesamiento automático de datos que proporcionan todos los algoritmos de aprendizaje supervisado de H2O. Esto incluye la imputación automática, la normalización (cuando sea necesario) y la codificación *one-hot* para los modelos que lo requieran. Los modelos basados en árboles H2O admiten divisiones de grupo en variables categóricas, por lo que los datos categóricos se pueden manejar de forma nativa

H2O AutoML incluye dos implementaciones de Gradient Boosting Machine (GBM) [83], *Random Forest* [71], redes neuronales [91] y modelos lineales generalizados (GLM) [75]. Además permite la aceleración del entrenamiento por GPU. El sistema entrena y valida de forma cruzada varios modelos prediseñados: tres modelos preespecificados de la biblioteca XGBoost, un modelo del GLM, un *Random Forest* predeterminada, un modelo de *Extremely Randomized Trees* [92], y variantes de estos algoritmos con

valores de hiperparámetros predefinidos. Para cada algoritmo, se identifica qué hiperparámetros se consideran más importantes y se definen rangos para esos parámetros, por lo que se emplea una búsqueda aleatoria para generar modelos.

Los hiperparámetros a considerar, así como sus rangos, se decidieron en función de la evaluación comparativa y la experiencia de expertos del dominio. Los modelos preespecificados se incluyen para ofrecer valores predeterminados rápidos y fiables para cada algoritmo. El orden de los algoritmos, que el usuario puede personalizar, está configurado para comenzar con modelos que brinden buenos resultados consistentemente (modelos XGBoost [83] preespecificados) en una amplia variedad de conjuntos de datos, seguidos de un GLM ajustado para un punto de referencia rápido. Luego se prioriza aumentar la diversidad en el conjunto de modelos mediante la introducción de varios modelos de *Random Forest*, GBM y redes neuronales. Una vez que este conjunto de modelos prediseñados se entrena y se agrega a la tabla de clasificación, se ejecuta una búsqueda aleatoria en esos mismos algoritmos. La proporción de tiempo dedicado a cada algoritmo en la ejecución de AutoML se define explícitamente para dar a algunos algoritmos (por ejemplo, XGBoost GBM, H2O GBM) más tiempo que a otros, de acuerdo con un criterio experto.

Después de entrenar los modelos base, se entrenan dos modelos de ensemble apilados [93]. Los ensembles apilados, también denominados Stacking o Super Learning, son una clase de algoritmos que implica entrenar a un meta-algoritmo de segundo nivel para encontrar la combinación óptima de los algoritmos base. Los ensembles apilados funcionan particularmente bien si los modelos base son individualmente fuertes y cometen errores no correlacionados. La búsqueda aleatoria en una variedad de familias de algoritmos produce un ensemble diverso de modelos base y, cuando se combina con el apilamiento, produce ensembles efectivos. Los dos tipos de ensemble de H2O AutoML son, respectivamente, uno con todos los modelos, y otro con el mejor modelo de cada familia que haya sido entrenado. Esta segunda variante incluye un modelo XGBoost GBM, un *Random Forest*, un modelo *Extremely Randomized Tree*, un GBM, una red neuronal y un modelo GLM. Este ensemble está optimizado para casos de uso de producción, ya que solo contiene seis (o menos) modelos base y puede generar predicciones rápidamente en comparación con el ensemble que contiene todos los modelos. En general,

ambos ensembles producen mejores modelos que cualquier modelo individual de la ejecución de AutoML; sin embargo, en casos excepcionales, modelos como un GLM simple pueden llegar a lo más alto de la tabla de clasificación.

Debido a las implementaciones eficientes de los algoritmos y la naturaleza distribuida de la plataforma H2O, H2O AutoML puede escalar a grandes conjuntos de datos (por ejemplo, más de 100 millones de filas). El entrenamiento de modelos individuales se paraleliza a través de núcleos de CPU en una sola máquina, o en un grupo de máquinas en red en una configuración multinodo. Los modelos XGBoost también admiten la aceleración por GPU para acelerar aún más el entrenamiento. Uno de los beneficios de crear un sistema AutoML sobre una biblioteca de aprendizaje automático rápido y escalable es que puede utilizar la velocidad y el paralelismo para entrenar más modelos en la misma cantidad de tiempo en comparación con otras bibliotecas de AutoML. Esto permite utilizar técnicas simples y directas como la búsqueda aleatoria y el apilamiento para lograr un rendimiento excelente en la misma cantidad de tiempo que los algoritmos que utilizan técnicas de ajuste más complejas, como la optimización bayesiana, o algoritmos genéticos.

AutoGluon Tabular

AutoGluon-Tabular [24] es un sistema AutoML que realiza procesamiento de datos avanzado, aprendizaje profundo y ensamblaje de modelos multicapa. Reconoce automáticamente el tipo de datos en cada columna para un preprocesamiento de datos robusto, incluido el manejo especial de campos de texto. AutoGluon se adapta a varios modelos que van desde árboles potenciados listos para usar hasta modelos personalizados de redes neuronales [91]. Estos modelos se ensamblan de una manera novedosa: los modelos se apilan en múltiples capas y se entrenan de una manera que garantiza que los datos sin procesar se puedan traducir en predicciones de alta calidad dentro de una restricción de tiempo determinada. El sobreajuste se mitiga a lo largo de este proceso al dividir los datos de varias maneras con un seguimiento cuidadoso de los ejemplos queestán en cada subconjunto de validación.

Además, presentan varias extensiones novedosas que aumentan aún más la precisión, incluido el uso de conexiones de salto(que conectan las capas iniciales y finales de una red neuronal) tanto en los *ensembles* [68] de múltiples capas como en los *embeddings* [81] de redes neuronales, así como el proceso

de *bagging* [68] para frenar el sobreajuste.

AutoGluon establece los siguientes principios para el diseño de un marco de AutoML:

Sencillez: Un usuario puede entrenar un modelo en los datos sin procesar directamente sin conocer los detalles sobre los datos y los modelos de Aprendizaje Automático.

Robustez: El marco puede manejar una gran variedad de conjuntos de datos estructurados y garantiza que el entrenamiento sea exitoso incluso cuando fallan algunos de los modelos individuales de Aprendizaje Automático.

Tolerancia a fallos: El entrenamiento se puede detener y reanudar en cualquier momento. Este comportamiento es preferible cuando se trata de instancias interrumpibles (puntuales) en la nube.

Tiempo predecible: Devuelve los resultados dentro del presupuesto de tiempo especificado por los usuarios.

Dentro de la llamada ajustar(), AutoGluon automáticamente: preprocesa los datos sin procesar, identifica qué tipo de problema de predicción es (clasificación binaria, de clases múltiples o regresión), divide los datos en varios subconjuntos para el entrenamiento del modelo frente a la validación, individualmente se adapta a varios modelos y, finalmente, crea un conjunto de modelos optimizado que supera a cualquiera de los modelos entrenados individuales. Para los usuarios dispuestos a tolerar tiempos de entrenamiento más largos para maximizar la precisión predictiva, fit() proporciona opciones adicionales que pueden especificarse:

- `hyperparameter_tune = True` optimiza los hiperparámetros de los modelos individuales.
- `auto_stack = True` elige de forma adaptativa una estrategia de conjunto de modelos basada en la agregación *bootstrap* [94] y la creación de *ensembles* (multicapa) [93].
- `eval_metric` especifica la métrica utilizada para evaluar el rendimiento predictivo.

- límites de tiempo que controlan el tiempo de ejecución de fit().

Todos los resultados intermedios se guardan en disco. Si se canceló una llamada, se puede ejecutar fit() con el argumento `continue_training = True` para reanudar el entrenamiento. TabularPrediction es una de las tareas en el marco general de AutoGluon, que también es compatible con AutoML en datos de texto e imágenes. Ofrece una amplia gama de características que incluyen el ajuste de hiperparámetros, la búsqueda de arquitecturas para redes neuronales y el entrenamiento distribuido.

Cuando el usuario no lo especifica, AutoGluon infiere primero el tipo de problema de predicción en cuestión en función de los tipos de valores presentes en la columna objetivo. Los valores no numéricos indican un problema de clasificación (con el número de clases igual al número de valores únicos observados en esta columna), mientras que los valores numéricos con pocas repeticiones indican un problema de regresión.

AutoGluon utiliza un conjunto de modelos a medida en un orden predefinido. Esto asegura que los modelos de rendimiento confiable, como *RandomForest* [71], se entrenan antes que los modelos más costosos y menos confiables, como los k vecinos más cercanos [77]. Esta estrategia es crítica cuando se le imponen límites de tiempo estrictos al proceso de AutoML. La lista de algoritmos de AutoGluon es mucho más pequeña que la multitud de candidatos considerados por los marcos de AutoML como TPOT [18], Auto-WEKA [23] y auto-sklearn [13]. No obstante, AutoGluon es lo suficientemente modular como para que los usuarios puedan agregar fácilmente sus propios modelos personalizados al conjunto de modelos que AutoGluon entrena, sintoniza y ensambla automáticamente.

Los datos tabulares carecen de la invariancia de traducción y la localidad de imágenes y texto que pueden explotarse mediante convoluciones o recurrencia. En cambio, los conjuntos de datos tabulares se componen de diversos tipos de valores y, por lo tanto, suelen elegirse arquitecturas de redes más clásicas. Sin embargo, las características sin procesar en una tabla de datos a menudo corresponden a variables significativas, más adecuadas para las divisiones de una sola variable alineadas con el eje de los modelos de árbol, que una capa de alimentación densa que combina linealmente todas las variables juntas en valores de activación de unidades ocultas individuales. No obstante se ha demostraron que las redes neuronales ajustadas adecuadamente pueden

proporcionar aumentos de precisión significativos cuando se agregan a un conjunto existente de otros tipos de modelos. En particular, los límites de decisión aprendidos por las redes neuronales difieren de la geometría alineada con el eje de los modelos basados en árboles y, por lo tanto, proporcionan una valiosa diversidad cuando se combinan con árboles.

La red aplica una capa de *embedding* separada a cada característica categórica, donde la dimensión se selecciona proporcionalmente al número de niveles únicos observados para esta característica. Para datos multivariados, las capas de *embeddings* individuales permiten a nuestra red aprender por separado sobre cada característica categórica antes de que su representación se mezcle con otras variables. Las incorporaciones de características categóricas se concatenan con las características numéricas en un vector grande que se alimenta a una red de 3 capas y se conecta directamente a las predicciones de salida a través de una conexión.

El modelo de *ensemble* multicapa puede mejorar las deficiencias de las predicciones base individuales y explotar las interacciones entre los modelos base que ofrecen un mayor poder predictivo. Dado que alimenta las predicciones generadas por los modelos de *ensemble* como entradas a modelos de *ensemble* de capas superiores adicionales. Iterar este proceso en múltiples capas ha sido una estrategia ganadora en importantes concursos de predicción [95]. Sin embargo, no es trivial implementarlo de manera robusta y, por lo tanto, no es común en otros marcos de AutoML. AutoGluon presenta una forma novedosa de conjunto de *ensemble* de múltiples capas. La primera capa tiene múltiples modelos base, cuyas salidas se concatenan y luego se alimentan a la siguiente capa, que a su vez consta de múltiples modelos de *ensemble*. Estos luego actúan como modelos base para una capa adicional.

Se amplia el método de *ensemble* tradicional con tres cambios que mejoran su precisión resultante. Para evitar otro problema de optimización CASH [13] en el nivel de selección de *ensemble*, generalmente se emplean modelos más simples en el *ensemble* que las capas base. En cambio, AutoGluon simplemente reutiliza todos sus tipos de modelos de capa base (con los mismos valores de hiperparámetros) como *ensemble*. Esta técnica puede verse como una forma alternativa de aprendizaje profundo que utiliza entrenamiento por capas, donde las unidades conectadas entre capas pueden ser modelos de Aprendizaje Automático arbitrarios. A diferencia de otras estrategias

existentes, los modelos de *ensemble* de AutoGluon toman como entrada no solo las predicciones de los modelos en la capa anterior, sino también las características de datos originales en sí mismas (los vectores de entrada son características de datos concatenados con predicciones del modelo de capa inferior). Con reminiscencias de las conexiones de salto en el aprendizaje profundo, esto permite a los *ensembles* de capa superior volver a visitar los valores de datos originales durante el entrenamiento. La capa de *ensemble* final aplica la selección de conjuntos para agregar las predicciones de los modelos de una manera ponderada. El uso de AutoGluon de la selección de conjuntos como la capa de salida de un conjunto de pila es una estrategia diferente. Si bien se ha abogado por la selección de conjuntos para combinar modelos base debido a su resistencia al sobreajuste, esta propiedad se vuelve aún más valiosa cuando se agregan predicciones en una pila de modelos de alta capacidad.

AutoGluon mejora aún más su rendimiento al utilizar todos los datos disponibles tanto para el entrenamiento como para la validación, a través del ensamble de conjuntos validación cruzada de todos los modelos en todas las capas de la pila. El *bagging*, también llamado comité de validación cruzada, es un método de conjunto simple que reduce la varianza en las predicciones resultantes. Esto se logra dividiendo aleatoriamente los datos en k fragmentos disjuntos (estratificamos en función de las etiquetas) y, posteriormente, entrenando k copias de un modelo con un fragmento de datos diferente retenido de cada copia. AutoGluon empaqueta todos los modelos y se le pide a cada modelo que produzca predicciones en el trozo que no vio durante el entrenamiento. Como cada ejemplo de entrenamiento es para una de las copias del modelo, esto permite obtener predicciones de cada modelo para cada ejemplo de entrenamiento.

Al construir el *ensemble*, es fundamental que los modelos de la capa superior solo se entrenan sobre la base de predicciones de la capa inferior en subconjuntos de validación. El entrenamiento sobre las predicciones de la capa inferior en la muestra podría amplificar el sobreajuste e introducir un cambio de covariables en el momento de la validación. El ensamblado ingenuo con una división tradicional de entrenamiento / validación en lugar de utilizar el método de *bagging*, por lo tanto, solo puede usar una fracción de los datos para entrenar. Este problema se agrava aún más con varias capas. El uso de predicciones en conjuntos de validación permite, en cambio, que

los modelos de capas superiores aprovechen la misma cantidad de datos de entrenamiento que los de la capa anterior. Si bien el este proceso reutiliza de manera eficiente los datos de entrenamiento, sigue siendo susceptible a una forma sutil de sobreajuste. El proceso de entrenamiento de ciertos modelos puede verse influenciado por los datos de validación a través de factores como el criterio de parada temprana(*early stopping*), que puede conducir a un sobreajuste menor en las predicciones de validación. Sin embargo, un modelo de *ensemble* entrenado en predicciones sobreajustadas de capas inferiores puede amplificar este problema en las capas superiores.

AutoGluon propone un proceso de *bagging* repetido para mitigar tal sobreajuste. Cuando a AutoGluon se le da suficiente tiempo de entrenamiento, repite el proceso en n particiones aleatorias diferentes de los datos de entrenamiento, promediando todas las predicciones sobre los conjuntos repetidos. El número de repeticiones, n, se selecciona estimando cuántas rondas de *bagging* se pueden completar dentro del tiempo de entrenamiento asignado. Las predicciones que se han promediado en múltiples iteraciones muestran menos varianza y es menos probable que se ajusten demasiado. Este proceso es particularmente efectivo para conjuntos de datos pequeños donde surge el sobreajuste debido a tamaños de datos de validación limitados.

TransmogrifAI

TransmogrifAI [54] es un sistema AutoML que encapsula cinco componentes del proceso de aprendizaje automático. En primer lugar, TransmogrifAI realiza inferencia de características, asumiendo el siguiente paradigma. El primer paso en cualquier flujo de aprendizaje automático es la preparación de datos. El científico de datos recopila todos los datos relevantes y aplana, une y agrega las diferentes fuentes de datos para extraer señales sin procesar que podrían tener poder predictivo. Las señales extraídas se rellenan luego en una estructura de datos flexible, comúnmente conocida como DataFrame, desde donde se pueden manipular posteriormente. Si bien estas estructuras de datos son simples y fáciles de manipular, no brindan al científico de datos protección contra errores posteriores, como suposiciones incorrectas sobre tipos o nulos en los datos.

TransmogrifAI intenta resolver esta situación permitiendo a los usuarios especificar un esquema para sus datos y extrayendo automáticamente el

predictor sin procesar y las señales de respuesta como características. Las características están fuertemente tipadas y TransmogrifAI admite una jerarquía de tipos de características rica y extensible. Esta jerarquía va más allá de los tipos primitivos para admitir tipos más específicos, como ubicación geográfica, números de teléfono, códigos postales y más, diferenciando los tipos que los científicos de datos querrían tratar de manera diferente. Además de permitir tipos especificados por el usuario, TransmogrifAI también hace inferencias propias. Por ejemplo, si detecta que una característica de texto con baja cardinalidad es de hecho una característica categórica disfrazada, la cataloga y la trata de manera apropiada. Las características fuertemente tipadas permiten a los desarrolladores detectar la mayoría de los errores en tiempo de compilación en lugar de en tiempo de ejecución.

El segundo paso en la concepción de AutoML según TransmogrifAI es lo los autores denominan "transfiguración", que es también conocida como ingeniería de características automatizada. Tener características fuertemente tipadas permite razonar sobre los datos y minimizar los errores posteriores, pero eventualmente, todas las características deben transformarse en una representación numérica que exponga las regularidades de los datos de una manera que se puede aprovechar fácilmente mediante algoritmos de aprendizaje automático. Este es el proceso que se denomina ingeniería de características. Hay infinitas formas de transformar los tipos de características que pueden aparecer en un problema de aprendizaje de máquina, y hacerlo de la manera correcta es el arte de la ciencia de datos.

La ingeniería automática de docenas de tipos de características diferentes en vectores numéricos es lo que le da a TransmogrifAI su nombre. El sistema propone una gran variedad de técnicas para todos los tipos de funciones compatibles, desde números de teléfono, direcciones de correo electrónico, ubicaciones geográficas hasta datos de texto. Estas transformaciones no solo pretenden representar los datos en un formato que los algoritmos puedan usar, sino también optimizar las transformaciones para facilitar que los algoritmos de aprendizaje automático aprendan de los datos. Además de proporcionar técnicas predeterminadas, el sistema está diseñado para facilitar la contribución y el intercambio rápido de técnicas de ingeniería de funciones para que los desarrolladores puedan personalizar y ampliar los valores predeterminados de forma reutilizable.

El tercer paso automatizado en TransmogrifAI es la validación de características. La ingeniería de características puede resultar en una explosión en la dimensionalidad de los datos, lo que acarrea múltiples problemas. Por ejemplo, el uso de campos particulares en los datos puede haber variado con el tiempo, y los modelos entrenados en estos campos pueden funcionar mal con datos nuevos. Otro problema es el sesgo retrospectivo o la fuga de datos [96], que ocurre cuando la información que en realidad no estará presente en el momento de la predicción se filtra en los ejemplos de entrenamiento. El resultado son modelos que con un rendimiento muy alto durante en el entrenamiento pero inútiles en la práctica.

TransgmogrifAI tiene algoritmos que realizan la validación automática de características para eliminar características con poco o ningún poder predictivo: características cuyo uso se ha desviado con el tiempo, características que presentan una variación cero o características cuya distribución en los ejemplos de entrenamiento varía significativamente de su distribución en el momento de la predicción. Estos algoritmos son especialmente útiles para preservar la calidad de los modelos cuando se trabaja con datos de gran dimensión y desconocidos que pueden contener sesgos retrospectivos. El sistema aplican pruebas estadísticas basadas en tipos de características y, además, utiliza todo el historial de transformaciones de cada característica para detectar y descartar dicho sesgo.

Como cuarto paso, TransmogrifAI también permite la selección automática de modelos. El selector de modelos de TransmogrifAI ejecuta un torneo de varios algoritmos de aprendizaje automático diferentes en los datos y utiliza el error de validación promedio para elegir automáticamente el mejor. También se ocupa automáticamente del problema de los datos desequilibrados mediante un muestreo adecuado de los datos y recalibrando las predicciones para que coincidan con la distribución a priori.

Subyacente a todas las etapas anteriores hay una capa de optimización de hiperparámetros. A diferencia de otros sistemas que consideran hiperparámetros solamente en los modelos, TransmogrifAI ajusta los hiperparámetros en todo el flujo de AutoML. Por ejemplo, durante la ingeniería de características, se puede ajustar el número de variables binarias que se derivan de un predictor categórico. La frecuencia de muestreo para tratar con datos desequilibrados es otro hiperparámetro que se puede ajustar. TransmogrifAI

implementa estrategias de búsqueda simples, como la búsqueda aleatoria, para ajustar automáticamente los hiperparámetros. Además, el sistema puede ser extendido fácilmente para incorporar estrategias de búsqueda más avanzadas.

TransmogrifAI se creó con el objetivo de mejorar la productividad de los desarrolladores de aprendizaje automático, no solo a través de la automatización del aprendizaje automático, sino también a través de una API que refuerza la seguridad de tipos en tiempo de compilación, la modularidad y la reutilización. Por este motivo, a diferencia de otros sistemas de AutoML, presenta algunas decisiones de diseño notables en este sentido.

En primer lugar, TransmogrifAI ejecuta sobre Apache Spark[1], plataforma de cómputo distribuído que fue seleccionada por los diseñadores de este sistema por varias razones. La primera razón es la necesidad de manejar grandes variaciones en el tamaño de los datos. Spark tiene primitivas para lidiar con combinaciones y agregados distribuidos en big data que dan soporte a la escalabilidad de TransmogrifAI. La segunda razón es para permitir la ejecución de modelos tanto en lotes como en *streaming* continuo de datos. Spark Streaming[2] permite extender TransmogrifAI para que funcione en ambos modos. Finalmente, al construir sobre una biblioteca activa de código abierto, aprovechan las mejoras continuas que se realizan constantemente en esa biblioteca.

En segundo lugar, TransmogrifAI propone una abstracción de características. Una característica es esencialmente un puntero de tipo seguro a una columna en un *DataFrame* y contiene toda la información sobre esa columna: su nombre, el tipo de datos que contiene, así como información sobre cómo se derivó. Las características se convierten en la primitiva principal con la que los desarrolladores interactúan, y definir y manipular características se parece más a trabajar con variables en un lenguaje de programación que a manipular columnas en un *DataFrame*. Esta funcionalidad diferencia a Transmogrif de otros sistemas de AutoML, que conciben los datos de entrada directamente como valores numéricos sin semántica de tipos explícita, y por tanto pueden aplicar algoritmos que no tienen sentido sobre ciertos tipos de datos.

[1] https://spark.apache.org/

[2] https://spark.apache.org/docs/latest/streaming-programming-guide.html

Por otro lado, las características en TransmogrifAI están fuertemente tipadas. Esto permite realizar comprobaciones de tipo en todo el flujo de aprendizaje automático y garantizar que los errores se detecten lo antes posible, incluso antes de ejecutar los algoritmos de aprendizaje. La seguridad de tipos también viene con otras ventajas para la productividad del desarrollador, incluida la capacidad de permitir que los IDE inteligentes sugieran la finalización del código. La seguridad de tipos también aumenta la transparencia en torno a las entradas y salidas esperadas en cada etapa de un flujo de aprendizaje automático. Finalmente, los tipos de características son clave para el procesamiento posterior específico del tipo, particularmente para la ingeniería de características automatizadas y la validación de características.

Otro criterio de relevancia en el diseño de TransmogrifAI es la personalización y extensibilidad. Los usuarios pueden hacer uso de los estimadores automáticos para generar rápidamente modelos de rendimiento, pero si desean un mayor control, cada algoritmo está parametrizado y sus parámetros pueden ser configurados y ajustados directamente por el usuario del sistema. Además, se pueden especificar fácilmente transformadores y estimadores personalizados que se utilizarán en un flujo.

Por último, TransmogrifAI ha sido diseñado teniendo en cuenta el rendimiento a gran escala. Con la ingeniería de funciones automatizada, es posible que el espacio de características crezca exponencialmente dando como resultado datos de alta dimensionalidad que son difíciles de manejar para Spark. Los flujos de trabajo de TransmogrifAI abordan esto al inferir el Grafo Dirigido y Acíclico (DAG) completo de transformaciones que se necesitan para computar cada característica y optimizar la ejecución de este DAG al contraer todas las transformaciones que ocurren en el mismo nivel del DAG en una sola operación. Al mismo tiempo, dado que está construido sobre Spark, TransmogrifAI se beneficia automáticamente de las mejoras continuas en la optimización subyacente de Spark DataFrame.

Hyperopt-Sklearn

Hyperopt-Sklearn [17] es un sistema AutoML que brinda los beneficios de la configuración automática de algoritmos a los usuarios de Python y scikit-learn [74]. Hyperopt-Sklearn usa Hyperopt [97] para describir un espacio de búsqueda sobre posibles configuraciones de componentes de scikit-learn,

incluidos módulos de preprocesamiento, clasificación y regresión. Una de las principales características de diseño de este proyecto es proporcionar una interfaz familiar para los usuarios de scikit-learn. Con muy pocos cambios, la búsqueda de hiperparámetros se puede aplicar a una base de código existente.

La biblioteca Hyperopt ofrece algoritmos de optimización para los espacios de búsqueda que surgen en la configuración del algoritmo. Estos espacios se caracterizan por una variedad de tipos de variables (continuas, ordinales, categóricas), diferentes perfiles de sensibilidad (p. ej., Escala uniforme frente a logarítmica) y estructura condicional (cuando se puede elegir entre dos clasificadores, los parámetros de un clasificador son irrelevante cuando se elige el otro clasificador). Para usar Hyperopt, un usuario debe definir / elegir tres cosas:

- Un dominio de búsqueda
- Un algoritmo de optimización
- Una función objetivo

El dominio de búsqueda se especifica mediante variables aleatorias, cuyas distribuciones deben elegirse de manera que las combinaciones más prometedoras tengan una alta probabilidad previa. El dominio de búsqueda puede incluir operadores y funciones de Python que combinan variables aleatorias en estructuras de datos más convenientes para la función objetivo. Cualquier estructura condicional se define dentro de este dominio. La función objetivo mapea un muestreo conjunto de estas variables aleatorias a una puntuación de valor escalar que el algoritmo de optimización intentará minimizar.

Habiendo elegido un dominio de búsqueda, una función objetivo y un algoritmo de optimización, la función `fmin` de Hyperopt lleva a cabo la optimización y almacena los resultados de la búsqueda en una base de datos (por ejemplo, una lista de Python simple o una instancia de MongoDB [98]). La llamada a `fmin` lleva a cabo el análisis simple de encontrar la configuración de mejor rendimiento y se la devuelve al método que realiza la llamada. La llamada a `fmin` puede usar varios hilos o procesos cuando se usa el backend de MongoDB, para implementar la selección de modelos paralelos en un clúster de cómputo.

El enfoque básico es configurar un espacio de búsqueda con hiperparámetros de variables aleatorias, usar scikit-learn para implementar la función objetivo que realiza el entrenamiento y la validación del modelo, y usar Hyperopt para optimizar los hiperparámetros. Scikit-learn incluye muchos algoritmos para aprender a partir de datos (clasificación o regresión), así como muchos algoritmos para preprocesar datos en los vectores esperados por estos algoritmos de aprendizaje. Los clasificadores incluyen, por ejemplo, los algoritmos KNearest-Neighbors [77], Support Vector Machines [29] y Random Forest [71]. Los algoritmos de preprocesamiento incluyen transformaciones como normalización por componentes y Análisis de componentes principales (PCA) [99]. Un algoritmo de clasificación completo normalmente incluye una serie de pasos de preprocesamiento seguidos de un clasificador. Por esta razón, scikit-learn proporciona una estructura de datos para representar flujos de algoritmos y usar una secuencia de pasos de preprocesamiento y un clasificador como si fueran solo un componente (generalmente con una API similar al clasificador). Aunque hyperopt-sklearn no usa formalmente el objeto de flujo de scikit-learn, proporciona una funcionalidad relacionada. Hyperopt-sklearn proporciona una parametrización de un espacio de búsqueda sobre flujos, es decir, de secuencias de pasos de preprocesamiento y clasificadores o regresores.

El espacio de configuración incluye actualmente 24 clasificadores, 12 regresores y 7 métodos de preprocesamiento. Al ser un proyecto de código abierto, es posible que este espacio se expanda en el futuro a medida que contribuyan más usuarios. En total, esta parametrización contiene 65 hiperparámetros: 15 variables booleanas, 14 categóricas, 17 discretas y 19 variables de valor real. Aunque el número total de hiperparámetros en el espacio de configuración completo es grande, el número de hiperparámetros activos que describen cualquier modelo es mucho menor: un modelo que consta de PCA y un RandomForest, por ejemplo, tendría solo 12 hiperparámetros activos (1 para la elección de preprocesamiento, 2 internos a PCA, 1 para la elección de clasificador y 8 internos a RandomForest). El lenguaje de descripción de hiperparámetros permite diferenciar entre hiperparámetros condicionales (que siempre deben asignarse) e hiperparámetros no condicionales (que pueden permanecer sin asignar cuando no se usarían).

Hyperopt-sklearn hace un uso extensivo de este mecanismo para que los algoritmos de búsqueda de Hyperopt no pierdan tiempo aprendiendo por

ensayo y error que, por ejemplo, los hiperparámetros de RandomForest no tienen ningún efecto sobre el rendimiento de SVM [29]. Esta característica es común a todos los sistemas de AutoML que definen un espacio jerárquico. Sin embargo, en Hyperopt-Sklearn incluso internamente dentro de los clasificadores, hay instancias de parámetros condicionales: KNN [77] tiene parámetros condicionales que dependen de la métrica de distancia, y LinearSVC [29] tiene 3 parámetros binarios (pérdida, penalización y dual) que admiten solo 4 asignaciones conjuntas válidas. Hyperopt-sklearn también incluye una lista negra de pares (preprocesamiento, clasificador) que no funcionan juntos. Por ejemplo, PCA y MinMaxScaler son incompatibles con MultinomialNB [78], TF-IDF [100] solo se puede usar para datos de texto y los clasificadores basados en árboles no son compatibles con las escasas características producidas por el preprocesador TF-IDF. Permitiendo una discretización de 10 niveles de hiperparámetros de valor real, y teniendo en cuenta estos hiperparámetros condicionales, una búsqueda exhaustiva en espacio de búsqueda aún requeriría un número inviable de evaluaciones (del orden de 10^12).

Finalmente, el espacio de búsqueda se convierte en un problema de optimización cuando se define define un objetivo de búsqueda con valores escalares. De forma predeterminada, Hyperopt-sklearn utiliza el método de evaluación de scikit-learn en los datos de validación para definir el criterio de búsqueda. Para los clasificadores, esta es la llamada "pérdida cero-uno": el número de predicciones de etiquetas correctas entre los datos que se han retenido del conjunto de datos utilizado para el entrenamiento (y también de los datos utilizados para las pruebas después del proceso de búsqueda de selección de modelo).

Es posible que un usuario agregue su propio clasificador al espacio de búsqueda siempre que se ajuste a la interfaz scikit-learn. Actualmente, esto requiere cierta comprensión de cómo está estructurado el código de hyperopt-sklearn, por lo que es necesario mejorar el soporte para la extensibilidad de forma que el usuario requiera un esfuerzo mínimo. También es posible que el usuario especifique funciones de evaluación alternativa además de la precisión predeterminada.

RECIPE

RECIPE [16] es un sistema de AutoML basado en un método de programación genética de gramáticas (GGP, *grammar genetic programming*) que genera flujos de algoritmos de clasificación [51]. El sistema recibe como entrada un conjunto de datos y una gramática libre del contexto (CFG), que se utiliza para inicializar una población de flujos. Cada individuo está representado por un árbol de derivación construido a partir de la gramática libre de contexto, que abarca todo el conocimiento recopilado de los especialistas sobre cómo generar un flujo de clasificación eficaz. Los individuos se mapean a flujos implementados en forma de clases de la biblioteca scikit-learn [74], que se ejecutan en una muestra de datos de la aplicación que se resuelve y evalúa de acuerdo con una métrica de precisión. Los operadores de cruzamiento y mutación se aplican después de una selección de torneo y garantizan que los nuevos individuos generados también respeten las reglas de producción de la gramática. La estrategia de búsqueda utiliza elitismo, y la evolución continúa hasta que se alcanza un número máximo de generaciones o el mejor individuo no mejora después de un número predefinido de generaciones.

En los sistemas GGP, una gramática es una forma eficaz de representar el conocimiento previo sobre el problema que se está resolviendo [51]. Una gramática G está representada por una tupla $< N, T, P, S >$ donde N representa un conjunto de no terminales, T un conjunto de terminales, P un conjunto de reglas de producción y $S \in N$, el símbolo de inicio. Las reglas de producción definen el lenguaje que representa la gramática combinando los símbolos gramaticales. En RECIPE, la gramática representa un conjunto de flujos que se pueden usar para resolver un problema de clasificación. Como se mencionó anteriormente, un flujo es una secuencia de tareas que deben aplicarse a un conjunto de datos dado para producir soluciones precisas a un problema de entrada dado.

Los sistemas de AutoML generalmente dividen los flujos de AutoML en tres pasos principales: preprocesamiento, procesamiento y posprocesamiento de datos respectivamente. RECIPE también sigue este marco básico. Los pasos de preprocesamiento de datos incluyen realizar transformaciones en los datos de entrada para hacerlos más adecuados para la tarea de aprendizaje. Estos métodos incluyen normalización de datos, elección de características, imputación de datos, entre otros. El segundo paso es el núcleo del flujo

y el único componente obligatorio. Implica la elección del algoritmo de clasificación y sus parámetros. Entre los métodos que se pueden utilizar se encuentran los que generan diferentes tipos de modelos de conocimiento, como Naive Bayes [78], SVM [29], árboles de decisión [71], redes neuronales [91], entre otros. Finalmente, la etapa de posprocesamiento se puede aplicar cuando se prueba más de un algoritmo en el segundo paso, Deep Learning ene 147 reglas de producción, en un total de 146 no terminales y 239 terminales, que corresponden a algoritmos o hiperparámetros de la biblioteca scikit-learn.

En la estrategia de búsqueda de RECIPE, basada en un algoritmo genético, los individuos representan flujos de aprendizaje automático centrados en la tarea de clasificación. Estos individuos se generan a partir de la gramática mediante un conjunto de pasos de derivación. Cada individuo es una representación de un flujo de clasificación de datos, y su evaluación implica ejecutar el flujo en una muestra del conjunto de datos de interés. Por lo tanto, el primer paso del proceso de evaluación es generar un flujo ejecutable a partir de la representación individual.

Después de este proceso de traducción del árbol de derivación gramatical a un flujo ejecutable, el flujo se ejecuta utilizando un procedimiento de validación cruzada triple creado sobre el conjunto de entrenamiento, es decir, se generan tres nuevos subconjuntos de entrenamiento y validación. Durante esta fase, cada conjunto de entrenamiento se utiliza para entrenar el modelo correspondiente, y el conjunto de validación se utiliza para calcular su rendimiento, definido como una F-mDeep Learning evaluación. Además, los conjuntos de entrenamiento y validación se vuelven a muestrear cada cinco generaciones para evitar el sobreajuste. La medida F, utilizada como función de rendimiento, es la media armónica entre las métricas de precisión y recuperación definidas en un problema de clase binaria. Esta métrica fue seleccionada por los autores ya que permite considerar diferentes niveles de desequilibrio de clases al evaluar el modelo.

KNIME

KNIME AutoML[3] entrena automáticamente modelos de aprendizaje automático supervisados para clasificación binaria y multiclase. KNIME es

[3] https://hub.knime.com/knime/spaces/Examples/latest/00_Components/Automation/AutoML

capaz de automatizar todo el ciclo de ML realizando alguna preparación de datos, optimización de parámetros con validación cruzada, puntuación, evaluación y selección. También captura todo el proceso de un extremo a otro y genera el flujo de trabajo de implementación mediante KNIME Integrated Deployment Extension.

Este sistema cuenta con una interfaz gráfica que permite arrastrar y soltar el componente desde KNIME Hub. Luego de suministrarle el acceso a su tabla de datos y a la columna que se desea predecir la herramienta ejecuta permitiendo que los modelos entrenen, afinen, y validen para seleccionar y exportar el mejor modelo encontrado. Permite inspeccionar el modelo implementado y personalizarlo. También puede inspeccionar todos los modelos entrenados aunque desafortunadamente requiere entrenar a todos los modelos nuevamente.

Antes de entrenar los modelos, los datos se limpian reemplazando los valores faltantes con el valor más frecuente de la columna categórica o la media de las columnas numéricas. Opcionalmente, los datos categóricos se pueden codificar en un solo uso y las columnas con demasiados valores únicos se eliminan según un parámetro definido por el usuario. Las características numéricas se convierten todas a doble precisión flotante y se normalizan mediante la normalización mínima-máxima. Los datos se dividen automáticamente en dos particiones de prueba y entrenamiento utilizando una técnica de muestreo estratificado en la clase objetivo y una división del 80 %. Los modelos de preparación de datos se almacenan para su implementación tanto para el preprocesamiento como para el posprocesamiento de los datos en torno al predictor del modelo.

Cada modelo tiene una serie de parámetros que se deben ajustar mediante la validación cruzada y la métrica de evaluación definida por el usuario en los datos de entrenamiento. El alcance de la optimización de los parámetros, la estrategia de optimización y otras configuraciones del modelo se pueden cambiar directamente en el sistema.

- Naive Bayes [78]: entrenado con un parámetro optimizado de probabilidad por defecto.
- Regresión logística [84]: entrenado con un parámetro optimizado de tamaño de paso.

- Red neuronal [91]: un perceptrón multicapa (MLP) entrenado con los parámetros optimizados del número de capas ocultas y el número de neuronas ocultas por capa.
- *Gradient Boosting Trees* [83]: entrenados con el parámetro optimizado del número de árboles.
- Árbol de decisión: entrenado con el parámetro optimizado del número mínimo de registros por nodo.
- Bosque aleatorio [71]: entrenado con los parámetros optimizados de profundidad del árbol, número de modelos y tamaño mínimo del nodo secundario.
- Árboles XGBoost [83]: entrenados con parámetros optimizados de factor de aprendizaje y profundidad máxima.
- Modelo lineal generalizado (H2O) [75]: entrenado con la integración de aprendizaje automático de KNIME para H2O [90].
- Redes neuronales (Keras): entrenado con la intregración de KNIME para Keras, sin optimización de parámetros y dos arquitecturas simples para clasificación binaria y multiclase. Se han implementado algunas heurísticas simples para dar forma a la arquitectura de red y al proceso de entrenamiento de Keras en función del tamaño de los datos de entrada.

Una vez que se completa el entrenamiento de los modelos especificados y todos los modelos se almacenan en una sola tabla, el sistema aplica el modelo al conjunto de prueba. Las predicciones de todos los modelos se puntúan en función del valor resultante en el conjunto de prueba y se calculan varias métricas de rendimiento. El mejor modelo se selecciona utilizando la métrica de rendimiento especificada por el usuario.

El preprocesamiento de datos, el mejor modelo y el posprocesamiento de datos se capturan a través de KNIME Integrated Deployment Extension. El flujo de algoritmos encapsulado de un extremo a otro se proporciona en la salida del componente y se puede utilizar para procesar nuevos datos en la implementación.

ML-Plan

ML-Plan [19] es un sistema de AutoML con un enfoque novedoso basado en redes de tareas jerárquicas (HTN) [101]. HTN es una técnica de planificación generalmente implementada como una búsqueda heurística del mejor primero (*best-first search*) sobre el gráfico inducido por el problema de planificación. En ML-Plan, la búsqueda está guíada aleatoriamente, construyendo flujos parciales como los de Auto-WEKA [23] y Auto-Sklearn [13]. De esta manera, ML-Plan ofrece una solución intermedia que combina ideas y conceptos de diferentes enfoques. ML-Plan también incluye un mecanismo explícito para evitar el sobreajuste, basado en un modelo de búsqueda de dos fases (búsqueda + selección).

La técnica de planificación usada en ML-Plan no se limita a un tipo particular de problema de aprendizaje, aunque su implementación actual se centra en la clasificación de clases múltiples. Dependiendo del problema de aprendizaje, la adaptación a otros entornos puede ser más sencilla (p. ej., para regresión o clasificación) o puede requerir algunos cambios en la rutina (p. ej., en la predicción de salida estructurada, donde no se pueden permitir todas las combinaciones de preprocesamiento, o sin supervisión).

Formalmente, el problema de AutoML en ML-Plan es modelado como encontrar un flujo de algoritmos de aprendizaje automático que aprenda a asociar elementos de salida de un espacio Y (en este caso clases) con objetos de entrada de un espacio de instancia X. A menudo, $X = R^d$ para algún entero d, es decir, las instancias son descrito en términos de d atributos numéricos (características); sin embargo, las características también pueden ser binarias o categóricas. Un conjunto de datos es un subconjunto (finito) $D = (x_i, y_i) \in X \times Y$.

En ML-Plan se definen dos tipos de algoritmos. Primero, los preprocesadores son funciones ϕ que mapean conjuntos de datos D a conjuntos de datos D', posiblemente cambiando el espacio de representación de $X \times Y a X' \times Y'$. Ejemplos de tales funciones incluyen métodos para la reducción de dimensionalidad (como análisis de componentes principales), selección de características, imputación, discretización, normalización, etc. En segundo lugar, los algoritmos de aprendizaje son funciones que mapean conjuntos de datos D a una función predictora $\Phi : X \to Y$. Un flujo para ML-Plan es un par que consta de un preprocesador parametrizado y un algoritmo de aprendizaje. Ambos tipos

de algoritmos pueden tener hiperparámetros continuos, discretos, ordinales o nominales.

Dado un conjunto de datos etiquetados D, la tarea consiste en combinar los algoritmos anteriores en un flujo C que, tomando los datos de entrenamiento D como entrada, produce un predictor óptimo $\Phi = C(D)$ como salida. Como en el resto de los sistemas, la optimización se enfoca en el rendimiento de generalización, es decir, el error esperado causado por Φ cuando se usa para predecir etiquetas de clase en datos nuevos (no contenidos en los datos de entrenamiento, pero producidos por el mismo proceso de generación de datos).

El algoritmo básico de planificación de HTN es un lenguaje de operadores de planificación [102]. El lenguaje tiene capacidades lógicas de primer orden libres de funciones, es decir, define un conjunto infinito de nombres de variables, nombres de constantes, nombres de predicados y cuantificadores y conectores para construir fórmulas. Un operador es una tupla $< name_o, pre_o, post_o >$, donde $name_o$ es un nombre y pre_o y $post_o$ son fórmulas de que constituyen precondiciones y poscondiciones, respectivamente. Por ejemplo, un operador PCA puede realizar un análisis de componentes principales en un conjunto de datos dado; pre_o especificaría las condiciones bajo las cuales el operador es aplicable, y $post_o$ el efecto que logra.

Un plan es una secuencia de operaciones base. Una operación es base si todas las variables en la condición pre_o y $post_o$ han sido sustituidas por términos que solo contienen constantes. Los operadores base también se denominan acciones.

La semántica de una acción es que modifica el estado en el que se aplica (por ejemplo, convertir atributos numéricos en discretos). Un estado es un conjunto de literales positivos básicos. En ML-Plan se asume que todo literal fundamental que no esté contenido explícitamente en un estado es falso.

Una red de tareas jerárquica (HTN) es un conjunto T de tareas parcialmente ordenado. Una tarea $t = (v_0, \ldots, v_n)$ es un nombre con una lista de parámetros, que son variables o constantes. Por ejemplo, `configureC45(c)` podría ser la tarea de crear un conjunto de opciones para un árbol de decisiones y asignarlas al árbol de decisiones `c`. Una tarea nombrada por un operador se llama primitiva, de lo contrario es compleja.

El procesor de optimización en ML-Plan se basa en refinar tareas com-

plejas de forma iterativa hasta llegar a una red de tareas que solo tiene tareas primitivas básicas, es decir, un conjunto de acciones parcialmente ordenadas. Si bien las tareas primitivas se pueden realizar mediante una sola operación, las tareas complejas deben descomponerse mediante métodos. Un método $m =< name_m, task_m, prem, T_m >$ consta de su nombre, la tarea (no primitiva) $task_m$ que refina, una precondición lógica pre_m y una red de tareas T_m que realiza la descomposición. Reemplazando las tareas complejas por la red de los métodos usados para descomponerlas, se derivan iterativamente nuevas redes de tareas hasta obtener una con tareas primitivas (acciones) solamente.

La definición de un problema de planificación de red de tareas simple es sencilla. Dado un estado inicial s_0 y una red de tareas T_0, el problema de planificación es derivar un plan de T_0 que sea aplicable en s_0. Un problema de planificación de red de tareas simple es un cuádruple $< s_0, T_0, O, M >$, donde O y M son conjuntos finitos de operadores y métodos, respectivamente.

Los problemas de HTN generalmente se resuelven reduciendo a un problema de búsqueda en grafos que se puede abordar con algoritmos como la búsqueda en profundidad (*depth-first search*), la búsqueda mejor primero (*best-first search*), etc. Una traducción típica del problema de HTN en un grafo consiste en seleccionar la primera tarea compleja en la red de un nodo y definir un sucesor para cada método que se puede utilizar para resolver la tarea; esto se llama descomposición directa [102]. Cada nodo del grafo resultante corresponde a un prefijo de plan (la parte del plan que se ha corregido) junto con las tareas restantes. El nodo raíz tiene un plan vacío con la red de tareas inicial y los nodos objetivo tienen planes de solución y redes de descanso vacías. Por lo general, existe una correspondencia uno a uno entre los elementos del espacio de búsqueda, por ejemplo, los flujos de aprendizaje automático y los nodos objetivo.

ML-Plan reduce AutoML a un problema de búsqueda de grafos mediante la planificación HTN. Más específicamente, ML-Plan invoca un algoritmo de búsqueda del mejor primero (*best-first search*) [103] en el grafo inducido por una descomposición directa del problema de planificación de HTN. ML-Plan codifica un problema HTN que divide el problema AutoML definido en una fase de selección de algoritmos y otra de configuración de algoritmos. ML-Plan se inicializa con un conjunto fijo de algoritmos de preprocesamiento,

algoritmos de clasificación y los parámetros respectivos con sus dominios. La primera fase es decidir el algoritmo de preprocesamiento de características (si lo hay) y luego el algoritmo de clasificación. A la inversa, la segunda fase configura primero el algoritmo de clasificación y luego el algoritmo de preprocesamiento (si lo hay).

Estas fases no deben entenderse como fases del algoritmo en el sentido de que ML-Plan primero elige los algoritmos y luego los configura, sino como fases (regiones) del grafo de búsqueda. Más precisamente, la formulación del problema HTN induce una parte superior e inferior del grafo de búsqueda; esto es lo que se entiende en ML-Plan por "fases". ML-Plan adopta una búsqueda global en primer lugar dentro de ese grafo y, por lo tanto, no elige de forma *greedy* algoritmos y luego los configura. Con el tiempo suficiente, ML-Plan detectará todas las soluciones.

La reducción realizada por ML-Plan no es canónica. De hecho, hay muchos problemas de HTN diferentes que pueden cubrir exactamente el mismo espacio de búsqueda. Por tanto, la forma en que se formula el problema de la HTN puede tener un impacto notable en la eficiencia de la búsqueda. Por ejemplo, además de la técnica anterior, también podríamos usar una red de dos pasos donde primero se elige y configura el preprocesador (o se elige no usar ninguno) y luego se elige y configura el clasificador. Si bien esto parece una alteración trivial que no influye en el conjunto de flujos construibles, tiene importantes consecuencias en la estructura del árbol de búsqueda.

En su implementación actual, ML-Plan elige los valores de los parámetros de un pequeño conjunto predefinido de valores posibles. Con este fin, los parámetros numéricos se discretizan en una escala lineal o en una escala logarítmica. La técnica de intervalo y discretización para un parámetro no está sujeta a optimización sino que se fija de antemano.

Como ya se ha indicado, ML-Plan adopta un algoritmo de búsqueda del mejor primero para identificar buenos flujos. Un algoritmo de búsqueda del mejor primero explora un grafo dado implícitamente asignando un número a cada nodo y eligiendo en cada iteración el nodo con el mejor valor conocido actualmente (generalmente el más bajo) para la expansión. La expansión significa calcular todos los sucesores de un nodo. La descripción del grafo consta del nodo raíz, la función de cálculo sucesora utilizada para la expansión y un predicado sobre los nodos que indica si un nodo es un nodo objetivo o no.

La tarea consiste en encontrar una ruta desde la raíz hasta un nodo objetivo con una puntuación mínima, y un algoritmo del mejor primero intenta encontrar dicha ruta expandiendo los nodos intermedios con puntuaciones mínimas.

ML-Plan adopta una búsqueda aleatoria en profundidad similar a la aplicada en la búsqueda de árboles de Monte Carlo (MCTS) [104]. Dado que la función de evaluación de nodos calcula soluciones (flujos de algoritmos de aprendizaje), estas soluciones se propagan al algoritmo de búsqueda. En concreto se propaga la mejor de las n soluciones extraídas para cada nodo a la rutina de búsqueda. De esta manera, ML-Plan (siempre que se haya evaluado al menos un nodo) siempre puede devolver soluciones incluso si la rutina de búsqueda principal no descubrió ningún nodo objetivo.

ML-Plan admite dos procedimientos para determinar las cualidades de las soluciones:

Validación cruzada k-fold (CV) Esta es la validación cruzada estándar utilizada por muchos enfoques. En este caso solo se aplica a la parte de los datos que se asignan para la búsqueda. El conjunto de datos se divide en k subconjuntos (definido por el usuario) y, en k iteraciones, el procedimiento de validación usa k - 1 de ellos para el entrenamiento y el restante para la validación. El rendimiento es entonces el rendimiento medio de las k ejecuciones.

Validación cruzada de Monte Carlo (MCCV) Esta técnica también se denomina retención". Los datos (asignados para la búsqueda) se dividen k veces en un conjunto de validación y entrenamiento estratificado. Para cada una de las k divisiones, el flujo solución se entrena con el conjunto de entrenamiento respectivo y se prueba en el conjunto de validación. El error medio de clasificación en esta evaluación es la puntuación de dicha solución.

Aunque la técnica de evaluación influye en el rendimiento general del algoritmo, no está diseñada para ser optimizada por el usuario, sino para brindar flexibilidad para el análisis científico. De hecho, ML-Plan está configurado para usar MCCV con 5 iteraciones, cada una de las cuales con una división del 70 % para entrenamiento y 30 % para validación.

Las estimaciones adquiridas por la estrategia anterior producen resultados ruidosos cuando se utilizan en la región superior del grafo de búsqueda. Esto conduce a que los subárboles con buenas soluciones a veces se podan solo porque todas las terminaciones del nodo superior de ese subárbol condujeron a soluciones subóptimas. Esto sucede porque los nodos de la capa superior abarcan muchos tipos diferentes de soluciones, por lo que las estimaciones pueden desviarse significativamente de la mejor puntuación que se puede obtener en un subárbol correspondiente.

Por este motivo, ML-Plan fue diseñado para expandir todos los nodos para la parte de selección del algoritmo del árbol de búsqueda sin calcular ninguna evaluación y adopta las terminaciones aleatorias solo para los nodos en las capas más profundas correspondientes a las decisiones de configuración del algoritmo. En concreto, ML-Plan asigna un valor de 0 (óptimo) a todos los nodos en los que aún no se ha elegido el clasificador. Esto implica deshabilitar la búsqueda informada para la parte de selección de algoritmo, pero dado que generalmente solo son posibles unos pocos cientos de selecciones de algoritmo, todas estas posibilidades se pueden enumerar de manera eficiente. La técnica de finalización aleatoria solo se aplica entonces a los nodos por debajo de la región de selección del algoritmo. Esta estrategia asegura que cada combinación de preprocesadores y clasificadores se considere al menos una vez con finalizaciones aleatorias y también aumenta la confiabilidad de estas estimaciones.

Para escoger entre las diferentes selecciones de algoritmos, ML-Plan define un preorden (preferencial) en los algoritmos de clasificación. Este orden se utiliza para clasificar los nodos hoja de la región de selección del algoritmo y, por tanto, define en qué orden se llevan a cabo las primeras evaluaciones de finalización aleatorias. El orden es similar al utilizado en Auto-WEKA: KNN [77], *Random Forest* [71], perceptrón [91], SVM [29], regresión logística [84].

Dado que ML-Plan usa dos funciones de evaluación de nodos diferentes dentro del mismo grafo, estas deben ser consistentes para evitar comportamientos extraños. Para ello, las puntuaciones de la parte superior se escalan por el factor $10^{-}3$. De esta manera, están en una escala consistente con las estimaciones de precisión y se prefieren (a menos que ML-Plan obtenga una estimación para una solución con una tasa de error inferior a $10^{-}3$, lo que

indicaría una solución casi perfecta).

Intuitivamente, una búsqueda extensa y sistemática de buenos predictores debería conllevar un fuerte riesgo de sobreajuste. Debido a su capacidad para elegir entre todos los clasificadores y construir clasificadores nuevos y arbitrariamente complejos utilizando métodos de ensemble, las herramientas de AutoML se encuentran en el extremo de este espectro. Si los datos disponibles para el proceso de búsqueda no son lo suficientemente representativos de los datos reales, el peligro de un ajuste excesivo en AutoML es mayor que para los algoritmos de aprendizaje básicos.

Este problema en ML-Plan se ataca con un mecanismo de búsqueda de dos fases. La primera fase cubre la búsqueda real en el espacio descrito anteriormente y produce una colección de soluciones candidatas. La segunda fase toma estos candidatos y selecciona el que minimiza el error de generalización estimado. Esta estimación se logra dividiendo los datos proporcionados a la herramienta AutoML en dos conjuntos D_{search} y D_{select}. La fase 1 solo tiene acceso a D_{search}, que se utiliza para la evaluación de nodos como se describe anteriormente. La fase 2 realiza la validación cruzada de Monte Carlo en $D_{search} \bigcup D_{select}$ para un número fijo de iteraciones. Para cada iteración se construye una división estratificada (70 % de entrenamiento y 30 % de validación) que se utiliza para entrenar y evaluar una solución candidata. El error de generalización se estima tomando el promedio de la evaluación interna de un flujo s como se describe anteriormente (solo en D_{search}), y el percentil 0,75 de las evaluaciones que incluyen D_{select}. Esto provoca que la estimación de la generalización sea más conservadora pero también robusta frente a los valores atípicos.

Dado que la fase 1 puede detectar cientos o incluso miles de modelos, la fase 2 solo opera en un pequeño subconjunto de estas soluciones. Las soluciones utilizadas en la segunda fase constan de dos subconjuntos igualmente grandes S_{best} y S_{random}. El tamaño de estos conjuntos se fija mediante un parámetro k. S_{best} y S_{random} contienen, respectivamente, las k mejores soluciones, y soluciones aleatorias cuya evaluación interna se desvía como máximo ϵ de la óptima. Los candidatos aleatorios son importantes para garantizar una cierta diversidad en el conjunto de selección, pero la calidad esperada debe ser razonablemente buena. Dado que el dominio de la función de error de predicción se fija en $[0, 1]$, ϵ no es una desviación relativa sino

absoluta del óptimo, establecida en ML-Plan como $\epsilon = 0{,}03$.

Esta estrategia de prevención de sobreajuste tiene un costo adicional. En primer lugar, hay menos datos disponibles para evaluar los nodos, lo que en particular implica que es más probable que se descarten los modelos con mayor varianza, aunque podrían ser opciones preferibles. En segundo lugar, la fase de selección consume una parte del tiempo asignado para búsqueda. Por este motivo, la búsqueda en la fase 1 va acompañada de un temporizador que estima el tiempo requerido por la fase 2, extrapolando los tiempos necesarios para evaluar los modelos durante la búsqueda. Cuando el tiempo esperado para la fase 2 se acerca al tiempo total restante, ML-Plan cambia automáticamente a la fase 2.

2.4.2. Sistemas de NAS

Auto-Keras

Auto-Keras [15] es un sistema AutoML para búsqueda de arquitecturas en redes neuronales que combina morfismos de red con optimización bayesiana. Se define un kernel de red neuronal basado en distancia de edición que mide cuántas operaciones se necesitan para cambiar una red neuronal a otra. Auto-Keras introduce una estrategia de búsqueda capaz de equilibrar la exploración y la explotación, diseñada especialmente para el espacio de búsqueda de estructura de árbol. Además, se define un morfismo de red a nivel de grafo para abordar los cambios en las arquitecturas neuronales basadas en el morfismo de red a nivel de capa.

Auto-Keras es un sistema AutoML de código abierto, que se puede descargar e instalar localmente. El sistema está diseñado con una interfaz concisa para que lo utilicen personas no especializadas en programación de computadoras y ciencia de datos. Para acelerar la búsqueda, la carga de trabajo en la CPU y la GPU se puede ejecutar en paralelo. Para abordar el problema de la memoria de la GPU diferente, que limita el tamaño de las arquitecturas neuronales, se diseña una estrategia de adaptación de la memoria para la implementación.

La idea clave del método propuesto en Auto-Keras es explorar el espacio de búsqueda mediante la transformación de las arquitecturas neuronales guiadas por un algoritmo de optimización bayesiana. La optimización bayesiana

tradicional consta de un ciclo de tres pasos: actualización, generación y observación [32]. En el contexto específico de NAS, el algoritmo de optimización bayesiano propuesto en Auto-Keras realiza los siguientes pasos:

- Actualización: entrena el modelo de proceso gaussiano subyacente con las arquitecturas existentes y sus rendimientos respectivos.
- Generación: genera la siguiente arquitectura para observar optimizando una función de adquisición definida específicamente para el espacio de búsqueda planteado.
- Observación: calcula el rendimiento real entrenando la arquitectura neuronal generada.

Hay tres desafíos principales en el diseño de un método para transformar las arquitecturas neuronales con optimización bayesiana.

El primer desafío es que el espacio NAS no es un espacio euclidiano, lo que no satisface el supuesto del proceso tradicional de optimización bayesiana. Vectorizar directamente la arquitectura neuronal no es práctico debido al número incierto de capas y parámetros que puede contener. Dado que el proceso gaussiano es un método de kernel, en lugar de vectorizar una arquitectura neuronal, en Auto-Keras se diseña una función de kernel de red neuronal. La intuición detrás de la función del kernel es la distancia de edición para transformar una arquitectura neuronal en otra. Más ediciones necesarias de una arquitectura a otra significa que hay una mayor distancia entre ellas, por lo que son menos similares.

El segundo desafío de utilizar la optimización bayesiana para guiar el morfismo de la red es la optimización de la función de adquisición. Las funciones de adquisición tradicionales se definen en el espacio euclidiano. Los métodos de optimización no son aplicables a la búsqueda estructurada en árbol a través del morfismo de red. El defecto más común del morfismo de red es que solo aumenta el tamaño de la arquitectura en lugar de reducirla. El uso de morfismo de red para NAS puede terminar con una arquitectura muy grande sin una exploración suficiente de las arquitecturas más pequeñas.

Para optimizar la función de adquisición de Auto-Keras, se diseña un método de búsqueda en árboles basado en A* con límite de confianza superior (UCB, *Upper Confidence Bound*) [105]. Este método mantiene una

cola de prioridad de nodos y sigue expandiendo el mejor nodo de la cola hasta encontrar un óptimo. Dado que A* siempre explota el mejor nodo, se introduce una estrategia de recocido simulado para equilibrar la exploración y la explotación al no seleccionar la mejor arquitectura estimada con una probabilidad.

El tercer desafío es mantener la consistencia de la forma del tensor de salida intermedio al transformar las arquitecturas. Desde el punto de vista de una red neuronal como grafo, cualquier cambio de una sola capa podría tener un efecto en toda la red. De lo contrario, se rompería la consistencia de la forma del tensor de entrada y salida. Auto-Keras introduce un morfismo a nivel de grafo para encontrar y transformar las capas influenciadas por una operación a nivel de capa en toda la red.

La arquitectura de software del sistema de Auto-Keras está diseñada para hacer un uso completo de los recursos computacionales tanto de la CPU como de la GPU. La herramienta propone una API a la que los usuarios llaman directamente. Esta API se encarga de llamar a los módulos de nivel medio correspondientes para completar determinadas funcionalidades. Un módulo de búsqueda de arquitectura neuronal contiene el Optimizador Bayesiano y el Proceso Gaussiano, algoritmos de búsqueda que se ejecutan en la CPU. Contiene otro módulo responsable del cálculo en GPU, que entrena redes neuronales dadas con los datos de entrenamiento en un proceso separado para el paralelismo. Adicionalmente, otro módulo procesa los grafos computacionales de las redes neuronales, controlado por el buscador para las operaciones de morfismo de la red. La arquitectura neuronal que se evalúa en cada iteración se coloca en la RAM para un acceso más rápido. Dado que el tamaño de las redes neuronales es grande y no se puede almacenar todo en la memoria, se guardan todos los modelos entrenados en los dispositivos de almacenamiento.

Un flujo de trabajo típico para el sistema Auto-Keras es el siguiente. El usuario inicia una búsqueda de la mejor arquitectura neuronal para el conjunto de datos. La API recibe la llamada, preprocesa el conjunto de datos y lo pasa al buscador para iniciar la búsqueda. El Optimizador Bayesiano en el Buscador genera una nueva arquitectura usando la CPU, y llama al módulo de grafos para construir la arquitectura neuronal generada en una red neuronal real en la RAM. La nueva arquitectura neuronal se copia en

la GPU para ser entrenada con el conjunto de datos. El modelo entrenado se guarda en un dispositivo de almacenamiento de modelos. Finalmente, el rendimiento del modelo es retroalimentado al buscador para actualizar el proceso gaussiano.

El diseño de la API de Auto-Keras sigue el diseño clásico de la API de *scikit-learn* [74], que es conciso y configurable. El entrenamiento de una red neuronal requiere tan solo tres líneas de código que llaman al constructor, la función de ajuste y predicción, respectivamente. Para satisfacer las necesidades de diferentes usuarios, Auto-Keras propone dos niveles de API. En el primer nivel, que se denomina nivel de tarea, los usuarios solo necesitan conocer su tarea, por ejemplo, clasificación de imágenes, o regresión de texto, para usar la API. El segundo nivel se denomina nivel de búsqueda, y está diseñado para usuarios avanzados. El usuario puede buscar un tipo específico de arquitecturas de redes neuronales, por ejemplo, perceptrón multicapa, o red neuronal convolucional. Para usar esta API, el usuario necesita preprocesar el conjunto de datos y saber qué tipo de red neuronal, por ejemplo, CNN o MLP, es la mejor para su tarea.

Auto-Keras adiciona varias adaptaciones para mejorar la experiencia del usuario. Primero, el usuario puede restaurar y continuar una búsqueda anterior que haya ser detenida accidentalmente. Desde la perspectiva de los usuarios, la principal diferencia de usar Auto-Keras en comparación con los sistemas AutoML que apuntan a modelos superficiales es el consumo de tiempo mucho más prolongado, ya que varias redes neuronales profundas se entrenan durante la búsqueda de arquitectura neuronal. Es posible que ocurra algún accidente que detenga el proceso antes de que finalice la búsqueda. Por lo tanto, la búsqueda genera todas las arquitecturas de redes neuronales evaluadas con sus parámetros entrenados en un directorio específico en el disco. Siempre que se proporcione la ruta al directorio, se puede restaurar la búsqueda anterior. En segundo lugar, el usuario puede exportar los resultados de la búsqueda, que son arquitecturas neuronales, como modelos de Keras guardados para otros usos. En tercer lugar, los usuarios avanzados pueden especificar los de hiperparámetros del proceso de búsqueda y el proceso de optimización de la red neuronal.

Para hacer un uso completo de los recursos de computación local limitados, el proceso de AutoML puede ejecutarse en paralelo en la GPU y la

CPU al mismo tiempo. Para mejorar la eficiencia, el entrenamiento de una red neuronal concreta se ejecuta en paralelo con la generación de nuevas arquitecturas en procesos separados. Se mantiene una cola de entrenamiento como búfer para sincronizar ambos procesos. De esta forma, el tiempo de inactividad de la GPU y la CPU se reduce drásticamente para mejorar la eficiencia del proceso de búsqueda.

En Auto-Keras el tamaño de las redes neuronales debe limitarse según la memoria de la GPU. De lo contrario, el sistema fallaría por falta de memoria GPU. El sistema implementa una función de estimación de memoria basada en la representación en forma de grafo de la red neuronal. Se utiliza un valor entero para marcar el límite superior del tamaño de la arquitectura neuronal estimado y se descarta cualquier grafo computacional nuevo cuyo tamaño estimado exceda el límite superior. Sin embargo, el sistema aún puede fallar porque la administración de la memoria de la GPU es complicada y no se puede estimar con precisión. Entonces, cada vez que se agota la memoria de la GPU, el límite superior se reduce para limitar aún más el tamaño de las redes neuronales generadas.

Auto-Pytorch

Auto-PyTorch [66], es un sistema de AutoML que utiliza optimización de fidelidad múltiple [106] para optimizar conjuntamente los parámetros arquitectónicos de redes neuronales y los hiperparámetros de entrenamiento. Esta estrategia de búsqueda es un tipo especial de optimización bayesiana que entrena los flujos en conjuntos de datos cada vez mayores. Como sugiere el nombre, Auto-PyTorch se basa en el marco PyTorch [107] como marco de aprendizaje profundo. Auto-PyTorch implementa y ajusta automáticamente el flujo de aprendizaje profundo completo, incluido el preprocesamiento de datos, la arquitectura neuronal, las técnicas de entrenamiento de la red y los métodos de regularización. Además, ofrece una inicialización de la optimización mediante el muestreo de configuraciones de conjunto de arquitecturas predefinidas, así como una selección de *ensembles* [68] automatizada.

El espacio de configuración proporcionado por Auto-PyTorch para datos tabulares contiene una gran cantidad de hiperparámetros, que van desde opciones de preprocesamiento (por ejemplo, codificación, imputación) hiperparámetros arquitectónicos (por ejemplo, tipo de red, número de capas)

hasta hiperparámetros de entrenamiento (por ejemplo, tasa de aprendizaje, disminución de peso). Específicamente, el espacio de configuración de Auto-PyTorch consta de todos estos hiperparámetros y está estructurado por una jerarquía condicional, de modo que los hiperparámetros de nivel superior pueden activar o desactivar hiperparámetros de subnivel en función de su configuración.

Para la arquitectura, AutoPytorch implementa MLP con ReLU [108] y *dropout* [109]. Las redes MLP con forma ofrecen un espacio de diseño eficiente al evitar los hiperparámetros por capas. Se utiliza una variante en forma de embudo que solo requiere un número predefinido de capas y un número máximo de unidades como entrada. La primera capa se inicializa con la cantidad máxima de neuronas y cada capa subsiguiente contiene la cantidad necesaria de neuronas para ajustar las dimensiones de los vectores y asignar solo una cantidad máxima de neuronas. El espacio de búsqueda incluye otro algoritmo de entrenamiento, Adam [110], y otras estrategias de regularización. Contiene opciones de preprocesamiento con SVD truncado para tratar con conjuntos de datos dispersos y redes residuales en forma de embudo (ResNets) [111], lo que permite entrenar redes más profundas. Al igual que los MLP, las ResNets permiten una hiperparametrización eficiente mediante la repetición de grupos con un número predefinido de bloques ResNet. La dimensión de salida de cada grupo se determina de manera idéntica a un MLP conformado con el número equivalente de capas.

Para garantizar la generalización del modelo, se requiere una buena estrategia de selección de modelos en cualquier marco de AutoML. Con ese fin, Auto-PyTorch implementa una variedad de opciones. AutoPytorsh permite la entrada de divisiones definidas por el usuario o la división automática, o la validación cruzada con cualquiera de los iteradores de validación cruzada de scikit-learn. A continuación, el rendimiento del modelo se evalúa mediante una métrica predefinida o especificada por el usuario en el conjunto de validación y se utiliza para seleccionar configuraciones de buen rendimiento. Auto-PyTorch también ofrece el uso de la detención temprana a través del rendimiento en el conjunto de validación para mejorar aún más la generalización.

La naturaleza iterativa de Auto-PyTorch implica que se entrenan y evalúan muchos modelos. Esto, permite impulsar aún más el rendimiento predictivo al combinar los modelos evaluados. Auto-PyTorch utiliza un

enfoque para *ensembles* inspirado en Auto-Sklearn [13], implementando un método de selección de *ensembles* post-hoc automatizado que opera de manera eficiente sobre las predicciones del modelo almacenadas durante la optimización. A partir de un conjunto vacío, la selección de *ensembles* agrega iterativamente el modelo que ofrece la mayor mejora de rendimiento hasta alcanzar un tamaño de conjunto predefinido. Permitir múltiples adiciones del mismo modelo da como resultado un conjunto ponderado. Dado que se ha demostrado que la regularización mejora el rendimiento del *ensemble*, AutoPytorch considera solo los k mejores modelos en nuestra selección de conjuntos. Una ventaja de del *ensemble* post-hoc es que también puede incluir fácilmente otros modelos además de redes neuronales profundas.

La estrategia de búsqueda de Auto-Pytorch, BOHB (*Bayesian Optimizacion + Hyperband*) [112] está orientado a un buen rendimiento en cualquier momento en grandes espacios de búsqueda. Sin embargo, comienza desde cero para cada nueva tarea. Por lo tanto, la utilización de un inicio con arquitecturas predefinidas en la optimización permite mejorar aún más el rendimiento inicial.

Auto-Sklearn utilizó metacaracterísticas de tareas para determinar configuraciones prometedoras para el inicio predefinido. Auto-PyTorch simplemente inicia la primera iteración de BOHB con un conjunto de configuraciones complementarias que cubren bien un conjunto de conjuntos de datos de meta-entrenamiento; luego pasa al muestreo convencional de BOHB.

NASH

La búsqueda de arquitectura neuronal por *hill climbing* (NASH) [58], es un enfoque iterativo simple que, en cada paso, aplica un conjunto de morfismos de red alternativos a la red actual, entrena las redes secundarias resultantes con ejecuciones cortas de optimización de recocido de coseno [113], y pasa a la red derivada más prometedora. NASH encuentra y entrena arquitecturas competitivas a un costo computacional del mismo orden de magnitud que entrenar una sola red.

El algoritmo de búsqueda propuesto es una estrategia de *hill climbing* (escalado de colina). Comienza con una pequeña red (posiblemente) previamente entrenada. Luego, aplica morfismos de red a esta red inicial para generar otros más grandes que puedan funcionar mejor cuando se entrenan más.

Estas nuevas redes "secundarias" pueden verse como vecinas de la red "principalïnicial en el espacio de las arquitecturas de red. Debido al morfismo de la red, las redes secundarias comienzan con el mismo rendimiento que su origen. En esencia, los morfismos de red pueden verse como una forma de inicializar las redes derivadas para que funcionen bien, evitando el costoso paso de entrenarlos desde cero y, por lo tanto, reduciendo el costo de su evaluación. Las diversas redes secundarias se pueden entrenar posteriormente durante un breve período de tiempo para explotar la capacidad adicional obtenida por el morfismo de la red, y la búsqueda puede pasar a la mejor red secundaria resultante. NASH puede ejecutar este paso varias veces hasta que se sature el rendimiento en un conjunto de validación. Este proceso utiliza una aproximación golosa que puede, en principio, quedarse atascado en una región de bajo rendimiento, de la que nunca podrá escapar, pero no presentan evidencia de esto en los experimentos.

- Hacer la red más profunda, es decir, agregar un bloque Çonv-BatchNorm-Relu". La posición donde agregar el bloque, así como el tamaño del kernel ($in\{3, 5\}$), se muestrean uniformemente. El número de canales se elige para que sea igual al número de canales de la convolución anterior más cercana.
- Ampliar la red, es decir, aumentar el número de canales utilizando el morfismo de red tipo II. La capa de conv que se ampliará, así como el factor de ampliación ($in\{2, 4\}$) se muestrean uniformemente al azar.
- Agregar una conexión de salto de la capa i a la capa j (ya sea por concatenación o adición, muestreada uniformemente) mediante el uso de morfismo de red. Las capas i y j también se muestrean uniformemente. El mejor modelo actual también se considera un derivado, es decir, el algoritmo no está obligado a seleccionar un nuevo modelo, sino que también puede mantener el anterior si nadie lo mejora.

Es importante para NASH que las redes secundarias solo necesiten ser entrenadas durante unas pocas épocas. Por lo tanto, se requiere un algoritmo de optimización con un buen rendimiento en cualquier momento. Para ello se elige la estrategia de recocido de coseno [113] mientras que la tasa de aprendizaje se reinicia implícitamente: el entrenamiento siempre comienza con una tasa de aprendizaje λ_{start} que converge a λ_{end} después de un número

determinado de épocas. Se utiliza el mismo planificador de velocidad de aprendizaje en el entrenamiento final (aparte de un número diferente de épocas). Si bien presentan el método como un método simple de escalada de colinas, se observa que también se puede interpretar como un algoritmo evolutivo muy simple con un tamaño de población n, sin cruzamiento, utilizando morfismos de red como mutaciones y un mecanismo de selección que solo considera al miembro de la población con mejor desempeño como el padre de la próxima generación.

EAS

EAS, *Efficient Architecture Search* [56], es un método de AutoML para NAS basado en aprendizaje por refuerzo [55, 114]. En esta propuesta el metacontrolador explora el espacio de la arquitectura mediante operaciones de transformación de red como ensanchar una determinada capa (más unidades o filtros), insertar una capa, agregar conexiones de salto, etc., dada una red existente entrenada en la misma tarea. Para reutilizar los pesos de redes entrenadas anteriormente, se consideran una clase de transformaciones que preservan la función aprendida por la red, y que permiten inicializar la nueva red para representar la misma función, pero usando una parametrización diferente para mejorar el rendimiento.

Dada una representación de baja dimensión de la arquitectura de una red neuronal, EAS propone dos agentes de aprendizaje por refuerzo que toman las decisiones necesarias para aplicar cierto tipo de acciones de transformación de la red. En concreto, se presentan dos agentes basados a su vez en redes neuronales: Net2Wider y el actor Net2Deeper, que corresponden a Net2WiderNet y Net2DeeperNet respectivamente [115].

El agente Net2Wider permite reemplazar una capa con una capa más ancha, lo que significa más unidades para capas completamente conectadas, o más filtros para capas convolucionales, a la vez que preserva la funcionalidad de la capa ya entrenada. Para ser flexible y eficiente, el agente Net2Wider determina simultáneamente si cada capa debe ampliarse. Específicamente, para cada capa, esta decisión la lleva a cabo un clasificador sigmoide compartido dado el estado oculto de la capa aprendido por la red del codificador bidireccional. Si el agente Net2Wider decide ampliar una capa, el número de filtros o unidades de la capa aumenta al siguiente nivel discreto, p. ej. de 32

a 64.

El agente Net2Deeper permite insertar una nueva capa que se inicializa agregando un mapeo de identidad entre dos capas para preservar la funcionalidad. Para una nueva capa convolucional, el kernel está configurado para ser filtros de identidad, mientras que para una nueva capa completamente conectada, la matriz de peso está configurada para ser una matriz de identidad. Por lo tanto, la nueva capa se establece con el mismo número de filtros o unidades que la capa inferior al principio, y podría ensancharse aún más cuando se realiza la operación Net2Wider en ella. Para preservar completamente la funcionalidad, la operación Net2Deeper tiene una restricción en la función de activación ϕ, es decir, ϕ debe satisfacer $\phi(I \cdot \phi(v)) = \phi(v)$ para todos los vectores v. Esta propiedad es válida para la activación lineal rectificada (ReLU) pero falla para la activación sigmoidea y tanh.

La estructura del agente Net2Deeper es una red recurrente cuyo estado oculto se inicializa con el estado oculto final de la red del codificador. Específicamente, una arquitectura CNN se divide en varios bloques de acuerdo con las capas de agrupación y el actor Net2Deeper determina secuencialmente en qué bloque insertar la nueva capa, un índice específico dentro del bloque y los parámetros de la nueva capa. Para una nueva capa convolucional, el agente necesita determinar el tamaño del filtro, mientras que para una nueva capa completamente conectada, no se necesita la predicción de parámetros. En las arquitecturas CNN, cualquier capa completamente conectada debe estar en la parte superior de todas las capas convolucionales y de *pooling* [116]. Para evitar que se generen arquitecturas irrazonables, si el actor de Net2Deeper decide insertar una nueva capa después de una capa completamente conectada o la capa final de agrupación promedio global, la nueva capa está restringida para ser una capa completamente conectada; de lo contrario, debe ser convolucional.

SNAS

SNAS, *Stochastic Neural Architecture Search* [31] es un método de AutoML para NAS que entrena los parámetros de operación neuronal y los parámetros de distribución de la arquitectura en la misma ronda de retropropagación, mientras se mantiene la integridad y diferenciabilidad de la arquitectura. Una de las motivaciones clave de SNAS es reemplazar el

mecanismo de retroalimentación activado por recompensas constantes en NAS basado en aprendizaje reforzado con retroalimentación de gradiente, que es más eficiente. En SNAS, el enfoque NAS se formula a partir de un modelo estocástico para evitar aplicar aprendizaje por refuerzo [55, 114]. Para combinar el muestreo de arquitectura con un grafo computacional de pérdida diferenciable arbitraria, el espacio de búsqueda se representa con un conjunto de variables aleatorias *one-hot* de una distribución conjunta completamente factorizable. El muestreo de este espacio de búsqueda se hace diferenciable al relajar la distribución de la arquitectura con distribución concreta. Desde una visión global, SNAS optimiza el mismo objetivo que el enfoque NAS basado en aprendizaje reforzado, excepto que la pérdida de entrenamiento se utiliza como recompensa.

SNAS busca operaciones y topología de esta celda al mismo tiempo [28]. La búsqueda de la estructura de una celda que luego se apila como bloques de construcción para una arquitectura profunda es una solución ad-hoc para compensar la eficiencia de búsqueda y la optimización de resultados. El espacio de búsqueda en SNAS, que es una celda, se representa mediante un grafo acíclico dirigido (DAG), que se denomina grafo principal. Los nodos x_i en este DAG representan la estructura latente que se produce en cada capa de la red neuronal entrenada. En las redes convolucionales, son mapas de características. Las aristas (i, j) representan los flujos de información y las posibles operaciones $O_{i,j}$ que se seleccionarán entre dos nodos x_i y x_j. Para que se incluya la operación de omisión, los nodos deben ordenarse, de forma que las aristas solo apuntan desde los nodos indexados inferiores a los superiores.

En lugar de utilizar dos distribuciones, en SNAS esto se introduce una operación cero, como en DARTS [30]. De la misma forma, cada celda está diseñada para tener dos entradas de la salida de las celdas anteriores. La salida de una celda es la concatenación de nodos intermedios.

En términos de cómo parametrizar y factorizar la función $p(Z)$ de distribución de las arquitecturas, SNAS se basa en la observación de que NAS es una tarea con recompensas completamente retrasadas en un entorno determinista [115]. Es decir, la señal de retroalimentación solo está lista después de que finaliza todo el episodio y todas las distribuciones de transición de estado son funciones delta. Por lo tanto, una suposición del proceso de decisión de

Markov no es necesaria. En SNAS, se asume que $p(Z)$ es completamente factorizable, cuyos factores se parametrizan con α y se aprenden junto con los parámetros de operación θ.

Todo el proceso de obtención de una estimación de este objetivo se realiza con un algoritmo de Monte Carlo. Una interpretación intuitiva es optimizar el rendimiento esperado de las arquitecturas muestreadas con $p(Z)$. Esto diferencia a SNAS de los NAS basados en atención como DARTS, que evita el proceso de muestreo tomando el valor esperado en cada arista de todas las operaciones.

DARTS

Propone un método para la búsqueda de arquitectura eficiente llamado DARTS (*Differentiable Architecture Search*) [30]. En lugar de buscar en un conjunto discreto de arquitecturas candidatas, se relaja el espacio de búsqueda para que sea continuo, de modo que la arquitectura pueda optimizarse con respecto al rendimiento de su conjunto de validación por descenso de gradiente. La eficiencia de los datos de la optimización basada en gradientes, a diferencia de la búsqueda de caja negra, permite a DARTS lograr un rendimiento competitivo con el estado del arte utilizando recursos de computación de órdenes de magnitud menor. Este método es lo suficientemente genérico como para manejar arquitecturas convolucionales [116] y recurrentes [117].

DARTS optimiza una celda neuronal como el bloque de construcción de la arquitectura final. La celda aprendida podría combinarse para formar una red convolucional o conectarse recursivamente para formar una red recurrente. Una celda es un grafo acíclico dirigido que consta de una secuencia ordenada de N nodos. Cada nodo $x(i)$ es una representación latente (por ejemplo, un mapa de características en redes convolucionales [116]) y cada arista dirigida (i, j) está asociada con alguna operación $o(i, j)$ que transforma a $x(i)$. La celda tiene dos nodos de entrada y un solo nodo de salida. Para las celdas convolucionales, los nodos de entrada se definen como las salidas de celda de las dos capas anteriores. Para las celdas recurrentes, se definen como la entrada en el paso actual y el estado del paso anterior. La salida de la celda se obtiene aplicando una operación de reducción (por ejemplo, concatenación) a todos los nodos intermedios.

Sea O un conjunto de operaciones candidatas (por ejemplo, convolución,

max pooling, cero) donde cada operación representa alguna función $o(\cdot)$ que se aplicará a $x(i)$. Para hacer que el espacio de búsqueda sea continuo, se relaja la elección categórica de una operación en particular a un *softmax* sobre todas las operaciones posibles. La operación de mezcla de pesos para un par de nodos (i, j) es parametrizada por un vector $\alpha(i, j)$ de dimensión $|O|$. Donde α corresponde a la codificación de la arquitectura. Después de la relajación continua, el objetivo es aprender conjuntamente la arquitectura α y los pesos w dentro de todas las operaciones mixtas (por ejemplo, pesos de los filtros de convolución). De forma análoga a la búsqueda de arquitectura utilizando aprendizaje por refuerzo o evolución, donde el rendimiento en un conjunto de validación se trata como recompensa o aptitud, DARTS tiene como objetivo optimizar la pérdida de validación, pero utilizando el descenso de gradiente [118]. Para ello se optimizan L_{train} y L_{val}, las pérdidas de entrenamiento y de validación, respectivamente. Ambas pérdidas están determinadas no solo por la arquitectura α, sino también por los pesos w en la red. El objetivo de la búsqueda de arquitectura es entonces encontrar α que minimice la pérdida de validación.

La evaluación exacta del gradiente de la arquitectura puede resultar prohibitiva debido a la costosa optimización interna. DARTS aproxima este gradiente adaptando w usando un solo paso de entrenamiento, sin resolver la optimización interna por completo entrenando hasta la convergencia. Actualmente no se conocen garantías de convergencia de este algoritmo de optimización, pero en la práctica es capaz de alcanzar un punto fijo con una elección adecuada del paso ϵ.

2.4.3. Sistemas Comerciales

Google Cloud AutoML / Vertex AI

Google Cloud AutoML[4] permite a los desarrolladores con experiencia limitada en aprendizaje automático entrenar modelos de alta calidad específicos para sus necesidades comerciales. Está orientado a los desarrolladores con experiencia en aprendizaje automático a crear fácilmente modelos de AA que funcionen con cualquier tipo de datos, de cualquier tamaño. Los servicios de aprendizaje automático que brinda, es decir, las API, incluyen Vision,

[4] https://cloud.google.com/vertex-ai/

Speech, NLP, Translation y Dialogflow, pueden construirse sobre modelos previamente entrenados para brindar una escala y velocidad mayores para las aplicaciones comerciales.

Este sistema ayuda a las empresas con experiencia limitada en aprendizaje automático a comenzar a crear modelos personalizados de alta calidad mediante el uso de técnicas avanzadas como *learning2learn* [119] y *transfer learning* [120] desarrolladas por Google. Cloud AutoML se propone que los expertos en inteligencia artificial sean más productivos, avanzando en nuevos campos en la inteligencia artificial y ayudando a los ingenieros menos capacitados a construir potentes sistemas de inteligencia artificial.

La primera versión de Cloud AutoML es Cloud AutoML Vision, un servicio que agiliza y facilita la creación de modelos de AA personalizados para el reconocimiento de imágenes. Su interfaz de arrastrar y soltar le permite cargar fácilmente imágenes, entrenar y administrar modelos, y luego implementar esos modelos entrenados directamente en Google Cloud. Los primeros resultados con Cloud AutoML Vision para clasificar conjuntos de datos públicos populares como ImageNet y CIFAR han mostrado resultados más precisos con menos clasificaciones erróneas que las API de aprendizaje automático genéricas.

Cloud AutoML Vision se basa en los principales enfoques de reconocimiento de imágenes de Google, que incluyen el aprendizaje por transferencia y las tecnologías de búsqueda de arquitectura neuronal [28]. Garantiza un tiempo de respuesta más rápido para modelos listos para producción, ya que permite puede crear un modelo simple en minutos para probar su aplicación, o crear un modelo completo listo para producción en tan solo un día. Este sistema proporciona una interfaz gráfica de usuario simple que permite especificar datos y luego convierte esos datos en un modelo de alta calidad personalizado para las necesidades específicas del usuario.

Recientemente, el sistema Cloud AutoML de Google se ha integrado con otros servicios en la plataforma Vertex AI. Con este paso, Google amplía sus servicios de AutoML en la nube hacia los dominios de lenguaje natural, datos tabulares y series de tiempo. Incorpora componentes para automatizar la selección y el preprocesamiento de características. Vertex AI proporciona modelos preentrenados y herramientas de búsqueda de arquitecturas de redes neuronales para tareas de clasificación, regresión, segmentación (e.j.,

detección de objetos en imágenes) y predicción estructurada (e.j., detección de entidades en lenguaje natural).

Este sistema se basa en optimización bayesiana [32], aunque brinda también la posibilidad de emplear búsqueda aleatoria. Al terminar de optimizar un flujo, es posible inspeccionar los modelos, y exportarlos en imágenes de Docker[5] para ser desplegados en otra plataforma.

Microsoft Azure Machine Learning

El sistema de Azure ML[6] se propone acelerar el ciclo de vida del aprendizaje automático, dotando a los científicos y desarrolladores de datos con una gama de experiencias productivas para crear, capacitar e implementar modelos de aprendizaje automático y fomentar la colaboración en equipo. Además de acelerar el tiempo de comercialización realizando operaciones de aprendizaje automático o DevOps para aprendizaje automático. Presenta dos interfaces: Jupyter Notebooks integrados con IntelliSense, y un diseñador gráfico. Contiene selección de algoritmos y funciones de optimización de hiperparámetros. Este sistema se distingue por proponer mecanismos para preservar la privacidad de los datos durante todo el ciclo de vida del aprendizaje automático con técnicas de privacidad diferenciales [121] y utilizar la informática confidencial para proteger los activos del aprendizaje automático.

El sistema de AutoML de Azure brinda utilidades de MLOps[7] para optimizar el ciclo de vida del aprendizaje automático, desde la creación de modelos hasta la implementación y la administración. Se integra con Azure DevOps y GitHub Actions[8] para programar, administrar y automatizar los flujos de aprendizaje automático. Proporciona además un análisis avanzado de deriva de datos para mejorar el rendimiento del modelo a lo largo del tiempo.

El sistema presenta soporte integrado para herramientas y marcos de código abierto para el entrenamiento y la inferencia de modelos de aprendizaje automático. Usa bibliotecas como PyTorch [107], TensorFlow [122] o

[5] https://docker.com/
[6] https://azure.microsoft.com/en-us/get-started/services/machine-learning/
[7] https://ml-ops.org/
[8] https://github.com

scikit-learn [74], así como el formato ONNX[9] abierto e interoperable. Permite seleccionar las herramientas de desarrollo que mejor se adapten a las necesidades del usuario, incluidos IDE populares, Visual Studio Code, Jupyter Notebooks y CLI, o lenguajes como Python y R. Utiliza ONNX Runtime para optimizar y acelerar la inferencia en la nube y dispositivos de IoT [67].

Azure ML es aplicable en múltiples dominios, incluyendo texto en lenguaje natural, datos tabulares, y series de tiempo. Permite entrenar desde modelos clásicos de aprendizaje automático hasta redes neuronales, y realiza automáticamente un proceso de construcción de *ensembles* [68]. Es aplicable en tareas de clasificación, regresión y agrupamiento, e incluye también herramientas para la imputación de datos y el preprocesamiento y selección de características. Este sistema se basa en optimización bayesiana [32] por defecto, aunque brinda la opción de aplicar búsqueda aleatoria. Permite inspeccionar y exportar los modelos obtenidos a través del formato ONNX.

Microsoft AI builder

AI Builder[10] es una componente de la plataforma Microsoft Power Platform que proporciona modelos de inteligencia artificial diseñados para optimizar procesos comerciales. AI Builder permite a usuarios empresariales utilizar técnicas de inteligencia artificial para automatizar procesos y obtener conocimientos de los datos en Power Apps y Power Automate. Este sistema brinda una interfaz gráfica sencilla, por lo que no necesita habilidades de codificación o ciencia de datos para interactuar con el sistema. Con AI Builder se pueden crear modelos personalizados que se adapten a las necesidades de los usuarios o elegir un modelo prediseñado que esté listo para usar en escenarios comerciales comunes.

También permite seleccionar los datos específicos de una empresa entre las opciones disponibles. Según el tipo de modelo, se puede modificar los modelos personalizados para optimizar el rendimiento. Describen el entrenamiento como un proceso automático que genera información como el resultado de una predicción o la lista y la cantidad de objetos detectados en una imagen. Los resultados del modelo de IA pueden combinarse en Microsoft Power Platform para crear soluciones que satisfagan las necesidades comerciales del

[9] https://onnx.ai/about.html
[10] https://docs.microsoft.com/en-us/ai-builder/

usuario.

AI Builder es aplicable en los dominios de imágenes, texto en lenguaje natural y series de tiempo. Permite realizar tareas de clasificación y regresión, e incluye algunas componentes para el preprocesamiento de características. También es aplicable en tareas de predicción estructurada y segmentación. A diferencia de Azure ML, los modelos entrenados no son fácilmente exportables ni inspeccionables fuera de la plataforma, por lo que está más dirigida hacia usuarios de Microsoft Power Platform que deseen mantenerse dentro de este ecosistema.

Hugging Face AutoNLP

Hugging Face AutoNLP[11] es un sistema de AutoML que se concentra en lenguaje natural. Permite entrenar e implementar modelos de PLN del estado del arte integrados con el ecosistema de Hugging Face[12]. El sistema permite realizar tareas como clasificación binaria, clasificación de clases múltiples, regresión, reconocimiento de entidades, resumen y reconocimiento automático de voz. Tiene disponibles miles de modelos en varios idiomas, incluyendo: inglés, alemán, francés, español, finlandés, sueco, hindi, holandés, entre otros.

En AutoNLP, los datos de entrenamiento de los usuarios permanecen en el servidor de HuggingFace durante el proceso de AutoML y son privados para cada cuenta. Todas las transferencias de datos están protegidas por SSL SSL. Es posible utilizar datos en formatos CSV, TSV o JSON, alojados en cualquier lugar. Una vez terminado el entrenamiento, los datos son eliminados de los servidores.

Entre las principales características que lista AutoNLP en su documentación se encuentran:

- Selección automática de los mejores modelos dados sus datos
- Ajuste automático
- Optimización automática de hiperparámetros
- Comparación de modelos después del entrenamiento

[11] https://huggingface.co/autonlp
[12] https://huggingface.co

- Despliegue inmediato después del entrenamiento
- CLI y API de Python disponibles

En concreto, AutoNLP admite las siguientes tareas específicamente el dominio del procesamiento del lenguaje natural:

Clasificación binaria: una oración tiene un objetivo asociado y hay dos objetivos únicos en el conjunto de datos.

Clasificación de clases múltiples: una oración tiene un objetivo asociado y hay más de dos objetivos únicos en el conjunto de datos.

Extracción de entidades: también conocida como reconocimiento de entidad con nombre o clasificación de token. Esta tarea consta de una oración y en la oración, cada token está asociado a una etiqueta particular.

Resumen: una secuencia de tareas en secuencia en la que la secuencia más grande se resume en una secuencia más pequeña.

Reconocimiento de voz: permite entrenar un modelo propio de reconocimiento de voz automático.

Regresión de una sola columna: una oración tiene asociado un valor numérico.

Respuesta a preguntas abstractiva: permite entrenar modelos personalizados de respuesta a preguntas en un conjunto de datos propio.

AutoNLP permite realizar ajuste fino de modelos preentrenados para una multitud de tareas e idiomas de forma automática. Los modelos se entrenan en la nube, y la inferencia se realiza a través de un servicio web o una API programática. En la documentación existente no es posible identificar si el sistema permite exportar los modelos fuera de la plataforma de Hugging Face para ser usados en otros entornos.

2.5. Conclusiones

La democratización de la Inteligencia Artificial es una de las preocupaciones fundamentales, tanto de la comunidad científica como de los expertos

de la industria. El campo del AutoML se presenta como una alternativa prometedora para disminuir substancialmente el esfuerzo que conlleva la aplicación de técnicas de inteligencia artificial, y específicamente de aprendizaje automático, a problemas concretos. Aunque existen varias herramientas de AutoML que han sido exitosas resolviendo problemas específicos de inteligencia artificial, estas herramientas son aún demasiado rígidas para ser utilizadas en problemas prácticos que requieren la combinación de algoritmos y tecnologías de diferente naturaleza.

La comunidad de AutoML se ha centrado en resolver varias partes de un flujo de trabajo estándar de aprendizaje automático. Se ha prestado especial atención a la selección de modelos y el ajuste de hiperparámetros y, en menor medida, al preprocesamiento y limpieza de datos [9]. Por lo tanto, la mayoría de las soluciones de AutoML existentes se centran en una subtarea específica de este flujo de trabajo (p. Ej., búsqueda de modelo completo [16, 18] o selección combinada de algoritmos y optimización de hiperparámetros [13, 23]) o en flujos de trabajo de un extremo a otro basados en un subdominio específico de modelos (por ejemplo, búsqueda de arquitectura neuronal [15, 66]).

Extender los sistemas AutoML para que se ocupen de los flujos de trabajo completos de un extremo a otro de manera transparente es un esfuerzo desafiante, en parte dado las características dispares de estas tareas y la falta de marcos unificados que abarquen todo el proceso de aprendizaje automático. Para alcanzar todo el potencial de AutoML como un paradigma sólido para resolver problemas prácticos de aprendizaje automático en escenarios del mundo real, un sistema debería poder:

- Representar todo tipo de problemas y dominios de interés (por ejemplo, aprendizaje supervisado estructurado y no estructurado, aprendizaje no supervisado, recuperación de información, procesamiento del lenguaje natural, procesamiento de señales).
- Implementar todos los pasos de un flujo estándar de aprendizaje automático (por ejemplo, recopilación y preprocesamiento de datos, ajuste del modelo, integración de recursos externos, despliegue).
- Permitir una búsqueda eficaz y eficiente sobre el espacio de todos los posibles flujos de trabajo para un problema determinado.

En un sentido general, un sistema AutoML podría ser una herramienta para ayudar en la toma de decisiones, ayudando a los expertos a encontrar mejores soluciones a problemas complejos y comprendiendo sus compensaciones y salvedades. Avanzar hacia esta visión de AutoML requiere diseñar representaciones computacionales para flujos de trabajo de aprendizaje automático que puedan unificar la amplia gama de tareas, técnicas y dominios de interés en escenarios prácticos. Esta investigación propone una representación en esta dirección, permitiendo el diseño de un sistema AutoML que puede abordar diferentes dominios combinando de forma transparente técnicas y herramientas dispares. Este enfoque permite generalizar AutoML a dominios novedosos y mejora la explicabilidad e interpretabilidad del proceso AutoML.

Capítulo 3

Paradigma de AutoML Heterogéneo

En el marco de esta Tesis se han definido, analizado y desarrollado diversas estrategias para la optimización de flujos de algoritmos de aprendizaje automático, utilizando un enfoque de AutoML. Se presenta una nueva formulación teórica para el problema de AutoML Heterógeno que abarca múltiples dominios y técnicas y se diseña una estrategia computacional para su solución. Además, se han realizado evaluaciones experimentales en diferentes escenarios comparando esta estrategia con otros enfoques de AutoML existentes, así como con enfoques diseñados por expertos. La unión de estos resultados constituye un nuevo paradigma en este campo que da respuesta al problema de AutoML Heterogéneo. Como resultado adicional se presenta una aplicación práctica de este paradigma en la implementación de un sistema de AutoML denominado AutoGOAL, que sirve de soporte a futuras investigaciones.

AutoGOAL es el resultado de una investigación y desarrollo gradual que comenzó con la definición de técnicas de optimización para flujos específicos de algoritmos de procesamiento de lenguaje natural y que evolucionó hacia una arquitectura de software extensible y escalable. El primer paso (Capítulo 5) consiste en el diseño de una técnica para la optimización de flujos lineales de algoritmos utilizando la estrategia de optimización *Population-Based Incremental Learning, PBIL*. Esta técnica es evaluada en un problema de clasificación de textos con un espacio de algoritmos compuesto por clasificadores,

reductores de dimensiones, y diversos pasos de preprocesamiento de texto. La principal restricción de esta técnica es que no permite la optimización de parámetros continuos, sino que requiere la discretización del rango de cada parámetro.

Para ampliar el espacio de soluciones se introduce una representación jerárquica (Capítulo 6) a partir del uso de gramáticas libres del contexto definidas por un experto del dominio. Esta formulación permite optimizar de forma conjunta parámetros binarios, discretos, enteros, y continuos, utilizando un modelo probabilístico del espacio de flujos de algoritmos y una variante de la estrategia de optimización *Probabilistic Grammatical Evolution* diseñada en el marco de esta tesis. Esta técnica es evaluada en un problema de extracción de entidades y relaciones a partir de lenguaje natural. Su principal desventaja es que requiere la definición e implementación de una gramática libre del contexto, por parte de un experto, que contemple todos los algoritmos posibles a escoger y los rangos de sus parámetros a optimizar. Esta representación jerárquica se formaliza en la definición de un lenguaje (Capítulo 7) que permite al experto describir diferentes espacios de flujos de algoritmos de forma declarativa. Para evaluar este proceso se formulan gramáticas para dominios diferentes y se comparan experimentalmente con herramientas de AutoML alternativas.

Como último paso se crea un algoritmo (Capítulo 8) capaz de inferir automáticamente una gramática a partir de un conjunto de algoritmos previamente seleccionados. De esta forma se libera al experto de la necesidad de diseñar una gramática para cada nuevo problema. Además, se define un protocolo que simplifica el proceso de incluir nuevos algoritmos a un sistema de AutoML basado en el paradigma definido en esta tesis. Se evalúa en problemas de múltiples dominios, incluyendo clasificación estructurada, clasificación de textos, y descubrimiento de conocimiento.

En los capítulos siguientes se presentan todos los resultados intermedios, mientras que este capítulo se concentra en el resultado práctico (el sistema AutoGOAL) que combina todas las técnicas y estrategias desarrolladas durante la investigación y da una solución computacional al problema teórico planteado. En la Sección 3.1 se describe la arquitectura de la biblioteca y se ilustra su uso por usuarios expertos y no expertos. En la Sección 3.2 se presentan los algoritmos y estrategias computacionales diseñados para

dar soporte a las herramientas que propone AutoGOAL. Finalmente, en la Sección 3.3 se presentan los detalles de implementación más relevantes y las soluciones de ingeniera de software que fueron aplicadas para llevar a la práctica los algoritmos diseñados en esta investigación.

3.1. Arquitectura General

AutoGOAL permite a los usuarios, tanto expertos como no expertos, reducir la complejidad de diseñar flujos de algortimos de aprendizaje automático a través de una serie de herramientas, algoritmos pre-definidos, y protocolos extensibles. La Figura 3.1 ilustra los componentes más relevantes de AutoGOAL, desde la API de alto nivel hasta la implementación de adaptadores de algoritmos existentes y las interfaces a recursos externos y bibliotecas de *back-end*. La herramienta se encuentra disponible bajo una licencia de código abierto.[1]

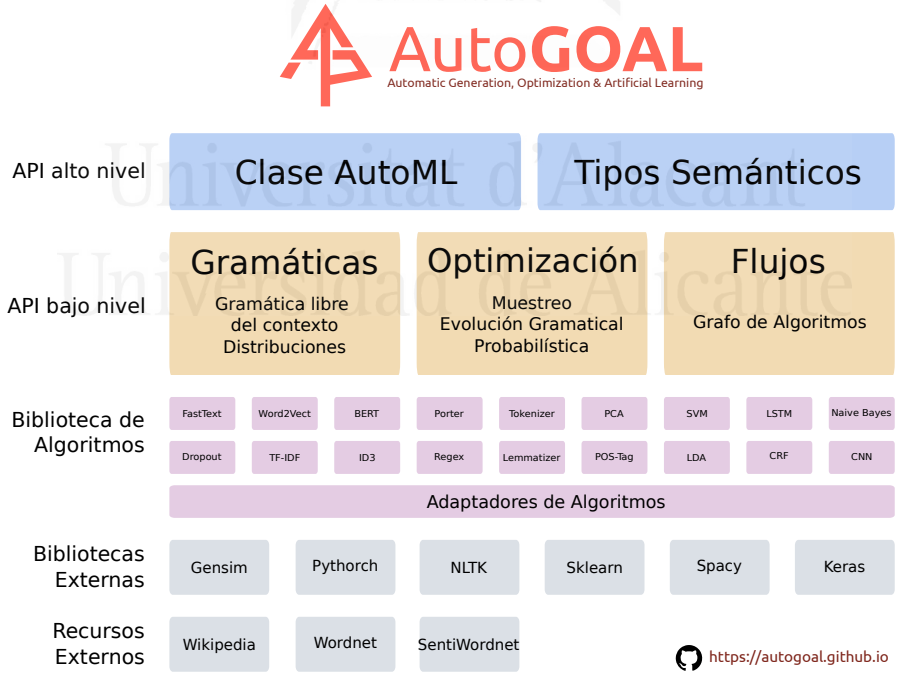


Figura 3.1: Arquitectura general de AutoGOAL.

[1] https://autogoal.github.io

El núcleo de la biblioteca AutoGOAL es la API de bajo nivel (ver Seccion 3.1.2), compuesta de los siguientes elementos: un módulo de gramáticas probabilísticas libres del contexto (*Grammar*); un módulo de muestreo y optimización (*Optimisation*); y, un módulo de descubrimiento de flujos (*Pipelines*). Esta API de bajo nivel permite a los usuarios: proporcionar sus propias implementaciones de algoritmos de aprendizaje automático; declarar los elementos que se deben optimizar (por ejemplo, hiperparámetros); y definir cómo se pueden conectar en flujos complejos de varios pasos.

El módulo *Grammar* proporciona un conjunto de anotaciones de tipo que se utilizan para definir el espacio de hiperparámetros de una técnica o algoritmo arbitrario. Cada técnica se representa como una clase de Python, y los hiperparámetros correspondientes se representan como argumentos anotados del método `__init__`, ya sea valores primitivos (por ejemplo, numéricos, texto, etc.) o instancias de otras clases, anotadas recursivamente. Dada una colección de clases anotadas, este módulo infiere automáticamente una gramática libre de contexto que describe el espacio de todas las instancias posibles de esas clases.

El módulo *Optimization* proporciona estrategias de muestreo sobre una gramática libre del contexto que construye recursivamente una instancia específica basada en las anotaciones. Se implementan dos estrategias de optimización: búsqueda aleatoria y evolución gramatical probabilística [40]. Esta última realiza un ciclo de muestreo/actualización que selecciona las instancias de mejor rendimiento de acuerdo con alguna métrica predefinida (p.e., precisión) y actualiza iterativamente el modelo probabilístico interno del algoritmo de muestreo.

El módulo *Pipelines* proporciona una abstracción para que los algoritmos se comuniquen entre sí a través de un patrón *Facade*, es decir, la implementación de un método `run` con anotaciones para los tipos de entrada y salida. Las clases que implementan este patrón se conectan automáticamente en un grafo de algoritmos donde cada ruta representa un posible flujo para resolver un problema, especificado por los tipos de datos de entrada y salida.

La API de alto nivel se basa en estos módulos, y proporciona la clase `AutoML` (ver Sección 3.1.1) y los tipos de datos semánticos. Los usuarios de la biblioteca pueden interactuar con la API de alto nivel de forma transparente, o interactuar directamente con los componentes internos de bajo nivel para

tener mayor control sobre las soluciones.

AutoGOAL también proporciona una biblioteca de algoritmos con adaptadores pre-definidos para tecnologías de aprendizaje automático disponibles en bibliotecas y recursos externos. Se proporcionan un total de 133 algoritmos de 7 bibliotecas externas[2], varios de los cuales se crean semi-automáticamente mediante introspección de código, y el resto se agregan manualmente por los desarrolladores de la biblioteca. Esta biblioteca está en continuo desarrollo.

AutoGOAL se puede instalar como un paquete de Python independientemente[3] de cualquier biblioteca de aprendizaje automático. En este caso funciona como una herramienta ligera que proporciona todos los bloques de construcción pero ninguno de los adaptadores pre-definidos. Los usuarios pueden instalar opcionalmente cualquiera de las bibliotecas externas compatibles. AutoGOAL la descubrirá automáticamente y registrará los adaptadores correspondientes, que estarán disponibles para su uso mediante la API de alto nivel. Además, se proporciona una imagen de Docker con todas las dependencias opcionales y bibliotecas externas ya instaladas [4].

3.1.1. Interfaz de Alto Nivel (*High-Level API*)

Esta API permite utilizar AutoGOAL como un algoritmo de clasificación o regresión de caja negra con una interfaz similar a la biblioteca *scikit-learn* [74]. Detrás de esta interfaz, se realiza un proceso completo que incluye preprocesamiento, selección de características, reducción de dimensionalidad y aprendizaje. El usuario debe definir un conjunto de datos de entrenamiento y evaluación, una métrica para optimizar (que por defecto es *precision*) y el tipo de datos de entrada y salida. En muchos casos, AutoGOAL puede inferir automáticamente el tipo de entrada y salida del conjunto de datos. Los tipos de datos pueden variar desde tabulares a tipos más complejos, como imágenes, texto en lenguaje natural con diferentes estructuras semánticas y combinaciones de los mismos. La figura 3.2 muestra un ejemplo de código fuente, específicamente en el contexto de un problema de clasificación de texto.

[2] Incluyendo *scikit-learn*, *nltk*, *gensim*, *spacy*, *keras*, *pytorch*, entre otros.
[3] `https://pypi.org/project/autogoal`
[4] `https://hub.docker.com/repository/docker/autogoal/autogoal`

```
from autogoal.ml import AutoML
from autogoal.datasets import haha
from autogoal.kb import List, Sentence, CategoricalVector

automl = AutoML(
    input=List(Sentences()),    # tipos de entrada
    output=CategoricalVector() # y salida
)

X, y = haha.load() # cargar datos del dominio especifico
automl.fit(X, y)   # ejecutar optimizacion
```

Figura 3.2: Ejemplo de código fuente para ejecutar AutoGOAL en un conjunto de datos específico, en este caso, un problema de PLN.

3.1.2. Interfaz de Bajo Nivel (*Low-Level API*)

Esta API está diseñada para usuarios con más experiencia que necesitan control sobre el proceso de AutoML. Para este tipo de usuario, AutoGOAL proporciona un lenguaje simple para definir una gramática que describe el espacio de la solución. Esto se realiza utilizando un enfoque orientado a objetos donde el usuario define una clase de Python para cada componente de la solución (por ejemplo, cada algoritmo). Los parámetros del constructor de estas clases se anotan con atributos que describen el espacio de valores posibles, que pueden ser tipos básicos (es decir, numéricos, texto, etc.) e instancias de otras clases, recursivamente. En base a las anotaciones, AutoGOAL puede construir automáticamente todas las formas posibles en las que se pueden instanciar las clases del usuario.

La API de bajo nivel proporciona utilidades para anotar los tipos de parámetros en constructores de clase y métodos, lo que indica el rango válido para sus valores. El usuario define un adaptador para un componente de una biblioteca externa, que sea compatible con la API de AutoGOAL, por ejemplo, un algoritmo de *scikit-learn*. Como ejemplo, la Figura 3.3 muestra la definición de clases que envuelven algoritmos *scikit-learn* y definen el espacio de búsqueda de hiper-parámetros al anotar los parámetros de interés en el constructor de la clase.

Además, cada clase debe definir un método `run` cuyos parámetros de en-

```
class LR(sklearn.linear_model.LogisticRegression):
    def __init__(
        self,
        penalty: Categorical("l1", "l2"),
        C: Continuous(0.1, 10)
    ):
        super().__init__(penalty=penalty, C=C)

    def run(self,
        X: MatrixContinuous,
        y: Supervised[CategoricalVector])
        -> CategoricalVector:
        if self.training:
            self.fit(X, y)
            return y
        else:
            return self.predict(X)
```

Figura 3.3: Ejemplo de definición de adaptadores para algoritmos de *scikit-learn*.

trada y salida se anotan con tipos semánticos. Esto permite que AutoGOAL detecte qué componentes se pueden conectar. Los algoritmos de aprendizaje supervisados de *scikit-learn* reciben como entrada tanto la matriz de características como las clases (durante el entrenamiento). Los métodos `run` en estas clases actúan como un adaptador entre la API de AutoGOAL y la API *scikit-learn*. Una implementación simplificada se muestra en la Figura 3.3.

Esta API se puede usar en cualquier nivel de detalle. Por ejemplo, de la misma manera puede implementarse un algoritmo para obtener representaciones de *word2vec* de palabras individuales utilizando *gensim*, como se observa en la Figura 3.4. En este caso el algoritmo se ejecuta a nivel de token, a diferencia de los adaptadores de *scikit-learn*, que se ejecutan a nivel del conjunto de datos completo. Como se explica en la Sección 3.3.3, AutoGOAL es capaz automáticamente de combinar algoritmos que funcionan a diferentes niveles de la estructura de los datos.

Como ejemplo final, esta API también puede representar técnicas de extracción de características que utilizan recursos externos, como Wikipedia y WordNet. La Figura 3.5 muestra la implementación de un extractor de

```
class Word2VecEmbedding:
    def __init__(self):
        # cargar modelo word2vec de la API de gensim
        self.model = gensim.downloader.load("glove-twitter-25")

    def run(self, input: Word) -> ContinuousVector:
        try:
            return self.model.get_vector(input)
        except:
            return np.zeros(25)
```

Figura 3.4: Ejemplo de definición de un componente para el cómputo de vectores de *embedding*.

```
class WikipediaSummary:
    def run(self, input: Word)-> Summary:
        try:
            return wikipedia.summary(input)
        except:
            return ""
```

Figura 3.5: Ejemplo de definición de un componente para extraer el resumen de un artículo de Wikipedia.

resumen de Wikipedia, que utiliza una biblioteca externa para acceder a Wikipedia y devolver el resumen de la primera coincidencia.

3.2. Diseño Algorítmico

En esta sección se describen los algoritmos y estrategias computacionales que forman parte del núcleo de funcionamiento de AutoGOAL. Se presenta un algoritmo para la construcción eficiente del espacio de búsqueda de todos los flujos dados los tipos semánticos de entrada y salida, una estrategia de muestreo y un algoritmo de optimización probabilístico basado en Evolución Gramatical [40].

3.2.1. Construcción del espacio de búsqueda

El espacio de búsqueda de este problema se compone de un grafo G_A dirigido y acíclico (DAG) donde cada nodo representa un algoritmo. Las aristas se definen entre algoritmos con tipos de entrada/salida compatibles. Los espacios de configuraciones de hiperparámetros de cada uno de los algoritmos corresponden a gramáticas libres del contexto definidas para cada nodo. AutoGOAL aprovecha estra estructura de grafo definida para un problema específico y genera automáticamente una gramática general a partir del DAG correspondiente.

El proceso de creación del espacio de búsqueda comienza por encontrar todos los algoritmos reconocidos por AutoGOAL y relacionarlos en dependencia de su compatibilidad. Una vez creado este grafo, se adicionan dos nuevos nodos simbólicos que representan los tipos de entrada y salida respectivamente. Los algoritmos con tipos de entrada compatible con el primero tendrán una arista desde el mismo, mientras que los compatibles con la salida presentarán una arista hacia el segundo. Para asegurar un espacio de flujos válidos se realizan dos recorridos en profundidad (DFS), uno con origen en el nodo Entrada y otro desde el nodo Salida pero considerando la dirección inversa de las aristas. Todo algoritmo visitado por ambos pertenece a un paso de algún flujo de solución del problema.

Por cada algoritmo en este grafo, AutoGOAL infiere una gramática probabilística libre del contexto. Dicha gramática contiene producciones por cada hiperparámetro. Los hiperparámetros con valores continuos, discretos, categóricos o booleanos generan producciones que producen un valor aleatorio de una distribución adecuada. En cambio, los hiperparámetros que representan otros algoritmos construyen recursivamente la gramática correspondiente, resolviendo correctamente cualquier dependencia recursiva.

Dado un nodo, la selección de alguno de sus vecinos puede ser vista como un hiperparámetro categórico. Por esta razón, se puede definir una gramática general que representa la estructura de este grafo. Añadiendo las producciones generadas por cada algoritmo, se obtiene una gramática que representa todo el espacio de búsqueda para el problema.

Dada un problema específico con tipos T^*_{in}, T^*_{out} de entrada y salida respectivamente, una solución consiste en un flujo $p =< a^1, \ldots, a^n >$ tal que

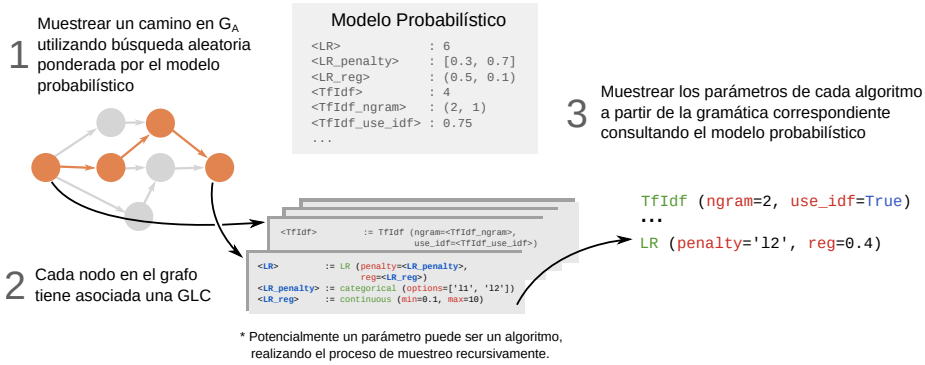


Figura 3.6: Representación visual del proceso de creación y muestreo del espacio de búsqueda.

$T^*_{in} \leq T^p_{in}$ y $T^p_{out} \leq T^*_{out}$. Considerando el grafo construido anteriormente, cualquier camino que comience en el nodo Entrada y termine en el nodo Salida representa un flujo que resuelve dicho problema. La Figura 3.6 muestra una representación visual del proceso descrito anteriormente.

3.2.2. Proceso de Optimización

Sobre el grafo de algoritmos G_A de un problema específico se realiza un proceso de optimización para descubrir los mejores flujos válidos mediante generación aleatoria. Este proceso de optimización se basa en evolución gramatical probabilística para gramáticas libres del contexto [123], y consiste en un ciclo de generación y evaluación utilizando una gramática construida a partir de G_A, dirigida por un modelo probabilistico σ.

Para generar los flujos de algoritmos, primero se debe considerar que, por construcción, cada nodo pertenece a un camino válido. Por tanto, realizando un recorrido aleatorio comenzando por el nodo Entrada, si no se repiten nodos y toda arista tiene probabilidad de cruzarse mayor que 0, entonces se garantiza que este recorrido termina en el nodo Salida. A cada algoritmo a_i en G_A se le asigna un peso (no normalizado) w_i, que se utiliza para seleccionar un vecino aleatorio durante la generación de un camino en G_A siguiendo una distribución multinomial Bernoulli.

Sea $p =< a^1, \ldots, a^n >$ un camino a generar en G_A, AutoGOAL infiere

una gramática por cada $a^i \in p$. A cada producción de la gramática se le asigna, además, un conjunto de parámetros en dependencia de la distribución utilizada (por ejemplo, media y varianza para valores numéricos). Se garantiza que los hiperparámetros de un mismo algoritmo en un flujo determinado comparten los mismos valores de parámetros. A partir de estas gramáticas es posible generar instancias de los flujos.

El modelo σ se inicializa con valores neutrales para cada distribución (pesos uniformes para distribución categórica, media centrada y máxima varianza para distribuciones continuas). El proceso de optimización consiste en un ciclo de generación y evaluación. Primeramente, se generan n flujos siguiendo el modelo de muestreo. Utilizando una función de evaluación $\varphi(p)$ (dada por el usuario para el problema a resolver) se evalúan los flujos y se seleccionan los $k < n$ mejores. El valor del mejor flujo del ciclo es comparado con el mejor global y actualizado en consecuencia. Tomando los valores de muestra de los hiperparámetros generados, se construye un modelo probabilístico marginal σ^*. Luego, el modelo σ^* y el modelo original σ son mezclados utilizando un factor de interpolación $\alpha \in [0, 1]$, ofreciendo un balance entre exploración y explotación.

Este ciclo se repite hasta que se llegue a un límite de tiempo de ejecución, una cantidad determinada de iteraciones, o hasta que no se encuentre mejora. Por cada iteración el modelo σ converge lentamente a un modelo que maximiza la probabilidad de producir los mejores flujos. Se define $\varphi(p) = 0$ si el flujo p resulta en un error de ejecución o sobrepasa el límite de memoria o de tiempo, provocando que eventualmente se generen solo flujos que cumplan con estas restricciones. Al terminar el proceso de optimización, el mejor flujo global es devuelto como resultado, siendo potencialmente la mejor solución para el problema en cuestión.

El proceso anterior optimiza los flujos a varios niveles. Primero, converge a la selección de las secuencias de algoritmos en G_A que producen una mejor evaluación. Luego, cada uno de los hiperparámetros de estos algoritmos son ajustados incrementalmente. Los hiperparámetros que consisten en instancias de otros algoritmos repiten este proceso recursivamente.

3.3. Implementación Computacional

En esta sección se presenta los elementos más relevantes de la implementación computacional que hacen posible el funcionamiento práctico de AutoGOAL. Se introduce la modelación de tipos semánticos así como de la función de compatibilidad entre los mismos. Se concreta el concepto de componentes en el ámbito de AutoGOAL, y se presentan dos "meta-tipos" que ayudan en el proceso de creación de flujos. Además, se describe la biblioteca de algoritmos, así como los métodos de instrospección y los protocolos utilizados para crear compatibilidad entre diferentes bibliotecas.

3.3.1. Tipos de Datos

Una de las tareas computacionales identificadas en esta investigación, es la necesidad de representar tipos de datos semánticos que recojan el concepto de compatibilidad de tipos. Un algoritmo en AutoGOAL representa una función que transforma una determinada entrada de tipo T_{in} en una salida de tipo T_{out} deseada. Estos tipos se representan como una jerarquía de clases en la cual la herencia determina la relación de compatibilidad $\leq$. Los tipos de datos tienen una interpretación semántica más allá de su estructura computacional subyacente. Por ejemplo, una cadena de caracteres (`string`) en términos computacionales puede ser un `Document`, `Sentence` o `Word`. De igual manera, una lista de valores puede representar un vector de valores discretos o un vector de categorías, dependiendo de la interpretación más adecuada para un problema determinado.

En su implementación actual, AutoGOAL define 23 tipos semánticos de datos (ver Figura 3.7), incluyendo varios para datos de lenguaje natural como `Token` o `Stem`. Estos tipos semánticos son los que en última instancia conectan implementaciones concretas de algoritmos dentro de la biblioteca.

3.3.2. Componentes

Cada algoritmo en AutoGOAL es implementado como una clase con un método `run(input:Tin) -> Tout` que realiza el proceso correspondiente, potencialmente encapsulando una implementación subyacente de una biblioteca externa. Los tipos semánticos de entrada y salida son expresados

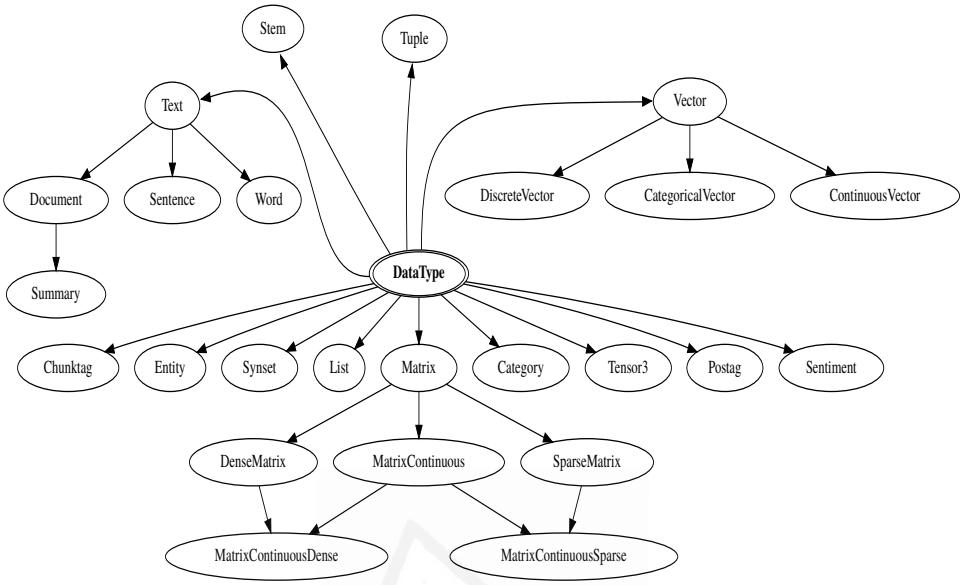


Figura 3.7: Tipos semánticos contemplados en AutoGOAL, incluyendo los "meta-tipos" de datos presentados en 3.3.3. Los nodos del grafo representan clases de tipos de datos, las flechas indican herencia directa en el orden contrario.

como anotaciones y corresponden a instancias de las clases presentadas en la sección 3.3.1.

En cada implementación de un algoritmo se define un contructor con argumentos semánticamente anotados que describen los posibles rangos de valores de sus hiperparámetros. Los argumentos pueden ser valores discretos, continuos, categóricos, booleanos, o instancias de alguna otra clase. Cada argumento provee los rangos válidos para el hiperparámetro correspondiente. Por ejemplo, anotaciones discretas y continuas definen valores máximos y mínimos, mientras que las categóricas presentan una lista de posibles valores (ver Figura 3.3). En el caso de los hiperparámetros que son instancias de otros algoritmos, AutoGOAL es capaz de encontrar el conjunto de clases válidas por los que pueden ser reemplazados.

Como se explica en la Sección 3.1.2, una componente debe tener un método `run(input:Tin)->Tout` con anotaciones adecuadas, que realice el procesamiento debido. Añadir un nuevo algoritmo solo requiere de una implementación válida de una clase con dicho método y AutoGOAL podrá incluirlo en los flujos posibles. Esto es gracias a una estrategia de inyección

de dependencias que permite al sistema reconocer automáticamente nuevos algoritmos. Este proceso dota a AutoGOAL de un alto grado de extensibilidad, permitiendo a desarrolladores incluir nuevas técnicas, sin requerir un conocimiento a fondo de la biblioteca.

3.3.3. Meta-tipos

Además de los tipos semánticos presentados en la Sección 3.3.1, AutoGOAL cuenta con dos meta-tipos que ayudan en el proceso de descubrimiento de flujos: `List[.]` y `Tuple[.]`. Estos meta-tipos permiten explorar y descubrir flujos complejos y realmente heterogéneos. De esta manera, es posible combinar automáticamente algoritmos de diferentes bibliotecas y conectar algoritmos que funcionan a diferentes niveles en la estructura de los datos (p.e., algoritmos a nivel de token con algoritmos a nivel de documento) sin requerir que el usuario implemente todas las posibles combinaciones de adaptadores.

Lista El tipo `List[.]` representa una colección de elementos de otro tipo. Permite generar automáticamente, dado un algoritmo $a : T_{int} \rightarrow T_{out}$, un meta-algoritmo que recibe `List[Tin]` y devuelve `List[Tout]`. De esta manera algoritmos como *stemmers* que, dada un tipo `Word`, devuelven un tipo `Stem`, pueden ser utilizados con naturalidad para tratar con el resultado de un tokenizador de palabras (que recibe `Sentence` y devuelve `List[Word]`).

Tupla El meta-tipo `Tuple[.]` permite la concatenación de distintos tipos para lograr entradas más complejas. Por ejemplo, un algoritmo supervisado de clasificación recibe como entrada un elemento de tipo `Tuple[Tin, CategoricalVector]`, representando los datos de entrenamiento y sus clases correspondientes. AutoGOAL puede deconstruir y construir estas tuplas como sea necesario. Esto permite a los algoritmos que no reciben etiquetas supervisadas (p.e., PCA, SVD, etc.), ser insertados automáticamente en un flujo donde las etiquetas supervisadas están presentes.

3.3.4. Biblioteca de Algoritmos

Como se muestra en la Sección 1.1, los sistemas AutoML utilizan implementaciones de algoritmos provistas por una o varias bibliotecas de aprendizaje automático. Una mayor cantidad de algoritmos integrados en el sistema implica un mayor espacio de búsqueda y, por tanto, más probabilidad de que exista un flujo adecuado para un problema determinado. AutoGOAL, en esta implementación, incluye un total de 107 algoritmos correctamente anotados. El código pertinente fue generado semi-automáticamente mediante instrospección de código de populares bibliotecas de aprendizaje automático como Scikit-learn [74], Keras [124], NLTK [125], Gensim [126] y Pytorch [107]; con excepción de algunas implementaciones manuales (e.g., spaCy [5]).

AutoGOAL implementa dos procesos de introspección de código. El primero explora cada biblioteca de aprendizaje de máquina y genera adaptadores correctamente preparados y anotados para los algoritmos identificados. El segundo recolecta todos las componentes producidas por el anterior así como las implementaciones manuales y construye una colección global de todas las componentes válidas para el grafo de algortimos dados los tipos de entrada/salida.

3.3.5. Adaptadores y Protocolos

AutoGOAL integra varios tipos de algoritmos, de distintas bibliotecas, gracias a un conjunto de adaptadores que se encargan de implementar un protocolo de ejecución común. Estos son representados como clases de Python, de las cuales los algoritmos heredan el método `run(input:Tin)->Tout`.

Durante el ciclo de optimización, AutoGOAL genera cierta cantidad de flujos que requieren ser evaluados. Al evaluar un flujo determinado, el sistema instancia cada una de las clases pertenecientes al mismo, y prepara los datos, dividiéndolos en conjuntos entrenamiento y prueba. Se ajusta el estado interno de las componentes para entrenar y se ejecuta el flujo con el conjunto de datos correspondiente. Una vez terminado el proceso, AutoGOAL cambia nuevamente el estado de los algoritmos, y realiza una nueva ejecución del flujo con los datos prueba. Estos pasan a ser analizados por la métrica $\varphi(p)$ definida por el usuario, obteniendo una puntuación que

[5] `https://spacy.io`

permite comparar los flujos entre sí.

3.4. AutoGOAL

El resultado práctico de esta Tesis se concreta en la herramienta AutoGOAL que ha sido presentada en este Capítulo. AutoGOAL permite resolver el problema de AutoML heterogéneo a partir de la combinación de algoritmos y técnicas de múltiples herramientas. La biblioteca es fácil de extender y se encuentra en continuo desarrollo. Puede ser utilizada tanto por usuarios expertos en dominios específicos, como por usuarios no expertos en problemas generales de AutoML. La herramienta se encuentra disponible bajo una licencia de código abierto en `https://autogoal.github.io`.

Capítulo 4

Resultados Experimentales

El objetivo de este capítulo es comparar el paradigma de AutoML diseñado en esta Tesis, con otros enfoques de AutoML existentes, así como estrategias diseñadas por investigadores para problemas específicos, a partir de evaluar la implementación computacional desarrollada. Las comparativas presentadas en este capítulo muestran la versatilidad de AutoGOAL para tratar con problemas de diferentes dominios y combinar algoritmos y tecnologías de distinta naturaleza. En la Sección 4.1 se presenta una comparativa experimental entre AutoGOAL y 6 herramientas de AutoML existentes en 7 datasets clásicos propuestos en el estado del arte. El objetivo de esta comparativa es evaluar la calidad de los flujos de algoritmos obtenidos por AutoGOAL en relación con las herramientas de AutoML comúnmente utilizadas por la comunidad. En la Sección 4.2 y la Sección 4.3 se compara AutoGOAL con soluciones diseñadas por expertos para 3 problemas de procesamiento de lenguaje natural de diferente naturaleza. Estas comparativas muestran el potencial de AutoGOAL para resolver problemas complejos que requieren conocimiento del dominio. Finalmente, en la Sección 4.4 se presenta un análisis general del proceso de optimización y en la Sección 4.5 una discusión sobre las potencialidades de AutoGOAL como herramienta de ayuda en el desarrollo de soluciones de aprendizaje automático.

4.1. Comparación con Sistemas de AutoML

AutoGOAL ha sido comparado en varios problemas de diversos dominios con herramientas alternativas. La metodología de comparación más usual en el campo de AutoML consiste en ejecutar el proceso de optimización durante un tiempo determinado y reportar la evaluación del mejor flujo obtenido. En esta investigación se compara AutoGOAL con 6 herramientas de AutoML distintas en 7 conjuntos de datos diferentes del estado del arte, reportando la media de precisión obtenida en 20 ejecuciones de 1 hora cada una, para cada herramienta. Además, AutoGOAL se evalúa en 2 problemas adicionales de procesamiento de lenguaje natural en los cuáles las herramientas de AutoML existentes no pueden ser aplicadas. La Tabla 4.1 resume estos resultados experimentales.

Dataset	**Cars**	**Credit G.**	**Abalone**	**Shuttle**	**Yeast**	**Dorothea**	**Gisette**	**HAHA**	**MEDDOCAN**
ML-Plan (Weka)	1.27	25.54	73.72	0.01	39.37	6.49	2.92	-	-
Auto-WEKA	0.66	26.50	73.46	0.12	39.72	-	3.90	-	-
ML-Plan (Sklearn)	0.34	24.56	73.77	0.02	39.52	8.69	2.76	-	-
Auto-Sklearn-v	1.38	25.95	82.92	0.02	40.51	6.32	2.56	-	-
Auto-Sklearn-we	1.26	25.39	80.59	0.02	38.99	6.02	2.24	-	-
TPOT	0.37	23.91	73.14	0.02	38.47	-	-	-	-
AutoGOAL	0.60	27.01	74.33	0.11	39.94	5.97	2.25	21.1	3.99

Tabla 4.1: Comparación de AutoGOAL y otros sistemas AutoML para 9 conjuntos de datos clásicos de aprendizaje automático en términos de precisión, excepto MEDDOCAN, en el que se utiliza la métrica F_1. Los valores para otros sistemas se obtuvieron de *ML-Plan*.

Los resultados experimental demuestran que las herramientas de AutoML existentes se comportan de forma similar en los problemas clásicos. A partir de pruebas estadísticas de comparación de medias, se llega a la conclusión de que no existen diferencias significativas entre ninguna de las herramientas evaluadas. Esto significa que AutoGOAL es competitivo con el estado del arte en los problemas de AutoML que dichas herramientas son capaces de

solucionar, a la vez que puede aplicarse en otros dominios.

4.2. Evaluación en Problemas de Procesamiento de Lenguaje Natural

Para evaluar el comportamiento de AutoGOAL en problemas más complejos que la clasificación de datos tabulares se presenta una comparación con el estándar humano en 2 problemas de procesamiento de lenguaje natural. Ambos problemas fueron propuestos en el taller *IberLEF 2019*. Los resultados alternativos con los que se compara AutoGOAL corresponden a las mejores técnicas desarrolladas por equipos de investigadores que participaron en el taller, que tuvo una duración de más de 1 mes desde la liberación de los datos hasta la recepción de los resultados. En contraste, AutoGOAL se ejecutó de manera automática durante 48 horas como máximo, y requiriendo desde el punto de vista del esfuerzo humano solamente la adaptación de los conjuntos de datos a los tipos semánticos.

La Tabla 4.2 muestra la comparación entre AutoGOAL y el estándar humano en el corpus HAHA (*Humor Analysis based on Human Annotation*). Este corpus presenta un problema de clasificación de texto en el dominio de Twitter, para identificar aquellos mensajes que muestran humor [127]. Los resultados se comparan en términos de F_1 de la clase positiva. El corpus contiene $30,000$ mensajes anotados manualmente, de los cuáles $24,000$ se usan para entrenamiento y $6,000$ para evaluación. Los resultados obtenidos por AutoGOAL son competitivos con el estándar humano. De haber participado en el taller, AutoGOAL hubiera obtenido un 5to lugar de un total de 19 sistemas, requiriendo un esfuerzo considerablemente menor por parte de los investigadores.

La Tabla 4.3 muestra la comparación entre AutoGOAL y el estándar humano en el corpus MEDDOCAN (*Medical Document Anonymity*). En este caso se presenta un problema de detección de entidades en lenguaje natural [128], un dominio en el cuál ninguno de los sistemas de AutoML analizados en el estado del arte puede ser aplicado. El corpus contiene $1,000$ documentos de estudios clínicos anotados manualmente con entidades que presentan problemas de privacidad, donde 750 documentos se usan para entrenamiento y 250 para evaluación. Los resultados obtenidos por AutoGOAL

Participantes	F_1	**Participantes**	F_1
adilism	82.1	LaSTUS/TALN	75.9
Kevin-Hiromi	81.6	Taha	75.7
bfarzin	81.0	LadyHeidy	72.5
jamestjw	79.8	Aspie96	71.1
INGEOTEC	78.8	OFAI–UKP	66.0
BLAIR GMU	78.4	acattle	64.0
UO UPV2	77.3	jmeaney	63.6
vaduvabogdan	77.2	garain	59.3
UTMN	76.0	Amrita CEN	49.5
AutoGOAL	**78.9**		

Tabla 4.2: Comparación de AutoGOAL con los resultados oficiales del corpus HAHA.

en este problema son también competitivos con el estándar humano. De haber participado en el taller, AutoGOAL hubiera obtenido un 3er lugar de un total de 17 sistemas, requiriendo un esfuerzo considerablemente menor por parte de los investigadores.

En ambos experimentos, AutoGOAL muestra un resultado superior a la mayoría de los sistemas diseñados por expertos. Esto significa que el uso de AutoGOAL proporciona una ventaja competitiva en problemas de aprendizaje automático en dominios complejos. Aunque no se obtiene el mejor resultado, el tiempo necesario para adaptar AutoGOAL a estos problemas es significativamente menor que el necesario para diseñar una estrategia desde cero.

Participantes	F_1	**Participantes**	F_1
lukas.lange	96.96	ccolon	93.22
Fadi	96.32	sohrab	93.11
nperez	96.01	Jordi	91.84
FSL	95.95	plubeda	90.38
mhjabreel	95.83	m.domrachev	90.00
lsi uned	94.33	lsi2 uned	89.97
jiangdehuan	94.01	vcotik	89.67
jimblair	93.75	VSP	86.00
AutoGOAL	**96.01**		

Tabla 4.3: Comparación de AutoGOAL con los resultados oficiales del corpus MEDDOCAN.

4.3. Evaluación en Problemas de Descubrimiento de Conocimiento

Como parte del desarrollo de la investigación, una versión inicial del mecanismo de optimización basada en gramáticas libres del contexto fue evaluado en el corpus *eHealth-KD 2018*, nuevamente comparado con el estándar humano, debido a la inhabilidad de los sistemas de AutoML existentes de lidiar con este tipo de problemas. Este corpus contiene 944 oraciones anotadas manualmente con un esquema semántico de entidades y relaciones. Se emplean 844 oraciones para el entrenamiento y 100 oraciones para la evaluación. En este caso se utilizaron un total de 257 horas de cómputo en los que evaluaron un total de 3000 flujos diferentes. La tabla 4.4 resume los resultados obtenidos.

El resultado final obtenido supera a todos las soluciones presentadas en esta tarea, nuevamente sin intervención humana más allá de la definición del espacio de búsqueda. Además, la solución encontrada por AutoGOAL consiste en un flujo de algoritmos clásicos (regresión logística, árboles de decisión y máquinas de soporte vectorial), mientras que las mejores estrategias diseñadas por expertos emplean modelos de aprendizaje profundo. Esto demuestra que un ajuste de parámetros riguroso es capaz de superar a modelos más complejos pero poco optimizados.

Enfoques	F_1 **(Escenario 1)**
Rivera-Zavala et al. [129]	0.744
López-Úbeda et al. [130]	0.710
Palatresi and Hontoria [131]	0.681
Suárez-Paniagua et al. [132]	0.310
AutoGOAL (*versión inicial*)	**0.754**

Tabla 4.4: Comparación de enfoques en el desafío eHealth-KD. Solo se consideran los investigadores que participaron en el Escenario 1.

4.4. Análisis del Proceso de Optimización

Analizar la evolución del rendimiento de los flujos proporciona una idea de la complejidad de un problema de aprendizaje automático. Como ejemplo ilustrativo, la Figura 4.1 (izquierda) muestra la evolución de la precisión promedio y óptima en el problema presentado en la Sección 4.3. La precisión promedio aumentó constantemente, evidenciando la mejora producida por el proceso de optimización. El valor inicial está en gran parte influenciada por la existencia de *flujos inválidos*. Los flujos inválidos se encuentran en dos casos: cuando se excede el tiempo de espera predefinido por el investigador, o cuando ocurren errores de tiempo de ejecución impredecibles, como errores de falta de memoria provocados por una combinación inviable de hiperparámetros. Estas circunstancias a menudo son imposibles de predecir de antemano y, como tal, no se pueden tener en cuenta en las gramáticas. Los flujos inválidos en tiempo de ejecución se les asigna un valor $\varphi(p) = 0$, lo cuál provoca que el proceso de optimización converja hacia regiones del espacio de hiperparámetros que no generen este tipo de flujos. Como muestra la Figura 4.1 (derecha), el número de flujos inválidos disminuye constantemente, lo cual es una indicación de que el modelo probabilístico está penalizando las características que aparecen consistentemente en los flujos inválidos.

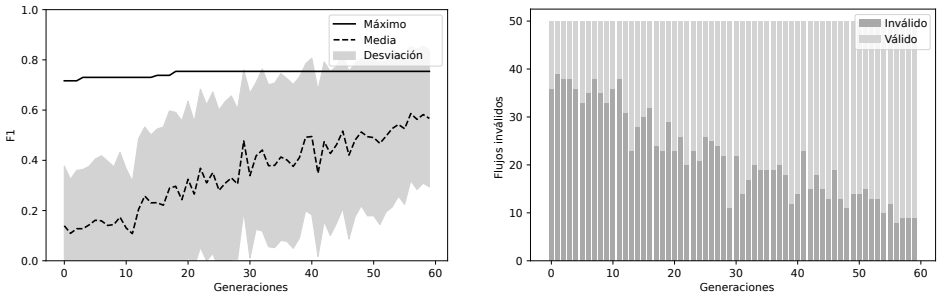


Figura 4.1: Izquierda: Evolución de la precisión promedio y óptima de los flujos en cada iteración. **Derecha:** Proporción de flujos inválidas vs válidas encontradas en cada iteración.

4.5. Discusión

En cuanto a los resultados experimentales, es importante tener en cuenta que AutoGOAL se implementa utilizando el mismo código en todos los experimentos (ver Figura 3.2), variando solo la definición de los tipos de entrada y salida. Por ejemplo, configurar los conjuntos de datos de HAHA y MEDDOCAN tomó aproximadamente una y cuatro horas respectivamente, dedicadas fundamentalmente a preparar los corpus en un formato adecuado. En los conjuntos de datos UCI, AutoGOAL converge rápidamente a una combinación de clasificadores simples y métodos no supervisados (reducción de dimensionalidad, selección de características, etc.). Sin embargo, en HAHA y MEDDOCAN, los clasificadores simples se descartan en favor de las técnicas de PLN y redes neuronales. En el caso de MEDDOCAN, AutoGOAL pudo detectar que las tecnologías de PLN son cruciales en este dominio dado el uso de la terminología médica, mientras que en el caso de HAHA, AutoGOAL identificó los *embeddings* de propósito general como los más efectivos considerando el contenido coloquial. Esto proporciona evidencia de que durante la optimización, AutoGOAL aprende automáticamente qué familias de algoritmos son las más adecuadas para cada problema.

Al abordar el problema de AutoML desde la perspectiva de su modelo de optimización subyacente, se hacen evidentes dos consideraciones interesantes: la naturaleza jerárquica de las soluciones y la naturaleza multiobjetivo de la métrica de rendimiento. La propuesta de esta Tesis representa explícitamente soluciones como un proceso de decisión jerárquico, de la misma manera que

otros enfoques alternativos de AutoML (p.e., RECIPE [16] y ML-Plan [19]). Esto permite modelar y tener en cuenta el hecho de que algunas decisiones tienen una mayor influencia en la optimización de los flujos. Por ejemplo, la selección entre clasificadores lineales, no lineales o basados en árboles es más importante que la selección de valores específicos de sus hiperparámetros. A medida que los flujos de aprendizaje automático crecen en complejidad e involucran algoritmos de diferentes bibliotecas y tecnologías, una conceptualización jerárquica como la propuesta en esta investigación será aún más relevante.

Con respecto a la métrica de rendimiento, la mayoría de los enfoques actuales de AutoML se centran en una única función objetivo, por ejemplo, precisión, recobrado o F_1, en correspondencia con el problema de aprendizaje automático en cuestión. Sin embargo, en escenarios prácticos, puede ser necesario equilibrar diferentes métricas de rendimiento, incluido también el uso del tiempo y la memoria, y cualidades más subjetivas como la interpretabilidad de los modelos o su capacidad para lidiar con datos sesgados. El enfoque de optimizar una métrica principal sujeta a restricciones de tiempo y memoria es insuficiente en un escenario en el que el usuario final tiene que decidir sobre cuestiones prácticas como el despliegue de estos flujos en un sistema de producción. A modo de ejemplo, TPOT considera este problema desde el enfoque multi-objetivo mediante la optimización conjunta de la precisión y la complejidad del modelo (en términos de longitud de los flujos). Sin embargo, los enfoques futuros deberán considerar este problema en profundidad, lo que podría incluir al usuario final en el ciclo de evaluación de los flujos de algoritmos.

Durante el proceso de optimización en AutoGOAL, se genera una cantidad significativa de datos en cada iteración sobre los mejores flujos y sus características. En principio, esta información se utiliza para actualizar el modelo probabilístico de forma que los nuevos flujos muestreados sean similares a los mejores flujos ya evaluados. Además, esta información puede proporcionar conocimiento adicional cuando se agrega para todo el proceso de optimización, analizando las características (es decir, las producciones de la gramática) que aparecen constantemente en los mejores flujos. Una tendencia reciente en el área de AutoML es incluir técnicas de meta-aprendizaje (*meta-learning*) que permitan reaprovechar la experiencia obtenida en evaluaciones anteriores en problemas similares para condicionar los algoritmos

de optimización a mejores regiones del espacio de búsqueda desde el inicio. Los datos obtenidos por AutoGOAL durante la optimización son una fuente valuable de información para el futuro desarrollo de este tipo de técnicas.

AutoGOAL permite a los investigadores y profesionales desarrollar rápidamente técnicas *baseline* sólidas en diversos problemas de aprendizaje automático. En algunos escenarios, la solución proporcionada por AutoGOAL y otros sistemas AutoML podría ser lo suficientemente buena. Sin embargo, los sistemas AutoML no solo deberían intentar reemplazar a expertos humanos, sino servir como herramientas complementarias que permitan a los investigadores obtener rápidamente mejores soluciones e información sobre las estrategias más prometedoras. Al igual que los compiladores trajeron una mejora significativa en la eficiencia del desarrollo de software, el campo de AutoML promete revolucionar la forma en que se realiza la investigación y la práctica del aprendizaje automático.

Parte II

Artículos Publicados

Capítulo 5

Modelación de Flujos de Algoritmos

En este capítulo se presenta el artículo *Optimizing Natural Language Processing Pipelines: Opinion Mining Case Study*. Este artículo introduce NLP-Opt, una técnica de AutoML para optimizar algoritmos de aprendizaje automático que se pueden aplicar a diferentes tareas de procesamiento del lenguaje natural. El proceso de selección de los algoritmos y sus parámetros se modela como un problema de optimización y se propuso una técnica para encontrar una combinación óptima basada en el algoritmo *Population Based Incremental Learning* (PBIL). Para fines de validación, este enfoque se aplica a un problema estándar de minería de opinión. NLP-Opt optimiza eficazmente los algoritmos y parámetros de las soluciones. Además, emite información probabilística sobre el proceso de optimización, revelando los componentes más relevantes. La técnica propuesta se puede aplicar a diferentes problemas de procesamiento del lenguaje natural, y la información proporcionada por NLP-Opt puede ser utilizada por los investigadores para obtener conocimiento sobre las características de las soluciones de mejor rendimiento. A diferencia de otros enfoques de AutoML, NLP-Opt proporciona un mecanismo flexible para diseñar flujos que pueden aplicarse a problemas de PLN. Además, el uso del modelo probabilístico proporciona un enfoque más integral del problema de AutoML que enriquece la comprensión del investigador de las posibles soluciones.

Entrada bibliográfica:
Estevez-Velarde S., Gutiérrez Y., Montoyo A., Almeida-Cruz Y. (2019) Optimizing Natural Language Processing Pipelines: Opinion Mining Case Study. In: Nyström I., Hernández Heredia Y., Milián Núñez V. (eds) *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. CIARP 2019. Lecture Notes in Computer Science, vol 11896. Springer, Cham.

Disponible en https://doi.org/10.1007/978-3-030-33904-3_15

Capítulo 6

Optimización de Flujos Jerárquicos

En este capítulo se presenta el artículo *AutoML strategy based on grammatical evolution: A case study about knowledge discovery from text*. En este artículo se propone una nueva estrategia AutoML basada en la evolución gramatical probabilística, que se evalúa en el dominio de la salud al enfrentar el problema de descubrimiento de conocimiento en documentos de texto en español. Este enfoque logra resultados del estado del arte y proporciona información interesante sobre la mejor combinación de parámetros y algoritmos para utilizar en este problema.

Entrada bibliográfica:
Estevez-Velarde, S., Gutiérrez, Y., Montoyo, A., Almeida-Cruz, Y. (2019, July). AutoML strategy based on grammatical evolution: A case study about knowledge discovery from text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (pp. 4356-4365).

Disponible en http://dx.doi.org/10.18653/v1/P19-1428

Capítulo 7

Modelación con Gramáticas Probabilísticas

Este capítulo presenta el artículo *General-purpose Hierarchical Optimisation of Machine Learning Pipelines with Grammatical Evolution*. Este artículo presenta HML-Opt, un sistema de AutoML que se basa en la evolución gramatical probabilística. HML-Opt ha sido diseñado para proporcionar un marco flexible donde un investigador puede definir el espacio de posibles flujos para resolver un problema específico de aprendizaje automático, que puede variar desde decisiones de alto nivel sobre representación y características hasta valores de hiperparámetros de bajo nivel. La evaluación de HML-Opt se presenta a través de dos estudios de caso diferentes, que demuestran que es competitiva con las herramientas AutoML existentes en una variedad de puntos de referencia. Además, HML-Opt se puede aplicar a problemas novedosos, como la extracción de conocimiento del texto en lenguaje natural, mientras que otras técnicas son insuficientemente flexibles para capturar la complejidad de estos escenarios.

Entrada bibliográfica:
Estévez-Velarde, S., Gutiérrez, Y., Almeida-Cruz, Y., Montoyo, A. General-purpose hierarchical optimisation of machine learning pipelines with grammatical evolution. Information Sciences, 543, 58-71.

Disponible en https://doi.org/10.1016/j.ins.2020.07.035

Capítulo 8

AutoML Heterógeneo

En este capítulo se presenta el artículo *Automatic Discovery of Heterogeneous Machine Learning Pipelines: An Application to Natural Language Processing*. Este artículo presenta un sistema de AutoML que utiliza técnicas heterogéneas. A diferencia de los enfoques existentes de AutoML, esta contribución puede construir automáticamente flujos de aprendizaje automático que combinen técnicas y algoritmos de diferentes bibliotecas, incluidos clasificadores lineales, herramientas de procesamiento de lenguaje natural y redes neuronales. Se define el problema heterogéneo de optimización de AutoML como la búsqueda de la mejor secuencia de algoritmos que transforma los datos de entrada específicos en la salida deseada. Esto proporciona un enfoque teórico y práctico novedoso para AutoML. La propuesta se evalúa experimentalmente en diversos problemas de aprendizaje automático y se compara con enfoques alternativos, lo que demuestra que es competitiva con otras alternativas de AutoML en los puntos de referencia estándar. Además, se puede aplicar a otros escenarios, como varias tareas de PLN, donde las alternativas existentes no se pueden implementar directamente.

Entrada bibliográfica:
Estevez-Velarde, S., Gutiérrez, Y., Montoyo, A., & Cruz, Y. A. (2020, December). Automatic Discovery of Heterogeneous Machine Learning Pipelines: An Application to Natural Language Processing. In *Proceedings of the 28th International Conference on Computational Linguistics* (pp. 3558-3568).

Disponible en http://dx.doi.org/10.18653/v1/2020.coling-main.317

Capítulo 9

Aplicación AutoGOAL

En este capítulo se presenta el artículo *Demo Application for the AutoGOAL Framework*. Este artículo introduce una aplicación web que permite interactuar con el sistema AutoGOAL. AutoGOAL es un sistema de AutoML implementado como una biblioteca de código abierto en el lenguaje de programación Python. A diferencia de otros sistemas AutoML, AutoGOAL puede ser utilizado directamente en múltiples dominios y tareas complejos, tales como el procesamiento de lenguaje natural. Se presenta un resumen general de la arquitectura de AutoGOAL y una evaluación experimental en varios problemas de aprendizaje automático, incluyendo dos problemas de procesamiento de lenguaje natural. Se brindan además varios detalles de implementación de la biblioteca.

Entrada bibliográfica:
Estevez-Velarde, Suilan, et al. Demo Application for the AutoGOAL Framework. In: *COLING 2020, The 27th International Conference on Computational Linguistics: Proceedings of System Demonstrations*, December 8-13, 2020 Barcelona, Spain (Online), ISBN 978-1-952148-28-6, pp. 18-22.

Disponible en http://hdl.handle.net/10045/110741

Parte III

Conclusiones y Recomendaciones

Capítulo 10

Conclusiones

La inteligencia artificial, y en particular el aprendizaje automático, es cada vez más demandado en la industria, debido al potencial que tiene para automatizar los procesos más complejos. En los últimos años se ha visto un incremento notable en la demanda de expertos en este campo en distintas áreas, desde el comercio minorista, el entretenimiento, las finanzas, la medicina, entre otras. Entre las principales dificultades para aplicar extensivamente técnicas de aprendizaje automático en problemas reales, están la poca disponibilidad de expertos unido al costo de diseñar, implementar y evaluar este tipo de soluciones.

Con el objetivo de mitigar esta situación, la comunidad científica se ha volcado hacia el desarrollo de herramientas de AutoML, que liberen a los expertos de las tareas menos creativas en la implementación de sistemas de aprendizaje automático. Aunque existen varias herramientas de AutoML que han sido exitosas resolviendo problemas concretos de inteligencia artificial, estas herramientas son aún demasiado rígidas para ser utilizadas en problemas prácticos que requieren la combinación de algoritmos y tecnologías de diferente naturaleza.

Este trabajo propone una nueva formulación de AutoML, mas completa y flexible, que puede ser adaptada a un amplio rango de problemas de aprendizaje automático. Para lograrlo, se define el problema de AutoML Heterogéneo, un marco conceptual que abarca los problemas usualmente considerados en el AutoML clásico y escenarios novedosos como el procesa-

miento de lenguaje natural y el descubrimiento de conocimientos. Además, se presenta una formalización del concepto de flujo de algoritmos a partir de una estructura jerárquica que permite representar computacionalmente soluciones algorítmicas a problemas complejos. Esta estructura permite definir desde las decisiones de alto nivel relacionadas con las clases de algoritmos a utilizar hasta decisiones de bajo nivel como el valor concreto de cada parámetro utilizado. Basado en esta estructura jerárquica se propone una estrategia de optimización probabilística que es capaz de optimizar flujos de algoritmos para un problema concreto en función del tiempo y los recursos de cómputo disponibles.

Como resultado práctico se ofrece la biblioteca AutoGOAL, un nuevo sistema de AutoML que combina tecnologías heterogéneas y se puede aplicar a una amplia variedad de escenarios de aprendizaje automático. Esta propuesta es competitiva con otros sistemas de AutoML en puntos de referencia estándar, y se puede aplicar a escenarios novedosos en los que varias herramientas existentes no son directamente aplicables.

AutoGOAL se ha diseñado para ser usado tanto por expertos como no-expertos en el área del aprendizaje automático. El despliegue del sistema para un problema determinado, a partir de un conjunto de datos, requiere de la definición de los tipos de entrada y salida correspondientes y la métrica de evaluación, aunque en múltiples ocasiones AutoGOAL puede inferir automáticamente estos elementos. De esta manera, la propuesta se hace accesible para profesionales sin conocimiento específico de aprendizaje de máquina. Para los usuarios expertos se ofrece una API de bajo nivel que permite personalizar el espacio de búsqueda y aplicar AutoGOAL en problemas de un dominio específico. Además, la biblioteca tiene un alto de grado de extensibilidad a partir de un sistema de introspección de código y una jerarquía de tipos unificada que permiten la adición de nuevos algoritmos implementados con cualquier tecnología subyacente, siempre y cuando cuenten con las anotaciones adecuadas. Esto permite una fácil adaptación del sistema a nuevas líneas de investigación.

El diseño del algoritmo de optimización jerárquico y la definición probabilística del espacio de búsqueda permiten la recolección de datos internos. El análisis de las probabilidades de las regiones de este espacio puede otorgar datos adicionales a los profesionales, que pueden llegar a conocer mejor sus

modelos y los problemas en estudio. A partir de analizar la distribución de probabilidades final asociada a un problema concreto, es posible identificar qué componentes de cada flujo de algoritmo influyeron en mayor grado en el rendimiento final obtenido. Esta información es más valiosa que solamente obtener el mejor flujo posible, ya que permite a los investigadores modificar en experimentos subsecuentes el espacio de búsqueda para concentrarse en las mejores combinaciones de algoritmos.

Una evaluación experimental en varios problemas diferentes muestra que este sistema puede obtener resultados competitivos con propuestas diseñadas por expertos, sin ninguna consideración de dominio o problema específico. Para demostrar esto se presenta una extensa experimentación que incluye 7 datasets clásicos de clasificación estructurada utilizados en el campo de AutoML y 3 escenarios adicionales en los que las herramientas de AutoML existentes no pueden utilizarse sin extensas modificaciones.

Una versión de AutoGOAL se encuentra disponible[1] para la comunidad científica. Esta contiene más de 100 adaptadores de algoritmos de distintas bibliotecas de aprendizaje automático o procesamiento de lenguaje natural y define 23 tipos semánticos de datos. Se encuentra disponible de IMT de código abierto y ha recibido 105 estrellas en github, 19 *forks*, 350 seguidores en Twitter y 429 descargas de la imagen de Docker.

10.1. Publicaciones

Durante el marco de esta investigación se ha colaborado en varias investigaciones que han contribuido de forma indirecta a este trabajo. Estás investigaciones han motivado el uso de herramientas de AutoML más poderosas para ser aplicadas en múltiples dominios de Procesamiento de Lenguaje Natural. A continuación se listan aquellas publicaciones en las que ha participado el autor:

- Overview of TASS 2018: Opinions, health and emotions [133]
- Overview of the eHealth Knowledge Discovery Challenge at IberLEF 2019 [134]

[1] https://autogoal.github.io

- Overview of the eHealth Knowledge Discovery Challenge at IberLEF 2020 [134]
- Gathering object interactions as semantic knowledge [135]
- A general-purpose annotation model for knowledge discovery: Case study in Spanish clinical text [136]
- A Neural Network Component for Knowledge-Based Semantic Representations of Text [137]
- Analysis of eHealth knowledge discovery systems in the TASS 2018 workshop [138]
- Demo Application for AutoGOAL framework [139]
- Overlapping Community Detection on a Graph of Chemicals, Diseases and Genes for Drug Repositioning and Adverse Reactions Prediction [140]
- Demo Application for LETO: Learning Engine Through Ontologies [141]

Adicionalmente, se listan a continuación las ediciones del evento *eHealth Knowledge Discovery* en las que se ha participado como organizador, y que han servido de escenarios de evaluación para AutoGOAL:

- eHealth KD 2018 como Workshop en el evento TASS 2018
  `http://www.sepln.org/workshops/tass/2018/task-3`
- eHealth KD 2019 como Workshop en el evento IBERLEF 2019
  `https://knowledge-learning.github.io/ehealthkd-2019`
- eHealth KD 2020 como Workshop en el evento IBERLEF 2020
  `https://knowledge-learning.github.io/ehealthkd-2020`

Capítulo 11

Trabajo Futuro

El paradigma de AutoML presentado en esta Tesis es utilizable en problemas prácticos a través de su implementación computacional (AutoGOAL). Sin embargo, aún se encuentra en una etapa de desarrollo inicial, por lo que es necesario continuar mejorando sus capacidades y promover su uso entre la comunidad del aprendizaje automático. Una vía para generalizar su uso es presentar la herramienta en eventos y competiciones internacionales, ya sea en el dominio propio de AutoML, o en problemas más específicos de aprendizaje automático. Como se pudo constatar en los resultados experimentales, AutoGOAL no solo es competitivo con otros sistemas AutoML, sino que brinda resultados similares a las mejores soluciones propuestas por equipos de investigadores tras meses de trabajo. Con vistas a desplegar AutoGOAL en problemas de aprendizaje más variados será necesario ampliar el conjunto de componentes y tipos semánticos para cubrir otras áreas como el procesamiento de imágenes y secuencias así como la inclusión de recursos basados en conocimiento para dominios específicos.

Aunque AutoGOAL está diseñado para usuarios técnicos con diferentes niveles de experticia, su uso sigue requiriendo conocimientos de programación. Para llevar el AutoML a usuarios no técnicos, es necesario diseñar aplicaciones con interfaces gráficas que permitan de forma interactiva introducir un conjunto de datos y obtener no solo resultados numéricos sino gráficas, diagramas y explicaciones intuitivas. En este sentido, el paradigma propuesta pudiera ser utilizado como el motor subyacente en el diseño de sistemas más

orientados a la industria. Uno de los retos más actuales en el área de AutoML es reducir el costo computacional que implica entrenar múltiples flujos de algoritmos de aprendizaje. En términos de rendimiento, el costo de ejecutar AutoGOAL está dominado por el costo de entrenamiento y evaluación de cada flujo individual. Dado que los algoritmos de aprendizaje automático a menudo son computacionalmente costosos, el costo adicional de actualizar el modelo probabilístico y el procesamiento asociado a la optimización en AutoGOAL son insignificantes. Por lo tanto, para reducir el costo computacional es necesario reducir la cantidad de flujos a evaluar y/o evaluar cada flujo con mayor eficiencia.

Dos enfoques iniciales con este fin consisten en el uso de estrategias distribuidas y paralelas para evaluar múltiples flujos simultáneamente, y el monitoreo de métricas de convergencia del modelo probabilístico subyacente para diseñar un mecanismo de detención temprana. Sin embargo, es posible que un mejor enfoque consista en una estrategia de meta-aprendizaje que utilice la experiencia previa para sesgar inicialmente la búsqueda hacia los flujos de algoritmos que anteriormente han dado mejores resultados. El paradigma propuesto en esta Tesis es compatible con estas estrategias a partir de inicial el modelo probabilístico en una región del espacio de búsqueda relacionada con el problema a resolver. Este tipo de estrategias puede disminuir aún más el costo de aplicar AutoML, al relacionar el nuevo conjunto de datos con los mejores flujos obtenidos en problemas similares previamente resueltos. Dichas estrategias aprovecharían cada ejecución anterior, potencialmente incluso por parte de otros investigadores, para resolver de manera más eficiente los problemas futuros.

Bibliografía

[1] Seyed Mahdi Shavarani, Mazyar Ghadiri Nejad, Farhood Rismanchian, and Gokhan Izbirak. Application of hierarchical facility location problem for optimization of a drone delivery system: a case study of amazon prime air in the city of san francisco. *The International Journal of Advanced Manufacturing Technology*, 95(9-12):3141–3153, 2018.

[2] Sanjay L Ahire, Manoj K Malhotra, and John B Jensen. Carton-mix optimization for walmart.com distribution centers. *Interfaces*, 45(4): 341–357, 2015.

[3] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, 2007.

[4] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[5] Lenny Koh, Guido Orzes, and Fu Jeff Jia. The fourth industrial revolution (industry 4.0): technologies disruption on operations and supply chain management. *International Journal of Operations & Production Management*, 2019.

[6] Rabab Benotsmane, György Kovács, and László Dudás. Economic, social impacts and operation of smart factories in industry 4.0 focusing on simulation and artificial intelligence of collaborating robots. *Social Sciences*, 8(5), 2019. ISSN 2076-0760. doi: 10.3390/socsci8050143. URL https://www.mdpi.com/2076-0760/8/5/143.

[7] Tom M Mitchell et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877, 1997.

[8] Yuval Heffetz, Roman Vainshtein, Gilad Katz, and Lior Rokach. Deepline: Automl tool for pipelines generation using deep reinforcement learning and hierarchical actions filtering. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2103–2113, 2020.

[9] F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018.

[10] Ekaba Bisong. Google automl: Cloud vision. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 581–598. Springer, 2019.

[11] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 731–737, 2020.

[12] Roger Barga, Valentine Fontama, Wee Hyong Tok, and Luis Cabrera-Cordon. *Predictive analytics with Microsoft Azure machine learning*. Springer, 2015.

[13] M. Feurer, A. Klein, K. Eggensperger, J.T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Adv. Neural Inf. Process. Syst.*, volume 2015-Janua, pages 2962–2970, 2015.

[14] C. Thornton, F. Hutter, H.H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, volume Part F1288, pages 847–855. ACM, 2013. ISBN 9781450321747. doi: 10.1145/2487575.2487629.

[15] H. Jin, Q. Song, and X. Hu. Auto-Keras: Efficient Neural Architecture Search with Network Morphism, 2018. ISSN 00086363 (ISSN).

[16] A.G.C. de Sá, W.J.G.S. Pinto, L.O.V.B. Oliveira, and G.L. Pappa. RECIPE: A grammar-based framework for automatically evolving classification pipelines. In *Lect. Notes Comput. Sci. (including Subser.*

Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), volume 10196 LNCS, pages 246–261. Springer, 2017. ISBN 9783319556956. doi: 10.1007/978-3-319-55696-3_16.

[17] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn. In *Proc. 13th Python Sci. Conf.*, pages 32–37. Citeseer, 2014. doi: 10.25080/majora-14bd3278-006.

[18] R.S. Olson and J.H. Moore. TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In *Work. Autom. Mach. Learn.*, pages 151–160, 2019. doi: 10.1007/978-3-030-05318-5_8.

[19] F. Mohr, M. Wever, and E. Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. *Mach. Learn.*, 107(8-10):1495–1515, sep 2018. ISSN 15730565. doi: 10.1007/s10994-018-5735-z. URL https://doi.org/10.1007/s10994-018-5735-z.

[20] Zhengying Liu, Olivier Bousquet, André Elisseeff, Sergio Escalera, Isabelle Guyon, Julio Jacques, Adrien Pavao, Danny Silver, Lisheng Sun-Hosoya, Sebastien Treguer, et al. Autodl challenge design and beta tests-towards automatic deep learning. 2018.

[21] B. Zoph, V. Vasudevan, J. Shlens, and Q.V. Le. Learning Transferable Architectures for Scalable Image Recognition. In *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pages 8697–8710, 2018. ISBN 9781538664209. doi: 10.1109/CVPR.2018.00907.

[22] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[23] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[24] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and

accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[25] Matthias Feurer and Frank Hutter. *Hyperparameter Optimization*, pages 3–33. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5_1. URL https://doi.org/10.1007/978-3-030-05318-5_1.

[26] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. URL http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[27] Gerda Claeskens and Nils Lid Hjort. *Model Selection and Model Averaging*. Number 9780521852258 in Cambridge Books. Cambridge University Press, June 2008. URL https://ideas.repec.org/b/cup/cbooks/9780521852258.html.

[28] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[29] Ingo Steinwart and Andreas Christmann. *Support vector machines*. Springer Science & Business Media, 2008.

[30] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[31] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[32] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[33] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[34] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

[35] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1 (1):1–23, 1993.

[36] K. Khan and A. Sahai. A Comparison of BA, GA, PSO, BP and LM for Training Feed forward Neural Networks in e-Learning Context. *Int. J. Intell. Syst. Appl.*, 4(7):23–29, 2012. ISSN 2074904X. doi: 10.5815/ijisa.2012.07.03.

[37] P. Kerschke and H. Trautmann. Automated algorithm selection on continuous black-box problems by combining exploratory landscape analysis and machine learning. *Evol. Comput.*, 27(1):99–127, 2018. ISSN 15309304. doi: 10.1162/evco_a_00236. URL https://doi.org/10.1162/evco_a_00236.

[38] Z. Li, X. Xiong, Z. Ren, N. Zhang, X. Wang, and T. Yang. An Aggressive Genetic Programming Approach for Searching Neural Network Structure Under Computational Constraints. *arXiv Prepr. arXiv1806.00851*, 2018. URL http://arxiv.org/abs/1806.00851.

[39] E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, J. Tan, Q.V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *34th Int. Conf. Mach. Learn. ICML 2017*, volume 6, pages 4429–4446, 2017. ISBN 9781510855144.

[40] A.M. De Silva and P.H.W. Leong. Grammatical evolution. *SpringerBriefs Appl. Sci. Technol.*, 5(9789812874108):25–33, 2015. ISSN 21915318. doi: 10.1007/978-981-287-411-5_3.

[41] A.R. Carvalho, F.M. Ramos, and A.A. Chaves. Metaheuristics for the feedforward artificial neural network (ANN) architecture optimization problem. *Neural Comput. Appl.*, 20(8):1273–1284, 2011. ISSN 09410643. doi: 10.1007/s00521-010-0504-3.

[42] I. Tsoulos, D. Gavrilis, and E. Glavas. Neural network construction and training using grammatical evolution. *Neurocomputing*, 72(1-3): 269–277, 2008. ISSN 09252312. doi: 10.1016/j.neucom.2008.01.017.

[43] K.O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, jun 2002. ISSN 10636560. doi: 10.1162/106365602320169811. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.5457&rep=rep1&type=pdfhttp://www.mitpressjournals.org/doi/10.1162/106365602320169811.

[44] F. Assuncao, N. Lourenco, P. Machado, and B. Ribeiro. Automatic generation of neural networks with structured Grammatical Evolution. In *2017 IEEE Congr. Evol. Comput. CEC 2017 - Proc.*, pages 1557–1564. IEEE, 2017. ISBN 9781509046010. doi: 10.1109/CEC.2017.7969488.

[45] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[46] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[47] D.P. Kingma and J.L. Ba. Adam: A method for stochastic optimization. *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, 2015.

[48] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*, pages 151–160. Springer, 2019.

[49] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming*. Springer, 1998.

[50] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[51] Robert I Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. Grammar-based genetic programming: a

survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[52] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in artificial intelligence*, pages 367–377. PMLR, 2020.

[53] Erin LeDell and Sebastien Poirier. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, volume 2020, 2020.

[54] Matthew Tovbin. Meet transmogrifai, open source automl that powers einstein predictions. sf big analytics meetup, 2020.

[55] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3), 2012.

[56] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *Thirty-Second AAAI conference on artificial intelligence*, 2018.

[57] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[58] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International Conference on Machine Learning*, pages 564–572, 2016.

[59] Gerd Brewka. Artificial intelligence—a modern approach by stuart russell and peter norvig, prentice hall. series in artificial intelligence, englewood cliffs, nj. *The Knowledge Engineering Review*, 11(1):78–79, 1996.

[60] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for cnns. In *Workshop on Meta-Learning at NIPS*, 2017.

[61] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte

carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[62] M. J. Bayarri and J. O. Berger. The Interplay of Bayesian and Frequentist Analysis. *Statistical Science*, 19(1):58 – 80, 2004. doi: 10.1214/088342304000000116. URL https://doi.org/10.1214/088342304000000116.

[63] Peter E Hart, David G Stork, and Richard O Duda. *Pattern classification*. Wiley Hoboken, 2000.

[64] F. Caraffini, F. Neri, and M. Epitropakis. HyperSPAM: A study on hyper-heuristic coordination strategies in the continuous domain. *Inf. Sci. (Ny).*, 477:186–202, 2019. ISSN 00200255. doi: 10.1016/j.ins.2018.10.033. URL http://www.sciencedirect.com/science/article/pii/S002002551830851X.

[65] E.K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu. Hyper-heuristics: A survey of the state of the art. *J. Oper. Res. Soc.*, 64(12):1695–1724, 2013. ISSN 14769360. doi: 10.1057/jors.2013.71. URL https://doi.org/10.1057/jors.2013.71.

[66] Lucas Zimmer, Marius Lindauer, and Frank Hutter. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[67] Felix Wortmann and Kristina Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221–224, 2015.

[68] Omer Sagi and Lior Rokach. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8 (4):e1249, 2018.

[69] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[70] Donald W Marquardt and Ronald D Snee. Ridge regression in practice. *The American Statistician*, 29(1):3–20, 1975.

[71] Andy Liaw, Matthew Wiener, et al. Classification and regression by random forest. *R news*, 2(3):18–22, 2002.

[72] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[73] J. Bergstra, R. Bardenet, Yoshua Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

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

[75] Peter McCullagh and John A Nelder. *Generalized linear models*. Routledge, 2019.

[76] Olivier Ledoit and Michael Wolf. Honey, i shrunk the sample covariance matrix. *The Journal of Portfolio Management*, 30(4):110–119, 2004.

[77] Evelyn Fix and Joseph Lawson Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique*, 57(3):238–247, 1989.

[78] Harry Zhang. The optimality of naive bayes. *AA*, 1(2):3, 2004.

[79] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[80] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *NIPS*, 2007.

[81] Sebastian Ruder, Ivan Vulic, and Anders Søgaard. A survey of cross-lingual word embedding models. *J. Artif. Intell. Res.*, 65:569–631, 2019.

[82] J. Vanschoren, J. N. Rijn, B. Bischl, and L. Torgo. Openml: networked science in machine learning. *SIGKDD Explor.*, 15:49–60, 2014.

[83] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[84] F. Pampel. Logistic regression: A primer. 2000.

[85] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. 2010.

[86] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13 (1):2171–2175, 2012.

[87] K. Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. Evol. Comput.*, 6:182–197, 2002.

[88] E.G. Talbi. *Metaheuristics: From Design to Implementation*, volume 74. John Wiley & Sons, 2009. ISBN 9780470278581. doi: 10.1002/9780470496916.

[89] M. Lichman. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]., 2013. URL `http://archive.ics.uci.edu/ml`.

[90] Uma N. Dulhare, Azmath Mubeen, and K. Ahmad. Hands-on h2o machine learning tool. 2020.

[91] Yoshua Bengio, Y. LeCun, and Geoffrey E. Hinton. Deep learning for ai. *Communications of the ACM*, 64:58 – 65, 2021.

[92] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63:3–42, 2006.

[93] MJ Van der Laan, EC Polley, and AE Hubbard. Super learner. statistical applications in genetics and molecular biology. *Super learner. Statistical applications in genetics and molecular biology*, 6(1), 2007.

[94] E. Ziegel, C. Mooney, and R. Duvall. Bootstrapping: A nonparametric approach to statistical inference. 1993.

[95] I. Guyon, Lisheng Sun-Hosoya, M. Boullé, H. Escalante, S. Escalera, Zhengying Liu, Damir Jajetic, Bisakha Ray, M. Saeed, M. Sebag,

A. Statnikov, Wei-Wei Tu, and E. Viegas. Analysis of the automl challenge series 2015-2018. In *Automated Machine Learning*, 2019.

[96] Ravi K. Samala, H. Chan, L. Hadjiiski, and Sathvik Koneru. Hazards of data leakage in machine learning: a study on classification of breast cancer using deep neural networks. In *Medical Imaging*, 2020.

[97] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*, pages 13–20. Citeseer, 2013.

[98] Kristina Chodorow and Michael Dirolf. Mongodb: The definitive guide. 2010.

[99] I. T. Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374, 2016.

[100] T. Roelleke and J. Wang. Tf-idf uncovered: a study of theories and probabilities. In *SIGIR '08*, 2008.

[101] K. Erol, J. Hendler, and D. Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, 1994.

[102] M. Ghallab, D. Nau, and P. Traverso. Automated planning - theory and practice. 2004.

[103] N. Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, 2017.

[104] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1–43, 2012.

[105] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte

carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[106] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet S. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[107] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[108] Abien Fred Agarap. Deep learning using rectified linear units (relu). *ArXiv*, abs/1803.08375, 2018.

[109] Nitish Srivastava, Geoffrey E. Hinton, A. Krizhevsky, Ilya Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15:1929–1958, 2014.

[110] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[111] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[112] S. Falkner, Aaron Klein, and F. Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *ICML*, 2018.

[113] T. Elsken, J. H. Metzen, and F. Hutter. Simple and efficient architecture search for convolutional neural networks. *ArXiv*, abs/1711.04528, 2018.

[114] R. Sutton and A. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 16:285–286, 2005.

[115] Bowen Baker, Otkrist Gupta, Nikhil Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *ArXiv*, abs/1611.02167, 2017.

[116] A. Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012.

[117] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45:2673–2681, 1997.

[118] Sebastian Ruder. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016.

[119] Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, T. Schaul, and N. D. Freitas. Learning to learn by gradient descent by gradient descent. In *NIPS*, 2016.

[120] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22:1345–1359, 2010.

[121] Martín Abadi, Andy Chu, I. Goodfellow, H. B. McMahan, Ilya Mironov, Kunal Talwar, and L. Zhang. Deep learning with differential privacy. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[122] Martín Abadi, P. Barham, Jianmin Chen, Z. Chen, Andy Davis, J. Dean, M. Devin, S. Ghemawat, Geoffrey Irving, M. Isard, M. Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, D. Murray, Benoit Steiner, P. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqian Zhang. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[123] H.T. Kim and C.W. Ahn. A New Grammatical Evolution Based on Probabilistic Context-free Grammar. In *Proc. 18th Asia Pacific Symp. Intell. Evol. Syst. 2*, pages 1–12. Springer, 2015. doi: 10.1007/978-3-319-13356-0_1.

[124] François Chollet et al. Keras. `https://keras.io`, 2015.

[125] Edward Loper and Steven Bird. NLTK: the natural language toolkit. *arXiv preprint cs/0205028*, 2002.

[126] Keyvan Khosrovian, Dietmar Pfahl, and Vahid Garousi. Gensim 2.0: a customizable process simulation model for software process evaluation. In *International conference on software process*, pages 294–306. Springer, 2008.

[127] Luis Chiruzzo, S Castro, Mathias Etcheverry, Diego Garat, Juan José Prada, and Aiala Rosá. Overview of haha at iberlef 2019: Humor analysis based on human annotation. In *Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2019). CEUR Workshop Proceedings, CEUR-WS, Bilbao, Spain (9 2019)*, 2019.

[128] Alicia Lara-Clares and Ana Garcia-Serrano. Key phrases annotation in medical documents: Meddocan 2019 anonymization task. 2019.

[129] R.M. Rivera-Zavala, P. Martínez, and I. Segura-Bedmar. A Hybrid Bi-LSTM-CRF model for knowledge recognition from ehealth documents. In *CEUR Workshop Proc.*, volume 2172, pages 65–70, 2018.

[130] P. López-Úbeda, M.C. Díaz-Galiano, M.T. Martín-Valdivia, and L.A. Ureña-López. SINAI in TASS 2018 Task 3. Classifying actions and concepts with UMLS on MedLine. In *CEUR Workshop Proc.*, volume 2172, pages 77–82, 2018.

[131] J.V. Palatresi and H.R. Hontoria. TASS2018: Medical knowledge discovery by combining terminology extraction techniques with machine learning classification. In *CEUR Workshop Proc.*, volume 2172, pages 89–95. CEUR-WS. org, 2018.

[132] V. Suárez-Paniagua, I. Segura-Bedmar, and P. Martínez. LABDA at TASS-2018 Task 3: Convolutional neural networks for relation classification in Spanish eHealth documents. In *CEUR Workshop Proc.*, volume 2172, pages 71–76, 2018.

[133] E. Martínez-Cámara, Y. Almeida-Cruz, M.C. Díaz-Galiano, S. Estévez-Velarde, M. García-Cumbreras, M. García-Vega, Y. Gutiérrez, A. Montejo-Ráez, A. Montoyo, R. Muñoz, A. Piad-Morffis, and J. Villena-Román. Overview of TASS 2018: Opinions, health and

emotions. In *CEUR Workshop Proc.*, volume 2172 of *CEUR Workshop Proceedings*, pages 13–27, Sevilla, Spain, 2018. CEUR-WS.

[134] Alejandro Piad-Morffis, Yoan Gutiérrez, Juan Pablo Consuegra-Ayala, Suilan Estevez-Velarde, Yudivián Almeida-Cruz, Rafael Munoz, and Andrés Montoyo. Overview of the ehealth knowledge discovery challenge at iberlef 2019. In *Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2019). CEUR Workshop Proceedings, CEUR-WS. org*, 2019.

[135] Suilan Estevez-Velarde, Yoan Gutierrez, Andres Montoyo, Alejandro Piad-Morffis, Rafael Munoz, and Yudivian Almeida-Cruz. Gathering object interactions as semantic knowledge. In *International Conference on Artificial Intelligence*, 2018.

[136] Alejandro Piad-Morffis, Yoan Guitérrez, Suilan Estevez-Velarde, and Rafael Muñoz. A general-purpose annotation model for knowledge discovery: Case study in spanish clinical text. In *Proceedings of the 2nd Clinical Natural Language Processing Workshop*, pages 79–88, 2019.

[137] Alejandro Piad-Morffis, Rafael Muñoz, Yoan Gutiérrez, Yudivian Almeida-Cruz, Suilan Estevez-Velarde, and Andrés Montoyo. A neural network component for knowledge-based semantic representations of text. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019)*, pages 904–911, 2019.

[138] Alejandro Piad-Morffis, Yoan Gutiérrez, Suilan Estévez-Velarde, Yudivián Almeida-Cruz, Andrés Montoyo, and Rafael Muñoz. Analysis of ehealth knowledge discovery systems in the tass 2018 workshop. *Procesamiento del Lenguaje Natural*, 62(0):13–20, 2019.

[139] Suilan Estévez-Velarde, Alejandro Piad-Morffis, Yoan Gutiérrez, Andres Montoyo, Rafael Muñoz, and Yudivian Almeida-Cruz. Demo application for the autogoal framework, 2020-12.

[140] María Elena García-Ochagavía, Yudivián Almeida-Cruz, Suilán Estévez-Velarde, Aimée Alonso-Reina, and María Elena Ochagavía-Roque. Overlapping community detection on a graph of chemicals, diseases and genes for drug repositioning and adverse reactions prediction.

GECONTEC: Revista Internacional de Gestión del Conocimiento y la Tecnología, 7(2):80–96, 2019.

[141] Suilan Estevez-Velarde, Andrés Montoyo, Yudivián Almeida-Cruz, Yoan Gutiérrez, Alejandro Piad-Morffis, and Rafael Muñoz. Demo application for LETO: learning engine through ontologies. In Ruslan Mitkov and Galia Angelova, editors, *Proceedings of the International Conference on Recent Advances in Natural Language Processing, RANLP 2019, Varna, Bulgaria, September 2-4, 2019*, pages 276–284. INCOMA Ltd., 2019. doi: 10.26615/978-954-452-056-4_032. URL https://doi.org/10.26615/978-954-452-056-4_032.