UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Aprendizaje computacional para la generación automática de programas

Mauro Picó

Marccio Silva

Programa de Grado en Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Diciembre de 2018

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Aprendizaje computacional para la generación automática de programas

Mauro Picó

Marccio Silva

Proyecto de Grado de Ingeniería en Computación presentada al Programa de Grado en Ingeniería en Computación, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Ingeniero en Computación

Directores:

Sergio Nesmachnow

Renzo Massobrio

Montevideo – Uruguay

Diciembre de 2018

Silva, Marccio
Picó, Mauro

Aprendizaje computacional para la generación automática de programas / Mauro Picó y Marccio Silva. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2018.

VII, 60 p. 29, 7cm.

Directores:

Sergio Nesmachnow

Renzo Massobrio

Proyecto de Grado de Ingeniería en Computación – Universidad de la República, Programa en Ingeniería en Computación, 2018.

Referencias bibliográficas: p. 58 – 60.

I. Nesmachnow, Sergio , Massobrio, Renzo. II. Universidad de la República, Programa de Grado en Ingeniería en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE PROYECTO DE GRADO

Luis Chiruzzo

Silvana Moreno

Néstor Rocchetti

Montevideo – Uruguay
Diciembre de 2018

RESUMEN

Este trabajo estudia el comportamiento del paradigma Savant Virtual el cual, mediante la aplicación de métodos de aprendizaje computacional o automático, permite resolver problemas de optimización. Savant Virtual aprende de los algoritmos que tradicionalmente se utilizan para resolver el problema que se desea abordar. Este proyecto de grado presenta un estudio comparativo entre máquinas de soporte vectorial (SVM) y redes neuronales como métodos de aprendizaje automático asociados al paradigma Savant Virtual. Con este propósito se implementan tres clasificadores basados en redes neuronales, variando las funciones de activación, y un clasificador SVM. El problema de optimización abordado es el *Heterogeneous Computing Scheduling Problem*, un clásico problema de planificación que consiste en encontrar una asignación de tareas a recursos de cómputo que maximice cierta métrica de calidad de servicio. Los clasificadores se entrenan con 100 instancias del problema de 512 tareas y 16 máquinas, lo que se traduce en 51200 instancias de entrenamiento. La evaluación experimental se realiza sobre instancias del problema en un rango de dimensiones que va desde 17 tareas y 16 máquinas hasta 1024 tareas y 16 máquinas, con el fin de analizar la escalabilidad del paradigma propuesto. Se utiliza el *makespan* como métrica de calidad para evaluar las soluciones halladas con los distintos clasificadores y también se analiza la precisión en la clasificación. Los resultados experimentales muestran que, para determinadas configuraciones de las redes neuronales, el *makespan* mejora con respecto a las soluciones calculadas por la SVM. De igual forma, se constatan mejoras en las redes neuronales sobre SVM al comparar los resultados alcanzados en términos de la precisión de las predicciones.

Tabla de contenidos

Capítulo 1

Introducción

Savant Virtual (SV) es un paradigma que mediante la aplicación de técnicas de aprendizaje automático aprende el comportamiento de un algoritmo conocido que resuelve un problema de alto costo computacional. SV genera automáticamente un nuevo programa que resuelve nuevas instancias del problema [1]. Así también, SV hace un uso más eficiente de los recursos computacionales al posibilitar la paralelización de programas originalmente no paralelos.

Es de interés encontrar nuevos paradigmas de resolución de problemas NP-difíciles que hagan uso de nuevas técnicas y algoritmos que minimicen los tiempos de búsqueda en los espacios de soluciones asociados. En este sentido, el enfoque de SV tiene como objetivo obtener soluciones aproximadas a problemas complejos con mayor velocidad de resolución que las heurísticas existentes, manteniendo niveles altos de calidad en las soluciones.

En este proyecto se pretende evaluar el desempeño y la aplicabilidad de las redes neuronales como clasificadores de aprendizaje automático, presentando una alternativa a la utilización de máquinas de soporte vectorial (SVM), contribuyendo a la investigación que dio origen a este paradigma. Con este objetivo, se estudió el rendimiento de las redes neuronales en el marco del estudio del problema *Heterogeneous Computing Scheduling Problem* (HCSP), un problema de optimización combinatoria NP-difícil. Se llevó adelante un estudio comparativo entre redes neuronales y SVM entrenando diferentes configuraciones de redes neuronales, y se compararon las soluciones obtenidas con las soluciones generadas por SVM.

Se encontraron mejoras en las medidas de desempeño seleccionadas para grandes dimensiones del problema, como se puede ver en el Capítulo 6.

Las principales contribuciones de este proyecto de grado son:

1. Extender el trabajo original de SV presentado por Dorronsoro et al. [1].
2. Analizar el comportamiento de SV cuando se utilizan redes neuronales como método de aprendizaje computacional.
3. Comparar el rendimiento de las redes neuronales y SVM al resolver el problema HCSP, usando el *makespan* como medida de la calidad de los resultados.
4. Explorar las configuraciones de redes neuronales más adecuadas para resolver el problema HCSP.

El resto del trabajo se estructura como se explica a continuación. En el Capítulo 2 se hace una presentación del problema HCSP y la heurística de referencia. El Capítulo 3 presenta el marco teórico en el cual se basó este trabajo, enfocado en aprendizaje automático y en los dos algoritmos comparados durante el trabajo. El Capítulo 4 aborda los trabajos relacionados, donde se presentan aquellos trabajos asociados a SV y a técnicas de clasificación y selección de atributos que sirvieron de referencia o punto de partida para este trabajo. El Capítulo 5 presenta la implementación del sistema utilizado para llevar a cabo la experimentación asociada a este trabajo, dejando para el Capítulo 6 el análisis experimental. Por último, el Capítulo 7 presenta las conclusiones y trabajo futuro.

Capítulo 2

Descripción del problema

Este capítulo presenta una introducción al problema HCSP basada fuertemente en el trabajo de Nesmachnow [2]. Este problema es abordado en el marco de este proyecto bajo el paradigma de Savant Virtual. En la Sección 2.1 se introduce al problema y su contexto. En la Sección 2.2 se presenta la formulación del problema. En la Sección 2.3 se analiza la complejidad computacional del HCSP. En la Sección 2.4 se discute la estimación de tiempos de ejecución utilizada en las instancias del problema. En la Sección 2.5 se analiza la estructura de las instancias del problema utilizadas en este proyecto de grado, y finalmente en la Sección 2.6 se presenta la heurística de referencia utilizada a la hora de llevar a cabo el aprendizaje automático.

2.1. Introducción

En un contexto donde el poder de cómputo se ve incrementado constantemente y la comercialización de computadores de bajo costo es común, se vuelve también común el uso de ambientes computacionales distribuidos basados en componentes heterogéneos para la resolución de problemas complejos y con diversas aplicaciones. Esto incluye a aquellos problemas que por escala o limitaciones de recursos no eran resolubles en el pasado y a aquellos problemas surgidos por la utilización de dichos sistemas heterogéneos y distribuidos.

Uno de los problemas fundamentales que surgen al emplear computación heterogénea (o HC, del inglés *Heterogeneous Computing*) es el de encontrar una manera de planificar un conjunto de tareas a ser ejecutadas tomando en cuenta las características y poder computacional de cada uno de los elementos que conforman al sistema. El objetivo de este problema de planificación, conocido como HCSP, es el de asignar tareas de manera óptima a recursos computacionales optimizando alguna métrica de eficiencia. Usualmente se tiende a utilizar estrategias que optimicen el *makespan*, una métrica que contempla el tiempo total desde el inicio de la ejecución de la primer tarea hasta el fin de la ejecución de la última tarea.

Los problemas de planificación en sistemas multiprocesador han sido ampliamente estudiados en el campo de la investigación operativa y numerosos métodos han sido propuestos para generar planificaciones precisas en tiempos razonables (El-Rewini et al. [3], Leung [4]). En su formulación clásica, los problemas de planificación asumen un entorno computacional compuesto por recursos homogéneos. Sin embargo, en la década de 1990, la comunidad de investigadores empezó a prestarle atención a problemas en entornos heterogéneos, dada la popularización de la computación distribuida y el uso cada vez más frecuente de clusters (Freund et al. [5], Eshaghian [6]).

Los problemas tradicionales de planificación son NP-difíciles (Garey y Johnson [7]), por lo cual los métodos clásicos y exactos sólo son útiles para resolver instancias reducidas, dado que su poca eficiencia hace que sea inviable aplicarlos para problemas de gran dimensión en tiempos de ejecución razonables. Al tratar con ambientes computacionales de grandes dimensiones, las heurísticas ad-hoc y metaheurísticas se han destacado como métodos prometedores para resolver problemas de HC. Aunque estos métodos no garantizan obtener una solución óptima para el problema, obtienen planificaciones cercanas a las óptimas que usualmente satisfacen los requerimientos de eficiencia para escenarios reales, en tiempos de ejecución razonables.

2.2. Formulación del problema

El contexto del problema se da en un sistema de HC compuesto por varios recursos computacionales o máquinas, para el cual se tiene un conjunto de tareas a ser ejecutadas con requerimientos computacionales variables. Se considera a una tarea como una unidad atómica de trabajo que puede ser asignada a un recurso computacional; no puede ser dividida en subtareas y su ejecución no puede ser interrumpida tras ser iniciada. El hecho de que las máquinas disponibles sean diferentes entre sí en términos de prestaciones, genera una suerte de competencia entre las tareas por ser asignadas a la máquina más apta y con mayor rendimiento. En este trabajo, la única característica que se toma en cuenta sobre una tarea es su tiempo de ejecución por cuestiones de simpleza, aunque claramente se están dejando de lado otras consideraciones que pueden ser cruciales para un sistema real, como puede ser el costo de utilizar un determinado equipo frente a otro. Puede suceder que el costo de utilizar un recurso computacional más eficiente que otro sea mayor, dada la demanda que puede llegar a existir sobre dicho recurso, como puede ocurrir en el caso de utilizar recursos computacionales alquilados bajo un formato de infraestructura como servicio (o *IaaS* del inglés *Infrastructure as a Service*). Todo esto se traduce en que una planificación óptima para la ejecución de tareas será aquella donde el tiempo que tarda la máquina que termina su ejecución por último es mínimo; este tiempo, como fue mencionado en la sección anterior, es conocido como *makespan*.

En términos formales, para una instancia del problema de dimensión $X \times Y$, se tiene un conjunto de tareas $T = \{t_1, t_2, \ldots, t_X\}$ y un conjunto de recursos computacionales o máquinas $M = \{m_1, m_2, \ldots, m_Y\}$. Dada una función de tiempo de ejecución $ET : T \times M \rightarrow R^+$ tal que $ET(t_i, m_j)$ es el tiempo requerido para ejecutar la tarea t_i en la máquina m_j, el objetivo que se persigue para resolver el problema HCSP es el de encontrar una asignación de tareas a máquinas, dada por una función $f : T \rightarrow M$, que minimice el makespan, definido como $max_{m_y \in M} \sum_{t_x \in T : f(t_x) = m_y} ET(t_x, m_y)$.

Finalmente, se puede expresar que el objetivo de la resolución del problema HCSP se traduce en encontrar aquella función que dados un conjunto de máquinas y un conjunto de tareas determine una planificación que minimice el valor del makespan obtenido.

Cabe destacar que este modelo no toma en cuenta eventuales dependencias entre las tareas, asumiendo que pueden ser ejecutadas de manera independiente. Además, en este trabajo se estudia la versión estática del HCSP. Esto quiere decir que las planificaciones tomadas en cuenta se generan de manera previa a la ejecución de un conjunto de tareas y no se adaptan o varían dinámicamente en tiempo de ejecución.

2.3. Complejidad computacional

El problema HCSP es NP-difícil. Su espacio de búsqueda de soluciones es de tamaño Y^X, teniendo X tareas e Y máquinas. Esto quiere decir que la cantidad de posibles planificaciones aumenta de manera exponencial de acuerdo a la cantidad de tareas manejadas. Incluso para instancias del problema muy pequeñas, el espacio de búsqueda del problema HCSP es muy grande, siendo por ejemplo de 1.048.576 posibles planificaciones para 4 máquinas y 10 tareas. Esto se traduce en que la complejidad computacional del problema hace que métodos clásicos exactos como programación dinámica o programación lineal sólo sean útiles para instancias reducidas del problema. Por todo esto, es de interés el estudio de heurísticas y metaheurísticas asociadas a este tipo de problema, ya que permiten obtener planificaciones que, si bien no resultan óptimas, pueden satisfacer los requerimientos de problemáticas reales en tiempos razonables.

2.4. Estimación de tiempos de ejecución

Si bien es posible evaluar la complejidad computacional de un programa mediante análisis estático del código, determinar de manera exacta su tiempo de ejecución en una pieza de hardware presenta diversas dificultades. Por ejemplo, el poder computacional de un procesador depende de componentes físicos de estado y características variables, afectados por múltiples factores físicos mientras desarrollan sus actividades. Esto hace que se vuelva una necesidad utilizar métodos para aproximar el tiempo de ejecución de un programa dado en relación a un recurso computacional específico.

En este contexto, el modelo de estimación de rendimiento ETC (del inglés *expected time to compute*) presentado en Ali et al. [8], ha sido utilizado ampliamente en la investigación asociada a la planificación de sistemas HC a la hora de generar una estimación para los tiempos de ejecución de las tareas. Este modelo toma en cuenta tres propiedades para generar estimaciones: la heterogeneidad de las máquinas, la heterogeneidad de las tareas y la consistencia del escenario.

La heterogeneidad de las máquinas está dada por la variación en los tiempos de ejecución obtenidos al ejecutar una tarea dada en todos los recursos del sistema. Si el sistema está compuesto por recursos computacionales similares, la heterogeneidad de las máquinas será baja. Por otra parte, si estamos frente a un sistema HC genérico se tendrá una heterogeneidad alta, dado que dichos sistemas generalmente están compuestos por recursos computacionales altamente variables.

La heterogeneidad de tareas está asociada a qué tanta variabilidad se tiene entre las tareas a ejecutar. Un conjunto de tareas es heterogéneo si se envían tareas variables al sistema; tareas grandes y complejas que utilizan una gran cantidad de tiempo de procesador, así como también tareas livianas. Si todas las tareas a ejecutar son de características similares, se está frente a un escenario con baja heterogeneidad de tareas.

Finalmente, un escenario es consistente si dado que una máquina m_y ejecuta una tarea t_x más rápido que una máquina m_z, entonces la máquina m_y ejecuta a todas las tareas más rápido que la máquina m_z, siendo más rápida de manera consistente. Este escenario únicamente aplica a aquellas tareas orientadas al uso de CPU o *CPU bound*, ya que en otros casos esta consistencia no estará presente. Por ejemplo, si tenemos que la máquina m_y tiene más poder computacional que la máquina m_z pero su velocidad de acceso a almacenamiento secundario o a una red de área local es menor, las tareas orientadas al uso de procesador ejecutarán más rápido en la máquina m_y. Por otra parte, si es necesario depender fuertemente de la entrada/salida, es probable que las tareas ejecuten más rápido en la máquina m_z. Todo esto constituye un escenario inconsistente. Un escenario será semi consistente cuando dentro de un conjunto de máquinas inconsistentes se pueden identificar subconjuntos de máquinas consistentes entre sí.

2.5. Instancias del problema

Ali et al. [8] propusieron dos métodos para diseñar matrices ETC (representación matricial que determina la estimación ETC de cada tarea para cada máquina) aleatorias para representar diversos escenarios del HCSP: el método basado en rangos (del inglés *range based*) y el método del coeficiente de variación (del inglés *coefficient of variation*). Estos métodos difieren en el modelo matemático empleado para definir los valores de heterogeneidad para tareas y máquinas.

El método basado en rangos define dos rangos, $(1, R_{maquina})$ para heterogeneidad de máquinas y $(1, R_{tarea})$ para heterogeneidad de tareas. Los valores de heterogeneidad para las máquinas (τ_M) y las tareas (τ_T) son muestreados aleatoriamente usando una distribución uniforme, y la estimación ETC para una tarea i en una máquina j es calculada como $ETC(i, j) = \tau_T(i) \times \tau_M(j)$. Ali et al. [8] sugirieron utilizar los valores de la primera fila de la Tabla 2.1 para generar escenarios del HCSP, mientras que Braun et al. [9] utilizaron los valores de la segunda fila de la Tabla 2.1 al generar un conjunto de escenarios aleatorios de HCSP para su trabajo.

Modelo	Heterogeneidad de tareas		Heterogeneidad de máquinas	
	Baja	**Alta**	**Baja**	**Alta**
Ali et al. [8]	$R_{tarea} = 10$	$R_{tarea} = 100000$	$R_{maquina} = 10$	$R_{maquina} = 1000$
Braun et al. [9]	$R_{tarea} = 100$	$R_{tarea} = 3000$	$R_{maquina} = 10$	$R_{maquina} = 1000$

Tabla 2.1: Parámetros de los modelos ETC

Por otra parte, el método del coeficiente de variación usa el cociente entre la desviación estándar y el promedio de los tiempos de ejecución como una medida de la heterogeneidad de las máquinas y de las tareas, mientras que los valores de ETC son generados aleatoriamente usando una distribución uniforme.

En este trabajo se utilizó un generador de instancias aleatorias del HCSP presentado en Nesmachnow [2], implementado en C y utilizando únicamente bibliotecas estándar. Este generador implementa el método basado en rangos mencionado anteriormente, recibiendo parámetros relevantes para el escenario de HCSP, como son la dimensión del problema (cantidad de tareas y máquinas involucradas), heterogeneidad de tareas y máquinas, consistencia y el modelo de heterogeneidad utilizado (contemplando tanto los rangos definidos por Ali et al. [8] como por Braun et al. [9]).

El formato de estas instancias del problema es el utilizado por Braun et al. [9], un vector columna de $X \times Y$ números en punto flotante (para X tareas e Y máquinas) que representa a la matriz ETC, ordenada por identificador de tarea. En la Figura 2.1 se muestra un ejemplo del formato de un escenario HCSP con X tareas e Y máquinas.

$ETC(t_1, m_1)$
$ETC(t_1, m_2)$
...
$ETC(t_1, m_Y)$
$ETC(t_2, m_1)$
$ETC(t_2, m_2)$
...
$ETC(t_X, m_1)$
$ETC(t_X, m_Y)$

Figura 2.1: Formato de escenario HCSP

2.6. Heurística de referencia

En el Algoritmo 1, como se puede ver en Nesmachnow [2], se presenta una versión genérica de una heurística generadora de planificaciones frente al problema HCSP. Particularmente, la heurística utilizada como referencia en este proyecto emplea al algoritmo *Min-Min* como criterio para seleccionar tareas en cada iteración del algoritmo. Según este criterio, se escoge de manera ávida a la tarea que pueda ser completada primero. Para seleccionar a esta tarea, se calcula el mínimo tiempo que cada una de las tareas sin asignar puede demorar en finalizar su ejecución para cada una de las máquinas disponibles, y se escoge a aquella tarea que en promedio lleve menos tiempo en completar.

Como se adelantó en el Capítulo 1, en este trabajo se evalúa el desempeño de las redes neuronales en el marco de SV. Min-Min fue la heurística de referencia escogida para entrenar a los clasificadores de aprendizaje automático utilizados en este proyecto de grado.

Algoritmo 1: Algoritmo genérico de planificación de tareas
Entrada: conjunto de tareas sin asignar y conjunto de máquinas
mientras *quedan tareas por asignar* **hacer**
    seleccionar tarea de acuerdo a criterio elegido
    para *cada tarea a asignar y cada máquina* **hacer**
        evaluar criterio (tarea, máquina)
    fin para
    asignar tarea seleccionada a máquina seleccionada
fin mientras
devolver *asignación de tareas*

Capítulo 3

Marco teórico

A continuación, se presentan los conceptos fundamentales relacionados al desarrollo y análisis experimental llevado a cabo en este proyecto de grado, involucrando principalmente conceptos de aprendizaje automático.

3.1. Aprendizaje automático

El aprendizaje automático es un campo de las ciencias de la computación y una rama de la inteligencia artificial dedicada a desarrollar técnicas que permitan construir programas que mejoran de forma automática basados en la experiencia. En términos formales, el concepto de que un programa aprenda es puesto en palabras por Mitchell [10] de la siguiente manera: *"Se dice que un programa de computadora aprende de la experiencia E con respecto a una tarea T y medida de desempeño P si su desempeño en la tarea T, medida en términos de P, mejora con la experiencia E"*.

Los algoritmos de aprendizaje automático pueden ser clasificados en algoritmos de aprendizaje supervisado y no supervisado, en función de la necesidad de la existencia de datos previos que oficien de experiencias. En el aprendizaje supervisado, las entradas del algoritmo son ejemplos de experiencias (conocidos como datos de entrenamiento) y sus respectivas salidas esperadas. De tal manera, el algoritmo aprende reglas generales de mapeo entre entradas y salidas. Ejemplos de algoritmos de aprendizaje supervisado son las redes neuronales con propagación hacia atrás (dado su empleo del algoritmo *Backpropagation*), así como las máquinas de soporte vectorial o los árboles de decisión.

En algoritmos no supervisados no se ofrecen las salidas esperadas en los datos de entrenamiento; los algoritmos detectan patrones en los datos y generan agrupaciones de los mismos en base a ello. Dadas estas agrupaciones, surge algo equivalente a una clasificación, calculada en base a la similitud que tenga un nuevo ejemplo con aquellos ejemplos de entrenamiento ya agrupados. El algoritmo *K-Means* es un ejemplo de algoritmo de aprendizaje automático no supervisado.

También es posible encontrar algoritmos de aprendizaje por refuerzo. Dentro del aprendizaje automático, el aprendizaje por refuerzo se ocupa de estudiar cómo agentes de software toman acciones en su entorno con el objetivo de maximizar una recompensa. Un ejemplo de este tipo de algoritmos es *Q-Learning*, que involucra el aprendizaje de una política que le indica al agente la decisión a tomar bajo ciertas circunstancias.

De manera adicional, los problemas estudiados en aprendizaje automático pueden ser de clasificación o de regresión. Un problema se considera de clasificación cuando se puede clasificar a las instancias del problema de acuerdo a valores de un dominio discreto, como en el ejemplo de reconocimiento de objetos en imágenes, donde la clasificación resulta binaria (el objeto se encuentra presente en la imagen o no). Un problema se considera de regresión cuando se puede clasificar a las instancias del problema de acuerdo a valores de un dominio continuo, por ejemplo, al intentar predecir el valor de un inmueble dado un conjunto de características asociadas al mismo.

Las técnicas basadas en aprendizaje automático han tomando mayor relevancia en los últimos años debido al crecimiento de los volúmenes y la variedad de datos disponibles, a la disminución de los costos de infraestructura computacional, acompañado por un crecimiento del poder de cómputo y de la capacidad de almacenamiento.

En este proyecto se aplican dos de los algoritmos de aprendizaje automático mencionados anteriormente a un problema de clasificación: redes neuronales y máquina de soporte vectorial.

3.2. Clasificadores utilizados

A continuación se presentan los conceptos teóricos fundamentales asociados a los clasificadores utilizados en este trabajo.

3.2.1. Redes neuronales artificiales

Una red neuronal artificial es un algoritmo de aprendizaje automático utilizado tanto para problemas de clasificación como de regresión. Su concepción está inspirada en la observación de los mecanismos de aprendizaje en sistemas biológicos. Las redes neuronales artificiales están formadas por unidades más simples e interconectadas. Cada unidad tiene entradas reales y calcula una única salida (también real) mediante la asignación de pesos en regresiones lineales. Estas unidades se denominan perceptrones. Un perceptrón recibe valores de n entradas $x_1, x_2, \ldots x_n$, realiza una combinación lineal con ellos, obteniendo una expresión $x_1w_1 + x_2w_2 + \cdots + x_nw_n$ donde w_i es el peso otorgado a x_i, para todo $i \in \{1, \ldots, n\}$. Esta expresión es evaluada con una función conocida como *función de activación*, que devuelve un valor de acuerdo a si la combinación lineal de los valores de entrada supera un umbral determinado o no. Por ejemplo, utilizando la función signo como función de activación, si la combinación lineal de los valores de entrada es mayor que cero, se devolverá 1 como salida y -1 en caso contrario. La Figura 3.1 muestra la estructura de un perceptrón con función de activación signo.

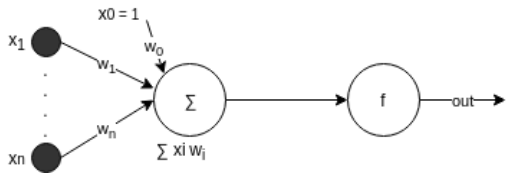

Figura 3.1: Perceptron con función de activación $f = signo(x)$

Cuando se calcula la combinación lineal de los valores de entrada, el valor resultante puede oscilar entre $-\infty$ y $+\infty$, dado que un perceptrón no posee una referencia de cuáles son los límites asociados a las posibles clases de clasificación para el problema de interés. Las funciones de activación se utilizan con el propósito de limitar este valor al producir la salida del perceptrón.

El perceptrón puede ser utilizado para modelar funciones lógicas, como AND, OR, NAND y NOR, ajustando la cantidad de entradas según la cantidad de entradas de la función lógica y ajustando los pesos de tal manera que las salidas sean 1 y -1. La capacidad de los perceptrones para representar funciones lógicas es relevante puesto que da lugar a que cualquier función lógica pueda ser representada mediante una red de perceptrones de dos niveles de profundidad, en la cual las entradas son conectadas a múltiples perceptrones y las salidas de estos son conectadas a la siguiente capa. De todas maneras, múltiples capas de perceptrones, dada su naturaleza de unidades lineales, producirán funciones lineales. Para expresar decisiones no lineales es necesario emplear neuronas cuyas salidas sean funciones no lineales de sus entradas. Una posibilidad es la utilización de unidades sigmoide, cuya estructura es similar a la del perceptrón, variando en la función de activación, que es la función *sigmoide*, una función no lineal diferenciable. La personalización de las neuronas artificiales, por lo tanto, genera la posibilidad de representar una gran variedad de funciones.

Por lo tanto, la forma de trabajar utilizando redes neuronales puede dividirse en dos fases. Primero, se debe encontrar una configuración arquitectónica de perceptrones o neuronas capaz de expresar la función que se está intentando aprender (que puede no ser conocida a priori y, por lo tanto, puede ser necesario evaluar diversas disposiciones en términos de cantidad de capas y cantidad de neuronas). Luego, se debe encontrar un vector de pesos $(w_0, w_1, \ldots, w_n)$ apropiado para cada perceptrón mediante el entrenamiento. En el contexto de un problema de clasificación, entrenar una red neuronal consiste en administrarle ejemplos ya clasificados del problema de interés. Para cada ejemplo administrado, la red primero producirá una clasificación para el problema dado y la comparará con la salida esperada, tras lo cual ajustará los pesos de cada una de sus unidades para reducir la eventual diferencia entre estos dos valores, utilizando un algoritmo definido previamente. En este trabajo, donde se utilizan redes multicapa, se utiliza el algoritmo *Backpropagation* para *aprender* los vectores de pesos de cada una de las unidades interconectadas que las componen.

Para describir el funcionamiento del algoritmo Backpropagation, y cómo este ajusta de manera iterativa los pesos de las unidades, es importante definir el concepto de *gradiente descendente*. Gradiente descendente es una técnica que busca encontrar un mínimo de una función, ya sea local o no. Para encontrar un mínimo local de una función usando gradiente descendente, se calcula el gradiente de la misma en un punto de partida y se lo hace variar en dirección negativa a dicho gradiente. De esta manera, se llegará a un mínimo local eventualmente. En el contexto de las redes neuronales, la búsqueda por gradiente descendente encuentra un vector de pesos que minimiza el error con respecto al hiperplano generado por los vectores de pesos asociados al conjunto de entrenamiento. La búsqueda comienza utilizando un vector de pesos inicial arbitrario, modificándolo en pequeños pasos, realizando cada modificación de tal manera que se produzca una disminución en el error con respecto al hiperplano de pesos asociados a la solución. Este proceso continúa hasta que se llega a un error global mínimo. Gradiente descendente es una estrategia de búsqueda en un espacio grande o infinito de hipótesis que se puede aplicar siempre que dicho espacio contenga hipótesis que puedan ser determinadas de forma paramétrica, como son los pesos en las unidades. Así también, se requiere que el error pueda ser diferenciado con respecto a estos parámetros. Esta estrategia tiene algunas dificultades fundamentales; una de ellas es que la velocidad de convergencia a un mínimo es lenta, pudiendo requerir varios miles de pasos para lograrla y además, si existen varios mínimos locales en la superficie de error, no garantiza que la búsqueda converja a un mínimo global.

El algoritmo Backpropagation es un método utilizado en redes multicapa para calcular el gradiente que se utiliza para el cálculo de los pesos de las neuronas, empleando gradiente descendente. El error es calculado en la salida de la red multicapa y es distribuido *hacia atrás*, hacia las capas internas de la red, calculando para cada unidad perteneciente a una capa intermedia su error y actualizando sus pesos. El bucle de actualización de pesos en Backpropagation puede iterar miles de veces en una aplicación típica de redes neuronales, por lo que existen una variedad de criterios de terminación, como detener el bucle luego de una cantidad fija de iteraciones, o detener el bucle luego de que el error alcanza cierto umbral definido, o por criterios definidos sobre un conjunto dado de prueba.

Los criterios de terminación son importantes y su elección es delicada dado que, de terminar antes de lo necesario con las iteraciones, la red neuronal puede devolver salidas con un error elevado. Por otra parte, si se realizan muchas iteraciones se puede generar un sobreajuste de la red neuronal al conjunto de entrenamiento, produciendo resultados de bajo error para el conjunto de entrenamiento, pero con más altos niveles de error para conjuntos de validación con datos no vistos durante el entrenamiento. Backpropagation no asegura la convergencia a un mínimo global debido al uso gradiente descendente en espacios de hipótesis de alta dimensionalidad (pudiendo haber tantas dimensiones como pesos). La alta dimensionalidad incrementa la probabilidad de que los movimientos en la superficie de error no permitan alcanzar un mínimo global, pudiendo alcanzar mínimos para una o más dimensiones, que no sean necesariamente mínimos para las otras dimensiones. A pesar de la falta de garantías con respecto a la convergencia del algoritmo a un mínimo global, Backpropagation es un método de aproximación de funciones altamente efectivo. Una característica que manifiestan las redes neuronales entrenadas con Backpropagation es la habilidad de descubrir relaciones no triviales presentes en los datos de entrada, a nivel de las capas intermedias ocultas de la red. Por ejemplo, dada una instancia de entrenamiento con N atributos, la utilización de Backpropagation puede conducir a que se descubran relaciones entre los atributos a_i y a_j, $\forall i, j \in \{1, \ldots, N\}, i \neq j$ no identificables por un humano a simple vista, y que contribuyen a la clasificación en gran medida.

En este trabajo se construyeron varios tipos de redes que varían en las funciones de activación utilizadas por las unidades. A continuación se presentan las funciones de activación que se usaron durante este trabajo. La función *tanh*, expresada de la siguiente manera: $f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$, es una función continua no lineal. Al componerla consigo misma se obtienen funciones no lineales, lo que permite combinar a unidades con esta función de activación sin perder la no linealidad. tanh es una función suave en su curva, mostrando que pequeñas variaciones en valores del dominio cercanos a 0 generan cambios grandes en los valores correspondientes del codominio. Esto implica que se le da una gran ponderación a los valores de los extremos del codominio, algo análogo a una tasa de aprendizaje.

La función *relu*, expresada de la siguiente manera: $f(x) = relu(x) = max(0, x)$ es no lineal y las composiciones de ella consigo misma son no lineales, pero por su forma, puede ocasionar que algunas neuronas den como resultado cero, constituyéndose una eventual pérdida de información. Este problema se llama *dying relu problem*. Sin embargo, computar relu es menos costoso que computar tanh porque implica operaciones matemáticas más simples.

Por último, la función *identity*, también llamada de activación lineal, expresada de la siguiente manera: $f(x) = identity(x) = x$, siempre retorna el mismo valor que recibe en su argumento, lo que implica que equivale a una regresión lineal utilizando los pesos de la unidad.

Entre las ventajas de utilizar redes neuronales se encuentra el hecho de que se adecuan correctamente a problemas en los cuales los datos de entrenamiento contienen ruido. Además, las redes neuronales suelen mantener tiempos bajos de clasificación. Por otra parte, como se menciona en la Sección 4.2, una desventaja de las redes neuronales es que su representación interna es difícil de entender para los humanos y, por este motivo, se dice que se comportan como una "caja negra".

3.2.2. SVM

Una máquina de soporte vectorial (SVM) es un algoritmo de aprendizaje automático supervisado utilizado fundamentalmente para problemas de clasificación. Dado un problema de aprendizaje automático supervisado de clasificación, donde las instancias del problema pueden ser clasificadas en N clases, y un conjunto de ejemplos de entrenamiento de dicho problema, una SVM busca encontrar un hiperplano que divida a estos ejemplos de entrenamiento de manera lineal en esas N clases. De esta manera, frente a una nueva instancia del problema, la SVM es capaz de clasificarla generando una correspondencia entre esta instancia y una de las N clases posibles.

Se denominan vectores de soporte a aquellos ejemplos de entrenamiento más cercanos al hiperplano. La distancia entre el hiperplano y un vector de soporte se conoce como margen y, para un conjunto de ejemplos de entrenamiento, SVM busca dividirlos de manera óptima con un hiperplano donde se maximicen estos márgenes para cada uno de los vectores de soporte, cada uno asociado a su vez a una de las posibles clases de clasificación del problema. En un problema donde las instancias pueden ser clasificadas en dos clases, el hiperplano de dos dimensiones es representado por una línea recta y los vectores de soporte son aquellos ejemplos de entrenamiento más cercanos a ella, desde las dos direcciones posibles.

La Figura 3.2 muestra el funcionamiento de una SVM en un problema de clasificación de dos clases.

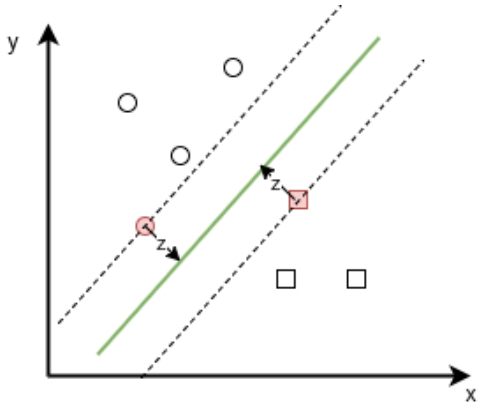

Figura 3.2: En esta figura se observan dos conjuntos de elementos (cuadrados y círculos), clasificados en dos clases, en base a dos características, x e y. El hiperplano generado por SVM se muestra como la recta verde y se marcan los márgenes en líneas punteadas. Así, en color rojo se muestran los vectores de soporte, uno perteneciente a cada clase, siendo z la distancia entre cada vector de soporte y el hiperplano.

El hiperplano está dado por la ecuación $g(\vec{x}) = \vec{w}^T\vec{x} + w_0$ donde $\vec{w}$ es un vector de pesos y $\vec{x}$ es el vector de características. La distancia a uno de los márgenes está dada por $z = |g(\vec{x})|/\|\vec{w}\| = 1/\|\vec{w}\|$, por lo que el margen total se computa como $1/\|\vec{w}\| + 1/\|\vec{w}\| = 2/\|\vec{w}\|$. Por lo tanto, minimizando la expresión $\|\vec{w}\|$ se maximizan los márgenes, maximizando la separabilidad de las clases. La confianza en la clasificación de una instancia de validación, por ejemplo, está dada por la distancia de dicha instancia al hiperplano de manera directamente proporcional.

Cuando se tienen más de dos clases objetivo para la clasificación, se utiliza una SVM multiclase en la cual se admiten dos posibles estrategias. Por un lado *One vs. Rest* (*OVR*) y por otro *One vs. One* (*OVO*). Dado un conjunto de N clases $\{c_1, c_2, \ldots, c_N\}$, *OVR* intentará clasificar c_i contra $\{c_1, c_2, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{N-1}\}$ dejando como representantes de la clase C_N los datos que no fueron clasificados en las otras clases. Por otro lado *OVO*, intentará clasificar cada clase i contra cada $C_j \forall j \neq i$, creando un hiperplano para cada una de las combinaciones. Comparando las dos estrategias, *OVR* genera C_{N-1} clasificadores, en contraste con *OVO* que genera una combinación de N tomada de a dos, generando un número mayor de clasificadores que *OVR*. Por otro lado, *OVR* es más sensible al desbalance de los datos, en el sentido de que si se intenta clasificar y la cantidad de datos representantes de esta clase es baja en comparación con el total de datos, esta clase podría tener una ponderación o importancia mucho menor que las demás, conduciendo a una clasificación incorrecta de la instancia evaluada. En cambio, con *OVO* este fenómeno no se manifiesta en igual medida, dado que se generan clasificadores uno a uno. La Figura 3.3 muestra una comparación entre estas dos estrategias.

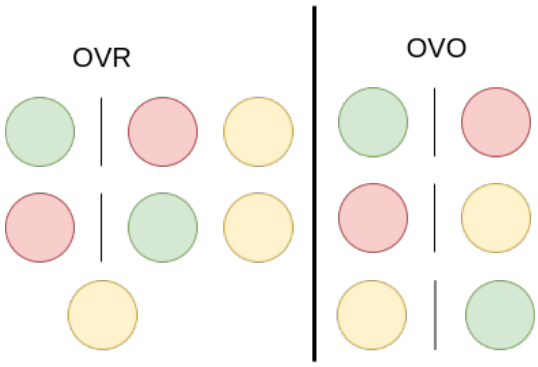

Figura 3.3: A la izquierda se muestra la estrategia de entrenamiento de SVM *OVR*, en la cual cada una de las clases se intenta clasificar contra el resto de las clases. A la derecha se muestra la estrategia *OVO*, que genera clasificadores para cada par de clases.

La obtención de un hiperplano que divida de manera lineal a los ejemplos de entrenamiento no siempre es posible. Cuando no es posible, se suele utilizar una técnica conocida como *kernelización* o *Kernel Trick* para aumentar la dimensión del dominio donde se está buscando un hiperplano e intentar obtener un hiperplano que separe linealmente a los ejemplos de entrenamiento en esa dimensión. De esta manera, es posible clasificar a nuevas instancias del problema de acuerdo a este nuevo hiperplano y transformar la clasificación obtenida a la dimensión original del problema dada por las clases de clasificación disponibles. De todas maneras, una utilización imprudente del *Kernel Trick* puede llevar a un sobreajuste del modelo.

Entre las desventajas de utilizar SVM se encuentra que los tiempos de entrenamiento asociados pueden ser significativos para grandes conjuntos de entrenamiento, pudiendo tener consumos considerables de memoria tanto durante el entrenamiento como en la validación. Así también la selección de parámetros de *Kernel* para el realizar el *Kernel Trick* puede ser compleja y se debe ejecutar evitando el sobreajuste.

Capítulo 4

Trabajos relacionados

En este capítulo se describen los principales trabajos relacionados a la temática abordada en este proyecto. El paradigma SV ha sido explorado únicamente en dos trabajos [1] [11], que son abordados en la Sección 4.1. La Sección 4.2 está dedicada a técnicas de clasificación en aprendizaje supervisado, mientras que la Sección 4.3 trata la temática de selección y extracción de atributos en aprendizaje automático. Finalmente, en la Sección 4.4 se aborda el modelo de programación MapReduce, que permite la ejecución paralela en un sistema implementado bajo el paradigma SV.

4.1. Savant Virtual

El síndrome de Savant es una condición donde los individuos que la padecen, conocidos como savants, son capaces de resolver tareas de cálculo, memoria o mnemotecnia, como encontrar el día de la semana para una fecha dada o listar números primos utilizando *métodos desconocidos*, en tiempos inferiores a los esperados normalmente y con una gran precisión.

Tomando como inspiración al síndrome de Savant, Dorronsoro et al. [1] introdujeron el paradigma SV con el objetivo de generar programas que, utilizando aprendizaje automático supervisado, encuentren soluciones a problemas con una calidad comparable a las soluciones ofrecidas por otros algoritmos conocidos. De esta manera, el componente de aprendizaje automático está asociado de manera simbólica a las reglas que un Savant incorpora automáticamente. Los autores aplicaron el paradigma SV sobre el problema HCSP, presentado

en la Sección 2, utilizando también a Min-Min como criterio de referencia para llevar a cabo el aprendizaje automático. Con el objetivo de aplicar SV a la resolución de este problema de manera paralela, el sistema planteado implementó el modelo MapReduce. Bajo este modelo, cada *mapper* se corresponde con un clasificador multiclase entrenado previamente (utilizando SVM). Un único *reducer* realiza una búsqueda local con el objetivo de mejorar la solución y posteriormente devuelve su solución. El sistema permite utilizar tantos *mappers* como tareas se tengan en la instancia del problema, posibilitando la ejecución en paralelo de cada uno. Por otra parte, cada clasificador del sistema se entrena utilizando las soluciones obtenidas mediante la aplicación del algoritmo Min-Min. Finalmente, se consigue un sistema que puede trabajar de manera paralela, generando soluciones basadas en un entrenamiento realizado utilizando el criterio Min-Min como referencia.

Los autores evaluaron el paradigma propuesto sobre un conjunto de instancias de prueba generadas de forma aleatoria (en la forma de matrices ETC, como fue presentado en la Sección 2.4), con el objetivo de comparar las soluciones obtenidas mediante Savant con las de Min-Min. Cada instancia de prueba estaba compuesta por un conjunto de tareas, un conjunto de máquinas y las duraciones de las tareas para cada máquina. Los resultados mostraron una precisión promedio (similitud con la solución de referencia) del 82 %, resultando mejor la precisión para instancias más pequeñas del problema (128 tareas y 4 máquinas) que para instancias de mayor tamaño (512 tareas y 16 máquinas). Fueron obtenidos mejores resultados al utilizar un *reducer* con búsqueda local frente a un *reducer* simple que genera una solución en base a la salida directa de cada *mapper* sin hacer modificaciones. En algunas instancias de prueba SV fue capaz de encontrar mejores soluciones que Min-Min.

El trabajo de Dorronsoro et al. [1] presenta oportunidades de trabajo futuro desde diversos puntos de vista; el *reducer* puede llevar a cabo una búsqueda paralela de mejores soluciones, se puede estudiar el uso de otro tipo de clasificadores y se pueden estudiar aplicaciones para verificar la adaptabilidad de SV a otros problemas y comparar su desempeño frente al obtenido para el problema HCSP.

Este trabajo representó el punto de partida para la investigación en torno a SV y proporciona el marco de trabajo que se sigue en el proyecto de grado, dado que involucra el mismo modelo conceptual, tipos de pruebas y modelado de problemas.

En Dorronsoro et al. [11] se estudió la aplicación de SV para la generación automática de programas para resolver el problema de la mochila. Evaluar SV para el problema de la mochila resulta interesante porque representa un problema de optimización combinatoria NP-difícil y además es un tipo de problema diferente al estudiado en el trabajo previo (HCSP) en términos de sus variables binarias y con restricciones simples, en vez de enteras y sin restricciones. En este problema se tiene un contenedor con una capacidad W y un conjunto E de n objetos, cada uno con un beneficio p_i y un peso w_i que no excede a W. El objetivo perseguido al resolver este problema es el de seleccionar objetos de E a introducir en el contenedor, maximizando el beneficio que estos aportan, sujeto a que la suma de los pesos de los objetos introducidos no exceda la capacidad W. Con respecto al modelado del problema, se optó por representar a una instancia con un peso asociado a la capacidad del contenedor estudiado y un vector donde cada índice representa a un objeto con su peso y su beneficio asociado. Se aplicó SV respetando el modelo planteado en Dorronsoro et al. [1], utilizando también SVM como clasificador y mapper. En particular, se utilizaron tantos *mappers* (replicados) como objetos disponibles en la instancia del problema y la clasificación de cada *mapper* indicaba si ese objeto había sido incluido o no en la selección de objetos asignados al contenedor. Con respecto al *reducer*, se aplicó una búsqueda local aleatorizada a la hora de generar la solución frente a una instancia del problema y dos mecanismos de corrección para soluciones infactibles: corrección ávida por beneficio y corrección ávida por peso. La corrección ávida por beneficio se basa en eliminar sucesivamente el objeto que aporte el menor beneficio hasta llegar a una solución factible, mientras que la corrección ávida por peso se basa en buscar al objeto de menor peso que tenga un peso mayor o igual al sobrepeso de la solución y eliminarlo, o eliminar al objeto de mayor peso en caso de no encontrarse objetos que cumplan con la condición anterior. Durante el trabajo se utilizaron 15.750 instancias del problema (en varios conjuntos de datos), de entre 100 y 1.500 objetos, con correlación de peso y beneficio variable para cada tamaño del problema. Con el fin de determinar una selección de atributos adecuada (dado que se contaba con información de peso y beneficio para cada objeto y además la capacidad del contenedor), se estudió la precisión media (porcentaje de soluciones correctas retornadas por la SVM) para cada conjunto de datos disponible, utilizando una selección diferente en cada instancia del estudio para el entrenamiento. Dado que la precisión media para todas las

configuraciones fue elevada y hubieron diferencias pequeñas entre las configuraciones con respecto a este valor, se optó por emplear al tipo de selección más sencillo y directo, que resultó ser el peso del objeto, su beneficio y la capacidad de la mochila. Experimentalmente se determinó que al utilizar más del 15 % de los datos para entrenar al clasificador las mejoras en precisión resultaron marginales, por lo que se optó por utilizar únicamente ese porcentaje de los datos, reduciendo de esta manera los tiempos de entrenamiento de la SVM. Como resultado se obtuvo un error medio del 3,2 % con respecto al resultado óptimo para cada instancia del problema, obtenido mediante el uso de un algoritmo exacto de referencia. El valor del trabajo de Dorronsoro et al. [11] para este proyecto de grado reside en el hecho de que amplía la información disponible en relación a SV.

4.2. Técnicas de clasificación

El trabajo de Kotsiantis [12] presenta una reseña sobre métodos de aprendizaje automático, principalmente de carácter expositivo, que ofrece guías acerca de cómo seleccionar apropiadamente clasificadores de aprendizaje automático para casos de uso comúnmente encontrados en la práctica. Se presenta a los árboles de decisión, que ofrecen buenos resultados para problemas donde los atributos toman valores discretos o categóricos; por lo tanto no sería acertado aplicarlos para un problema de optimización combinatoria con variables numéricas, como es el caso del problema estudiado en este proyecto de grado. Se destaca el hecho de que los algoritmos estadísticos en general resultan menos precisos que aquellos más sofisticados como las redes neuronales artificiales, aunque tienen tiempos de aprendizaje menores. Los algoritmos como las redes neuronales artificiales o SVM requieren de un mayor ajuste de parámetros, no son modelos interpretables por humanos y requieren de un gran conjunto de datos de entrenamiento para ser precisos al clasificar. Por otra parte, se presentan algoritmos lógicos como el de *k-nearest neighbors* o *KNN*, que tienen parámetros sencillos de ajustar y son transparentes en términos de su funcionamiento, pero que involucran una carga de memoria poco conveniente a la hora de estudiar problemas de grandes dimensiones.

El valor del trabajo de Kotsiantis [12] para este proyecto de grado reside en la guía comparativa de selección de métodos de aprendizaje automático que ofrece, que presenta los algoritmos más comunes para cada variante de aprendizaje automático, como *ID3* para árboles de decisión y *Backpropagation* para redes neuronales.

4.3. Selección y extracción de atributos

El trabajo de Chandrashekar y Sahin [13] ofrece información acerca de las técnicas más utilizadas para la selección de atributos en aprendizaje automático. La selección de atributos refiere a la acción de seleccionar un subconjunto de atributos o *features* de las instancias del problema a resolver, que efectivamente puedan describir al problema reduciendo el efecto del ruido o variables irrelevantes. Además, al reducir la cantidad de variables que describen un problema es posible comprender los datos de mejor manera y reducir los requerimientos computacionales asociados a su manejo. El problema de seleccionar atributos resulta no trivial dado que si las instancias de entrenamiento utilizadas tienen N atributos posibles, es necesario tomar en cuenta 2^N subconjuntos posibles de atributos. Chandrashekar y Sahin [13] presentan los siguientes tipos de métodos de selección de atributos para aprendizaje supervisado:

- *Filter*: Ordena a los atributos de acuerdo a algún criterio de ordenamiento que les asigna un valor y elimina aquellos atributos que queden por debajo de un umbral mínimo. Se sugieren algoritmos para la determinación de la relevancia de cada atributo, siendo el algoritmo *Mutual Information* el más relevante, que mediante el cálculo de la entropía condicional entre dos atributos determina qué tan dependientes son entre sí, lo que permite determinar qué tan dependiente es un atributo de la clasificación de una instancia de entrenamiento.
- *Wrapper*: Se utilizan algoritmos de búsqueda para encontrar un subconjunto de atributos de manera heurística, no exacta. Cada subconjunto se evalúa construyendo un clasificador entrenado con los datos sesgados utilizando únicamente los atributos del subconjunto de atributos obtenido y evaluando su precisión con respecto a un conjunto de validación. Este tipo de método de selección resulta muy costoso en términos computacionales, especialmente si el conjunto de datos es muy extenso.

- *Embedded*: Este tipo de métodos surgió como respuesta a los problemas de eficiencia asociados a los métodos de tipo *Wrapper*. Utilizan una alternativa para la evaluación, dada por una función objetivo basada en la entropía condicional entre los atributos del subconjunto de atributos actual y la clasificación. Este tipo de método de selección reduce el tiempo computacional requerido por métodos de tipo *Wrapper*.

A pesar de que cada algoritmo de selección de atributos puede comportarse de manera diferente de acuerdo al tipo de datos utilizado, Chandrashekar y Sahin [13] llevaron a cabo un experimento para mostrar la diferencia entre utilizar selección de atributos y no hacerlo. En dicho experimento se generaron conjuntos de datos, cada uno correspondiente a una condición médica, teniendo potencialmente distintos atributos. Cada instancia de entrenamiento constaba de entradas asociadas a señales eléctricas provenientes de un sistema médico y una salida o clasificación que indicaba a qué condición médica correspondían esas entradas (cáncer, diabetes, etc.). Para cada conjunto de datos se destinó el 50 % al entrenamiento y el 50 % a la validación y se consideró como medida de rendimiento la precisión de cada clasificador sobre el conjunto de validación. Se encontró que, en general, la precisión de los clasificadores que usaban selección de atributos (reduciendo la cantidad de atributos a la mitad) se mantenía muy próxima a la de los clasificadores que utilizaban todos los atributos. Por ejemplo, para un conjunto de datos que determinaba si un paciente tenía cáncer de acuerdo a 10 atributos, un clasificador (en particular SVM) obtuvo una precisión mayor al 96 %, mientras que el mismo clasificador entrenado con 5 atributos mantuvo una precisión del 95 %. Este trabajo resultó de gran importancia para determinar cómo llevar a cabo la selección de atributos para la representación del problema estudiado en este proyecto de grado.

El trabajo de Khalid et al. [14] analiza un conjunto de técnicas para la selección y extracción de atributos, con el propósito de determinar qué tan efectivas son estas técnicas a la hora de lograr un alto desempeño en clasificación. Se presentan métodos de selección de atributos como mecanismos para la obtención de conjuntos de datos de menor dimensión. Los métodos de selección de atributos pueden ser caracterizados por las siguientes etapas: búsquedas sobre los datos, generación de subconjuntos de datos y medida de la mejora en el

subconjunto de datos. Por otro lado, la extracción de atributos se presenta como la generación de nuevas características para reducir la complejidad y dar una representación más simple de los datos, expresando cada variable en el espacio de características nuevas como una combinación lineal de las variables originales. Experimentalmente, se estudiaron siete conjuntos de datos médicos que incluían información sobre enfermedades oncológicas. Los resultados obtenidos mostraron que el uso de técnicas de extracción de características permite obtener una mejor precisión que el uso de selección de características. En particular, se ubicó a PCA (*Principal Component Analysis*) como el método de extracción de características más utilizado. PCA es un método no paramétrico simple que consiste en una transformación lineal de los datos que minimiza la redundancia, medida como la covarianza, maximizando la información, medida como la varianza. Este método tiene las siguientes limitaciones: asume que la relación entre las variables es lineal, su interpretación de los datos resulta razonable únicamente si se asume que todas las variables están escaladas numéricamente y carece de un modelo probabilístico. Para sobrellevar las dos primeras limitaciones se propuso el método PCA no lineal, donde las variables son observadas como categóricas. Para afrontar la última limitación se propuso el método PPCA (*Probabilistic Principal Component Analysis*), en el que PCA es uno de los parámetros de un modelo probabilístico. El trabajo de Khalid et al. [14] fue un gran aporte a la hora de determinar cómo llevar a cabo la selección y/o extracción de atributos para la representación del problema estudiado en este proyecto de grado.

4.4. MapReduce

En el trabajo de Dean y Ghemawat [15] se presenta una introducción al modelo MapReduce, así como detalles de su implementación y de su rendimiento. Mapreduce es un modelo de programación utilizado para procesar y generar grandes conjuntos de datos. En términos genéricos, un sistema implementado bajo el modelo MapReduce recibe datos en forma de un conjunto de pares (clave, valor), los transforma mediante la aplicación de una función definida por el usuario y devuelve un conjunto de pares (clave, valor) de salida agregados de acuerdo a un criterio también definido por el usuario. La transformación aplicada a los datos en la entrada del sistema se llama *map* y la transformación aplicada a los datos en la salida del sistema se llama *reduce*. En la implemen-

tación propuesta por los autores, la computación dada en las funciones *map* y *reduce* es paralelizada automáticamente en múltiples recursos de cómputo, posibilitando el procesamiento masivo y paralelo de datos. En concreto, el sistema presentado se encarga de inicializar múltiples hilos de ejecución en un régimen maestro-esclavo, donde el hilo maestro se encarga de asignar tareas *map* o *reduce* a los hilos esclavos. De esta manera, se cuenta con *map workers* y *reduce workers*. Cada *map worker* recibe una porción de los datos, los procesa y guarda su salida en un disco local. Cada *reduce worker* recibe una ubicación (como por ejemplo una referencia a otra máquina de la red en un cluster) de donde deberá leer los datos intermedios generados por los *map workers*, procesarlos y guardar su salida en un archivo. Cuando todos los hilos finalizan su ejecución, el hilo *master* retoma el control y reanuda la ejecución del programa de usuario que haya desencadenado la ejecución del MapReduce. En el trabajo de Dean y Ghemawat [15] también se mencionan técnicas utilizadas para el control de fallas, balanceo de carga y detalles específicos de implementación. Además, se muestra una serie de resultados experimentales que consisten en una evaluación del rendimiento del modelo ejecutado en un cluster de 1800 máquinas de iguales características, en dos tipos de computación: búsqueda de un patrón en un terabyte de datos y ordenamiento de un terabyte de datos. El valor de este trabajo en relación al proyecto de grado reside en el hecho de que SV propone generar programas concurrentes automáticamente y la utilización de MapReduce puede resultar útil para posibilitar el paralelismo en dichos programas.

4.5. Resumen

La Tabla 4.1 presenta un resumen de los trabajos relacionados, manteniendo el orden en el que fueron mencionados en las secciones previas. Se indica el autor, el año de publicación del trabajo y una breve descripción con los conceptos clave del mismo.

El análisis de la literatura relacionada muestra que el paradigma SV en particular no ha sido estudiado de manera extensiva y permite llegar a la conclusión de que existe lugar para contribuir, por ejemplo profundizando en la utilización de métodos de aprendizaje automático distintos a SVM.

Autores	Año	Conceptos clave
Dorronsoro et al. [1]	2013	Se presenta el paradigma Savant para la paralelización de una heurística de planificación con aprendizaje automático utilizando SVM
Dorronsoro et al. [11]	2016	Estudio del problema de la mochila bajo el paradigma Savant
Kotsiantis [12]	2007	Compendio de las técnicas de clasificación más utilizadas en aprendizaje automático supervisado
Chandrashekar y Sahin [13]	2014	Estudio comparativo de técnicas de selección de atributos en aprendizaje automático
Khalid et al. [14]	2014	Compendio de técnicas de selección de atributos en aprendizaje automático, con foco en PCA
Dean y Ghemawat [15]	2008	Utilización de MapReduce para procesamiento de grandes volúmenes de datos en clusters

Tabla 4.1: Trabajos relacionados al proyecto de grado

Capítulo 5

Implementación

En este capítulo se presenta el sistema de software construido para evaluar el rendimiento comparativo de las redes neuronales y SVM en términos del aprendizaje del problema HCSP. Este sistema fue implementado utilizando *Python* y la biblioteca de aprendizaje automático *Scikit-learn*, entre otras.

La arquitectura del sistema está conformada por tres componentes fundamentales. El primero es el módulo encargado de generar los datos utilizados en forma de ejemplos de entrenamiento. Este módulo se describe en la Sección 5.1. También se tiene un módulo dedicado a la generación de clasificadores, incluyendo redes neuronales y SVM. Este módulo se describe en la Sección 5.2. Finalmente, descrito en la Sección 5.3, se tiene un módulo encargado de clasificar nuevas instancias del problema.

5.1. Generación de instancias del problema

Dado el problema HCSP, surgen dos posibles maneras de utilizar una instancia del problema (dado por una matriz ETC) y su resolución (una planificación) en el contexto de aprendizaje automático. Una opción implica utilizar a la matriz ETC completa como entrada del clasificador de elección y, hacer que la salida esperada sea la planificación, es decir todas las asignaciones tarea-máquina esperadas. Esta opción constituye un problema de clasificación multiclase, y la desventaja principal que presenta es la de no permitir escalar en términos de máquinas o tareas. Por lo tanto, si un clasificador es entrenado para aprender a resolver el problema HCSP con instancias del problema de

dimensión $M \times N$, no será aplicable para ninguna otra dimensión. La otra opción que se presenta, y la que fue escogida finalmente en este trabajo tomando como referencia el trabajo de Dorronsoro et al. [1], es la de descomponer a una matriz ETC en varios ejemplos de entrenamiento. Por lo tanto, dado un ejemplo de entrenamiento bajo este paradigma, se posee la información asociada únicamente a una tarea como la entrada, y a la asignación tarea-máquina de esa tarea como la salida esperada. La utilización de este paradigma causa que una instancia del problema (con su planificación asociada) de dimensión $M \times N$ se convierta en M potenciales ejemplos de entrenamiento, y proporciona la posibilidad de escalar en términos de la cantidad de tareas a ejecutar. Esta oportunidad surge dado el hecho de que un ejemplo de entrenamiento está limitado únicamente por la cantidad de máquinas, y se explora con más detalle en la Sección 5.2.

Con el objetivo de generar ejemplos de entrenamiento y validación como los discutidos anteriormente para entrenar y evaluar a los clasificadores construidos, fue necesario desarrollar un componente de software que pudiera hacerlo a demanda. Este componente está conformado fundamentalmente por scripts, y en la Figura 5.1 se puede ver un diagrama de secuencia donde se explicita su funcionamiento.

Inicialmente, se utiliza un script extraído de la tesis de Nesmachnow [2] para generar instancias del problema, y se genera una solución para dicha instancia utilizando una implementación de Min-Min. Estos elementos son persistidos directamente al sistema de archivos de manera iterativa, de acuerdo a la cantidad de ejemplos de entrenamiento que se deseen generar. Una vez finalizada esta etapa, es necesario modificar la estructura de los datos generados, de manera de obtener aquella estructura soportada por los clasificadores de aprendizaje automático. Por lo tanto, se procede a tomar cada par constituido por una instancia del problema y su planificación esperada, y se generan tantos ejemplos de entrenamiento como tareas existan en la instancia del problema. Es decir que para una instancia del problema de dimensión $M \times N$ tendremos M ejemplos de entrenamiento. Finalmente, se agrupan estos ejemplos de entrenamiento y se persisten en formato CSV para su uso posterior. Este formato es utilizado por ser el estándar soportado más comúnmente por las bibliotecas de manejo de datos utilizadas.

El componente de software encargado de la generación de instancias del problema, como el resto de los mencionados en este capítulo, fue desarrollado de manera de permitir su ejecución automatizada, a pesar de existir un usuario presente en el diagrama de secuencia. Dicho usuario puede bien ser un usuario final o un script encargado de invocar a este sistema. La posibilidad de automatización favorece la generación de ejemplos de entrenamiento a gran escala y de manera paralela mediante el uso de hilos. Además, se ofrece al usuario la posibilidad de determinar cuántas instancias del problema (con sus correspondientes soluciones) serán generadas, dónde se persisten, y las características de los tipos de instancias del problema a generar, como por ejemplo la dimensión de las mismas.

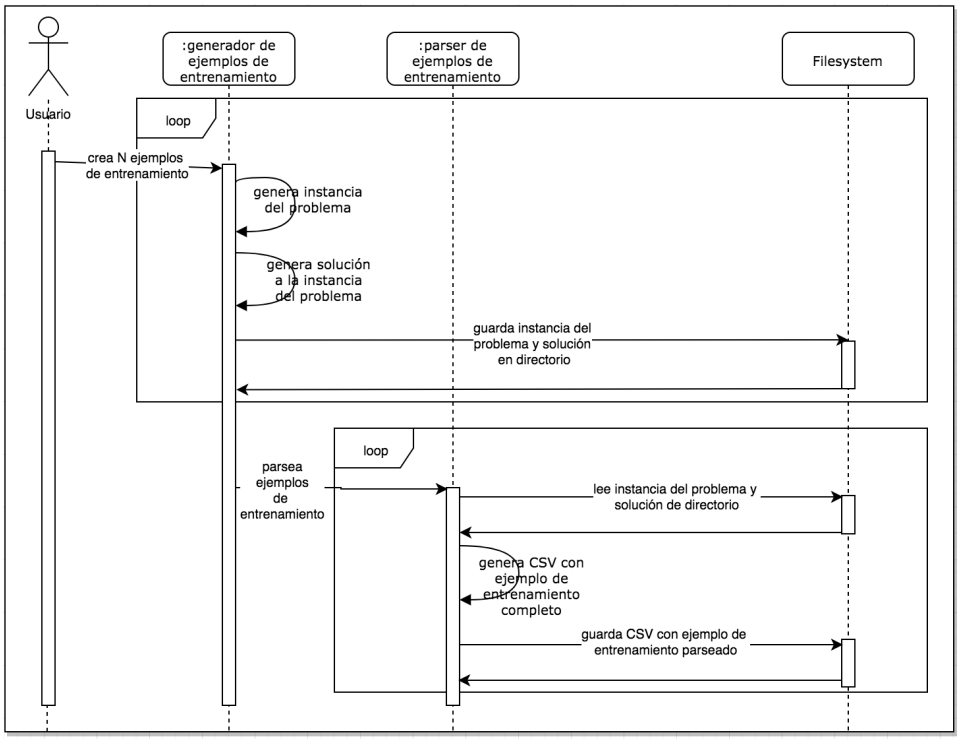

Figura 5.1: Diagrama de secuencia de generación de ejemplos de entrenamiento.

5.2. Generación de clasificadores y entrenamiento

Se implementó un componente de software dedicado a crear, entrenar y persistir clasificadores de aprendizaje automático. Este componente permite seleccionar como clasificador de aprendizaje automático a utilizar *SVM* o redes neuronales.

Se utilizó la biblioteca *Pandas* [16] de *Python*. Esta biblioteca simplifica el manejo de datos mediante estructuras de datos indizadas, proporcionando un conjunto de métodos que permiten realizar operaciones sobre estas estructuras para facilitar el análisis de los datos, haciendo un uso eficiente de los recursos computacionales sobre los que se ejecute.

Como se mencionó en la Sección 5.1, se utilizó cada fila de la matriz ETC del problema como los atributos para un único ejemplo de entrenamiento. Esto quiere decir que para una matriz de 512 tareas y 16 máquinas se obtienen 512 ejemplos de entrenamiento, cada uno clasificado en una de las 16 clases objetivo. Estas clases se corresponden una a una con las máquinas asignadas por el algoritmo Min-Min. La figura 5.2 muestra un ejemplo de matriz ETC de 512 tareas y 16 máquinas.

$$\begin{bmatrix} & m_1 & m_2 & \dots & m_{16} & objetivo \\ t_1 & 35.74 & 39.62 & \dots & 456.89 & 4 \\ t_2 & 75.55 & 97.41 & \dots & 579.19 & 3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ t_{512} & 130.12 & 216.59 & \dots & 789.84 & 5 \end{bmatrix}$$

Figura 5.2: Ejemplo de matriz ETC con atributo objetivo para el aprendizaje. Cada fila t_i con $i = 1 \dots 512$ incluye información correspondiente a una tarea. La columna objetivo muestra la asignación de máquina para la correspondiente tarea i y éste es el atributo objetivo para los clasificadores. La información asociada a la tarea, que incluye los tiempos de ejecución estimados de la tarea t_i para cada una de las máquinas disponibles, junto con su clasificación, equivale a un ejemplo de entrenamiento para el clasificador.

A diferencia de utilizar la matriz ETC completa como ejemplo de entrenamiento, utilizar cada fila de la matriz se traducirá en menores tiempos de entrenamiento y además da flexibilidad a la hora de clasificar problemas con otro número de tareas e igual número de máquinas. Esto es, como los clasificadores se entrenan con información referente a cada tarea, se pueden clasificar problemas que tengan un número menor o mayor de tareas. Este tipo de escalado en función de tareas es además realista, ya que en la práctica el hardware tiende a ser una restricción menos flexible que la cantidad de tareas, algo que puede fluctuar con mayor frecuencia en función del tiempo dependiendo del caso de estudio.

Para el entrenamiento se utilizaron 100 instancias del problema de dimensión 512×16, lo que se traduce en 51200 instancias de entrenamiento, cada una con un atributo objetivo que varía entre 1 a 16, siendo éste el identificador de la máquina la cual la tarea es asignada. El componente de software implementado carga en memoria 100 archivos CSV que se encuentran en un directorio definido. Cada uno de estos archivos es una matriz ETC con el atributo objetivo correspondiente asignado para cada tarea, como se muestra en la figura 5.2. Estos atributos objetivo se agregan en un sólo *DataFrame* de *Pandas* [17]. Este *DataFrame* es escalado y utilizado para el entrenamiento de los clasificadores.

Debido a que los datos contienen atributos cuyas magnitudes tienen una gran varianza y los algoritmos utilizados manejan distancias euclidianas para el aprendizaje, los datos fueron escalados antes de realizarse el entrenamiento de los clasificadores. Otra razón para escalar los datos antes del entrenamiento es que *Gradiente Descendente* tiende a converger más rápido cuando los datos están escalados [18].

El escalado de los datos se realiza mediante la clase *preprocessing.StandardScaler* de la biblioteca *Scikit-Learn* para *Python* [19]. Esta biblioteca provee el soporte para los métodos de aprendizaje automático y herramientas para el preprocesamiento de los datos. En particular la clase *StandardScaler* analiza cada atributo de manera independiente y almacena la mediana y la desviación estándar para luego ser utilizados en datos nuevos que ingresan al modelo [20].

Para la construcción de los clasificadores de aprendizaje automático se utilizaron las clases *svm.SVC* para la construcción de *SVM* y *neural_network.MLPClassifier* para la construcción de las redes neuronales, ambas clases pertenecientes a la biblioteca *Scikit-Learn* de *Python*.

El clasificador *SVM* se utilizó siguiendo la estrategia *OVR*, dado que genera menos clasificadores que mediante la estrategia *OVO* de acuerdo a lo expuesto en la Sección 3.2.2. Además, se entendió a *OVR* como la mejor estrategia dada la baja heterogeneidad de las instancias del problema, por lo cual es razonable no tener un desbalance en la cantidad de representantes de cada clase.

Con respecto a las redes neuronales, en particular sobre su arquitectura, se siguió la heurística recomendada por Lane [21], dada la falta de consenso existente en la comunidad con respecto a la manera óptima de determinar la cantidad de neuronas en las capas ocultas de una red neuronal de acuerdo a los casos de uso en los que se apliquen dichos clasificadores. Según esta heurística, la cantidad de neuronas en una capa oculta (o N_h) se determina con la siguiente fórmula $N_h = \frac{N_s}{(\alpha*(N_i+N_o))}$, siendo:

N_i = cantidad de neuronas de entrada

N_o = cantidad de neuronas de salida

N_s = cantidad de instancias de entrenamiento

α = factor de escalamiento arbitrario, con valor igual a 2 para este estudio

El resto de los parámetros iniciales de configuración de la red neuronal se definieron utilizando la clase *model_selection.GridSearchCV*, que realiza una búsqueda exhaustiva de los mejores parámetros para el modelo mediante validación cruzada. A medida que se avanzó con el trabajo, estos parámetros fueron modificados para ajustarlos a las necesidades puntuales del proyecto. El detalle de los parámetros iniciales seleccionados se encuentra en la Sección 6.1.

A un nivel de abstracción superior, tanto para SVM como para redes neuronales, se utilizó la clase *pipeline.Pipeline* de la biblioteca *Scikit-Learn*. Esta clase envuelve al clasificador escogido, y permite que los datos fluyan desde su forma original, pasando secuencialmente por cada etapa del preprocesamiento, para finalmente ingresar en el clasificador. La utilización de esta clase permite la evaluación del modelo con estrategias como validación cruzada, simplificando la implementación del preprocesamiento de los datos mediante una encadenación de transformaciones atómicas. En particular, se utilizó la clase *Pipeline* con el escalado encadenado al clasificador seleccionado.

Para realizar el entrenamiento de los clasificadores se utilizaron los servicios de máquinas virtuales EC2 de Amazon Web Services [22]. Este servicio permite escalar de forma sencilla los recursos computacionales. En particular se generó una máquina virtual con 2 CPUs virtuales cada uno de 2.5GHz, y 16GB de RAM. El uso de estos recursos permitió realizar el entrenamiento de los clasificadores en un ambiente aislado y seguro para prevenir fallas por utilizar ambientes locales.

Finalmente, el componente de software desarrollado se encarga de persistir los clasificadores generados. Para esto se utiliza la clase *externals.Joblib* de *Scikit-Learn*. El uso de *Joblib* es sugerido cuando se utilizan clasificadores de *Scikit-Learn*, dado que provee una manera eficiente de almacenar objetos que contienen gran cantidad de *arrays* de *numpy* [23], que es el caso de modelos de *Scikit-Learn* ya entrenados [24]. Esta biblioteca no solo permite llevar a cabo la persistencia de los modelos sino que también permite la carga en memoria de los modelos para su utilización.

5.3. Clasificación

En lo que respecta a la clasificación, se implementó un componente encargado de clasificar un conjunto de ejemplos de validación dado uno o más clasificadores ya entrenados. Llevar a cabo esta implementación fue necesario dado que por restricciones de los recursos disponibles, evaluar el rendimiento de los clasificadores construidos mediante validación cruzada insumía una gran cantidad tiempo, limitando la agilidad del desarrollo. Como consecuencia, se consideró como alternativa válida separar un conjunto de datos de validación a clasificar con este módulo, con el fin de obtener métricas de rendimiento de esta manera.

El componente implementado, además de disponibilizar las funcionalidades descritas anteriormente, proporciona métricas de interés adicionales. Dentro de las métricas calculadas se encuentran el makespan esperado y el obtenido mediante predicciones. Estas métricas son consideradas como fundamentales para evaluar el rendimiento de los clasificadores, dado que de manera independiente a la precisión, es necesario evaluar la métrica fundamental asociada al éxito de una solución del problema HCSP, que como se explica en la Sección 2.4, es el tiempo insumido por la máquina que finaliza su ejecución por último. Con estos dos valores se puede evaluar qué tan exitosa fue la planificación obtenida mediante el uso de clasificadores en comparación con la solución obtenida mediante la heurística de referencia. Finalmente, se calcula el porcentaje de ocasiones en las que el clasificador, al predecir erróneamente, escoge una máquina más rápida (constituyendo una acción ávida). Esta métrica se calcula con el fin de evaluar si un clasificador se alejó de lo esperado en términos de precisión y makespan por haber aprendido a comportarse de manera ávida y por lo tanto aprendiendo a maximizar el beneficio local para cada tarea.

La implementación del componente de clasificación consta fundamentalmente de scripts, y la Figura 5.3 presenta un diagrama de secuencia que explicita su funcionamiento.

En la implementación, inicialmente se invoca el script encargado de realizar todo este análisis sobre un conjunto de ejemplos de validación. Estos ejemplos de validación deben estar presentes en un directorio especificado mediante opciones al invocar el script. Posteriormente, el script invoca al script de clasificación sobre cada uno de los ejemplos de validación encontrados. El script de clasificación se encarga de cargar el clasificador a utilizar desde una ubicación especificada del sistema de archivos. Una vez que todas las instancias de validación se encuentran clasificadas, se efectúa el cálculo de métricas en el script de análisis. Finalmente, el resultado es devuelto al usuario.

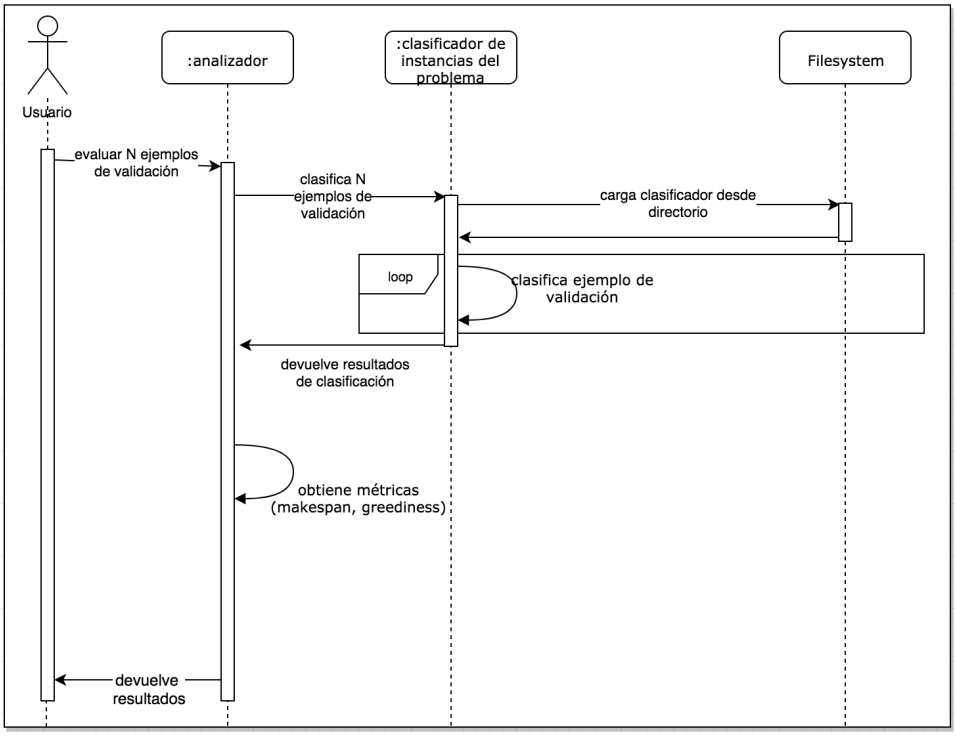

Figura 5.3: Diagrama de secuencia de clasificación de ejemplos de validación.

Capítulo 6

Análisis Experimental

En este capítulo se presentan las decisiones tomadas en torno a las configuraciones de los clasificadores y el análisis de los resultados obtenidos durante los experimentos.

6.1. Decisiones de configuración

Durante el análisis experimental fueron realizadas comparaciones entre las clasificaciones obtenidas por redes neuronales y SVM. En particular, fue de interés estudiar el comportamiento de las redes neuronales y SVM en términos del tamaño del problema.

Tomando el trabajo de Dorronsoro et al. [1] como inspiración, se optó por trabajar con clasificadores entrenados con una determinada dimensión del problema (en particular 512 tareas sobre 16 máquinas o 512×16) a la hora de clasificar y evaluar instancias del problema con menor, igual y mayor cantidad de tareas. Para hacer esto posible, se generaron instancias del problema de dimensión 512×16, utilizadas para el entrenamiento de los clasificadores. Así también, como se presentó en la Sección 5.1, para cada dimensión del problema desde 17×16 hasta 1024×16, se generaron conjuntos de 10 instancias del problema, las cuales fueron utilizadas para calcular y comparar la precisión en la clasificación para redes neuronales y SVM.

Las redes neuronales fueron entrenadas con diferentes cantidades de capas ocultas, habiéndose generado con 2, 3 y 4 capas ocultas, siendo 4 el máximo número de capas ocultas empleado debido a limitaciones de recursos y tiempos de entrenamiento altos.

Como fue mencionado en el Capítulo 5, para obtener una primera selección de parámetros de configuración para el entrenamiento de las redes neuronales se utilizó el método *GridSearch* de la clase *model_selection.GridSearchCV* de *scikit-learn*. Este método realiza una búsqueda, mediante validación cruzada, de los mejores parámetros para el entrenamiento de un clasificador. Para eso es necesario definir conjuntos de posibles valores a ser utilizados en los parámetros de los clasificadores. A continuación se presentan los valores considerados para cada uno de esos parámetros. Para el parámetro *solver*, se consideraron el método de optimización de los pesos *Broyden–Fletcher–Goldfarb–Shanno* para memoria limitada (*lbfgs*), Gradiente Descendente (*sgd*) y el método Adam (*adam*). Para el coeficiente de aprendizaje, llamado *alpha*, se consideraron los valores *0.01* y *0.0001*. Finalmente, para el parámetro *activation*, que indica la función de activación utilizada, se consideraron las funciones *relu*, *tanh* e *identity*. Los resultados del método *GridSearch* para los parámetros mencionados se muestran en la Tabla 6.1. Con respecto al parámetro *activation* se observó que el valor seleccionado por el método *GridSearch* fue *relu*; esto fue de particular interés debido a que dicha función de activación cambia de forma abrupta en su pendiente. Las primeras pruebas realizadas utilizando la red neuronal con activación *relu*, brindaron resultados de *makespan* mayores a los resultados de *makespan* generados por SVM. Al variar la función de activación de la red neuronal, se obtuvieron resultados de *makespan* menores a los valores de *makespan* obtenidos con SVM. Esto llevó a variar las funciones de activación para estudiar el comportamiento de las redes neuronales con funciones de activación con pendientes que no tuvieran cambios abruptos. El resto de los parámetros se mantuvieron de acuerdo a los resultados del método *GridSearch*.

Se evaluaron las redes neuronales resultantes de las combinaciones de las funciones de activación *relu*, *tanh* e *identity* y las cantidades de capas ocultas, profundizando en las pruebas para aquellas combinaciones que mostraban resultados más prometedores.

Parámetro	Valor
solver	lbfgs
alpha	0.01
activation	relu

Tabla 6.1: Valores para los parámetros de las redes neuronales, obtenidos mediante el método *GridSearch* de la biblioteca *scikit-learn*

6.2. Análisis combinado para redes neuronales de 2, 3 y 4 capas ocultas

En primer lugar fueron comparados los resultados de *makespan* para soluciones obtenidas a partir de las redes neuronales, utilizando como funciones de activación a *tanh*, *identity* y *relu*, con 2, 3 y 4 capas ocultas. Asimismo, en esta comparación, también fueron analizados los resultados obtenidos con SVM.

Cada clasificador fue entrenado con 100 instancias del problema de dimensión 512×16, lo que se traduce en 51200 instancias de entrenamiento para los clasificadores. Para cada dimensión del rango estudiado, se utilizaron 10 instancias del problema para validar el rendimiento de las redes neuronales y SVM. Los resultados presentados en esta sección son promedios de los resultados individuales de estas 10 instancias.

Fueron identificados aquellos clasificadores que obtuvieron mejores resultados en *makespan*. Las Figuras 6.1, 6.2 y 6.3 muestran las diferencias porcentuales de *makespan* con respecto al *makespan* obtenido por el algoritmo Min-Min, para la clasificación de instancias del problema en un rango de dimensiones que va desde 17×16 a 1024×16.

De manera similar, los resultados también fueron agrupados de acuerdo a su dimensión. Esta agrupación fue realizada de a 200 tareas con la excepción de que el primero y el último de los conjuntos van desde la dimensión 17×16 a 200×16 y 1000×16 a 1024×16, respectivamente.

La Figura 6.1 muestra los resultados para las redes neuronales utilizando la función de activación *relu*. En dicha figura se observa que el *makespan* obtenido por los resultados de la SVM se aproxima más al *makespan* dado por los

resultados del algoritmo Min-Min que para los resultados obtenidos con las redes neuronales, cualquiera sea la cantidad de capas ocultas utilizada.

Cabe destacar que a medida que aumenta la cantidad de capas ocultas de la red neuronal, los resultados se alejan de los valores esperados llegando, para clasificaciones sobre dimensiones pequeñas, a estar a más del 100 % por sobre los valores esperados. Los mejores resultados de las redes neuronales se observan para aquellas redes neuronales con solo dos capas ocultas.

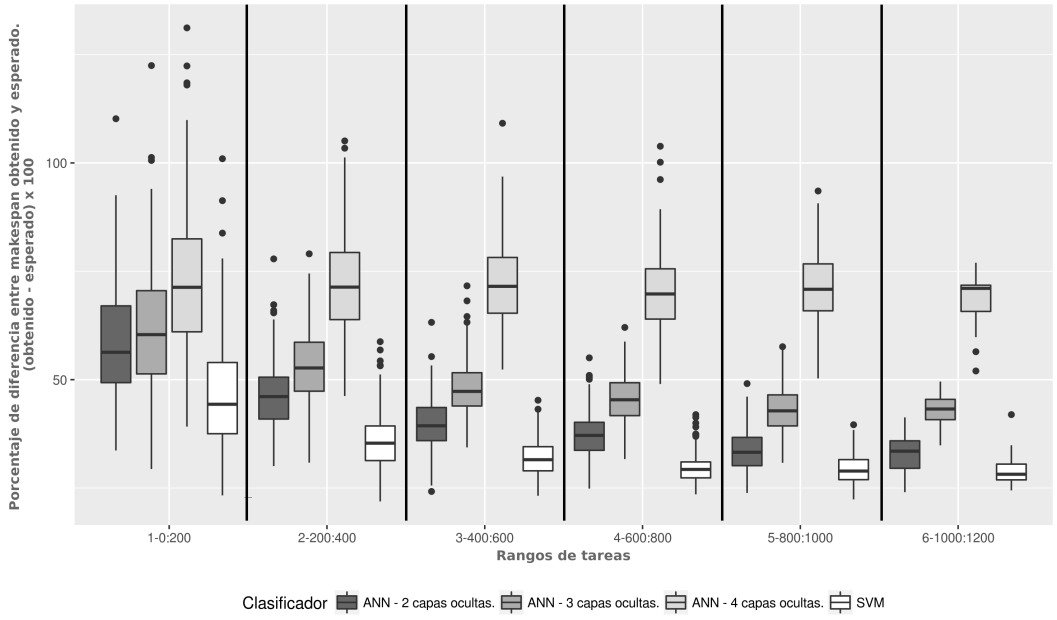

Figura 6.1: Comparación de la diferencia porcentual de los resultados de *makespan* para redes neuronales con función de activación *relu* y para SVM, con respecto al *makespan* obtenido por el algoritmo Min-Min. Se comparan redes neuronales de 2, 3 y 4 capas ocultas.

La Figura 6.2 muestra los resultados para las redes neuronales utilizando la función de activación *identity*. En este caso los resultados para las redes neuronales son más próximos a los valores esperados dados por el algoritmo Min-Min que los resultados dados por SVM. Ya desde tareas de dimensión 200×16 se comienzan a observar mejoras en el *makespan*, siendo aún más evidentes para instancias del problema de dimensión mayor. Nuevamente se observa que los resultados más próximos al *makespan* esperado están dados por la red neuronal con dos capas ocultas.

En la Figura 6.3 se observan los resultados para las redes neuronales utilizando la función de activación *tanh*. En este caso, la red neuronal con dos capas ocultas mejora los resultados obtenidos a partir de instancias del problema de dimensión 400×16 en adelante.

En los casos presentados en las Figuras 6.1, 6.2 y 6.3, los mejores resultados de las redes neuronales se encontraron para aquellas de solo dos capas ocultas. Fue profundizado el estudio de las redes neuronales con solo dos capas ocultas, de cara a entender mejor la naturaleza de las soluciones generadas.

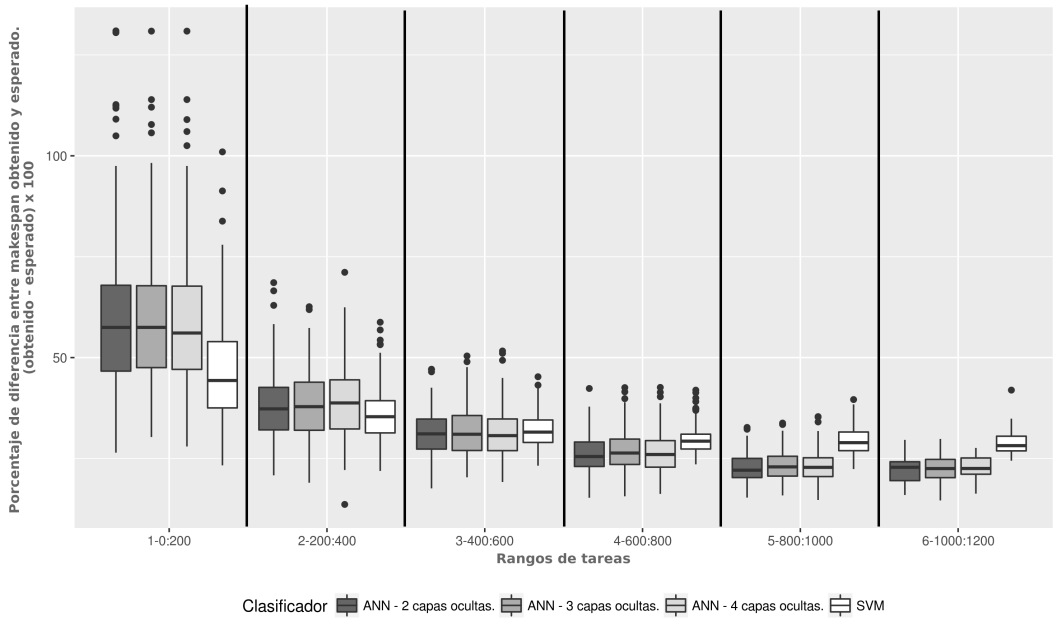

Figura 6.2: Comparación de la diferencia porcentual de los resultados de *makespan* para redes neuronales con función de activación *identity* y para SVM, con respecto al *makespan* obtenido por el algoritmo Min-Min. Se comparan redes neuronales de 2, 3 y 4 capas. Así también se muestran los resultados obtenidos con el algoritmo de SVM.

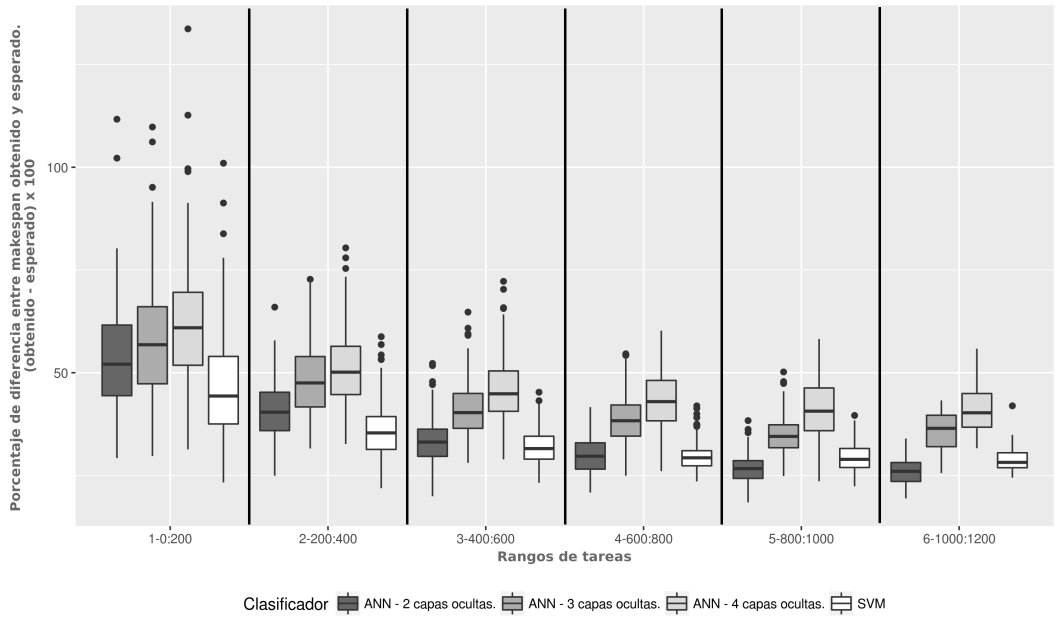

Figura 6.3: Comparación de la diferencia porcentual de los resultados de *makespan* para redes neuronales con función de activación *tanh* y para SVM, con respecto al *makespan* obtenido por el algoritmo Min-Min. Se comparan redes neuronales de 2, 3 y 4 capas. Así también se muestran los resultados obtenidos con el algoritmo de SVM.

6.3. Red neuronal con activación *relu* de dos capas ocultas

La Figura 6.4 muestra la diferencia porcentual de *makespan* entre la red neuronal de dos capas ocultas con función de activación *relu* y la SVM con respecto al *makespan* esperado. Como ya se observó, la SVM muestra valores más próximos a los valores esperados que la red neuronal.

Por otro lado, la Figura 6.5 muestra la precisión de la clasificación para ambos clasificadores. Se observa que la precisión en la clasificación aumenta a medida que la dimensión de las instancias de prueba aumenta, tanto para la red neuronal como para la SVM.

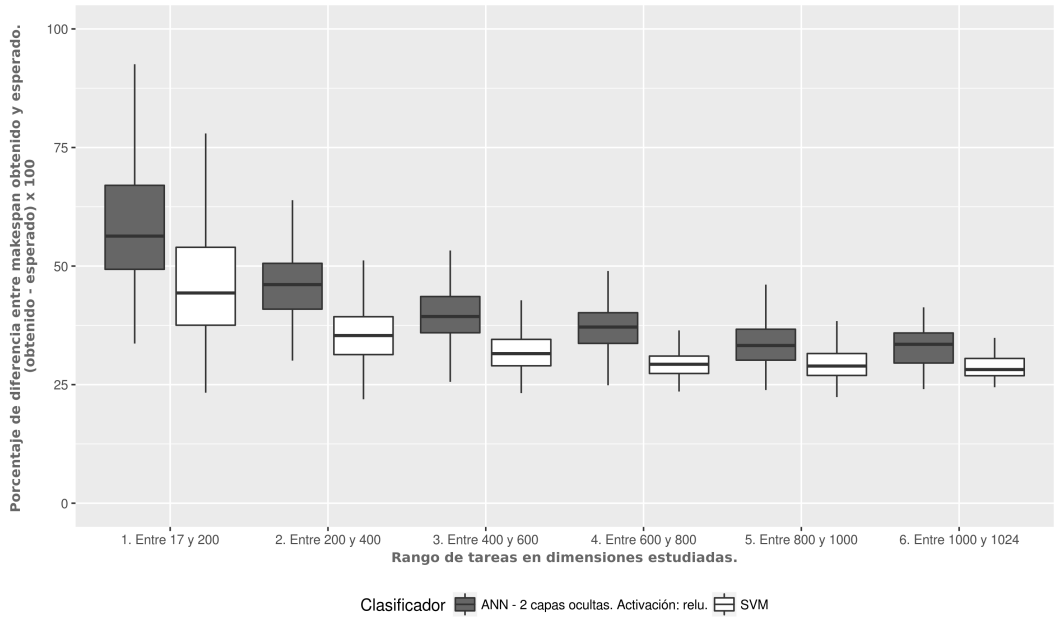

Figura 6.4: Comparación de la diferencia porcentual de *makespan* para la red neuronal con activación *relu*, de dos capas ocultas con respecto a los valores esperados obtenidos con el algoritmo Min-Min. Así también se muestran los resultados obtenidos para la SVM. Los resultados se muestran divididos en rangos de dimensión desde 17×16 a 1024×16.

La precisión de la red neuronal es levemente mayor que la precisión de la SVM. Esto, en comparación con la diferencia de *makespan* de la Figura 6.4, es de interés dado que, si bien la precisión de la red neuronal es levemente mejor, la SVM genera mejores resultados en términos de *makespan*. Este escenario también fue identificado para las demás redes neuronales en las cuales se profundizaron los estudios.

Para poder explicar lo antedicho, se calculó el porcentaje de selección de mejores máquinas para ambos clasificadores, como se menciona en la Sección 5.3. La Figura 6.6 muestra dicha métrica para ambos clasificadores. Se puede observar que SVM elige más máquinas con menor tiempo de ejecución que la red neuronal, cuando se seleccionan máquinas diferentes a las esperadas.

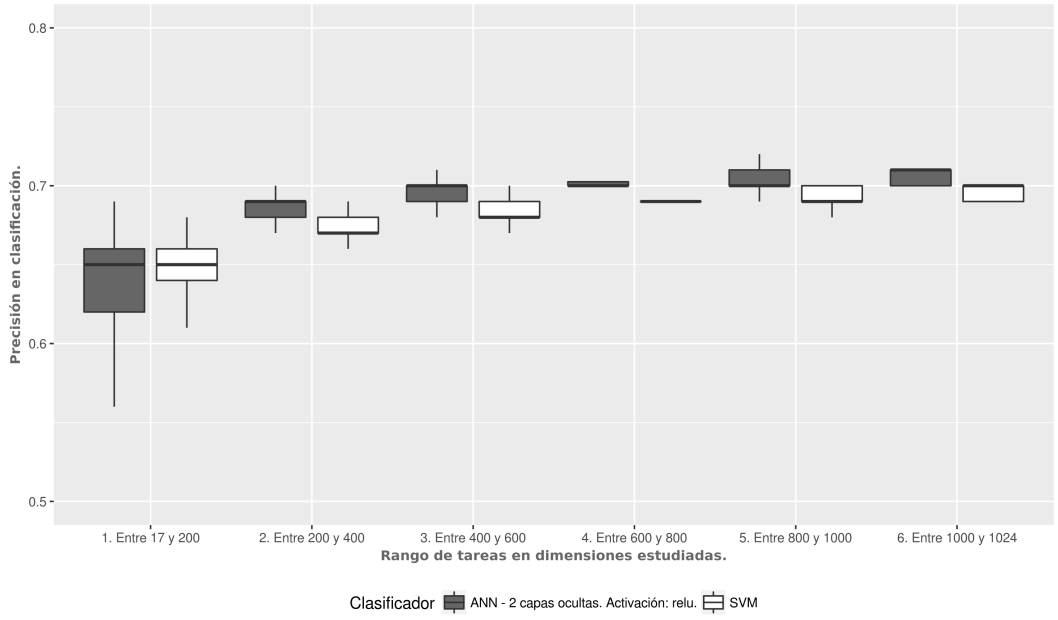

Figura 6.5: Precisión en clasificación para la red neuronal con función de activación *relu* de dos capas ocultas y para la SVM. Los resultados se muestran divididos en rangos de dimensión desde 17×16 a 1024×16.

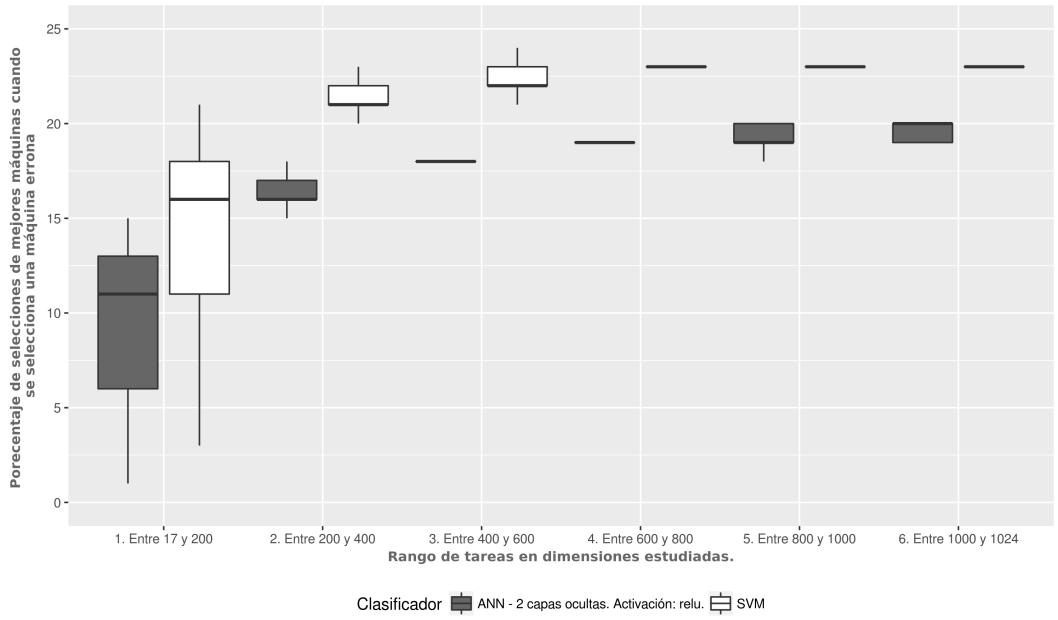

Figura 6.6: Porcentaje de selección de máquinas mejores frente a una selección diferente a la esperada para la red neuronal con activación *relu* de dos capas ocultas y para la SVM.

6.4. Red neuronal con activación *identity* de dos capas ocultas

La Figura 6.7 muestra la diferencia porcentual de *makespan* para la red neuronal con función de activación *identity* y la SVM, con respecto a los resultados esperados obtenidos con el algoritmo Min-Min. Se observa que para dimensiones mayores a 400×16, la red neuronal conduce a valores de *makespan* levemente mejores que los valores de *makespan* obtenidos con SVM. En este caso, la precisión en la clasificación, que se observa en la Figura 6.8, no tiene diferencias sustanciales.

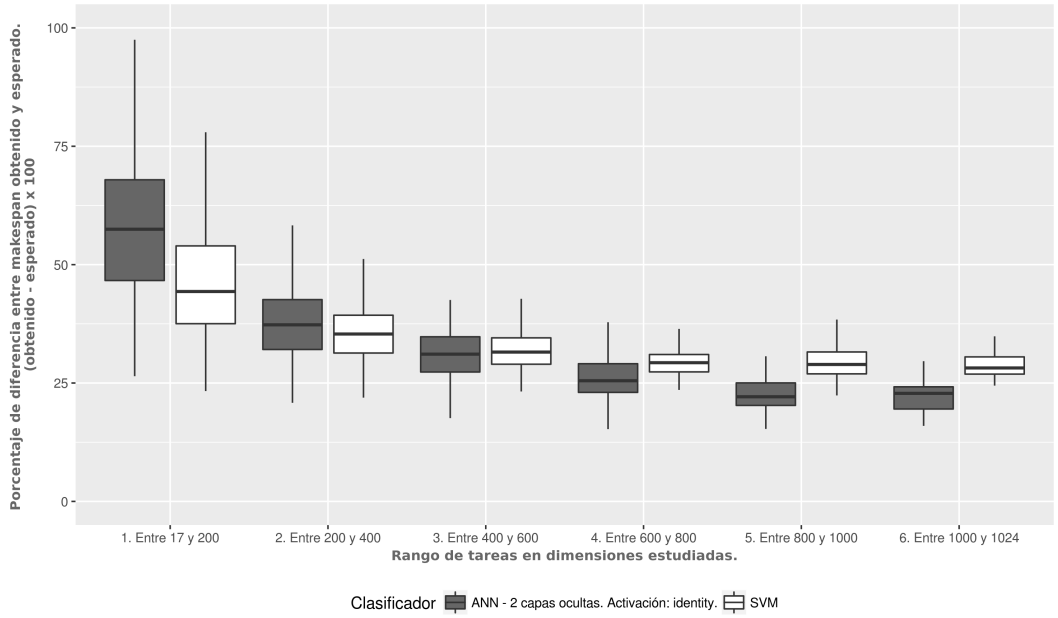

Figura 6.7: Comparación de la diferencia porcentual de *makespan* para la red neuronal con activación *identity*, de dos capas ocultas con respecto a los valores esperados obtenidos con el algoritmo Min-Min. Así también se muestran los resultados obtenidos para la SVM. Los resultados se muestran divididos en rangos de dimensión desde 17×16 a 1024×16

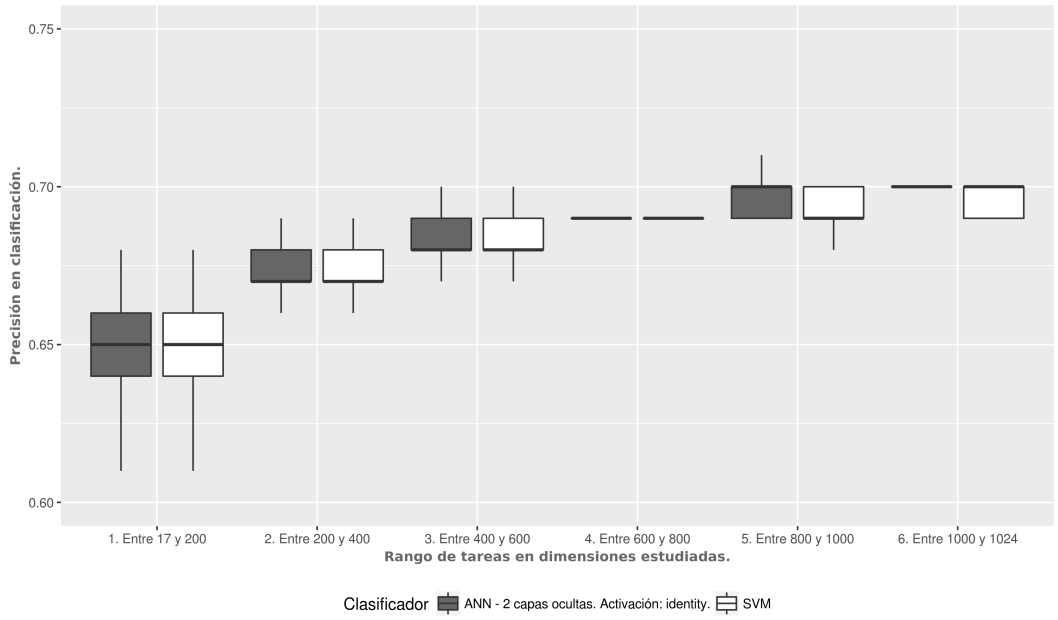

Figura 6.8: Precisión en clasificación para la red neuronal con función de activación *identity* y para la SVM. Los resultados se muestran divididos en rangos de dimensión desde 17×16 a 1024×16.

Al observar el porcentaje de mejores máquinas seleccionadas frente a un error en la Figura 6.9, se observa que la red neuronal con función de activación *identity* selecciona máquinas más rápidas en proporciones similares a la SVM, a diferencia de los resultados presentados para la red neuronal con función de activación *relu*, que tiende a elegir máquinas más lentas que la SVM frente a un error. Estos resultados llevan a pensar que el hecho de que la proporción de selección de mejores máquinas por parte de la red neuronal con función de activación *identity* sea mayor que para el caso de *relu*, conduce a una leve disminución del *makespan*.

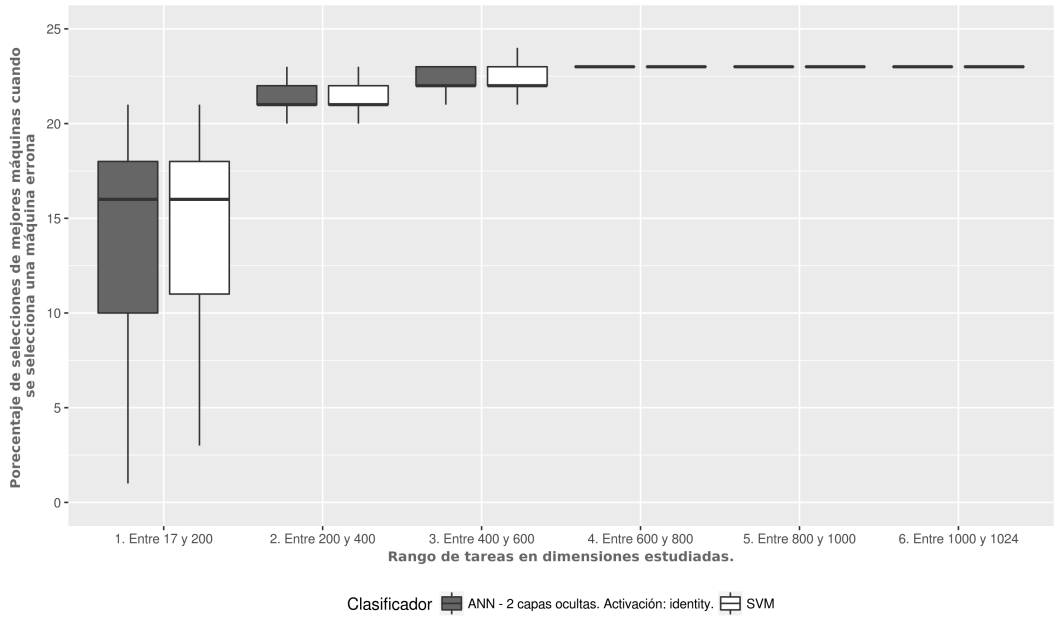

Figura 6.9: Porcentaje de selección de máquinas mejores frente a una selección diferente a la esperada para la red neuronal con activación *identity* de dos capas ocultas y para la SVM.

6.5. Red neuronal con activación *tanh* de dos capas ocultas

La Figura 6.10 muestra las diferencias porcentuales de *makespan* para la red neuronal con activación *tanh* y para la SVM. En esta se observa una leve mejora en *makespan* para la red neuronal con respecto al *makespan* obtenido con SVM, para dimensiones grandes. Esta mejora va acompañada de una mejora en la precisión, sustancial en comparación con las precisiones de las otras funciones de activación estudiadas, como se observa en la Figura 6.11.

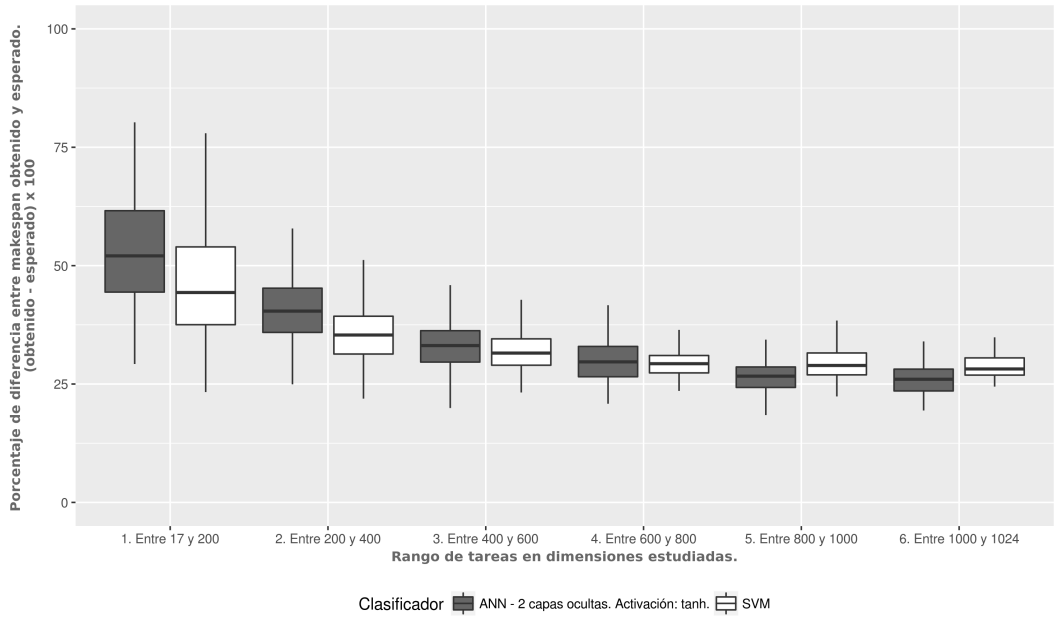

Figura 6.10: Comparación de la diferencia porcentual de *makespan* para la red neuronal con activación *tanh*, de dos capas ocultas con respecto a los valores esperados obtenidos con el algoritmo Min-Min. Así también se muestran los resultados obtenidos para la SVM. Los resultados se muestran divididos en rangos de dimensión desde 17×16 a 1024×16

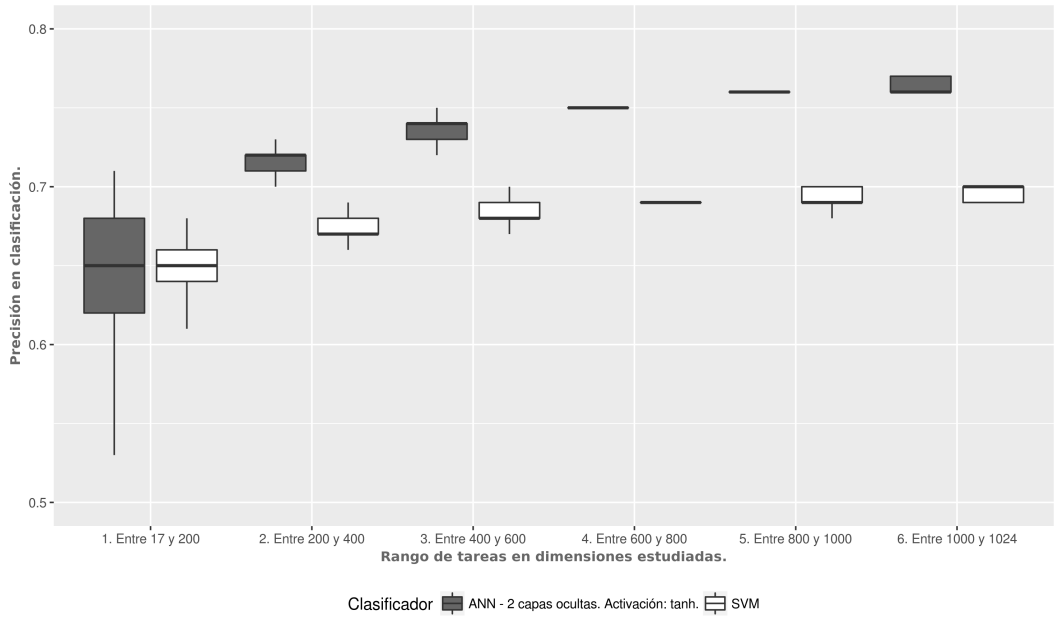

Figura 6.11: Precisión en clasificación para la red neuronal con función de activación *tanh* y para la SVM. Los resultados se muestran divididos en rangos de dimensión desde 17×16 a 1024×16.

La Figura 6.12 muestra que la red neuronal con función de activación *tanh* selecciona máquinas más lentas que SVM cuando se selecciona una máquina diferente a la esperada.

Para dimensiones grandes se selecciona alrededor de un 15 % de mejores máquinas para la función de activación *tanh*, siendo este porcentaje el más bajo obtenido para todas las funciones de activación estudiadas. Esto conduce a pensar que la precisión es fundamental para aproximarse al *makespan* esperado y que las decisiones que se toman a la hora de seleccionar una máquina diferente a la esperada pierde importancia frente a una precisión elevada en clasificación, donde un error puede costar caro en términos de tiempos de ejecución o *makespan*.

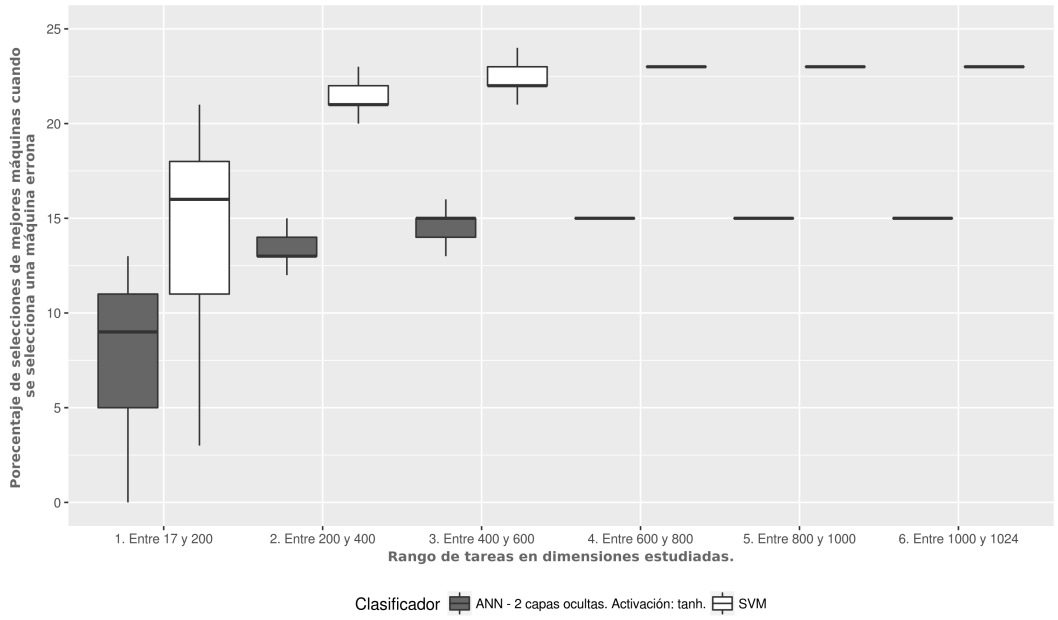

Figura 6.12: Porcentaje de selección de máquinas mejores frente a una selección diferente a la esperada para la red neuronal con activación *tanh* de dos capas ocultas y para la SVM.

6.6. Observaciones generales

En términos generales, entrenar redes neuronales con menor cantidad de capas ocultas, generó mejores resultados de *makespan* en clasificación, que entrenar redes neuronales con mayor cantidad de capas ocultas. Esto se traduce en mejores resultados de *makespan* con una menor inversión en tiempo de entrenamiento.

Para redes neuronales entrenadas utilizando *tanh* e *identity* como funciones de activación, el *makespan* de los resultados mejoró levemente con respecto al *makespan* de los resultados obtenidos con SVM para dimensiones grandes del problema. Se observó una relación directa entre la precisión y la mejora porcentual de *makespan* para la red neuronal con activación *tanh*. También se observó una relación entre el porcentaje de selección de mejores máquinas frente a un error y la mejora porcentual de *makespan*, donde una buena selección de máquina frente a un error en la clasificación, puede mitigar los efectos negativos sobre el *makespan* que genera tener una precisión baja. La red neuronal de dos capas ocultas con función de activación *tanh* obtuvo resultados similares en cuanto a *makespan* que la red neuronal de dos capas ocultas con función de activación *identity*, pero la primera obtuvo mejores valores de precisión en clasificación, lo cual hace de la red neuronal con activación *tanh* una configuración de red neuronal potencialmente adecuada para aprender a resolver el problema HCSP.

Los resultados obtenidos con la red neuronal con función de activación *relu* no mejoraron los resultados obtenidos con SVM en términos de *makespan*, obteniendo precisiones en clasificación similares, teniendo un porcentaje menor de selección de mejores máquinas frente a un error. Esto hace que *relu* sea una función de activación poco adecuada si el objetivo perseguido es aprender a resolver el problema HCSP.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones del trabajo realizado en este proyecto de grado y las principales líneas de trabajo futuro.

7.1. Conclusiones

En este proyecto se presentó un estudio comparativo entre dos tipos de clasificadores de aprendizaje automático, SVM y redes neuronales, en el contexto de aprender a resolver el problema *HCSP*, bajo el paradigma Savant Virtual. Se entrenaron diferentes configuraciones de redes neuronales y de SVM, y se compararon las soluciones obtenidas por cada uno de los clasificadores en términos de su *makespan*, precisión y decisiones de selección de máquinas frente a asignaciones erróneas. Las redes neuronales fueron entrenadas utilizando distintas funciones de activación, así como también variando la cantidad de capas ocultas, manteniendo los demás parámetros de configuración constantes. De esta manera, se generaron 9 redes neuronales de 2, 3 y 4 capas ocultas para las funciones de activación *tanh*, *relu* e *identity* respectivamente. Fueron utilizadas para el entrenamiento 100 instancias del problema de 512 tareas y 16 máquinas, lo que se traduce en 51200 instancias de entrenamiento. Se profundizó el estudio para aquellas redes neuronales que mostraron mejores resultados en cuanto al *makespan* obtenido.

El análisis experimental fue realizado clasificando instancias del problema de diferentes dimensiones, desde 17 tareas y 16 máquinas hasta 1024 tareas y 16 máquinas, con el fin de analizar el comportamiento de los clasificadores

para instancias más pequeñas, de igual y mayor tamaño en comparación a los datos utilizados durante el entrenamiento, de 512 tareas y 16 máquinas. Para cada dimensión del problema se utilizaron 10 instancias del problema como instancias de validación y se calculó el *makespan* obtenido mediante la clasificación con las redes neuronales y SVM, así como la precisión y el porcentaje de selección de máquinas más rápidas frente a un error para el promedio de las 10 instancias del problema.

Los resultados experimentales muestran que las redes neuronales de 2 capas ocultas generaron soluciones con un menor *makespan* que aquellas con 3 y 4 capas ocultas; esto es una característica común a todas las redes neuronales utilizadas sin importar su función de activación. Además, el *makespan* tiende a ser menos variable para estas redes neuronales, lo cual es de importancia dado que el *makespan* se entiende como la métrica fundamental del éxito de una solución generada para el problema.

En comparación con SVM, las redes neuronales de dos capas ocultas con funciones de activación *tanh* e *identity* mostraron mejoras en *makespan* para dimensiones grandes. Para dimensiones a partir de 400 tareas y 16 máquinas se comenzó a observar mejoras en el *makespan* para los resultados obtenidos con ambas redes neuronales. En particular, la red neuronal de 2 capas ocultas con activación *tanh*, mejoró el *makespan* para dimensiones grandes, teniendo una mejor precisión en clasificación que SVM y teniendo un porcentaje de selección más bajo de máquinas más rápidas frente a errores que SVM. En cuanto a la red neuronal de dos capas ocultas con función de activación *identity* se observa que la precisión en clasificación es muy similar a la precisión en clasificación de SVM, con menos variabilidad. El porcentaje de selección de máquinas más rápidas para la red neuronal con activación *identity* también presentó poca variabilidad, también mostrando leves mejoras en el *makespan* de las soluciones. Para las redes neuronales de dos capas ocultas entrenadas con la función de activación *relu* los resultados en cuanto a *makespan* no mejoraron el *makespan* obtenido por SVM. Para las redes neuronales con activación *relu*, la precisión en clasificación resultó levemente mayor que la precisión en clasificación de SVM, teniendo un porcentaje de selección de mejores máquinas frente a un error menor que el de SVM.

En este trabajo se extendió el trabajo original de SV, se probaron nuevos clasificadores y se mejoraron los resultados.

7.2. Trabajo futuro

A continuación se detallan las principales líneas de trabajo futuro que surgen a raíz del trabajo realizado para este proyecto de grado.

Las instancias del problema *HCSP* utilizadas para el entrenamiento y prueba de los clasificadores tienen como característica una baja heterogeneidad de tareas y de máquinas. En instancias reales del problema, lo habitual es tener tareas heterogéneas ejecutando en máquinas de características homogéneas o heterogéneas. Por este motivo, una de las principales líneas de trabajo futuro consiste en entrenar y evaluar clasificadores utilizando instancias del problema de mayor heterogeneidad en tareas y máquinas. Este estudio permitiría analizar si el comportamiento de aquellas configuraciones de redes neuronales que durante este trabajo mostraron resultados más prometedores es generalizable a instancias reales del problema. También es de interés probar distintas configuraciones de hiperparámetros de las redes neuronales, ya que durante este trabajo se analizaron redes neuronales entrenadas con diferentes arquitecturas y funciones de activación, dejando el resto de los parámetros de configuración con sus valores predeterminados. Además, es de interés realizar pruebas con otros clasificadores de aprendizaje automático realizando estudios comparativos con los resultados ya obtenidos.

Finalmente, se considera necesario extender este estudio comparativo aplicando búsquedas locales tanto a las soluciones generadas con redes neuronales como a aquellas generadas por SVM, para evaluar el efecto que esto pudiera tener en la calidad de las mismas, ya sea en el marco de una implementación de MapReduce o no, siendo esta la propuesta fundamental del paradigma SV.

Referencias bibliográficas

[1] Bernabé Dorronsoro, Frédéric Pinel, y Samee U. Khan. Savant: Automatic parallelization of a scheduling heuristic with machine learning. *World Congress on Nature and Biologically Inspired Computing*, 2013.

[2] Sergio Nesmachnow. Parallel evolutionary algorithms for scheduling on heterogeneous computing and grid environments. 2010.

[3] Hesham El-Rewini, Theodore G. Lewis, y Hesham H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. ISBN 0-13-099235-6.

[4] J.Y.T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC Computer and Information Science Series. CRC Press, 2004. ISBN 9780203489802. URL `https://books.google.com.uy/books?id=MAY1ZstmGPkC`.

[5] Richard Freund, Vaidy Sunderam, Allan Gottlieb, Kai Hwang, y Sartaj Sahni. *J. Parallel Distrib. Comput.*, 21(3), 1994. ISSN 0743-7315.

[6] M.M. Eshaghian. *Heterogeneous Computing*. Artech House computer science library. Artech House, 1996. ISBN 9780890065525. URL `https://books.google.com.uy/books?id=LNlQAAAAMAAJ`.

[7] Michael R. Garey y David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[8] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Sahra Ali, y Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proceedings of the 9th Heterogeneous Computing Workshop*, HCW '00, pages 185–, Washington, DC, USA, 2000. IEEE Computer

Society. ISBN 0-7695-0556-2. URL http://dl.acm.org/citation.cfm?id=795691.797919.

[9] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, y Richard F Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837, 2001.

[10] T.M. Mitchell. Machine learning. *McGraw-Hill international editions - computer science series*, 1997.

[11] Bernabé Dorronsoro, Renzo Massobrio, Sergio Nesmachnow, Francisco Palomo-Lozano, y Frédéric Pinel. Generación automática de programas: Savant virtual para el problema de la mochila. *XI Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*, 2016.

[12] Sotiris Kotsiantis. Supervised machine learning: A review of classification techniques. *informatica.si 31 249-268*, 2007.

[13] Girish Chandrashekar y Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40, 2014.

[14] Samina Khalid, Tehmina Khalil, y Shamila Nasreen. A survey of feature selection and feature extraction techniques in machine learning. *Proceedings of 2014 Science and Information Conference*, 2014.

[15] Jeffrey Dean y Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51, 2008.

[16] Pandas: Python data analysis library. https://pandas.pydata.org/, Visitado el 07/10/2018.

[17] pandas.dataframe: two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes. https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html, Visitado el 2/10/2018.

[18] Christian Szegedy Sergey. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[19] 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, y E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[20] sklearn.preprocessing.standardscaler: Standardize features by removing the mean and scaling to unit variance. `http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`, Visitado el 2/10/2018.

[21] Hobson Lane. How to choose the number of hidden layers and nodes in a feedforward neural network? 2017. `https://stats.stackexchange.com/q/136542`, Visitado el 21/05/2018.

[22] aws-ec2. `https://aws.amazon.com/es/ec2/`, Visitado el 5/10/2018.

[23] Numpy: Numpy is the fundamental package for scientific computing with python. `http://www.numpy.org/`, Visitado el 3/10/2018.

[24] Scikit-learn model persistence. `http://scikit-learn.org/stable/modules/model_persistence.html`, Visitado el 4/10/2018.