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
Setiembre 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 Ingeniería en Computación.

Directores:

D.Sc. Prof. Sergio Nesmachnow

D.Sc. Prof. Renzo Massobrio

Montevideo – Uruguay

Setiembre 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.

VI, 44 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. 44 – 44.

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

Montevideo – Uruguay
Setiembre de 2018

# Tabla de contenidos

# Capítulo 1

# Introducción

El Savant Virtual o SV es un paradigma que mediante la aplicación de técnicas de aprendizaje automático aprende el comportamiento de un algoritmo conocido que resuelve un problema y genera automáticamente un nuevo programa que resuelve nuevas instancias del problema de manera aproximada al programa original, incluso para problemas de tamaño diferente. SV aprende de las heurísticas que tradicionalmente resuelven estos problemas de interés, pudiendo hacer un uso más eficiente de los recursos al posibilitarse 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 espacios de soluciones. En este sentido, el enfoque de SV tiene como objetivo obtener soluciones 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 aplicabilidad de las redes neuronales como clasificador 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. Para esto, se estudia su rendimiento en el marco del estudio del problema *Heterogeneous Computing Scheduling Problem*, 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, encontrando me-

joras en las medidas de desempeño seleccionadas para grandes dimensiones del problema.

El resto del trabajo se estructura como se explica a continuación. En el Capítulo 2 se hace una presentación del problema *Heterogeneous Computing Scheduling Problem* y la heurística de referencia, el Capítulo 3 presenta marco teórico dentro del cual se trabajará, enfocado en aprendizaje automático y los dos algoritmos que se comparan durante el trabajo. El Capítulo 4 aborda los trabajos relacionados, donde se presentan aquellos trabajos relacionados a SV así como técnicas de clasificación y selección de atributos. El Capítulo 4.6 presenta la implementación de software que se utiliza para construir los experimentos, dejando para el Capítulo 5 el análisis experimental. Por último, el Capítulo 6 presenta las conclusiones y trabajo a futuro.

# Capítulo 2

# Descripción del problema

Este capítulo presenta el problema abordado en el marco de este proyecto y el paradigma de Savant Virtual, conocido como *Heterogeneous Computing Scheduling Problem* o HCSP, que consiste en generar una planificación para la ejecución de tareas en un conjunto determinado de máquinas de manera óptima. En la sección 2.1 se presenta la formulación del problema, mientras que en la sección 2.2 se presenta la heurística de referencia utilizada a la hora de llevar a cabo el aprendizaje automático.

## 2.1. Formulación del problema

En un contexto donde el poder de cómputo se ve incrementado constantemente y la comercialización de computadores de bajo costo es común, se hace posible la utilización de componentes eventualmente heterogéneos interconectados en sistemas distribuidos para la resolución de problemas grandes que no podían ser atacados en el pasado. Esto da lugar a nuevos problemas con los cuales lidiar, uno de los cuales es la generación de una planificación de ejecución frente a una colección de tareas potencialmente heterogéneas; es decir asignar de manera eficiente una máquina a cada tarea.

En el estudio realizado sobre este problema en este proyecto de grado, se considera a una tarea como una unidad atómica de trabajo que puede ser asignada a un recurso computacional. El hecho de que las máquinas disponibles sean diferentes entre sí en términos de prestaciones, genera una suerte de competencia entre las tareas por "elegir" a aquella máquina más apta y con mayor

rendimiento. La única característica que se toma en cuenta sobre una tarea es su tiempo de ejecución, por cuestiones de simpleza. 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 sea mínimo; este tiempo 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 $M = \{m_1, m_2, \ldots, m_Y\}$. Además, dada una función de tiempo de ejecución $ET : \{1, \ldots, X\} \times \{1, \ldots, Y\} \rightarrow R^+$ tal que $ET(i, j)$ es el tiempo requerido para ejecutar la tarea $t_i$ en la máquina $m_j$, se puede construir una matriz con la información del tiempo de ejecución de cada tarea para cada máquina, conocida como matriz *ETC* (del inglés *Expected Time to Compute*). Esta matriz constituye la representación del problema utilizada como entrada para el sistema utilizado en este proyecto de grado.

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

## 2.2. Heurística de referencia

En el algoritmo 1, como se puede ver en Nesmachnow [8], se presenta una versión genérica de una heurística generadora de planificaciones frente al problema de HCSP. De manera particular, la heurística utilizada como referencia en este proyecto emplea a Min-Min como criterio para seleccionar tareas en cada iteración del algoritmo. Según este criterio, se escoge de manera *greedy* a la tarea que pueda ser completada primero. Para seleccionar esta tarea, se calcula el mínimo tiempo que cada una de las tareas sin asignar puede demorar en completar para cada una de las máquinas disponibles y se escoge a aquella tarea que en promedio lleve menos tiempo en completar.

Adicionalmente, para generar instancias del problema en forma de matrices ETC, se utilizó un programa también extraído de Nesmachnow [8].

**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 presentarán los conceptos fundamentales que apoyan al desarrollo y análisis experimental llevado a cabo en este proyecto de grado, involucrando principalmente conceptos de aprendizaje automático.

## 3.1. Aprendizaje automático

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, se dice que estos programas *aprenden* con la experiencia. En términos formales, el concepto de que un programa *aprenda* es puesto en palabras por Mitchell [7] de la siguiente mantera: *"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 aprendizaje supervisado las entradas del algoritmo son ejemplos de experiencias (conocidos como datos de entrenamiento) y sus respectivas salidas, 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* , así como las *máquinas de soporte vectorial* o los *árboles de*

*decisión*, entre otros. En algoritmos no supervisados no se ofrecen las salidas de los datos de entrenamiento, dejando a los algoritmos detectar patrones en los datos por sí mismo, el algoritmo *K-Means* es un ejemplo de este tipo de aprendizaje automático.

También podemos encontrar algoritmos de aprendizaje por refuerzos. Dentro del aprendizaje automático, el aprendizaje por refuerzos se ocupa de estudiar cómo agentes de software toman acciones en su entorno en función de maximizar una recompensa. Un ejemplo de este tipo de algoritmos es *Q-Learning*, que implica el aprendizaje de una política que le indique al agente qué decisión tomar bajo cuáles circunstancias.

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 flores mediante imágenes. 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 una propiedad dado un conjunto de características asociadas a la misma.

Las técnicas basadas en aprendizaje automático han tomando mayor relevancia en los últimos años debido al crecimiento de los volumenes y variedades de datos, disminución de los costos de infraestructura, acompañado por un crecimiento en capacidad computacional del hardware y capacidad de almacenamiento a precios accesibles.

En este proyecto trabajaremos con dos algoritmos de aprendizaje automático conocidos como *máquina de soporte vectorial* y *redes neuronales*, en un problema de clasificación.

## 3.2. Clasificadores de interés

En el marco del aprendizaje automático supervisado, existen dos tipos de clasificadores de interés para este proyecto de grado. Estos son las redes neuronales y las máquinas de soporte vectorial (o SVM).

### 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 parte por la observación de los mecanismos de aprendizaje en sistemas biológicos. Las redes neuronales artificiales están formadas por unidades más simples e interconectadas, donde cada unidad tiene entradas reales y produce una única salida también real mediante asignación de pesos en regresiones lineales.

Los perceptrones son las unidades sobre las cuales están construidos los sistemas de redes neuronales, por sí solos únicamente puede expresar decisiones lineales, pero su composición en redes neuronales multicapa puede expresar una variedad de funciones objetivo no lineales. Un perceptrón recibe valores de $n$ entradas $x_1, x_2, \ldots x_n$, realiza una combinación lineal con ellas obteniendo una expresión $x_1w_1 + x_2w_2 + \ldots x_nw_n$ 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 se supera un umbral determinado o no con la combinación lineal de los valores de entrada. Por ejemplo, si se utiliza a 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 dicho perceptrón.

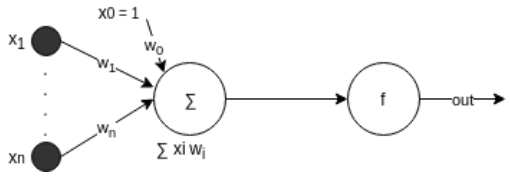


**Figura 3.1:** Perceptron con función de activación $f = signo(x)$

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

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 menera que las salidas sean 1 y -1. El hecho de que los perceptrones puedan 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 multiples perceptrones y las salidas de estos son conectadas a las siguientes capas. En general es de interés el estudio de redes multicapa de unidades, dado que de esta manera se pueden representar grandes variedades de funciones.

Los perceptrones son un tipo de unidades que pueden ser utilizados en la creación de redes multicapa. En particular, múltiples capas de perceptrones dada su naturaleza de unidades lineales, producirán funciones lineales. Lo que se requiere son unidades cuyas salidas sean funciones no lineales de sus entradas. Una alternativa viable es la *unidad sigmoide* cuya estructura es similar a la del perceptrón, variando en la función de activación, utilizando la función *sigmoide* que es una función no lineal diferenciable.

Dada la estructura de la unidad, el aprendizaje se traduce en el problema de encontrar un vector de pesos $(w_0, w_1, \ldots, w_n)$ tal que haga que la salida de la unidad sea la esperada. Dado que en este trabajo se utilizan redes multicapa, se utiliza el algoritmo *Backpropagation* para *aprender* los vectores de pesos de una red que tiene una cantidad dada fija de unidades interconectadas. Antes de explicar la naturaleza de el algoritmo *Backpropagation* y cómo este ajusta de manera iterativa los pesos de las unidades, es importante tener en cuenta el concepto de *gradiante descendente*. *Gradiante descendente* es una técnica que busca encontrar un mínimo de una función, ya sea local o no. La búsqueda por *gradiante 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, comenzando por un vector de pesos inicial arbitrario, modificándolo en pequeños pasos, realizando cada modificación de tal manera que se produce 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. *Gradiante descendente* es una estrategia de búsqueda en un espacio grande o infinito de hipótesis que se puede aplicar siempre que el

espacio de hipótesis contenga hipótesis que pueden 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 se garantiza que la búsqueda converga a un mínimo global.

El algoritmo *Backpropagation* es un método que se utiliza en redes multicapa para calcular el gradiante que se utiliza para el cálculo, utilizando *gradiante descendete*, de los pesos de la red. El error es calculado en la salida de la red multicapa y el mismo es distribuído *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 *loop* 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 condiciones de parada como, detener el *loop* luego de una cantidad fija de iteraciones o, detener el *loop* luego de que el error pasa por debajo de cierto humbral 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 alto y, 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 prueba. *Backpropagation* no asegura la convergencia a un mínimo global debido al uso *gradiante 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 lleven a un mínimo global también pudiendo alcanzar mínimos para una o más dimensiones, que no son necesariamente mínimos para las otras dimensiones. A pesar de la falta de seguridad con respecto a la convergencia del algoritmo a un mínimo global, *Backpropagation* es un método de aproximación de funciones altamente efectivo. Una propiedad importante este es la habilidad de descubrir representaciones de los datos no tribiales en las capas intermedias escondidas de la red, generando así características intermedias más allá de los datos de entrada y la salida proporsionada por los datos de entrenamiento, reafirmando la

capacidad de las redes neuronales de *aprender* patrones en grandes cantidades de datos que no son aprendidos por otros métodos.

En este trabajo se construyen varios tipos de redes que varían en las funciones de activación que usan 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 y 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. 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 serán no lineales, pero por su forma, puede generar que algunas neuronas den como resultado cero, constituyéndose una eventual pérdida de información. Este problema se llama *dying relu problem*. Así también, computar relu es menos costoso que computar tanh porque implica operaciones matemáticas más simples. 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 encuentran el hecho de que se adecuan correctamente a problemas en los cuales los datos de entrenamiento contienen ruido y en contextos en los cuales tiempos largos de entrenamiento son aceptables. Además, suelen mantener tiempos bajos en clasificación. Además, 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 SVM es un algoritmo o clasificador 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 los divida de manera lineal en esas $N$ clases. De esta manera, frente a una nueva instancia del problema, la SVM será 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 y esta denominación se refiere a que sin su presencia el hiperplano podría no ser el mismo, ya que podría utilizarse uno más óptimo. Esta 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 confianza en la clasificación de una instancia del problema, por tanto, estará dada por la distancia de dicho instancia al hiperplano de manera directamente proporcional. La obtención de un hiperplano que divida de manera lineal a los ejemplos de entrenamiento no siempre es posible y se suele utilizar una técnica conocida como *kernelización* cuando esto sucede, 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.

Entre las ventajas de utilizar SVM se encuentran que funciona bien en conjuntos de datos pequeños y limpios (con datos linealmente separables), resultando eficiente ya que utiliza un subconjunto de los ejemplos de entrenamiento únicamente, asociados a la definición del hiperplano. Las desventajas incluyen la poca efectividad que tienen en conjuntos de datos con ruido o clases dife-

renciadas de manera poco clara y que los tiempos de entrenamiento asociados pueden ser significativos para grandes conjuntos de entrenamiento.

# Capítulo 4

# Trabajos relacionados

En este capítulo se describen los principales trabajos relacionados a la temática abordada en este proyecto. El paradigma de SV ha sido explorado únicamente en dos trabajos [4] [3], 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 de 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 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. [4] introdujeron el paradigma de 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 de 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 (en particular SVM), un único *reducer* realiza una búsqueda local con el objetivo de mejorar la solución y posteriormente devuelve su solución. El sistema con 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 criterio 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.

La evaluación experimental se realizó sobre un conjunto de instancias de prueba generadas de forma aleatoria 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 (similaridad 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. [4] presenta oportunidades de trabajo a 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. [3] 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. [4], utilizando también SVM como clasificador y mapper. En particular, 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 que habían sido asignados al contenedor. Con respecto al *reducer*, se aplicó una búsqueda local aleatoria 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, basada en eliminar sucesivamente el objeto que aporte el menor beneficio hasta llegar a una solución factible, y corrección ávida por peso, basada 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 res-

pecto 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. [3] 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 [6] presenta una reseña sobre métodos de aprendizaje automático, principalmente de carácter expositivo, que ofrece guías para 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 [6] 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 [1] 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 [1] 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 valor mínimo o *threshold*. 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 [1] 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 considera 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. [5] 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 de 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 característi-

cas. 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. [5] 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 [2] 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 implementación propuesta en Dean et al. (2008), 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 *master-worker*, donde el hilo *master* se encarga de asignar tareas *map* o *reduce* a los hilos *workers*. 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 estos 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 [2] 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 de 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. [4] | 2013 | Se presenta el paradigma Savant para la paralelización de una heurística de planificación con aprendizaje automático utilizando SVM |
| Dorronsoro et al. [3] | 2016 | Estudio del problema de la mochila bajo el paradigma Savant |
| Kotsiantis, S. [6] | 2007 | Compendio de las técnicas de clasificación más utilizadas en aprendizaje automático supervisado |
| Chandrashekar et al. [1] | 2014 | Estudio comparativo de técnicas de selección de atributos en aprendizaje automático |
| Khalid et al. [5] | 2014 | Compendio de técnicas de selección de atributos en aprendizaje automático, con foco en PCA |
| Dean et al. [2] | 2008 | Utilización de MapReduce para procesamiento de grandes volúmenes de datos en clusters |

**Tabla 4.1:** Trabajos relacionados al proyecto de grado

## 4.6. Implementación

En esta sección será presentado el sistema de software construido utilizando Python y la biblioteca de aprendizaje automático scikit-learn.

La arquitectura del sistema está conformada por componentes que tienen interacción directamente con la capa de persistencia, donde se alojan ejemplos de entrenamiento en diversos formatos y clasificadores de aprendizaje automático.

### 4.6.1. Generación de instancias del problema e instancias de entrenamiento

Al trabajar bajo el paradigma de aprendizaje automático se vuelve necesario contar con una manera accesible y automatizable de generar ejemplos de entrenamiento, cada uno constituido por una instancia del problema y su correspondiente solución esperada. Dichos ejemplos son utilizados para entrenar los clasificadores a utilizar. También se generan ejemplos del problema con el objetivo de evaluar el rendimiento de los clasificadores entrenados previamente.

Se desarrolló un componente para realizar a demanda la generación de ejemplos de entrenamiento y de validación. Para cada ejemplo de entrenamiento es generada una instancia del problema haciendo uso de un algoritmo generador de instancias del problema HCSP extraído de Nesmachnow (2010). Posteriormente, es aplicado el algoritmo Min-Min para generar una solución a dicha instancia del problema. Finalmente, son obtenidos dos archivos para cada ejemplo de entrenamiento, conteniendo las instancias de entrenamiento o validación y la solución provista por el algoritmo Min-Min.

Este componente de software, como el resto de los aquí mencionados, fue desarrollado de manera de permitir su ejecución automatizada, algo que 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.

Luego de generados los archivos, se requiere cambiar la estructura de los datos contenidos en ellos para obtener la estructura de datos con la cual los clasificadores se pueden entrenar y validar. Así mismo, se requiere generar un numero importante de instancias del problema para entrenamiento y validación, las cuales serán pares de archivos diferentes. Por este motivo se cuenta con un componente que se encarga de procesar instancias del problema y sus soluciones para convertirlas en un archivo único con un formato CSV procesable por librerías de manejo de datos. De manera adicional, este componente ofrece al usuario la posibilidad de determinar la ubicación de los datos de entrada y la ubicación de los nuevos archivos a generar. Así, este archivo CSV contiene las instancias del problema y sus soluciones organizadas en instancias de entrenamiento o validación para los clasificadores.

### 4.6.2. Generación de clasificadores

Fue necesario implementar un componente dedicado a generar clasificadores de aprendizaje automático con el fin de evaluar su desempeño. Este componente carga datos generados por el componente mencionado en la sección anterior y, utilizando herramientas de *scikit-learn*, se encarga de generar un clasificador de aprendizaje automático, ya sea SVM o una red neuronal.

Desde el punto de vista del entrenamiento de los clasificadores, se optó por utilizar la información de cada tarea en forma individual, sin tomar en cuenta a la matriz ETC del problema en su totalidad, en parte debido a estudios tempranos realizados que arrojaron resultados pobres en términos de precisión y makespan, sumado al precedente dado por en Dorronsoro et al. (2013), investigación en la cual también fue descartada la utilización de la matriz ETC completa. De esta manera, por ejemplo, si se tiene una instancia del problema de 512 máquinas y 16 tareas junto a su solución, esto representa 512 ejemplos de entrenamiento. Esta forma de utilizar a los ejemplos de entrenamiento se traduce en menores tiempos de entrenamiento y además permite la escalabilidad de los clasificadores directamente, al darse la posibilidad de su aplicación a otras dimensiones del problema en términos de tareas, manteniendo fija la cantidad de máquinas. Esto es posible, dado que si se tiene una dimensión del problema de $m \times n$, siendo $m$ la cantidad de tareas y $n$ la cantidad de máquinas, por ejemplo en el caso de una red neuronal, el clasificador entrenado tendrá $n$

neuronas de entrada, por lo que será aplicable a cualquier valor de $m$, ya que un ejemplo de entrenamiento bajo este paradigma siempre tendrá el mismo tamaño. Además, se considera algo realista que exista una mayor variabilidad en la cantidad de tareas y no en la cantidad de máquinas, dado que las máquinas son un recurso rígido con poca volatilidad generalmente.

De manera adicional, en lo que respecta a la arquitectura de las redes, se sigue la heurística recomendada por Lane (2017), dada la falta de consenso existente 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

$\alpha$ = factor de escalamiento arbitrario, con valor igual a 2 para este estudio

Por otra parte, dado que en la generación de instancias del problema los valores de la matriz ETC para dos problemas distintos no necesariamente mantienen una relación normal, se vuelve necesario escalar los datos utilizados a la hora de entrenar o utilizar clasificadores.

Para esto, es utilizado un clasificador de tipo *pipeline*, ofrecido por la librería *scikit-learn*. Este componente de software permite que sean aplicadas una serie de transformaciones a los datos de manera previa a ser utilizados por el clasificador. A su vez, el clasificador puede mantener la configuración que el usuario prefiera, tomando información de la arquitectura deseada para una red neuronal o parámetros de inicialización en caso de optar por utilizar una SVM.

En particular fueron utilizadas las clases *neural_network.MLPClassifier* y *svm.SVC* de *scikit-learn* para construir las redes neuronales y la SVM, respectivamente.

Para encontrar los mejores parámetros para comenzar con las pruebas, fue utilizado *model_selection.GridSearchCV* que permite seleccionar los mejores parámetros para entrenar un modelo.

Finalmente, este componente se encarga de persistir los clasificadores generados, proveyendo una interfaz al usuario para realizar todo lo mencionado de manera automática dado un conjunto de ejemplos de entrenamiento.

### 4.6.3. Clasificación

En lo que respecta a la clasificación, fue implementado un componente encargado de clasificar un conjunto de ejemplos de validación dado uno o más clasificadores ya entrenados y de obtener métricas que se describen en los párrafos siguientes.

Dentro de las métricas calculadas, se encuentran el makespan esperado y el obtenido mediante predicciones. Esta métrica es considerada como fundamental para evaluar el rendimiento de los clasificadores, dado que más allá de la precisión, es necesario evaluar la métrica fundamental del problema HCSP, que es el tiempo insumido por la máquina que finaliza su ejecución por último.

Además, se calcula el porcentaje de ocasiones en las que el clasificador, al predecir erróneamente, escoge una máquina más rápida (constituyendo una acción avariciosa o *greedy*). 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 avariciosa y por lo tanto aprendiendo a maximizar el beneficio local para cada tarea.

# Capítulo 5

# Análisis Experimental

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

## 5.1. 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.

Inspirado en Dorronsoro et al. (2013), se optó por trabajar con clasificadores entrenados con una determinada dimensión del problema (en particular 512 tareas sobre 16 máquinas o $512 \times 16$) a la hora de clasificar instancias del problema de menor, igual y mayor cantidad de tareas. Para hacer esto posible, se generaron instancias del problema de dimensión $512 \times 16$, utilizadas para el entrenamiento de los clasificadores. Así también, para cada dimensión del problema desde $17 \times 16$ hasta $1024 \times 16$, 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 de capas ocultas empleado debido a limitaciones de recursos y tiempos de entrenamiento altos.

Como fue mencionado en el capítulo *Implementación*, para obtener una primera selección de parámetros de configuración para el entrenamiento de las redes neuronales se utilizó *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. Entre los valores de los parámetros que se seleccionaron para utilizar el método *GridSearchCV()* con las redes neuronales, se encuentran: *lbfgs*, *sgd*, *adam* para el parámetro *solver*; *1e-2* y *1e-4* para el coeficiente de aprendizaje llamado *alpha* y *relu*, *tanh* e *identity* como funciones de activación para el parámetro *activation*. Los resultados del método *GridSearch()* para los parámetros mencionados se muestran en el Cuadro 5.1. 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 y estudios tempranos demostraron resultados poco prometedores que, al variar la función de activación, mejoraban notoriamente. Esto llevó a variar las funciones de activación para probar además 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()*.

| **Parámetro** | **Valor** |
|---|---|
| solver | lbfgs |
| alpha | 1e-2 |
| activation | relu |

**Tabla 5.1:** Valores para los parámetros de las redes neuronales, obtenidos mediante el método *GridSearch() de la librería scikit-learn*

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.

## 5.2. 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. Así mismo, en esta comparación también fueron analizados los resultados SVM.

Cada clasificador fue entrenado con 100 instancias del problema de dimensión $512 \times 16$, lo que se traduce en 512000 instancias de entrenamiento para los clasificadores.

Para cada dimensión del rango estudiado, la clasificación fue realizada sobre 10 instancias del problema y los resultados mostrados son un promedio de estas.

Fueron identificados aquellos clasificadores que obtuvieron mejores resultados en *makespan*. Las figuras 5.1, 5.2 y 5.3 muestran las diferencias de *makespan* porcentuales 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 $17 \times 16$ a $1024 \times 16$.

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 último de los conjuntos van desde la dimensión $17 \times 16$ a $200 \times 16$ y $1000 \times 16$ a $1024 \times 16$ respectivamente.

La Figura 5.1 muestra los resultados para las redes neuronales utilizando la función de activación *relu*. En dicha Figura se observa que el *makespan* generado 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 dos capas ocultas.

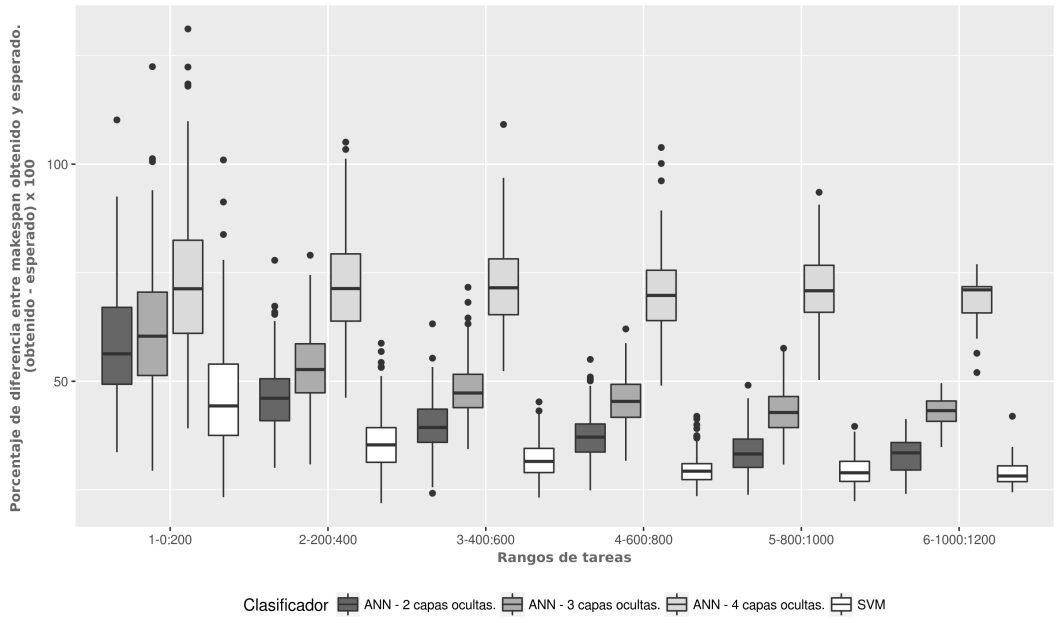


**Figura 5.1:** Comparación de la diferencia porcentual de los resultados de *makespan* para redes neuronales con función de activación *relu*, con respecto al *makespan* obtenido por el algoritmo Min-Min. Se comparan redes neuronales de 2, 3 y 4 capas ocultas. Así también se muestran los resultados obtenidos con el algoritmo de SVM.

La Figura 5.2 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 $200 \times 16$ 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 5.3 se observan los resultados para las redres 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 $400 \times 16$ en adelante.

En los casos presentados en las figuras 5.1, 5.2 y 5.3, los mejores resultados de las redes neuronales se encuentran para aquellas de dos capas ocultas. En éstas fue profundizado el estudio de cara a entender mejor la naturaleza de las soluciones generadas.

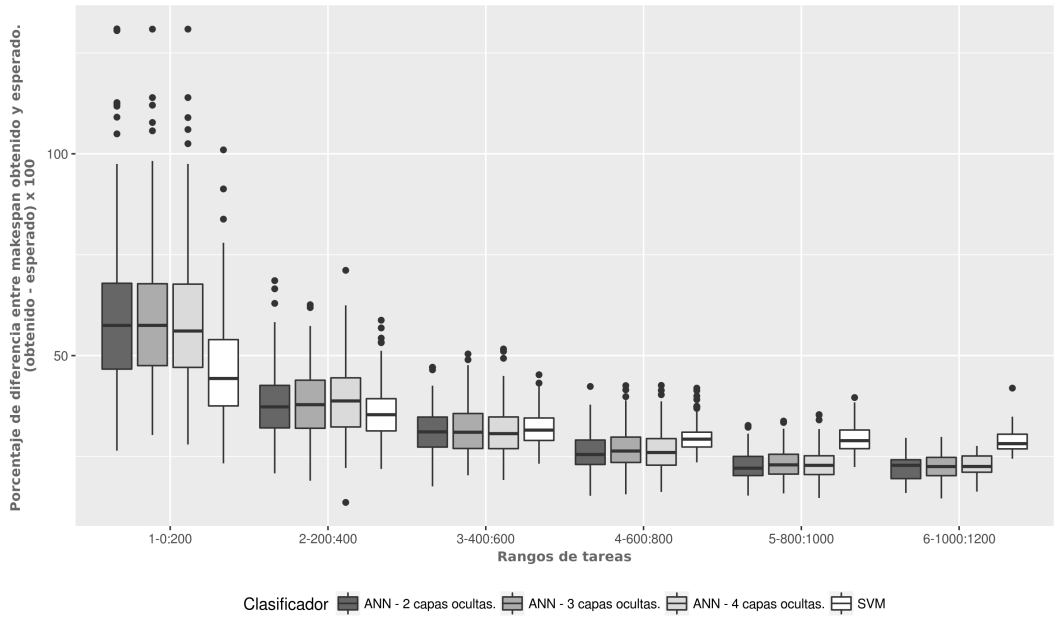


**Figura 5.2:** Comparación de la diferencia porcentual de los resultados de *makespan* para redes neuronales con función de activación *identity*, con respecto al *makespan* obtenido por el algoritmo Min-Min. Se comparan redes neuronales de 2, 3 y 4 capas. Así también se muestran los resultados obtenidos con el algoritmo de SVM.

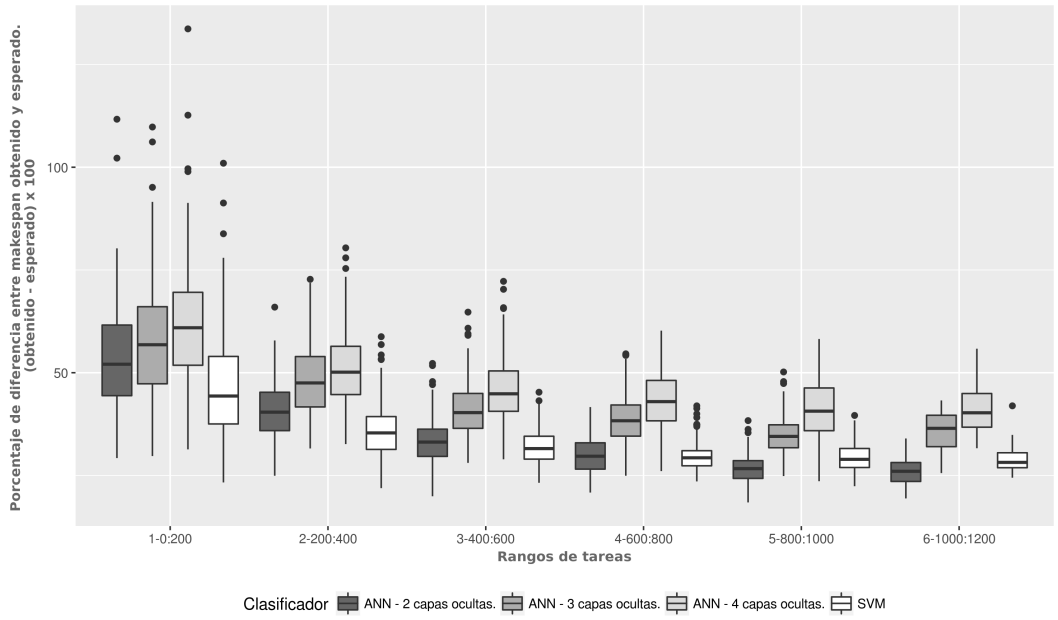


**Figura 5.3:** Comparación de la diferencia porcentual de los resultados de *makespan* para redes neuronales con función de activación *tanh*, con respecto al *makespan* obtenido por el algoritmo Min-Min. Se comparan redes neuronales de 2, 3 y 4 capas. Así también se muestran los resultados obtenidos con el algoritmo de SVM.

## 5.3. Red neuronal con activación *relu* de dos capas ocultas

La Figura 5.4 muestra con más claridad 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 5.5 muestra la precisión de la clasificación para ambos clasificadores. Se observa que la precisión en clasificación aumenta a medida que la dimensión de las instancias de prueba aumenta, tanto para la red neuronal como para la SVM.

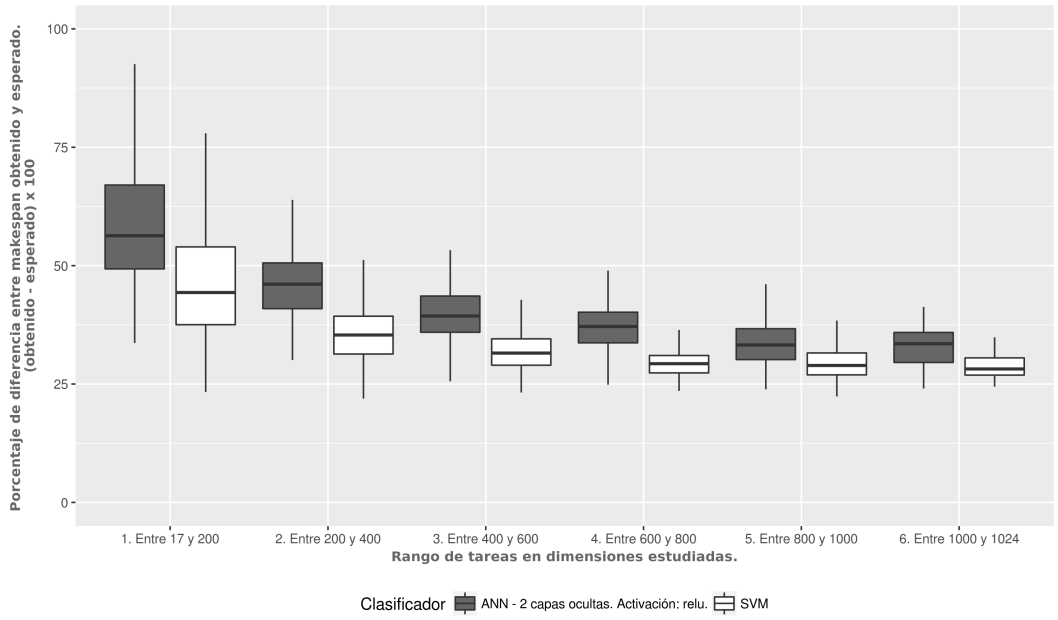


**Figura 5.4:** Comparación de la diferencia porcentual de *makespan* para la red neuronal con activación *relu*, de dos capas ocultas con respecto a los valores esperados obtenidos con el algoritmo Min-Min. Así también se muestran los resultados obtenidos para la SVM. Los resultados se muestran divididos en rangos de dimensión desde $17 \times 16$ a $1024 \times 16$.

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 5.4, 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, fue calculado el porcentaje de selección de mejores máquinas para ambos clasificadores, como se menciona en la sección dedicada a la implementación. La Figura 5.6 muestra dicha métrica para ambos clasificadores. En esta se observa que la SVM elige más máquinas con menor tiempo de ejecución que la red neuronal, cuando se seleccionan máquinas diferentes a las esperadas.

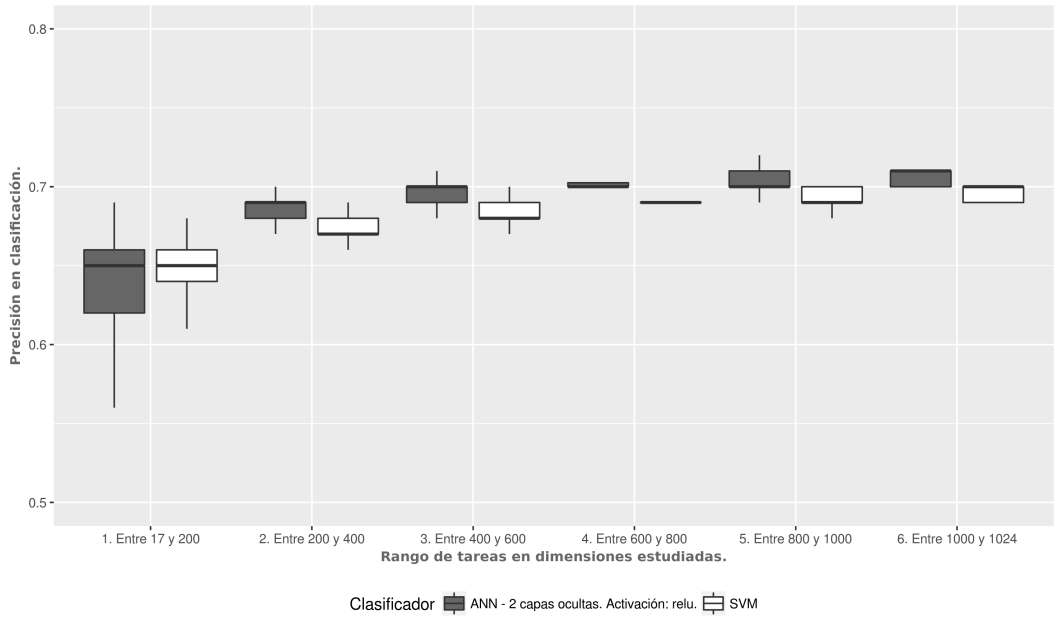


**Figura 5.5:** Precisión en clasificación para la red neuronal con función de activación *relu* de dos capas ocultas y para la SVM. Los resultados se muestran divididos en rangos de dimensión desde $17 \times 16$ a $1024 \times 16$.

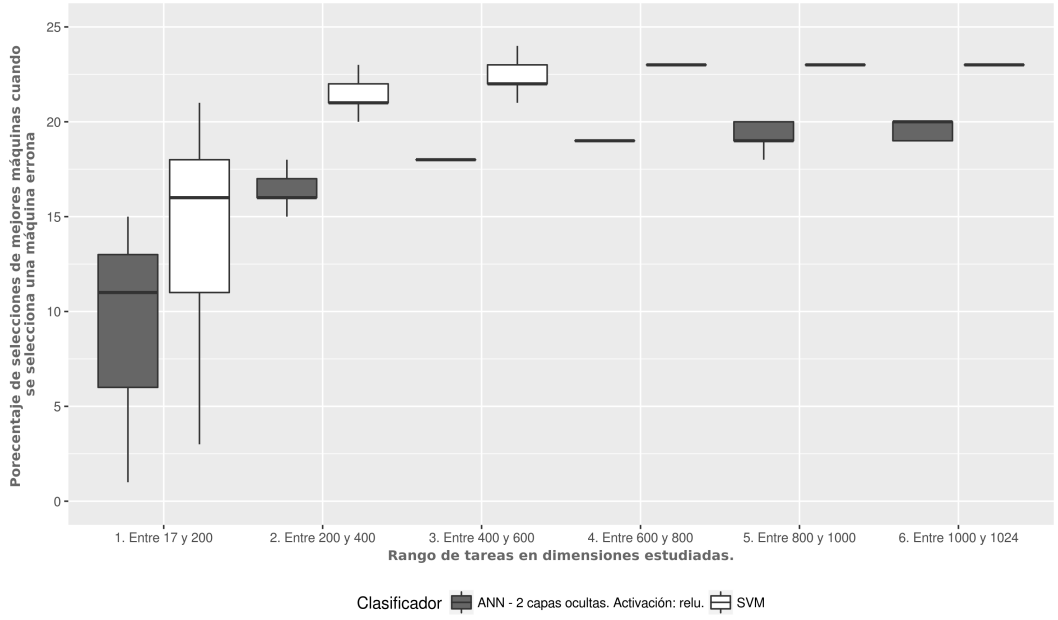


**Figura 5.6:** Porcentaje de selección de máquinas mejores frente a una selección diferente a la esperada para la red neuronal con activación *relu* de dos capas ocultas y para la SVM.

## 5.4. Red neuronal con activación *identity* de dos capas ocultas

La Figura 5.7 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 $400 \times 16$, la red neuronal conduce a levemente mejores valores de *makespan* que la SVM. En este caso, la precisión en la clasificación, que se observa en la Figura 5.8, no tiene diferencias sustanciales.

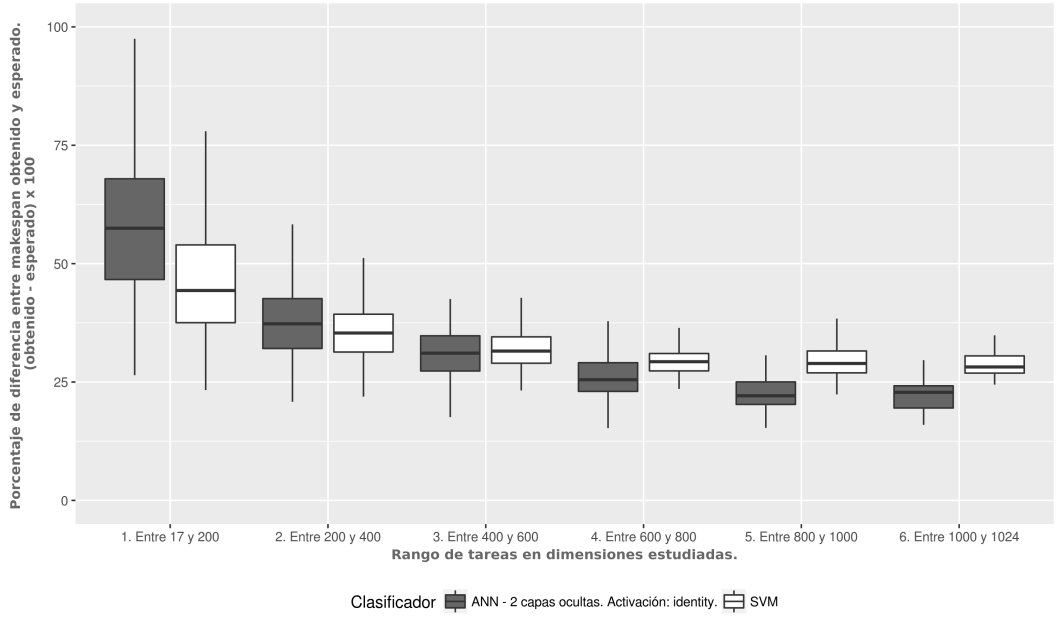


**Figura 5.7:** Comparación de la diferencia porcentual de *makespan* para la red neuronal con activación *identity*, de dos capas ocultas con respecto a los valores esperados obtenidos con el algoritmo Min-Min. Así también se muestran los resultados obtenidos para la SVM. Los resultados se muestran divididos en rangos de dimensión desde $17 \times 16$ a $1024 \times 16$

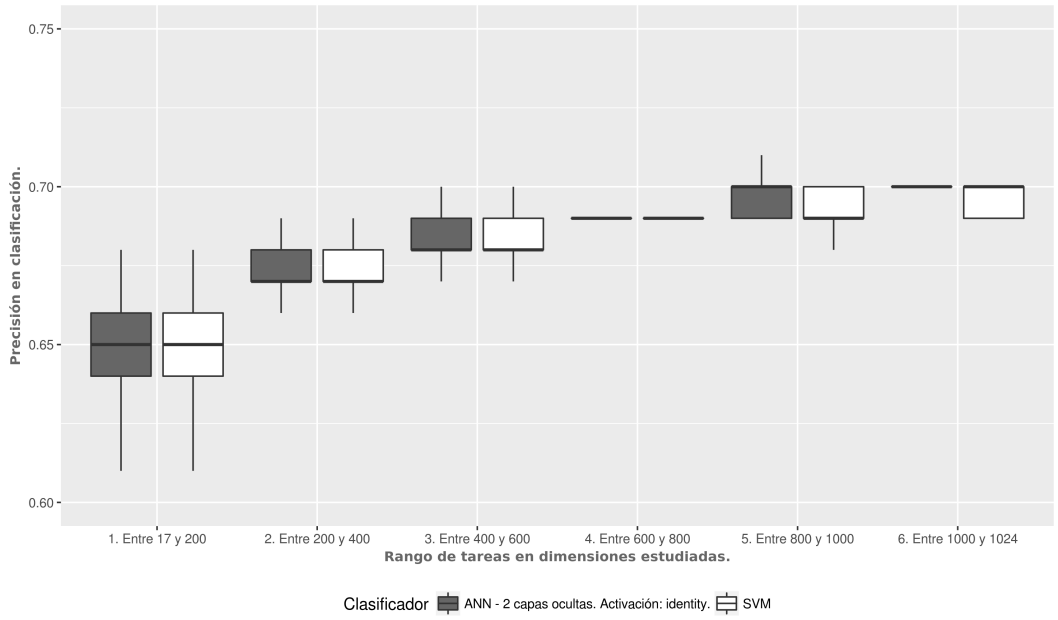


**Figura 5.8:** Precisión en clasificación para la red neuronal con función de activación *identity* y para la SVM. Los resultados se muestran divididos en rangos de dimensión desde $17 \times 16$ a $1024 \times 16$.

Al observar el porcentaje de mejores máquinas seleccionadas frente a un error en la Figura 5.9, se observa que la red neuronal 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*, donde la red neuronal 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*.

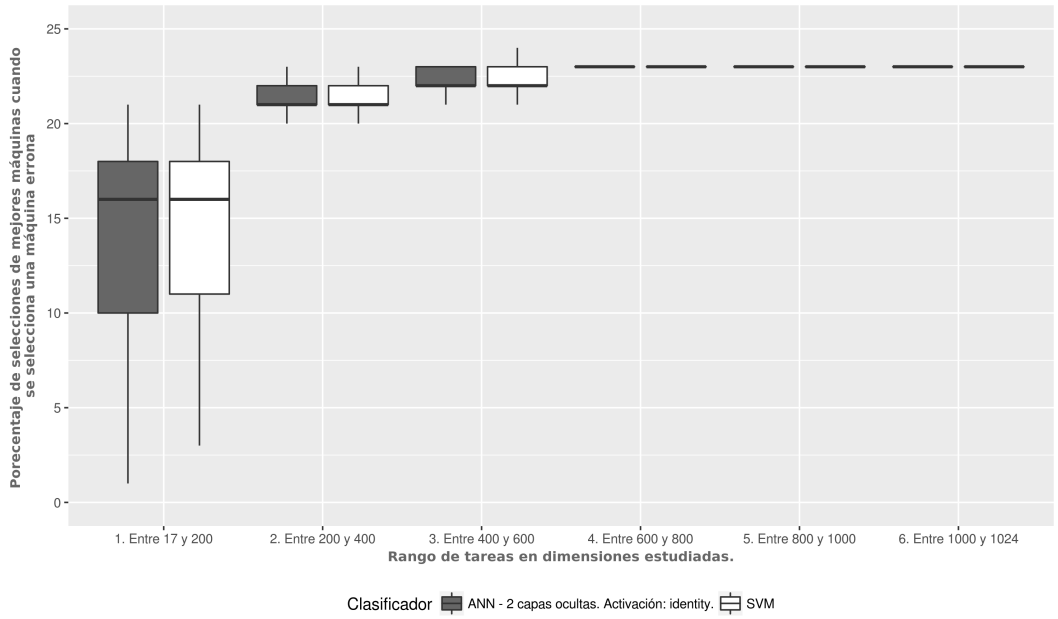


**Figura 5.9:** Porcentaje de selección de máquinas mejores frente a una selección diferente a la esperada para la red neuronal con activación *identity* de dos capas ocultas y para la SVM.

## 5.5. Red neuronal con activación *tanh* de dos capas ocultas

La Figura 5.10 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 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 5.11.

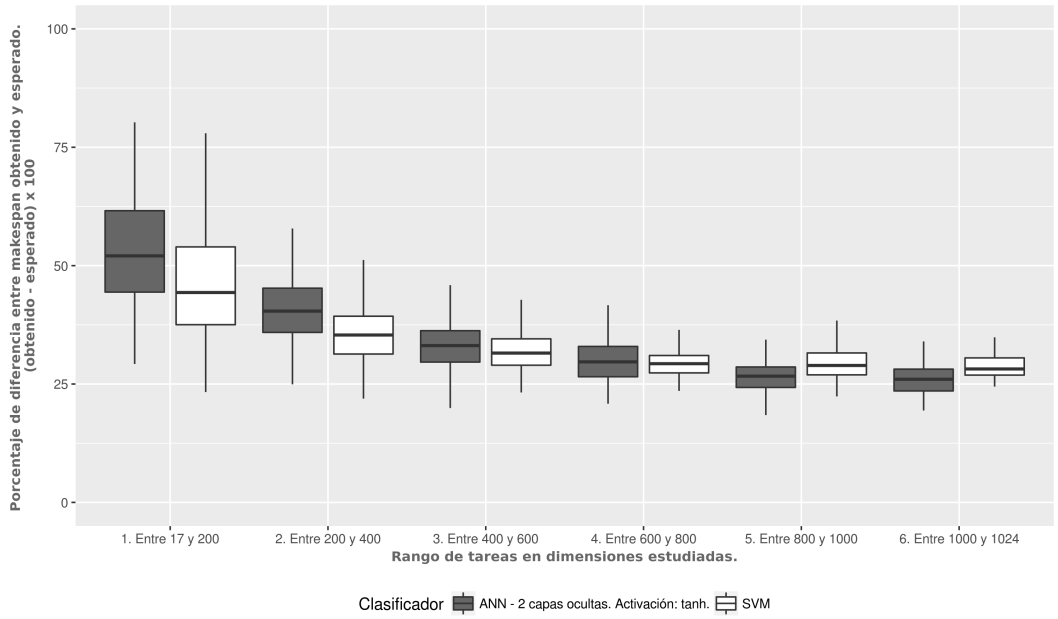


**Figura 5.10:** Comparación de la diferencia porcentual de *makespan* para la red neuronal con activación *tanh*, de dos capas ocultas con respecto a los valores esperados obtenidos con el algoritmo Min-Min. Así también se muestran los resultados obtenidos para la SVM. Los resultados se muestran divididos en rangos de dimensión desde $17 \times 16$ a $1024 \times 16$

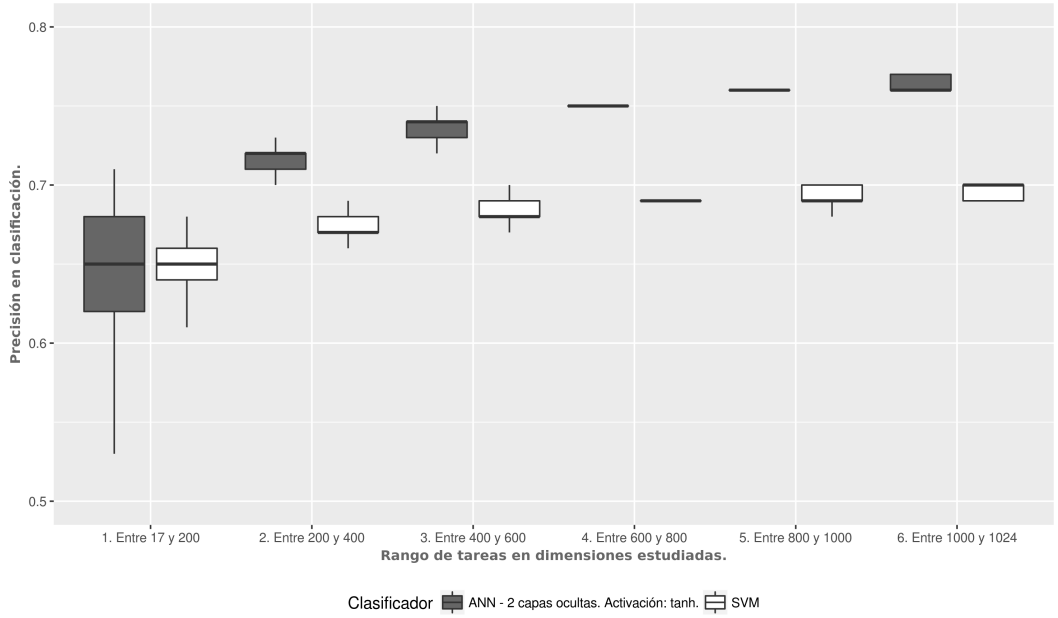


**Figura 5.11:** Precisión en clasificación para la red neuronal con función de activación *tanh* y para la SVM. Los resultados se muestran divididos en rangos de dimensión desde $17 \times 16$ a $1024 \times 16$.

Al estudiar las decisiones que se toman frente a un error en la selección de la máquina a la que una tarea es asignada, se encuentra que la red neuronal que utiliza la función de activación *tanh* selecciona máquinas más lentas que la SVM. Lo antedicho se observa en la Figura 5.12

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

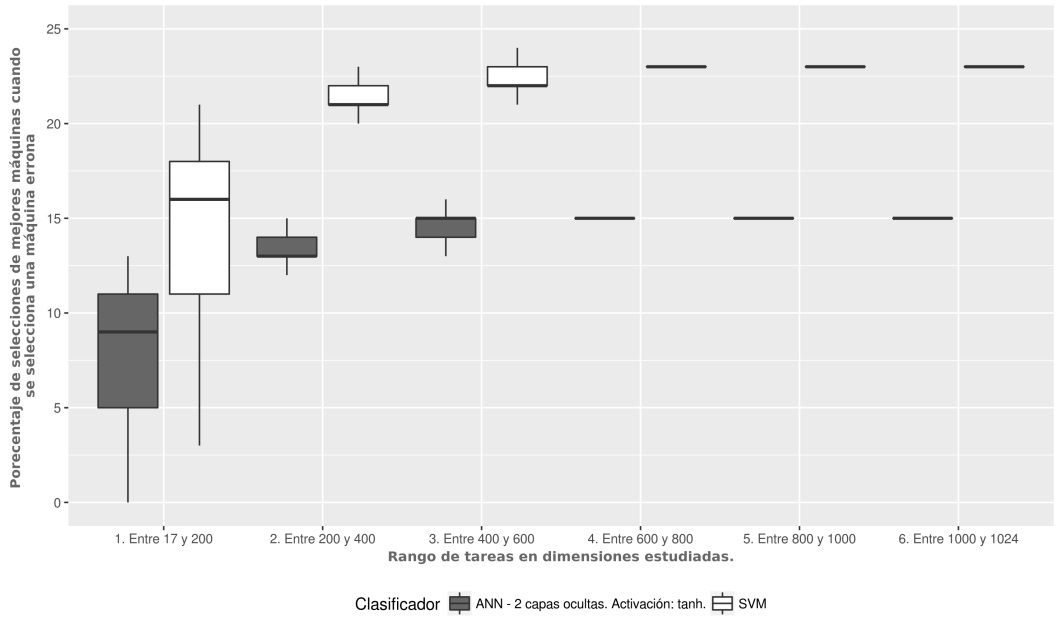


**Figura 5.12:** Porcentaje de selección de máquinas mejores frente a una selección diferente a la esperada para la red neuronal con activación *tanh* de dos capas ocultas y para la SVM.

## 5.6. Observaciones generales

En términos generales, entrenar redes neuronales con menor cantidad de capas ocultas, genera mejores resultados de *makespan* en clasificación, que en-

trenar redes neuronales con mayor cantidad de capas ocultas. Esto se traduce en mejores resultados de *makespan* invirtiendo un menor tiempo de entrenamiento.

Para redes neuronales entrenadas utilizando *tanh* e *identity* como funciones de activación, el *makespan* de los resultados mejora levemente con respecto al *makespan* de los resultados obtenidos con SVM para dimensiones grandes del problema. Se observa que hay una relación directa entre la precisión y la mejora porcentual de *makespan*, así como también entre el porcentaje de selección de mejores máquinas frente a un error y la mejora porcentual de *makespan*.

Los resultados obtenidos con la red neuronal con función de activación *relu* no mejoran 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, haciendo que *relu* sea una función de activación poco adecuada para trabajar con el problema abordado, el *Heterogeneous Computing Scheduling Problem*.

# Capítulo 6

# Conclusiones y trabajo a futuro

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

## 6.1. Conclusiones

En este proyecto se presentó un estudio comparativo entre dos tipos de clasificadores de aprendizaje automático, SVM y redes neuronales, para el problema de *HCSP* en el marco del paradigma de Savant Virtual. Se entrenaron diferentes configuraciones de redes neuronales 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 errores. 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 512000 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, 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. Todo esto es importante 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* muestran mejoras en *makespan* para dimensiones grandes. Para dimensiones de a partir de 400 tareas y 16 máquinas se comienzan 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*, mejora 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, al igual que el porcentaje de selección de máquinas más rápidas, 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 mejoran el *makespan* obtenido por SVM, aunque la precisión en clasificación es 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.

## 6.2. Trabajo a futuro

A continuación se detallan las principales lineas de trabajo a futuro que surgen del trabajo realizado para el 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, lo cual hace difícil la aplicación de dichos clasificadores a la resolución de instancias reales del problema, donde se presume que lo corriente es tener tareas heterogéneas ejecutando en máquinas de características homogéneas o heterogéneas. Dentro de las líneas de trabajo a futuro se incluye el entrenamiento y prueba de clasificadores utilizando instancias del problema de mayor heterogeneidad en tareas y máquinas, siendo de interés analizar el comportamiento de aquellas configuraciones de redes neuronales que durante este trabajo mostraron resultados más prometedores. También es de interés probar distintas configuraciones parámetros 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.

Por otro lado, 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.

Finalmente, también es de interés realizar pruebas con otros clasificadores de aprendizaje automático realizando estudios comparativos con los resultados ya obtenidos para el modelo de SV y eventualmente probar el enfoque de SV para la resolución de problemas reales.

# Referencias bibliográficas


[1] Chandrashekar, G. y Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering*, 40.

[2] Dean, J. y Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51.

[3] Dorronsoro, B., Massobrio, R., Nesmachnow, S., Palomo-Lozano, F., y Pinel, F. (2016). Generación automática de programas: Savant virtual para el problema de la mochila. *XI Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*.

[4] Dorronsoro, B., Pinel, F., y Khan, S. U. (2013). Savant: Automatic parallelization of a scheduling heuristic with machine learning. *2013 World Congress on Nature and Biologically Inspired Computing*.

[5] Khalid, S., Khalil, T., y Nasreen, S. (2014). A survey of feature selection and feature extraction techniques in machine learning. *Proceedings of 2014 Science and Information Conference, SAI 2014*.

[6] Kotsiantis, S. (2007). Supervised machine learning: A review of classification techniques. *informatica.si 31 249-268*.

[7] Mitchell, T. (1997). Machine learning. *McGraw-Hill international editions - computer science series*.

[8] Nesmachnow, S. (2010). Parallel evolutionary algorithms for scheduling on heterogeneous computing and grid environments.