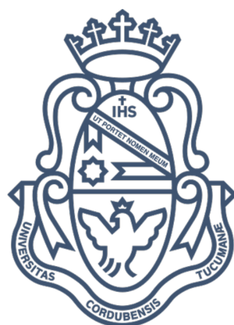
Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía, Física, y Computación



Uso de Planes Relajados en Grounding Heurístico

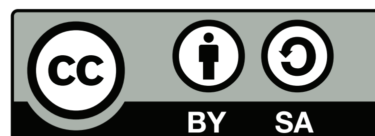
Autor: Nicolás Benjamín Ocampo

Directores: Dr. Carlos Areces y Dr. Martín Dominguez

Trabajo de tesis presentado para obtener el título de Licenciado en Ciencias de la Computación

Córdoba, Argentina Diciembre, 2021

Esta obra está bajo una licencia Creative Commons "Reconocimiento-CompartirIgual 4.0 Internacional".



Resumen

Los planificadores en planning clásico encuentran planes con éxito aún para tareas realmente complejas. Para esto, la mayoría de planificadores realizan una etapa de preprocesamiento sobre la especificación de la tarea para obtener una representación de mucho más bajo nivel de abstracción. Este proceso es conocido como proceso de grounding. Cada vez que la tarea instanciada es demasiado grande para ser generada, la tarea no puede ser resuelta por el planificador. En esta tesis proponemos un proceso alternativo, denominado grounding heurístico, que guía el proceso de grounding, instanciando aquellas partes de la tarea que son relevantes. Para ello, se trabajó sobre modelos de aprendizaje supervisado, planes relajados y codificaciones ad-hoc y por word embeddings.
Palabras claves: Planning, aprendizaje automático, proceso de grounding, codificación one-hot, word embeddings.

Abstract

Planners in classical planning are successful in finding plans, even for complex tasks. To do so, most planners rely on a preprocessing stage that computes a grounded representation of the task. This process is known as the grounding process. However, if the grounded task is too big to be generated, it can not be tackled by the planner. In this thesis, we propose an alternative approach called heuristic grounding. This method, guides the grounding process, instantiating only the parts of the task that are relevant using machine learning techniques, relaxed plans, and ad-hoc and word embeddings encoders.
Keywords: Planning, machine learning, grounding process, one-hot encoding, word embeddings.

Agradecimientos

Esta tesis es fruto del trabajo y colaboración de un montón de personas que compartieron conmigo su grano de arena durante mi formación. A mi familia que me acompañó en mi decisión de mudarme a la ciudad de Córdoba y seguir esta carrera que me ha enseñado tanto. A todos mis hermanos que a pesar de verlos una vez al año por tan solo algunas semanas, logran compartir su cariño que se queda conmigo durante el resto de la temporada. A mis directores, Carlos y Martín que me brindaron su paciencia, tiempo, y conocimiento, haciéndome sentir realmente cómodo y seguro durante el desarrollo de esta tesis. A los profesores y profesoras de FaMAF que me compartieron y contagiaron su pasión por la ciencia. A mis amigos que fueron mi cable a tierra en numerosas ocasiones. Juntarse con ellos, para compartir risas, experiencias, o simplemente matar el rato, es algo que no tiene precio.

¡Muchas gracias a todos!

Índice general

1. Introducción **1**

1.1. Contexto y motivación . . . 1
1.2. Trabajos relacionados . . . 3
1.3. Estructura de la tesis . . . 3

I Fundamentos teóricos **5**

2. Planning Clásico **7**

2.1. ¿Qué se entiende por planning? . . . 7
2.2. Estados y acciones . . . 8
2.3. Tareas STRIPS . . . 8
2.4. Representaciones STRIPS . . . 10
2.5. El lenguaje PDDL . . . 13
2.6. Relajación por deletes . . . 14
2.7. Proceso de grounding . . . 16
2.8. Grounding heurístico . . . 17

3. Aprendizaje automático **19**

3.1. Orígenes y evolución . . . 19
3.2. Aprendizaje supervisado . . . 20
3.3. Aprendizaje no supervisado . . . 20
3.4. Algoritmos de clasificación . . . 21
3.4.1. Modelos lineales . . . 21
3.4.2. Modelos probabilísticos . . . 22
3.4.3. Función de costo . . . 23
3.4.4. Descenso por el gradiente . . . 23
3.4.5. Sobreajuste (overfitting) . . . 24
3.4.6. Regresión logística . . . 25
3.4.7. Redes neuronales . . . 26
3.4.8. XGBoost . . . 30
3.5. Codificación de características . . . 33
3.5.1. One-hot encoding . . . 34
3.5.2. Vectores de palabras (Word embeddings) . . . 35
3.6. Métricas de clasificación . . . 40

II Metodología 45

4. Etapas de preparación y ejecución de modelos 47
4.1. Dominio de planning: Satellite . . . 48
4.1.1. Descripción del dominio . . . 48
4.2. Manejo de datos . . . 50
4.2.1. Generación de tareas de planning . . . 50
4.2.2. Preservación de datos . . . 51
4.2.3. Etiquetado de ejemplos . . . 53
4.2.4. Preprocesamiento . . . 55
4.2.5. Generación de ventanas de planes relajados . . . 58
4.3. Selección de modelos . . . 61
4.3.1. Entrenamiento . . . 61
4.3.2. Evaluación . . . 61
4.4. Registro de resultados . . . 61
4.4.1. Preservación de modelos, métricas, e imágenes . . . 61
4.4.2. Monitoreo y visualización . . . 62
4.4.3. Adaptación de modelos y transformadores . . . 62

5. Experimentos y resultados 65
5.1. Modelos predictivos por word embeddings . . . 66
5.1.1. Configuración del experimento . . . 66
5.1.2. Resultados: take_image . . . 68
5.2. Modelos predictivos ad-hoc . . . 73
5.2.1. Configuración del experimento . . . 73
5.2.2. Resultados . . . 73
5.3. Otros experimentos . . . 75
5.3.1. Modelos end-to-end . . . 76
5.3.2. El problema de inclusión sobre planes relajados . . . 78
5.3.3. Mayorización . . . 78
5.3.4. Orden de ventanas como características . . . 78

III Conclusión 81

6. Conclusiones generales 83
6.1. Reflexión . . . 83
6.2. Trabajo a futuro . . . 84

Capítulo 1

Introducción

1.1. Contexto y motivación

La planificación automática, o simplemente *planning*, es una de las áreas centrales de la inteligencia artificial debido a su extenso uso en dominios, tales como, control de misiones espaciales (Rabideau et al., 2001), manejo de crisis (Bienkowski et al., 1995), generación de textos narrativos (Goudoulakis et al., 2016), o robótica (Muñoz et al., 2016).

El objetivo es definir un modelo que simule una tarea descripta por medio de una especificación que detalle sus componentes. Estas son el estado inicial, la meta, y un conjunto de acciones. El estado inicial describe las propiedades verdaderas iniciales del problema. La meta o estado final representa cuáles son las propiedades que deben ser verdaderas para finalizar la tarea. Y el conjunto de acciones está compuesto de transformadores que alteran si una propiedad pasa a ser verdadera o deja de serlo. Este conjunto de propiedades recibe el nombre de estado, y modelan que propiedades son verdaderas y falsas, y a que estados puede cambiar por medio de alguna acción. Si se obtiene una secuencia de acciones que sea aplicable en el estado inicial, y que luego de su ejecución conlleve a la meta, entonces dicha secuencia es considerada un plan del problema.

Para hallar un plan que resuelva la tarea, se realizan técnicas de búsqueda y optimización, efectuadas por *planificadores*, algoritmos que computan el comportamiento de un agente por medio de una descripción del problema comúnmente definida en el *Planning Domain Definition Language* (PDDL). En PDDL, se especifican las propiedades del entorno en términos de predicados, y las transformaciones por medio de esquemas de acción. Estas consisten en expresiones parametrizadas que pueden ser instanciadas por un conjunto de objetos. El planificador por medio de la especificación, define un espacio de búsqueda sobre el cual encontrar un plan.

Sin embargo, la mayoría de los planificadores trabajan sobre una representación sin variables libres. Por consecuente, estos computan todas las instanciaciones que asignan los objetos a los argumentos de los predicados y esquemas de acción definidos por la especificación en PDDL del problema. Este proceso, conocido como *grounding*, es exponencial en la cantidad de argumentos de los esquemas de acción y predicados, llegando a obtener una cantidad inmensurable de instancias cuando el número de parámetros definidos es alto. Esto puede llevar a la falla por parte del planificador para resolver

la tarea sin antes haber tenido la posibilidad de realizar la búsqueda en el espacio de instancias, incluso cuando en la práctica solo una pequeña fracción de ellas ocurren en los planes del problema. (Gnad et al., 2019)

Ahora bien, si se obtienen las acciones necesarias para confeccionar por lo menos un plan, entonces el proceso de búsqueda encontraría alguna de tales soluciones. Por ende, surge la siguiente pregunta, ¿Cómo determinamos que acciones son relevantes para algún plan de tal manera que puedan ser incluidas en el proceso de grounding?

Esta pregunta es la que nos llevó a considerar el uso de técnicas de *aprendizaje automático*. Los modelos de aprendizaje automático consisten en aproximar una función a partir de datos de entrenamiento, es decir, ejemplos a los cuales se conoce el valor de entrada y de salida de la función. La idea principal se basó en encontrar una función que prediga la probabilidad de que una acción sea relevante a partir de información del problema. Dado que las tareas que nos interesan resolver son aquellas que no pueden ser groundeadas, es necesario recurrir a problemas equivalentes pero más sencillos durante la construcción del material de entrenamiento.

Otro inconveniente es decidir que información del problema se puede brindar al modelo de aprendizaje automático de tal manera que identifique si una acción es importante. Evidentemente, no puede ser el plan de la tarea, ya que para aquellos problemas que fallan durante el proceso de grounding no se puede encontrar el plan que lo resuelve. Debe ser una característica que brinde información, aunque sea aproximada, de la estructura del plan real y que pueda ser computada previamente al proceso de grounding. En (Hoffmann and Nebel, 2001) se estudia el uso de funciones heurísticas definidas a partir de *planes relajados* que permiten guiar el proceso de búsqueda de un plan. Si los planes relajados son capaces de guiar el proceso de búsqueda, entonces también existe una posibilidad de que pueda ser usado para guiar el proceso de grounding. Esta relajación del plan real conocida como *plan relajado* es la característica principal de los modelos entrenados durante nuestros experimentos y que analizaremos con más detalle durante el desarrollo de esta tesis.

Por otro lado, los modelos de aprendizaje automático dependen fuertemente de como se representan los datos que uno tiene disponible (Heaton, 2016). Para obtener un modelo de aprendizaje supervisado se necesita construir un vector de *features* que permita codificar nuestros datos, en particular, un plan relajado y una acción. Una opción es proyectar los ejemplos a un espacio vectorial a partir de una codificación del tipo *one-hot*, donde enumeramos los posibles esquemas de acción, y objetos de un problema para representar numéricamente una acción y por consecuente una secuencia de ellas. Otra posibilidad es utilizar el concepto de *word embeddings* (Mikolov et al., 2013, Pennington et al., 2014, Bojanowski et al., 2016) proveniente del procesamiento del lenguaje natural. Estos tienen como objetivo proyectar palabras y oraciones a un espacio N dimensional que preserve su semántica, es decir, expresiones con un significado similar, son dispuestas cerca en el espacio. La intuición principal es obtener un modelo de lenguaje, pensando un plan como una oración, que caracterice el lenguaje generado por los planes relajados y las acciones instanciadas.

En resumen, la hipótesis inicial de este trabajo es investigar distintas codificaciones de nuestros datos que permitan al modelo de *machine learning* reconocer la relación entre los planes relajados, acciones, y planes reales, de tal manera que guíen el proceso

de *grounding*.

1.2. Trabajos relacionados

En la literatura de planning se pueden encontrar numerosos avances que intentan guiar el proceso de grounding efectuado por el planificador. Algunas se basan en evitar instanciar acciones que no son alcanzables relajadamente desde el estado inicial, tal es el caso del planificador Fast Downward (Helmert, 2011) utilizado en competencias de planning. En (Röger et al., 2018) estudian nociones de simetría sobre la especificación de la tarea de planning para evitar trabajo redundante. Otra opción es reducir el tamaño de las interfaces de los esquemas de acción al dividirla en subpartes. De esta manera la explosión exponencial de acciones groundeadas se disminuye drásticamente. Esta técnica se denomina *action schema splitting* y es desarrollada en (Areces et al., 2014). También existen métodos que evitan el proceso de grounding completamente y realizan la búsqueda del plan a partir de su especificación PDDL (Penberthy and Weld, 1992).

En particular, esta tesis continua con la línea de investigación desarrollada en (Gnad et al., 2019) donde también proponen guiar el proceso de grounding por medio de aprendizaje automático. La diferencia radica en las características usadas para entrenar los modelos, en lugar de planes relajados y acciones, se codifica una acción por medio de reglas relacionales que capturan propiedades significativas de una acción.

Salvo esta última heurística no se han visto muchos estudios sobre el uso de técnicas de aprendizaje automático en el área de planning, lo cual es una fuerte iniciativa para empezar a combinar estos dos campos.

1.3. Estructura de la tesis

Los capítulos 2 y 3 detallan los conceptos fundamentales de planning y aprendizaje automático que se utilizaran para unir estas dos áreas.

El capítulo 4 describe que problemas de planning utilizaremos para entrenar los modelos de aprendizaje automático, junto a la exploración de los datos, preprocesamiento, separáción en entrenamiento y test, y evaluación de modelos. A su vez, se presentarán los 2 métodos principales de codificación de planes relajados y acciones que trabajamos durante la tesis.

El capítulo 5 presenta los experimentos realizados, mostrando el desempeño de varios modelos predictivos a partir de las codificaciones propuestas en el capìtulo 3. Además, se mencionan otros experimentos que fueron parte del proceso científico, que a pesar no lograr un resultado esperado, fueron de gran utilidad para reevaluar nuestros objetivos.

Por último, en el capítulo 6 se escriben las conclusiones de la tesis junto a las líneas de trabajo a futuro.

Parte I

Fundamentos teóricos

Capítulo 2

Planning Clásico

En este capítulo se detallan las definiciones teóricas, necesarias para comprender el enfoque de esta tesis y sobre qué parte de este campo aplicaremos técnicas de aprendizaje automático. Partiendo desde el área de *planning*, detallando más concretamente que son los estados y acciones, representación STRIPS de un problema, el lenguaje PDDL, el proceso de *grounding*, la complejidad de encontrar un plan que resuelva la tarea, y la necesidad de tomar ventaja de planes relajados. Gran parte del material de este capítulo fue basado en el trabajo "Learning How to Ground a Plan" llevado a cabo en (Gnad et al., 2019), el libro "Planning Theory & Practice" (Nau et al., 2004), y los artículos con el funcionamiento de Fast Downward y PDDL en (Helmert, 2011, McDermott et al., 1998). Así mismo, se hizo referencia a detalles más específicos provenientes de otros artículos científicos que también fueron de gran importancia para comprender los fundamentos de planning clásico.

2.1. ¿Qué se entiende por planning?

Durante nuestras actividades de la vida cotidiana, siempre estamos actuando y anticipando el resultado de nuestras decisiones, aun de manera inconsciente, sin estar explícitamente planeando que hacer antes de realizar una acción. Por otro lado, una tarea que abarque objetivos nuevos y complejos necesita de ser planeada conscientemente al consistir de acciones que uno no está acostumbrado, tareas de alto riesgo, o cooperación con alguien más. Hay otras tareas en las que incluso no pueden ser planeadas por una persona, sino que se deben determinar de manera automática, tal es el caso por ejemplo de una operación de rescate después de algún desastre natural, como un terremoto o inundación. La operación podría necesitar de una gran cantidad de rescatistas, sistemas de comunicación con una base general, y actividades de logística. Todos dependiendo de una planificación cuidadosa y evaluación de varias alternativas en un tiempo de rescate limitado y por ende una urgencia inmediata de toma de decisión. Este tipo de tareas justifican el uso de herramientas de planificación automática y son la principal motivación de *Automated Planning*.

Automated Planning es el área de la inteligencia artificial que busca automatizar tareas de planificación, en particular, tareas que resulta inviable ser planificadas por un humano, ya sea, por razones de costos, riesgos, o recursos. Esto se logra basándose en

el razonamiento sobre representaciones abstractas y formales del dominio, configuración inicial, combinación de acciones, y objetivos a ser cumplidos. El modelo conceptual del dominio en el cual las acciones son ejecutadas es llamado *dominio de planning*, y los requerimientos inicial y final son denominados *estado inicial* y *meta*. Estas 3 componentes definen una *tarea de planning* cuyo objetivo es encontrar alguna combinación de acciones que determinen un *plan*.

2.2. Estados y acciones

Un estado posible de una tarea de *planning* se representa a partir de un conjunto de símbolos proposicionales que modelan aspectos del entorno. Por ejemplo, el símbolo proposicional p puede modelar la situación "el agente a_1 se encuentra en la ubicación l_1". Aquellos símbolos que no están mencionados en un estado, se asumen como falsos en dicho mundo. De esta manera, si se tienen los símbolos proposicionales p, q, r, $\{p, q\}$ representa el estado en el que p y q son verdaderos, pero no r.

Las acciones son representadas en términos de una precondición y un efecto. La precondición es una fórmula proposicional que representa la condición necesaria para que la acción pueda ser llevada a cabo. Mientras que el efecto es una fórmula que determina los cambios que produce la ejecución de la acción sobre el entorno. Por ejemplo, consideremos la siguiente acción A:

$$\begin{aligned} &Action : A \\ &pre : p \wedge \neg r \\ &eff : \neg p \wedge q \end{aligned}$$

A no es aplicable en el estado $\{p, r\}$, ya que, no se cumple su precondición, pero si en el estado $\{p\}$ transformándolo en el estado $\{q\}$. Se puede interpretar las acciones como operadores de transformación de estados en el que su ejecución genera que ciertos símbolos proposicionales se hagan verdaderos o falsos según si estos ocurren positivamente o negativamente en el efecto de la acción.

El tipo de fórmula que se permite en la precondición y en el efecto de las acciones determina el tipo de la tarea de *planning*. En particular, se utilizaron tareas de *planning* STRIPS durante el desarrollo de esta tesis.

2.3. Tareas STRIPS

El tipo de una tarea de *planning* está dada por la lógica de las fórmulas que ocurren en las acciones y en la meta. En STRIPS, las fórmulas son conjunciones de literales, es decir, son de la forma $\bigwedge_i l_i$ con l_i un símbolo proposicional o su negación.

Definición 1. *Una fórmula STRIPS es una fórmula ϕ tal que ϕ es de la forma $\bigwedge_i l_i$ con l_i un literal. Una acción a es del tipo STRIPS si su precondición y su efecto son fórmulas STRIPS.*

Una forma más conveniente de trabajar sobre esta representación es utilizando únicamente conjuntos de símbolos proposicionales. Vimos que en el caso de los estados, solo

se mantienen aquellos que son verdaderos, es decir, los literales positivos. De manera similar, ocurre con la precondición de una acción. Sin embargo, en el caso de su efecto se mantienen dos conjuntos, uno con los símbolos proposicionales positivos, y otro con los negativos.

Por ejemplo, la acción A que describimos anteriormente puede verse de la siguiente forma:

$$\begin{aligned} &Action : A \\ &pre : \{p\} \\ &add : \{q\} \\ &del : \{p\} \end{aligned}$$

La interpretación de A es similar a la que se dio anteriormente, la precondición contiene los símbolos proposicionales necesarios para que A sea aplicable en un estado. Mientras que add y del son los conjuntos de símbolos proposicionales que agrega y elimina la acción producto de su ejecución.

Esta nueva representación permite abstraer fórmulas y operadores proposicionales, lo cual ayudará para introducir de manera más natural algunas técnicas del área.

Algo importante a recalcar es el orden en que se ejecuta una acción. Para un estado aplicable de A optaremos por primero eliminar los símbolos que sean verdaderos a partir de su lista del para luego agregar los de la lista add. Esto con el fin de garantizar de que el estado resultante contenga todos los símbolos de esta última lista. Para ejemplificar esto, supongamos que tenemos la siguiente acción A', pero eligiendo está vez el orden contrario de aplicación.

$$\begin{aligned} &Action : A' \\ &pre : \{p\} \\ &add : \{q\} \\ &del : \{p, q\} \end{aligned}$$

$$(\{p\} \cup add(A')) - del(A') = \{p, q\} - del(A') = \emptyset$$

Notar que de esta manera, no tenemos ninguna garantía de que los objetos que agrega A' efectivamente pertenezcan al estado resultante de su ejecución.

Definición 2. *Una tarea de planning STRIPS es una 4-upla* $\Pi = (F, A, I, G)$ *donde* F *es un conjunto finito de símbolos proposicionales denominados facts,* A *es un conjunto finito de acciones STRIPS,* $I \subseteq F$ *el estado inicial,* $G \subseteq F$ *el estado final.*

Definición 3. *Sea* $\Pi = (F, A, I, G)$ *una tarea STRIPS.*

- *Un estado* $s \subseteq F$ *es un conjunto de facts. Diremos que un símbolo proposicional* $p \in F$ *vale en un estado* s *sii* $p \in s$*.*

- *Una acción STRIPS es una 3-upla* $a = (pre, add, del)$, *tal que,* $pre, add,$ *y* del *son subconjuntos de* F, *y los denotaremos como* $pre(a)$, $add(a)$, *y* $del(a)$ *respectivamente.*
- *Una acción* a *es aplicable en un estado* s *si* $pre(a) \subseteq s$, *en tal caso, el estado resultante es* $s' = (s - del(a)) \cup add(a)$. *Escribimos* $s \xrightarrow{a} s'$ *para la transición de* s *a* s' *vía* a. *Para una secuencia de acciones* $\vec{a} \in A^*$, *escribimos* $s \xrightarrow{\vec{a}} t$ *si estos pueden ser iterativamente aplicados a* s, *resultando en* t.
- *Un plan para* Π *es una secuencia* $\vec{a} \in A^*$ *con* $I \xrightarrow{\vec{a}} s_G$ *si* $G \subseteq s_G$.
- *Una tarea* Π *es satisfacible si un plan para* Π *existe. El plan es denominado óptimo si es el que tiene longitud más corta de entre todos los planes para* Π.

Para ejemplificar estas definiciones consideremos el problema de la Figura 2.1 Tenemos único agente a_1 que se encuentra inicialmente en la posición l_1 marcada en azul y desea ir hacia la ubicación l_4 en rojo. Los enlaces de la figura indican los posibles movimientos dentro del mapa. La tarea STRIPS en este caso estaría dada por:

$$\begin{aligned}
F &= \{at(a_1, l_1), at(a_1, l_2), at(a_1, l_3), at(a_1, l_4)\} \\
A &= \{move(a_1, l_1, l_2), move(a_1, l_2, l_1), \\
&\quad move(a_1, l_1, l_3), move(a_1, l_3, l_1), \\
&\quad move(a_1, l_3, l_4), move(a_1, l_4, l_3)\} \\
I &= \{at(a_1, l_1)\} \\
G &= \{at(a_1, l_4)\}
\end{aligned}$$

Para cada agente x y ubicaciones l, l' se tiene que:

$$move(x, l, l') = (\{at(x, l)\}, \{at(x, l')\}, \{at(x, l)\})$$

Un posible plan de la tarea podría ser la secuencia $\vec{a} = move(a_1, l_1, l_3)move(a_1, l_3, l_4)$.

2.4. Representaciones STRIPS

Un dato de entrada necesario para cualquier algoritmo de planificación es una descripción del problema a ser resuelto. En la práctica, es usualmente imposible incluir una enumeración explícita de todos los estados y transiciones que se pueden realizar en el dominio a partir de una tarea STRIPS. Por lo tanto, es necesario una representación que permita computarlas dinámicamente.

Consideremos el problema del ejemplo anterior donde se tiene el agente a_1 pero esta vez n ubicaciones. Para este caso, se va a necesitar n símbolos proposicionales p_i tal que representen la acción "El agente a_1 está en la ubicación i". Si en lugar de un solo agente hubiese una cantidad m de ellos. Se necesitarían $n \times m$ símbolos para modelar esta característica de la tarea. Una forma más compacta de representar esta propiedad

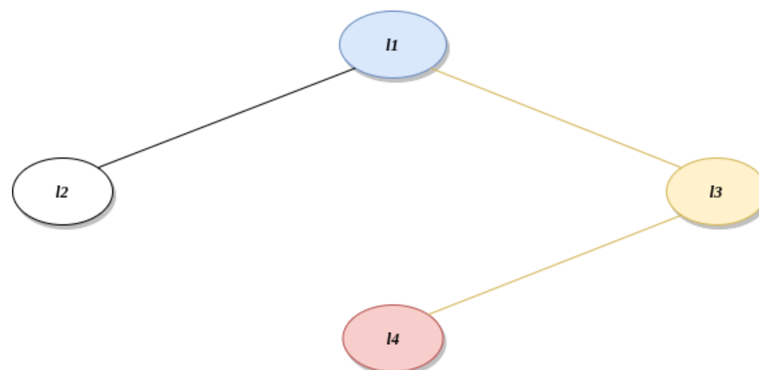


Figura 2.1: Ejemplo del problema del agente. Su posición inicial es la ubicación l_1 en azul queriendo llegar a la posición l_4 marcada con rojo.

es por medio de un predicado de la forma $at(x, l)$ donde x denota a un agente, y l su ubicación, siendo mucho más flexible debido a su naturaleza esquemática.

De manera similar, si se quisiera modelar una acción de cambio de posición, haría falta describir una por cada agente y par de ubicaciones obteniendo un total de $m \times n \times n$ instancias. Una mejor alternativa es representar esta transformación por medio de un esquema de acción de la forma $move(x, l, l')$ donde x representa al agente, l la ubicación actual, y l' la ubicación a la cual moverse.

De esta manera resulta más adecuado para la especificación utilizar predicados y esquemas de acción en lugar de símbolos proposicionales. Esto lleva a las siguientes definiciones:

Definición 4. *Una especificación de una tarea STRIPS es una 6-upla* $(\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$ *donde* $\mathcal{P}$ *es un conjunto de predicados,* $\mathcal{A}$ *es un conjunto de esquemas de acción,* Σ^C *es un conjunto de símbolos constantes,* Σ^O *es un conjunto de variables,* $I \subseteq \mathcal{P}^{\Sigma^C}$ *es el estado inicial, y* $G \subseteq \mathcal{P}^{\Sigma^c}$ *es la meta.*

- *Un esquema de acción es una 3-upla* $a[X] = (pre(a), add(a), del(a))$ *donde cada elemento es subconjunto de* $\mathcal{P}$*, y* X *es el conjunto de variables que ocurren en* $pre(a) \cup add(a) \cup del(a)$ *y que son parte de* Σ^O*.*

- *Un predicado es un símbolo atómico de* $\mathcal{P}$ *de la forma* $p[X]$ *con* X *un conjunto de variables que ocurren en su interfaz y que son parte de* Σ^O*.*

- $\mathcal{P}^{\Sigma^C}$ *es el conjunto de todos los posibles facts que se pueden instanciar con los predicados en* $\mathcal{P}$ *a partir de los símbolos en* Σ^C*, es decir, el fact* $\hat{p} \in \mathcal{P}^{\Sigma^C}$ *si* $\hat{p}$ *se obtiene a partir de reemplazar todas las variables en* $p[X]$ *por objetos concretos de* Σ^C*, para algún* $p[X] \in \mathcal{P}$*.*

Para la tarea STRIPS del problema del agente una primera especificación podría ser definida de la siguiente manera:

$$\begin{aligned}
\mathcal{P} &= \{at(x, l)\} \\
\mathcal{A} &= \{move(x, l, l')\} \\
\Sigma^C &= \{a_1, l_1, l_2, l_3, l_4\} \\
\Sigma^O &= \{x, l, l'\} \\
I &= \{at(a_1, l_1)\} \\
G &= \{at(a_1, l_4)\} \\
move(x, l, l') &= (pre(move), add(move), del(move)) \\
&= (\{at(x, l)\}, \{at(x, l')\}, \{at(x, l)\})
\end{aligned}$$

Una diferencia importante es que ahora $move(x, l, l')$ es un esquema de acción donde x, l, l' son variables libres pertenecientes a Σ^O. De esta manera, las variables se hicieron parte de la especificación, logrando expresar predicados y acciones de manera esquemática.

Por otro lado, está especificación de la tarea STRIPS, instancia más acciones que las del problema anterior, en particular, $move(a_1, l_2, l_4)$ está especificada como un movimiento posible. Por lo tanto, para representar exactamente el grafo de la figura 2.1 es necesario agregar a la especificación un predicado extra, $is_connected(l, l')$, que represente la existencia de un camino entre l y l'. Luego, la estructura del grafo de la figura 2.1 debe incluirse en el estado inicial. Además, el esquema de acción $move(x, l, l')$ debe poder aplicarse en un estado siempre y cuando exista un camino entre l y l'.

$$\begin{aligned}
\mathcal{P} &= \{at(x, l)\} \\
\mathcal{A} &= \{move(x, l, l')\} \\
\Sigma^C &= \{a_1, l_1, l_2, l_3, l_4\} \\
\Sigma^O &= \{x, l, l'\} \\
I &= \{at(a_1, l_1), \\
&\qquad is_connected(l_1, l_2), is_connected(l_2, l_1), \\
&\qquad is_connected(l_1, l_3), is_connected(l_3, l_1), \\
&\qquad is_connected(l_3, l_4), is_connected(l_4, l_3)\} \\
G &= \{at(a_1, l_4), \\
&\qquad is_connected(l_1, l_2), is_connected(l_2, l_1), \\
&\qquad is_connected(l_1, l_3), is_connected(l_3, l_1), \\
&\qquad is_connected(l_3, l_4), is_connected(l_4, l_3)\} \\
move(x, l, l') &= (pre(move), add(move), del(move)) \\
&= (\{at(x, l), is_connected(l, l')\}, \{at(x, l')\}, \{at(x, l)\})
\end{aligned}$$

Un detalle de esta especificación es que es no tipada. Por ejemplo $move(a_1, a_1, a_1)$ es una acción instanciable. Lo cual deja en claro la necesidad de una representación de tareas que preserve los tipos de los esquemas de acción y predicados, y así evitar preprocesar acciones innecesarias durante el proceso de grounding.

Por último, un plan que resuelve la tarea STRIPS instanciada, es a su vez un plan para la especificación siendo está relación la que permite vincular ambas representaciones.

2.5. El lenguaje PDDL

Para especificar una tarea de planning a una computadora se necesita definir un lenguaje que permita al usuario definir cada una de sus componentes. Para dar respuesta a esta necesidad surge el *Planning Domain Specification Language* (PDDL), Propuesta para la competencia de planning AIPS-98 (McDermott et al., 1998) siendo el lenguaje de especificación de dominios y problemas de planning más usado dentro de la comunidad de planning.

Una especificación en PDDL consiste de dos archivos, la definición del dominio, donde se aclaran los predicados y esquemas de acción disponibles; y la definición de la tarea donde se mencionan los objetos concretos a modelar, el estado inicial, y la meta. Esta separación resulta importante, ya que diferentes tareas de un mismo dominio comparten una estructura común. Los listing 2.1 y 2.2 muestran los archivos de dominio y tarea del ejemplo del agente escrita en PDDL.

```
1   (
2     define (domain AGENT-DOMAIN)
3
4     (:requirements :strips :typing)
5
6     (:types AGENT LOCATION)
7
8     (:predicates (at ?x - AGENT ?l - LOCATION)
9                  (is_connected ?l - LOCATION ?l' - LOCATION)
10    )
11
12    (:action move
13        :parameters (?x - AGENT ?s - LOCATION ?d - LOCATION)
14        :precondition (and (at ?x ?s) (is_connected ?l ?l'))
15        :effect (and (not (at ?x ?s)) (at ?x ?d))
16    )
17  )
```

Listing 2.1: Dominio Agent en PDDL.

Observar que tanto los predicados como las acciones están parametrizados con variables de un cierto tipo. Por ejemplo para el predicato $at(x, y)$, x es de tipo *AGENT* e y del tipo *LOCATION* ambos previamente listados en la línea 6 del listing 2.1. Luego en la línea 7 del listing 2.2 se especifican todos los objetos disponibles y el tipo al cual corresponden.

```
(
  define (problem AGENT-TASK)

  (:domain AGENT-DOMAIN)

  (:objects a1 - AGENT l1 l2 l3 l4 - LOCATION)

  (:init at(a1, l1)
         (is_connected l1 l2)
         (is_connected l2 l1)
         (is_connected l1 l3)
         (is_connected l3 l1)
         (is_connected l3 l4)
         (is_connected l4 l3)
         (is_connected l1 l2)
         (is_connected l2 l1)
         (is_connected l1 l3)
         (is_connected l3 l1)
         (is_connected l3 l4)
         (is_connected l4 l3)
  )

  (:goal (and
         (at a1 l4)
         (is_connected l1 l2)
         (is_connected l2 l1)
         (is_connected l1 l3)
         (is_connected l3 l1)
         (is_connected l3 l4)
         (is_connected l4 l3)
         (is_connected l1 l2)
         (is_connected l2 l1)
         (is_connected l1 l3)
         (is_connected l3 l1)
         (is_connected l3 l4)
         (is_connected l4 l3)
         )
  )
)
```

Listing 2.2: Ejemplo de una tarea del dominio Agent en PDDL.

2.6. Relajación por deletes

El problema de decidir si una tarea de planning STRIPS tiene un plan es PSPACE-completo. Es por eso que las implementaciones que lo resuelven son a partir de algoritmos de búsqueda de orden exponencial. El planificador instancia un problema y realiza una búsqueda exhaustiva guiada por medio de una función heurística definida a partir de *planes relajados*. Un plan relajado es un plan en un dominio donde a todas las acciones se les eliminaron los efectos negativos. Es decir su lista del. Si bien no es exactamente el dominio sobre el cual se desea resolver la tarea original, brinda información relevante para

hallar una solución. En particular, si los planes relajados permiten guiar la búsqueda para encontrar un plan de la tarea, entonces también podría ser usado para guiar el proceso de grounding.

Los planes relajados son secuencias de acciones cuyos efectos negativos son eliminados, son aplicables en el estado inicial, y su estado resultante contiene a la meta. Esto proviene de la *relajación por deletes*, una simplificación de la tarea de planning que consiste en eliminar los efectos negativos de las acciones. Esta relajación es ampliamente utilizada en planning y su importancia subyace en la siguiente propiedad: en la *relajación por deletes* si un *fact* p se vuelve verdadero en un estado s, entonces para toda secuencia de acciones $\vec{a} \in A^*$ aplicables en s, el estado resultante también hará verdadero p. Por ejemplo, en el problema del agente, si este se desplazó por varias ubicaciones, tendríamos que se encuentra en más de un lugar al mismo tiempo, ya que la proposición que indicaba su ubicación anterior no puede ser borrada. Esta propiedad es la que permite que computar un plan en el dominio relajado sea polinomial en comparación al dominio original.

Definición 5. *Sea* $\Pi = (F, A, I, G)$ *una tarea STRIPS.*

- *Sea* $a \in A$*, definimos a la acción relajada por delete* a^+ *dada por* $pre(a^+) = pre(a)$*,* $add(a^+) = add(a)$*, y* $del(a^+) = \emptyset$*.*
- *Denotaremos con* $A^+ = \{a^+ : a \in A\}$ *al conjunto de acciones relajadas por delete.*
- *Para una secuencia de acciónes* $\vec{a}$ *denotaremos con* $\vec{a}^+$ *a la secuencia de acciones relajadas por delete.*
- *Denotaremos con* $\Pi^+ = (F, A^+, I, G)$ *a la tarea STRIPS relajada por deletes.*
- *Un plan relajado para* Π *es un plan para* Π^+*.*

Claramente es una modelización no realista, no obstante, relajar una tarea STRIPS provee información importante sobre la tarea original, en particular:

Lemma 2.6.1. *Sea* $\Pi = (F, A, I, G)$ *Una tarea STRIPS*

- *Si una secuencia de acciones* $\vec{a}$ *es un plan de* Π*, entonces* $\vec{a}^+$ *es un plan relajado de* Π*.*
- *Si no existe un plan para* Π^+ *entonces no existe un plan para* Π*.*

Estas propiedades serán muy relevantes al introducir los algoritmos de grounding en las secciones 2.7, y 2.8. La primera de ellas nos dice que un plan de una tarea STRIPS, es a la vez un plan de la tarea relajada. No obstante el recíproco no se cumple. Por lo general, un plan en la tarea relajada no es un plan de la tarea original. La segunda de ellas menciona que si no existe un plan para la tarea relajada, entonces no hay un plan para la tarea original. Esto también es importante, ya que todo plan en la tarea relajada puede encontrarse con solo instanciar aquellas acciones que sean alcanzables. Si no se encuentra un plan de la tarea relajada para ese conjunto de acciones, entonces sabremos que no existirá solución en la tarea original.

2.7. Proceso de grounding

Dada una especificación de una tarea STRIPS $(\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$ se puede obtener su correspondiente tarea $\Pi = (F, A, I, G)$ recolectando todas las instancias posibles de predicados en $\mathcal{P}$ y esquemas de acción de $\mathcal{A}$ con objetos de Σ^C. Es decir, F contiene un *fact* por cada posible asignación de objetos a los argumentos de cada predicado $P[X] \in \mathcal{P}$, y A contiene una acción por cada posible asignación de objetos a los argumentos de cada esquema de acción $a[X] \in \mathcal{A}$. Este proceso se conoce como *grounding cartesiano*.

No obstante, en la práctica los planificadores no instancian todas las posibles acciones y proposiciones, sino aquellas que son alcanzables relajadamente desde el estado inicial debido al lemma 2.6.1.

Este proceso de grounding es el implementado por *Fast Downward* (Helmert, 2011), el sistema de planning clásico basado en búsqueda heurística más popular por la comunidad debido a su excelente desempeño en benchmarks, competencias internacionales de planning, e implementaciones eficientes de algoritmos genéricos de búsqueda. Estas implementaciones son altamente configurables por Fast Downward permitiendo ajustarse a distintos dominios de planning. Su popularidad y versatilidad fue la motivación que nos llevó a utilizarlo en esta tesis en comparación a otros sistemas.

Definición 6. *Sea $\Pi = (F, A, I, G)$ una tarea STRIPS.*

- *Un estado $s \subseteq F$ es alcanzable en Π si existe una secuencia $\vec{a} \in A^*$ tal que $I \xrightarrow{\vec{a}} s$.*

- *Una proposición $p \in F$ es alcanzable en Π si existe un estado s alcanzable en Π tal que $p \in s$.*

- *Una acción $a \in A$ es alcanzable en Π si existe un estado s alcanzable en Π tal que a es aplicable en s.*

- *Una proposición $p \in F$ es alcanzable relajadamente en Π si p es alcanzable en Π^+.*

- *Una acción $a \in A$ es alcanzable relajadamente en Π si a^+ es alcanzable en Π^+.*

El algoritmo 1 implementado por Fast Downward parte de una cola q que almacena todos los símbolos proposicionales del estado inicial. Luego la cola q almacena elementos a preprocesar que pueden ser tanto facts como acciones. A partir de allí, se remueve elemento a elemento de q consultando si es un fact o una acción.

En el caso de encontrarse con un fact, se lo instancia y se agregan a q todas las acciones aplicables (aún no procesadas) a partir de las proposiciones groundeadas hasta el momento. Caso contrario, se obtiene una acción que también se instancia y se agregan a q todas las proposiciones (aún no procesadas) que ocurren positivamente en su efecto, es decir, su lista add.

Este procedimiento se repite hasta que la cola se encuentre vacía, obteniendo de esta manera todas las acciones y facts alcanzables relajadamente.

Algorithm 1 Grounding por alcanzabilidad relajada

Input: Especificación de una tarea STRIPS $(\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$
Output: Tarea STRIPS (F, A, I, G)

```
1: q ← Queue(I)
2: F ← ∅
3: A ← ∅
4: while ¬q.empty() do
5:     e ← q.pop()
6:     if e.isFact() then
7:         F ← F ∪ {e}
8:         for a ∉ A ∧ pre(a) ⊆ F do
9:             q.insert(a)
10:        end for
11:    else
12:        A ← A ∪ {e}
13:        for f ∉ F ∧ f ∈ add(e) do
14:            q.insert(f)
15:        end for
16:    end if
17: end while
18: return (F, A, I, G)
```

2.8. Grounding heurístico

Mencionamos que un plan de una tarea es también un plan relajado de la misma. Sin embargo, el recíproco no se cumple. Es más, un hecho o una acción que es relajadamente alcanzable, puede no ser alcanzable en la tarea original o ser irrelevante al no pertenecer a ningún plan que la resuelva. Es justamente esta pérdida de información la que origina que computar en el mundo relajado sea polinomial. Es aquí donde surge el foco de este trabajo donde se intentará predecir cuál de estos hechos y acciones son verdaderamente relevantes para alcanzar la meta, priorizando su instanciación por sobre el resto.

El algoritmo de grounding heurístico que presentaremos es el mismo que se desarrolló en (Gnad et al., 2019) y está basado en el que se mostró en la sección anterior. Las principales diferencias son el criterio de parada, donde se procesan no más de N acciones, y el orden en que lo hacen, según su nivel de relevancia. Debido a que el algoritmo puede terminar antes de que la cola se vacíe, se está obligado a procesar la totalidad de los facts existentes en la cola antes de considerar una siguiente acción. Esto garantiza que todos los facts que se encuentran en los efectos de las acciones ya procesadas están en F, siendo el algoritmo *fact consistente*.

Definición 7. *Sean F y A el conjunto de facts y acciones retornadas por un algoritmo de grounding. Decimos que este es fact consistente si para todo $a \in A$ se cumple que, si $p \in pre(a) \cup add(a) \cup del(a)$ entonces $p \in F$.*

En grounding heurístico, el algoritmo puede detenerse mucho antes según el parámetro N que indica la cantidad de instancias que puede realizar el proceso sin que los

Algorithm 2 Grounding heurístico

Input: Especificación de una tarea STRIPS $(\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$. Número máximo de instanciaciones permitidas N

Output: Tarea STRIPS (F, A, I, G)

```
1: q ← PriorityQueue(I)
2: F ← ∅
3: A ← ∅
4: while ¬(q.empty() ∨ G ⊆ F) ∧ |A| ≤ N do
5:     if q.containsFact() then
6:         f ← q.popFact()
7:         F ← F ∪ {f}
8:         for a ∉ A ∧ pre(a) ⊆ F do
9:             q.insert(a)
10:        end for
11:    else
12:        a ← q.popHighPriorityAction()
13:        A ← A ∪ {a}
14:        for f ∉ F ∧ f ∈ add(a) do
15:            q.insert(f)
16:        end for
17:    end if
18: end while
19: return (F, A, I, G)
```

recursos de tiempo y memoria se vean comprometidos. Por lo general, muchas de los problemas de planning requieren planes cortos del orden de a lo sumo cientos de acciones en comparación a posiblemente millones de operadores groundeados.

Es importante mencionar que este algoritmo es *correcto*, si hallamos un plan de la tarea obtenida por grounding heurístico, entonces ese mismo plan lo es para la tarea obtenida por grounding total. No obstante, no es *completo*, una tarea obtenida por grounding heurístico tiene solución si y sólo si las acciones correspondientes de al menos un plan (de la tarea obtenida por grounding total) fueron procesadas.

Por último, algo a considerar es el remplazo de la cola del Algoritmo 2, por una cola de prioridades que asigna a las acciones un nivel de relevancia por la cual son ordenadas. La relevancia de una acción está directamente relacionada con la probabilidad de que esta pertenezca a un plan de la tarea. Notar que una estimación ideal sería una probabilidad de 1 para las acciones que ocurren en algún plan de la tarea y 0 para las restantes. Pero computar esto es tan difícil como obtener una solución y sabemos que es PSPACE-completo. Es por eso que para estimar la relevancia se utilizaron técnicas de aprendizaje automático y serán el foco del siguiente capítulo.

Capítulo 3

Aprendizaje automático

En este capítulo se profundizará en el área de *aprendizaje automático* abarcando la representación de palabras y oraciones dadas por codificaciones del tipo one-hot, word embeddings, conceptos de aprendizaje supervisado y no supervisado, modelos de clasificación, y métricas. Este capítulo se basó en mayor parte del contenido en (Bishop, 2006), y (Chen and Guestrin, 2016) para la explicación de conceptos de aprendizaje supervisado, y (Bojanowski et al., 2016, Mikolov et al., 2013) para representación de palabras.

3.1. Orígenes y evolución

El aprendizaje automático es el campo de la inteligencia artificial que busca desarrollar programas que mejoren en base a la experiencia, automatizar el reconocimiento de patrones estadísticos sobre volúmenes de datos, y utilizar algoritmos de computación que tomen acción sobre datos no aprendidos.

Por moderno que pueda parecer este campo, tuvo sus comienzos en los años 50 a partir del famoso *Test de Turing*, una máquina cuyo objetivo era engañar a un humano haciéndole creer que se encontraba delante de otra persona en lugar de un ordenador, llegando a la conclusión de que los computadores de propósito general, podrían ser capaces de aprender y ser de alguna manera originales. (*Página web: Test Turing*, n.d.)

En los últimos años, varias aplicaciones se han beneficiado de esta área, desde programas relacionados con la detección fraudulenta de transacciones con tarjeta de crédito (Fang et al., 2021), sistemas de recomendación que guían usuarios en un servicio de acuerdo a sus preferencias (Burke and Robin, 2007), o incluso vehículos que se manejan sin necesidad de la intervención del conductor (Grigorescu et al., 2019). Incluso en algunas tareas de reconocimiento de patrones el aprendizaje automático ha logrado obtener tasas menores al error humano (Kühl et al., 2020). Al mismo tiempo, una importante cantidad de avances teóricos y algorítmicos se fueron realizando formando las bases de este campo.

3.2. Aprendizaje supervisado

El aprendizaje supervisado es una subárea del aprendizaje automático cuyo objetivo es deducir un modelo a partir de datos anotados que permita mapear ejemplos no vistos previamente. Esta información está compuesta por un conjunto de ejemplares y sus correspondientes etiquetas. En ocasiones, dada por un anotador humano que indica el resultado esperado del modelo a partir del dato como entrada. Las etiquetas pueden ser categóricas o continuas determinando un problema de clasificación o regresión, respectivamente. Algunas tareas más frecuentes de clasificación son la categorización de documentos (Wan et al., 2019), reconocimiento de lenguaje ofensivo (Wei et al., 2021), o análisis de sentimiento (Dang et al., 2020). Mientras que para regresión, lo son las estimaciones de precios de artículos, objetos, o viviendas (Yeh and Deng, 2011).

El resultado de ejecutar un algoritmo de machine learning supervisado se puede expresar como una función $f(\boldsymbol{x})$ que recibe un ejemplar $\boldsymbol{x}$ como entrada y genera su predicción y de salida. Las muestras utilizadas para ajustar f están dadas por pares (vector, etiqueta) $\{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$ conformando el *conjunto de entrenamiento*.

La forma precisa de f es determinada durante la fase de entrenamiento. Una vez transcurrida, se puede estimar la etiqueta de nuevos datos que no pertenezcan al conjunto de entrenamiento. En particular, se suele medir la performance del modelo comparando las predicciones con la etiqueta real de nuevos datos que no fueron utilizados en la fase de entrenamiento. En el caso que la predicción se aproxime a la esperada para estas nuevas entradas, entonces el modelo habría logrado generalizar la tarea.

Sin embargo, los modelos de aprendizaje supervisado están limitados por la disponibilidad de las anotaciones y la dificultad para obtenerlas. Algunas tareas son relativamente sencillas de etiquetar por cualquier persona, (e.g., como determinar si en una foto aparece un gato), mientras que en otros puede llegar a requerir humanos que sean expertos de dominio (e.g., abogados, médicos, lingüistas, etc.).

3.3. Aprendizaje no supervisado

En contraste al método anterior, el aprendizaje no supervisado consiste en descubrir automáticamente patrones sobre los datos de entrenamiento que permitan explicarlos sin depender de datos anotados. Es considerada como parte del área del análisis y exploración. Es por eso que no pueden ser evaluados basándose en exactitud o precisión, sino más bien en la cantidad de información que podamos extraer de los datos a partir del uso de estas técnicas. Algo a considerar es la facilidad para disponer de estos datos (en comparación a los no supervisados) al no requerir ningún tipo de anotación previa. Un ejemplo de uso de aprendizaje no supervisado es la segmentación de mercado en aplicaciones e-commerce. En este se buscan desarrollar estrategias de mercado a partir de la identificación de similitudes entre clientes por medio de sus necesidades, y preferencias (Tiwari et al., 2018). Otro ejemplo muy común es en la selección de características y reducción de dimensionalidad, permitiendo eliminar información de nuestro conjunto de datos que es irrelevante y que afecta de manera negativa la eficiencia y efectividad de algoritmos de aprendizaje automático (Farahat et al., 2013).

3.4. Algoritmos de clasificación

Comenzaremos estudiando algunos métodos de aprendizaje supervisado sin preocuparnos inicialmente por la manera en que los ejemplares están codificados con excepción de las etiquetas. Es decir para un conjunto de entrenamiento $\{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$ la codificación de los $\boldsymbol{x}_i's$ será irrelevante por el momento. Las únicas restricciones que se pedirán es que sean vectores numéricos. Los $y_i's$ aparte de ser numéricos, deben ser categóricos, dado que trataremos con algoritmos de clasificación.

El objetivo principal de un clasificador es aproximar una función no conocida f a partir de un modelo $h : \mathbb{R}^D \to \{c_1, ..., c_k\}$ la cual mapea ejemplos de entrada a una categoría c_j. En particular, nos enfocaremos en clasificadores binarios $k = 2$ que fueron los utilizados para la realización de este trabajo.

3.4.1. Modelos lineales

La forma más sencilla de representar una función que discrimine datos de entrada en dos clases es a partir de una función lineal que dependa de estos datos:

$$h_{\boldsymbol{w},w_0}(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + w_0 \tag{3.1}$$

$$= \sum_{j=1}^{D} w_j x_j + w_0 \tag{3.2}$$

Donde $\boldsymbol{w}$ es llamado el *vector de pesos*, y w_0 es el *sesgo*. Ambos son parámetros de la función h y son los que se buscan ajustar para lograr separar el conjunto de datos donde $\boldsymbol{x}$ es uno de tales ejemplos. Las dimensiones de los vectores $\boldsymbol{x}$, y $\boldsymbol{w}$ son ambas $(D \times 1)$ a diferencia del sesgo que es un escalar. La expresión $\boldsymbol{w}^\top \boldsymbol{x}$ es el producto interno entre $\boldsymbol{w}$ y $\boldsymbol{x}$. Por lo que el resultado de $\boldsymbol{w}^\top \boldsymbol{x} + b$ es también un escalar. Dado que nos interesa asignar a $\boldsymbol{x}$ una categoría, se debe decidir a cuál pertenece basándose en el resultado obtenido por el producto interno. Es por eso que surge la necesidad de agregar una nueva operación que discrimine un valor de entrada en una clase. Para ello definimos una *función de decisión* $d : \mathbb{R} \to \{c_1, c_2\}$ tal que mapea valores a una clase c_1, o c_2.

Una posible función de decisión podría estar definida como $d(t) = c_1$ si $t \geq 0$, $d(t) = c_2$ caso contrario. Por lo tanto la frontera de decisión queda definida por la relación $h_{\boldsymbol{w},w_0}(\boldsymbol{x}) = 0$ y corresponde a un hiperplano de $(D - 1$ dimensiones) en uno de D dimensiones. El sesgo w_0 desplaza el hiperplano del origen de coordenadas de D. Un ejemplo de pares de puntos separados por un hiperplano (en este caso una recta) se muestra en la figura 3.1.

Algunas veces es conveniente usar una notación más compacta en la que introducimos una "inocente" componente de entrada más $x_0 = 1$ al vector $\boldsymbol{x}$ para luego definir $\boldsymbol{w}' = (w_0, \boldsymbol{w})$ y $\boldsymbol{x}' = (x_0, \boldsymbol{x})$. Luego las ecuaciones 3.1 y 3.2 se pueden expresar como:

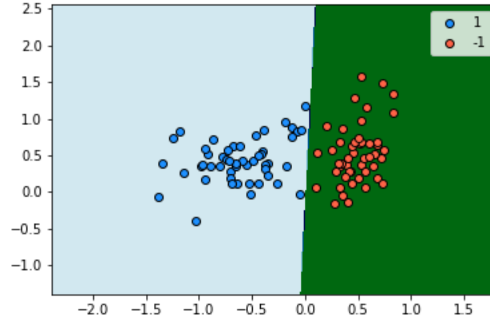


Figura 3.1: Frontera de decisión de un modelo lineal separando los puntos que tienen etiqueta $c_1 = 1$ y $c_2 =$ -1.

$$h_{\boldsymbol{w}'}(\boldsymbol{x}') = \boldsymbol{w}'^{\top}\boldsymbol{x}' \tag{3.3}$$

$$= \sum_{j=0}^{D} w_j x_j \tag{3.4}$$

Ahora la frontera de decisión corresponde a un hiperplano de dimensión D que pasa por el origen del espacio $D+1$ dimensional. A partir de ahora, cada vez que se omita el sesgo estaremos utilizando esta notación.

3.4.2. Modelos probabilísticos

En el capítulo 2 mencionamos que el objetivo de usar un algoritmo de aprendizaje automático es determinar una función que modele la probabilidad de que una acción sea relevante para encontrar un plan de una tarea STRIPS. Sin embargo, desconocemos aún de que manera se obtienen estas probabilidades, es por eso que el foco de esta sección es indagar sobre los modelos probabilísticos.

En estos modelos se busca modelar la densidad condicional $p(c_k|\boldsymbol{x})$ que representa la probabilidad de la clase c_k dado $\boldsymbol{x}$. El inconveniente con esto es que cada condicionamiento define una densidad probabilística. Por lo que se tendría una por cada posible ejemplar $\boldsymbol{x}$ resultando intratable de modelar. No obstante, si se considera el condicionamiento $p(\boldsymbol{x}|c_k)$ solo es necesario modelar una cantidad pequeña de clases, en especial para un clasificador binario. Para nuestra suerte $p(c_k|\boldsymbol{x})$, está relacionado con $p(\boldsymbol{x}|c_k)$ por la *regla de Bayes*.

$$p(c_1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|c_1)\,p(c_1)}{p(\boldsymbol{x}|c_1)\,p(c_1) + p(\boldsymbol{x}|c_2)\,p(c_2)} \tag{3.5}$$

$$= \frac{1}{1+\exp(-a)} \tag{3.6}$$

$$= \sigma(a) \tag{3.7}$$

Donde definimos $a = \ln \frac{p(\boldsymbol{x}|c_1)p(c_1)}{p(\boldsymbol{x}|c_2)p(c_2)}$. La función $\sigma(a)$ es conocida como la función *sigmoide* y juega un importante rol en muchos algoritmos de clasificación por su propiedad de simetría $\sigma(-a) = 1 - \sigma(a)$, el significado de su inversa $a = \ln\left(\frac{\sigma}{1-\sigma}\right)$ que representa el logaritmo del radio de las probabilidades de las clases c_1 y c_2, y la capacidad de mapear el eje real en un intervalo finito.

Se puede generalizar esta ecuación para el caso $k > 2$ de la siguiente manera:

$$p\left(c_k|\boldsymbol{x}\right) = \frac{p\left(\boldsymbol{x}|c_k\right)p(c_k)}{\sum_{j=1}^{k} p\left(\boldsymbol{x}|c_j\right)p(c_j)} \tag{3.8}$$

$$= \frac{\exp\left(a_k\right)}{\sum_{j=1}^{k} a_j} \tag{3.9}$$

$$\tag{3.10}$$

Donde $a_k = \ln p\left(\boldsymbol{x}|c_k\right)p\left(c_k\right)$. La función $\frac{\exp(a_k)}{\sum_{j=1}^{k} a_j}$ es conocida como la función *softmax*, ya que es una versión suavizada de la función max en el sentido de sí $a_k >> a_j$ para todo $j \neq k$ entonces $p(c_k|\boldsymbol{x}) \approx 1$ y $p(c_j|\boldsymbol{x}) \approx 0$.

3.4.3. Función de costo

Varios de los métodos de aprendizaje automático existentes no pueden obtener los pesos W de manera analítica. Es por eso que se hacen uso de métodos iterativos en base a una *función de costo* que mide la precisión de su predicción en comparación con la etiqueta. Si tenemos ejemplos de entrenamiento $\{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$ podemos definir la función de costo en base a cada ejemplar i.

Un ejemplo clásico de función de costo es el error cuadrático medio:

$$J(\boldsymbol{w}) = \frac{1}{2n} \sum_{i=1}^{n} \left(h_{\boldsymbol{w}}\left(\boldsymbol{x}_i\right) - y_i\right) \tag{3.11}$$

Por lo tanto, al minimizar J se estará reduciendo el error de estimación de h obteniendo que $h_{\boldsymbol{w}}\left(\boldsymbol{x}_i\right) \approx f(\boldsymbol{x}_i)$. Para lograr tal minimización, existe una técnica denominada *descenso por el gradiente* que actualiza de manera iterativa el vector $\boldsymbol{w}$ logrando la convergencia de la función de costo J a un mínimo valor.

3.4.4. Descenso por el gradiente

Definimos $\boldsymbol{\Delta J}\left(\boldsymbol{w}\right)$ al gradiente de J en $\boldsymbol{w}$ como $\boldsymbol{\Delta J}\left(\boldsymbol{w}\right) = \left(\frac{\partial J(\boldsymbol{w})}{\partial w_1}, ..., \frac{\partial J(\boldsymbol{w})}{\partial w_n}\right)$, el vector de derivadas parciales. Este algoritmo empieza con una inicialización aleatoria de $\boldsymbol{w}$ y calcula $\boldsymbol{\Delta J}\left(\boldsymbol{w}\right)$ por una cantidad de iteraciones. El negativo del gradiente indica la dirección de mayor descenso en el cual la función $J(\boldsymbol{w})$ disminuye. Por lo tanto, si al vector $\boldsymbol{w}$ se le suma el negativo del gradiente se estaría dirigiéndolo en torno a un mínimo local de la función de costo. La magnitud del desplazamiento vectorial en cada iteración está determinado por una constante α conocida como la *tasa de aprendizaje*.

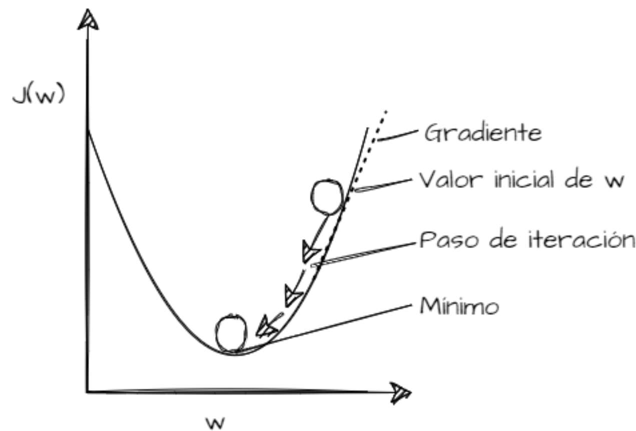


Figura 3.2: Intuición del método de descenso por el gradiente a partir del error cuadrático medio como función de costo J.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{\Delta J} \left(\boldsymbol{w} \right) \tag{3.12}$$

En cada iteración se realiza la operación de la ecuación actualizando $\boldsymbol{w}$. Dado que α >0, el valor de cada parámetro en $\boldsymbol{w}$ disminuirá y eventualmente la función de costo J convergerá a su valor mínimo en T iteraciones. Algo a notar es que en la ecuación 3.12, al ser una operación vectorial, cada componente se actualiza simultáneamente. La figura 3.2 muestra la intuición detrás de este método utilizando el error cuadrático medio como función de costo.

El método de descenso por el gradiente es también llamado el optimizador de la función de costo. En la literatura hay muchas variantes de este método que llevan a optimizadores más complejos y eficientes, tales como *descenso por el gradiente estocástico*(Kiefer and Wolfowitz, 1952), *Adam* (Kingma and Ba, 2017), *NAdam* (Zhang, 2018), entre otros.

3.4.5. Sobreajuste (overfitting)

Una vez optimizado la función de costo y haber encontrado la configuración de pesos $\boldsymbol{w}$, se pueden utilizar estos pesos aprendidos para hacer predicciones de ejemplos no antes vistos en el proceso de entrenamiento. Sin embargo, algo que suele ocurrir al usar solamente la función de costo para optimizar modelos de aprendizaje automático, es que tienden a sobreajustar los datos de entrenamiento reduciendo su error pero obteniendo malas predicciones en estos nuevos ejemplares. Por lo general, suele asociarse con la magnitud de los coeficientes en $\boldsymbol{w}$ siendo muy alta para estos casos. Otras razones de esto suele ser la dimensión de $\boldsymbol{w}$. En cuanto mayor la cantidad de parámetros de ajuste, existen más chances de que el modelo de aprendizaje automático memorice los datos de entrenamiento. Para solucionar esto se suele recurrir a varias técnicas con el fin de obtener modelos sencillos, pero que logren generalizar la tarea que se está queriendo clasificar. Uno de ellos es agregar a la función de costo un *término de regularización* sobre

las componentes de $\boldsymbol{w}$ para penalizar el modelo si la magnitud $\boldsymbol{w}$ crece demasiado. Por ejemplo para el error cuadrático medio:

$$J(\boldsymbol{w}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{w}}\left(\boldsymbol{x}_i\right) - y\right) + \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{3.13}$$

Donde $||\boldsymbol{w}||$ es la norma de $\boldsymbol{w}$, y λ es la fuerza en que la regularización impacta.

Otros métodos para el control del overfitting dependen del método de clasificación que se utilice, función de costo, parámetros de ajuste, entre otros que iremos explicando a medida que avancemos con los modelos utilizados en los experimentos.

3.4.6. Regresión logística

El primer modelo de clasificación que veremos es la *regresión logística*. Es un clasificador binario que busca modelar la probabilidad a posterior de c_1 a partir de una función sigmoide actuando sobre los vectores de entrada:

$$p\left(c_1|\boldsymbol{x}\right) = \sigma\left(\boldsymbol{w}^\top \boldsymbol{x}\right) \tag{3.14}$$

Luego $p\left(c_2|\boldsymbol{x}\right) = 1 - p\left(c_1|\boldsymbol{x}\right)$. Para un espacio de características $\boldsymbol{x}$ de D dimensiones este modelo consta de D parámetros para ajustar. En particular este método suele codificar las etiquetas de las clases c_1 y c_2, como 0 y 1, ya que permite algunas simplificaciones en la definición de su función de costo.

De esta manera podemos pensar la etiqueta y como observaciones discretas de una distribución Bernoulli.

$$p(y=1|\boldsymbol{x}) = h_{\boldsymbol{w}}(\boldsymbol{x}) \tag{3.15}$$

$$p(y=0|\boldsymbol{x}) = 1 - h_{\boldsymbol{w}}(\boldsymbol{x}) \tag{3.16}$$

$$p(y|\boldsymbol{x}) = (h_{\boldsymbol{w}}(\boldsymbol{x}))^y(1 - h_{\boldsymbol{w}}(\boldsymbol{x}))^{1-y} \tag{3.17}$$

Luego los pesos $\boldsymbol{w}$ se obtienen a partir de un optimizador del tipo del descenso por el gradiente minimizando la función de costo definida como:

$$J\left(\boldsymbol{w}\right) = \sum_{i=1}^{n} p(y_i|\boldsymbol{x}_i) \tag{3.18}$$

$$= -\sum_{i=1}^{n} y_i \log\left(h_{\boldsymbol{w}}\left(\boldsymbol{x}_i\right)\right) + (1-y_i)\log\left(1 - h_{\boldsymbol{w}}\left(\boldsymbol{x}_i\right)\right) \tag{3.19}$$

Algo a notar es que la función de costo se puede expresar como suma de penalizaciones individuales:

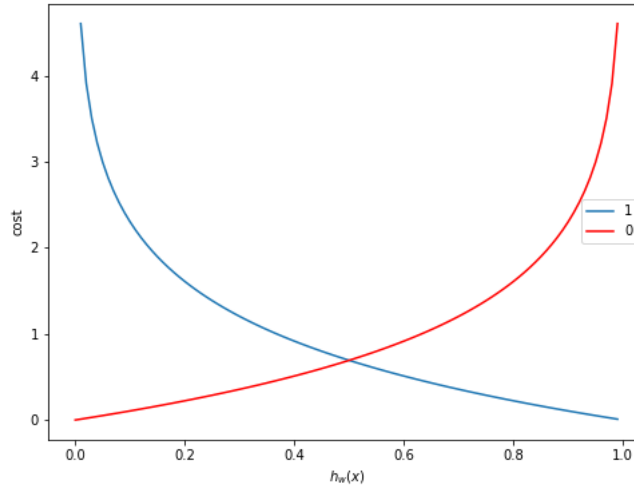


Figura 3.3: Gráfica de $h_{\boldsymbol{w}}(\boldsymbol{x})$ vs $\ell(\boldsymbol{x})$ separando por etiqueta.

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \ell\left(h_{\boldsymbol{w}}\left(\boldsymbol{x}_i\right), y_i\right) \tag{3.20}$$

Donde ℓ es la penalización a nivel de ejemplo:

$$\ell(\boldsymbol{x}, y) = \begin{cases} -\log\left(h_{\boldsymbol{w}}(\boldsymbol{x})\right) & y = 1 \\ -\log\left(1 - h_{\boldsymbol{w}}(\boldsymbol{x})\right) & y = 0 \end{cases} \tag{3.21}$$

Si analizamos la forma de ℓ para ambos casos podemos observar que, J captura la intuición de que mayores errores en la predicción deben recibir mayores penalizaciones.

Para el caso $y = 1$. La función de costo a nivel de ejemplo es la curva que se marca en color azul de la figura 3.3. En ella se puede ver que si $h_{\boldsymbol{w}}(\boldsymbol{x}) \to 0$ (la función de predicción asigna a $\boldsymbol{x}$ una probabilidad cercana a 0), entonces el costo incrementa y por lo tanto, $\ell(\boldsymbol{x}, y) \to \infty$. Si $h_{\boldsymbol{x}}(\boldsymbol{x}) \to 1$, entonces el costo disminuye y $\ell(\boldsymbol{x}, y) \to 0$. De manera similar ocurre para el caso $y = 0$ y la curva marcada en rojo.

3.4.7. Redes neuronales

Una red neuronal artificial es un modelo computacional que en los últimos tiempos se ha convertido en el estado del arte y el enfoque más utilizado en la mayoría de las áreas relacionadas con el diseño e implementación de sistemas predictivos. Estas áreas incluyen, entre otras el procesamiento del habla, la visión por computadoras, el procesamiento del lenguaje natural y la toma de decisiones y control en agentes situados.

El término surge bajo de los intentos de definir un modelo matemático de una red neuronal biológica (Pitts, 1990). No obstante, enfocaremos nuestra atención en un tipo

específico de red neuronal reconocida por su alto valor práctico, denominado *perceptrón multicapa*.

Las redes neuronales surgen de la necesidad de aprender la representación de los datos cuya disposición en el espacio requieren de una modelización no lineal mucho más compleja. Similar a un modelo lineal, las redes neuronales reciben un vector de entrada $\boldsymbol{x}$ donde cada componente x_i es ponderada mediante una combinación lineal por el peso w_i en $\boldsymbol{w}$. La única diferencia es que al resultado se le aplica una función de transformación no lineal g conocida como *función de activación*.

$$z_{\boldsymbol{w}}\left(\boldsymbol{x}\right) = g\left(\sum_{i=0}^{D} w_i x_i\right) \tag{3.22}$$

Notar además que se puede pensar está red neuronal simple como un modelo de regresión logística si la función de activación es la sigmoide, es decir, si $g = \sigma$.

Perceptrón multicapa

Este modelo puede describirse como una serie de transformaciones similares a la ecuación 3.22. Primero construimos $M + 1$ combinaciones lineales a partir de las componentes del ejemplo de entrada $\boldsymbol{x} = (x_0, ..., x_D)$.

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{3.23}$$

Donde $j = 0, ..., M$. El supra-índice (1) indica que los parámetros correspondientes $w_{ji}^{(1)}$ están en la primera capa de la red. Luego cada a_j es transformado por una función de activación no lineal g.

$$z_j = g(a_j) \tag{3.24}$$

Estas cantidades corresponden a las salidas de cada neurona o unidad. Luego para estos valores construimos nuevamente K combinaciones lineales obteniendo K neuronas de salida.

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \tag{3.25}$$

Donde $k = 1, ..., K$. Esta transformación corresponde con la segunda capa de la red, como lo indica el supra-índice (2). Finalmente, las unidades de salidas realizan su predicción y_k activando los datos por última vez con otra función de activación. Esta última depende de la codificación de la variable objetivo, y si se está trabajando sobre un problema de regresión o clasificación. Para el caso de regresión la función de activación de salida suele ser la función de identidad, para clasificación binaria una sigmoide, y para multiclase una softmax.

$$y_k = \sigma\left(a_k\right) \tag{3.26}$$

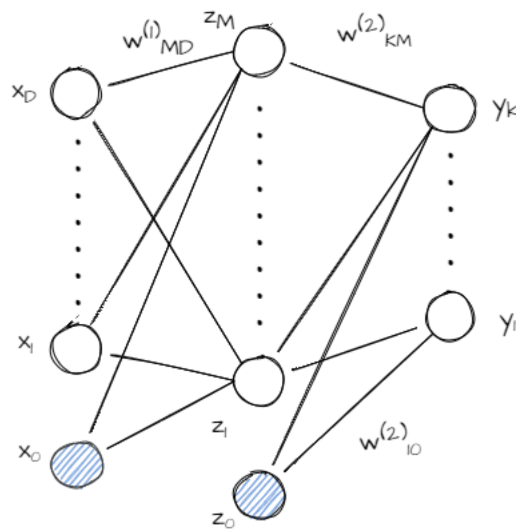


Figura 3.4: Perceptrón multicapa de una sola capa oculta.

Podemos combinar estas dos transiciones y dar una expresión del modelo completo en función de $\boldsymbol{x}$ y $\boldsymbol{w}$.

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} g\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right) \tag{3.27}$$

Por lo tanto, un modelo neuronal es simplemente una función no lineal que depende de un ejemplo $\boldsymbol{x}$, que retorna un vector de salida $\boldsymbol{y}$, y que es controlado por un vector de parámetros $\boldsymbol{w}$, donde las dimensiones de estos vectores son $D+1$, K, y $(D+1 \times M+1)$ respectivamente. Esta función también puede representarse a partir de un gráfico de red como el que se vé en la figura 3.4. El proceso de evaluar un ejemplo puede interpretarse como una propagación hacia adelante (*forward propagation* en inglés), que transmite la información a través de la red. Por último, algo a notar son las componentes $w_{j0}^{(1)}$ y $w_{k0}^{(2)}$ que se corresponden con el sesgo de los modelos lineales.

Backpropagation

Una vez determinado explícitamente el modelo, es necesario encontrar el vector $\boldsymbol{w}$ que ajuste a los datos de entrenamiento y sea capaz de generalizar a ejemplares nuevos. Para ello, similar a los modelos lineales se debe definir una función objetivo a optimizar, es decir, una función de costo en compañía de una regularización para prevenir sobreajuste, a la cual minimizar mediante algún método del tipo descenso por el gradiente. El inconveniente para este caso es que la función objetivo, en particular la de costo, depende del modelo en cuestión para comparar su predicción con la etiqueta real. Lo cual dificulta el cálculo del gradiente y el cálculo de los parámetros en la red. Es por eso que, el método de *backpropagation*, propaga el error de la capa de salida

hacia las capas iniciales derivando el gradiente de manera iterativa y asignándole a cada neurona una porción de error en relación a su aporte generado en la salida original.

En resumen el algoritmo de aprendizaje de una red neuronal consiste en:

- Inicializar el vector de pesos $\boldsymbol{w}$ aleatoriamente.
- Realizar forward propagation sobre una entrada para obtener predicciones.
- Calcular el error cometido en base a una función de optimización (función de perdida y regularización).
- Realizar backpropagation para propagar el error a los parámetros de cada interconexión neuronal actualizándolos mediante descenso por el gradiente.
- Repetir los pasos para cada ejemplo de entrada.

Funciones de activación

Hasta el momento no mencionamos la forma de la función de activación g siendo la protagonista para obtener un modelo no lineal. Algunas de las funciones más comunes es la *sigmoide* (figura 3.7) introducida en la sección 3.4.2. Esta toma valores reales y los mapea al rango $[0, 1]$ siendo muy utilizada en procesos de clasificación.

Otras funciones muy usadas son la *tanh* (tangente hiperbólica, figura 3.6) y *ReLU* (unidad rectificadora lineal, figura 3.5).

$$\tanh(x) = \frac{2}{1 + \exp(-2x)} - 1 \tag{3.28}$$

$$ReLU\,(x) = \text{máx}(0, x) \tag{3.29}$$

Notar que la tangente hiperbólica realiza un cambio de escala de la sigmoide mapeando valores reales al rango $[-1, 1]$ como se muestra en la ecuación 3.30.

$$\tanh(x) = 2\sigma(2x) - 1 \tag{3.30}$$

En el caso de la *ReLU* cumple el rol de quitar los valores negativos dejando invariantes los positivos.

Variantes de arquitectura y métodos de entrenamiento

El modelo y procedimiento definido en las secciones 3.4.7, 3.4.7, conforman una de las variantes más simples de una red neuronal multicapa. En particular existen formas de alterar este proceso que permiten dar cierta flexibilidad al predictor en algunas circunstancias. Una de ellas es la posibilidad de “apagar“ neuronas aleatoriamente con el fin de evitar el sobreajuste. Por lo general a mayor cantidad de capas y neuronas, hay más posibilidad de sobreajustar el conjunto de entrenamiento. Es por eso que este método, denominado *dropout*, puede reducir la complejidad del modelo dinámicamente durante el proceso de entrenamiento.

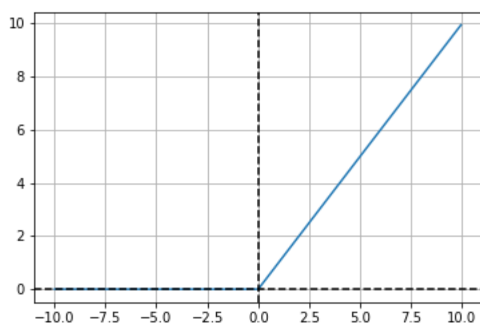


Figura 3.5: Función de activación $ReLU$.

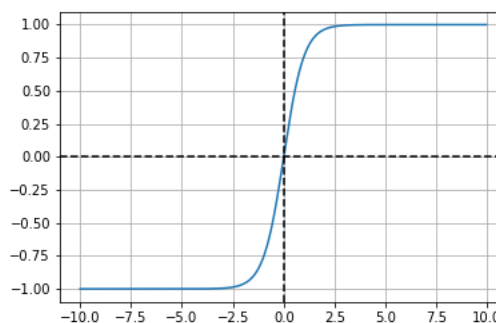


Figura 3.6: Función de activación $tanh$.

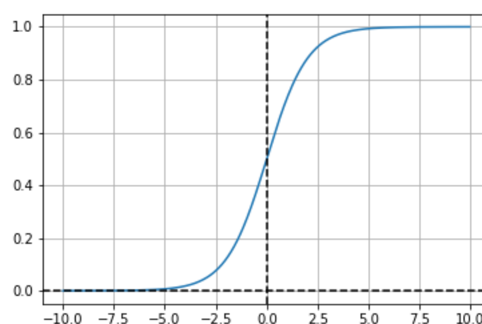


Figura 3.7: Función de activación σ.

Otra variante es el uso de *batches* al actualizar $\boldsymbol{w}$ recopilando el error de varios ejemplares para su posterior propagación a diferencia de procesar a nivel de ejemplos.

También es posible agregar normalización de características en las capas iniciales de la red neuronal. El uso de escalas distintas en las componentes de los vectores de entrada puede llevar a ponderar el cálculo del error cometido en una dimensión por sobre otra.

3.4.8. XGBoost

El último modelo de aprendizaje que veremos es *XGBoost* (eXtreme Gradient Boosting en inglés) (Chen and Guestrin, 2016), un modelo de *ensemble* que consiste en combinar la respuesta de varios modelos para obtener una final. Es decir, en lugar de consultar la predicción de un modelo, K de ellos son utilizados. Este tipo de heurísticas y en particular la de XGBoost han sido enormemente utilizadas por científicos de datos siendo el estado del arte en múltiples problemas de aprendizaje automático. Desde obtener un gran desempeño en *benchmarks*, hasta lograr los primeros puestos en competencias de aprendizaje automático como Netflix prize, Kaggle, y KDDCup.

Otro de los aspectos importantes detrás del éxito de XGBoost es su escalabilidad

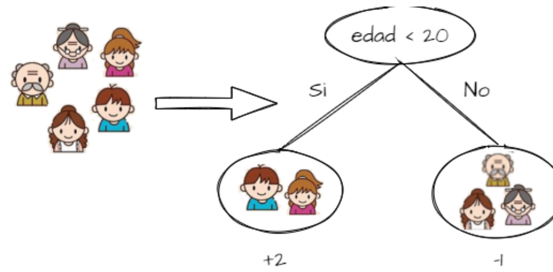


Figura 3.8: Árbol que decide por edad de las personas si le interesa un videojuego de computadora. Imagen adaptada de (Chen and Guestrin, 2016).

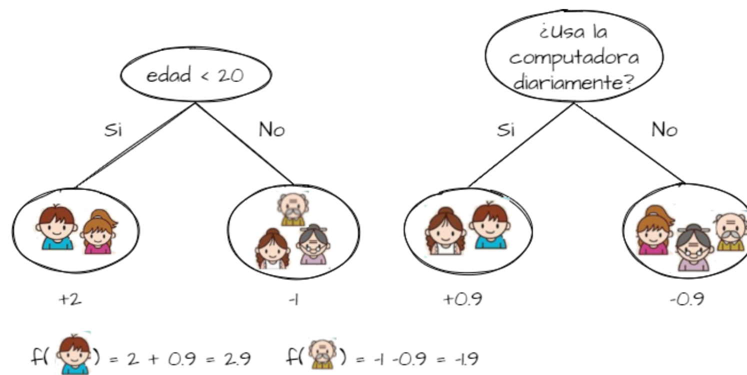


Figura 3.9: Dos arboles que en base a la edad y uso de la computadora deciden si una persona le interesa un videojuego. Imagen adaptada de (Chen and Guestrin, 2016).

en todo tipo de escenarios, siendo muy rápido para manejar consultas de billones de ejemplos en dominios donde se cuenta con una cantidad limitada de recursos. Esto motivó aún más su uso en esta tesis, principalmente por la necesidad de modelos con un rápido tiempo de respuesta durante el proceso de grounding heurístico.

A diferencia de la regresión logística y las redes neuronales, XGBoost se basa en una estructura completamente distinta para la definición de sus modelos. Estas consisten de árboles de decisión donde cada uno aporta su predicción a un ejemplo de entrada. El objetivo de utilizar varios clasificadores es con la iniciativa de que se complementen entre sí. Por ejemplo, supongamos que queremos averiguar si una persona le gusta un videojuego de computadora. Los datos de entrada serían vectores cuyas componentes representen su edad y la frecuencia en la que utilizan el ordenador. Si mantenemos dos árboles para cada una de estas componentes podemos complementar sus respuestas (figura 3.8) en lugar de tener solo uno (figura 3.9).

Nuevamente, consideremos un conjunto de datos etiquetados $\{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$ donde cada $\boldsymbol{x}_i \in \mathbb{R}^D$. Un modelo de *ensemble* del tipo *boosting tree* puede escribirse matemáticamente como una suma de K funciones que dependan de un ejemplo $\boldsymbol{x}_i$.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(\boldsymbol{x}_i) \tag{3.31}$$

Donde cada $f_k \in \mathcal{F}$, siendo $\mathcal{F}$ la familia o espacio de todos los árboles. Ahora bien, ¿Cómo están definidos cada uno de los árboles de decisión f_k? ¿De qué forma son los elementos en $\mathcal{F}$? ¿Cuáles son los parámetros del modelo?.

Debemos representar matemáticamente la estructura del árbol que asigne un ejemplo de entrada $\boldsymbol{x}$ a una correspondiente hoja y por ende a un puntaje como vimos en las figuras 3.8, y 3.9. Los parámetros a encontrar son tales puntajes y se suele referir a ellos como los pesos del modelo. Otro factor importante es la manera en que se determina la estructura que debe tener cada árbol.

Formalmente $\mathcal{F} = \{f(\boldsymbol{x}) = w_{q(\boldsymbol{x})}\}$ donde $q : \mathbb{R}^D \to T$ y $\boldsymbol{w} \in \mathbb{R}^T$. Aquí q representa el mapeo de un ejemplo $\boldsymbol{x}$ a una hoja en el árbol. T es el número de hojas de este último. Y $w_{q(\boldsymbol{x})}$ es una componente de $\boldsymbol{w}$ que representa el puntaje asociado al ejemplo $\boldsymbol{x}$ según la estructura del árbol determinada por q.

Luego como los modelos vistos hasta ahora, para determinar los pesos $\boldsymbol{w}$ y la estructura del árbol es necesario definir una función objetivo a minimizar compuesta por una función de costo y su regularización. No obstante, aprender la estructura es mucho más complejo que sólo utilizar algún método del tipo descenso por el gradiente. Es por eso que la estrategia utilizada por los algoritmos de *boosting* es agregar de manera iterativa un nuevo árbol a lo aprendido, en particular, aquellos árboles que optimicen algún objetivo. Esto da lugar a definir la predicción del modelo al paso (t).

$$
\begin{aligned}
\hat{y_i}^{(0)} &= 0 \\
\hat{y_i}^{(1)} &= f_1(\boldsymbol{x}_i) = \hat{y_i}^{(0)} + f_1(\boldsymbol{x}_i) \\
&\dots \\
\hat{y_i}^{(t)} &= \sum_{k=1}^{t} f_k(\boldsymbol{x}_i) = \hat{y_i}^{(t-1)} + f_t(\boldsymbol{x}_i)
\end{aligned}
$$

Luego, la función a optimizar en la iteración (t) se define como:

$$
\begin{aligned}
\mathcal{L}^{(t)} &= \sum_{i=1}^{n} \ell\left(y_i, \hat{y_i}^{(t)}\right) + \Omega(f_t) && (3.32) \\
&= \sum_{i=1}^{n} \ell\left(y_i, \hat{y_i}^{(t-1)} + f_t(\boldsymbol{x}_i)\right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 && (3.33)
\end{aligned}
$$

Con $\Omega(f_t)$ la regularización para el predictor. γ y λ son parámetros de Ω que intensifican o disminuyen la regularización.

El último requerimiento es optimizar $\mathcal{L}$ para un paso arbitrario y el significado de la función de costo ℓ. XGBoost admite cualquier función de perdida que aproxima a partir de una expansión de Taylor de segundo orden.

$$
\mathcal{L}^{(t)} = \sum_{i=1}^{n} [\ell\left(y_i, \hat{y_i}^{(t-1)}\right) + g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i)] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \qquad (3.34)
$$

Donde g_i y h_i son definidas como:

$$g_i = \partial_{\hat{y_i}^{(t-1)}} \ell \left(y_i, \hat{y_i}^{(t-1)} \right) \tag{3.35}$$

$$h_i = \partial^2_{\hat{y_i}^{(t-1)}} \ell \left(y_i, \hat{y_i}^{(t-1)} \right) \tag{3.36}$$

Algo a notar de la ecuación 3.34 es que $\sum_{i=1}^{n} \ell \left(y_i, \hat{y_i}^{(t-1)} \right)$ es una suma realizada hasta un paso anterior, estando ya calculado por el algoritmo y, por lo tanto, se considera constante para el paso actual (t). Omitiendo dicho término se obtiene:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{3.37}$$

$$= \sum_{i=1}^{n} [g_i w_{q(\boldsymbol{x}_i)} + \frac{1}{2} h_i w_{q(\boldsymbol{x}_i)}^2] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{3.38}$$

Se puede identificar la penalización de la función de costo que se agrega al peso w_j como $G_j = \sum_{i \in I_j} g_i$ y $H_j = \sum_{i \in I_j} h_i$ con $I_j = \{i | q(\boldsymbol{x}_i) = j\}$ (Conjunto de índices de ejemplares que son asignados a la j-ésima hoja). Es decir, cada componente w_j es penalizada $|I_j|$ veces por g_i y por h_i. Luego la ecuación 3.38 se puede escribir como:

$$\mathcal{L}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2] + \gamma T \tag{3.39}$$

Finalmente, igualando a 0 y despejando w_j se obtienen los w_j óptimos para un árbol con estructura $q(\boldsymbol{x})$ y la mejor forma de evaluar su performance son los de las ecuaciones 3.40 y 3.41.

$$w_j^* = -\frac{G_j}{H_j + \lambda} \tag{3.40}$$

$$\mathcal{L}^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{3.41}$$

En resumen, para una estructura de árbol dada, cada ejemplo es evaluado asignándole una hoja. Se recopilan los ejemplos que pertenecen a cada una de ellas y se penaliza según los estadísticos g_i y h_i y finalmente obtener el error cometido por el árbol $\mathcal{L}^*$. Aquel árbol que tenga menor error es el mejor para ser incluido en la iteración actual.

3.5. Codificación de características

Una vez revisado los modelos de aprendizaje, otro factor importante para obtener un buen modelo de aprendizaje automático es la representación de los datos de entrada.

Preparar los ejemplares para acoplarse apropiadamente a un algoritmo con el fin de mejorar la performance del modelo es una de las tareas que los científicos de datos disponen la mayor parte de su atención y tiempo. Alguna de las obligaciones que incluye son la imputación de valores faltantes, manejo de valores atípicos, estandarización y escalado, transformación de características numéricas a categóricas, y codificación.

En particular pondremos nuestro foco en la codificación de características desarrollando métodos del tipo *one-hot* y *word embeddings*.

3.5.1. One-hot encoding

A menudo los datos disponibles presentan características que no están dadas en un espacio continuo sino más bien categórico. Por ejemplo, se podría describir el continente de un país por las clases *enAsia*, *enAfrica*, *enEurope*, *enAmerica*, *enOceanía*, *enAntártida* siendo eficientemente representables con enteros $[0, 1, 2, 3, 4, 5]$. De esta manera, si tenemos un variable que puede tomar una serie de valores categóricos, entonces se puede enumerar cada uno de sus objetos. Esta representación es conocida como *ordinal encoder*.

$$\begin{bmatrix} & enContinente \\ Argentina & 3 \\ Brasil & 3 \\ España & 2 \\ USA & 3 \\ Italia & 2 \end{bmatrix}$$

Otra opción es usar un esquema *one-of-K* que transforma una característica categórica con N clases, en N categorías binarias con una de ellas 1 y el resto 0. Para el ejemplo anterior tendríamos:

La misma técnica es válida si se quisiera representar oraciones o documentos del lenguaje natural. Supongamos que tenemos los documentos.

D_1: "El sol es una estrella, no es un planeta."
D_2: "La tierra es un planeta."

Basado en estos dos textos, un vocabulario de 10 palabras distintas es construido. Por lo tanto, cada documento es representado como un vector de 10 dimensiones.

-	El	sol	es	una	estrella	no	un	planeta	la	tierra
D_1	1	1	2	1	1	1	1	1	0	0
D_2	0	0	1	0	0	0	1	1	1	1

Observar que para este caso una codificación ordinal no sería adecuada dado que se deberían enumerar todas las palabras del vocabulario.

3.5.2. Vectores de palabras (Word embeddings)

Un vector denso de palabras (comúnmente conocido como word embedding, en inglés) es una de las técnicas de representación de vocabulario más popular en el área del procesamiento de lenguaje natural.

Si bien hay distintas formas de representar palabras en una serie de documentos, los word embeddings proveen otra perspectiva que busca encontrar una representación vectorial compacta donde cada dimensión logre capturar las propiedades subyacentes y latentes de la palabra. De esta manera, los embeddings son superiores a una representación que permanece en un nivel poco profundo.

Su principal característica es que las expresiones relacionadas entre sí (ya sea por contexto, semántica o sintaxis) se encuentren cercanas en el espacio vectorial al cual se proyectan. Por ejemplo, si tenemos las oraciones: "Que tengas una buena mañana" y "Que tengas una buena tarde". Difieren semánticamente solo por las palabras "mañana" y "tarde". No obstante, se utilizan en contextos similares, con lo cual esperaríamos que estén cercanas vectorialmente.

Otro ejemplo muy famoso que deja en evidencia el concepto de *analogía* en word embeddings, es el que se muestra en la Figura 3.10. En ella se identifican por medio de barras de colores la dimensión de los vectores asociados a las palabras *king*, *man*, *woman*, *queen*, y la operación vectorial *king* - *man* + *woman*. Cada dimensión se encuentra en la escala de $[-2, 2]$ y se le asigna el color de rojo para valores cercanos al extremo derecho, azul para las del extremo izquierdo, y blanco para las que se encuentren alrededor del 0. Algo a notar es que *man* y *woman* son mucho más similares de lo que cada uno es con *king* o *queen*. Otro aspecto interesante son las franjas rojas y azul que se muestran para cada uno de los ejemplos, indicando que son parecidos en tal dimensión. Probablemente, corresponda a alguna característica humana aunque se desconoce que puedan llegar a significar debido a la poca interpretabilidad del espacio al cual se proyectan.

Aun así, lo importante es la comparación entre palabras. Dado que ahora tienen una representación vectorial, se pueden sumar o restar obteniendo nuevos resultados que forman parte del espacio. Este es el caso del vector dado por *king* - *man* + *woman*. No conocemos que dimensiones exactamente capturan que rey es utilizado para identificar al monarca de un reino, pero se puede substraer el carácter masculino del vector *man* y agregar las de *woman* esperando que se aproximen a las de *queen*. Sorprendentemente, de un total de 400000 palabras sobre las cuales se realizó el experimento, reina fue la palabra más cercana. Los detalles del mismo se pueden ver desde (*The illustrated Word2Vec*, 2019).

Esto refleja el gran potencial que tienen los word embeddings y su importancia para estudiarlos como métodos de codificación. A continuación analizaremos como se calculan dichas representaciones y que algoritmos fueron finalmente utilizados en esta tesis.

Modelos de lenguaje

Después de ver el potencial de los embeddings, surge la pregunta, ¿Cómo funcionan internamente? ¿De qué manera se obtienen estos vectores de palabras? En (Firth, 1957) se mostró que el significado de una palabra está determinado según sus palabras vecinas. Si contemplamos la oración "Que tengas un buen" y se quisiese averiguar cuál es la

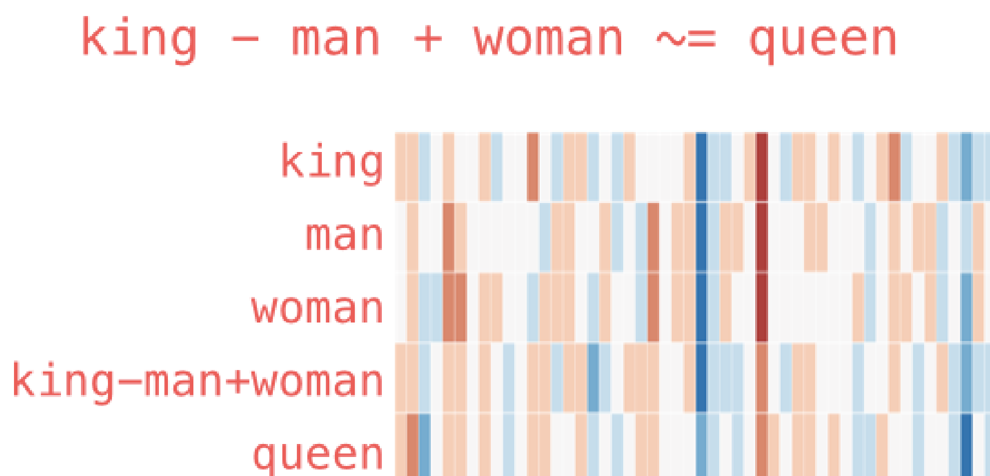


Figura 3.10: Vectores de palabras para “king“, “man“, y “woman“.

expresión que le sigue, lo más probable es que las palabras días, tarde, noche, o semana, hayan sido las primeras candidatas. Este tipo de tareas en las que se intenta predecir información de la oración a partir del contexto o viceversa son denominadas tareas de pretexto y son utilizadas para entrenar un clasificador que resuelva esta tarea, únicamente para preservar su representación a nivel de palabras.

Skipgram

Es un método desarrollado en (Mikolov et al., 2013) cuyo objetivo es, para un vocabulario de tamaño W donde una palabra es identificada por su índice $w \in \{1, ...W\}$, aprender una representación vectorial para cada w. Estas representaciones son entrenadas para predecir palabras vecinas a partir de una actual. Es decir, dado un largo corpus de entrenamiento, considerarlo como una sola secuencia de palabras $w_1, ...w_T$ y maximizar la expresión en 3.42.

$$\frac{1}{T} \sum_{n=1}^{T} \sum_{c \in C_t} \log P(w_c | w_t) \tag{3.42}$$

Donde C_t es el conjunto de índices de palabras que rodean a w_t. La probabilidad de observar una palabra del contexto w_c dado w_t es modelada a partir de sus vectores representación. Por el momento, abstraeremos esto a partir de una función *score* s que mapea pares (palabra, contexto) a un puntaje en $\mathcal{R}$. Una forma posible de definir la probabilidad de una palabra en el contexto es por medio de la función *softmax*.

$$P(w_c | w_t) = \frac{e^{s(w_t, w_c)}}{\sum_{j=1}^{W} e^{s(w_t, j)}} \tag{3.43}$$

No obstante, este cálculo es bastante costoso computacionalmente siendo que al final solo es de interés predecir una palabra del contexto w_c dado una palabra w_t. Es

por eso que el problema puede ser visto como una tarea de clasificación binaria en lugar de una multiclase. El objetivo es predecir independientemente la presencia o ausencia de palabras de contexto. Para una palabra en la posición t, consideramos todos los contextos como ejemplos positivos, y palabras seleccionadas aleatoriamente del vocabulario, como negativos. Para un contexto c, utilizando la función de costo *binary logistic*, obteniendo la siguiente función de probabilidad:

$$\log\left(1 + e^{-s(w_t, w_c)}\right) + \sum_{n \in \mathcal{N}_{t,c}} \log\left(1 + e^{s(w_t, n)}\right) \tag{3.44}$$

Donde $\mathcal{N}_{t,c}$ es una muestra aleatoria de ejemplares negativos tomados del vocabulario. Si denotamos la función logística como $\ell : x \mapsto \log\left(1 + e^{-x}\right)$, podemos reescribir la función a optimizar del problema como:

$$\sum_{n=1}^{T} \sum_{c \in C_t} \ell\left(s(w_t, w_c)\right) + \sum_{n \in \mathcal{N}_{t,c}} \ell\left(-s(w_t, n)\right) \tag{3.45}$$

Una parametrización natural para s entre una palabra w_t y un contexto w_c es a partir del uso de vectores de palabras. Definimos dos vectores $\boldsymbol{u}_w$ y $\boldsymbol{v}_w$ en $\mathbb{R}^D$ también conocidos como vectores *input* y *output*. Es decir, tenemos vectores $\boldsymbol{u}_{w_t}$ y $\boldsymbol{v}_{w_c}$ asociados a palabras w_t y w_c. Luego el *score* puede computarse como $s\left(w_t, w_c\right) = \boldsymbol{u}_{w_t}^{\top} \boldsymbol{v}_{w_c}$.

Para ejemplificar como es el proceso de entrenamiento de este modelo, tomemos la secuencia "Me gustaría dos rebanadas de pizza mozzarella". Podemos pensar un contexto como una ventana que se desliza a través del conjunto de entrenamiento. La figura 3.11 muestra las 3 posibles ventanas de tamaño 5 que se pueden obtener a partir de la secuencia deslizándose con un paso de 1.

Con cada ventana, seleccionamos aquella palabra que cargue más valor informativo, que por lo general suele ser la del medio, y la utilizamos para predecir sus vecinas. De esta manera, generamos un conjunto de entrenamiento compuesto por pares (palabra, palabra de contexto) donde etiquetamos con 1, si ambas son vecinas, y 0 en caso contrario. El resultado final para cada contexto se puede ver en la figura 3.12. Esta es la simplificación del problema para trabajar sobre una tarea de clasificación binaria. En lugar de ser multiclase cuya etiqueta sería la palabra de contexto, la incluimos como característica y reemplazamos las anotaciones por 1's y 0's. Ahora bien, con nuestros datos actuales solo tendríamos etiquetas con 1's. En tal caso, el modelo en lugar de aprender lo que necesitamos, estaría sesgado a siempre predecir afirmativamente. Es por eso que agregar ejemplos negativos de manera aleatoria (figura 3.13) a cada w_t resulta una buena idea.

Luego, al inicio del proceso de entrenamiento se crean dos matrices con valores aleatorios, una para los embeddings y otra para los contextos. Estas dos matrices de tamaño $V \times D$, tienen una fila por cada palabra en el vocabulario. En este caso, D es la cantidad de dimensiones que tendrán sus representaciones. En cada paso de entrenamiento, se selecciona un ejemplo positivo y sus correspondientes ejemplos negativos (figura 3.14). Este grupo es evaluado por el modelo a partir de los vectores representación *input* y *output* según el producto escalar entre ellas y su conversión en probabilidades por medio de la función sigmoide. A partir de la etiqueta real, se puede determinar el puntaje de error

Me gustaría dos rebanadas de pizza mozzarella

Me gustaría dos rebanadas de pizza mozzarella

Me gustaría dos rebanadas de pizza mozzarella

Figura 3.11: Todas las ventanas de contexto con tamaño 5 que se pueden formar con la frase "Me gustaría dos rebanadas de pizza mozzarella".

Input	Output	Label
dos	Me	1
dos	gustaría	1
dos	rebanadas	1
dos	de	1
rebanadas	gustaría	1
rebanadas	dos	1
rebanadas	de	1
rebanadas	pizza	1
de	dos	1
de	rebanadas	1
de	pizza	1
de	mozzarella	1

Figura 3.12: Material de entrenamiento obtenido de las ventanas.

Input	Output	Label
dos	Me	1
dos	quiero	0
dos	piezas	0
dos	gustaría	1
dos	papas	0
dos	gaseosa	0
dos	rebanadas	1
dos	sandwich	0
dos	monedas	0
dos	de	1
dos	la	0
dos	pasar	0

Figura 3.13: Selección aleatoria de 2 ejemplos negativos por cada positivo.

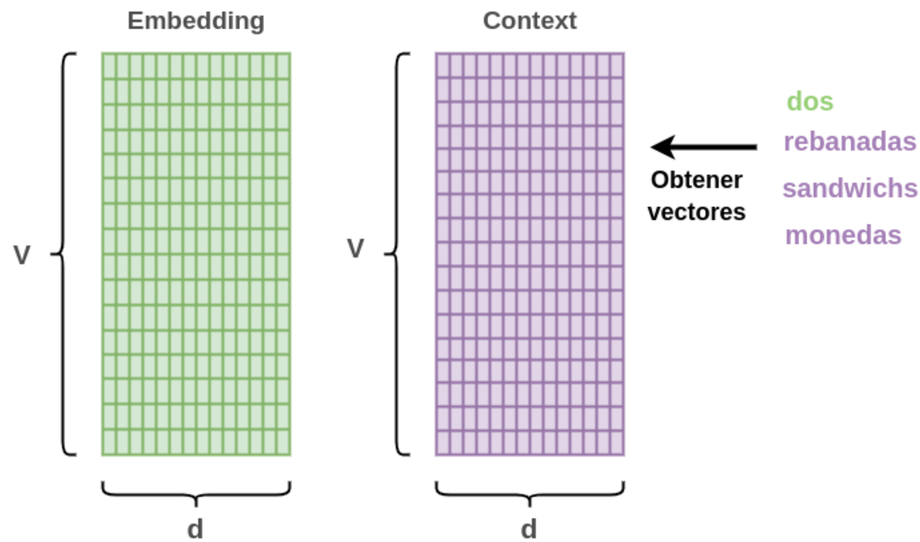


Figura 3.14: Matrices para los embeddings y contextos.

para cada ejemplo y finalmente actualizar los parámetros de las matrices de embeddings y contexto. Eso concluye un paso de entrenamiento, que se repite por cada grupo de ejemplos positivos y negativos por una cierta cantidad de épocas. Una vez concluido el proceso de entrenamiento, la matriz de embeddings contiene las representaciones finales.

Modelos de lenguaje a nivel de subpalabra

El modelo de skipgrams al usar vectores representación por cada palabra se ignora su estructura interna. Esto fue lo que inspiró el estudio de modelos a nivel de subpalabra en (Bojanowski et al., 2016). La principal característica de estos modelos radica en la función de score s. En lugar de considerar una palabra w en su totalidad, esta es dividida en n-gramas de caracteres. Para delimitar inicio y fin de una palabra se agregan símbolos especiales < y > respectivamente. Esto a su vez permite distinguir prefijos y sufijos de otras secuencias de caracteres. Por último la palabra w también es incluida como un n-grama o secuencia especial. Por ejemplo, la palabra *color* y $n = 4$ estaría representada por los 4-gramas:

`<col, colo, olor, lor>`

Junto a la secuencia completa `<color>`. Notar que la secuencia `<olor>` correspondiente a la palabra *olor* es diferente al 4-grama `olor`. Luego lo único que queda restante es modificar la función *score* s del método skipgram por la ecuación 3.46.

$$s\left(w,c\right) = \sum_{g\in\mathcal{G}_w} \boldsymbol{z}_g^\top \boldsymbol{v}_c \tag{3.46}$$

Donde, $\mathcal{G}_w \subset \{1,...G\}$ es el conjunto de n-gramas de la palabra w, $\boldsymbol{z}_g$ el vector representación del n-grama g, y G la cantidad de n-gramas en total.

Este modelo permite compartir representaciones entre palabras logrando aprender una confiable codificación a palabras poco comunes siendo útil para los experimentos de esta tesis en la tarea de codificación de planes relajados.

Representación vectorial de oraciones

Como cierre de esta sección presentaremos algunos métodos sencillos de codificación de oraciones a partir de los embeddings orientadas a tareas de clasificación que involucren manipulación de texto.

Concatenación. Este método consiste en concatenar los vectores de las palabras que conforman la oración a codificar obteniendo un vector de largo igual a la suma de las dimensiones de los vectores individuales. En este caso, el problema evidente que surge es la representación de oraciones con distinto largo y el tamaño del vector resultante.

Promedio. Como su nombre lo indica, este método computa el centroide de los embeddings de todas las palabras que conforman una oración. Usualmente, esta codificación resulta ser adecuada siempre y cuando los vectores de las palabras se encuentren en la misma escala, lo cual no siempre suele ser el caso. Otro problema menos evidente es la posibilidad de ocurrencia de palabras poco representativas, pero que tengan mucho peso en la suma debido a la frecuencia en que ocurren. Si en la oración la palabra que buscamos ocurre pocas veces puede verse opacada.

Promedio normalizado. Por último, una mejora de la representación anterior es la normalización por norma L2 de cada vector previo a realizar el promedio.

Pueden encontrarse otras maneras de codificar los datos a través de (Iacobacci et al., 2016) donde se explican como utilizar word embeddings para desambiguación de sentidos, proponiendo distintos métodos de codificación de instancias de entrenamiento a partir de un modelo de lenguaje. Esto muestra que el uso de representaciones a nivel de palabra en la codificación de oraciones suelen ser patrones comunes al momento de incluirlo en una arquitectura supervisada.

3.6. Métricas de clasificación

Con el fin de evaluar que tan bien se desempeña el modelo se contrastan resultados predichos por el clasificador contra otros previamente anotados que no formaron parte como material de entrenamiento. Para un problema de clasificación binaria donde los positivos corresponden con la clase 1, y los negativos con la 0 dispusimos de las siguientes métricas.

Precisión. Cociente entre verdaderos positivos y el total de predicciones positivas.

$$precision = \frac{TP}{TP + FP} \tag{3.47}$$

Recall. Cociente entre verdaderos positivos y el total de positivos.

$$recall = \frac{TP}{TP + FN} \tag{3.48}$$

F-score (F_β). El inconveniente de las primeras dos métricas es que ambas alcanzan el $100\,\%$ en modelos triviales. Es decir aquellos que predicen siempre la clase positiva

obtienen un valor máximo en recall, mientras los que predicen siempre la clase negativa consiguen un valor máximo en precisión. La F_β mide el desbalance entre las dos anteriores ponderando una por sobre la otra de acuerdo al valor de β pero esta vez penalizando los modelos triviales.

$$F_\beta = (1+\beta^2)\frac{precision.recall}{\beta^2 precision + recall} \tag{3.49}$$

Valores comunes de β suelen ser $0{,}5$ (priorizar precisión), 1 (ninguna prioridad), $1{,}5$ (priorizar recall).

H-score (H_β). Durante el desarrollo de la tesis se definió una variante de la F_β, que denominamos H_β, permitiendo lidiar con datos desbalanceados. Si la cantidad de datos negativos es mucho mayor que la de positivos, la precisión puede verse más comprometida que la recall, al haber más oportunidades de cometer un falso positivo. Para resolver este problema, en vez de considerar precisión y recall en la ecuación de F_β, se utilizó la *tasa de verdaderos positivos y negativos* (TPR, y TNR).

$$TPR = \frac{TP}{TP+FN} \tag{3.50}$$

$$TNR = \frac{TN}{TN+FP} \tag{3.51}$$

$$H_\beta = (1+\beta^2)\frac{TNR.TPR}{\beta^2 TNR + TPR} \tag{3.52}$$

Notar que TNR continua penalizando por la cantidad de falsos positivos de manera similar a precisión. No obstante, la métrica es relativa a la cantidad de negativos. Si el desbalance es sobre los negativos, entonces la métrica no se ve comprometida. De manera similar a F_β, un valor de $0{,}5$ prioriza TNR, de 1 considera un balance entre ambas métricas, y $1{,}5$ prioriza TPR.

Matrices de confusión. En problemas de clasificación, las matrices de confusión permiten evaluar la calidad de la clasificación comparando las predicciones del modelo, con las etiquetas reales. Por definición, una matriz de confusión C es tal que C_{ij} es igual al numero de ejemplos cuya etiqueta es i, y la predicción es j. Por lo tanto para un problema de clasificación binaria tenemos que:

- La cantidad de verdaderos negativos (TN) es $C_{0,0}$.
- La cantidad de falsos negativos (FN) es $C_{1,0}$.
- La cantidad de verdaderos positivos (TP) es $C_{1,1}$.
- La cantidad de falsos positivos (FP) es $C_{0,1}$

La figura 3.15 muestra una matriz de confusión para un problema de clasificación binaria donde las columnas representan las predicciones, y las filas la etiqueta real. Para este ejemplo se pueden observar 60 TP, 5 FP, 187 TN, y 46 FN.

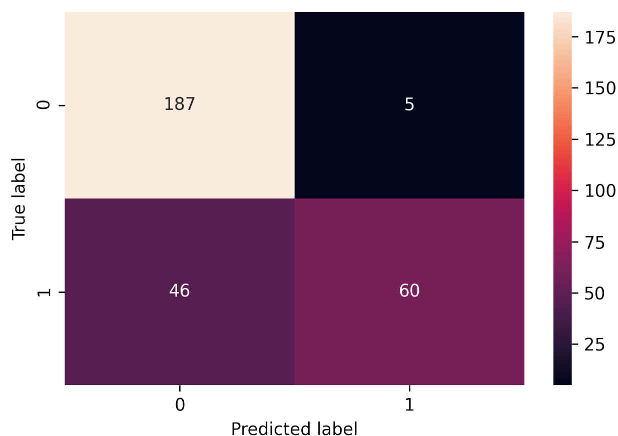


Figura 3.15: Ejemplo de matriz de confusión en un problema de clasificación binaria.

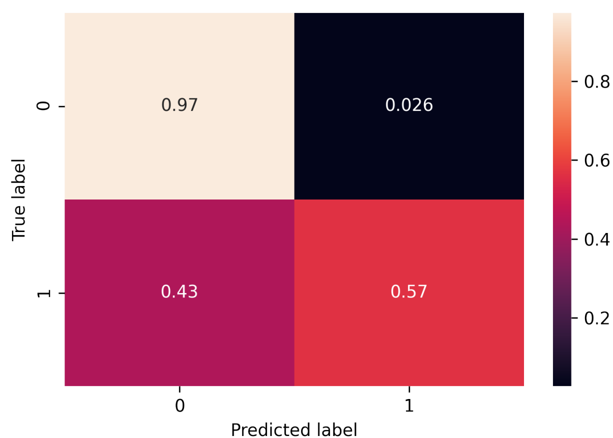


Figura 3.16: Ejemplo de matriz de confusión normalizada en un problema de clasificación binaria.

Para los casos en que tengamos datos desbalanceados, es conveniente normalizar la matriz de confusión. Sea una matriz de confusión C, definimos CN, la matriz de confusión normalizada de C, tal que $CN_{i,j} = \frac{C_{i,j}}{\sum_j C_{i,j}}$.

Para el caso de la figura 3.15, cada celda se dividió por la cantidad de elementos en una fila completa obteniendo la matriz de la figura 3.16. De esta manera, podemos comparar los resultados en dos categorías distintas aun si estas se encuentran desbalanceadas. En este ejemplo observamos que un 57 % de los datos pertenecientes a la clase positiva se clasificaron correctamente como positivos, mientras que un 97 % de las negativas fueron clasificados como negativas.

Gráficos de distribución de predicciones. Por último, la métrica estrella de este trabajo son los gráficos de distribución de predicciones. Como el algoritmo de grounding

heurístico utiliza las probabilidades asignadas a una acción dado su plan relajado, un histograma que refleje la distribución de las probabilidades asociadas a cada par es excelente para determinar si las clases positivas se distribuyen con una probabilidad mayor que las negativas.

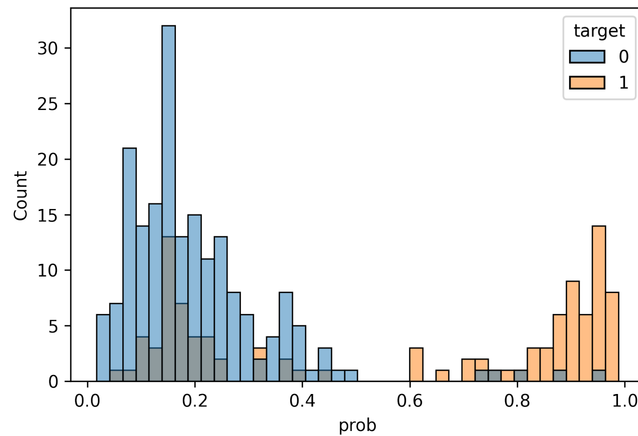


Figura 3.17: Gráfico de distribución de predicciones en un problema de clasificación binaria.

La figura 3.17 muestra el gráfico de distribución de predicciones para el mismo problema de clasificación binaria que utilizamos para explicar las matrices de confusión. Las barras azules representan la clase negativa, y las barras naranjas la clase positiva. Se busca que ambas barras estén lo más separadas posibles. En particular, que las predicciones cuya clase es positiva, tengan asociada una probabilidad cercana a 1, y las predicciones cuya clase es negativa, tengan asociada una probabilidad cercana a 0.

Parte II

Metodología

Capítulo 4

Etapas de preparación y ejecución de modelos

Inferir si una hipótesis brindará resultados positivos o negativos en un proyecto de investigación es una de las tareas más complicadas de conocer con antelación. Aún investigadores experimentados prueban decenas de ideas antes de obtener algún descubrimiento concreto. Construir sistemas de aprendizaje automático usualmente requieren de:

1. Una hipótesis inicial la cual construir el sistema.
2. Una implementación en algún lenguaje de programación.
3. Ejecución de experimentos que permitan concluir si la idea original funciona de acuerdo a lo esperado.

Basado en lo aprendido en esos 3 pasos, se vuelven a plantear nuevas ideas e iterar sobre este proceso. Mientras más rápido se pueda finalizar un ciclo, mayor será el progreso en la investigación. Debido a la larga lista de posibles hipótesis para verificar, se deben obtener resultados prontamente, por lo general, en a lo sumo una semana de haber iniciado el ciclo. Es por eso que una prueba de concepto a partir de una implementación prototipo es más importante que construir un sistema complejo en una etapa temprana de investigación.

No obstante, una vez validadas varias de las hipótesis, empezar nuevas implementaciones a partir de los prototipos puede llegar a ralentizar el proceso de desarrollo, ya que ciertos estándares de código no son tenidos en cuenta. Tales como manejo de excepciones, documentación adecuada, o herramientas de monitoreo y mantenimiento de código. El objetivo de estas pruebas de concepto es la de otorgar experiencia a los investigadores para identificar los requerimientos y resultados iniciales. Esto para luego construir un entorno de desarrollo en esa dirección.

Durante las pruebas de concepto realizadas durante esta tesis se analizó la factibilidad de la premisa inicial, encontrar un modelo de aprendizaje automático a partir de una codificación de datos de planes relajados y acciones etiquetadas que permitan guiar el proceso de grounding. Sin embargo, a medida que el proyecto fue necesitando de

experimentos más complejos, se agregaron funcionalidades hasta construir un sistema de experimentación completo, configurable, adecuado a nuestras necesidades, y que permita otorgar resultados en el corto plazo de iniciado el ciclo de experimentación. Es por eso que en este capítulo se describirán los detalles de implementación de cada uno de los módulos que integran el flujo de ejecución de un experimento, abarcando el dominio de planning utilizado, especificaciones de tareas STRIPS que se pueden obtener a partir del dominio, persistencia de datos, preprocesamiento, entrenamiento, evaluación de modelos, y registro y visualización de resultados. Esto con el fin de describir los experimentos con mucha más claridad en el capítulo 5.

4.1. Dominio de planning: Satellite

4.1.1. Descripción del dominio

Para la construcción del conjunto de datos nos centramos en el dominio de planning *Satellite*, siendo parte del learning track de la competencia internacional de planning (IPC) del año 2011. Este dominio es un modelo del problema de programación de observación satelital. Este implica el uso de uno o más satélites para hacer observaciones, recopilar datos, y descenderlos a una estación terrestre. Los satélites están equipados con diferentes instrumentos, cada uno con características en términos de objetivos de calibración, producción de datos, consumo de energía, y requisitos para el calentamiento y enfriamiento. Los satélites pueden apuntar a diferentes objetivos con el fin de obtener información de tal objeto. Existen restricciones sobre qué objetivos son accesibles para un satélite debido a las capacidades de oclusión y rotación. Los datos que generan los instrumentos de un satélite deben ser enviados a tierra una vez ocurra una ventana de comunicación terrestre. La meta consiste en hallar la cobertura (más eficiente) de las observaciones dadas las capacidades de los satélites.

El cuadro 4.1 resume los esquemas de acción del dominio con su correspondiente tamaño de interfaz (Int), cantidad de precondiciones atómicas (Pre), y cantidad de efectos atómicos (Eff). El Listing 4.1 muestra la especificación del dominio en PDDL.

Esquema	Int	Pre	Eff
take_image	4	6	1
calibrate	3	4	1
turn_to	3	2	2
switch_on	2	2	3
switch_off	2	2	2

Cuadro 4.1: Cantidad de argumentos, precondiciones, y efectos por esquema de acción.

```
(define (domain satellite)
(:requirements :strips :equality :typing)
(:types satellite direction instrument mode)
(:predicates
         (on_board ?i - instrument ?s - satellite)
         (supports ?i - instrument ?m - mode)
         (pointing ?s - satellite ?d - direction)
         (power_avail ?s - satellite)
         (power_on ?i - instrument)
         (calibrated ?i - instrument)
         (have_image ?d - direction ?m - mode)
         (calibration_target ?i - instrument ?d - direction))
(:action turn_to
 :parameters (?s - satellite ?d_new - direction ?d_prev - direction)
 :precondition (and (pointing ?s ?d_prev)
                    (not (= ?d_new ?d_prev)))
 :effect (and  (pointing ?s ?d_new)
               (not (pointing ?s ?d_prev))))
(:action switch_on
 :parameters (?i - instrument ?s - satellite)
 :precondition (and (on_board ?i ?s)
                    (power_avail ?s))
 :effect (and (power_on ?i)
              (not (calibrated ?i))
              (not (power_avail ?s))))
(:action switch_off
 :parameters (?i - instrument ?s - satellite)
 :precondition (and (on_board ?i ?s)
                    (power_on ?i))
 :effect (and (not (power_on ?i))
              (power_avail ?s)))
(:action calibrate
 :parameters (?s - satellite ?i - instrument ?d - direction)
 :precondition (and (on_board ?i ?s)
                    (calibration_target ?i ?d)
                    (pointing ?s ?d)
                    (power_on ?i))
 :effect (calibrated ?i))
(:action take_image
 :parameters (?s - satellite ?d - direction ?i - instrument ?m - mode
)
 :precondition (and (calibrated ?i)
                    (on_board ?i ?s)
                    (supports ?i ?m)
                    (power_on ?i)
                    (pointing ?s ?d)
                    (power_on ?i))
 :effect (have_image ?d ?m)))
```

Listing 4.1: Dominio Satellite en PDDL.

4.2. Manejo de datos

4.2.1. Generación de tareas de planning

Las instancias de problemas a partir del dominio Satellite se obtuvieron automáticamente mediante un generador de problemas configurable de acuerdo a las características que deseamos que tenga. Estas son la cantidad de satélites, la cantidad máxima de instrumentos, el número de modos, el número de objetivos, y el número de observaciones. A mayor sea algunos de estos parámetros, mayor el número de acciones y facts necesarios para resolverlos. Una vez definida una parametrización que represente las características de la tarea a modelar, el generador devuelve su representación STRIPS especificada en PDDL.

Sean $\Pi^{PDDL} = (\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$ la representación STRIPS en PDDL dada por el generador de problemas, y $\Pi = (F, A, I, G)$ su tarea STRIPS asociada. La información que necesitamos recolectar es:

1. Un plan relajado $\vec{a}^+$ para Π^+.
2. Un plan (en lo posible el óptimo) $\vec{a}$ para Π.
3. El conjunto de objetos Σ^O.
4. El conjunto de acciones instanciadas A de Π.

El punto 3 puede obtenerse a partir de los archivos PDDL devueltos por el generador. En cambio, 1, 2, y 4 requieren resolver la tarea por medio del planificador de Fast Downward para ser obtenidos. Dado que Π se obtiene a partir de Fast Downward, el conjunto A de acciones instanciadas contiene únicamente aquellas que son alcanzables relajadamente desde el estado inicial I. A su vez, en la medida de lo posible, se intentará obtener el plan óptimo que resuelve Π. Esto se debe a la noción de relevancia de una acción para resolver una tarea STRIPS. En un plan óptimo, las acciones que lo componen son todas necesarias en la secuencia. Si una acción es eliminada, la secuencia resultante no puede ser un plan de la tarea debido a que se obtendría un plan con menos acciones que el plan óptimo y, por lo tanto, una contradicción (Lo cual no es el caso para un plan genérico).

La información asociada a un problema obtenida por el generador y el planificador se almacenan en directorios que contienen los siguientes archivos:

- *all_operators.bz2*: Archivo comprimido del conjunto de acciones instanciadas.
- *objects*: Conjunto de objetos del problema.
- *problem.pddl*: Especificación en PDDL del problema.
- *relaxed_plan*: Plan relajado que resuelve la tarea relajada.
- *optimal_plan* o *sas_plan*: Plan (óptimo) que resuelve la tarea.

Por último, queda mencionar que tipo de problemas se generaron. Se dividieron en 3 grupos principales de acuerdo a su dificultad. 240 problemas a los cuales pueden resolverse de manera óptima por el planificador, 48 a los cuales solo un plan fue encontrado, y 18 que no pudieron resolverse debido a una falla en el proceso de grounding. Es sobre este último grupo que nos interesa realizar inferencia sobre las acciones que puedan ser relevantes para resolverlo y en los que el modelo de aprendizaje automático debe guiar durante el proceso de grounding. Es por eso que estos 25 problemas fueron elegidos para formar parte de los problemas de test. Mientras que aquellos que sí logran resolverse (de manera óptima o no) forman parte del material de entrenamiento.

Como veremos en la sección 4.2.3, requeriremos de los planes reales para etiquetar estos datos y poder de esta forma realizar un análisis del comportamiento del modelo. Por lo que es necesario encontrarles una solución a los problemas de test. Afortunadamente, Satellite es uno de los dominios trabajados en (Gnad et al., 2019) por lo que limitaremos los problemas de test a aquellos problemas para los que una solución fue hallada en este trabajo.

Otro inconveniente en los problemas de test, es la ausencia de las acciones instanciadas para resolverlo. Aún habiendo logrado calcular un plan de la tarea, la cantidad de instancias alcanzables relajadamente en A es muy grande por la cual no son dadas por el planificador al terminar el proceso de grounding, como sí es el caso de los datos entrenamiento. Para resolver este problema, solo se generó una muestra aleatoria de acciones instanciadas obtenidas a partir de grounding cartesiano, por fuera del planificador.

Por lo tanto, tomaremos provecho de estas soluciones como una manera de verificar que el modelo de aprendizaje logra desempeñarse correctamente antes de probarlo en el planificador en tareas que no puedan instanciarse.

4.2.2. Preservación de datos

Una vez identificado qué problemas son utilizados como entrenamiento y cuáles como test, es necesario preservar estos datos de manera organizada y de fácil acceso. Si bien los datos se encuentran en archivos en una cierta estructura, no pueden ser extraídos de manera programática. Por tal razón se tabularon y almacenaron los datos en una base de datos relacional, para su posterior acceso por medio de una herramienta de consultas.

Para el almacenamiento utilizamos una base de datos relacional de tipo SQL. La figura 4.1 muestra su diagrama de entidad relación.

La tabla *Problem* representa el conjunto de problemas. Algunas de sus propiedades son, el nombre del problema (nombre dado por el generador), y el conjunto al cual pertenece (entrenamiento o test). Para representar en la base de datos tanto los planes relajados, como los planes que resuelven la tarea, se utilizan relaciones *many-to-many* a la tabla de acciones, manteniendo una tabla intermedia que relaciona un problema con las acciones que conforman el plan. Dado que las tablas en bases de datos relacionales se definen a partir de la teoría de conjuntos, la repetición y el orden puede perderse para las acciones que ocurren en un plan. Para resolver este problema se agregó el orden de manera explícita como atributos en sus tablas. Ahora bien, notar que la tabla *Problem* tiene como atributos 2 listas, donde cada atributo representa de manera ordenada (según algún criterio, en nuestro caso, el de los planes relajados) los planes relajados, y los

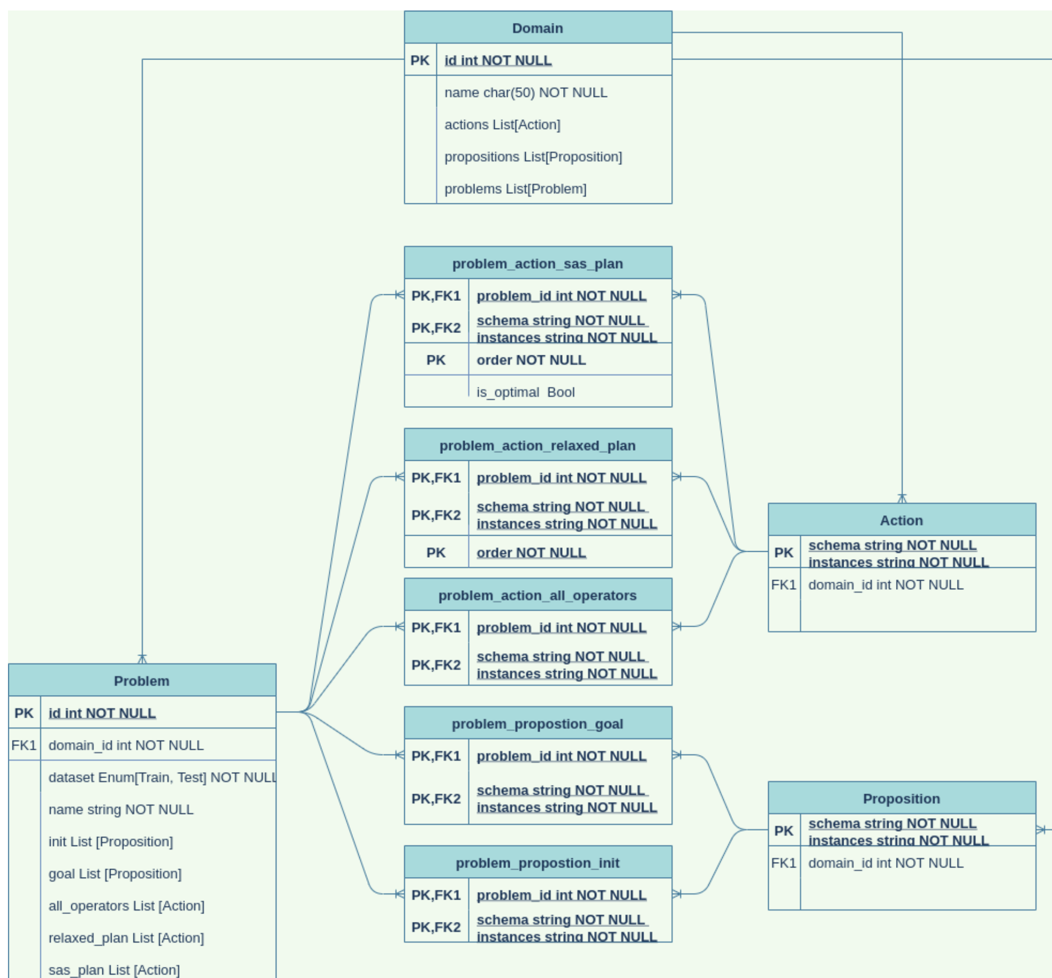


Figura 4.1: Diagrama de entidad relación de la base de datos.

planes reales. Esto permite accederlos de una manera mucho más práctica sin tener que utilizar el atributo del orden de manera explícita al realizar una consulta. Para almacenar el conjunto de acciones instanciables, el estado inicial, y la meta de un problema, simplemente se utilaron otras tablas intermedias pero sin la necesidad de mantener el orden esta vez.

Las tablas *Action* y *Proposition* almacenan todas las acciones y facts instanciables relajadamente que hayan ocurrido en cualquier problema. Para evitar repeticiones se agregó como clave primaria el nombre del esquema junto con el de sus parámetros.

Por último, la tabla *Domain* mantiene registro del dominio en caso de que busquemos utilizar múltiples dominios además de Satellite en nuestros experimentos.

Para realizar consultas a la base de datos empleamos un *object relational mapping* (ORM) que permite gestionar el acceso por medio de un lenguaje de programación. El ORM considera cada tabla como una clase y elementos de la tabla como objetos de tal clase. Por lo tanto, al efectuar una consulta el ORM mapea los elementos partícipes de esa consulta a objetos del lenguaje de programación Eso incluye cualquier otra relación

u objeto que esté involucrado en la consulta de manera directa o indirecta, cono es el caso del plan relajado, o el plan que resuelve la tarea. De esta manera el ORM dispone al programador de todo el poder de la programación orientada a objetos para manipular los datos.

4.2.3. Etiquetado de ejemplos

A partir de la base de datos es necesario generar una *matriz de entrenamiento* que será dada como entrada al modelo de aprendizaje automático. En particular, mencionamos qué problemas se van a usar para entrenar los modelos y cuáles para evaluarlos. Pero aún nos queda explicar cuál es la estructura explicita de los pares $\{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$ de los conjuntos de entrenamiento y test que recibirán como entrada los modelos mencionados en el capítulo 3.

Dado que buscamos aprender un modelo que prediga si una acción es relevante para el plan real de un problema, necesitamos incluir en $\boldsymbol{x}$ información de tal problema y la acción a cuál queramos averiguar su relevancia. Además, debe ser fácil de calcular, tanto para las tareas de entrenamiento como las de test. Como mencionamos en la sección 2.6, para una tarea especificada en PDDL, el planificador instancia el problema y realiza una búsqueda exhaustiva guiada por medio de una función heurística definida usando planes relajados. Si los planes relajados permiten guiar la búsqueda para encontrar un plan de la tarea, entonces también podrían ser usados para guiar el proceso de grounding. Por lo tanto, la estructura de los vectores de entrada incluirán tanto el plan relajado como la acción que queremos estimar.

Por otro lado, en la sección 4.2.1 no sólo obtuvimos el plan relajado de cada tarea generada, sino además el plan que lo resuelve junto a las instancias de las acciones alcanzables relajadamente durante el proceso de grounding. Por lo tanto, para una tarea en particular (y por ende un plan relajado), se pueden identificar dos clases distintas de acciones generadas:

1. Las acciones que pertenecen a la solución (óptima o no).
2. Las acciones que no pertenece a la solución, pero que sí fueron generadas por el proceso de grounding por alcanzabilidad relajada de Fast Downward.

A las acciones de 1 las denominamos *good operators* de la tarea, y las de 2, *bad operators* de la tarea. A estos dos conjuntos los denotaremos con A^{good} y A^{bad} respectivamente.

El algoritmo 3, dado el plan relajado de un problema y los conjuntos A^{good} y A^{bad}, genera los datos etiquetados de una sola tarea obteniendo una matriz como se muestra en el cuadro 4.2.

La primera columna contienen una lista de acciones ordenadas y representan el plan relajado de un problema, la segunda columna es la acción a consultar por su relevancia, y la tercera columna es la etiqueta que indica si pertenece o no al plan de la tarea.

Para generar los ejemplos de los problemas de entrenamiento y test, basta con ejecutar el algoritmo 3 con cada uno de los problemas uniendo las matrices resultantes.

Un inconveniente de esta representación es que la información del plan relajado se multiplica $|A^{good}| + |A^{bad}|$ veces. Esto por cada terna plan relajado, good operators,

Algorithm 3

Input: Plan relajado $\vec{a}^+$, A^{good}, y A^{bad} de una tarea de la sección 4.2.1
Output: Matriz de ejemplos M de tamaño $(|A^{good}| + |A^{bad}|) \times 3$
  $M \leftarrow [\,]$
  for $a \in A^{good}$ **do**
    $M.push_back((\vec{a}^+, a, 1))$
  end for
  for $a \in A^{bad}$ **do**
    $M.push_back((\vec{a}^+, a, 0))$
  end for
  return $M.to_array()$

Plan relajado	Acción	Etiqueta
[switch_on instrument0 satellite0, take_image ...]	calibrate satellite0 instrument0 star4	1
[switch_on instrument0 satellite0, take_image ...]	switch_on instrument0 satellite0	1
[switch_on instrument0 satellite0, take_image ...]	switch_on instrument3 satellite1	1
...	...	...
[switch_on instrument0 satellite0, take_image ...]	turn_to satellite0 planet5 planet5	0
[switch_on instrument0 satellite0, take_image ...]	switch_off instrument0 satellite0	0
[switch_on instrument0 satellite0, take_image ...]	switch_on instrument3 satellite1	0

Cuadro 4.2: Ejemplos etiquetados a partir de un plan relajado y una acción.

y bad operators. Veremos luego en la etapa de preprocesamiento que el tamaño de la matriz además dependerá del largo de los planes relajados, siendo una dificultad a superar durante los experimentos.

Por otro lado, notar que las matrices de los problemas de entrenamiento y de test contienen información de todos los esquemas de acción de los problemas. En la sección de experimentos veremos que nos interesará filtrar las filas por esquemas de acción. Es decir, si tenemos un total de 5 esquemas en Satellite, dividir la matriz del cuadro 4.2 en 5 submatrices donde la acción objetivo corresponda a un solo esquema.

Esto refleja una de las ventajas de haber utilizado una base de datos para preservar la información. En lugar de almacenar los datos de todos los problemas tanto de entrenamiento como de test ya etiquetados. Podemos realizar consultas sobre información específica de los problemas haciendo consultas únicamente de aquello que sea necesario para construir la matriz reduciendo de esta manera el uso en memoria.

A partir de ahora, distinguiremos la noción de problemas de entrenamiento (test) y conjunto de entrenamiento (test). Los problemas de entrenamiento (test) serán aquellos que fueron separados en la sección 4.2.1, mientras que el conjunto de entrenamiento (test) serán las matrices resultantes de la forma del cuadro 4.2 que se obtienen a partir de los problemas.

4.2.4. Preprocesamiento

Como mencionamos en el capítulo 3 los conjuntos de entrenamiento y de test deben ser codificados en algún valor numérico previamente a ser dados como entrada a un modelo de aprendizaje automático. En particular, se trabajó con dos tipos de codificación para representar los planes relajados y las acciones de una tarea, una codificación ad-hoc basándonos en los métodos de tipo one-hot y one-hot ordinal que describimos en la sección 3.5.1, y otra codificación usando word embeddings descriptos en la sección 3.5.2.

Codificación ad-hoc de acciones y planes relajados

En la sección 3.5.1 vimos que una de las codificaciones más sencillas para una oración del lenguaje natural es por medio de una bolsa de palabras. Podemos usar una idea similar para obtener una representación de los planes relajados.

Tomemos una oración dada en el contexto de planning. La siguiente secuencia muestra un plan relajado de largo 4 asociado a una tarea STRIPS, cuyos esquemas de acción son `calibrate`, `switch_on`, `take_image` y `turn_to`. El resto de las expresiones son objetos concretos del dominio. También es importante mencionar que cada objeto de un cierto tipo está enumerado. Por ejemplo el objeto `instrument1` es del tipo `instrument` cuyo índice es 1.

```
[calibrate satellite0 instrument1 groundstation0,
switch_on instrument1 satellite0,
take_image satellite0 planet5 instrument1 image1,
turn_to satellite0 groundstation0 planet5]
```

La primera dificultad durante la codificación es decidir qué es una palabra de la oración. Una posibilidad es definir cada acción como palabra y utilizar un one hot encoding. Pero, por lo general una acción no suele ocurrir más de una vez en un plan de la tarea, con lo que se perdería la información de la frecuencia en que ocurren los objetos en la secuencia. Es clave capturar tanto el tipo de los objetos como su índice. Por último, cada acción debe tener la misma cantidad de componentes en su representación vectorial, independientemente del esquema o el número de parámetros que reciba, y se debe asegurar el orden de la secuencia.

Para resolver estas dificultades se definió la siguiente codificación ad-hoc:

- Cada elemento en la interfaz de una acción junto a su esquema son definidos como palabras. Eso incluye la numeración de los objetos.
- Cada vector que representa a una acción tiene dimensión $2 \times N + 1$ siendo N la longitud de la interfaz más larga de un esquema. A aquellas acciones con una interfaz más chica se les agregará $0's$ hasta completar el largo requerido.
- Los esquemas de acción son enumerados en el rango $1, ..., M$, con M la cantidad de esquemas.

- El tipo de los objetos son enumerados en el rango de $1, ..., K$, con K la cantidad de tipos.
- Si un objeto tiene el índice i se lo incrementa en 1 (para evitar que aquellos objetos que tengan índice 0 se malinterpreten como márgenes).

Ejemplo: Dada la siguiente enumeración de esquemas de acción y tipos:

{`calibrate`: 1, `turn_to`: 2, `switch_on`: 3, `take_image`: 4, `switch_off`: 5}
{`satellite`: 1, `instrument`: 2, `planet`: 3, `groundstation`: 4, `image`: 5, `star`: 6}

Como la longitud de la interfaz más larga es 4, cada acción tendrá asociado un vector de dimensión $2 \times 4 + 1 = 9$ y la codificación resultante sería la siguiente:

`calibrate satellite0 instrument1 groundstation0`	[1 1 1 2 2 4 1 0 0]
`switch_on instrument1 satellite0`	[3 2 2 1 1 0 0 0 0]
`take_image satellite0 planet5 instrument1 image1`	[4 1 1 3 6 2 2 5 2]
`turn_to satellite0 groundstation0 planet5`	[2 1 1 4 1 3 6 0 0]

Luego la codificación del plan relajado es la concatenación de los cuatro vectores.

[1 1 1 2 2 4 1 0 0 3 2 2 1 1 0 0 0 0 4 1 1 3 6 2 2 5 2 2 1 1 4 1 3 6 0 0]

Por lo tanto, la matriz del cuadro 4.2 estaría codificada de la siguiente manera:

Plan relajado	Acción	Etiqueta
[3 2 1 1 1 0 0 0 0 4 ...]	[1 1 1 2 1 6 5 0 0]	1
[3 2 1 1 1 0 0 0 0 4 ...]	[3 2 1 1 1 0 0 0 0]	1
[3 2 1 1 1 0 0 0 0 4 ...]	[3 2 4 1 2 0 0 0 0]	1
...	...	...
[3 2 1 1 1 0 0 0 0 4 ...]	[2 1 1 7 6 7 6 0 0]	0
[3 2 1 1 1 0 0 0 0 4 ...]	[5 2 1 1 1 0 0 0 0]	0
[3 2 1 1 1 0 0 0 0 4 ...]	[3 2 4 1 2 0 0 0 0]	0

Cuadro 4.3: Planes relajados y acciones etiquetadas usando codificación ad-hoc.

Codificación por word embeddings de acciones y planes relajados

Para esta codificación nuevamente surge la pregunta de qué consideramos una palabra en un plan relajado. Para el caso de la codificación ad-hoc una palabra estaba dada por cada una de las partes de una acción incluyendo la numeración de los objetos. Para este caso, podríamos utilizar una interpretación similar para las palabras, ya que cada una de esas componentes son parte importante de una acción. Sin embargo, está vez optamos por considerar como una palabra al esquema de acción y a las instancias de objetos en su interfaz, sin distinguir la numeración como una palabra. El objetivo de esta decisión es intentar que el modelo de word embeddings dependa por sí mismo, es

decir, buscamos que dos objetos del mismo tipo como `satellite0` y `satellite1` sean próximos en la codificación.

Recordemos que los word embeddings tienen la propiedad de que palabras semánticamente similares, son proyectadas cercas en un espacio N-dimensional. Sin embargo, ¿Cuál es el significado de similitud en nuestro corpus de planes relajados? ¿Cuál es el comportamiento que deseamos que aprenda el modelo del lenguaje? Las propiedades que nos interesan capturar son:

1. Las palabras de los esquemas de acción `take_image`, `turn_to`, `calibrate`, `switch_on`, `switch_off`, y la de los objetos que las utilizan, sean proyectadas cerca en el espacio vectorial. Es decir, si `switch_on` usualmente se instancia con los objetos de tipo `instrument` y `satellite`. Entonces las palabras `switch_on`, `instrumentX`, `satelliteY` estén cerca vectorialmente para toda numeración de los objetos `X`, `Y`.

2. En la codificación ad-hoc, distinguíamos la numeración de los objetos como una dimensión más en el espacio vectorial al cual proyectamos los datos. Como contraste, al utilizar word embeddings a nivel de subpalabras, buscamos que todos los objetos numerados de un mismo tipo se proyecten cerca en el espacio. Es decir, si consideramos el tipo `instrument`, entonces para toda numeración `X`, los vectores representación de `instrumentX` están cerca.

Para capturar las propiedades 1 y 2 decidimos utilizar un modelo de lenguaje a nivel de subpalabra (explicado en la sección 3.5.2). En el capítulo 5 uno de los objetivos será verificar que nuestras suposiciones se cumplen para una parametrización concreta del modelo de lenguaje.

La implementación utilizada del modelo de la sección 3.5.2 es la de *FastText* (Bojanowski et al., 2016) desarrollada por *Facebook*. Para el entrenamiento, la implementación requiere como entrada un conjunto de oraciones, en nuestro caso los planes relajados, separados por espacios para distinguir sus palabras. Para distinguir una acción por sobre otra en la secuencia, agregamos además 2 símbolos especiales, `(` y `)` al comienzo y cierre de una acción. Por ejemplo para el plan relajado que mostramos en la codificación ad-hoc:

```
( calibrate satellite0 instrument1 groundstation0 )
( switch_on instrument1 satellite0 )
( take_image satellite0 planet5 instrument image1 )
( turn_to satellite0 groundstation0 plante5 )
```

Esto se repitió para todos los planes relajados de los problemas de entrenamiento, y se dio como entrada al modelo de FastText.

Una vez obtenido el modelo de lenguaje, conseguir la representación de una acción y un plan relajado resulta sencillo. Solo es necesario proyectar cada una de las partes que componen la acción o el plan relajado y calcular el promedio o promedio normalizado (detallado en la sección 3.5.2) de tales vectores. Por ejemplo si tenemos la acción:

```
calibrate satellite0 instrument1 groundstation0
```

Dividimos la oración en las palabras `calibrate`, `satellite0` `instrument1` y `groundstation0`. Entonces el vector de la oración es obtenido como el promedio o promedio normalizado de los vectores de cada palabra.

El caso de los planes relajados es similar. Supongamos que queremos codificar el plan relajado anterior. Nuevamente separamos la oración en palabras (siendo también un símbolo especial una palabra) y promediamos la representación de cada una de ellas.

```
(
calibrate
satellite0
instrument1
groundstation0
)
(
take_image
...
```

Por lo tanto, para un tamaño de 5 dimensiones para el vector de salida del modelo de lenguaje, la matriz del cuadro 4.2 estaría codificada de la siguiente manera:

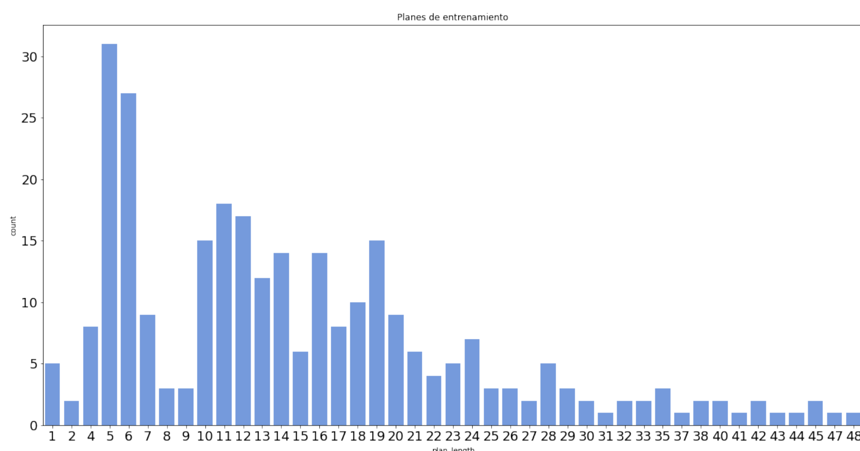
Plan relajado	Acción	Etiqueta
[0.1915, 0.0830, 0.2350, -0.0901, 0.0952]	[0.1674, 0.0807, 0.1190, -0.0124, 0.0586]	1
[0.1915, 0.0830, 0.2350, -0.0901, 0.0952]	[0.2077, 0.0557, 0.0735, -0.0509, 0.0772]	1
[0.1915, 0.0830, -0.2350, -0.0901, 0.0952]	[0.1797, 0.0926, 0.1182, -0.0022, 0.0544]	1
...	...	...
[0.1915, 0.0830, 0.2350, -0.0901, 0.0952]	[0.1694, 0.0787, 0.1193, -0.0063, 0.0626]	0
[0.1915, 0.0830, 0.2350, -0.0901, 0.0952]	[0.1805, 0.0261, 0.0897, -0.0131, 0.0651]	0
[0.1915, 0.0830, 0.2350, -0.0901, 0.0952]	[0.1717, 0.0793, 0.1258, -0.0068, 0.0550]	0

Cuadro 4.4: Planes relajados y acciones etiquetadas usando codificación por word embeddings.

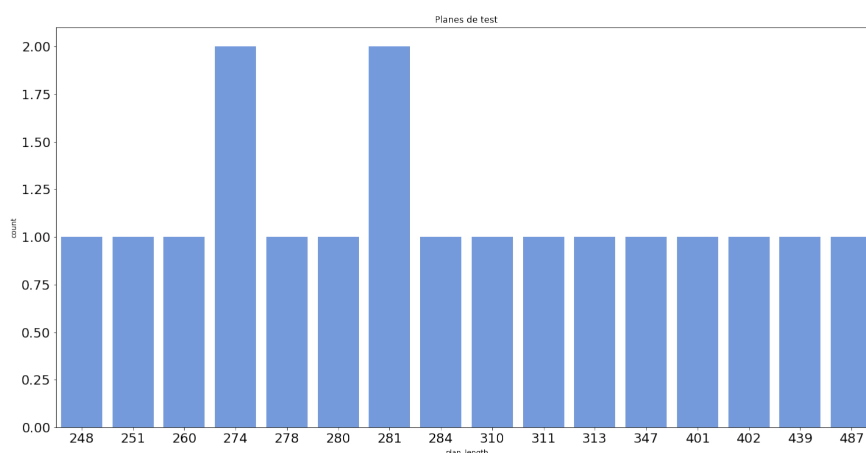
Notar que la dimensión de los vectores del plan relajado y acción son las mismas a diferencia de la codificación ad-hoc. En particular, para todo par de planes relajados en la matriz, sus representaciones tienen la misma dimensión, lo cual no ocurre para la codificación ad-hoc. Esto permite reducir enormemente la dimensionalidad de los vectores que forman parte del conjunto de entrenamiento como así también permite mantener la misma estructura y dimensión con el conjunto de test. Lo cual es requerido por el clasificador que buscamos obtener.

4.2.5. Generación de ventanas de planes relajados

Mencionamos que el largo de los planes relajados de los problemas de entrenamiento y de test varían. Lo cual para la codificación ad-hoc produce vectores de planes relajados con dimensiones distintas. Esto no ocurre en la codificación con word embeddings. Sin embargo, dado que el largo de los planes relajados de test varía entre 250 a 500 acciones en comparación a los de entrenamiento (Figura 4.2), realizar el promedio puede generar



(a) Planes relajados de entrenamiento.



(b) Planes relajados de test.

Figura 4.2: Distribución del largo de planes relajados.

que ciertas acciones de un esquema sean importantes, pero terminen resultando opacadas por otro grupo con mayor frecuencia en la secuencia. Para resolver estos problemas, los planes relajados de cada problema se partieron en ventanas de contexto. En lugar de consultar sobre un plan relajado y una acción, transformaremos los conjuntos de entrenamiento y de test a matrices compuestas por ventanas de plan relajado y acción.

La figura 4.3 muestra un ejemplo para una ventana de tamaño 3 y un paso de tamaño 2. Algo a notar de este ejemplo es que si la última ventana generada no contiene la cantidad necesaria de acciones para generar una ventana completa, se agregan palabras vacías hasta completar el largo restante. Es importante también mencionar que ahora el tamaño de la matriz de entrenamiento depende del largo de los planes relajados. En

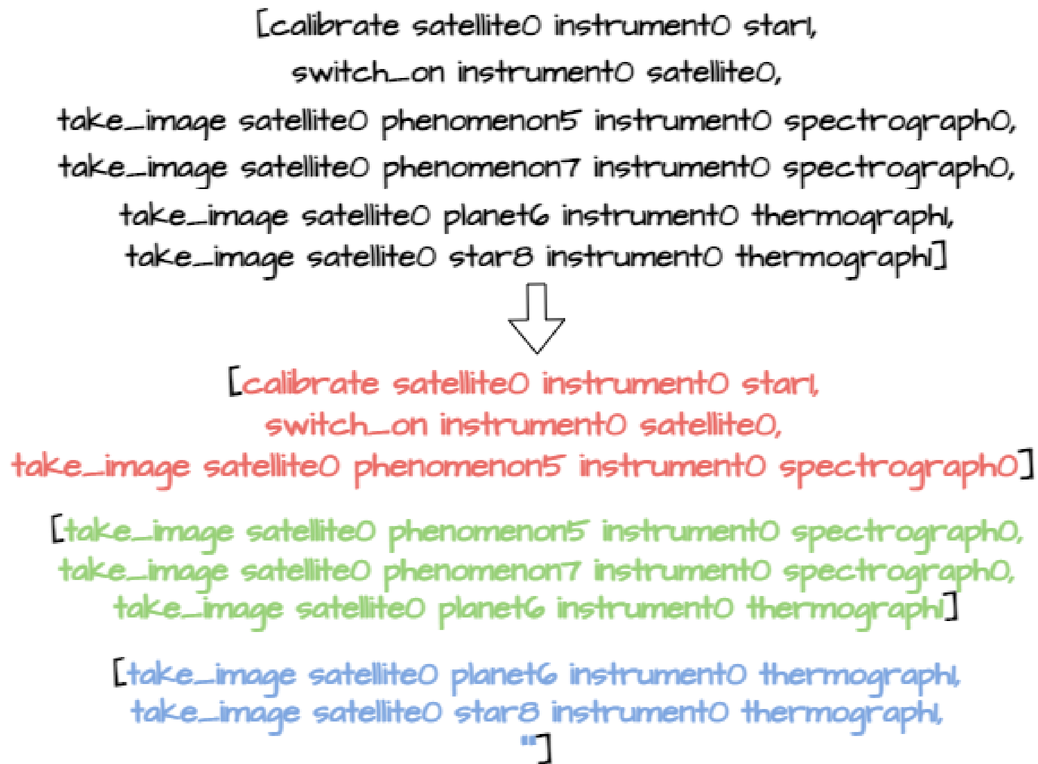


Figura 4.3: Ejemplo de generación de ventanas para un plan relajado de largo 6, ventana de tamaño 3 y paso de 2.

general, para una cantidad de ventanas generadas W, good operators N, y bad operators M. Los ejemplos generados son $W \times N \times M$. Por ejemplo, si para el plan relajado de la figura 4.3, se tienen 300 good operators y 1000 bad operators. La cantidad de filas para ese problema es equivalente a 900000 entradas. Esto es mucho peor para los problemas de test. Es por esto mismo que no se pudieron generar muchos bad operators para estos problemas en la sección 4.2.1.

Una pregunta que queda por resolver es el valor de la etiqueta. Antes de incluir las ventanas, nos interesaba determinar la probabilidad de que una acción sea relevante para el plan real dado su plan relajado. Al utilizar las ventanas de contexto, la pregunta es ligeramente distinta. Para este caso, nos interesa determinar si la probabilidad de una acción es relevante para el plan real dada una ventana del plan relajado. Si una acción es relevante en alguna ventana del plan real entonces también lo es para el plan completo. Por lo tanto, para cada uno de los planes relajados y acciones del conjunto de entrenamiento, propagamos su etiqueta a su representación por partes.

Una vez generada las ventanas de planes relajados, obtenemos sus codificaciones de manera ad-hoc o por word embeddings igual a como describimos en la sección 4.2.4.

4.3. Selección de modelos

4.3.1. Entrenamiento

Una vez realizado el preprocesamiento estamos en condiciones de entrenar los clasificadores que estudiamos en la sección 3.4. Los pares de entrenamiento $\{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$ para entrenar un modelo de aprendizaje automático están compuestos por una representación de una ventana perteneciente al plan relajado de un problema, y la representación de una acción a estimar. Esto conforma el vector de características $\boldsymbol{x}_i$. Las etiquetas reciben una codificación binaria de 1 o 0. Una etiqueta de 1 representa que la acción es relevante para el plan dado esa ventana del plan relajado y una etiqueta de 0 indica lo contrario.

Para una configuración de parámetros, se entrena un clasificador utilizando el conjunto de entrenamiento. A eso se agregan k entrenamientos realizando validación cruzada en k grupos distintos. La única diferencia es que las particiones son sobre los problemas de entrenamiento y no sobre el conjunto de entrenamiento. Agrupar sobre este último puede llevar a que ventanas de un mismo problema pertenezcan a grupos distintos y, por lo tanto, el problema esté incompleto.

4.3.2. Evaluación

Una vez realizado los $k + 1$ entrenamientos del clasificador, deben ser evaluados a partir del conjunto de test para obtener registro de las métricas descriptas en la sección 3.6.

Inicialmente, es necesario generar las etiquetas, ventanas de planes relajados, y codificación de los problemas de test. El conjunto de test resultante es dado como entrada al modelo para obtener sus predicciones. Sin embargo, la evaluación para nuestro problema es ligeramente distinta a la de un problema usual de aprendizaje automático. Nuestro objetivo es estimar la probabilidad de que una acción sea relevante para el plan real dado el plan relajado, pero el modelo realizó sus predicciones a partir de ventanas del plan relajado. Es por eso que la predicción final es la probabilidad máxima de todas las ventanas de un plan relajado asociadas a una acción.

Ejemplo: Supongamos que tenemos la tabla de predicciones a nivel de ventanas de plan relajado dado por la figura 4.4. Para la acción `switch on instrument0 satellite0`, la probabilidad máxima es la de la ventana de color rojo. Por lo tanto, la predicción para esta acción dada el plan relajado completo es la que predijo esa ventana.

4.4. Registro de resultados

4.4.1. Preservación de modelos, métricas, e imágenes

Finalmente, evaluados los modelos, se almacenan los resultados (tanto métricas como imágenes) de cada uno de los k grupos, preservando únicamente los pesos y parámetros del modelo entrenado con todo el conjunto de entrenamiento. Esto último para aprovechar la totalidad de los ejemplos disponibles.

Ventana plan relajado	Acción	Prob
[calibrate satellite0 instrument0 star1, switch_on instrument0 satellite0, . . .]	switch_on instrument0 satellite0	0.654
[take_image satellite0 phenomenon5 instrument0 spectrograph0, . . .]	switch_on instrument0 satellite0	0.407
[take_image satellite0 planet6 instrument0 thermograph1, . . .]	switch_on instrument0 satellite0	0.356

Figura 4.4: Ejemplo de evaluación por ventanas.

4.4.2. Monitoreo y visualización

Para el registro de métricas e imágenes se utilizó la librería *MLFlow*. Esta herramienta permite el almacenamiento de datos y provee herramientas de visualización, registro de tiempos de ejecución de experimentos, comparación de experimentos, etc. Además de ser compatible con varias implementaciones de modelos. Algunas imágenes de las visualizaciones se pueden ver en la figura 4.5.

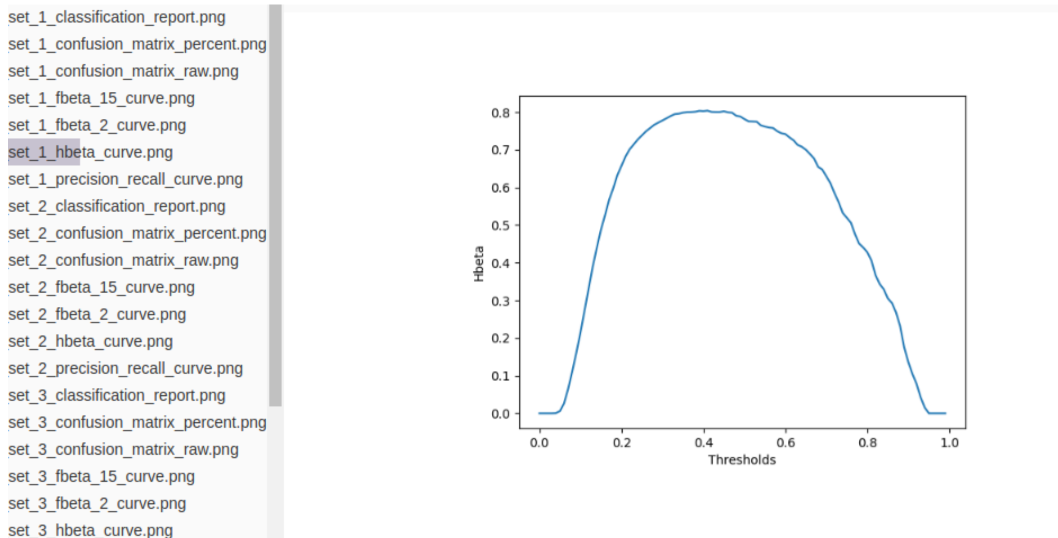
4.4.3. Adaptación de modelos y transformadores

En las secciones 4.2.4, y 4.3 se puso mucho énfasis en la generación de los datos de esta tesis, así como los pasos de exploración, preprocesamiento, entrenamiento y evaluación. Cada una de estas etapas requirieron:

1. Generación de especificaciones de problemas del dominio Satellite.
2. Extracción de planes relajados, acciones instanciadas, y soluciones de los problemas a partir del planificador de Fast Downward.
3. Separación en problemas de entrenamiento y test.
4. Almacenamiento y preservación en una base de datos.
5. Generación de conjuntos de entrenamiento y test.
6. Transformación de los conjuntos de entrenamiento utilizando ventanas de planes relajados.
7. Codificación de ventanas y acciones.
8. Entrenamiento.

		Metrics >			Parameters >		
Start Time	Models	fold_1__accurac	fold_1__f1	fold_1__fbeta_0!	C	act_sch	class_weight
4 days ago	-	-	-	-	-	-	-
4 days ago	sklearn	0.093	0	0	1	switch_off	balanced
4 days ago	sklearn	0.638	0.54	0.46	1	switch_on	balanced
4 days ago	sklearn	0.703	0.441	0.339	1	calibrate	balanced
4 days ago	sklearn	0.655	0.243	0.169	1	take_image	balanced
4 days ago	sklearn	1	0	0	0.1	switch_off	balanced
4 days ago	sklearn	0.619	0.536	0.45	0.1	switch_on	balanced
4 days ago	sklearn	0.701	0.433	0.333	0.1	calibrate	balanced
4 days ago	sklearn	0.662	0.246	0.171	0.1	take_image	balanced
4 days ago	sklearn	0	0	0	0.01	switch_off	balanced
4 days ago	sklearn	0.619	0.527	0.445	0.01	switch_on	balanced
4 days ago	sklearn	0.733	0.448	0.352	0.01	calibrate	balanced
5 days ago	sklearn	0.667	0.248	0.173	0.01	take_image	balanced

(a) Tabla de resultados.



(b) Visualización de gráficos.

Figura 4.5: Visualización de resultados desde MLFlow.

9. Evaluación.

10. Registro de resultados.

Para la implementación de estos pasos se utilizó como lenguaje de programación *Python*. No obstante las librerías y herramientas principales requirieron ser adaptadas a una interfaz común de tal manera que puedan ejecutarse de manera sistemática. Esto se debió a que cada herramienta sigue sus propias convenciones. Por ejemplo, la extracción de datos y lectura de archivos necesitó de librerías que trabajen sobre el sistema de archivos, la persistencia de datos fue por medio de *SQL*, las consultas de bases de datos fueron realizadas por medio del ORM *SqlAlchemy*, las etapas de preprocesamiento

en mayor parte fueron con herramientas de *Numpy* y *Pandas*. La regresión logística y XGBoost proveían una interfaz a través de *Scikit-learn*, pero las redes neuronales fueron implementadas en *Pytorch*. La validación cruzada, evaluación, y obtención de métricas también se hicieron con *Scikit-learn*. Y el registro de datos con *MLflow*. Al momento de usar estas librerías claramente fue necesario implementar una interfaz en común que minimice su acoplamiento, este fue también uno de los desafíos de este trabajo.

Capítulo 5

Experimentos y resultados

En este capítulo presentaremos los experimentos principales realizados a partir de lo desarrollado en el capítulo 4. Entrenaremos 2 grupos de modelos predictivos. Un grupo con una codificación ad-hoc, y otro grupo con una codificación por word embeddings. Los vectores de características tendrán como información una ventana del plan relajado y una acción. Se destacarán aquellos modelos que presenten mejores resultados en validación cruzada y sobre ejemplos del conjunto de test.

Por otro lado, se entrenarán los 3 algoritmos de clasificación vistos en el capítulo 3, *XGBoost*, *regresión logística*, y *perceptrón multicapa*. Los conjuntos de entrenamiento y de test son divididos por esquemas de acción, entrenando un modelo por cada uno. Esto con el fin de capturar la información significativa de un solo esquema por modelo. Por lo tanto, la estructura de nuestros experimentos para ambos grupos es la siguiente:

- Codificación ad-hoc o por word embeddings:
    - Regresión logistica:
        - `take_image`
        - `turn_to`
        - `calibrate`
        - `switch_on`
        - `switch_off`
    - XGBoost:
        - `take_image`
        - `turn_to`
        - `calibrate`
        - `switch_on`
        - `switch_off`
    - Redes neuronales:
        - `take_image`
        - `turn_to`
        - `calibrate`

- `switch_on`
- `switch_off`

Por último, mencionaremos otros experimentos que se trabajaron durante la tesis y la importancia que tuvieron para guiar futuros experimentos.

5.1. Modelos predictivos por word embeddings

5.1.1. Configuración del experimento

En primer lugar, se necesitó asignar la configuración para la construcción de las ventanas y planes relajados. Observamos que por lo general los planes relajados del conjunto de entrenamiento tienen entre 3 y 10 acciones. Nuestro algoritmo debe ser capaz de generalizar para ejemplos de entrenamientos no antes visto, por lo que el preprocesamiento que realicemos sobre estos datos debe ser orientado a reducir las discrepancias entre un ejemplo nuevo y los que fueron utilizados en la etapa de entrenamiento. Como el largo de los planes relajados de los problemas de test es por encima de los cientos de acciones, generamos ventanas que aproximen en tamaño a los planes relajados de los problemas de entrenamiento. Es por ello, que optamos por un paso y tamaño de ventana de 3.

Para el entrenamiento de los clasificadores, se realizó una búsqueda de hiperparámetros cuya métrica a optimizar es H_β con $\beta = 1{,}5$. Dado que en nuestro problema es más importante predecir correctamente todas las acciones relevantes, optamos por priorizar la tasa de verdaderos positivos sobre la tasa de verdaderos negativos. Es decir, buscamos que el clasificador sea penalizado si no logra predecir todas las acciones relevantes para obtener un plan, y si no logra filtrar una cantidad aceptable de acciones no relevantes.

Recordemos que para evaluar los modelos, generamos las predicciones a nivel de ventana de plan relajado y acción. Pero, buscamos predecir si una acción es relevante dada la información del plan relajado completo, por lo que es necesario agrupar las predicciones que cada ventana hizo sobre esa acción. Por ejemplo, supongamos que tenemos M ventanas de un plan relajado $\vec{a}$ de una tarea Π y se busca determinar la probabilidad de una acción a' dado $\vec{a}$. El modelo de aprendizaje automático propuesto realiza M predicciones para a' dado las M ventanas de $\vec{a}$ donde cada estimación es una probabilidad. En lugar de evaluar el modelo con las M predicciones, las agrupamos para obtener solo una. Para este experimento, seleccionamos el máximo de las M probabilidades. Notar que bajo esta decisión, una probabilidad baja significa que para todas las ventanas del plan relajado, el planificador le asignó una baja probabilidad a a', mientras que una probabilidad alta requiere que por lo menos una ventana del plan relajado haya provisto la información necesaria para considerar relevante a a'.

Por último, los algoritmos presentados en el capítulo 3 implementan una solución genérica de los métodos de aprendizaje. Por lo que se requiere especificar como van a estar configurados antes de empezar el proceso de entrenamiento. La siguiente lista de parámetros muestra aquellos que fueron utilizados para cada algoritmo en este experimento:

Nombre del modelo = `Regresión logística`
penalty = `[elasticnet]`
C = `[1, 0.1, 0.01]`
class_weight = `[balanced]`
solver = `[saga]`
max_iter = `[1000, 10000]`
random_state = `[0]`
l1_ratio = `[0, 1]`

Nombre del modelo = `XGBoost`
objective = `[binary:logistic]`
n_estimators = `[500, 1000]`
gamma = `[0.001, 1]`
max_depth = `[10, 20]`
alpha = `[0.001, 1]`
lambda = `[0.001, 1]`
booster = `[gbtree]`
colsample_bytree = `[1]`
subsample = `[0.5]`
eval_metric = `[logloss]`
use_label_encoder = `[false]`
random_state = `[0]`

Nombre del modelo = `PytorchNN`
h_size = `[32, 64]`
n_layers = `[2, 4, 8, 16]`
bn_bool = `[False]`
p = `[0, 0.1]`
epochs = `[20]`
batch_size = `[32]`
balanced = `[True]`
lr = `[0.001]`

Estos parámetros corresponden con la interfaz provista por las librerías de *Scikit-learn* y *Pytorch* que implementan los algoritmos de aprendizaje. Cada parámetro muestra una lista de uno o más valores donde cada elemento representa una posible asignación de ese parámetro. La cantidad de configuraciones posibles de un modelo es igual al producto cartesiano de cada una de las listas de los parámetros de un modelo. Por ejemplo para el caso de la regresión logística, los parámetros *C*, *max_iter*, y *l1_ratio* tienen 3, 2, y 2 asignaciones respectivamente, el resto de parámetros solo tiene una. Por lo tanto, hay un total de $3 \times 2 \times 2 = 12$ configuraciones del modelo.

Los parámetros que seleccionamos siempre buscaban plantear variantes en la función de costo a optimizar, el tipo de optimizador, y métodos para evitar el sobreajuste. En particular, seguimos como guía valores comunes que la comunidad de aprendizaje automático suele utilizar para estos modelos, la experiencia transmitida por el ´´boca

a boca´´, y conocimientos tanto del área de planning como del funcionamiento interno de los modelos utilizados.

En el caso de la regresión logística, los parámetros *penalty*, *l1_ratio*, y *C* permiten realizar variaciones en la regularización de la función de costo, evitando que la norma del vector de pesos del modelo crezca demasiado por sobreajuste. *solver* indica el método de optimización de la función de costo. *max_iter* es el número de iteraciones que requiere el algoritmo para converger. Por último, *class_weight* permite configurar el modelo de tal manera que balancee la cantidad de ejemplares positivos y negativos del material de entrenamiento. Este último es importante para nosotros, ya que nuestro conjunto de entrenamiento está desbalanceado, teniendo una mayor cantidad de bad operators que de good operators por problema.

XGBoost requiere que se le especifiquen la cantidad de árboles a ensamblar con *n_estimators*, la fuerza del parámetro de regularización *gamma*, y la profundidad máxima de los árboles *max_depth*. *objetive* indica la función objetivo a optimizar, en este caso la misma función de costo que una regresión logística. *booster* permite seleccionar variantes del método iterativo de boosting explicado en el capítulo 3.4.8. Por último *colsample_bytree* y *subsample* permiten tomar una muestra con reposición de las filas o columnas en la confección de un árbol en el ensemble con el propósito de reducir el sobreajuste.

La red neuronal necesita del número de neuronas para todas las capas *h_size*, la cantidad de capas ocultas *n_layers*, si los vectores de entrada son normalizados *bn_bool*, y la probabilidad *p* de apagar momentáneamente neuronas de la red de manera aleatoria.

Por cada posible asignación obtenemos aquella que mejor se comporte con el conjunto de test. Recordemos que nuestro objetivo es encontrar modelos que se puedan utilizar para algún esquema de acción. De un total de 5 esquemas de acción en el dominio *Satellite*, el mejor caso sería encontrar 5 modelos que den buenos resultados en esos esquemas.

5.1.2. Resultados: take_image

Empecemos por el esquema de acción `take_image` con el cual obtuvimos mejores resultados. De los 3 modelos regresión logística (LGR), xgboost (XGB), y redes neuronales (NN), LGR resultó el de mejor comportamiento.

Performance en validación cruzada

La figura 5.1 muestra la distribución de las predicciones en validación cruzada para la mejor configuración de parámetros de LGR.

Para cada partición, se muestran dos histogramas independientes, el de la clase positiva, y el de la clase negativa. Cada punto en el gráfico representa la probabilidad de una acción dado su plan relajado (completo). En ambos histogramas se genera una cantidad N de bins equidistantes en el rango $[0; 1]$. Luego, para cada bin b, se muestra la cantidad de acciones que recibieron una probabilidad contenida en b divido el tamaño de la clase a la que pertenecen.

Para los histogramas de la clase positiva, observamos que las acciones relevantes (good operators) se agrupan en los bins con probabilidades entre $r = [0,6; 1]$. Cada bin

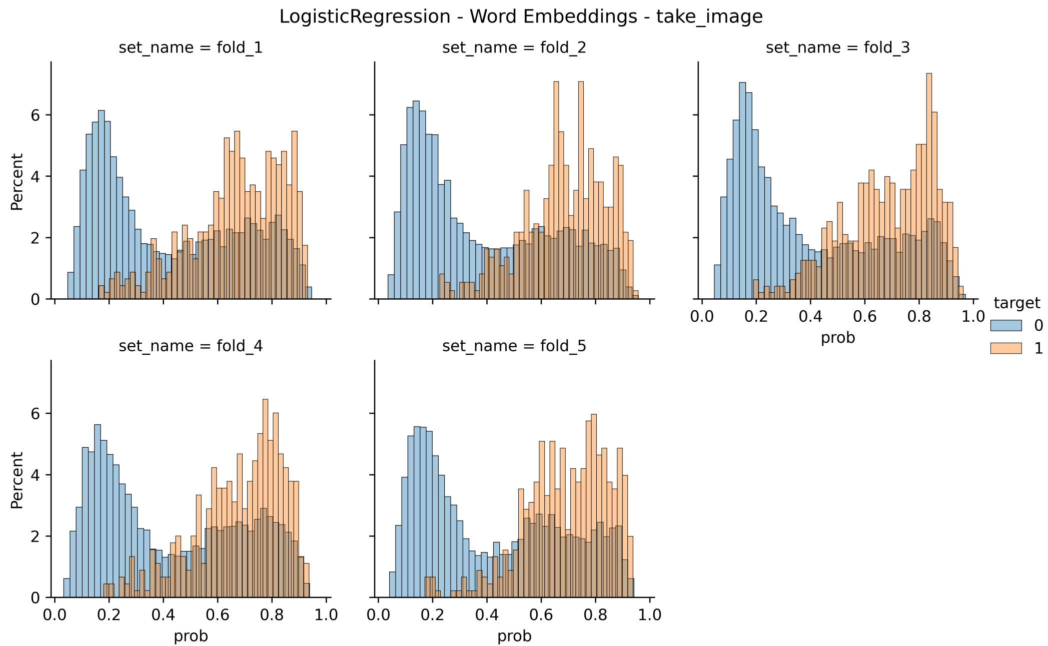


Figura 5.1: Distribución de predicciones en validación cruzada para el modelo de LGR con word embeddings.

contiene entre el 2 % al 7 % de las acciones. Si sumamos los porcentajes de cada uno de los bins en r, tendremos que más de la mitad de las acciones se encuentran en r.

Por otro lado, los histogramas de la clase negativa, muestran que más de la mitad de las acciones no relevantes (bad operators) se agrupan en los bins con probabilidades entre $[0; 0{,}4]$.

Desde el punto de vista de grounding heurístico este es el comportamiento que buscamos. El algoritmo almacenará en una cola de prioridades acciones tanto relevantes como no relevantes con una probabilidad asociada, dándole prioridad a las que tienen una mayor probabilidad. Si el modelo asigna a las acciones relevantes una alta probabilidad entonces el proceso de grounding heurístico les dará prioridad para su instanciación.

Notar que el objetivo de utilizar validación cruzada es variar el conjunto de entrenamiento con el fin de determinar si el modelo de LGR es robusto ante cambios, lo cual podemos decir que indudablemente fue este el caso en las 5 particiones.

Si observamos la matriz de confusión del modelo por cada partición dada en la figura 5.2, podemos ver que en cada uno, más del 90 % de las acciones positivas se predicen como positivas, y más del 63 % de las negativas se predicen como negativas. La función de decisión se obtiene a partir de optimizar $H_{1,5}$. Es decir, se computa $H_{1,5}$ para distintas funciones de decisión con umbral en el intervalo [0, 1] y utilizamos aquel que maximice $H_{1,5}$.

Performance en test

Los resultados anteriores fueron sobre 5 particiones del conjunto de entrenamiento donde una parte efectivamente se utilizaba para entrenar el modelo y otra para su evaluación. Sin embargo, lo que buscamos es que lo aprendido se logre generalizar a problemas no instanciables. Por lo tanto, la evaluación final la realizamos sobre el conjunto de test.

La figura 5.3 muestra nuevamente un histograma de la probabilidad de una acción

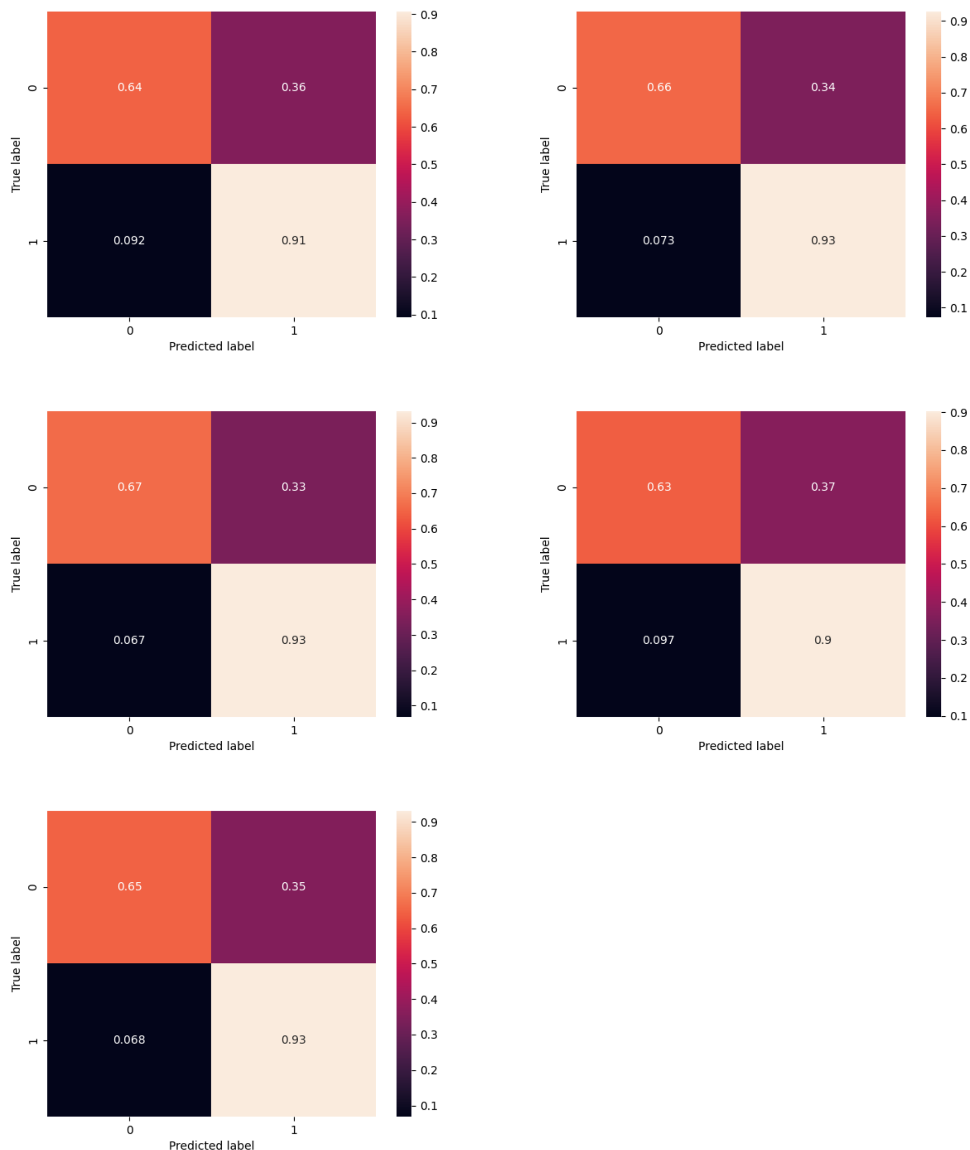


Figura 5.2: Matrices de confusión en validación cruzada para el modelo de LGR con word embeddings.

Clase	Fold	Precisión	Recall	$H_{1,5}$	$F_{1,5}$	Umbral de decisión
Relevante	1	0.14	0.91	0.804	0.338	0.41
No relevante	1	0.99	0.64	0.703	0.7181	0.41
Relevante	2	0.095	0.93	0.822	0.251	0.42
No relevante	2	1	0.66	0.720	0.7371	0.42
Relevante	3	0.14	0.93	0.83	0.340	0.38
No relevante	3	0.99	0.67	0.7295	0.744	0.38
Relevante	4	0.12	0.9	0.797	0.300	0.43
No relevante	4	0.99	0.63	0.6944	0.709	0.43
Relevante	5	0.12	0.93	0.821	0.3022	0.44
No relevante	5	0.99	0.65	0.7147	0.7268	0.44

Cuadro 5.1: Métricas en validación cruzada del modelo LGR con word embeddings.

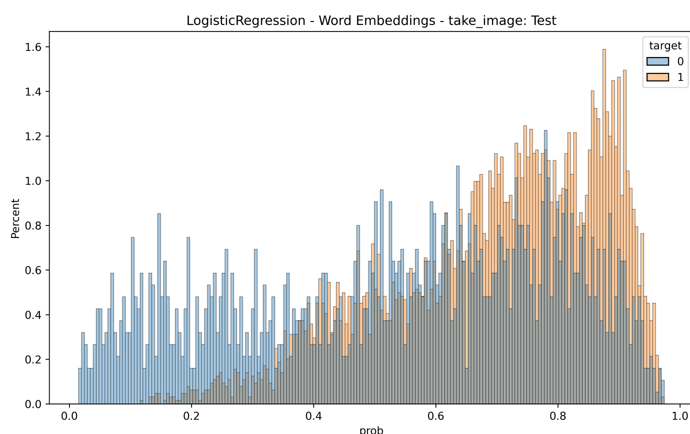
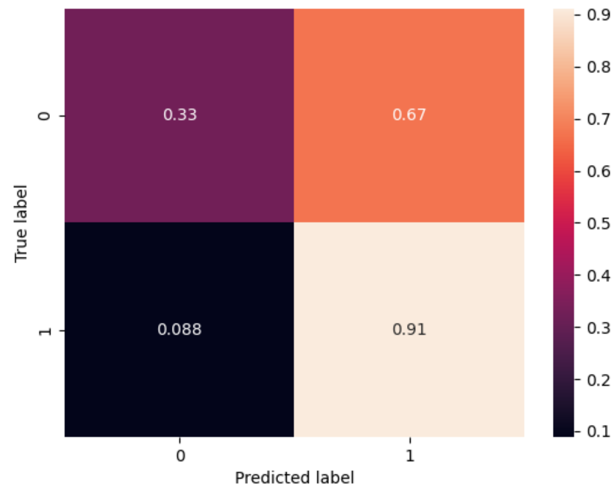
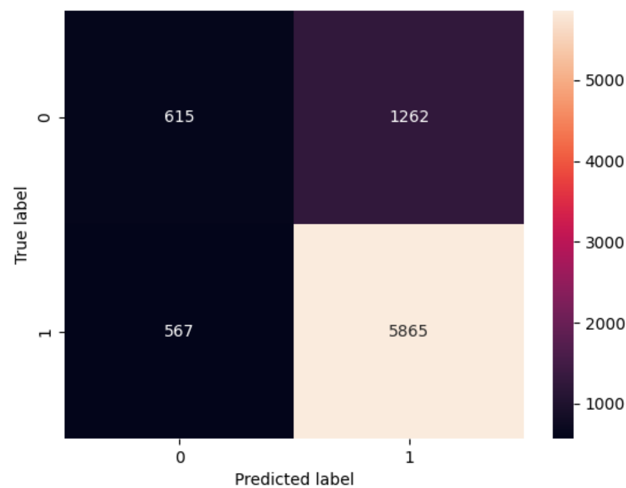


Figura 5.3: Distribución de predicciones sobre el conjunto de test del modelo de LGR con word embeddings.

dado su plan relajado. Para el caso de las acciones relevantes, se sigue manteniendo el comportamiento que vimos en validación cruzada lo cual es un buen indicio, ya que estarán en las primeras posiciones en la cola de prioridades en grounding heurístico. Sin embargo, si observamos el histograma de acciones no relevantes, se tiene que ahora a una gran parte de ellas se le asigna una alta probabilidad. Estas acciones también serían ubicadas al comienzo de la cola de prioridades y, por lo tanto, instanciadas. Si observamos la matriz de confusión en la figura 5.4a, la proporción de acciones relevantes se mantiene en un 91 %, mientras que esta vez solo un 33 % de las negativas se predicen como negativas. Cabe recalcar que esta vez, el umbral de decisión en test es el promedio de los umbrales obtenidos en validación cruzada. El objetivo era aprender un umbral que sea lo más óptimo para predecir nuevos ejemplos, utilizando únicamente información de los datos de entrenamiento. Por lo que se usó el promedio de los umbrales de validación cruzada siendo este un valor que toma en cuenta los umbrales de las 5 particiones. Podríamos haber empleado un umbral de decisión a partir de las etiquetas de test,



(a) Matriz normalizada por clase



(b) Matriz sin normalización

Figura 5.4: Matrices de confusión sobre el conjunto de test del modelo de LGR con word embeddings.

pero tal predicción representaría la performance máxima o ideal que podríamos obtener en test, y no la que realmente estamos obteniendo con información del conjunto de entrenamiento.

Observemos la matriz de confusión de la figura 5.4b del modelo LGR, pero esta vez sin normalizar por porcentajes. Se puede ver que LGR tiende a predecir ligeramente sobre la clase positiva, pero además es la mayoritaria en test para este esquema. Esto sucede debido a como se generan los bad operators en los problemas de test. Recordemos

Clase	Precisión	Recall	$H_{1,5}$	$F_{1,5}$	Umbral de decisión
Relevante	0.823	0.912	0.589	0.8826	0.416
No relevante	0.52	0.33	0.4081	0.3717	0.416

Cuadro 5.2: Métricas sobre el conjunto de test del modelo de LGR con word embeddings.

que estos no pudieron obtenerse a través de Fast Downward, por lo se generó una muestra aleatoria usando grounding cartesiano pero respetando los tipos de las interfaces. Como solo se pudieron generar una cantidad de 5000 bad operators por problema, y los planes en los problemas de test son de mayor tamaño que los de entrenamiento, surgió este desbalance hacia la clase positiva. No obstante, en una situación real, la cantidad de bad operators sería nuevamente superior a la de good operators. Otro aspecto importante sobre la generación de los bad operators es que podrían no ser alcanzables relajadamente y, por ende, tener una distribución distinta a la que tenemos en el conjunto de entrenamiento siendo una de las posibles causas del desempeño de LGR para la clase negativa. Es decir, LGR fue entrenado para predecir, sobre acciones alcanzables relajadamente, pero lo estamos evaluando sobre bad operators que no lo son.

Por último, en el cuadro 5.2 se puede ver que precision y recall sobre la clase positiva es de 0.823, y 0.912, respectivamente. Si calculamos la métrica $F_{1,5}$ priorizando recall, obtendriamos una performance de 0.8826 para LGR. Sin embargo, no es un buen indicio de la correctitud del modelo en la clase negativa. En contraparte, si en lugar de considerar $F_{1,5}$ utilizamos $H_{1,5}$ la performance es de 0.589 debido a que esta métrica penaliza si se tiene un baja en la tasa de verdaderos negativos.

5.2. Modelos predictivos ad-hoc

5.2.1. Configuración del experimento

Para esta sección usaremos la codificación ad-hoc, bajo el mismo conjunto de entrenamientos, construcción de ventanas, y configuraciones de búsqueda de los modelos.

Describiremos el mejor modelo que obtuvimos con la codificación ad-hoc sobre un esquema de acción distinto. En particular, el esquema *calibrate* y el perceptrón multicapa como modelo de aprendizaje.

5.2.2. Resultados

Performance en validación cruzada

La figura 5.5 muestra la distribución de las predicciones en validación cruzada para la mejor configuración de parámetros de NN. Si observamos los histogramas de la clase positiva, tenemos que aproximadamente el 90 % de las acciones etiquetadas como positivas se agrupan en el intervalo $r = [0{,}8; 1]$ mientras que de las negativas solo un 10 % al 20 % se encuentra en r. Además, la mayoría de acciones negativas tienen probabilidades en el rango del $[0; 0{,}4]$ lo cual nuevamente estamos ante un muy buen desempeño en validación cruzada en términos de grounding heurístico.

Clase	Fold	Precisión	Recall	$H_{1,5}$	$F_{1,5}$	Umbral de decisión
Relevante	1	0.31	0.85	0.798	0.5534	0.7
No relevante	1	0.97	0.71	0.74533	0.7738	0.7
Relevante	2	0.26	0.87	0.8	0.50525	0.77
No relevante	2	0.98	0.68	0.7307	0.7507	0.77
Relevante	3	0.3	0.9	0.793	0.55714	0.72
No relevante	3	0.97	0.63	0.6920	0.70616	0.72
Relevante	4	0.31	0.86	0.803	0.5563	0.75
No relevante	4	0.97	0.7	0.7430	0.76557	0.75
Relevante	5	0.26	0.87	0.811	0.50526	0.57
No relevante	5	0.98	0.7	0.7446	0.76747	0.57

Cuadro 5.3: Métricas sobre validación cruzada del modelo NN ad-hoc.

El cuadro 5.3 muestra los resultados de predicción por cada fold. Notar como el umbral de decisión optimizado por $H_{1,5}$ llegó a alcanzar hasta un valor de 0.77 en el segundo fold. Aún para este valor de umbral, los resultados en *precision* no fueron muy altos para la clase positiva. Recordemos que *precision* en clase positiva penaliza los falsos positivos (FPs). Observando el gráfico de distribuciones, el umbral estaría separando adecuadamente las clases relevantes de las irrelevantes por lo cual no se logra observar el porqué de estos resultados en *precision* con este gráfico.

Sin embargo, si en lugar de considerar las distribuciones revisamos las matrices de

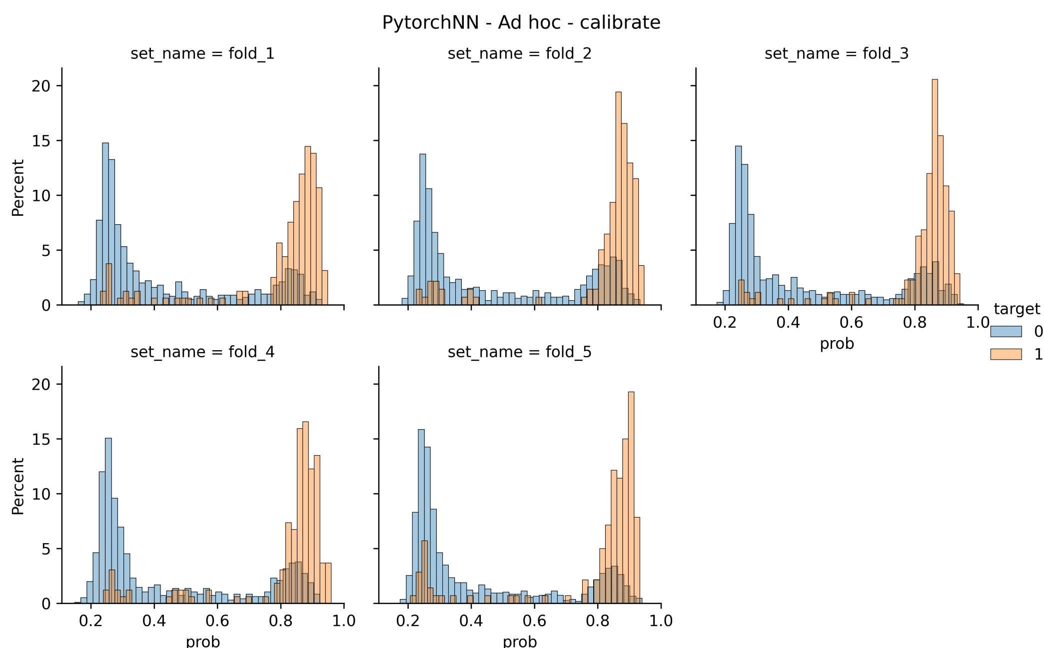


Figura 5.5: Distribución de predicciones en validación cruzada sobre el modelo NN ad-hoc.

confusión sin normalizar (Figura 5.6), observamos que los datos se encuentran desbalanceados para la clase 0. Asemejándose a lo que usualmente ocurre con los problemas de planning donde los good operators suelen ser menor en cantidad a los bad operators. En particular, esto puede ser un problema para *precision* dado que penaliza por la cantidad de FPs. Por ejemplo, si observamos el primer fold de la validación cruzada, se logra predecir correctamente por encima del 70 % de las acciones no relevantes. Sin embargo, esto es relativo a la cantidad de acciones que tengamos de la clase 0. Si tenemos en cuenta la fórmula de *precision*:

$$precision = \frac{TP}{TP + FP}$$

Se logra observar que al depender de los FPs, entonces también depende de la cantidad de desbalance que tenga el conjunto de datos. De hecho, si se incrementa la cantidad de bad operators, pero las proporciones 70-30 % se mantienen, la métrica disminuiría. Sin embargo, en términos de grounding heurístico el modelo seguiría desempeñándose correctamente.

Performance en test

Nuevamente, lo que buscamos es que este modelo logre generalizar a problemas no instanciables. La figura 5.7a muestra como se distribuyen las probabilidades que el modelo asigna a las acciones partícipes del conjunto de test. Se puede observar un comportamiento muy distinto al que ocurrió en validación cruzada. El umbral de predicción promedio obtenido en validación cruzada es de 0.702. Bajo ese umbral, el modelo predice que todas las acciones son irrelevantes obteniendo una performance prácticamente nula. Por lo que a primera impresión parecería que no es un modelo que podamos utilizar en grounding heurístico. De hecho, un resultado como este para un problema clásico de aprendizaje automático llevaría a descartar la solución y probar con otra configuración. No obstante, en el contexto de grounding heurístico, buscamos que las acciones relevantes para encontrar un plan tengan una probabilidad superior a las que no lo son. Sin importar en que parte del intervalo $[0; 1]$ se asignen las probabilidades. Mientras se respete el orden que necesitamos para ubicar las acciones en la cola de prioridades, entonces podemos considerarlo como un buen modelo. Es por eso la razón por la que el umbral de decisión tampoco se mantiene fijo, y varía de acuerdo a H_β. Para este problema el umbral es una manera de cuantificar esta idea más cualitativa que buscamos en grounding heurístico, pero que es determinante en la elección del modelo de aprendizaje.

Por ejemplo, para un umbral de 0.1, podemos ver que el modelo se comporta de manera similar al de *take_image* como muestra la figura 5.7b.

5.3. Otros experimentos

Los experimentos presentados en las secciones 5.1, 5.2, fueron el principal eje de la tesis. No obstante, se intentaron otras vías de análisis que llevaron a muchos más experimentos con el fin de lograr aquel modelo que nos permita guiar el proceso de

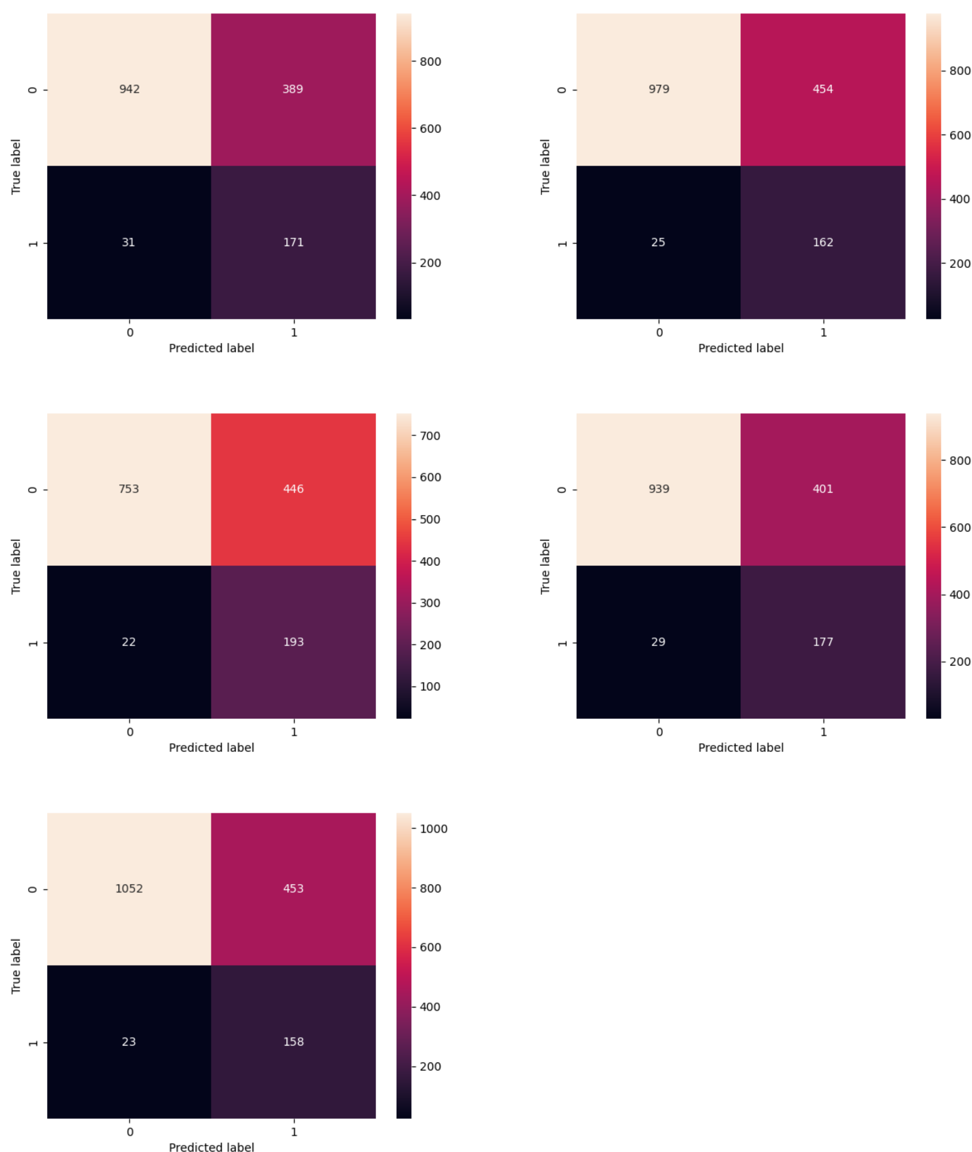
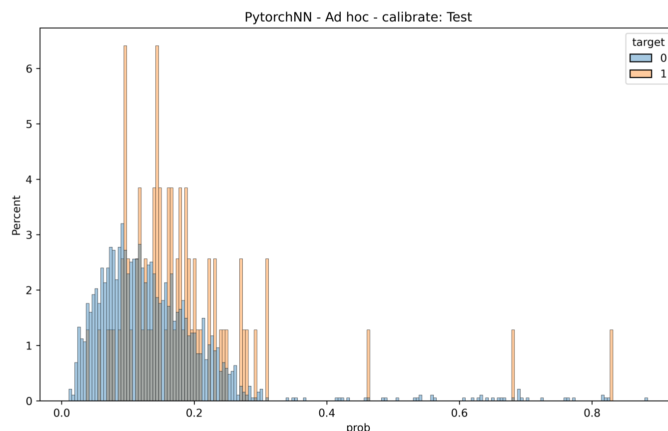


Figura 5.6: Matrices de confusión en validación cruzada para el modelo NN ad-hoc.

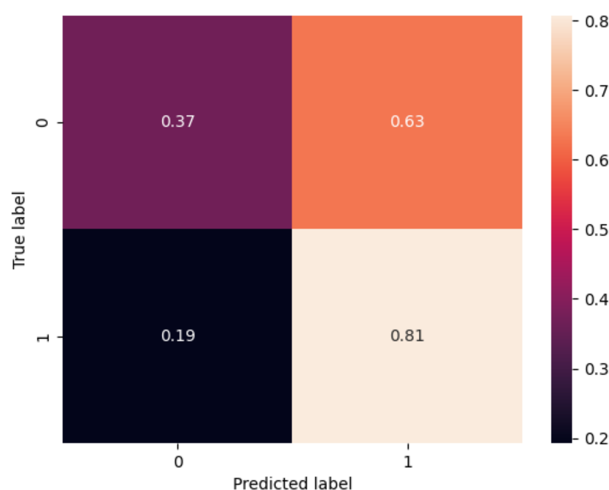
grounding. A pesar de no lograr resultados concretos fueron parte importante del trabajo y otorgaron conocimiento invaluable para involucrarse más en el dominio del problema. En esta sección mencionaremos algunos de ellos.

5.3.1. Modelos end-to-end

Los modelos End-to-end (E2E) refieren a sistemas complejos representados por un único modelo neuronal profundo ensamblando las capas de preprocesamiento como capas intermedias del modelo. En particular, este tipo de pruebas se realizaron en una etapa



(a) Distribución de predicciones



(b) Matriz de confusión

Figura 5.7: Resultados en el conjunto de test para el modelo NN ad-hoc.

temprana de la tesis donde la idea de generar ventanas de planes relajados aún no se había investigado. En particular, se experimentaron con modelos E2E de FastText donde se agrega una capa de predicción *sigmoide* o *softmax* al modelo de embeddings para que sean utilizados en clasificación. No obstante, se optó por descontinuar estos experimentos por no haber mostrado buenos resultados iniciales. Fueron experimentos prototipos donde se buscaba verificar la factibilidad del uso de word embeddings en grounding heurístico. De todas formas, a la luz de lo aprendido en el marco de la tesis, es quizás una tarea interesante para trabajo futuro revisitar esta idea.

5.3.2. El problema de inclusión sobre planes relajados

Otra estrategia que utilizamos durante el análisis en la generación de ventanas de planes relajados, fue plantear el problema de clasificación binaria como uno multiclase. En lugar de sólo 2 clases, manteníamos 4 que representaban la noción de sí una acción pertenecía al plan relajado, y si era parte de los good operators del problema. Es decir, una ventana de plan relajado w y acción a es etiquetada como:

- 0: Si a no es good operator y no pertenece a w.
- 1: Si a no es good operator y pertenece a w.
- 2: Si a es good operator y no pertenece a w.
- 3: Si a es good operator y pertenece a w.

Luego al agrupar por ventanas y acción, consideramos como predicción final la clase 3 si al menos una de las ventanas predijo 3. Caso contrario, si para alguna ventana la predicción fue 2, la predicción del agrupamiento es 2. Y así sucesivamente hasta la clase 0.

De esta manera obteníamos un resultado similar a la agregación por el máximo que realizamos en los experimentos. Es decir, si una acción es relevante dada una ventana del plan relajado, entonces debe serlo dado el plan completo.

Esta idea agrega como información adicional si la acción está en el plan relajado o no, con la hipótesis de que una acción en el plan relajado tiene más chances de estar en el plan original. Sin embargo, esto no pareció resultar en buena performance.

5.3.3. Mayorización

La repetición de ejemplos con etiquetas diferentes puede dificultar el aprendizaje de un modelo, si no son preprocesadas con alguna técnica de mayorización. Es decir, de entre todas las repeticiones del mismo ejemplo, pero con distintas clases, mantener una sola replica preservando la etiqueta que haya ocurrido en mayor cantidad. Esta técnica fue aplicada para la codificación por word embeddings, donde una hipótesis para mejorar la separación de los ejemplos de entrenamiento fue asignar la etiqueta mayoritaria para todo los vectores de dimensión D (plan relajado, acción) que estén cerca en un rango en $\mathbb{R}^D$.

5.3.4. Orden de ventanas como características

Por último, otra implementación que se agregó al sistema, y que es un patrón configurable de las etapas de ejecución, es la posibilidad de mantener el orden en que ocurren las ventanas. Es decir, se agrega un índice en la matriz de características indicando la posición en el plan relajado como se muestra en el cuadro 5.4.

Plan relajado	Orden de ventana	Acción	Etiqueta
[-0.04166, -0.26048, ...]	1	[0.0124, ...]	1
[0.020155, -0.17644, ...]	2	[0.0124, ...]	1
[-0.03441, 0.161433, ...]	3	[0.0124, ...]	1
...	...	...	...

Cuadro 5.4: Ventanas de planes relajados y acciones etiquetadas manteniendo el orden de ventanas.

El objetivo de este experimento fue agregado con el fin de evitar perder la información del orden durante la creación de las ventanas de un plan relajado y se buscaba analizar si es un factor importante para el entrenamiento y posterior evaluación del modelo.

Parte III

Conclusión

Capítulo 6

Conclusiones generales

6.1. Reflexión

La motivación de grounding heurístico surgió tras observar que gran parte de los problemas de planning solo utilizan una pequeña fracción del total de posibles acciones para resolver una tarea. Es común en el área trabajar sobre tareas de con cientos de miles de acciones, pero que resultan en planes de solamente cientos de ellas. Aun así, los planificadores para encontrar una solución de la tarea, deben realizar una búsqueda exhaustiva sobre el total de acciones. Esta búsqueda es guiada a partir de una función heurística definida según información que se puede extraer de la tarea. En el caso de Fast Downward, el framework de planning más utilizado por la comunidad para la resolución de problemas de planning, implementa un algoritmo genérico de búsqueda cuya función heurística se basa en el uso de planes relajados. Esto motivó el uso de planes relajados en grounding heurístico. Si los planes relajados tienen la información para guiar el proceso de búsqueda, entonces también pueden guiar el proceso de grounding.

Por otro lado, el poder predictivo de los algoritmos de aprendizaje automático ha llegado a incluso alcanzar tasas menores al error humano. Por lo que en este trabajo se buscó investigar de que manera se podían aplicar técnicas de aprendizaje automático que permitiesen reconocer la relación entre los planes relajados, acciones, y planes reales. Nuestro objetivo fue encontrar un modelo que permita estimar la probabilidad de que una acción sea relevante para algún plan real, dado su plan relajado.

A partir de esto, surgió una etapa de pruebas de concepto donde verificábamos de que manera disponer el material de entrenamiento y test para los modelos de aprendizaje automático y de que forma codificar los planes relajados y las acciones. A la par de estas pruebas de concepto, se desarrolló la arquitectura del sistema que utilizaríamos para ejecutar futuros experimentos.

El algoritmo final estuvo constituido por una etapa de extracción de los datos a partir del generador de tareas, su preservación de datos, selección de problemas de entrenamiento y test, generación de ejemplos etiquetados, construcción de ventanas de planes relajados, y codificación de características.

En la etapa de codificación fue donde pusimos nuestro mayor enfoque proponiendo dos codificaciones, una de tipo ad-hoc inspirada por una codificación one-hot, y otra a partir de word embeddings del preprocesamiento del lenguaje natural. Dos métodos muy

distintos pero igual de prometedores para entrenar modelos de aprendizaje automático. Con respecto a los clasificadores utilizados, se utilizaron los representantes de tres grandes familia de modelos. En el caso de los modelos lineales, una regresión logística. De los algoritmos de ensemble, XGBoost. Y del mundo del deep learning y las redes neuronales, el perceptrón multicapa. Cada método y clasificador trabajado fueron muy distintos y con características únicas, lo cual resultó excelente para enriquecer nuestro espacio de búsqueda y experimentación.

Con respecto a los resultados finales, presentamos dos modelos que obtuvieron excelentes resultados entrenados sobre acciones provenientes de los esquemas de acción *take_image*, y *calibrate*.

Para el caso de *take_image*, el mejor modelo se obtuvo sorprendentemente con una regresión logística donde, para el conjunto de test, fue capaz de predecir 5865 acciones relevantes para un total de 6432 de ellas. Para el caso de las no relevantes, pudo filtrar 615 de 1877. Esto es equivalente a un 91 % de acciones relevantes que fueron correctamente identificadas y se les asignó una probabilidad mayor a un umbral de 0.416, y un 33 % de acciones no relevantes que recibieron una probabilidad menor a ese umbral. Además, se pudo observar a través de validación cruzada que el modelo se mantuvo robusto ante variantes en los datos de entrenamiento, lo cual lo hace uno de los mejores resultados de este trabajo.

Por otro lado, el perceptrón multicapa dio también buenos resultados con el esquema de acción *calibrate* donde las predicciones en el conjunto de test se distribuyeron de manera distinta a los resultados en validación cruzada, se mantuvo el orden las acciones relevantes, siguen manteniendo una probabilidad mayor que las no relevantes. Lo cual en términos de grounding heurístico es importante. En particular, para un umbral de 0.1 el modelo fue capaz de predecir correctamente un 81 % de las acciones relevantes correctamente y logró filtrar hasta un 33 % de las no relevantes. Lo cual lo convierte en otro candidato para ser utilizado en el algoritmo de grounding heurístico.

6.2. Trabajo a futuro

Una de los experimentos principales como trabajo a futuro es agregar los modelos obtenidos en la implementación de grounding heurístico en el framework de Fast Downward y corroborar que tales modelos de aprendizaje, efectivamente permiten resolver los problemas de test que un principio no son instanciables y, por ende, no tienen solución bajo la implementación vanilla de Fast Downward.

Otro experimento posible es mantener un registro de los tiempos de predicción de cada modelo a nivel de acción o a nivel de batch de acciones. Recordemos que para realizar la predicción de una acción se debe primero realizar su preprocesamiento, construir las ventanas del plan relajado, realizar la predicción de cada ventana, y finalmente agrupar las predicciones para determinar su probabilidad. Este preprocesamiento podría ser un cuello de botella si no es implementado y optimizado de manera adecuada en el algoritmo de grounding heurístico, teniendo en cuenta que además los planificadores deben encontrar la solución del problema en un tiempo limitado. Por lo tanto, un debido análisis del tiempo necesario para llevar a cabo el preprocesamiento y predicción de los modelos es requerido antes de ser incluidos en el algoritmo de Fast Downward.

Para el caso de la codificación por word embeddings, se puede tomar ventaja de la codificación por promedio normalizado desarrollada en la sección 3.5.2. Una posibilidad es utilizar distintos tamaños de ventana entre los datos de entrenamiento y de test. Por ejemplo, mantener un tamaño de ventana de 3 para los planes relajados del conjunto de entrenamiento, y otro de 5 para los planes relajados del conjunto de test. Resolviendo los problemas de dimensiones en las matrices promediando las representaciones de las palabras que componen cada ventana. Como los embeddings mantienen la misma dimensión de salida para todas las palabras, entonces las matrices de entrenamiento y de test mantendrían la misma estructura y dimensión. Este método puede ser una solución a la explosión de ejemplares a causa de la confección de las ventanas de planes relajados, pero puede degradar la información al considerar tamaños de ventanas mucho mayores. Investigar el balance necesario entre calidad del modelo, y degradación de información puede ser un posible lineamiento para experimentaciones futuras.

En virtud de mejorar el sistema de experimentación desarrollado en el capítulo 4 se puede sincronizar la plataforma de *MLFlow* con la base de datos SQL para que se mantengan registro de los modelos. Mantener persistencia organizada de modelos en la nube. Otra opción es generar tareas en *Jenkins*, un sistema de gestión de pipelines de ejecución, que permitan entrenar, evaluar, registrar, y almacenar los resultados de manera independiente, sin necesidad de conocer su implementación. También se puede optimizar algunos procesos de manera que aprovechen el hardware disponible de la computadora con *Cython*.

Bibliografía

Areces, C., Bustos, F., Dominguez, M. and Hoffmann, J. (2014), 'Optimizing planning domains by automatic action schema splitting', *Proceedings of the International Conference on Automated Planning and Scheduling* **24**(1), 11–19.
URL: *https://ojs.aaai.org/index.php/ICAPS/article/view/13622*

Bienkowski, M., desJardins, M. and Desimone, R. (1995), 'Socap: System for operations crisis action planning'.

Bishop, C. M. (2006), *Pattern Recognition and Machine Learning*, Information science and statistics, 1st ed. 2006. corr. 2nd printing edn, Springer.
URL: *libgen.li/file.php?md5=6b552b24cae380bb656f7aaef7f81b46*

Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T. (2016), 'Enriching word vectors with subword information', *CoRR* **abs/1607.04606**.
URL: *http://arxiv.org/abs/1607.04606*

Burke, R. and Robin (2007), Hybrid web recommender systems, Vol. 4321.

Chen, T. and Guestrin, C. (2016), 'Xgboost: A scalable tree boosting system', *CoRR* **abs/1603.02754**.
URL: *http://arxiv.org/abs/1603.02754*

Dang, N. C., García, M. N. M. and de la Prieta, F. (2020), 'Sentiment analysis based on deep learning: A comparative study', *CoRR* **abs/2006.03541**.
URL: *https://arxiv.org/abs/2006.03541*

Fang, W., Li, X., Zhou, P., Yan, J., Jiang, D. and Zhou, T. (2021), 'Deep learning anti-fraud model for internet loan: Where we are going', *IEEE Access* **9**, 9777–.

Farahat, A. K., Ghodsi, A. and Kamel, M. S. (2013), 'Efficient greedy feature selection for unsupervised learning', *Knowledge and Information Systems* **35**, 285–310.
URL: *http://doi.org/10.1007/s10115-012-0538-1*

Firth, J. R. (1957), 'A synopsis of linguistic theory 1930-55.', **1952-59**, 1–32.

Gnad, D., Torralba, A., Domínguez, M., Areces, C. and Bustos, F. (2019), 'Learning how to ground a plan – partial grounding in classical planning', *Proceedings of the AAAI Conference on Artificial Intelligence* **33**(01), 7602–7609.
URL: *https://ojs.aaai.org/index.php/AAAI/article/view/4753*

Goudoulakis, E., El Rhalibi, A. and Merabti, M. (2016), 'A novel multi-agent planning system for digital interactive storytelling', *Comput. Entertain.* **14**(1).
URL: *https://doi.org/10.1145/2735385*

Grigorescu, S. M., Trasnea, B., Cocias, T. T. and Macesanu, G. (2019), 'A survey of deep learning techniques for autonomous driving', *CoRR* **abs/1910.07738**.
URL: *http://arxiv.org/abs/1910.07738*

Heaton, J. (2016), An empirical analysis of feature engineering for predictive modeling, *in* 'SoutheastCon 2016', pp. 1–6.

Helmert, M. (2011), 'The fast downward planning system', *CoRR* **abs/1109.6051**.
URL: *http://arxiv.org/abs/1109.6051*

Hoffmann, J. and Nebel, B. (2001), 'The ff planning system: Fast plan generation through heuristic search', *Journal of Artificial Intelligence Research* **14**, 253–302.
URL: *http://dx.doi.org/10.1613/jair.855*

Iacobacci, I., Pilevar, M. T. and Navigli, R. (2016), Embeddings for word sense disambiguation: An evaluation study.

Kiefer, J. and Wolfowitz, J. (1952), 'Stochastic Estimation of the Maximum of a Regression Function', *The Annals of Mathematical Statistics* **23**(3), 462 – 466.
URL: *https://doi.org/10.1214/aoms/1177729392*

Kingma, D. P. and Ba, J. (2017), 'Adam: A method for stochastic optimization'.

Kühl, N., Goutier, M., Baier, L., Wolff, C. and Martin, D. (2020), 'Human vs. supervised machine learning: Who learns patterns faster?', *CoRR* **abs/2012.03661**.
URL: *https://arxiv.org/abs/2012.03661*

McDermott, D., Ghallab, M., Howe, A. E., Knoblock, C. A., Ram, A., Veloso, M. M., Weld, D. S. and Wilkins, D. E. (1998), Pddl-the planning domain definition language.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J. (2013), Distributed representations of words and phrases and their compositionality, *in* C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani and K. Q. Weinberger, eds, 'Advances in Neural Information Processing Systems', Vol. 26, Curran Associates, Inc.
URL: *https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf*

Muñoz, P., R-Moreno, M. D. and Barrero, D. F. (2016), 'Unified framework for path-planning and task-planning for autonomous robots', *Robotics and Autonomous Systems* **82**, 1–14.
URL: *https://www.sciencedirect.com/science/article/pii/S0921889016302184*

Nau, D., Ghallab, M. and Traverso, P. (2004), *Automated Planning: Theory &; Practice*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Penberthy, J. S. and Weld, D. S. (1992), Ucpop: A sound, complete, partial order planner for adl, Morgan Kaufmann, pp. 103–114.

Pennington, J., Socher, R. and Manning, C. (2014), GloVe: Global vectors for word representation, *in* 'Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)', Association for Computational Linguistics, Doha, Qatar, pp. 1532–1543.
URL: *https://aclanthology.org/D14-1162*

Pitts, W. S. M. W. (1990), 'A logical calculus of the ideas immanent in nervous activity', *Bulletin of Mathematical Biology* **52**, 99–115.

Página web: Test Turing (n.d.).
URL: *https://www.turing.org.uk/scrapbook/test.html*

Rabideau, G., Reder, L., Chien, S. and Booth, A. (2001), Automated planning for interferometer configuration and control, *in* '2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)', Vol. 2, pp. 2/629–2/638 vol.2.

Röger, G., Sievers, S. and Katz, M. (2018), 'Symmetry-based task reduction for relaxed reachability analysis', *Proceedings of the International Conference on Automated Planning and Scheduling* **28**(1), 208–217.
URL: *https://ojs.aaai.org/index.php/ICAPS/article/view/13904*

The illustrated Word2Vec (2019).
URL: *https://jalammar.github.io/illustrated-word2vec/*

Tiwari, R., Saxena, M., Mehendiratta, P., Vatsa, K., Srivastava, S. and Gera, R. (2018), 'Market segmentation using supervised and unsupervised learning techniques for e-commerce applications', *Journal of Intelligent and Fuzzy Systems* **35**, 1–12.

Wan, L., Papageorgiou, G., Seddon, M. and Bernardoni, M. (2019), 'Long-length legal document classification', *CoRR* **abs/1912.06905**.
URL: *http://arxiv.org/abs/1912.06905*

Wei, B., Li, J., Gupta, A., Umair, H., Vovor, A. and Durzynski, N. (2021), 'Offensive language and hate speech detection with deep learning and transfer learning', *CoRR* **abs/2108.03305**.
URL: *https://arxiv.org/abs/2108.03305*

Yeh, T.-H. and Deng, S. (2011), 'Application of machine learning methods to cost estimation of product life cycle', *International Journal of Computer Integrated Manufacturing* **25**, 1–13.

Zhang, Z. (2018), Improved adam optimizer for deep neural networks, *in* '2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)', pp. 1–2.