



## Tesis

Automatización de lectura de Curriculum Vitae  
para selección de personal en el sector IT

Calonge, Federico Matias  
calongefederico@gmail.com

16 de julio de 2022

## Resumen

En la Tesis de Ingeniería en Informática que se presenta se diseña un *sistema de lectura automática de Curriculum Vitae* accesible vía Web. La finalidad del mismo es ayudar al reclutador laboral a elegir a los mejores candidatos para los puestos laborales de IT que tenga disponible. Esta elección se realiza mediante el uso de algoritmos de *machine learning* y basándose, principalmente, en una medición de similitud entre textos: Curriculum Vitae de los candidatos por un lado, y descripciones de los puestos laborales de IT por el otro. El sistema esta desarrollado utilizando el lenguaje de programación Python, permitiendo verificar la teoría desarrollada.

## Abstract

This Computer Engineering Thesis introduces an *automatic Curriculum Vitae reading system* accesible via the Web. The purpose of it is to help the job recruiter to choose the best candidates for the available IT job positions. This choice is made through the use of *machine learning* algorithms and based mainly on a measurement of similarity between texts: Curriculum Vitae of the candidates on the one hand, and IT job descriptions on the other hand. The system is developed using the Python programming language allowing to verify the developed theory.

# Índice

<b>1. Introducción.</b>	<b>7</b>
1.1. Algoritmos y técnicas utilizadas. . . . .	9
1.2. Objetivos del Proyecto. . . . .	10
1.2.1. Objetivo general. . . . .	10
1.2.2. Objetivos específicos. . . . .	10
1.3. Alcance del Proyecto. . . . .	11
1.4. Organización. . . . .	11
<b>2. Reclutamiento y selección laboral.</b>	<b>13</b>
2.1. Introducción. . . . .	13
2.2. Reclutamiento vs selección. . . . .	13
2.3. Evolución de los procesos de reclutamiento y selección laboral. . . . .	15
2.4. Cribado o screening. . . . .	16
2.4.1. Screening manual vs screening automatizado . . . . .	16
2.5. Sistemas de screening: Estado del arte. . . . .	17
2.6. Enfoque del Proyecto. . . . .	19
<b>3. Algoritmos de Machine Learning.</b>	<b>20</b>
3.1. Introducción. . . . .	20
3.2. Machine Learning (ML). . . . .	20
3.2.1. Aprendizaje supervisado y no supervisado. . . . .	21
3.2.1.1. Regresión. . . . .	22
3.2.1.2. Clasificación. . . . .	23
3.2.1.3. Agrupación (clustering). . . . .	24
3.2.1.4. Algoritmos más conocidos. . . . .	25
3.2.2. Aprendizaje transductivo . . . . .	26
3.2.3. Separación de los datos. . . . .	27
3.2.4. ¿Cómo implementar un modelo de ML? . . . . .	28
3.2.4.1. Cross Validation. . . . .	32
3.2.4.2. Los Problemas de ML: Overfitting y Underfitting. . . . .	33
3.3. K-Nearest Neighbor (KNN). . . . .	34
3.3.1. Principal limitación KNN. . . . .	35

3.3.2.	Funcionamiento y ejemplo de KNN. . . . .	35
3.3.3.	Métrica de distancia a emplear. . . . .	36
3.3.4.	Eligiendo el valor de k: overfitting y underfitting. . . . .	38
3.4.	K-means. . . . .	40
3.4.1.	Funcionamiento y ejemplo de K-means. . . . .	40
3.4.2.	Objetivo de k-means y su función de coste. . . . .	42
3.4.3.	Limitaciones K-means. . . . .	44
3.4.3.1.	Obtención del k mediante Elbow Method. . . . .	45
3.4.3.2.	Inicialización de los centroides: k-means++. . . . .	46
3.5.	Redes Neuronales Artificiales (ANN). . . . .	48
3.5.1.	Perceptrón simple. . . . .	48
3.5.2.	Desventaja del Perceptrón simple. . . . .	50
3.5.3.	Perceptrón multicapa (MLP). . . . .	51
3.5.4.	Funciones de activación en MLP. . . . .	53
3.5.4.1.	Softmax en la capa de salida. . . . .	54
3.5.4.2.	Sigmoide en la capa de salida. . . . .	55
3.5.5.	Entrenamiento en una red neuronal. . . . .	56
3.5.5.1.	Hiper-parámetro n. . . . .	58
<b>4.</b>	<b>Natural Language Processing.</b>	<b>59</b>
4.1.	Introducción. . . . .	59
4.2.	Aplicaciones NLP. . . . .	60
4.3.	NLP en la práctica. . . . .	62
4.4.	Preprocesamiento de textos. . . . .	63
4.4.1.	Limpieza y normalización. . . . .	64
4.4.2.	Tokenización. . . . .	65
4.4.3.	Stemming y lemmatization. . . . .	66
4.4.4.	N-grams. . . . .	67
4.5.	Obtención de representaciones vectoriales. . . . .	67
4.5.1.	Bag of Words (BoW). . . . .	68
4.5.2.	TF-IDF. . . . .	70
4.5.3.	Desventajas BoW & TF-IDF. . . . .	72
4.5.4.	Word embeddings. . . . .	73

4.5.4.1.	Embeddings. . . . .	73
4.5.4.2.	Word2vec. . . . .	75
4.5.4.3.	¿Cómo obtener nuestros Word embeddings? . . . . .	77
4.5.4.4.	CBOW vs Skipgram. . . . .	78
4.5.4.5.	One hot encoding. . . . .	80
4.5.4.6.	Obteniendo nuestros Word Embeddings con Skipgram. . .	82
4.5.4.7.	Arquitectura del modelo Skipgram. . . . .	85
4.5.4.8.	Entrenamiento y función de costo con Softmax. . . . .	87
4.5.4.9.	Optimizaciones: Muestreo Negativo. . . . .	91
4.5.4.10.	Entrenamiento y función de costo con Sigmoid. . . . .	94
4.5.4.11.	Desventajas Word2Vec. . . . .	96
4.6.	Obtención de las mediciones de similitud entre textos. . . . .	98
4.6.1.	Maneras de medir la similitud entre textos. . . . .	99
4.6.2.	¿Por qué decidimos utilizar Cosine similarity y WMD? . . . . .	104
4.6.3.	Cosine Similarity. . . . .	105
4.6.4.	Word Mover's Distance (WMD). . . . .	107
<b>5. Implementación.</b>		<b>110</b>
5.1.	Obtención del modelo de clasificación. . . . .	111
5.1.1.	Introducción. . . . .	111
5.1.2.	Esquema. . . . .	111
5.1.3.	Obtención de sets de datos. . . . .	112
5.1.3.1.	Curriculum Vitae. . . . .	112
5.1.3.2.	Descripciones Puestos Laborales. . . . .	112
5.1.4.	Preprocesamiento de textos. . . . .	113
5.1.5.	Cantidad final del set de datos y su uso en las distintas etapas. . . .	114
5.2.	Comparando textos y obteniendo similitudes. . . . .	116
5.2.1.	TF-IDF & Cosine Similarity. . . . .	117
5.2.2.	Word embeddings (Word2vec) & WMD. . . . .	118
5.2.2.1.	Elección de hiper-parámetros Word2vec. . . . .	119
5.2.2.2.	Word Mover's Distance (WMD). . . . .	120
5.3.	Armado del modelo de clasificación KNN. . . . .	121
5.4.	Clasificación de nuevas muestras y resultados obtenidos. . . . .	122

5.5.	Integración al Sistema Web. . . . .	123
5.5.1.	Base de datos. . . . .	124
5.5.2.	Secciones del sistema . . . . .	126
5.5.3.	Manejo de los datos. . . . .	129
5.5.3.1.	Modelado. . . . .	130
5.5.3.2.	Filtrado. . . . .	131
5.5.3.3.	Visualización. . . . .	132
5.6.	Pipeline Flow final del Sistema. . . . .	133
5.7.	Caso de Uso. . . . .	134
5.8.	Limitaciones del sistema. . . . .	135
<b>6.</b>	<b>Conclusiones.</b>	<b>136</b>
<b>7.</b>	<b>Anexos.</b>	<b>137</b>
7.1.	Ejemplo de funcionamiento KNN. . . . .	137
7.2.	Ejemplo de funcionamiento K-means. . . . .	138
7.3.	Posición inicial de los centroides en K-means. . . . .	141
7.4.	Funciones de activación. . . . .	143
7.5.	Ejemplo de obtención de Word Embeddings mediante skipgram y softmax.	144
7.6.	Palabras repetidas en nuestro Corpus y palabras polisémicas. . . . .	152

# 1 Introducción.

Los procesos de *reclutamiento y selección laboral* se han vuelto cruciales para el manejo de recursos humanos en el mundo moderno. Con las transformaciones digitales de las empresas y del mercado laboral en general, identificar los perfiles más acordes a las necesidades de la empresa se convirtió en uno de los retos más ambiciosos de Recursos Humanos, en especial cuando hablamos del *Sector IT*, donde año tras año se van generando nuevos puestos de trabajo y estos mismos van creciendo en demanda. Este crecimiento de la demanda en distintos puestos del Sector IT lo podemos evidenciar, a modo de resumen, en la figura 1.1.

En estos últimos años se implementaron una gran cantidad de herramientas de Software que utilizan algoritmos inteligentes y que permiten automatizar y gestionar información de los candidatos de una manera mucho más intuitiva[12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. La *automatización* se transformó en un factor clave e indispensable para que el reclutador pueda conseguir a los candidatos más relevantes para los puestos que ofrece la empresa en un tiempo muy corto. Se espera que para el año 2025 muchas tareas realizadas actualmente por humanos, sean automatizadas por máquinas: esto lo podemos evidenciar en la figura 1.2.

↗ Increasing demand	↘ Decreasing demand
1 Data Analysts and Scientists	1 Data Entry Clerks
2 AI and Machine Learning Specialists	2 Administrative and Executive Secretaries
3 Big Data Specialists	3 Accounting, Bookkeeping and Payroll Clerks
4 Digital Marketing and Strategy Specialists	4 Accountants and Auditors
5 Process Automation Specialists	5 Assembly and Factory Workers
6 Business Development Professionals	6 Business Services and Administration Managers
7 Digital Transformation Specialists	7 Client Information and Customer Service Workers
8 Information Security Analysts	8 General and Operations Managers
9 Software and Applications Developers	9 Mechanics and Machinery Repairers
10 Internet of Things Specialists	10 Material-Recording and Stock-Keeping Clerks
11 Project Managers	11 Financial Analysts
12 Business Services and Administration Managers	12 Postal Service Clerks
13 Database and Network Professionals	13 Sales Rep., Wholesale and Manuf., Tech. and Sci. Products
14 Robotics Engineers	14 Relationship Managers
15 Strategic Advisors	15 Bank Tellers and Related Clerks
16 Management and Organization Analysts	16 Door-To-Door Sales, News and Street Vendors
17 FinTech Engineers	17 Electronics and Telecoms Installers and Repairers
18 Mechanics and Machinery Repairers	18 Human Resources Specialists
19 Organizational Development Specialists	19 Training and Development Specialists
20 Risk Management Specialists	20 Construction Laborers

Figura 1.1: Top 20 demanda de roles laborales en aumento y disminución para el año 2020, por participación de las empresas encuestadas por el *Foro Económico Mundial*[2].

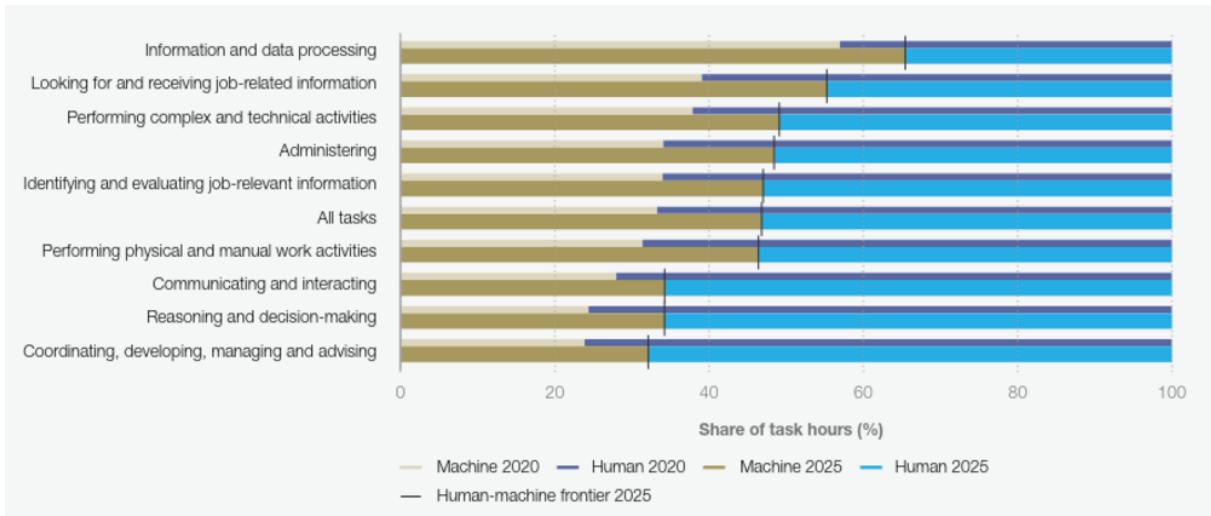


Figura 1.2: Porcentaje de tareas realizadas por humanos frente a máquinas, 2020 y 2025 (previsto), por participación de las empresas encuestadas por el *Foro Económico Mundial*[2].

El tema de este Proyecto de Tesis será desarrollar un *sistema de lectura automática de Curriculum Vitae* accesible vía Web. La finalidad del mismo es ayudar al reclutador laboral a elegir a los mejores candidatos para los puestos laborales de IT que tenga disponible. Esta elección se realiza mediante el uso de algoritmos de machine learning y basándose, principalmente, en una medición de similitud entre textos: Curriculum Vitae de los candidatos por un lado, y descripciones de los puestos laborales de IT por el otro.

La medición de similitudes entre documentos de texto (text similarity measurement) es uno de los problemas más cruciales del *Procesamiento del Lenguaje Natural (NLP)*. Encontrar similitudes entre documentos se utiliza en varios dominios de NLP, tales como en sistemas de recomendación, de information retrieval (IR), de análisis de sentimientos, etc.[1]

Para que las máquinas puedan describir esta similitud entre documentos, se necesita definir una forma de medir matemáticamente la similitud, la cual debe ser comparable para que la máquina pueda identificar qué documentos son más (o menos) similares. Previamente a esto necesitamos representar el texto de los documentos en una forma cuantificable (que suele ser en forma vectorial), de modo que podamos realizar los posteriores cálculos de similitud sobre él. Una distancia pequeña entre los vectores significa un alto nivel de similitud, mientras que grandes distancias significan un bajo nivel de similitud[1].

Por lo tanto, resumidamente los pasos necesarios para poder medir similitudes entre documentos son[1]:

1. Convertir cada uno de los documentos de texto en un objeto matemático (vector).
2. Definir y emplear una métrica de distancia que será utilizada como nuestra medida de similitud entre los textos.

En el sistema desarrollado, una vez obtenidas estas mediciones de similitud entre los Curriculum Vitae de los candidatos y las descripciones de los puestos laborales de IT mediante distintos algoritmos y técnicas de machine learning, estos valores se utilizarán para alimentar y generar *un modelo de clasificación*, el cual nos servirá para lograr, en base a los valores de similitud de nuevos candidatos, clasificar qué tan similares son dichos candidatos con respecto a la descripción de un puesto de IT: similitud escasa, similitud media, similitud alta, similitud muy alta.

## 1.1 Algoritmos y técnicas utilizadas.

A continuación nombraremos resumidamente los algoritmos y técnicas de machine learning utilizadas para el desarrollo del *sistema de lectura automática de Curriculum Vitae*.

Como se mencionó en *Introducción*, el primer paso para lograr nuestra medición de similitudes entre documentos consiste en convertir los documentos en vectores: para esto se utilizaron las técnicas de vectorización **TF-IDF** y **Word Embeddings**.

Para el segundo paso, definir y emplear métricas de distancia, se decidió emplear una combinación de las técnicas **Cosine Similarity** y **Word Mover's Distance (WMD)**. En la sección *¿Por qué decidimos utilizar Cosine similarity y WMD?* se detallan las razones por las cuales se decidieron elegir estos métodos; las cuales están justificadas en base al análisis realizado en la sección *Maneras de medir la similitud entre textos*, en el cual se detallan las numerosas técnicas existentes para obtener similitudes entre textos.

Como tercer y último paso, una vez obtenidas estas mediciones de similitud entre los Curriculum Vitae de los candidatos y las descripciones de los puestos laborales de IT, estos valores se utilizaron para alimentar un **algoritmo de clustering K-means** que a su vez, con sus datos de salida (4 clusters), alimentan a un **modelo de clasificación K-Nearest Neighbor (KNN)**.

Finalmente, con este modelo KNN logramos, en base a los valores de similitud de nuevos candidatos, clasificar qué tan similares son dichos candidatos con respecto a la descripción de un puesto de IT.

Estos algoritmos, técnicas y modelos utilizados serán detallados en las secciones posteriores, tanto teóricamente como en su implementación.

## **1.2 Objetivos del Proyecto.**

### **1.2.1 Objetivo general.**

El objetivo de este Proyecto de Tesis es lograr un desarrollo, tanto teórico como práctico, de un *sistema de lectura automática de Curriculum Vitae* accesible vía Web. La finalidad del mismo es ayudar al reclutador laboral a elegir a los mejores candidatos para los puestos laborales de IT que tenga disponible. Esta elección se realiza mediante el uso de algoritmos de machine learning y basándose, principalmente, en una medición de similitud entre textos:

- los Curriculum Vitae de los candidatos por un lado,
- descripciones de los puestos laborales de IT por el otro.

### **1.2.2 Objetivos específicos.**

Los objetivos específicos de este Proyecto de Tesis son:

- Describir el estado del arte actual de los Sistemas de lectura y análisis de Curriculum Vitae en las fases de reclutamiento y selección laboral.
- Implementar un Sistema de lectura automática de Curriculum Vitae basado en la comparación y medición de similitudes entre textos, para finalmente obtener una visualización de los mejores candidatos para un puesto laboral de IT determinado.
- Aprender los conceptos y técnicas principales utilizadas dentro del procesamiento de lenguaje natural (NLP) aplicando técnicas de preprocesamiento y limpieza de textos.
- Implementar diferentes técnicas para medir similitudes entre los textos (Cosine Similarity y Word Mover's Distance -WMD-) y diferentes técnicas de vectorización (TF-IDF y Word Embeddings), analizando su funcionamiento tanto teórica como matemáticamente, ventajas y desventajas.
- Conocer, implementar e integrar el algoritmo de clustering K-means junto al algoritmo de clasificación KNN para obtener un modelo de clasificación de candidatos en base a las medidas de similitud entre los textos.
- Evaluar los Frameworks disponibles para tener una UI<sup>1</sup> accesible vía web e integrar el mismo al Sistema.
- Almacenar datos de candidatos, reclutadores y puestos laborales en una base de datos.

---

<sup>1</sup>La interfaz de usuario o user interface (UI) de una página web refiere a todo aquello tangible con lo que los usuarios interactúan de forma directa en la misma.

## **1.3 Alcance del Proyecto.**

El alcance de esta Tesis de Grado de Ingeniería incluye el desarrollo de conceptos de análisis de datos y machine learning, procesamiento de lenguaje natural, técnicas de preprocesamiento y limpieza de los datos, técnicas de vectorización y técnicas para medir la similitud entre textos, algoritmos de clasificación y clustering, integración con frameworks, visualización de datos, y gestión de Base de Datos, de acuerdo a lo enunciado en los objetivos específicos.

## **1.4 Organización.**

Este Proyecto de Tesis fue organizado para trabajarla en tres secciones:

### **1. Análisis e investigación inicial.**

Esta sección abarca principalmente la parte teórica del trabajo, haciendo hincapié en el análisis e investigación de:

- El estado del arte (actual y pasado) de los sistemas de lectura y análisis de Curriculum Vitae.
- Técnicas usadas para el procesamiento del lenguaje natural (NLP).
- Técnicas para medir similitudes entre textos: Cosine Similarity y WMD.
- Técnicas de vectorización: TF-IDF y Word Embeddings.
- Algoritmos de machine learning para tareas de clasificación (KNN) y clustering (K-means).

Esta primera sección abarca los capítulos *Reclutamiento y selección laboral*, *Algoritmos de Machine Learning* y *Natural Language Processing* de este Informe de Tesis.

### **2. Implementación de distintas técnicas y algoritmos para la obtención del modelo de clasificación KNN.**

Esta sección hace referencia a la aplicación práctica dentro del marco teórico desarrollado en la primera sección, mediante la realización de una serie de análisis en documentos de Jupyter Notebook<sup>2</sup> utilizando Python <sup>3</sup>, para la obtención final de nuestro modelo de clasificación KNN capaz de clasificar, en base a los valores de similitud de nuevos candidatos, qué tan similares son dichos candidatos con respecto a la descripción de un puesto de IT: similitud escasa, similitud media, similitud alta, similitud muy alta.

---

<sup>2</sup>Aplicación cliente-servidor que permite crear documentos web en formato JSON que siguen un esquema versionado y una lista ordenada de celdas de entrada y de salida. Estas celdas albergan, entre otras cosas, código, texto (en formato Markdown), fórmulas y ecuaciones matemáticas. Estos documentos que se generan funcionan en cualquier navegador estándar.

<sup>3</sup>Lenguaje de programación interpretado y multiplataforma de código abierto, popularizado en los últimos años por su facilidad para trabajar con inteligencia artificial, big data, machine learning y data science, entre muchos otros campos en auge.

Los items que abarca esta sección son:

- Obtención de sets de datos: currículums vitae y descripciones laborales.
- Preprocesamiento de los textos.
- Comparación entre textos y obtención de similitudes entre los mismos mediante el uso de las técnicas para medir distancias y obtener dichas similitudes (WMD y Cosine Similarity) y las técnicas de vectorización (TF-IDF y Word Embeddings).
- Obtención del modelo de clasificación KNN utilizando como datos de entrada los clusters devueltos por el algoritmo K-means obtenidos en base a las mediciones de similitud previamente realizadas.
- Análisis y primeras visualizaciones de los resultados.

Esta sección abarca el capítulo *Implementación* (desde *Obtención del modelo de clasificación* hasta *Clasificación de nuevas muestras y resultados obtenidos*) de este Informe de Tesis.

### 3. Integración al sistema web.

Esta última sección hace referencia a la reutilización de las funciones que contienen la lógica de los distintos algoritmos utilizados junto con el modelo de clasificación KNN obtenidos previamente en la sección 2, para integrar todo este conjunto en el sistema web que, a su vez, está integrado a una base de datos relacional. De esta manera, el sistema cuenta con una interfaz gráfica permitiendo interactuar entre candidatos y reclutadores y, principalmente, permitiendo que el reclutador sea capaz de obtener un listado con los N candidatos más similares a un puesto determinado, y ordenados de mayor a menor de acuerdo a esta *similitud*. Dicha *similitud* representa el resultado obtenido de la clasificación por nuestro modelo KNN.

Los items principales de esta sección son:

- Definición de los usuarios que accederán al sistema.
- Definición de los datos que se almacenarán.
- Integración de frameworks y bases de datos.
- Modelado, filtrado y visualización de los datos.
- Reutilización e integración al sistema de los algoritmos y del modelo KNN utilizados en la fase previa.
- Evaluación del funcionamiento de todo el Sistema integrado.

Esta última sección abarca el capítulo *Implementación* (desde *Integración al Sistema Web* hasta el final del capítulo) de este Informe de Tesis.

## 2 Reclutamiento y selección laboral.

En este capítulo se va a realizar una introducción a los procesos de reclutamiento y selección laboral, sus diferencias y diferentes tareas involucradas. Por último, se llevará a cabo un análisis del Estado de Arte actual de los sistemas de cribado (o más conocidos como sistemas de *screening*) y se detallará el enfoque utilizado para este Proyecto.

### 2.1 Introducción.

Los procesos de *reclutamiento y selección laboral* se han vuelto cruciales para el manejo de recursos humanos en el mundo moderno. Con las transformaciones digitales de las empresas y del mercado laboral en general, identificar los perfiles más acordes a las necesidades de la empresa se convirtió en uno de los retos más ambiciosos de Recursos Humanos, en especial cuando hablamos del *Sector IT*, donde año tras año se van generando nuevos puestos de trabajo y estos mismos van creciendo en demanda.

Los *reclutadores*, dentro de un departamento de recursos humanos, son los encargados de llevar a cabo los procesos de *reclutamiento y selección* laboral. Uno de sus objetivos principales es buscar talento humano para cubrir los puestos de trabajo vacantes que tenga la empresa.

### 2.2 Reclutamiento vs selección.

El **reclutamiento** es el proceso de atracción, búsqueda, recolección e identificación de candidatos que encajan con la oferta de trabajo y, en definitiva, con la empresa. Como mencionan Anwar y Abdullah, “*el reclutamiento es el proceso de descubrir y capturar candidatos calificados o apropiados para ocupar el puesto vacante*”[9].

El objetivo del reclutamiento es *atraer, buscar e identificar* a los candidatos más adecuados y mejor calificados para el puesto disponible, según las necesidades de la empresa.

El proceso de reclutamiento incluye las siguientes actividades[9]:

- Identificación de las necesidades del puesto a cubrir.
- Análisis de la descripción y especificaciones del puesto.
- Identificación de posibles fuentes de candidatos cualificados para el puesto.
- Publicación del puesto vacante en dichas fuentes.
- Atracción de candidatos para aplicar al puesto.
- Manejo apropiado en las respuestas y en los escrutinios a las postulaciones.

En cambio, la **selección** es un proceso posterior al reclutamiento, donde se evalua más detalladamente y se entrevista a los candidatos para el trabajo en particular. Como mencionan Rahman y Abdullah, “*la selección es un proceso de evaluar y entrevistar a los candidatos para un trabajo en particular y seleccionar a la persona adecuada para el puesto correcto*”[9].

El objetivo de la selección es *elegir y hacer efectiva la contratación* del candidato más adecuado y mejor calificado para el puesto disponible, según las necesidades de la empresa. Este procedimiento partitiona a los candidatos en dos secciones: a los que se les ofrecerán el trabajo, y a los que se descartarán.

El proceso de selección incluye las siguientes actividades[9]:

- Recepción de la aplicación al puesto.
- Cribado o Screening de los candidatos, lo que permite avanzar con los candidatos adecuados y descartar a los no adecuados para el puesto.
- Entrevistas a los candidatos.
- Manejo de tests a los candidatos, tales como tests médicos o psicológicos.
- Manejo de exámenes a los candidatos, tales como exámenes técnicos, de aptitud, inteligencia, performance, etc.
- Evaluación de las referencias de los candidatos.
- Decisión final acerca de la contratación o no contratación del candidato.

Como conclusión, podemos decir que en la fase de **reclutamiento** se trata de encontrar muchos candidatos que cumplan con los requisitos de la oferta; mientras que en la etapa de **selección** se debe elegir al mejor candidato para las necesidades de la empresa.

## 2.3 Evolución de los procesos de reclutamiento y selección laboral.

Los procesos de reclutamiento y selección laboral fueron evolucionando a lo largo del tiempo[22]:

En los modelos de reclutamiento y selección de **primera generación**, las empresas anunciaban sus vacantes de puestos laborales en diarios, revistas, radio y en televisión. Los candidatos enviaban sus currículums por correo postal y los mismos se clasificaban manualmente: algo muy tedioso y que llevaba mucho tiempo en realizar. Una vez preseleccionados los candidatos, los reclutadores llamaban a los mismos para realizar las rondas de entrevistas.

Luego pasamos a la **segunda generación**. En esta época las empresas comenzaron a crecer y también lo hicieron las necesidades de reclutamiento y selección. Las empresas empezaron a subcontratar sus procesos de reclutamiento y selección, naciendo de esta manera las consultoras o agencias de contratación. Estas consultoras requerían que los candidatos cargaran sus currículums en sus sitios web en formatos particulares. Luego, las consultoras revisaban los datos de los candidatos y preselecciónaban a los mismos para la empresa. El gran inconveniente de este proceso fue que habían numerosas consultoras y cada una tenía su propia y única forma de selección, no era un proceso uniforme.

Para intentar superar los problemas anteriores, se llegó a una **tercera generación**, en la que estamos actualmente. En esta generación se crearon, y siguen creándose, una gran cantidad de herramientas de Software que utilizan algoritmos inteligentes y permiten automatizar y gestionar información de los candidatos de una manera mucho más intuitiva. Estos sistemas ayudan a los reclutadores dentro de las empresas y consultoras a analizar la información de cualquier Curriculum Vitae y clasificarlos o listarlos en función de los puestos disponibles. De esta manera, cuando el reclutador publica una oferta de trabajo, estos sistemas clasifican o listan a los currículums basándose en distintas métricas (por ejemplo palabras clave) mostrando así los candidatos más relevantes para la empresa o consultora.

## 2.4 Cribado o screening.

Como vimos anteriormente en *Reclutamiento vs selección*, el screening (o tambien conocido como cribado) es una etapa del proceso de selección. En esta etapa los reclutadores revisan los Currículums Vitae que fueron recibiendo por parte de los candidatos, y

preseleccionan a los que mejor se adapten a los requisitos de la oferta de empleo de la empresa. Este proceso es muy importante dentro del proceso de selección, ya que permite valorar en ese momento si el candidato es apto para continuar en el proceso de selección o, en caso contrario, si el mismo se descarta por no considerarlo adecuado para el puesto.

### 2.4.1 Screening manual vs screening automatizado

Si el proceso de screening sigue el modelo tradicional y se realiza manualmente, es un proceso muy tedioso y que lleva mucho tiempo por parte del reclutador, el cual tiene que evaluar una gran cantidad de Currículums Vitae. Existe un estudio[11] relizado en 2018 por Ladders Inc., empresa líder en sitios de carrera, donde se estudió cuánto tiempo en promedio tarda un reclutador en dar una primera vista rápida a un Curriculum Vitae de un candidato. Se descubrió que este tiempo es, en promedio, de 7.4 segundos. Este es el tiempo en el que el reclutador decide inicialmente si el candidato sigue (o no) con el proceso de selección.

Sin embargo, este tiempo es engañoso: ya que este estudio fue realizado con la aplicación de muchos Currículums vitae que no cumplían con los criterios mínimos para calificar a los puestos; es por esto que los reclutadores realizaban una primer vista rápida y los descartaban (o no) a los pocos segundos. Si las aplicaciones provenieran de Currículums Vitae que cumplen con los requisitos mínimos del puesto, este tiempo de observación sería mucho mayor, ya que el reclutador examinaría con mayor detalle los curriculums.

Adicionalmente, este estudio tampoco tiene en cuenta el tiempo consumido por el reclutador al comparar el Curriculum Vitae del candidato con la descripción o requisitos del puesto laboral, por lo que el tiempo en realizar un screening manual se incrementaría significativamente.

Otra de las desventajas al considerar el screening como un proceso manual, es que muchos son los factores que pueden influir al reclutador en el momento de tomar la decisión de descarte o selección del candidato, ya sea cansancio por el volumen de los currículums ya revisados, la estructura, elementos discriminatorios o información incompleta dentro de los mismos, etc. Es por esto que las probabilidades de descartar prematuramente a un candidato válido son muy altas.

Frente a estas desventajas del modelo tradicional y poco eficiente del screening manual existe una solución: la *automatización*. Actualmente existen sistemas automatizados y basados en Inteligencia Artificial que permiten realizar el screening de un volumen importante de currículums en unos pocos segundos (Ver *Sistemas de screening: Estado del arte*). De esta manera, el proceso de screening no estaría afectado a factores externos que puedan influir en la decisión del reclutador, y además la herramienta sería capaz de entregar un listado justificado de los mejores candidatos para la siguiente fase del proceso en un tiempo relativamente corto.

Uno de los beneficios potenciales de utilizar sistemas automatizados e inteligentes en los procesos de screening es el acortamiento de los ciclos de contratación, lo que hace que la organización responda mejor a los solicitantes y sea más capaz de competir con otras organizaciones por los mejores candidatos[8].

Muchos gerentes de recursos humanos depositan sus esperanzas en la tecnología y las herramientas automatizadas e inteligentes: desde el aumento de la eficiencia y la reducción de costos hasta el aumento de las aplicaciones de candidatos y la estandarización de todos sus sistemas de selección[8].

## 2.5 Sistemas de screening: Estado del arte.

Tal como mencionamos en la sección anterior, un sistema de screening automatizado e inteligente nos permite entregar al reclutador una shortlist con los candidatos que mejor se adaptan a los requisitos de la oferta de empleo de la empresa. De esta manera, al realizar dicha preselección de candidatos, el reclutador podrá seguir con las etapas posteriores de selección o podrá realizar otro screening manual para descartar a otros candidatos.

A continuación detallaremos una serie de trabajos de relevancia (tanto de implementación como de investigación) donde se utilizaron distintos enfoques para desarrollar o sugerir un desarrollo de un sistema de screening de candidatos.

En [12] utilizaron el *algoritmo KNN* para clasificar los curriculums de los candidatos en diferentes categorías, y luego utilizaron la métrica de similitud *Cosine Similarity* para averiguar qué tan cerca está el currículum del candidato con respecto a la descripción de los puestos, y realizar un ranking acorde a estos resultados.

En [13] utilizaron una herramienta inteligente llamada *EXPERT mapping-based candidate screening* que utiliza *ontology mapping*<sup>4</sup> para crear un sistema automatizado para el screening inteligente de candidatos.

En [15] y [16] también se utilizan sistemas basados en *ontologías* para extraer datos de curriculums y realizar macheos con los puestos disponibles.

En [17] sugieren un método de matching basado en un *modelo probabilístico* para ser utilizado en la selección y recomendación de candidatos.

En [18] se propuso un sistema automatizado de screening de currículums que usa *Vector Space Model*<sup>5</sup> para machejar cada currículum con la descripción del puesto correspondiente y *cosine similarity* como medida de similitud.

---

<sup>4</sup>La *ontología (ontology)* permite la especificación explícita de un dominio de discurso, que permite acceder y razonar sobre el conocimiento de un agente. Las ontologías elevan el nivel de especificación del conocimiento, incorporando semántica a los datos. El *mapeo de ontologías (ontology mapping)* es el proceso mediante el cual dos ontologías se relacionan semánticamente a nivel conceptual y las instancias de ontología de origen se transforman en entidades de ontología de destino de acuerdo con esas relaciones semánticas.[14]

<sup>5</sup>Modelo algebraico para representar documentos de textos en lenguaje natural de una manera formal mediante el uso de vectores de identificadores. Es utilizado para la recuperación, filtrado, indexado y cálculo de relevancia de información.

En [19] y [20] se propuso un sistema para el screening de candidatos, para el cual se utiliza *cosine similarity*, *KNN* y un *sistema de recomendación basado en contenido (Content Based Filtering, CBF)* para buscar los currículums más cercanos a la descripción del puesto.

En [21] se propuso un sistema para el ordenamiento y clasificación de currículums utilizando el concepto de inteligencia artificial (sin especificar qué algoritmos o técnicas específicas se deberían usar). Lo que se propuso fue un esquema de trabajo para luego seguir con la implementación del sistema; el cual permitiría clasificar a todos los currículums de acuerdo con los requisitos de la empresa y los enviaría a Recursos Humanos para su posterior consideración.

En [22] se diseñó un sistema que ayuda a los reclutadores a seleccionar los currículums basados en la descripción del trabajo. Ayuda en un proceso de contratación fácil y eficiente al extraer los requisitos automáticamente. Este sistema utiliza un *modelo NER (Named entity recognition)*<sup>6</sup>.

En [23] y [24] se implementan sistemas basados en *semantic annotation*<sup>7</sup> y *ontologías*, permitiendo realizar macheos entre las ofertas de trabajo y los currículums de los candidatos.

En [25] se utiliza *relevance feedback*<sup>8</sup> para la implementación de un sistemas que intenta encontrar a los mejores candidatos para las distintas ofertas de trabajo.

*Paul Resnick y Hal R. Varian* fueron pioneros en los *sistemas de recomendación*<sup>9</sup>[28]. En [26] y [27] se utilizaron dichos *sistemas de recomendación* junto a otros algoritmos para implementar aplicaciones de screening y clasificación de candidatos. Esto es debido a que, dado que el perfil del candidato es necesario para un puesto en particular, la forma en que se comparan los currículums de los candidatos es muy similar a un sistema de recomendación.

---

<sup>6</sup>Subtarea de *information extraction (IE)* que permite buscar y categorizar entidades específicas en un cuerpo o cuerpos de textos.

<sup>7</sup>Proceso de etiquetado de documentos con conceptos relevantes. Los documentos se enriquecen con metadatos que permiten vincular el contenido a los conceptos, descritos en un gráfico de conocimiento. Esto hace que el contenido no estructurado sea más fácil de encontrar, interpretar y reutilizar.

<sup>8</sup>Característica de algunos sistemas de *information retrieval (IR)*. La idea de relevance feedback es tomar los resultados que se devuelven inicialmente de una consulta/query determinada, recopilar los comentarios de los usuarios y usar información para evaluar si esos resultados son relevantes o no para realizar una nueva consulta/query.

<sup>9</sup>Un *sistema de recomendación* es una subclase de los sistemas de *Information filtering (IF)* que busca predecir la calificación o la preferencia que un usuario le puede dar a un artículo. En palabras simples, es un algoritmo que sugiere artículos relevantes para los usuarios.

## 2.6 Enfoque del Proyecto.

Como vimos en la sección anterior, hay cientos de enfoques y combinaciones posibles para realizar un sistema de screening automático e inteligente.

El enfoque elegido para este Proyecto de Tesis fue realizar un sistema híbrido que utilice técnicas de mediciones de similitud entre los currículums de los candidatos y las descripciones de los puestos disponibles, junto con técnicas de machine learning (de clustering y clasificación).

Como métricas de medición de similitud se utilizó una de las más comúnmente usadas, *similitud del coseno*, y una métrica que **no se usó en ninguno de los trabajos anteriores** pero, que sin embargo, es muy reciente y prometedora, y es la principal técnica utilizada para la medición semántica de la distancia entre textos, *Word Mover's Distance (WMD)*. Estas métricas serán explicadas más en detalle en las secciones posteriores (Ver *Maneras de medir la similitud entre textos*).

Una vez utilizadas estas mediciones de similitud se utilizaron algoritmos de machine learning: un algoritmo de clustering K-means, y luego un modelo de clasificación KNN.

Además del uso de *Word Mover's Distance*, otro aspecto distintivo de este sistema es que se realizó una **integración a una interfaz web**, la cual la mayoría de los sistemas descriptos anteriormente no poseen. De esta manera, mediante la interfaz web se logra una interacción entre usuarios candidatos y reclutadores y, principalmente, permite que el reclutador sea capaz de visualizar un listado con los N candidatos más similares a un puesto determinado.

Por último, cabe destacar que los trabajos anteriormente mencionados no tienen un fácil acceso a los datasets ni al código que utilizaron para dichos sistemas, por lo que seguir el trabajo de ellos aplicando las mejoras que mencionan en sus artículos es una tarea casi imposible. En cambio, **este sistema será de código abierto**: los datasets, modelos y códigos utilizados estarán disponibles en Github y Git LFS<sup>10</sup>.

---

<sup>10</sup>[https://github.com/FedericoCalonge/automatic\\_reading\\_of\\_CVs\\_using\\_text\\_similarity](https://github.com/FedericoCalonge/automatic_reading_of_CVs_using_text_similarity)

### 3 Algoritmos de Machine Learning.

#### 3.1 Introducción.

Continuando con el marco teórico de esta Tesis, es necesario explicar qué es *machine learning (ML)* y cómo se pueden clasificar a los distintos algoritmos y modelos según su tipo de aprendizaje, haciendo énfasis en los algoritmos K-Nearest Neighbor (KNN) y K-means, y el modelo de Redes Neuronales. Estas técnicas son las que se utilizarán en la implementación del proyecto.

Además, detallaremos la importancia de la separación de los datos y mencionaremos cuáles son los pasos a seguir para implementar un modelo de ML teniendo en cuenta algunos de los problemas clásicos que pueden afectar a nuestros resultados.

#### 3.2 Machine Learning (ML).

La **Inteligencia Artificial (IA)** es una disciplina dentro de las ciencias de la computación que consiste en el desarrollo de algoritmos<sup>11</sup> que imiten el razonamiento humano, teniendo la capacidad de solucionar problemas que comúnmente resuelve la inteligencia natural pero de la manera más eficiente posible. Esencialmente, la IA permite que las máquinas puedan actuar e implementar distintas tareas que están fuera del alcance de los humanos [29].

**Machine Learning (ML)** o **Aprendizaje Automático** es un subconjunto de la Inteligencia Artificial que se encarga de construir algoritmos que aprenden a hacer algo útil a partir de los datos. Como ML es un subconjunto de IA, esto implica que todos los algoritmos de ML son técnicas de inteligencia artificial, pero no todos los métodos de inteligencia artificial califican como algoritmos de ML.

El objetivo principal de ML es permitir que las computadoras aprendan sin intervención o asistencia humana. Esencialmente, los algoritmos de ML aprenden *patrones* ocultos basados en datos históricos de entrada, los cuales posteriormente utilizan para aprender a clasificar la información o realizar predicciones relacionadas con el problema que se quiera resolver.

Los algoritmos de ML hoy en día se usan en todo tipo de aplicaciones: en Amazon para recomendarnos qué productos comprar, en Twitter para recomendarnos qué usuarios seguir, en Google para predecir qué páginas son las más relevantes para una consulta, en Facebook para reconocer qué fotos contienen rostros de personas, en el mercado inmobiliario para predecir los precios de las propiedades, en el mercado de valores para predecir el precio de las acciones, etc. Son tantas las aplicaciones de ML en la industria que se ha producido una verdadera explosión del tema en los últimos años[10].

---

<sup>11</sup>Un algoritmo informático es un conjunto de instrucciones definidas, ordenadas y acotadas para resolver un problema, realizar un cálculo o desarrollar una tarea.

### 3.2.1 Aprendizaje supervisado y no supervisado.

Podemos dividir a ML en dos grandes categorías: aprendizaje supervisado y aprendizaje no supervisado. Resumidamente, la diferencia entre estas dos categorías es la existencia de etiquetas (labels) en el subconjunto de datos de entrenamiento (Ver *Separación de los datos*).

- En el **aprendizaje supervisado**[30], el modelo es entrenado utilizando datos que están “etiquetados”. Esto quiere decir que el set de datos que es utilizado para enseñar al algoritmo tiene una etiqueta (o label) que define la respuesta/salida correcta.

Dicho de otra manera, el modelo recibe datos etiquetados (tanto para la entrada como para la salida esperada) durante su entrenamiento teniendo como objetivo generar una función de mapeo  $f(x)$  que pueda identificar la salida esperada  $y$  para una entrada  $x$  dada. El proceso de entrenamiento continúa hasta que el algoritmo alcanza el nivel de precisión deseado.

De esta manera al finalizar el entrenamiento idealmente la función  $f$  predecirá resultados con gran precisión para aquellos nuevos datos que no tienen etiqueta.

Los algoritmos de aprendizaje supervisado se utilizan para resolver problemas de **clasificación** y de **regresión**.

- El **aprendizaje no supervisado**[30], en cambio, se refiere al proceso de entrenamiento de un modelo de machine learning sin un set de datos etiquetado. Este tipo de aprendizaje es una rama muy importante en Data Science ya que, al trabajar solo con datos sin necesidad de tener un “label” para los mismos, únicamente necesitamos los datos en crudo y estos en general son mucho más fáciles de conseguir que datos ya previamente clasificados. Por ejemplo, las aplicaciones de redes sociales tienen grandes cantidades de datos sin etiquetar.

En otras palabras, el modelo recibe un conjunto de datos de entrada sin etiquetar y sin clasificar, y el algoritmo de aprendizaje no supervisado experimentará con este conjunto de datos aprendiendo patrones, similitudes y diferencias entre los mismos sin ningún entrenamiento previo. El objetivo final es generar una función  $f$  que permita identificar estructuras ocultas en el conjunto de datos dado.

Los algoritmos de aprendizaje no supervisado se utilizan para resolver problemas de **clustering**.

En la Figura 3.1 se detalla un resumen de la división en las categorías de ML y su sub-división en modelos de clasificación, regresión y agrupación (clustering).

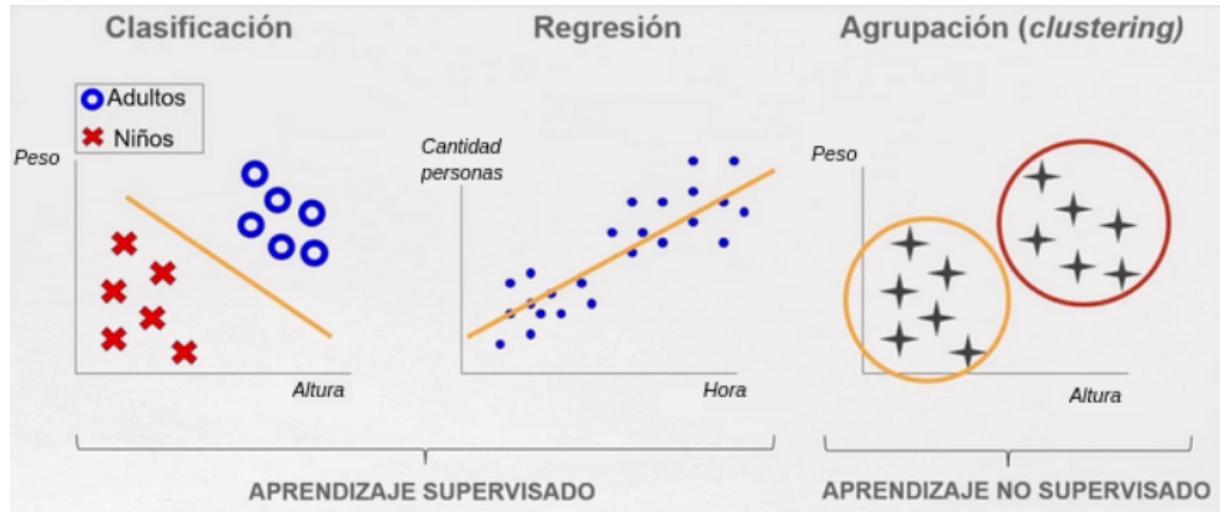


Figura 3.1: Tipos de aprendizaje en ML<sup>12</sup>

### 3.2.1.1 Regresión.

En un problema de regresión se intenta predecir el valor de una variable numérica y continua a partir de un cierto conjunto de datos.

En general contamos con un set de entrenamiento en el cual conocemos el valor de la variable que queremos predecir. El objetivo es entonces construir un modelo que nos permita predecir el valor de nuestra variable de salida a partir de datos nuevos. Dicho de otra forma, lo que se hace es buscar una función que represente los datos de entrenamiento y que permita generalizar correctamente<sup>13</sup>.

El caso más simple es la regresión lineal (observado previamente en la Figura 3.1), en el cual nuestro modelo es una recta, la recta que mejor se ajusta a los puntos de nuestro set de entrenamiento. En este ejemplo se intenta predecir la cantidad de personas en un lugar a una hora determinada.

Los problemas de regresión pueden usarse para realizar distintas predicciones, ya sea el valor de las acciones en el mercado de valores, predecir el costo de una propiedad, estimar las ganancias de un negocio, etc.

Las claves para identificar un problema de regresión[10] son las siguientes:

- Queremos predecir una variable que es numérica y continua.
- Contamos con un set de entrenamiento para el cual conocemos el valor de dicha variable.

<sup>12</sup>Sitio Web. Modelos de Machine Learning. <https://openwebinars.net/blog/modelos-de-machine-learning/>. (Consultado el 30 marzo de 2022).

<sup>13</sup>Generalizar es la capacidad de nuestro modelo de ML de obtener buenos resultados con datos nuevos reconociendo patrones generales de los datos de entrenamiento en lugar de reconocer particularidades específicas.

### 3.2.1.2 Clasificación.

En los problemas de clasificación la variable que se intenta predecir no es continua sino discreta, frecuentemente tiene pocos valores posibles y en muchos casos los valores posibles son solo dos: clasificación binaria<sup>14</sup>. La idea general es la misma que antes, para este caso contamos con un set de entrenamiento en el cual para cada dato conocemos la clase a la cual pertenece el mismo, y queremos construir un modelo que nos permita clasificar automáticamente datos nuevos cuya clase desconocemos.

Por ejemplo, en la Figura 3.1, observamos que tenemos una serie de personas, de las cuales contamos con su peso y altura, y queremos clasificarlas en adultos o niños. Lo que va a hacer un modelo de clasificación es aprender dónde están estos puntos y crear un clasificador que, para nuevos datos de entrada, consiga segmentarlos correctamente en adultos o niños.

Otras aplicaciones, por ejemplo, podrían ser analizar si las reviews de un producto son buenas o malas, medir la actitud del público en general ante determinadas noticias por los comentarios que existen en redes sociales, clasificar si un email es spam o no lo es, etc.

Las claves para reconocer un problema de clasificación[10] son:

- Queremos determinar la clase / grupo a la que pertenece cada dato.
- La clase es una variable discreta con un set de valores posibles limitado y definido.
- Contamos con un set de entrenamiento para el cual conocemos los datos y a qué clase pertenecen.

---

<sup>14</sup>Un caso típico de la clasificación binaria es un problema de análisis de sentimiento, donde queremos saber si un cierto texto es positivo o negativo, es decir si habla bien o mal de un cierto tema. Como set de entrenamiento entonces deberíamos contar con textos para los cuales ya conocemos su sentimiento.

### 3.2.1.3 Agrupación (clustering).

En un problema de clustering contamos con datos que queremos dividir en grupos de forma automática. Estos datos no tienen etiquetas, no sabemos a qué grupo pertenecen. Lo que hace un algoritmo de clustering es intentar buscar agrupaciones en los datos, creando de esta forma clusters<sup>15</sup> con características similares. En algunos casos la cantidad de clusters la debemos indicar previamente y en otros el algoritmo es capaz de determinarla por sí mismo.

En el ejemplo de la Figura 3.1, podemos ver que se agrupan los puntos de la misma forma que en el problema de clasificación. La diferencia es que en este caso no sabemos si se trata de adultos o niños, sino que es el propio algoritmo el que va a identificar que hay dos grupos, y nosotros somos los que tenemos que interpretar los resultados.

Otros ejemplos de clustering podrían ser agrupar películas automáticamente de forma que queden juntas las que son de un mismo género; o la detección de comunidades en una red social, el cual es un típico problema de clustering en donde los puntos son los usuarios y queremos agruparlos automáticamente en comunidades. De esta forma podemos descubrir grupos de usuarios que tienen un cierto interés en común aun sin saber exactamente cuál es dicho interés.

Las claves para reconocer un problema de clustering[10] son:

- Contamos con un set de datos y queremos agruparlos en clusters/grupos.
- No son necesarios labels.

---

<sup>15</sup>Cluster, o grupo, es un conjunto de objetos que son "similares" entre ellos y "diferentes" de los objetos que pertenecen a los otros grupos. La palabra cluster viene del inglés y significa agrupación. Desde un punto de vista general, el cluster puede considerarse como la búsqueda automática de una estructura o de una clasificación en una colección de datos no etiquetados.

### 3.2.1.4 Algoritmos más conocidos.

Resumidamente, en la Figura 3.2 se detallan los algoritmos más conocidos para las distintas clasificaciones de machine learning previamente explicadas.

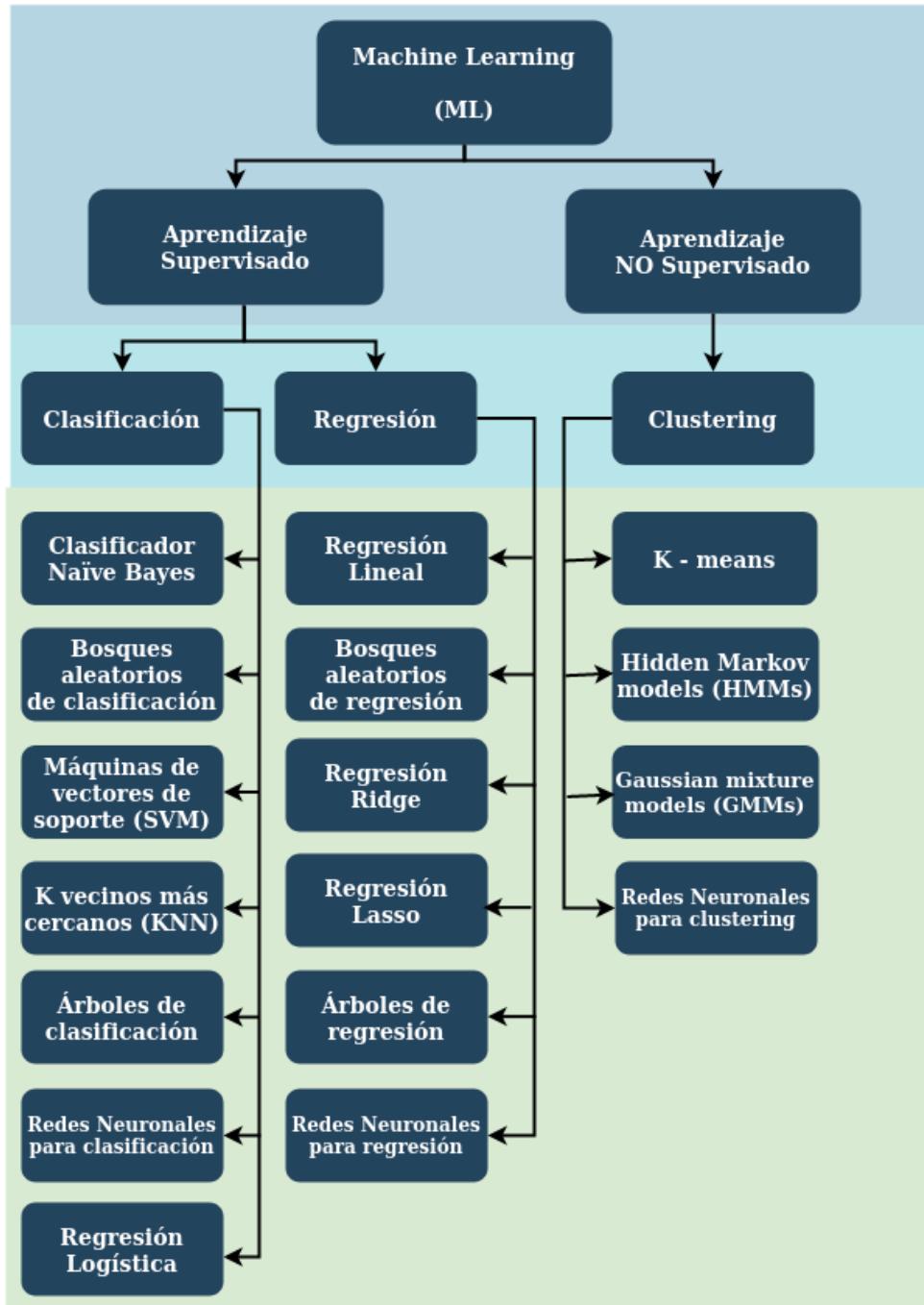


Figura 3.2: Algoritmos de machine learning más conocidos

Cabe destacar que algunos algoritmos se pueden adaptar para resolver problemas de otro tipo al cual están inicialmente categorizados. Por ejemplo, KNN que está categorizado como un algoritmo de clasificación, también se puede utilizar para resolver problemas de regresión. Otro caso conocido es Support Vector Regression (SVR), que utiliza las bases de Máquinas de vectores de soporte (SVM) para resolver problemas de regresión.

### 3.2.2 Aprendizaje transductivo

Existe una cierta relación entre los problemas de clustering y de clasificación, por ejemplo dado un problema de clasificación podríamos aplicar clustering primero y luego clasificar a cada punto de acuerdo al cluster al cual pertenece en base a la clase mayoritaria de dicho cluster. Este procedimiento no es muy frecuente pero es conveniente tenerlo en cuenta porque permite entender el funcionamiento de ciertos algoritmos que combinan las propiedades de un problema de clustering y uno de clasificación.

Una aplicación que combina clustering (aprendizaje no supervisado) y clasificación (aprendizaje supervisado) es la que denominamos **Aprendizaje transductivo**[31][10]. De esta manera, para predecir nuevos datos no etiquetados se utilizan los datos previamente etiquetados, así como los datos sin etiquetar, como forma de ayuda a un clasificador tradicional.

Consideremos el ejemplo de la Figura 3.3. Aquí tenemos solo dos puntos clasificados, uno clasificado como “blanco” y el otro como “negro”. Sin mayor información el punto marcado con el signo de pregunta, cuya clase desconocemos, quedaría clasificado como “blanco” ya que está mucho mas cerca del punto blanco que del punto negro.



Figura 3.3: Aprendizaje transductivo

En la Figura 3.4 al agregar puntos cuya clase desconocemos, vemos que en nuestros datos existen dos clusters. De esta manera, si tenemos que asociar cada cluster con un color entonces el de arriba es “negro” y el de abajo es “blanco”, ya que el punto negro pertenece al cluster de arriba y el blanco al de abajo. De este modo, el punto que inicialmente consideramos blanco, con esta nueva información de clusters, sería considerado negro.

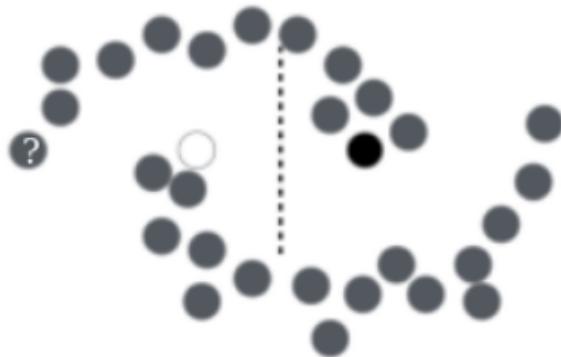


Figura 3.4: Aprendizaje transductivo

El aprendizaje transductivo es un área muy nueva dentro de Data Science y que sin dudas merece ser explorada.

En este proyecto de Tesis se utilizará el aprendizaje transductivo para lograr inicialmente una clusterización de nuestros candidatos en 5 grupos mediante un algoritmo de clustering (K-means), y luego se realizará una clasificación mediante un algoritmo de clasificación (KNN).

### 3.2.3 Separación de los datos.

Dos conceptos importantes que mencionaremos a lo largo del informe son: los “datos de entrenamiento” y los “datos de prueba”.

Como mencionamos anteriormente, los algoritmos de machine learning aprenden de los datos con los que los entrenamos. A partir de ellos, intentan encontrar o inferir el patrón que les permita predecir el resultado para un nuevo caso. Pero, para poder calibrar si un modelo funciona, necesitaremos probarlo con un conjunto de datos diferente. Por ello, en todo proceso de aprendizaje automático, los datos de trabajo se deben dividir mínimamente en dos partes:

- Los **datos de entrenamiento** son los datos que usamos para entrenar un modelo. La calidad de nuestro modelo de aprendizaje automático va a ser directamente proporcional a la calidad de estos datos. Por ello son muy importantes las tareas de limpieza y preprocesamiento de los mismos.
- Los **datos de prueba o evaluación** son los datos que nos “reservamos” para comprobar si el modelo que hemos generado a partir de los datos de entrenamiento “funciona”. Es decir, si las respuestas predichas por el modelo para un caso totalmente nuevo son acertadas o no.

Es importante que el conjunto de datos de prueba tenga un volumen suficiente como para generar resultados estadísticamente significativos, y a la vez, que sea representativo del conjunto de datos global. Normalmente el conjunto de datos se suele dividir en un **70 %/80 % de datos de entrenamiento** y un **30 %/20 % de datos de prueba**, pero se puede variar la proporción según el caso[10].

### 3.2.4 ¿Cómo implementar un modelo de ML?

Resumidamente, los pasos a seguir para implementar un modelo de ML y utilizarlo para realizar predicciones sobre los datos son los siguientes:

- Paso 1: Recolección de Datos.
- Paso 2: Preparación y preprocesamiento de los datos.
- Paso 3: Elección del modelo de ML.
- Paso 4: Entrenamiento del algoritmo.
- Paso 5: Evaluación del modelo.
- Paso 6: Parameter Tuning o configuración de parámetros.
- Paso 7: Utilizando nuestro modelo.

A continuación detallaremos en mayor detalle los mencionados pasos.

- Paso 1: Recolección de Datos.

Dada la problemática que deseemos resolver mediante algoritmos de ML, nuestro primer paso será recolectar los datos que utilizaremos posteriormente para “alimentar” a dicho algoritmo.

En este paso hay que tener muy en cuenta la calidad y cantidad de información que consigamos ya que impactará directamente en lo bien o mal que luego funcione nuestro modelo. Estos datos los podemos sacar de bases de datos, planillas de cálculo, utilizando técnicas de web scraping<sup>16</sup> o mediante APIs<sup>17</sup> para recopilar información de manera automática de diversas fuentes de Internet, etc.

- Paso 2: Preparación y preprocesamiento de los datos.

En este paso generalmente se realizan visualizaciones de los datos y se revisa si existen correlaciones entre las distintas “features”<sup>18</sup> de nuestros datos. Al pre-procesar nuestros datos nos referimos a normalizar los mismos: eliminando duplicados y realizando distintas correcciones de errores. El preprocesamiento de datos usualmente tiene un impacto significativo en la performance de generalización de nuestro algoritmo de machine learning[33].

---

<sup>16</sup>Proceso dentro de Data Science que se utiliza para la extracción de datos de sitios web simulando cómo navegaría un ser humano por los mismos.

<sup>17</sup>API o interfaz de programación de aplicaciones, es un conjunto de métodos o funciones que ofrece cierta biblioteca para ser utilizada por otro software como una capa de abstracción.

<sup>18</sup>Features generalmente son las columnas de nuestro dataframe, archivo o base de datos -dependiendo cómo almacenamos nuestros datos-.

En esta etapa, además, es importante **separar** nuestros datos en dos grupos:

- un set de entrenamiento.
- un set de prueba.

Como mencionamos previamente, el set de test en general es un 20 % o 30 % del set de entrenamiento. Esta partición de los datos en estos dos conjuntos diferenciados permite generar el modelo a partir de los datos de entrenamiento para después comprobar su eficiencia con los datos reservados para test.

- Paso 3: Elección del modelo de ML.

Una vez obtenidos y preprocesados estos datos, lo que se hace es **elegir el modelo de ML**<sup>19</sup> de acuerdo al objetivo que tengamos o problema que deseemos resolver.

De esta manera, utilizaremos algoritmos de clasificación, regresión o clustering para construir nuestro modelo de ML a partir de los datos, de forma tal de luego poder usar dicho modelo para predecir nuevos datos.

- Paso 4: Entrenamiento del algoritmo.

El siguiente paso es **entrenar** a nuestro algoritmo de ML. En este proceso mediante una serie de iteraciones nuestro algoritmo detecta patrones en nuestros datos que luego nos servirán para poder realizar predicciones con los nuevos datos que se incorporen al sistema.

La idea es entrenar a nuestro algoritmo con el set de entrenamiento (mediante una función *fit()*) para luego, en las etapas posteriores, aplicarlo al set de prueba (mediante una función *predict()*). De esta forma, los datos para los cuales queremos probar el algoritmo (set de test) nunca fueron vistos por el mismo en la etapa de entrenamiento, lo cual permite saber si el modelo fue capaz de generalizar correctamente.

- Paso 5: Evaluación del modelo[10].

Hacer predicciones correctas sobre datos futuros suele ser el principal problema que queremos resolver al utilizar algoritmos de ML. Luego de entrenar el modelo se tiene que **evaluar** el mismo. Evaluar un modelo, resumidamente, es estimar su rendimiento para saber qué tan bien se desempeñará / predecirá para datos nuevos no vistos por el mismo.

Para poder evaluar un modelo correctamente, tenemos que contar con la separación de nuestros datos en set de entrenamiento y set de prueba que realizamos en los pasos previos. Esto lo hacemos, ya que evaluar la precisión predictiva de un modelo de ML con los mismos datos que se han utilizado para el entrenamiento no es útil, ya que compensa a los modelos que pueden “recordar” los datos de entrenamiento en lugar de generalizar.

---

<sup>19</sup>Un modelo de machine learning es la salida de información que se genera cuando se entrena un algoritmo de ML con datos. Después del entrenamiento, al proporcionar un modelo con una entrada, se le dará una salida. Por ejemplo, un algoritmo predictivo creará un modelo predictivo.

De esta manera, luego de haber entrenado nuestro modelo de ML, resumidamente lo que hacemos en esta etapa es comparar las predicciones realizadas sobre set de pruebas devueltas por el modelo de ML contra el valor de destino conocido para el mismo set de pruebas; y por último generar alguna **métrica de evaluación** que nos permite verificar la performance de nuestro modelo indicando la efectividad de las predicciones.

Dependiendo del tipo de modelo que tengamos, podemos utilizar distintas métricas de evaluación para verificar su performance. Como observación, se detallan algunas de las métricas más conocidas en la Tabla 1.

Modelos de regresión	Modelos de clasificación	Modelos de clustering
Mean square error (MSE)	Matriz de confusión o error	Inertia
Root MSE (RMSE)	Accuracy (Exactitud)	Homogeneity
Normalized RMSE (NRMSE)	Precision (Precisión)	Majority-representation
Mean absolute error (MAE)	Recall (Sensibilidad o TPR)	Adjusted Rand Index
Mean absolute percentage error (MAPE)	FP Rate (Especificidad o TNR)	Silhouette coefficient
		Dunn index

Tabla 1: Métricas para evaluar distintos tipos de modelos[34, 35, 36, 37].

Por ejemplo, si queremos verificar la performance de un modelo de clasificación podemos utilizar el **accuracy**. Esta métrica mide el % de aciertos: es el ratio de las predicciones correctas sobre el número total de instancias evaluadas. Por ejemplo, si el Accuracy es menor o igual al 50% este modelo no será útil ya que sería como lanzar una moneda al aire para tomar decisiones. Si alcanzamos un 90% o más podremos tener una buena confianza en los resultados que nos otorga el modelo.

- Paso 6: Parameter Tuning o configuración de parámetros[10].

Si durante la evaluación no obtuvimos buenas predicciones y nuestra métrica de evaluación no logró ser la mínima deseada, es posible que tengamos problemas de overfitting (ó underfitting) y deberemos retornar al paso de entrenamiento (Paso 4) haciendo antes una nueva configuración de hiper-parámetros de nuestro modelo.

Cada modelo tiene un conjunto de parámetros e hiper-parámetros que necesita para funcionar:

- Los **parámetros** son los valores obtenidos por el propio algoritmo a partir de los datos, no son indicados manualmente.
- Los **hiper-parámetros**, en cambio, son datos que debemos pasarle al algoritmo para funcionar.

Como ejemplos de parámetros tenemos los coeficientes en una regresión lineal o logística, o los pesos en una red neuronal artificial. Y como ejemplos de hiper-parámetros, tenemos al ‘k’ en KNN, o los ‘EPOCHs’ que nos permiten incrementar la cantidad de veces que iteramos sobre nuestros datos de entrenamiento en una red neuronal artificial.

Para encontrar los hiper-parámetros óptimos, es decir, aquellos que mejor funcionan para nuestro set de datos, lo que se hace es, nuevamente, realizar un entrenamiento y una evaluación de nuestro modelo de ML seleccionando e iterando sobre los distintos hiper-parámetros que tengamos: a esto se le conoce como “*Parameter Tuning*”. Esta iteración puede ser de manera aleatoria (método conocido como *random search*) o completa (*grid search*). Luego de realizar esta iteración se obtiene la configuración de hiper-parámetros más óptima.

Algo a tener en cuenta es que, para realizar la evaluación, no debemos usar los datos del set de prueba, ya que podríamos caer en un caso de overfitting seleccionando los parámetros que funcionan mejor para los datos del set de pruebas, pero tal vez no los parámetros que generalicen mejor. Para evaluar el modelo y realizar dicho “*Parameter Tuning*” lo que necesitamos es dividir el set de entrenamiento original en dos: un set de entrenamiento y un **set de validación**.

La idea es entrenar el modelo con el set de entrenamiento y luego probarlo con dicho set de validación (NO con el set de pruebas) a efectos de encontrar los mejores hiper-parámetros. El set de validación en general es un 20-30 % del set de entrenamiento original.

Por último, una vez hallados los hiper-parámetros más óptimos, lo que haremos es realizar una **evaluación final** sobre el set de pruebas con los hiper-parámetros que encontramos, y de esta manera nos dará el accuracy (o la métrica de evaluación que elegimos) para los datos que el modelo no vió.

No obstante, existen 2 problemas con el esquema anterior:

- Siendo el set de validación siempre el mismo podemos caer en el problema de que los hiper-parámetros que encontramos solo sean óptimos para nuestro set de validación (el cual es un pequeño conjunto de nuestros datos).
- La división del set de entrenamiento en un nuevo set de entrenamiento y un set de validación hace que haya menos datos disponibles para el entrenamiento. Esto resulta un problema, especialmente para conjuntos de datos pequeños, ya que siempre es mejor utilizar el mayor número de datos posible para el entrenamiento.

Para evitar estos 2 problemas utilizamos el método de **Cross Validation** (o validación cruzada), el cual es prácticamente universal para optimizar algoritmos de ML. Este método nos permite evaluar modelos de ML solucionando los 2 problemas mencionados anteriormente previniendo el overfitting. Para mayor detalle de su funcionamiento ver *Cross Validation*.

#### ■ Paso 7: Utilizando nuestro modelo.

Una vez que obtuvimos buenas predicciones en la etapa de evaluación y nuestra métrica de evaluación llegó a ser la mínima deseada, ya podemos afirmar que estamos en condiciones de utilizar nuestro modelo de machine learning entrenado para realizar predicciones con nuevos datos que están fuera del set de entrenamiento y del set de test.

### 3.2.4.1 Cross Validation.

El método de **Cross Validation**, o validación cruzada[10][32], consiste en entrenar modelos de ML en subconjuntos de los datos de entrada disponibles y evaluarlos con un subconjunto complementario de los datos.

En la Figura 3.5 podemos observar el funcionamiento del enfoque básico de Cross Validation: **k-fold Cross Validation**. Este método comienza particionando el set de entrenamiento en k bloques (o “folds”). Luego, se realizan varias iteraciones en las cuales entrenamos nuestro algoritmo con k - 1 bloques como set de entrenamiento y lo validamos con el bloque restante (el cual sería nuestro set de prueba), para obtener el valor de alguna métrica de evaluación, como por ejemplo el accuracy. Este proceso se repite k veces con el objetivo de que todos los bloques de datos hayan participado alguna vez del set de pruebas.

El resultado de la métrica de evaluación final devuelta por k-fold Cross Validation es el **promedio** de los resultados de dicha métrica de evaluación calculada dentro de las k iteraciones del algoritmo. Esto hay que hacerlo además por cada valor posible para nuestros hiper-parámetros, por lo que, dependiendo de los datos, puede resultar un proceso costoso.

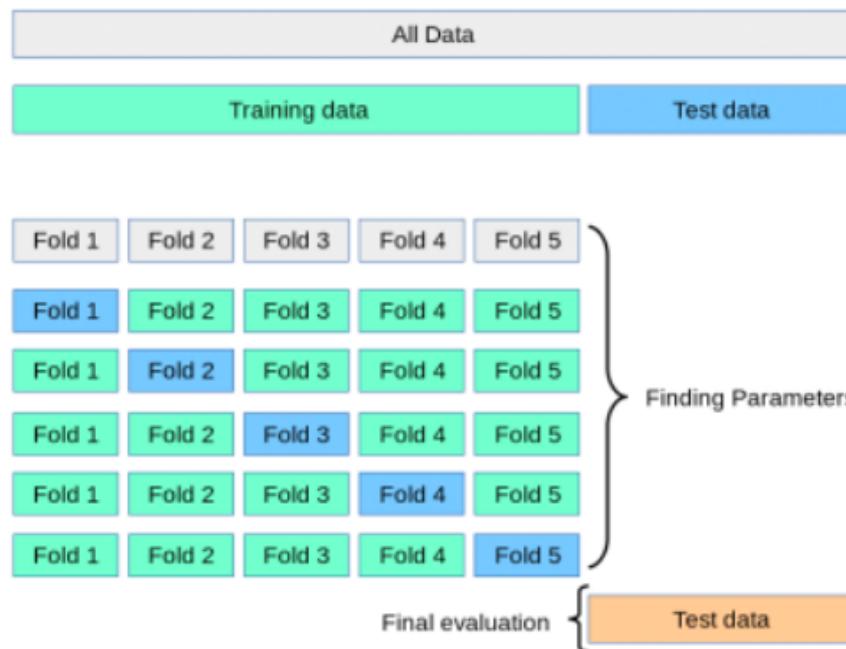


Figura 3.5: k-fold Cross Validation<sup>20</sup>

En conclusión, luego de realizar dichas iteraciones mediante k-fold Cross Validation, obtendremos la métrica de evaluación final para cada uno de las combinaciones de nuestros hiper-parámetros, y elegiremos los hiper-parámetros más óptimos (el que mejor métrica de evaluación nos haya dado).

Una vez hallados los mismos, el último paso, como comentamos con el primer esquema, es realizar una evaluación final sobre el set de pruebas con los hiper-parámetros que

<sup>20</sup>Librería Scikit-learn. *Cross-Validation: evaluating estimator performance*. [https://scikit-learn.org/stable/modules/cross\\_validation.html](https://scikit-learn.org/stable/modules/cross_validation.html). (Consultado el 30 marzo de 2022).

encontramos, y de esta manera nos dará el accuracy (o la métrica de evaluación que elegimos) para los datos que el modelo no vió.

Un ejemplo de la utilización de estos esquemas para la evaluación de modelos, lo podemos observar en *Armado del modelo de clasificación KNN*.

### 3.2.4.2 Los Problemas de ML: Overfitting y Underfitting.

En esta sección vamos a nombrar y explicar algunos de los problemas clave en tareas de ML: overfitting y underfitting. Podemos observar un ejemplo de estos conceptos en la Figura 3.6.

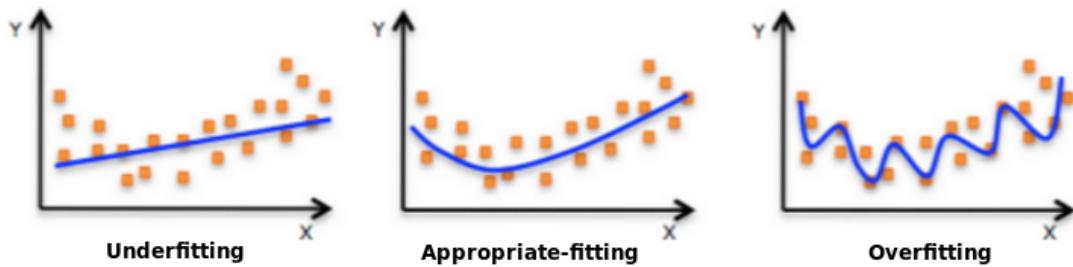


Figura 3.6: Overfitting y Underfitting[10]

**Overfitting**, o sobre-ajuste, es un problema clave en las tareas de aprendizaje automático supervisado. Es el fenómeno que se detecta cuando un algoritmo de aprendizaje se ajusta/entrena tan bien al set de datos de entrenamiento que se memorizan el ruido y peculiaridades específicas de los datos de entrenamiento. Entonces, se vuelve difícil para el modelo generalizar a nuevos ejemplos que no estaban en el conjunto de entrenamiento. Es por esto que la precisión del modelo cae cuando se prueba en un conjunto de datos desconocido. De esta manera, termina siendo mayor la precisión en el entrenamiento que la precisión sobre los set de pruebas.

La cantidad de datos utilizados para el proceso de aprendizaje es fundamental en este contexto. Los conjuntos de datos pequeños son más propensos al overfitting que los conjuntos de datos grandes. El ajuste excesivo de los datos de entrenamiento conduce al deterioro de las propiedades de generalización del modelo y da como resultado un rendimiento o precisión del mismo poco fiable[38].

El concepto de overfitting está asociado a la complejidad del modelo. Un modelo excesivamente complejo puede ajustar tan bien como queramos al set de entrenamiento pero funcionar muy mal para el set de test[10].

En cambio **Underfitting**, o sub-ajuste, es el opuesto de Overfitting. Esto ocurre cuando el modelo es incapaz de capturar la variabilidad de los datos. Se produce cuando el modelo es demasiado simple. Si nuestros puntos están distribuidos en forma curva, un modelo lineal es demasiado simple, no tiene el poder expresivo necesario para representar correctamente el set de entrenamiento[10].

Por lo tanto, podemos decir que el modelo óptimo es aquel que tiene la complejidad necesaria para capturar lo que los datos expresan pero no más.

### 3.3 K-Nearest Neighbor (KNN).

K-Nearest Neighbor o K Vecinos más cercanos (KNN), es un algoritmo ML que se puede usar tanto en tareas de regresión como de clasificación. En nuestra implementación, lo usaremos para resolver un problema de clasificación (ver *Clasificación*). Como mencionamos anteriormente, en los problemas de clasificación la variable que se intenta predecir es discreta. En este tipo de problemas contamos con un set de entrenamiento en el cual para cada dato conocemos la clase a la cual pertenece el mismo, y queremos construir un modelo que nos permita clasificar automáticamente datos nuevos cuya clase desconocemos.

Como lo dice su nombre, el algoritmo KNN se basa en encontrar para un determinado punto  $m$ , sus  $K$ -vecinos más cercanos. Esto es asumiendo que nuestro set de datos está formado por un conjunto de  $m$  puntos en  $n$  dimensiones siendo todos los valores numéricos.

Para poder utilizar KNN hay que definir previamente dos hiper-parámetros:

- La métrica a usar para calcular las distancias.
- El valor de  $k$ , es decir cuantos vecinos vamos a considerar.

Algo a destacar, es que KNN es un tipo de aprendizaje basado en la memoria, también llamado aprendizaje basado en instancias, que pertenece al aprendizaje perezoso (“lazy learning”). Esto quiere decir que KNN **no tiene una fase de entrenamiento**. ¿Qué quiere decir que KNN no tenga un proceso de entrenamiento?, y entonces en *Armado del modelo de clasificación KNN* ¿qué se hace realmente al utilizar la función `.fit()` provista por sklearn?. A continuación se contestarán estas preguntas:

- **A nivel conceptual**, entrenar un clasificador significa tomar un conjunto de datos como entrada, y obtener a la salida un modelo clasificador identificado con distintos parámetros, los cuales son obtenidos en esta etapa de entrenamiento mediante iteraciones realizando cálculos numéricos o resolviendo problemas de optimización (como por ejemplo la obtención del mejor hiperplano en SVM o el ajuste de los pesos en redes neuronales artificiales). En el caso de KNN, el clasificador no se obtiene luego de iterar y obtener dichos parámetros, sino que se identifica por los propios datos de entrenamiento. Entonces, conceptualmente, entrenar un clasificador KNN simplemente requiere almacenar el conjunto de entrenamiento.
- **A nivel de implementación**, evaluar un clasificador KNN en un nuevo punto de datos requiere buscar sus vecinos más cercanos en el conjunto de entrenamiento, lo que puede ser una operación costosa cuando nuestro conjunto de entrenamiento es grande. Existen varios métodos para acelerar esta búsqueda, que generalmente funcionan creando estructuras de datos basadas en el conjunto de entrenamiento. La idea general, es que parte del trabajo computacional necesario para clasificar nuevos puntos es común en todos los puntos. Por lo tanto, este trabajo se puede hacer con anticipación y luego reutilizarse, en lugar de repetirse para cada nueva instancia. De esta manera, en la fase de entrenamiento, al llamar a la función `.fit()` de sklearn, lo que se hace internamente es guardar todo el set de entrenamiento

completo mediante estructuras de datos que nos permitan minimizar los futuros cálculos de medición de distancias. Estas estructuras de datos suelen ser árboles kd (*KD tree*) o árboles de bolas (*Ball tree*). De esta manera, posteriormente a utilizar `.fit()` utilizaremos `.predict()`, y en este paso el cálculo de medición de distancias para el nuevo punto a clasificar será mucho más rápido.

### 3.3.1 Principal limitación KNN.

Una de las principales limitaciones de KNN[39] es la **complejidad computacional requerida al trabajar con datasets de gran tamaño**.

Esto ocurre debido a que, como mencionamos previamente, cada vez que se va a realizar una predicción para un nuevo punto del set de test, es necesario calcular las distancias entre este punto y todos los demás puntos del set de entrenamiento completo. Dichos cálculos llevan un gran costo computacional si se trata de un dataset de grandes dimensiones.

Además, KNN requiere almacenar todos los datos de entrenamiento para funcionar, por lo que también existe un gran costo computacional en términos de almacenamiento. Es por esto que se recomienda utilizar datasets de pocas dimensiones para que el clasificador KNN complete su ejecución rápidamente.

### 3.3.2 Funcionamiento y ejemplo de KNN.

Luego de haber entrenado / construido nuestro modelo KNN habiendo previamente determinado el valor de k y la métrica de distancia a utilizar, el funcionamiento del algoritmo KNN para la predicción o clasificación de nuevas muestras es el siguiente:

- Paso 1: Dado un nuevo punto de entrada del set de pruebas, se calculan las distancias entre este nuevo punto y todos los puntos del set de entrenamiento.
- Paso 2: Se ordenan las distancias y se determinan los K vecinos más cercanos basándose en los valores mínimos de dichas distancias.
- Paso 3: Se analiza la clase de esos vecinos y se asigna una clase para ese punto de entrada basado en el voto de la mayoría.
- Paso 4: Se retorna la clase predicha.

En la sección del Anexo *Ejemplo de funcionamiento KNN* detallamos un breve ejemplo gráfico en el cual usamos KNN para clasificación utilizando la distancia euclíadiana como métrica para calcular las distancias. Viendo este ejemplo podemos observar que KNN se basa esencialmente en un método estadístico. Cuando queremos simplemente clasificar un punto cuya clase no conocemos no hacen falta las probabilidades; podemos simplemente asignarlo a la clase con mayoría entre los k-vecinos del punto.

### 3.3.3 Métrica de distancia a emplear.

La función de distancia entre dos vectores  $x$  e  $y$  es una función  $d(x, y)$  que define la distancia entre ambos vectores como un número real no negativo. Esta función es considerada como una métrica si satisface las siguientes 4 propiedades[40]:

1. Valor no-negativo: La distancia entre  $x$  e  $y$  siempre es un valor mayor o igual a cero.

$$d(x, y) \geq 0$$

2. Identidad de los indiscernibles: La distancia entre  $x$  e  $y$  es igual a cero si y sólo si  $x$  es igual a  $y$ .

$$d(x, y) = 0, \text{ si } x = y$$

3. Simetría: La distancia entre  $x$  e  $y$  es igual a la distancia entre  $y$  y  $x$ .

$$d(x, y) = d(y, x)$$

4. Desigualdad triangular: Considerando la presencia de un tercer punto  $z$ , la distancia entre  $x$  e  $y$  es siempre menor o igual que la suma de la distancia entre  $x$  y  $z$  y la distancia entre  $y$  y  $z$ .

$$d(x, y) \leq d(x, z) + d(y, z)$$

De esta manera, puede usarse cualquier métrica para medir las distancias en KNN, siempre y cuando cumpla con las propiedades descritas anteriormente.

Dos de las distancias más utilizadas en la implementación de KNN son la distancia euclíadiana y la distancia Manhattan. En Figura 3.7 podemos observar gráficamente las diferencias entre las mismas.

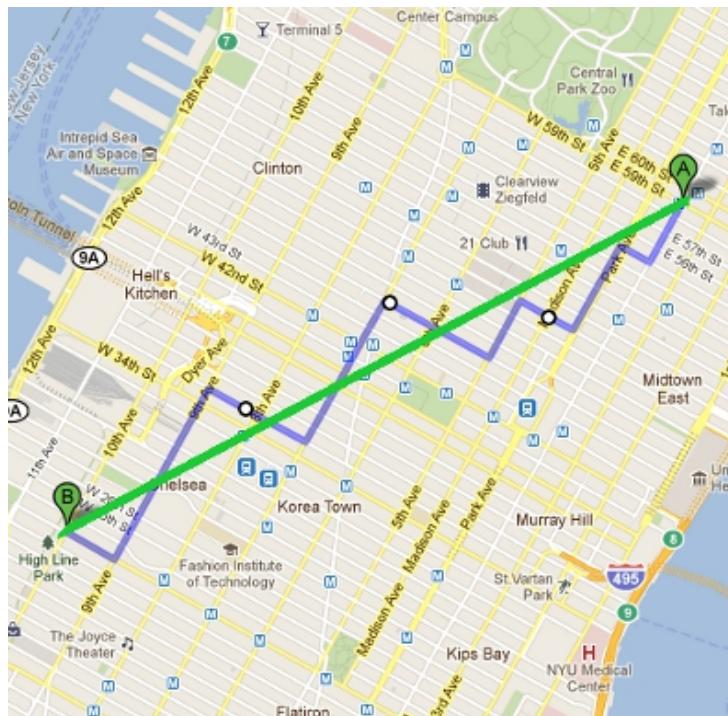


Figura 3.7: Distancia Manhattan vs Distancia Euclíadiana[10].

La **distancia euclíadiana**, o también conocida como Norma L2 o distancia de regla, es una extensión del Teorema de Pitágoras. Esta distancia representa la raíz de la suma al cuadrado de las diferencias absolutas entre los valores opuestos de los vectores. Considerando  $x$  e  $y$  como vectores a los cuales queremos calcular su distancia euclíadiana, su cálculo se obtiene mediante la Fórmula 1.

$$d(x, y) = \sqrt{\sum_{i=1}^n |x_i - y_i|^2} \quad (\text{Ver [40]}) \quad (1)$$

Donde  $x = (x_1, x_2, \dots, x_n)$ ,  $y = (y_1, y_2, \dots, y_n)$ , y  $n$  = dimensiones de los puntos.

En la Fórmula 2 y Figura 3.8 podemos observar el cálculo de la distancia euclíadiana entre  $x$  e  $y$  considerando un espacio bidimensional. Este es el cálculo que se realiza en la implementación de nuestro KNN, ya que utilizamos la distancia euclíadiana como métrica, y únicamente contaremos con 2 ejes.

$$d(x, y) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2)$$

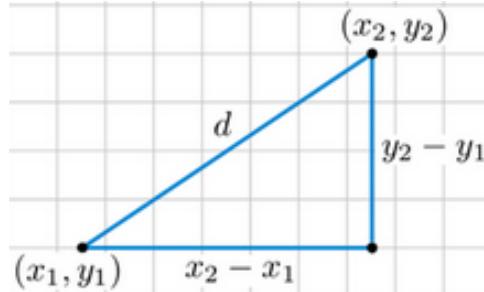


Figura 3.8: Distancia Euclíadiana en 2 dimensiones.

En cambio, la **distancia Manhattan**<sup>21</sup>, o también conocida como Norma L1 o distancia rectilínea, se calcula como la sumatoria de la diferencia absoluta entre los valores opuestos de los vectores. Considerando  $x$  e  $y$  como vectores a los cuales queremos calcular su distancia Manhattan, su cálculo se obtiene mediante la Fórmula 3.

$$d(x, y) = \sum_{i=1}^n |x_i - y_i| \quad (\text{Ver [40]}) \quad (3)$$

Donde  $x = (x_1, x_2, \dots, x_n)$ ,  $y = (y_1, y_2, \dots, y_n)$ , y  $n$  = dimensiones de los puntos.

---

<sup>21</sup>Su nombre es debido a que es la forma de calcular distancias en una ciudad con forma de grilla en la cual solo nos podemos mover por las calles en forma horizontal y vertical

### 3.3.4 Eligiendo el valor de $k$ : overfitting y underfitting.

En el caso de KNN el único hiper-parámetro que manejamos -además de la métrica usada para la distancia-, es  $k$ . La elección del valor  $k$  tendrá un impacto significativo en los resultados del algoritmo.

Para determinar el valor óptimo para  $k$  lo que se hace es probar diferentes valores de  $k$  y ver cual es el que nos da mejores resultados. Aquí hay que tener un cierto cuidado ya que hay que entender qué implica aumentar o disminuir la cantidad de vecinos más cercanos. Veamos en la Figura 3.9 qué pasa cuando usamos  $k = 1$ , es decir cuando a cada punto lo clasificamos únicamente en base al punto más cercano.

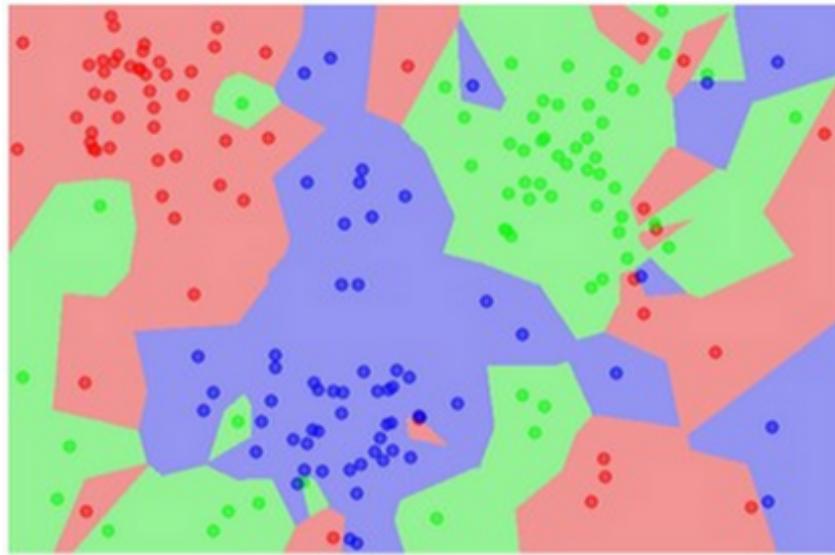


Figura 3.9: KNN con  $k = 1$ . [10]

En este caso nuestro set de entrenamiento en KNN es simplemente perfecto porque dado un registro cualquiera, el más parecido es el registro mismo y, por lo tanto, vamos a tener 100 % de precisión para el set de entrenamiento. Sin embargo, esto no quiere decir que el algoritmo generalice bien y para el set de pruebas los resultados pueden ser catastróficos.

Lo que podemos ver es que la distribución que aprende nuestro clasificador no es homogénea, es decir que hay puntos de distintas clases mezclados en zonas en las cuales predomina otra clase, esto implica que nuestro clasificador va a funcionar muy bien para el set de entrenamiento pero no aprendió a generalizar y esto quiere decir que no va a ser muy bueno para predecir la clase de puntos nuevos que no hayamos observado en el set de entrenamiento. Esta es la definición de overfitting que discutimos en la sección previa (Ver *Los Problemas de ML: Overfitting y Underfitting*): cuando un algoritmo funciona bien para el set de entrenamiento y mal para datos nuevos. El concepto es que aprender a predecir el set de entrenamiento no es lo mismo que aprender a generalizar.

Ahora veamos en la Figura 3.10 qué sucede con  $k = 5$ .

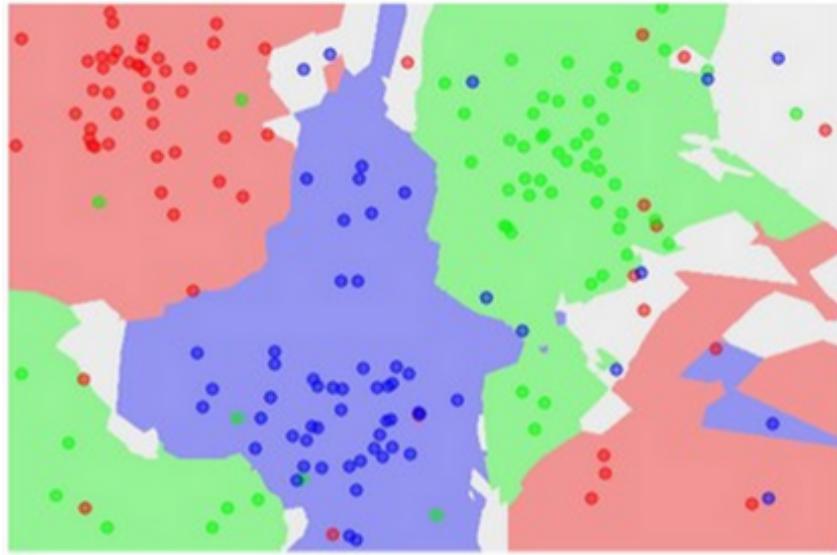


Figura 3.10: KNN con  $k = 5$ . [10]

Como podemos ver en la Figura 3.10 el clasificador aprendió a generar áreas más suaves y por consiguiente es un clasificador que generalizará mejor para predecir puntos que no estaban en el set de entrenamiento. Podriamos pensar entonces que es conveniente usar valores de  $k$  grandes como  $k = 1000$  o incluso  $k = n$  (donde  $n$  es la cantidad de puntos de datos del set de entrenamiento). Sin embargo, al aumentar el valor de  $k$  estamos dando cada vez mayor peso a las clases que tienen mayor cantidad de puntos en el set de entrenamiento, en el extremo si  $k = n$  vamos a predecir para todos los puntos la clase que mayor cantidad de puntos tiene en el set de entrenamiento lo cual no es bueno.

En resumen:

- Cuando  $k$  es muy bajo, el algoritmo toma muy pocos puntos para clasificar un punto nuevo. Es un caso de overfitting. El modelo clasifica los puntos nuevos en base a muy poca evidencia. Nuestro algoritmo alucina, las fronteras se vuelven muy complejas, demasiado complejas para el set de datos que tenemos.
- Cuando  $k$  es un número muy grande, KNN considera demasiados puntos para clasificar un punto nuevo. Este es un caso de underfitting, donde nuestro algoritmo tiene visión borrosa por estar mirando demasiados puntos, las fronteras entre nuestras clases se vuelven difusas y el poder expresivo del algoritmo es pobre.

En definitiva, el  $k$  óptimo en KNN es aquel que nos de un buen desempeño en cuanto a la precisión de clasificación para el mayor  $k$  posible. Una de las mejores formas para seleccionar nuestro  $k$  óptimo, es utilizando el método de **Cross Validation** (ó validación cruzada): ver *Cross Validation*.

### 3.4 K-means.

K-means o K-medias, o también conocido como algoritmo de Lloyd, es un algoritmo de ML de clustering (ver *Agrupación (clustering)*). Como mencionamos anteriormente, en problemas de clustering contamos con datos que queremos dividir en grupos de forma automática. Estos datos no tienen etiquetas, no sabemos a qué grupo pertenecen. K-means nos permite encontrar cómo repartir m puntos en n dimensiones dentro de k clusters.

El nombre de K-means viene porque representa cada uno de los clusters por la media de sus puntos, es decir, por su centroide. Cada clúster, por tanto, es caracterizado por su centroide que se encuentra en el centro de los elementos que componen el clúster.

Para este algoritmo, el único hiper-parámetro que se necesita definir previamente es k, el número de clusters. Este k se puede obtener de distintas formas, en nuestra implementación utilizaremos el *método del codo* para calcularlo. Antes de explicar en qué consiste este método, detallaremos el funcionamiento de k-means, su objetivo, sus desventajas y las mejoras que se pueden aplicar.

#### 3.4.1 Funcionamiento y ejemplo de K-means.

Resumidamente, habiendo definido previamente nuestro k, el número de clusters, y teniendo como entrada nuestros puntos de datos  $x_1, \dots, x_m$ , el funcionamiento de K-means consta de los siguientes 4 pasos [43]:

- Paso 1: Se inicializan aleatoriamente los centroides  $C_1, \dots, C_k$  colocándose en posiciones al azar. Cada centroide  $C_j$  pertenece a un cluster  $j$  específico.
- Paso 2: Se asignan los puntos de datos a su cluster más cercano: Para esto, lo que se hace es, para cada punto de datos  $x_i$ , se calculan las distancias euclidianas con cada uno de los centroides  $C_j$ . Luego, se asigna cada punto de datos  $x_i$  al cluster  $j$ , cuyo centroide  $C_j$  es el más cercano al punto  $x_i$  basándose en la mínima distancia euclidiana. Matemáticamente, se utiliza la Fórmula 1 para calcular la distancia euclidiana entre cada uno de nuestros puntos de datos  $x_i$ , y cada uno de nuestros centroides  $C_j$ , y obtener el mínimo valor. Reformulando dicha ecuación, nos quedaría la Fórmula 4.

$$d(x, C_j) = \sqrt{\sum_{i=1}^d |x_i - C_{ji}|^2} \quad (\text{Ver [41]}) \quad (4)$$

Donde  $d$  = dimensiones de nuestros puntos de datos,  $x_i = (x_1, x_2, \dots, x_d)$  la posición de un punto  $x$  de nuestro set de datos para la dimensión  $i$ , y  $C_{ji} = (C_{1j}, C_{2j}, \dots, C_{dj})$  la posición de nuestro centroide  $C$  del cluster  $j$  para la dimensión  $i$ .

En la Fórmula 4, podemos visualizar que los centroides tienen igual cantidad de dimensiones que nuestros puntos y, además, pueden o no coincidir con puntos de los datos. Un centroide que es también un punto de los datos se lo llama “clusteroid”. En general, no se pide que los centroides sean también puntos sino que se permite que tomen cualquier posición dentro del espacio de los datos.

- Paso 3: Se actualizan las posiciones de los  $C_j$  (centrodes  $C$  pertenecientes a los clusters  $j$ ). De esta manera, se recalcularon los centrodes como el promedio de todos sus puntos (los que pertenecen a su cluster) de forma tal de minimizar la distancia desde el centrode a los puntos asignados al mismo. Como resultado, los centrodes se mueven al centro promedio de los puntos a los que se les asignaron. Matemáticamente, se utiliza la Fórmula 5 para calcular, para cada cluster  $j = (1, \dots, k)$  la nueva posición del centrode  $C_j$ , el cual se calcula como el promedio de todos los puntos  $x_i$  asignados al cluster  $j$  en el paso anterior.

$$C_j(a) = \frac{1}{m_j} \sum_{x_i \in C_j}^{x_m} x_i(a), \text{ para } a = (1, \dots, d) \quad (5)$$

Donde  $d$  = dimensiones de nuestros puntos de datos y centrodes,  $m_j$  = cantidad de puntos de nuestro set de datos pertenecientes al cluster  $j$ ,  $x_i(a)$  = posición de nuestro punto del set de datos  $x_i$  para la dimensión  $a$ , y  $C_j(a) = (C_1(1), C_1(2), \dots, C_K(D))$  la posición de nuestro centrode  $C$  del cluster  $j$  para la dimensión  $a$ .

- Paso 4: Se repiten los pasos 2 y 3 hasta que los centrodes convergen. La convergencia puede verificarse mediante la diferencia entre los centrodes entre el paso anterior y el actual. Cuando los centrodes prácticamente ya no cambian de posición o cuando los puntos de datos no cambian de cluster se declara la **convergencia** del algoritmo.

A modo de ejemplo, en la sección del Anexo *Ejemplo de funcionamiento K-means*, detallamos un ejemplo gráfico en el cual utilizamos K-means para clusterizar nuestros datos hasta llegar a su convergencia.

### 3.4.2 Objetivo de k-means y su función de coste.

El objetivo de K-means es encontrar cómo repartir m puntos en n dimensiones dentro de k clusters de la “mejor forma posible”.

¿A qué llamamos “de la mejor forma posible”? se puede definir a cada cluster a partir de su centro (centroide), en cuyo caso la mejor distribución posible es aquella que minimiza la distancia entre cada punto del cluster y el centroide que se le asignó. Esto es lo que se conoce como **Inercia** (Inertia) o **WCSS** (within-cluster sum-of-squares, suma de cuadrados intra cluster), y es lo que busca *minimizar* K-means.

A este criterio de Inercia también se lo considera como la **función de coste J**<sup>22</sup> de K-means, la cual se intenta *minimizar*.

Dicho de otra manera, el objetivo de k-means es encontrar la ubicación de los centroides que minimicen la inercia, esta será la posición óptima de los mismos.

La Inercia[41] se puede reconocer como una medida de la coherencia interna de los clústeres, permitiendo medir qué tan bien K-means realiza la agrupación del conjunto de datos. La misma se calcula como la sumatoria de las diferencias de las distancias (generalmente euclidiana) al cuadrado entre cada uno de los puntos de datos  $x_i$  pertenecientes a un cluster  $j$  y el centroide al que fue asignado dicho punto,  $C_j$ : Fórmula 6.

$$J = \text{Inertia} = \text{WCSS} = \sum_{C_j}^{C_k} \left( \sum_{x_i \in C_j}^{x_m} \|x_i - C_j\|^2 \right) \quad (6)$$

Donde  $k$  = número de centroides ó clusters,  $m$  = cantidad de puntos de nuestro set de datos,  $C_j$  = centroide de cluster  $j$ , y  $x_i$  = punto  $i$  de nuestro set de datos asignado a  $C_j$ . Además, el término  $\|x_i - C_j\|$  se corresponde a la distancia euclidiana entre cada punto del set de datos,  $x_i$ , y el centroide que tiene asignado,  $C_j$ . Este cálculo se corresponde a la Fórmula 4 que obtuvimos anteriormente.

Considerando una inicialización aleatoria de nuestros centroides, y por ejemplo, definiendo que nuestro número de clusters es 3 ( $k$  en kmeans = 3), entonces la Fórmula 6 se podría reescribir como la Fórmula 7. Podemos observar que al variar la inicialización de nuestros centroides, las posiciones de nuestros  $C_j$  cambiarían, por lo que nuestra WSS sería distinta.

$$\text{WCSS} = \sum_{x_i \in C_1} \|x_i - C_1\|^2 + \sum_{x_i \in C_2} \|x_i - C_2\|^2 + \sum_{x_i \in C_3} \|x_i - C_3\|^2 \quad (\text{Ver [42]}) \quad (7)$$

---

<sup>22</sup>La función de coste o también llamada función de distorsión u objetivo, es una función que trata de determinar la diferencia o el error entre el valor estimado / predicho por un modelo de machine learning y el valor real, con el fin de optimizar los parámetros en dicha función y obtener un error mínimo. En el caso de k-means, nuestros valores estimados son las posiciones de los centroides -los cuales tenemos que descubrir y encontrar los más óptimos / los que minimicen nuestra J- y nuestros valores reales son nuestros puntos del cluster.

Es importante entender la Fórmula 6 para comprender el problema genérico de clustering. Se sabe que hay  $k$  centroides y que cada uno de estos centroides puede estar en cualquier punto del espacio. El objetivo de k-means es **encontrar la posición óptima para estos centroides de forma tal de minimizar la distancia total entre los puntos y los centroides que se le han asignado**. Es evidente que cada punto debe estar asignado a su centroide más cercano para minimizar la distancia por lo que el problema puede resumirse a **encontrar la posición óptima para los  $k$  centroides**.

Obtener el mínimo valor de nuestro  $WCSS$ , o que es lo mismo que encontrar el mínimo global, es un proceso muy complejo debido a la inmensa cantidad de formas en las que nuestros  $m$  puntos de datos se pueden repartir en  $k$  clusters. En lugar de esto, k-means trata de encontrar una solución que, aun no siendo la mejor de entre todas las posibles, sea buena (óptimo local) y garantice un agrupamiento en el que los clusters sean poco dispersos y se encuentren separados entre sí.

Es de destacar que, suponiendo que tenemos  $m$  puntos de datos y especificamos que este  $m$  sea nuestro número de clusters ( $k = m$ ), entonces  $WCSS$  será 0 ya que cada punto de nuestros datos actuará como centroide de sí mismo y la distancia euclíadiana entre ellos será nula. De esta forma, cada cluster contendrá solo un punto. Esto idealmente es un cluster perfecto, pero no tiene ningún sentido tener tantos clusters como puntos de datos tengamos. Por lo tanto, existe un valor de umbral para  $k$  que podemos encontrar utilizando el *método del codo*, el cual veremos más adelante en *Obtención del  $k$  mediante Elbow Method*).

### 3.4.3 Limitaciones K-means.

K-means tiene tres principales limitaciones, las cuales describiremos a continuación.

1. **Outliers:** Un outlier es una observación atípica dentro de una muestra de datos que es notablemente diferente del resto. Los outliers representan errores en la medición, mala recolección de datos, o simplemente muestran variables no consideradas al recolectar los datos. En k-means los outliers pueden incrementar la función de coste o Inercia. Varios investigadores observaron que, cuando los datos contienen outliers, existe una variación en el resultado que significa que no hay un resultado estable al realizar diferentes ejecuciones de k-means con los mismos datos[43]. Por esta razón, es muy importante eliminar los outliers de nuestro conjunto de datos. Los valores atípicos se pueden eliminar aplicando técnicas de preprocesamiento en el conjunto de datos original.
2. **Número de los  $k$  clusters:** Uno de los principales inconvenientes de K-means es la necesidad de determinar el número óptimo de los  $k$  clusters por adelantado. Esto perjudica la eficacia del algoritmo ya que en la práctica, no se conoce a priori el número de clusters final. Este defecto lo perjudica al compararlo con otros algoritmos, ya que en muchos la inicialización del número de clusters no es necesaria. Todavía es un problema saber cuál es el número correcto de  $k$  clusters que debemos asignar[43].
3. **Inicialización de los centroides:** La selección inicial de los centroides dirige el proceso de K-means y las particiones resultantes y su efectividad están condicionadas a la elección de estos centroides[41]. Esto lo podemos ver en mayor detalle en la sección del Anexo *Posición inicial de los centroides en K-means*. Existen numerosos estudios[44] que muestran que la performance de K-means es fuertemente dependiente de la estrategia de inicialización de la ubicación de los centroides a utilizar.

Todas estas limitaciones descritas anteriormente fueron abordadas y resueltas en la implementación de k-means realizada para este proyecto de la siguiente forma:

- Para el primer problema, los outliers, se aplicó un pre-procesamiento de nuestros datos que permitió removerlos: ver *Armado del modelo de clasificación KNN*.
- Para el problema de la determinación óptima y por adelantado del número de los  $k$  clusters, se aplicó el *método del codo*: ver *Obtención del  $k$  mediante Elbow Method*.
- Para el último caso de inicialización de los centroides, como se menciona en *Posición inicial de los centroides en K-means*, una de las soluciones para no caer en un mínimo local malo es realizar varias ejecuciones de K-means con inicializaciones aleatorias para los centroides, computar la función de distorsión  $J$  para cada resultado final y quedarnos con el mínimo.

En la implementación de esta Tesis, se utilizó una estrategia más novedosa, inicializar nuestros centroides utilizando K-means++: ver *Inicialización de los centroides: k-means++*. La ventaja de utilizar esta estrategia es que generalmente funciona mejor y más rápido que la detallada anteriormente, por lo que actualmente es uno de los métodos estándar para inicializar nuestros centroides en k-means. [45]

### 3.4.3.1 Obtención del $k$ mediante Elbow Method.

Como se mencionó anteriormente, es necesario encontrar el valor óptimo de  $k$ . Un buen modelo k-means es uno que tenga la menor Inertia y un bajo número de clusters  $k$ . Sin embargo esto no es posible, ya que si  $k$  aumenta, Inertia decrece.

*Elbow method o método del codo* nos permite determinar el óptimo valor de  $k$ , o dicho de otra forma, el óptimo número de clusters en los cuales nuestros puntos de datos serán clusterizados. Mediante este método lo que se hace es graficar cómo varía la función de costo o Inercia en función de  $k$ : recordemos que la Inercia se calcula mediante la Fórmula 6.

Luego de obtener este gráfico, se encuentra el punto “del codo”, el cual indicará nuestro  $k$  óptimo. Este es el punto después del cual la Inercia comienza a disminuir de forma lineal, o dicho de otra forma, el punto “del codo” indica que después de este punto, el cambio en la disminución del valor de la Inercia no es significativo.

Veamos el siguiente ejemplo donde obtendremos nuestro  $k$  óptimo en base al *método del codo*: para los puntos de datos observados en la Figura 3.11, podemos observar visualmente 3 clusters separados, por lo que el número óptimo de clusters debería ser 3. En este caso es bastante sencillo observar esto, pero hay muchos casos donde no es fácil visualizar a simple vista el número de  $k$  óptimo, y es por esto que el *método del codo* es de gran utilidad.

Utilizando esta técnica, graficamos la Inercia en función de  $k$ : iteramos valores de  $k$  desde 1 hasta 9 y calculamos los valores de inercia para cada valor de  $k$  dentro de ese rango, obteniendo el gráfico de la Figura 3.12.

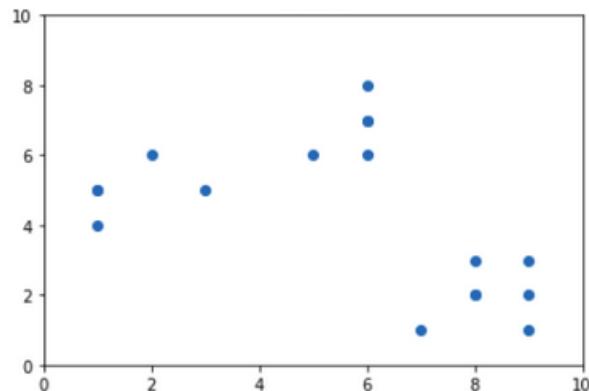


Figura 3.11: Puntos de datos.

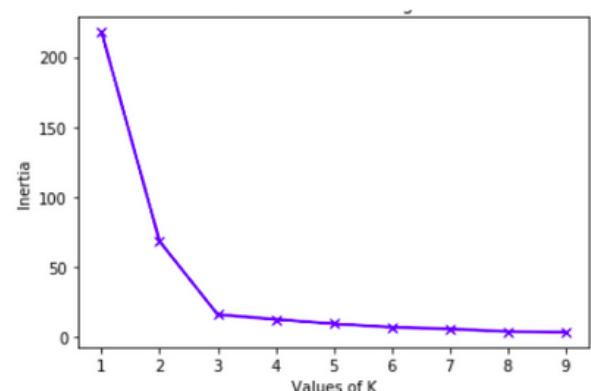


Figura 3.12: Método del codo usando Inercia.

De esta manera, concluimos que, para estos datos, viendo el gráfico de la Figura 3.12, el punto “del codo” está en  $k = 3$ , siendo 3 nuestro  $k$  óptimo.

En la Figura 3.13 podemos visualizar los puntos de datos clusterizados para diferentes valores de  $k$ , observando de esta manera que con  $k = 3$  obtenemos la clusterización más óptima y lógica.

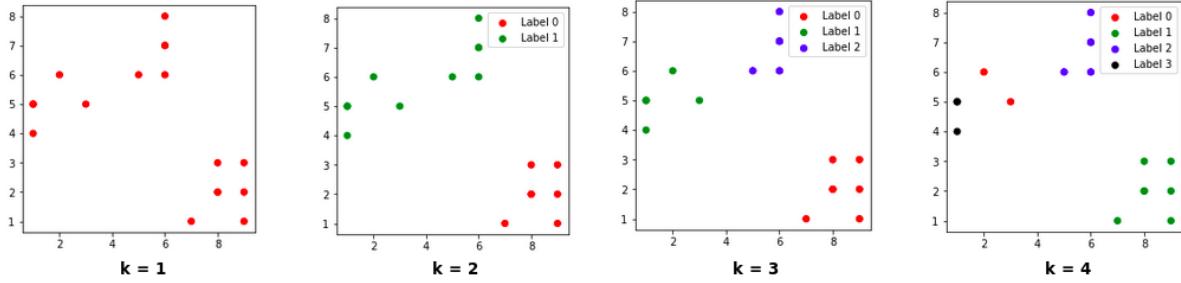


Figura 3.13: Puntos de datos clusterizados para diferentes valores de  $k$ .

### 3.4.3.2 Inicialización de los centroides: k-means++.

K-Means++[45] es una variante de K-Means en donde lo único que cambia con respecto a este es la forma en que se inicializan los centroides (el paso 1 de k-means descrito en la sección *Funcionamiento y ejemplo de K-means*). Actualmente es uno de los métodos estándar para inicializar los centroides en k-means.

De esta manera, habiendo predefinido nuestro  $k$ , los pasos para inicializar nuestros centroides con k-means++ son:

1. Se elige un punto al azar como primer centroide. Este paso es el mismo que ocurre en k-means, solo que en este caso únicamente el primer centroide será elegido al azar.
2. Calculamos la distancia  $D(x)$  entre cada punto de nuestros datos  $x$  y su centroide.
3. Uno de nuestros puntos de datos  $x$  será elegido como el nuevo centroide. Este nuevo centroide será el que tenga la mayor distancia al cuadrado  $D(x)^2$  con respecto a su centroide, es decir, el punto  $x$  que tenga la distancia más lejana con su centroide.
4. Repetimos los pasos 2 y 3 hasta que nuestros  $k$  centroides sean asignados.

La idea de K-means++ es asignar los centroides de forma espaciada, de esta forma el óptimo local obtenido por K-Means tiene una mayor probabilidad de estar cerca del óptimo global.

En la secuencia de la Figura 3.14 podemos observar un ejemplo de asignación de centroides por medio de k-means ++, habiendo previamente definido  $k = 3$ . Observamos que, a partir de la muestra de datos (1), obtendremos la inicialización de nuestros centroides (6).

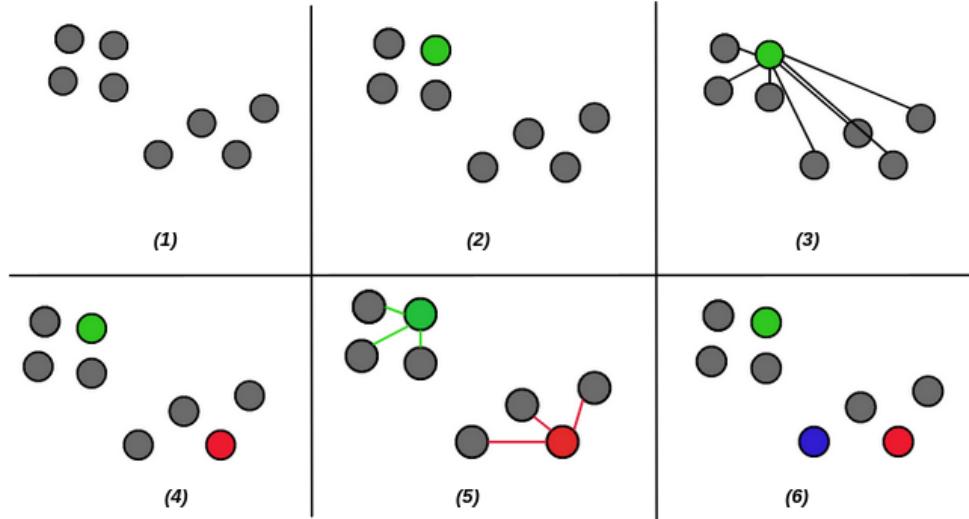


Figura 3.14: Inicialización de centroides mediante K-means.

En (2) se elige aleatoriamente un punto al azar como centroide inicial (en este caso el punto verde). En (3) se calcula la distancia  $D(x)$  entre cada uno de los puntos  $x$  y su centroide. Para este caso únicamente tenemos el centroide verde. En (4) se encuentra al segundo centroide (rojo) de manera que esté alejado del centroide verde. Este segundo centroide es el punto  $x$  que tiene la mayor distancia al cuadrado  $D(x)^2$  a su centroide. En (5) se calculan nuevamente las distancias  $D(x)$  entre cada uno de los puntos y su centroide más cercano. Por último, en (6) finalizamos la inicialización de nuestros centroides encontrando al último y tercer centroide (azul). El mismo fue elegido ya que su punto  $x$  es el que tiene la mayor distancia al cuadrado  $D(x)^2$  a su centroide con respecto a cualquier otro punto y su centroide.

De esta manera, concluimos que K-means++ genera una inicialización más dispersa de los centroides iniciales comparándolo con el método de K-means estándar de inicialización aleatoria. El problema de K-Means++ es que para datos realmente masivos necesita hacer  $k$  iteraciones sobre los datos para elegir los centroides, tarea que es todavía más ineficiente cuando  $k$  es un número largo. Sin embargo, en general, K-Means++ genera una inicialización mejor para K-Means y esto permite no solo obtener una mejor solución final sino que también acelera la velocidad de convergencia del algoritmo.

## 3.5 Redes Neuronales Artificiales (ANN).

Las redes neuronales artificiales (o artificial neural network, ANN) son técnicas de Machine Learning que permiten imitar o reproducir la capacidad de aprender propia del comportamiento de un cerebro humano. Las ANN están compuestas por un conjunto de elementos simples (neuronas artificiales) que se interconectan masivamente en paralelo y con organización jerárquica[46].

En esta sección detallaremos el funcionamiento básico y los componentes de una red neuronal; la cual será utilizada como base en la implementación del modelo Word2vec para obtener word embeddings: Ver *Word embeddings (Word2vec) & WMD*.

Existen muchísimos modelos de redes neuronales, los cuales pueden ser entrenados para resolver distintas tareas, tanto de clasificación o de regresión, como de clustering. En esta sección se explicarán dos de ellos, el modelo más básico y simple de una red neuronal, conocido como *Perceptrón simple*, y su modelo predecesor, *Perceptrón Multicapa* (o también conocido como Multi Layer Perceptron, *MLP*). Ambos modelos son considerados como algoritmos de clasificación, formando parte de la rama de aprendizaje supervisado.

### 3.5.1 Perceptrón simple.

El Perceptrón Simple es un algoritmo de clasificación lineal que surge a fines de la década del 50 y fue desarrollado por Frank Rosenblatt. En aquel tiempo se buscaba crear un algoritmo que imitase el funcionamiento del cerebro humano. El perceptrón puede verse biológicamente como una neurona, siendo la unidad básica de procesamiento que se encuentra en una red neuronal. La teoría era que un perceptrón (o neurona) podía programarse como una salida binaria que dependía del resultado de la combinación lineal entre las entradas y pesos asignados a las mismas. De esta forma, la neurona respondía de manera binaria (0 ó 1) según si el resultado de la combinación lineal fuese mayor o menor a un cierto umbral mediante el uso de una “función de activación”[10], la cual detallaremos más adelante.

En resumen, esta neurona (o perceptrón) tiene conexiones con nodos de entrada, a partir de los cuales recibe estímulos externos (valores de entrada). Con estos valores la neurona realiza un procesamiento interno y genera un valor de salida, decidiendo de esta manera a cuál de las dos clases posibles pertenece la observación[46]. Este procesamiento depende de las distintas funciones / cálculos matemáticos llevados a cabo mediante la *función de entrada* y la *función de activación* de la neurona (o también llamada *función de transferencia*)[47]. Podemos observar un diagrama de la arquitectura del Perceptrón Simple en la Figura 3.15.

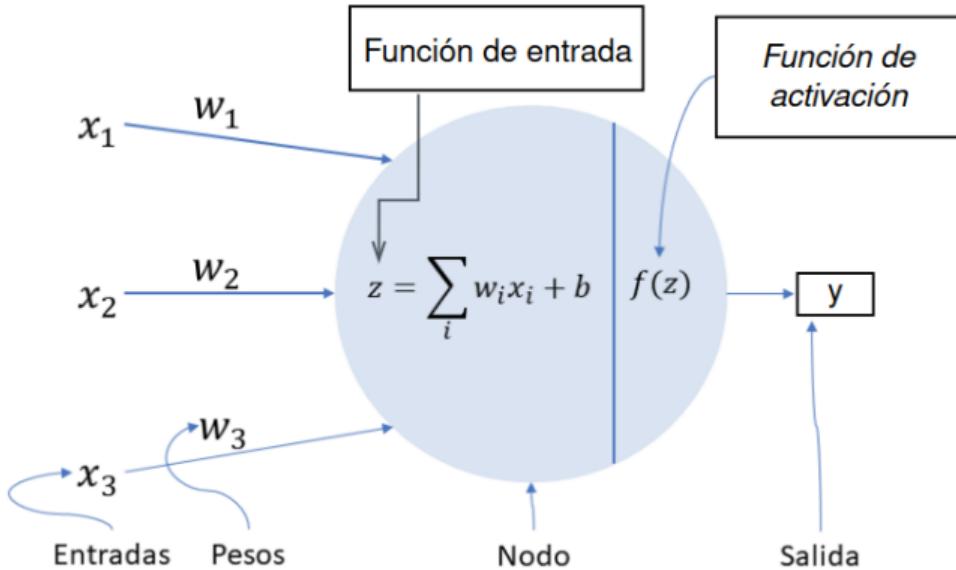


Figura 3.15: Modelo Perceptrón Simple

Mencionaremos las partes involucradas en dicho esquema:

- **Entradas** ( $x_1, x_2, x_3, x_n$ ): Las *entradas* son los valores de entrada a la neurona que provienen de estímulos externos.
- **Pesos** ( $w_1, w_2, w_3, w_n$ ): Los *pesos* son los coeficientes que determinan la intensidad o fuerza de la conexión de las señales de entrada registradas por la neurona. Si el peso de una entrada  $x_1$  es mayor al peso de la entrada  $x_2$ , entonces la entrada  $x_1$  tiene mayor influencia en la neurona. Si el peso es positivo se habla de una *excitación* de la entrada, en cambio si el peso es negativo se habla de una *inhibición* de la misma.
- **Función de entrada:** En esta parte del esquema se aplica una función de entrada a nuestros valores de entrada, sus pesos y el bías -representado en la figura como  $b$ <sup>23</sup>-.

Existen distintas funciones de entrada, tales como la sumatoria de las entradas ponderadas, la productoria de las entradas ponderadas o el máximo de las entradas ponderadas[48]. En el ejemplo descrito en la figura 3.15 se utilizó la función de entrada más común: la *sumatoria de las entradas ponderadas*, que representa la suma de todos los valores de entrada a la neurona, multiplicados por sus correspondientes pesos. De esta manera, mediante la Fórmula 8 obtenemos la salida  $Z$ .

$$Z = \sum_i w_i x_i + b \quad (8)$$

Donde  $x_i$  es cada una de las entradas externas a la neurona/perceptrón,  $w_i$  son los pesos asociados a cada entrada y  $b$  es el sesgo descrito anteriormente.

---

<sup>23</sup>El bias o sesgo es el término independiente que permite cambiar o disparar la función de activación para garantizar un aprendizaje exitoso. En otras palabras, permite controlar qué tan predisposta está la neurona a disparar un 1 o un 0 independientemente de los pesos. Un sesgo alto hace que la neurona requiera una entrada más alta para generar una salida de 1. Un sesgo bajo lo hace más sencillo.

- **Función de activación  $f(Z)$ :**

Las neuronas artificiales tienen diferentes estados de activación, algunas de ellas solamente dos -activa (excitada) o inactiva (no excitada)-, pero otras pueden tomar cualquier valor dentro de un conjunto determinado.

La función activación calcula dichos estados de activación de la neurona, transformando la entrada global  $z$  en un valor (estado) de activación o salida  $y$ , cuyo rango normalmente va de (0 a 1) o de (-1 a 1). Esto es así, porque una neurona puede estar totalmente inactiva (0 o -1) o activa (1)[48].

Resumiendo, la función de activación tiene como objetivo determinar si la neurona es activada o no aplicando una *función no lineal*  $f$  a  $Z$ , obteniendo así la salida de la neurona  $y$ : ver la Fórmula 9. ¿Por qué esta función de activación tiene que ser una *función no lineal*? Esto lo explicaremos en la sección *Funciones de activación en MLP*.

$$y = f(z) = \begin{cases} 1, & \text{si } z > \text{umbral} \\ 0, & \text{si } z \leq \text{umbral} \end{cases} \quad (9)$$

Por ejemplo, viendo la Figura 3.15, si utilizamos la *función escalón* (o *step*) como nuestra función de activación, entonces nuestro umbral sería = 0 y nuestra salida se podría representar mediante la Fórmula 10.

$$y = f(z) = \begin{cases} 1, & \text{si } \sum_i w_i x_i + b > \text{umbral} \\ 0, & \text{si } \sum_i w_i x_i + b \leq \text{umbral} \end{cases} \quad (10)$$

De esta manera si es 1 se activa la neurona, y si es 0 no se activa. Observemos que este resultado depende de los pesos y las entradas a la neurona, si variamos los mismos nuestras salidas seguramente también lo harán.

Se pueden aplicar otras funciones de activación, las cuales detallaremos en el Anexo: ver sección *Funciones de activación*.

- **Salida  $y$ :** son los valores que permiten retornar los resultados procesados por la red neuronal (en este caso formada por un único perceptrón).

### 3.5.2 Desventaja del Perceptrón simple.

La desventaja de este esquema es que el Perceptrón Simple es un algoritmo que encuentra un hiperplano separador y no más que esto, por lo que dista bastante de imitar la capacidad de aprender propia de un cerebro humano.

Observando la Figura 3.16, podemos ver que la neurona de tipo perceptrón simple con una función de activación lineal (función identidad) únicamente permite discriminar entre dos clases linealmente separables mediante una única recta o hiperplano (dependiendo del número de entradas), permitiendo solucionar por ejemplo la función OR (que es linealmente separable). Sin embargo, un Perceptrón simple no permite solucionar funciones no lineales, como OR-exclusiva (o XOR), debido a que en esta función no existe ninguna recta que separe los patrones de una clase de los de la otra. Para esto es necesario

que se introduzca una capa intermedia (o capa oculta, “hidden layer”) compuesta por dos neuronas que determinen dos rectas en el plano (parte derecha de la Figura 3.16)[49]).

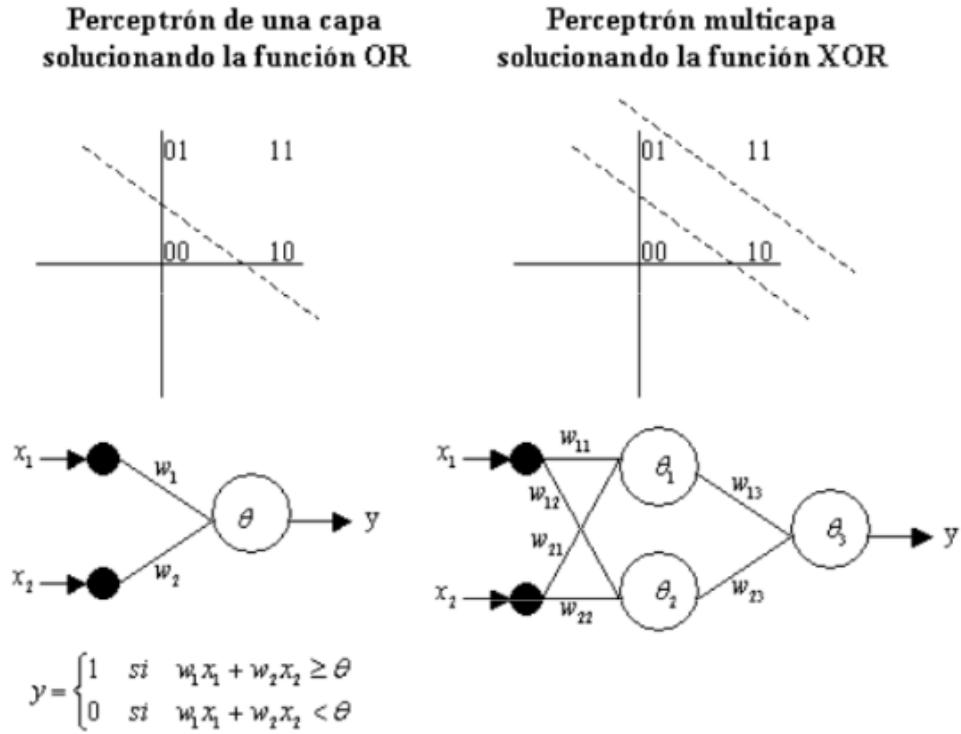


Figura 3.16: Perceptrones solucionando la función OR y XOR[49].<sup>24</sup>

Esta limitación al utilizar una única neurona es una de las principales razones por la cual surgió el esquema de Perceptrón multicapa, el cual se detalla en la siguiente sección.

### 3.5.3 Perceptrón multicapa (MLP).

El modelo de Perceptrón multicapa (MLP) es el modelo más conocido y utilizado de Redes Neuronales. Al igual que el modelo de Perceptrón Simple, MLP es un modelo de clasificación. Generalmente el modelo MLP se utiliza para describir el funcionamiento de una red neuronal, y es el esquema que utilizaremos para nuestra implementación del modelo Word2vec. En el modelo MLP la distribución de neuronas dentro de la red se realiza formando niveles o capas, con un número determinado de dichas neuronas en cada una de ellas.

A continuación mencionaremos las tres capas existentes en el modelo MLP, cada una con distintas funciones.

---

<sup>24</sup>Desde el punto de vista lógico la salida de la función OR es verdadera si algunos de los argumentos  $x_1$  y  $x_2$  son verdadero, y es falsa si todos los argumentos son falsos: entonces, la combinación  $x_1 = 0$  y  $x_2 = 0$  (en el gráfico de la izquierda representado por 00) da una salida falsa; y las combinaciones 01, 10, 11 verdadera. Lo que hacemos mediante el Perceptrón de una capa es separar estos resultados. En cambio, la salida de la función XOR es verdadera si las entradas no son iguales, y falso si son iguales; de esta manera 01 y 10 dan una salida verdadera, y 11 y 00 falsa. En este caso el Perceptrón multicapa puede separar estos dos resultados.

- **Capa de entrada:** contiene las variables de entrada; reciben directamente la información proveniente de las fuentes externas de la red.
- **Capas ocultas:** pueden haber 1 o N capa/s oculta/s. La longitud de N determina la profundidad de la red, dando origen al término de aprendizaje profundo (o “deep learning”). Colocando neuronas en forma secuencial permite que cada neurona reciba información procesada por una neurona anterior, lo que significa que la red puede generar conocimiento jerarquizado: de esta manera la red podría aprender conocimiento básico en las primeras capas y mientras va pasando la información a las neuronas de las siguientes capas, va generando conocimiento más abstracto y complejo.
- **Capa de salida:** permite retornar los resultados procesados por la red neuronal. Si estamos resolviendo un problema de clasificación esta capa puede estar formada por 2 neuronas (si se trata de una clasificación binaria) o N neuronas (en caso de clasificación multiclas).

Cabe destacar que en nuestro modelo MLP, una única neurona de nuestra capa oculta o de nuestra capa de salida puede representarse como un Perceptrón Simple (Figura 3.17). En cambio, una neurona de nuestra capa de entrada NO puede representarse como un Perceptrón simple, ya que en esta capa no existe ningún procesamiento interno. Con “procesamiento interno” nos referimos a la función de entrada y función de activación dentro de la neurona: por esta razón, las funciones de activación también residen en las neuronas de la capa oculta y/o en las neuronas de la capa de salida (las neuronas de la capa de entrada no poseen funciones de activación).

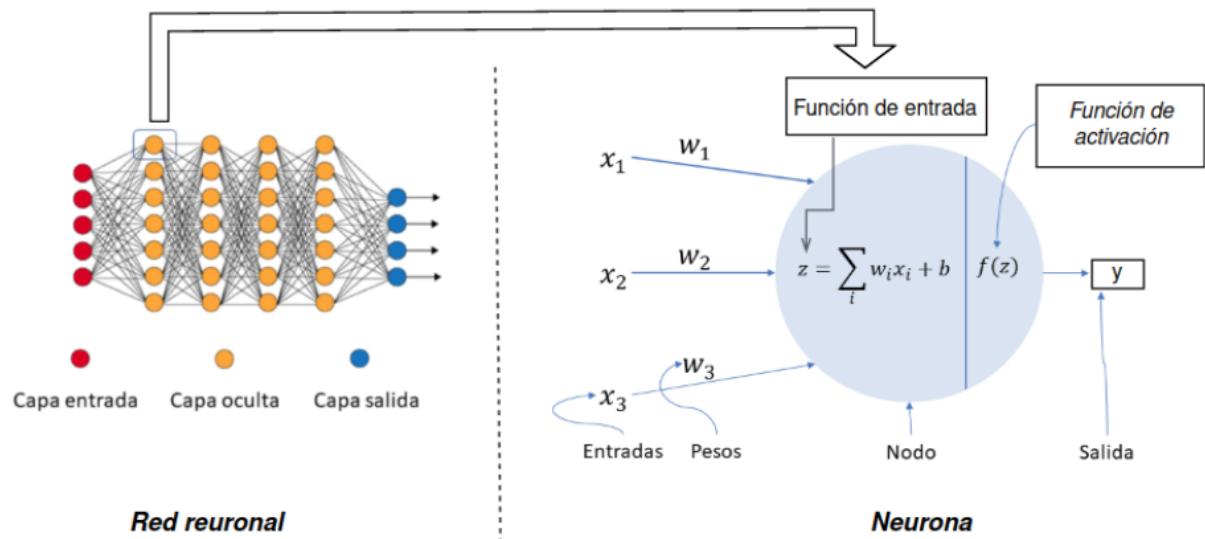


Figura 3.17: Modelo Perceptrón Multicapa -MLP- (izquierda) y Perceptrón Simple (derecha).

Observando la parte izquierda de la Figura 3.17, podemos apreciar que cada una de las capas pueden estar compuesta por una o más neuronas, lo que significa que cada capa es un vector de neuronas. La dimensión del vector determina el ancho del modelo de una red neuronal. Cada capa recibe los datos de salida de la capa anterior. Luego, cada

neurona de la capa actual realizará sus procesamientos en base a esta entrada (aplicando su función de entrada y función de activación), y su salida la pasará a cada neurona de la capa siguiente. Este tipo de redes se denominan *feedforward*, debido a que el flujo de información va de una capa a la que sigue, sin bucles de retroalimentación.

A continuación, explicaremos el modelo MLP desde un punto de vista matemático[50]. Como mencionamos previamente, con el modelo de MLP es posible combinar varios perceptrones en una capa de procesamiento, con lo cual obtenemos una función  $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$  para cada capa (oculta o de salida), donde  $m$  es el tamaño de la entrada, y  $n$  es el número de perceptrones, que a su vez determinan el tamaño de la salida. De esta manera, dada una entrada  $x$ , la función  $f$  procesa dicha entrada mediante la Formula 11.

$$f(x) = \phi(Wx + b) \quad (\text{Ver [50]}) \quad (11)$$

Donde  $\phi$  es la función de activación que elegimos usar para esa capa,  $W$  es la matriz de pesos, y  $b$  es el vector que contiene los sesgos.

Además, como mencionamos previamente, con el modelo MLP podemos componer varias de estas capas, con lo cual obtenemos una *red*, donde los perceptrones de una capa reciben como entrada a los de la capa anterior y alimentan a los de la capa que sigue. Dada una red  $f$  y una entrada  $x$ , a la salida  $f(x)$  se la denomina, naturalmente, capa de salida, mientras que a la entrada  $x$  se la suele denominar capa de entrada. Cuando tenemos una red con varias capas, todas las capas que no son de salida ni de entrada se denominan capas ocultas.

Anteriormente vimos que la red  $f$  depende de los parámetros  $W$  y  $b$ , que en su conjunto vamos a denotar con  $\theta$ , y estará formada por una *composición* sucesiva de varias capas  $f^i$ , donde el superíndice indica el número de la capa. De esta manera, cada capa cuenta con sus propios parámetros  $\theta$  y procesa la información que recibe de acuerdo a la Fórmula 12.

$$f^i(a^{i-1}, \theta^i) = f^i(a^{i-1}, w^i, b^i) = \phi(W^i a^{i-1} + b^i) \quad (\text{Ver [50]}) \quad (12)$$

Donde  $a^{i-1}$  es la salida, o activación, de la capa anterior,  $W^i$  es la matriz de pesos de la capa  $i$  y  $b^i$  es el vector de sesgos.

### 3.5.4 Funciones de activación en MLP.

Como mencionamos previamente, las funciones de activación residen en las neuronas de la capa oculta y/o en las neuronas de la capa de salida (las neuronas de la capa de entrada no poseen funciones de activación).

Algo que vale la pena mencionar es que todas las capas ocultas suelen utilizar la misma función de activación. La capa de salida normalmente utiliza una función de activación diferente de las capas ocultas y depende del tipo de predicción requerida por el modelo.

El propósito de la función de activación es introducir una *no-linealidad* dentro de la red neuronal, lo que le permite a la misma obtener una variable de salida  $y$  que varíe de forma no lineal con respecto a sus variables de entrada  $x$ . *¿Por qué esta función de activación tiene que ser una función no lineal?*:

- Esto es debido a que si una red neuronal únicamente posee funciones de activación lineales (también conocidas como *función identidad*), la red se comportará como si fuera un perceptrón de una sola capa: esto es debido a que si sumamos todas las capas obtendremos otra función lineal. Independientemente de cuán compleja sea la arquitectura de la red, una función de activación lineal solo nos permite modelar una relación lineal entre las entradas y salidas, siendo efectiva solo en una capa de profundidad y para problemas de regresión lineal (por ejemplo al predecir precios de la vivienda), y no siendo útil para la mayoría de las aplicaciones del mundo real que son altamente no lineales. Resumidamente, que la función de activación sea no lineal permite a la red apilar múltiples capas de neuronas para crear lo que se conoce como una red neuronal profunda, que se requiere para aprender conjuntos de datos complejos.

En nuestra implementación de red neuronal para el modelo Word2vec (ver sección *Word embeddings (Word2vec) & WMD*). no utilizamos funciones de activación en las neuronas de la capa oculta, este modelo *únicamente utiliza la función de activación para la capa de salida*. El modelo Word2vec, en su esquema básico, utiliza una función de activación softmax para la capa de salida. En nuestra implementación se realizó una mejora a dicho esquema, introduciendo lo que se conoce como *muestreo negativo* (el cual está explicado en la sección *Optimizaciones: Muestreo Negativo*), que implementa una función de activación sigmoide para la capa de salida. En las siguientes secciones se detallarán dichas funciones de activación.

#### 3.5.4.1 Softmax en la capa de salida.

La función Softmax comprime las salidas de cada neurona para que estas sean de 0 y 1 de tal forma que la suma de las salidas sea igual a 1. La función Softmax produce salidas que se asemejan a probabilidades. Esto es, cada salida de la red produce una probabilidad entre 0 y 1, mientras que la suma de estas probabilidades es 1. Consecuentemente, esta función es usada en la capa de salida en ANN para resolver problemas de clasificación, ya que en estos problemas se requiere separar los datos de entrada en grupos. En la Fórmula 13 podemos observar la representación matemática de la función softmax; mientras que en la Figura 3.18. podemos observar la aplicación de esta función a la capa de salida de una red neuronal.

$$Z_k = \frac{e^{y_k}}{\sum_{k=1}^M e^{y_k}} \quad (\text{Ver [49]}) \quad (13)$$

*Función softmax.*

Donde  $0 \leq Z_k \leq 1$ , mientras que  $y_k$  es la salida de la neurona  $k$ , la cual pasa a la función softmax para obtener la salida final  $Z_k$ , y  $M$  es la cantidad total de neuronas en la capa de salida.

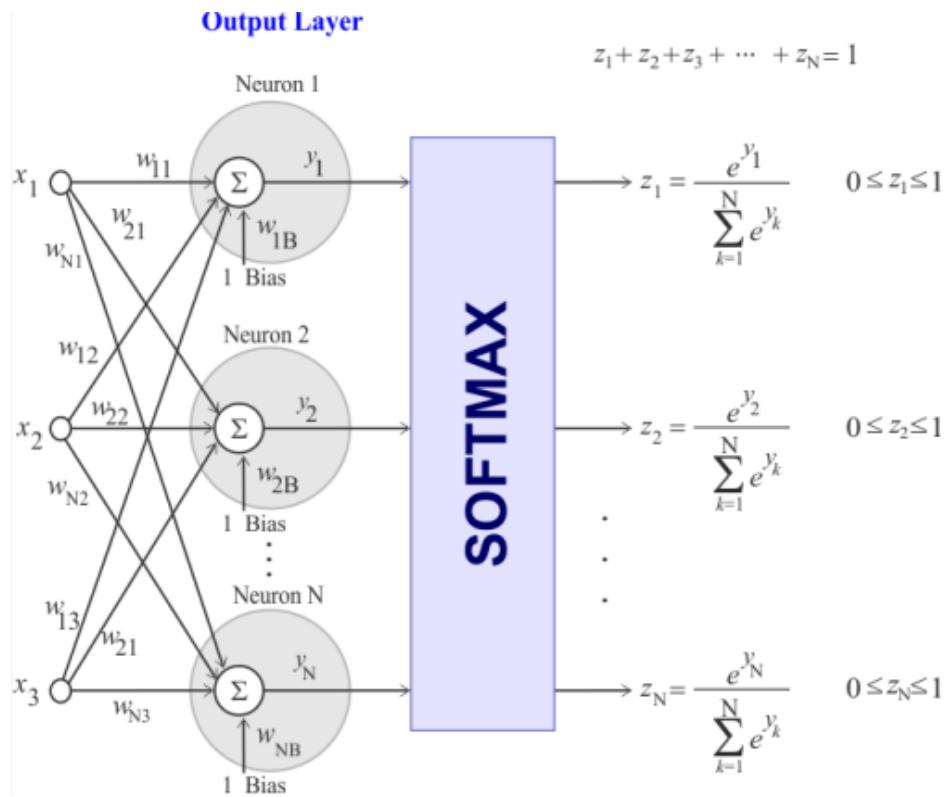


Figura 3.18: Función Softmax aplicada a la capa de salida<sup>25</sup>.

### 3.5.4.2 Sigmoide en la capa de salida.

La función sigmoide, o también llamada curva logística, es una aproximación suave de la función escalón. Como resultado de aplicar la función sigmoide a la salida  $y_k$  de cada neuronal, al igual que en la Softmax, como salida obtenemos un valor  $Z_k$  entre 0 y 1. Pero, a diferencia de la Softmax, la suma de las salidas (de  $Z_1$  a  $Z_M$ ) no dan 1 como resultado, ya que en el denominador no se consideran a todas las salidas  $y_K$  para realizar el cálculo de la función. Utilizando la función de activación sigmoide simplemente se aplica una función  $\sigma(y_k)$  por cada  $y_k$  salida de la respectiva neurona. Matemáticamente podemos observar la función sigmoide en la Fórmula 14. Mientras que en la Figura 3.19 podemos observar la representación gráfica de la función sigmoide comparándola con la función escalón.

$$\sigma(y_k) = \frac{\exp(y_k)}{1 + \exp(y_k)} = \frac{1}{1 + \exp(-y_k)} \quad (\text{Ver [50]}) \quad (14)$$

*Función sigmoide.*

Donde  $0 \geq \sigma(y_k) \geq 1$ .

---

<sup>25</sup>Obtenida del sitio web <http://sintesis.ugto.mx/WintemplaWeb>. (Consultado el 30 de abril de 2022).

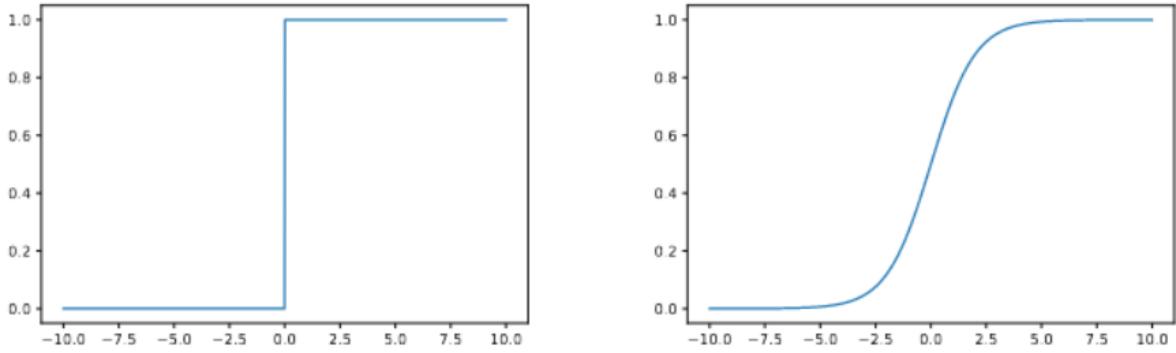


Figura 3.19: Funciones de activación escalón y sigmoide[50].

### 3.5.5 Entrenamiento en una red neuronal.

Como se comentó anteriormente, el modelo MLP de Redes Neuronales es un modelo de clasificación, lo que significa que realiza un aprendizaje supervisado. De esta manera, a la red neuronal durante el proceso de entrenamiento se le muestra un gran número de datos de entrenamiento para que la red ajuste / modifique sus pesos de manera iterativa y de este modo, lograr minimizar el error producido en la respuesta: lograr minimizar la *función de costo*, la cual penaliza de alguna manera los errores que comete la red al clasificar. Luego de varias iteraciones de ejemplos, se espera que la red neuronal sea efectiva para resolver la tarea para la que fue entrenada. En otras palabras, una red aprende resolviendo un problema de optimización, donde se buscan los parámetros de la red (pesos y sesgos,  $\theta$ ) que minimizan la función de costo o error  $J$  para cada ejemplo  $x_i$  en el conjunto de entrenamiento  $X$ .

A continuación describiremos en mayor detalle el ciclo iterativo de procesos del entrenamiento de una red neuronal[50].

1. Al comienzo del algoritmo, por única vez, se inicializan las matrices de pesos  $W$  y los vectores de bias  $b$  de toda la red con valores aleatorios. Estos son los *parámetros* de nuestro modelo de red neuronal.
2. Se aplica *forward propagation* (o propagación hacia adelante) para cada una de las muestras del set de entrenamiento y se obtienen los puntajes de salida  $y$ . Forward propagation es la manera en que una ANN computa los valores de entrada y los clasifica. Matemáticamente, durante este proceso la red neuronal recibe un vector de entrada  $x$  y devuelve un vector de salida  $y$ <sup>26</sup> aplicando en cada una de las capas de la red la Fórmula 12 (que alimentará a la capa  $f^{i+1}$ , y que generalmente involucra sumatoria, multiplicación y uso de funciones de activación. De esta manera la señal se mueve desde la capa de entrada, pasando por las capas ocultas intermedias, hasta llegar a la capa de salida con un valor.
3. Los vectores de salida  $y$  obtenidos / predichos de todas las muestras de entrenamiento se comparan con los valores de sus vectores de salida  $y$

---

<sup>26</sup>Cada elemento del vector de salida  $y$  se corresponde a la salida de una  $k$  neurona de la capa de salida.

correspondientes que tiene las etiquetas reales, computándose de esta manera el *error* (o *loss*) de clasificación (entre el valor predicho y el real) mediante el cálculo de la *función de costo / pérdida (loss function)*.

La función de pérdida  $J(\theta)$  será diferente para cada modelo. Más adelante obtendremos los cálculos de las  $J(\theta)$  que tendremos en consideración para nuestra implementación, las cuales son la Fórmula 22 y la Fórmula 28, siendo esta última la que finalmente utilizaremos en nuestro modelo y buscaremos minimizar. Ambas funciones de costo son la sumatoria sobre los costos individuales en los que incurre la red al clasificar cada elemento del conjunto de entrenamiento, y se calculan 1 vez por epoch<sup>27</sup>.

4. Se aplica el proceso conocido como *backpropagation* (o propagación hacia atrás). No se explicará en detalle este procedimiento aquí porque no entra dentro de los objetivos de este trabajo. Lo que hay que tener en cuenta es que backpropagation utiliza la función de optimización<sup>28</sup> *descenso del gradiente* para poder ajustar / modificar los pesos y el bias de cada neurona de nuestra red *con el objetivo de minimizar la función de costo  $J(\theta)$  obtenida en el paso 3*, y con ello minimizar el error de clasificación. En backpropagation calculamos el gradiente de  $J(\theta)$  (o dicho de otra forma derivamos el error) en función de cada uno de los parámetros de nuestra red. Para esto se opera recursivamente capa tras capa propagando el error desde la salida hacia la entrada de la red.

De esta manera, considerando que nuestra función de pérdida es  $J(\theta)$ , donde nuestros parámetros  $\theta$  son  $W$  (matriz de pesos) y  $b$  (vector de bias), el objetivo de nuestro modelo será encontrar los valores óptimos de  $W$  y  $b$  ( $\theta^{(new)}$ ) para minimizar  $J(\theta)$ . De esta manera, en la ecuación 15 podemos observar la ecuación general de actualización de nuestros parámetros  $\theta$  para la red reuronal.

$$\theta^{(new)} = \theta^{(old)} - n \cdot \nabla J(\theta) \quad (\text{Ver [50]}) \quad (15)$$

*Fórmula de actualización de parámetros de nuestra ANN.*

Donde  $n$  es la tasa de aprendizaje (o learning rate), del cual hablaremos con mayor detalle en la sección *Hiper-parámetro n*, y  $\nabla J(\theta)$  es el gradiente (también llamado pendiente o derivada) de la función  $J(\theta)$  para nuestros parámetros  $\theta^{(old)}$ . Al tener un “-” por delante obtenemos el gradiente negativo, el cual apunta en la dirección de máximo decrecimiento de la función. Este gradiente negativo permite “guiar” al algoritmo descenso del gradiente para acercarse, de manera progresiva debido a  $n$ , al mínimo ideal de la función.

5. Volver al paso 2 hasta completar la cantidad de epochs seteados como hiper-parámetro de nuestro modelo de red neuronal.

Cabe destacar que este entrenamiento descrito tiene en cuenta el uso del *algoritmo de gradiente descendiente “básico”* y no el *algoritmo de gradiente descendiente estocástico*

---

<sup>27</sup>Epoch es un hiper-parámetro de nuestra red neuronal, donde cada epoch representa una iteración sobre nuestros datos de entrenamiento completo. Esta iteración incluye el cálculo de los forward propagation y back propagation correspondientes.

<sup>28</sup>Una función de optimización en redes neuronales tiene como objetivo encontrar los pesos W que minimicen los errores de clasificación / la función de coste.

(stochastic gradient descent, *SDG*), algo que explicaremos en mayor detalle en la sección *Entrenamiento y función de costo con Softmax*.

### 3.5.5.1 Hiper-parámetro $n$ .

Como detallamos anteriormente,  $n$  es la tasa de aprendizaje de nuestra red neuronal e interviene en la actualización de nuestros parámetros  $\theta$  al aplicar backpropagation utilizando el algoritmo descenso del gradiente.  $n$  define y permite regular la velocidad con la que nos desplazamos por el espacio de parámetros y por lo tanto cuán rápido nos acercamos al mínimo de la función  $J(\theta)$ .  $n$  definirá la cantidad de iteraciones requerida para que el algoritmo descenso del gradiente encuentre el mínimo de la función.

$n$  es un parámetro que a diferencia de  $\theta$  no es ajustado por el algoritmo de optimización del descenso del gradiente, sino que debe ser configurado por separado. Como mencionamos anteriormente, a este tipo de parámetros se los denomina hiper-parámetros, y requieren de cierto cuidado en su elección para garantizar el buen rendimiento de la red neuronal.

Mientras más rápido nos acerquemos al mínimo de la función, más rápido la red estará aprendiendo a realizar su tarea. Sin embargo, como ya mencionamos antes, valores muy altos del hiper-parámetro  $n$  pueden tener un efecto contrario, llevando a que eventualmente el algoritmo pase de largo de manera reiterada esta mínima, impidiendo el aprendizaje. A veces aprender requiere ir más lento y ser más cuidadosos. Lo mismo ocurre con la red, y en ese caso es conveniente ajustar  $n$  para conseguir que la red no pase por alto los detalles. En la Figura 3.20 podemos observar el efecto de diferentes tasas de aprendizaje durante el entrenamiento.

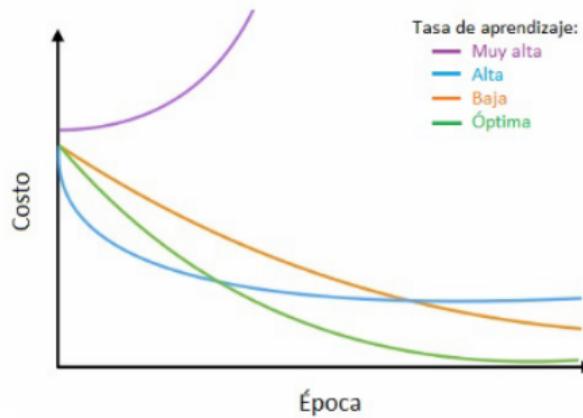


Figura 3.20: Efecto de diferentes tasas de aprendizaje  $n$  durante el entrenamiento[50].

Una tasa de aprendizaje demasiado alta puede ocasionar que el algoritmo de entrenamiento no pueda disminuir el costo de entrenamiento más allá de cierto punto, o en el peor de los casos puede ocasionar que el mismo aumente. Una tasa de aprendizaje demasiado baja puede hacer que el costo disminuya muy lentamente, ralentizando el aprendizaje. La tasa de aprendizaje  $n$  ideal es un compromiso entre lograr una disminución significativa del costo y mantener una buena velocidad de aprendizaje[50].

## **4 Natural Language Processing.**

### **4.1 Introducción.**

El Procesamiento del Lenguaje Natural o Natural Language Processing (NLP) es un subconjunto de la Inteligencia Artificial (IA) que tiene como objetivo principal la construcción de sistemas que permitan procesar el lenguaje humano capturando la complejidad del mismo, y traducirla en información resumida produciendo *resultados* comprensibles por la computadora o *respuestas* comprensibles por humanos como parte de la aplicación/implementación de IA. De esta manera, NLP permite que las computadoras puedan comprender el lenguaje humano.

Para abordar este problema y poder extraer el significado de los lenguajes humanos, la mayoría de las técnicas de NLP se basan y utilizan enfoques del aprendizaje automático (machine learning) y del aprendizaje profundo (deep learning)[51, 52].

NLP podría definirse como “la disciplina que estudia los aspectos lingüísticos de la comunicación humano-humano y humano-máquina, desarrolla modelos de competencia y desempeño lingüísticos, emplea marcos computacionales para implementar procesos que incorporan tales modelos, identifica metodologías para el refinamiento iterativo de tales procesos/modelos e investiga técnicas para evaluar los sistemas resultantes”[53].

NLP es un área interdisciplinaria basada en muchos campos de estudio. Estos campos incluyen a la informática, que proporciona técnicas para la representación de modelos y el diseño e implementación de algoritmos; la lingüística, que identifica modelos y procesos lingüísticos; las matemáticas, que aporta modelos y métodos formales; la psicología, que estudia modelos y teorías del comportamiento humano; la filosofía, que proporciona teorías y preguntas sobre los principios subyacentes del pensamiento, el conocimiento lingüístico y los fenómenos; estadísticas, que proporciona técnicas para predecir eventos basados en datos de muestra; ingeniería eléctrica, que aporta teoría de la información y técnicas para el procesamiento de señales; y biología, que explora la arquitectura subyacente de los procesos lingüísticos en el cerebro[53].

## 4.2 Aplicaciones NLP.

Como mencionamos previamente, NLP generalmente utiliza algoritmos de machine learning y deep learning para resolver distintos problemas. Algunas de las áreas más interesantes donde se aplica NLP junto con ML y/o DL para la resolución de problemas son[54, 55]:

- **Traducción automática (Machine Translation, MT).** Estas aplicaciones implican el uso de técnicas matemáticas y algorítmicas para traducir documentos de un idioma humano a otro. Realizar una traducción eficaz es muy complicado incluso para los seres humanos, ya que requiere de gran comprensión y conocimiento de las palabras, frases, sensibilidades culturales, sintaxis y la semántica de los dos idiomas involucrados.
- **Resumidores de texto (Text Summarization).** Estas aplicaciones permiten encontrar elementos de interés en los documentos para producir un resumen del contenido más importante. Estas tareas incluyen procesamiento de texto sintáctico, semántico y del nivel de voz. Hay dos tipos principales de resumen: extractivo y abstractivo. El primero se enfoca en la extracción, simplificación, reordenación y concatenación de oraciones para transmitir la información importante en los documentos usando texto tomado directamente de los documentos. En cambio, los resúmenes abstractivos se basan en expresar el contenido de los documentos a través de la abstracción de estilo generacional, posiblemente usando palabras nunca vistas en los documentos.
- **Generación de texto (Text Generation).** Muchas tareas de NLP requieren la generación de un lenguaje similar al humano. Las aplicaciones de *Text Summarization* y *Machine Translation* convierten un texto en otro de secuencia a secuencia (seq2seq). Otras tareas, como los subtítulos de imágenes y videos y los informes meteorológicos y deportivos automáticos, convierten datos no textuales en texto. Algunas tareas, sin embargo, producen texto sin datos de entrada para convertir (o con solo pequeñas cantidades utilizadas como guía). Estas tareas incluyen la generación de poesía, la generación de chistes y la generación de historias.
- **Clasificación de texto (Text Classification).** Text Classification permite la asignación de documentos de texto a clases predefinidas. Pueden tratarse de muchas clases o simplemente de una clasificación binaria. Text Classification tiene numerosas aplicaciones, un ejemplo es el *análisis de sentimiento (Sentiment Analysis)*. Un ejemplo de una tarea de Sentiment Analysis es la extracción del sentimiento de un escritor y su clasificación: si su inclinación fue positiva, negativa o neutral hacia algún tema o idea.
- **Sistemas de recuperación de información (Information Retrieval -IR-systems).** El propósito de los sistemas IR es ayudar al usuario a encontrar la información correcta (más útil) en el formato correcto (más conveniente) en el momento correcto (cuando la necesitan). Esta información se encuentra en documentos digitales; y lo que se intenta recuperar son textos, imágenes, sonidos o datos de otras características que sean relevantes para el usuario. Entre muchos problemas en IR, un problema principal que debe abordarse se relaciona con la

clasificación de documentos con respecto a una cadena de consulta en términos de puntajes de relevancia para tareas de recuperación ad-hoc, similar a lo que sucede en un motor de búsqueda. Los modelos de DL sirven para resolver estos problemas, permitiendo para la recuperación ad-hoc hacer coincidir los textos de las consultas con los textos de los documentos para obtener puntajes de relevancia.

- **Sistemas de extracción de información (Information Extraction -IE-systems).** El propósito de los sistemas IE es extraer información explícita o implícita del texto. Los resultados de los sistemas varían, pero a menudo los datos extraídos y las relaciones dentro de ellos se guardan en bases de datos relacionales. La información comúnmente extraída incluye entidades, relaciones o eventos en el texto.
  1. *Named Entity Recognition (NER)*: Se refiere a identificar y reconocer entidades en los textos y “tagearlas”: por ejemplo personas, empresas, fechas, lugares, precios, etc.
  2. *Extracción de eventos*: Se ocupa de identificar palabras o frases que se mencionan o refieren a la ocurrencia de eventos, junto con los participantes, roles de los mismos, objetos, momentos en que dicho evento ocurrió y desencadenantes del mismo.
  3. *Extracción de relaciones*: Implica la extracción de relaciones en los textos, las cuales pueden ser relaciones posesivas, antónimas o sinónimas, familiares o geográficas.
- **Respuesta a preguntas (Question Answering, QA).** Al igual que las aplicaciones de Text Summarization e IE, los sistemas de QA recopilan palabras, frases u oraciones relevantes de un documento. QA devuelve esta información de manera coherente como respuesta a una solicitud o consulta de un usuario. Dada una pregunta y un conjunto de documentos, un sistema de QA intentará encontrar la respuesta exacta, o al menos la parte exacta del texto en la que aparece la respuesta. Los métodos actuales se asemejan a los de Text Summarization
- **Reconocimiento de voz (Speech Recognition).** Speech Recognition es el proceso de asignar señales de voz acústicas a un conjunto de palabras. Las dificultades de esta tarea surgen debido a las amplias variaciones en la pronunciación de las palabras y las ambigüedades homónimas y acústicas.
- **Combinación de voz (Speech Combination).** Speech Combination permite la producción o creación de expresiones de oraciones en lenguaje natural. Para poder crear expresiones, el texto debe procesarse. Por lo tanto, NLP sigue siendo un componente esencial de cualquier sistema de combinación de voz.
- **Sistemas de intercambio (Exchange Systems).** Por lo general estos sistemas se enfocan en una aplicación de caracterización limitada (por ejemplo, una heladera o sistema de sonido doméstico) y utilizan los niveles fonéticos y léxicos del idioma.
- **Imitación de autores (Author mimicking).** Teniendo suficientes datos se pueden utilizar modelos de DL para generar textos que repliquen el estilo particular de un escritor.

## 4.3 NLP en la práctica.

En la práctica, el uso de NLP generalmente nos sirve como *complementación* a algoritmos de ML o DL para realizar distintas tareas, las cuales pueden ser por ejemplo las mencionadas en la sección sección *Aplicaciones NLP*.

Para la implementación de este Proyecto, la *tarea específica* que involucra el uso de técnicas de NLP es la obtención de mediciones de similitud entre dos textos, las cuales posteriormente serán utilizadas como entradas a algoritmos de ML (en nuestro caso K-means, y luego KNN) para realizar la *tarea global* de clasificación de nuestro Proyecto: en base a dichas mediciones de similitud lograr clasificar qué tan similares son los currículums vitae de nuevos candidatos con respecto a la descripción de un puesto de IT.

La obtención de estas mediciones de similitud, resumidamente, involucra dos pasos[1]:

1. Convertir cada uno de los documentos de texto en un objeto matemático (vector numérico).
2. Definir y emplear una métrica de distancia que será utilizada como nuestra medida de similitud entre los vectores numéricos obtenidos del paso anterior, los cuales representan nuestros textos.

Sin embargo, anteriormente al primer paso, y en general para cualquier tipo de tarea de NLP que implica manipulación de datos de texto, nuestros textos deben *pre-procesarse* para lograr una limpieza y normalización de los mismos de modo que posteriormente puedan ser convertidos en vectores numéricos[52].

A modo de resumen, en la Figura 4.1 se describen los pasos generales necesarios para obtener finalmente una representación vectorial de nuestros documentos de texto y posteriormente aplicar las mediciones de similitud sobre los mismos.

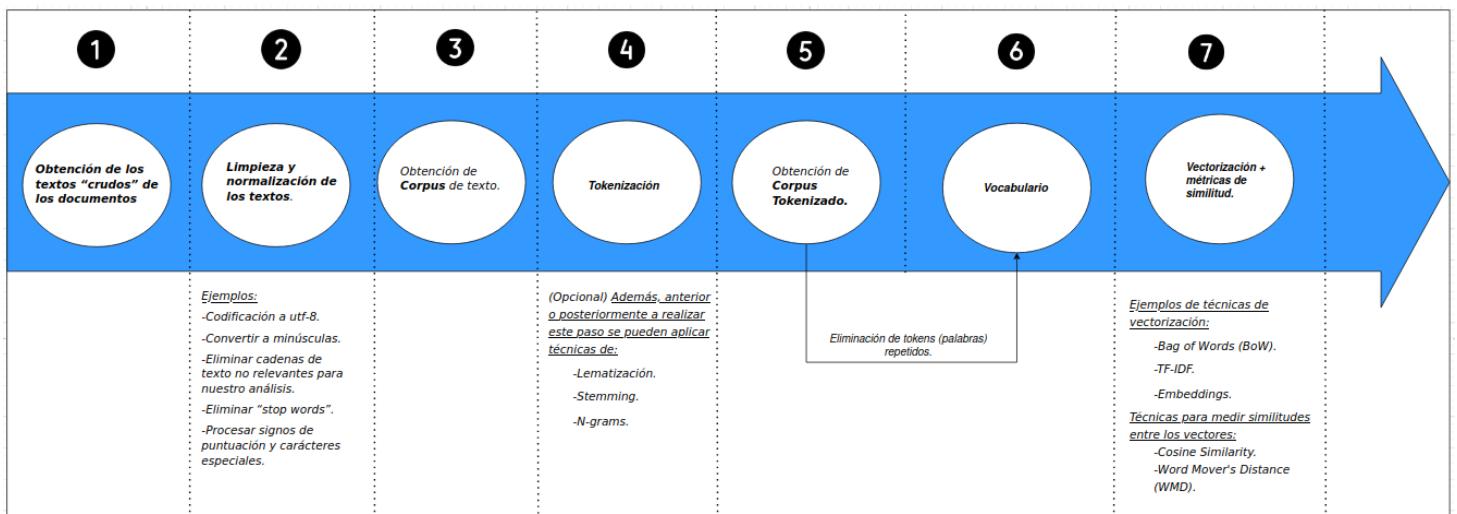


Figura 4.1: Pasos generales de nuestro proceso NLP.

El primer paso, que consiste en la obtención de los textos “crudos” de los documentos que utilizaremos en los sub-siguientes pasos, será explicado directamente en la sección de implementación, donde se obtienen los textos de los Curriculum Vitae y de las descripciones de puestos laborales de IT (Ver sección *Obtención de sets de datos*). Mientras que los pasos 2-7 serán explicados en detalle en las secciones posteriores.

Algo a tener en cuenta es la diferencia entre *corpus tokenizado* y *vocabulario* (pasos 5 y 6 de la Figura 4.1), debido a que mencionaremos estos conceptos muy a menudo. El *corpus tokenizado* son todas las palabras de nuestro corpus -el cual fue previamente tokenizado-, mientras que el *vocabulario* es el set de palabras *únicas* -sin repetir- obtenidas de dicho *corpus tokenizado*.

Para entender mejor esto, detallaremos un ejemplo pasando por todas las fases descritas en la Figura 4.1. Partiendo de “She will park the car on the street so we can walk in the central park” como nuestro texto “crudo” de documento, pasamos a la segunda fase de *limpieza y normalización* (donde, entre otras cosas, se eliminan las stop words). De esta manera nos quedaría el *corpus*: “park car street walk central park”. Este corpus pasa luego por la fase de *tokenización*, en la cual se partitiona el texto en pequeñas partes llamadas “tokens”. Estos tokens se agrupan formando el *corpus tokenizado*, el cual será un vector [“park”, “car”, “street”, “walk”, “central”, “park”] con un tamaño de 6. Por último, para formar nuestro *vocabulario* simplemente eliminamos los tokens (o palabras) repetidos, quedándonos el vector [“park”, “car”, “street”, “walk”, “central”] de tamaño 5.

## 4.4 Preprocesamiento de textos.

Este paso consiste en limpiar y normalizar nuestros textos de modo que posteriormente puedan ser convertidos en vectores numéricos. El pre-procesamiento de datos usualmente tiene un impacto significativo en la performance de generalización de nuestro algoritmo de machine learning[33].

Algunas palabras no son útiles en términos de poder predictivo porque son demasiado frecuentes y, por lo tanto, poco discriminantes. El objetivo del pre-procesamiento es preparar la representación de nuestros textos para que puedan alimentar a nuestros modelos de ML.

Las tareas de pre-procesamiento de textos más esenciales incluyen eliminar caracteres especiales o caracteres específicos que no son relevantes para nuestros análisis, convertir a minúscula nuestros textos, remover las ‘stopwords’, aplicar stemming y/o lemmatization, tokenizar nuestros textos, usar n-gramas, etc. A continuación explicaremos en mayor detalle estas tareas[52].

#### 4.4.1 Limpieza y normalización.

En esta limpieza inicial podemos considerar las siguientes sub-tareas:

1. *Codificar nuestros textos a utf-8*: es común enfrentar problemas de representación y codificación cuando nuestros textos tienen algunos caracteres específicos, como por ejemplo acentos. Para evitar estos tipos de problemas generalmente se aplica una codificación a UTF-8 antes de procesar los textos.
2. *Convertir nuestros textos a minúsculas*: Pasar cada letra de nuestros textos a minúsculas permitirá eliminar cualquier ambigüedad entre letras minúsculas y mayúsculas.
3. *Eliminar palabras o cadenas de texto no relevantes*: Por ejemplo, podemos eliminar e-mails, páginas web o palabras que no necesitamos tener en cuenta en nuestros análisis. Para eliminar e-mails o páginas web podemos utilizar *expresiones regulares*<sup>29</sup>, las cuales nos ayudan a identificar los formatos de URLs o casillas de mail y aplicar algún tipo de regla para eliminarlos (reemplazandolos por palabras vacías).
4. *Procesar signos de puntuación y caracteres especiales*: Esta tarea no precisamente consiste en eliminar los signos de puntuación y caracteres especiales, ya que si bien algunos de ellos no son necesarios, otros sí lo son. Por ejemplo, en tareas de análisis de sentimiento, un signo de exclamación en un tweet puede expresar un énfasis en su sentimiento, ya que las personas tienden a aumentar el número de signos de exclamación cuando se trata de sentimientos fuertes. Por lo tanto, los caracteres especiales y de puntuación pueden conservarse o eliminarse (reemplazandolos por palabras vacías). Estas palabras con signos de puntuación y/o caracteres especiales, también pueden tratarse utilizando expresiones regulares.
5. *Eliminación de “stop words”*: Los humanos al comunicarnos usamos palabras como “el”, “y” o “a” para que las oraciones tengan sentido. Sin embargo, estas palabras no tienen ningún significado y no son útiles para el análisis de datos. Las “stop words” se consideran palabras vacías y deshacerse de las mismas es un paso del pre-procesamiento que permite centrarse en las palabras que importan. Cada idioma posee su propia lista de palabras vacías que se pueden ajustar según el problema.

---

<sup>29</sup>Las expresiones regulares nos permiten identificar patrones y aplicar reglas cuando estos se detectan. De esta manera, podemos extraer información textual que sigue un formato particular: por ejemplo, “Mr /. (. \*)” es una expresión regular que permite buscar los textos que empiecen con ‘Mr’ y tener en cuenta el grupo que le sigue, el cual se supone que representa el nombre de la persona[52].

#### 4.4.2 Tokenización.

Como explicamos previamente, los textos necesitan ser tratados previamente a ser convertidos en números y ser procesados por un modelo. Para lograr esto, luego de realizar las tareas de limpieza inicial detalladas en la sección anterior, se debe tokenizar (o particionar) los textos en pequeñas partes llamadas “tokens” y luego re-agrupar los mismos formando una lista de tokens.

De esta manera, la *tokenización* consiste en dividir un texto en partes más pequeñas llamadas *tokens*. Un token puede ser un párrafo del texto, una oración, una palabra o cada uno de los caracteres de nuestro texto. Generalmente se arman los tokens dividiendo al texto en palabras. En la Figura 4.2 podemos observar una tokenización a nivel de palabras, mientras que en la Figura 4.3 se realiza a nivel de caracteres.

```
?]: └ i = 5
    sentence = Tweet[i:i+1]['text'][i]
    sentence

it[32]: 'http://www.dothebouncy.com/smfc - some shameless plugging for the best Rangers forum on earth'

!]: └ print(preprocessing.preprocess_generic(sentence))

Result: ['-', 'some', 'shameless', 'plugging', 'for', 'the', 'best', 'rangers', 'forum', 'on', 'earth']
```

Figura 4.2: Tokenización por palabras[52].

Figura 4.3: Tokenización por caracteres[52].

Tokenizar las palabras nos permite encontrar similitudes entre las mismas: por ejemplo, detectar las palabras con las mismas raíces. Para encontrar estas similitudes podemos utilizar los métodos de stemming y lemmatization, los cuales se verán en la próxima sección.

La lista de tokens que obtendremos luego de aplicar tokenización (y en caso de ser necesario stemming, lemmatization y N-grams) sobre todos los documentos de texto incluidos en nuestros datos, es el *corpus tokenizado* que mencionamos previamente. Luego, al eliminar los tokens repetidos, podemos obtener nuestro *vocabulario*, el cual será útil en tareas posteriores.

#### 4.4.3 Stemming y lemmatization.

Como mencionamos previamente, estas dos técnicas nos permiten encontrar similitudes entre las palabras, y por ejemplo, detectar las palabras con las mismas raíces.

- *Stemming (derivación)*: Las reglas gramaticales a menudo exigen que se agreguen algunos sufijos a las palabras para que una oración tenga sentido; sin embargo, estos sufijos no son imperativos cuando se trata de comprender el significado de una palabra. Stemming es un paso del pre-procesamiento que elimina las partes innecesarias de una palabra manteniendo su *raíz* (“root” o también llamado “*stem*” en inglés). Podemos ver un ejemplo de stemming en la Figura 4.4.

	original_word	stemmed_words		original_word	stemmed_word
0	connect	connect	0	trouble	troubl
1	connected	connect	1	troubled	troubl
2	connection	connect	2	troubles	troubl
3	connections	connect	3	troublesome	troublemsom
4	connects	connect			

Figura 4.4: Stemming[52].

- *Lemmatization (lematización)*: Mientras que stemming elimina los sufijos y prefijos de una palabra y solo mantiene su raíz, lemmatization es un algoritmo que reemplaza una palabra por su forma más básica, también llamada “lema”. Un lema puede ser una forma infinitiva, un sustantivo, un adjetivo, etc., mientras que una raíz a menudo no significa nada: podemos ver por ejemplo en la Figura 4.4 que el stem ‘troublemsom’ no tiene ningún significado.

Utilizar lemmatization es útil porque en algunos idiomas, las palabras con diferentes significados pueden tener la misma raíz, y utilizando stemming nos daría la misma raíz para todas las palabras, lo que supondría algo incorrecto. Por lo tanto, los lemas tienen como objetivo transmitir una idea con una palabra real en lugar de una raíz básica sin un significado definido, corrigiendo el problema de tener dos conceptos diferentes con la misma raíz. Por lo tanto, la lematización requiere un análisis morfológico de la palabra y la existencia de un diccionario detallado para que funcione el algoritmo, lo que hace que su implementación sea más compleja que utilizar stemming. Podemos ver un ejemplo de lemmatization en la Figura 4.5.

Form	Morphological information	Lemma
studies	Third person, singular number, present tense of the verb <b>study</b>	study
studying	Gerund of the verb <b>study</b>	study
niñas	Feminine gender, plural number of the noun <b>niño</b>	niño
niñez	Singular number of the noun <b>niñez</b>	niñez

Figura 4.5: Lemmatization[52].

#### 4.4.4 N-grams.

A veces, asociar / juntar palabras puede cambiar el significado o traer información diferente. Por ejemplo, si estamos prediciendo el sentimiento de una oración, se esperaría que la palabra “malo” empujara la salida de un modelo hacia una etiqueta negativa. Sin embargo, “no está mal” haría lo contrario. Debido a esto, es posible que debamos considerar “no está mal” como una palabra única.

La construcción de nuevas “palabras” provenientes de la co-ocurrencia de palabras se llama “n-grams”, donde “n” representa el número de palabras que nos gustaría considerar (en nuestro ejemplo “no está mal” es un “3-grams”). Este procesamiento tiene como objetivo enriquecer el diccionario de palabras[52].

### 4.5 Obtención de representaciones vectoriales.

Una vez que nuestros datos están limpios y ya tenemos nuestro *corpus tokenizado* (formado por tokens de palabras de nuestros textos), el siguiente paso es transformar estos datos *textuales* en representaciones vectoriales *numéricas* mediante **técnicas de vectorización**.

Esta operación se lleva a cabo debido a que las computadoras y los algoritmos de machine learning no pueden entender ni aprender utilizando textos o palabras, ya que operan en un espacio de valores numéricos. Por lo tanto, para realizar tareas de ML sobre textos, necesitamos transformar nuestros documentos en representaciones vectoriales con valores numéricos. El objetivo es que la representación vectorial de nuestros documentos *representen nuestro lenguaje de la mejor forma posible para que los modelos de ML usados posteriormente puedan aprender los patrones necesarios para entender dicho lenguaje*[56].

Existen distintas técnicas de vectorización para lograr esto. A continuación se detallarán algunas de las más populares y utilizadas[52].

#### 4.5.1 Bag of Words (BoW).

*Bag of Words (BoW)*, o también llamado ‘Count Vector’, es el método fundamental y más básico para obtener nuestras representaciones vectoriales de documentos. Suponiendo que tenemos  $N$  documentos de texto, el modelo BoW permite representar cada uno de estos documentos como una bolsa, la cual contiene algunas palabras. Estas palabras se encuentran dentro de un *diccionario* de palabras. De esta manera, BoW mediante un modelado basado en diccionarios, *describe la frecuencia u ocurrencia de cada una de las palabras en los documentos*.

El primer paso del método BoW consiste en crear dicho diccionario (al que representaremos como  $D$ ), el cual contiene nuestro *vocabulario*. Este vocabulario representa a todas las palabras (sin repetir) de nuestro conjunto de documentos de texto  $d$ .

Luego, suponiendo que nuestro conjunto de documentos  $d$  está formado por  $n$  documentos, y queremos obtener una representación vectorial para cada una de los  $d[N]$  documentos, lo que se hace es utilizar el diccionario  $D$  obtenido del paso anterior para generar una matriz que cuente la cantidad de veces que aparece cada una de las palabras únicas de  $d$  para cada uno de nuestros  $d[N]$  documentos.

En la Fórmula 16 podemos observar la representación matemática de nuestro set de documentos  $d$ .

$$d = (w_i), \text{ para } i \in [1, n_d] \quad (16)$$

Donde:

- $w$  es un *segmento* formado por una secuencia finita de *caracteres*. Si nuestros *caracteres* son letras, un *segmento* podría ser una palabra.
- $d \in D$ , siendo  $D$  nuestro diccionario de secuencias finitas de segmentos  $w$  sin repetir (o vocabulario).
- $n_d$  es la longitud de  $d$  (número total de palabras del set de documentos).

Resumidamente,  $d$  es un conjunto de segmentos  $w_i$ , y  $D$  está compuesto por un conjunto de segmentos  $w_i$  sin repetir (nuestro vocabulario).

Como mencionamos previamente, la representación vectorial mediante BoW indica la cantidad de veces que aparece cada palabra en nuestro corpus. Cada componente del vector indica esta cantidad. En el ejemplo de la Tabla 2, cada columna representa las palabras que componen nuestro vocabulario / diccionario  $D$ , y cada fila indica la representación vectorial mediante BoW para cada documento (sin realizar un pre-procesamiento).

Vocabulario⇒ Corpus ↓	the	cat	sat	on	hat	dog	ate	and
<i>The cat sat on the hat</i>	2	1	1	1	1	0	0	0
<i>The dog ate the cat and the hat</i>	3	1	0	0	1	1	1	1

Tabla 2: Representación mediante BoW.

Ejemplo obtenido de [57].

Usualmente se coloca un 1 como componente de nuestros vectores cuando el conteo de la palabra es  $> 1$ , representando de esta manera si una palabra en particular está presente o no. De esta manera, nos quedaría la representación BoW mediante un vector binario, como observamos en la Tabla 3.

Vocabulario⇒ Corpus ↓	the	cat	sat	on	hat	dog	ate	and
<i>The cat sat on the hat</i>	1	1	1	1	1	0	0	0
<i>The dog ate the cat and the hat</i>	1	1	0	0	1	1	1	1

Tabla 3: Representación binaria mediante BoW.

Ejemplo obtenido de [57].

Consideremos el siguiente ejemplo para explicar el funcionamiento detallado de BoW[52]: Siendo  $d = ["He is a good boy", "She is a good girl", "Boys and girls are good"]$ , nuestro objetivo es obtener una representación vectorial para cada uno de los  $d[N]$  documentos<sup>30</sup>. Luego del pre-procesamiento -que incluye conversión a minúsculas, lematización, limpieza de stopwords<sup>31</sup>, etc.-,  $d$  se transforma en  $d' = ["good boy", "good girl", "boy girl good"]$ .

Construimos nuestro diccionario  $D$  con las palabras disponibles en  $d'$ :  $D = ["good", "boy", "girl"]$ .

Usando este diccionario y contando la ocurrencia de cada palabra del diccionario en cada documento  $d'[i]$ , obtendremos la representación mediante BoW para nuestro set de documentos  $d'$  observada en la Tabla 4. En esta tabla cada fila representa el vector numérico obtenido mediante BoW para cada uno de los  $d'[N]$  documentos.

---

<sup>30</sup>También podríamos considerar a  $d$  como un único documento y  $d[N]$  cada oración del documento; los cálculos se realizan de la misma manera solo que en lugar de obtener las representaciones vectoriales para cada documento ahora lo haríamos para cada oración.

<sup>31</sup>En BoW la frecuencia de los stopwords, al estar muy presente en gran parte de los textos, serán mayores (aunque las mismas no tengan impacto en la representación del documento). Como mencionamos anteriormente, las stopwords no son útiles para el análisis de texto por considerarse palabras vacías. Es por esto que generalmente las mismas se remueven en la etapa de pre-procesamiento previamente a crear el modelo de BoW.

$d'[N]$	<b>good</b>	<b>boy</b>	<b>girl</b>
$d'[1] = \text{"good boy"}$	1	1	0
$d'[2] = \text{"good girl"}$	1	0	1
$d'[3] = \text{"boy girl good"}$	1	1	1

Tabla 4: Representación mediante BoW.  
Ejemplo obtenido de [52].

#### 4.5.2 TF-IDF.

Las representaciones mediante BoW son muy sencillas de comprender, sin embargo sólo describen un documento de manera independiente, sin tener en cuenta el contexto de las palabras del corpus. Un mejor enfoque sería considerar la frecuencia relativa o rareza de las palabras / tokens en el documento frente a su frecuencia en los otros documentos. La idea central es que lo más probable es que el significado esté codificado en los términos más raros de un documento. Dicho de otra manera, en BoW las palabras que tienen mayor peso son las que aparecen en muchos documentos (sin embargo, debería ser lo opuesto).

*TF (Term Frequency)* e *IDF (Inverse Document Frequency)* solucionan este problema otorgando más peso a las palabras que aparecen en pocos documentos y menos peso a las palabras que aparecen en muchos documentos. La razón de esto es que una palabra que aparece en muchos documentos mayormente no es relevante (como por ejemplo las stop-words); en cambio palabras que aparecen en pocos documentos son más discriminantes y deberían tener mayor peso. Por ejemplo, en un corpus tokenizado obtenido de un corpus de texto deportivo, los tokens como "corner", "penalz" "botines" aparecen con más frecuencia en documentos que tratan sobre fútbol, mientras que otros tokens que aparecen con frecuencia en todo el corpus tokenizado, como "correr", "puntuación" y "jugar" son menos importantes[56].

Otra de las desventajas de BoW es su incapacidad de tener en cuenta qué palabras son más importantes para cada uno de nuestros documentos[52]. Por ejemplo, viendo la Tabla 4, podemos observar que hay muchas palabras que tienen los mismos pesos (1 o 0): para  $d'[1] = [1, 1, 0]$  observamos que las palabras "good" (1) y "boy" (1) tienen el mismo valor/peso (1). TF e IDF también permiten solucionar este problema: vemos la nueva representación de  $d'[1]$  en la Tabla 7; de esta manera "boy" tiene mayor peso/relevancia que "good" por estar presente en menos documentos.

De esta manera, como mencionamos previamente, el modelo BoW puede ser mejorado añadiendo TF (Term Frequency) e IDF (Inverse Document Frequency). TF permite contar la cantidad de veces que aparece cada palabra/token en cada documento; mientras que la idea de IDF es darle a cada palabra/token un peso que sea inversamente proporcional a su frecuencia. Las palabras que aparecen en muchos documentos serán entonces menos importantes que las palabras que solo aparecen en unos pocos documentos[10]. Matemáticamente, podemos describir TF como en la Fórmula 17, e IDF como en la Fórmula 18.

$$tf(w, d[i]) = \frac{\text{Nº veces que la palabra } w \text{ aparece en el documento } d[i]}{\text{Total de palabras en el documento } d[i]} \quad (17)$$

$$idf(w) = \log\left(\frac{N}{df(w)}\right) \quad (18)$$

Donde  $N$  es el número de documentos  $d[i]$  presente en el conjunto de documentos  $d$ , y  $df(w)$  es el número de documentos  $d[i]$  en los que aparece la palabra  $w$ .

Multiplicando  $tf(w, d[i])$  e  $idf(w)$  para cada palabra  $w$  y cada documento  $d[i]$ , obtenemos una nueva matriz que tendrá en cuenta el contexto de las palabras en nuestro corpus, y permitirá saber mediante los pesos qué palabras son más importantes para cada uno de nuestros documentos.

Siguiendo el ejemplo de BoW, en la Tabla 5 obtendremos los cálculos de TF, mientras que en la Tabla 6 obtendremos los cálculos de IDF.

$d'[N]$	<b>good</b>	<b>boy</b>	<b>girl</b>
$d'[1]$	0,5	0,5	0
$d'[2]$	0,5	0	0,5
$d'[3]$	0,33	0,33	0,33

Tabla 5: Cálculos de **TF** para nuestro set de documentos  $d[52]$ .

Por ejemplo, los cálculos de TF para la palabra “boy” (tercer columna) y cada uno de nuestros documentos serían:

$$\begin{aligned} tf("boy", "good boy") &= 1 / 2 = 0,5 \\ tf("boy", "good girl") &= 0 / 2 = 0 \\ tf("boy", "boy girl good") &= 1 / 3 = 0,33 \end{aligned}$$

	<b>good</b>	<b>boy</b>	<b>girl</b>
<b>IDF</b>	0	0.176	0.176

Tabla 6: Cálculos de **IDF** para nuestro set de documentos  $d[52]$ .

Por ejemplo, los cálculos de IDF para cada palabra  $w$  (recordando que  $d' = ["good boy", "good girl", "boy girl good"]$ ) serían:

$$\begin{aligned} idf("good") &= \log(3 / 3) = 0 \\ idf("boy") &= \log(3 / 2) \simeq 0.176 \\ idf("girl") &= \log(3 / 2) \simeq 0.176 \end{aligned}$$

Por último, en la Tabla 7 multiplicamos  $tf(w, d[i])$  e  $idf(w)$  para obtener nuestra Representación vectorial TF-IDF para cada uno de los  $d'[N]$  documentos.

Matemáticamente, debido a la relación de la función logarítmica, la puntuación TF-IDF siempre estará entre 0 y 1; en ese sentido cuanto más cercana a 1 sea la puntuación TF-IDF de una palabra, más informativa será esa palabra para ese documento. Cuanto más cerca esté la puntuación de cero, menos informativa será esa palabra [56].

$d'[N]$	<b>good</b>	<b>boy</b>	<b>girl</b>
$d'[1]$	0	$0,5 \times 0,176 = \mathbf{0.088}$	0
$d'[2]$	0	0	$0,5 \times 0,176 = \mathbf{0.088}$
$d'[3]$	0	$0,33 \times 0,176 = \mathbf{0.058}$	$0,33 \times 0,176 = \mathbf{0.058}$

Tabla 7: Representación mediante **TF-IDF**[52].

Comparando los resultados obtenidos con la representación previa de BoW (Tabla 4), podemos observar que la palabra “good” que estaba representada en cada documento con un 1; ahora con TF-IDF está representada por 0s ya que “good” al estar presente en todos los documentos, se considera una palabra muy común y se le da menor importancia/peso.

Además, observamos que para  $d'[1]$ , comparando con la Tabla 4, en su representación con TF-IDF “boy” tiene mayor peso/relevancia que “good” por estar presente en menos documentos (en BoW estaban representados ambos con 1). En  $d'[2]$  sucede lo mismo: la palabra “girl” tiene mayor relevancia que “good”.

#### 4.5.3 Desventajas BoW & TF-IDF.

Las representaciones vectoriales obtenidas mediante BoW y TF-IDF tienen 2 principales desventajas[71, 52]:

- **Son dispersas (sparse) y de alta dimensionalidad (high-dimensional).** Que un vector sea *disperso (sparse)* significa que los mismos están dispersamente distribuidos de información: sus valores son en gran mayoría 0s y minoría 1s o valores que representen información (como el puntaje de TF-IDF). En cuanto a la dimensionalidad, esta es igual al tamaño de nuestro vocabulario; de esta manera, como generalmente contamos con miles e incluso millones de palabras en nuestro vocabulario, entonces nuestro vector tendrá miles de columnas, lo cual hace de esta representación *ineficiente* y dificulta el entrenamiento de los algoritmos de ML.
- **Son discretas.** Esto quiere decir que tratan a las palabras ( contenidas en cada posición del vector) como unidades atómicas de un solo valor, perdiendo así la habilidad de capturar relaciones entre palabras. TF-IDF, a diferencia de BoW, añade un peso IDF a las palabras, sin embargo tampoco existe una relación entre las mismas. Debido a esto, estas representaciones no permiten capturar la similitud entre palabras, siendo las mismas tratadas como índices de nuestro set de vocabulario. Dicho de otra forma, estas representaciones discretas no permiten entender el *contexto* de las palabras utilizadas en los documentos de texto: no permiten detectar las diferencias *semánticas*<sup>32</sup> entre las palabras ni detectar *sinónimos*.

---

<sup>32</sup>En el análisis semántico se busca entender el significado de la oración. Las palabras pueden tener múltiples significados, la idea es identificar el significado apropiado por medio del contexto de la oración.

#### 4.5.4 Word embeddings.

En la sección previa observamos dos principales desventajas con respecto a dichas representaciones básicas de texto. Para solventar estas limitaciones introduciremos una nueva forma de representar nuestros textos, también mediante vectores: **embeddings**.

En esta sección explicaremos en detalle qué son embeddings, cómo obtener dichos embeddings para representar a nuestras palabras (Word embeddings) y algunos conceptos que derivan de esta forma de representación: redes neuronales, one hot encoding, Word2vec, CBOW.

##### 4.5.4.1 Embeddings.

Los **embeddings** son una de las representaciones vectoriales más populares de texto y de palabras, los cuales son obtenidos mediante redes neuronales: ver sección *Redes Neuronales Artificiales (ANN)*. Estas representaciones vectoriales son densas (*dense*) y de baja dimensionalidad (*low-dimensional*)<sup>[71]</sup>. De esta manera, las dos limitaciones previas son solucionadas mediante las ventajas de los embeddings descritas a continuación:

- A diferencia de BoW y TF-IDF donde las representaciones son *dispersas (sparse)* y *de alta dimensionalidad (high-dimensional)*, como mencionamos anteriormente los embeddings son representaciones **densas (dense)** y generalmente **de menor dimensionalidad**. Que un vector sea denso significa que los mismos están densamente distribuidos de información: sus valores son en gran mayoría elementos distintos de 0, conteniendo mucha más información relevante que un vector disperso de igual dimensionalidad<sup>[58]</sup>. Con respecto a la dimensionalidad, teniendo en cuenta que un embedding es un espacio representado en otro, el cual puede ser de mayor o menor cantidad de dimensiones que el espacio original, podemos decir que el espacio original de nuestros datos reales generalmente tiene una dimensionalidad muy grande, y mediante los embeddings logramos obtener una representación de menor dimensionalidad (son más compactos)<sup>[10]</sup>. Todo esto hace que las tareas de ML utilizando estas representaciones sean más eficientes.
- A diferencia de BoW y TF-IDF donde las representaciones son discretas y no permiten capturar las relaciones entre palabras, los embeddings utilizan un vector para representar a cada palabra o documento, **siendo capaces de capturar el contexto de una palabra en un documento, similitudes semánticas y sintácticas, relaciones entre palabras o documentos, sinónimos, etc.**

Existen embeddings de palabras (*Word embeddings*) como de documentos (*Document embeddings*)<sup>[59]</sup>. En nuestro caso utilizaremos embeddings de palabras (*Word embeddings*), ya que luego esto nos permitirá aplicar mediciones de similitud mediante WMD.

Resumiendo el concepto de Word Embeddings, podemos decir que es una técnica de NLP que nos permite asignar un vector a cada palabra. Este vector es obtenido mediante el entrenamiento de redes neuronales y permiten guardar información semántica, lo que permite que pueda ser asociado a otros vectores de palabras según distintos contextos gramaticales. De esta manera, palabras similares tendrán word embeddings similares, estando dichos vectores a distancias cercanas entre sí.

Hay dos conceptos claves para entender el uso de los word embeddings:

- **Similitud distribucional**[71]: es la idea de que el significado de una palabra puede ser entendida desde el contexto en donde la palabra aparece<sup>33</sup>. Por ejemplo, “NLP rocks”, el significado literal de la palabra “rocks” es “stones” (piedras), pero teniendo en cuenta el contexto, esta palabra es usada para referirnos a otra cosa: a algo bueno, de moda. *Los word embeddings permiten capturar las similitudes distribucionales entre palabras*. De esta manera, si nos dan la palabra ‘USA’, palabras distribucionalmente similares podrían ser países (como Canada, Germany, Argentina, etc.) o ciudades de USA. Y si nos dan la palabra ‘beautiful’, palabras que comparten una relación con esta palabra (por ej. sinónimos o antónimos) pueden ser consideradas palabras distribucionalmente similares. Dicho de otra forma, existen palabras que generalmente ocurren en contextos similares.
- **Hipótesis Distribucional**[71]: En lingüística, esta hipótesis hace referencia a que las palabras que ocurren en contextos similares tienen similares significados. Por ejemplo, las palabras “dog” y “cat” ocurren en similares contextos. Por lo tanto, de acuerdo a esta hipótesis, debe existir una fuerte similitud entre los significados de esas dos palabras. **Las representaciones vectoriales mediante word embeddings se basan en esta hipótesis**: de esta manera, si dos palabras ocurren en un contexto similar, sus correspondientes representaciones vectoriales estarán cerca entre sí.

Estos word embeddings, al ser vectores, son un elemento de un espacio vectorial (o también llamado “espacio de embedding”), teniendo una magnitud y una dirección. Como ejemplo, podemos observar en la Figura 4.6 la conversión de 4 palabras en vectores (o word embeddings) de 2 dimensiones. Este es un ejemplo meramente visual, ya que en la práctica las palabras las transformamos en vectores de muchas más dimensiones (100 por ejemplo).

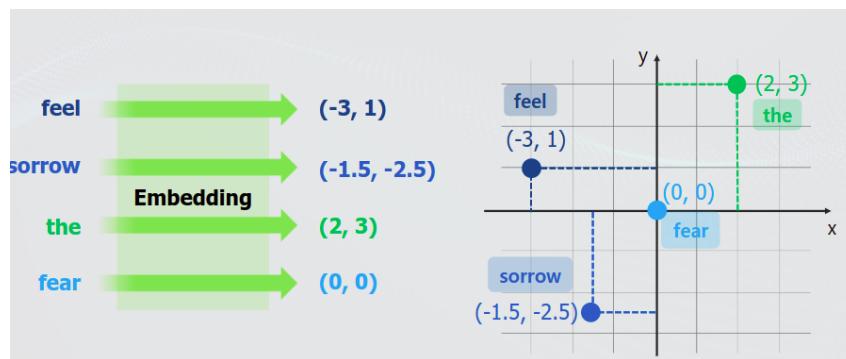


Figura 4.6: Ejemplo de embedding de 2 dimensiones [73].

En conclusión, al generar embeddings nuestras palabras con significados similares estarán colocadas cercas entre sí en este espacio de embedding. En cambio, si convertimos palabras en vectores que no almacenan ningún significado, no podemos llamar a esto embeddings.

<sup>33</sup>La similitud distribucional también es llamada *connotation*: el significado está definido por contexto; y es lo opuesto a *denotation*, que es el significado literal de cualquier palabra.

#### 4.5.4.2 Word2vec.

En 2013, Miklov[60] sugirió un modelo para obtener una representación vectorial numérica de las palabras basado en el entrenamiento de una red neuronal, conocido como “Word2vec”. Este modelo está basado en el concepto de *similitud distribucional* explicado anteriormente, permitiendo considerar el contexto del cual se toman las palabras. De esta forma, Word2vec es capaz de capturar relaciones de analogías de palabras, como por ejemplo “*King - Man + Woman = Queen*”. Estas representaciones de palabras obtenidas mediante Word2vec son nuestros *word embeddings*. En la Figura 4.7 podemos observar un ejemplo de estos word embeddings luego de entrenar un modelo Word2vec.

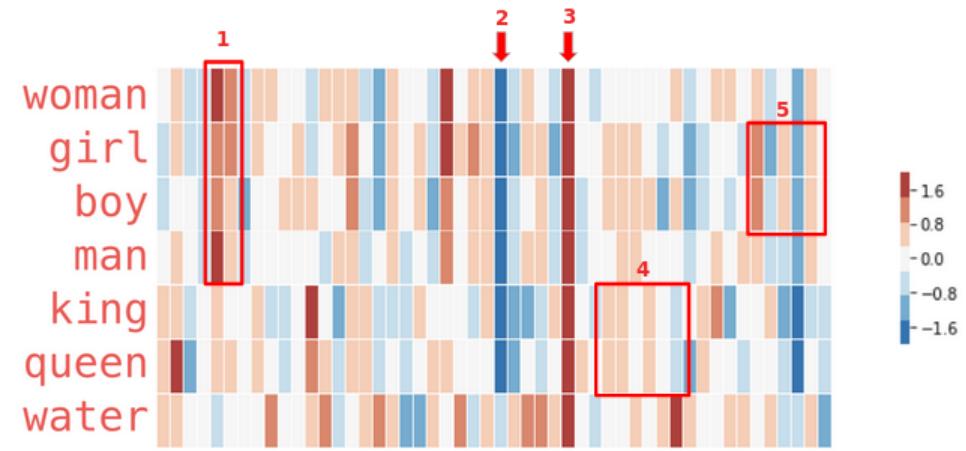


Figura 4.7: Ejemplo de Word Embeddings (1)[74].

Observamos que los valores van de 1.6 a -1.6. Los valores en rojo son los más positivos y en azul los más negativos. Vemos que tenemos varias palabras que pueden tener sus similitudes y diferencias. Por ejemplo, a simple vista podemos notar de los valores marcados en recuadros rojos (1,2,3,4,5) que los embeddings permiten codificar lo siguiente:

- Recuadro 1: Podemos ver que estos embeddings capturan muy bien la relación entre mujer, niña, niño y hombre (ya que tienen valores muy similares en ese recuadro). Podemos suponer<sup>34</sup> que la segunda columna del recuadro codifica el género y la primera la edad, entonces la combinación de esas columnas nos dan esas representaciones para esas 4 palabras.
- Recuadro 2: Vemos que la columna está activa en todas las palabras excepto en “water”. Esto podría significar que esta columna expresa si la palabra hace referencia a una persona (algo que el agua no tiene en común).
- Recuadro 3: Vemos que esa columna está siempre activa, ya que el modelo encontró que todas estas palabras tienen algo en común / son semejantes de alguna forma (por ej. podría expresar que son sustantivos).

<sup>34</sup>Vamos a hablar en estos casos de suposiciones, ya que este embedding de salida del modelo surge en base a las dimensiones que encontró el mismo mediante el entrenamiento de grandes cantidades de texto, y que nosotros no podemos comprender a simple vista.

- Recuadro 4: Vemos que en esta parte son muy parecidas las representaciones de las palabras rey y reina, quizás el modelo esté codificando allí algún significado de nobleza o de piezas de ajedrez o de tamaños de camas (esto depende con qué corpus se haya entrenado el modelo).
- Recuadro 5: Vemos que en esta parte son muy parecidas las representaciones de las palabras niño y niña, quizás el modelo esté codificando allí algún significado de juventud.

En conclusión esto es lo que buscamos, *obtener embeddings que permitan codificar los significados de las palabras*.

Además, al ser los embeddings vectores numéricos, esto nos permite operar entre ellos mediante álgebra entre vectores: ver Figura 4.8. Por ejemplo si restamos “king” y “man” y lo sumamos a “woman” nos da un vector muy similar al de “queen”. *Esta es una gran utilidad de los embeddings, nos permiten formar relaciones entre palabras y que nuestro modelo de alguna forma -que nosotros no podemos explicar- entienda estas relaciones.*

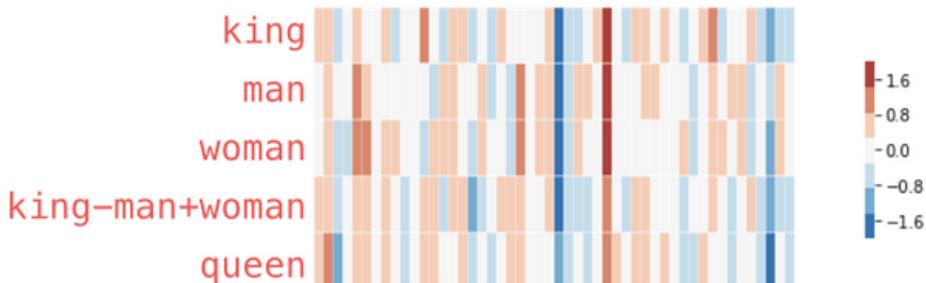


Figura 4.8: Ejemplo de Word Embeddings (2)[74].

Además de Word2vec existen otros modelos para obtener dichos word embeddings, como GloVe<sup>35</sup> y Fasttext<sup>36</sup>. En nuestra implementación utilizamos Word2vec en lugar de Glove o Fasttext ya que, según estudios[61, 62] la diferencia entre los resultados obtenidos mediante estas tres técnicas son estadísticamente similares y todas demuestran ser métodos competitivos en cuanto a performance. Los resultados utilizando estos tres métodos de word embeddings dependen de los datos de entrenamiento que se utilicen y para el dominio y tarea que se necesite. La razón por la cual se optó por utilizar Word2vec es debido al futuro uso de WMD, ya que en el paper original de WMD[3] se utilizan embeddings obtenidos mediante Word2vec, decidiendo de esta manera utilizar la misma base en nuestra implementación.

---

<sup>35</sup><https://nlp.stanford.edu/projects/glove/>

<sup>36</sup><https://fasttext.cc/>

#### 4.5.4.3 ¿Cómo obtener nuestros Word embeddings?

A continuación, explicaremos cómo trabaja el modelo ‘Word2vec’ para obtener los word embeddings y de qué maneras podemos hacerlo[71].

Como mencionamos previamente, para “deducir” el significado de la palabra, Word2vec utiliza los conceptos de **similitudes distribucionales e Hipótesis Distribucional**. Es decir, *deduce el significado del contexto: palabras que aparecen como “vecinas” en el texto. Entonces, el significado de una palabra viene dado por las palabras que aparecen con frecuencia cerca de ella: si dos palabras diferentes ocurren en contextos similares, es muy probable que sus significados también sean similares.* De esta manera podemos interpretar el resultado de una *palabra central* a partir de las palabras que están alrededor de ella (*palabras de contexto*), o dicho de otra forma “inferir por contexto qué quiere decir esa palabra central”.

Word2vec materializa esto entrenando una red neuronal y obteniendo los embeddings para cada una de las palabras del corpus de manera que el vector que representa a la palabra en el espacio vectorial / de embedding pueda capturar lo mejor posible el significado de la misma: las palabras con significados similares tienden a agruparse, y las palabras con significados muy diferentes están lejos unas de otras en dicho espacio vectorial / de embedding.

Resumidamente, Word2vec utiliza una *red neuronal de 2 capas* que toma un largo corpus de texto como input, y “aprende” a representar las palabras en vectores de igual tamaño en función de los contextos en los que aparecen en el corpus.

Existen 2 maneras de obtener nuestros word embeddings[71]:

1. *Word embeddings pre-entrenados:* Entrenar nuestros propios word embeddings es un proceso costoso en cuanto a tiempo y procesamiento. Para no realizar esta tarea costosa, podemos usar word embeddings pre-entrenados. ¿Qué son estos word embeddings pre-entrenados?: alguien realizó el duro trabajo de entrenar word embeddings sobre, generalmente, un largo corpus de texto (como Wikipedia, artículos de noticias, páginas web, etc.). Estos embeddings pueden ser descargados y podemos utilizarlos para obtener los vectores de las palabras que necesitemos. Los embeddings pueden ser pensados como una larga colección de pares clave-valor, donde la clave es la palabra en el vocabulario y el valor es el correspondiente vector de la palabra. Algunos de los más populares word embeddings pre-entrenados son Word2vec de Google<sup>37</sup>, GloVe de Stanford<sup>38</sup> y fasttext embeddings de Facebook<sup>39</sup>. Estos embeddings están disponibles en varias dimensiones (25,50,100,200,300,600, etc.).

---

<sup>37</sup><https://code.google.com/archive/p/word2vec/> - Sitio Web consultado el 5 de abril de 2022

<sup>38</sup><https://nlp.stanford.edu/projects/glove/> - Sitio Web consultado el 5 de abril de 2022

<sup>39</sup><https://fasttext.cc/docs/en/english-vectors.html> - Sitio Web consultado el 5 de abril de 2022

2. *Entrenando nuestros propios word embeddings*: para entrenar y obtener nuestros propios word embeddings, Word2vec nos ofrece 2 arquitecturas disponibles -las cuales fueron propuestas en el paper original de Word2vec[60]-:

- *Continuous bag of words (CBOW)*.
- *SkipGram*.

En nuestra implementación no utilizamos word embeddings pre-entrenados ya que verificamos que no existían varias palabras en el vocabulario de dichos embeddings (también llamadas palabras *out-of-vocabulary, OOV*): como por ejemplo, palabras de 2-grams como *data\_scientist*, *machine\_learning*, *software\_engineer*; o palabras específicas de lenguajes de programación como Python o Java. Por esta razón decidimos entrenar nuestros propios word embeddings utilizando todo el Corpus disponible de nuestros CVs de candidatos y descripciones de puestos de IT. De esta manera se tendrán en cuenta las relaciones entre palabras como *data\_science*, *machine\_learning* y demás palabras que estén presentes en nuestros CVs o Descripciones de puestos de IT).

#### 4.5.4.4 CBOW vs Skipgram.

Como mencionamos previamente, para entrenar nuestros propios word embeddings tenemos 2 posibilidades: *CBOW* o *Skipgram*. A continuación realizaremos una comparación entre estas dos arquitecturas y elegiremos una de ellas para nuestra implementación.

Tanto CBOW como SkipGram aplican el concepto de semántica distribucional -las palabras obtienen su embedding cuando miramos a qué otras palabras tienden a aparecer junto a ellas- pero visto de 2 formas distintas, y armando el dataset de entrenamiento también de distinta manera. Para analizar una comparación vemos la Figura 4.9.

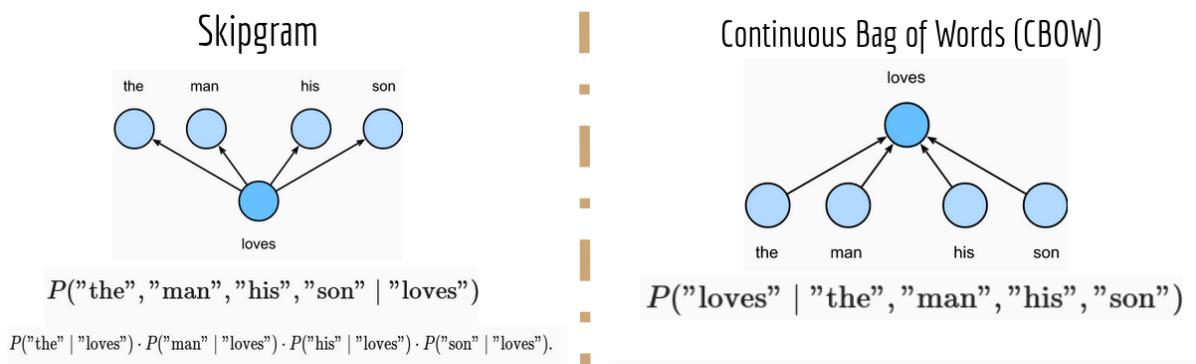


Figura 4.9: Skipgram y CBOW[72].

La diferencia principal entre CBOW y SkipGram es que en CBOW el objetivo del entrenamiento para obtener nuestros word embeddings es *predecir una palabra target de un texto en función de las palabras de contexto* (palabras vecinas a la palabra target), mientras que en SkipGram el objetivo del entrenamiento es *predecir las palabras de contexto en base a la palabra target*. Estas diferencias podemos observarlas en la Figura 4.9.

A continuación realizaremos una comparación entre las fórmulas matemáticas de Skipgram y CBOW.

- En **Skipgram** se calcula cuál es la probabilidad de las palabras de contexto (las que tratamos de adivinar / predecir) dada la palabra central. En el ejemplo de la Figura 4.9 calcula la probabilidad de las palabras “the”, “man”, “his” y “son”, dado que la palabra central es “loves”. Como vemos, esto se traduce en la multiplicación de cada una de las probabilidades condicionales individuales: “probabilidad de la palabra de contexto “the” dada la palabra central “loves” x la probabilidad de “man” dado “loves” x la probabilidad de “his” dado “loves” x la probabilidad de “son” dado “loves”. Esto termina siendo *dos productorias*: ver Fórmula 19.

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)}|w^{(t)}) \quad (\text{Ver [72]}) \quad (19)$$

*Cálculo de probabilidades mediante SkipGram.*

Teniendo en cuenta que:

- La *primer productoria* ( $\prod_{-m \leq j \leq m, j \neq 0}$ ) es aplicada sobre la “ventana de contexto” <sup>40</sup>: es decir,  $j$  va desde un  $-m$  hasta un  $+m$ , donde ese  $m$  es el tamaño de nuestra ventana (en este ejemplo es 2, por lo que tomamos 2 palabras a la izquierda y otras 2 a la derecha de la palabra central, obteniendo como palabras de contexto “the”, “man”, “his”, “son”). Y lo que se calcula en  $P(w^{(t+j)}|w^{(t)})$  es la probabilidad de la palabra de contexto  $w^{(t+j)}$  dada la palabra central  $w^{(t)}$ . De esta manera multiplicamos todas esas probabilidades y obtenemos la probabilidad de 1 palabra.
- Luego, en la *segunda productoria* ( $\prod_{t=1}^T$ ) multiplicamos por todas las otras palabras de nuestro corpus, ya que  $T$ =palabras de nuestro corpus.
- En cambio, en **CBOW** es al revés: se calcula cuál es la probabilidad de la palabra central (la que tratamos de adivinar / predecir) dado que tenemos todas las palabras de contexto. En el ejemplo de la Figura 4.9 calcula la probabilidad que la central sea “loves” dadas las palabras de contexto “the”, “man”, “his”, “son”. En este caso esto se traduce a *una sola productoria* (la que itera sobre las palabras de nuestro corpus de texto T): Fórmula 20.

$$\prod_{t=1}^T P(w^{(t)}|w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}) \quad (\text{Ver [72]}) \quad (20)$$

*Cálculo de probabilidades mediante CBOW.*

---

<sup>40</sup>Ventana de contexto (o Context Window, “C”) es una ventana de tamaño  $m$  que podemos colocar en cualquier parte de nuestro corpus de texto, y define qué tan largo o corto tomamos el contexto (palabras de contexto) para realizar el entrenamiento de nuestro modelo y obtener nuestra representación vectorial[71]. La cantidad de palabras a usar como tamaño de ventana es un *hiper-parámetro* de nuestro modelo que se define a partir de nuestros datos, y que discutiremos más adelante en la sección *Elección de hiper-parámetros Word2vec*

Entonces, ¿qué nos conviene elegir: CBOW o Skipgram?. Para responder a esto detallaremos las conclusiones obtenidas del paper original de Word2vec[60]:

- *Skipgram* necesita menor cantidad de datos y se encontró que representa mejor las palabras raras: debido a que Skipgram por cada palabra genera más ejemplos, las palabras que aparecen pocas veces (“raras” o infrecuentes) están mejor representadas en el dataset.
- *CBOW* funciona más rápido (ya que por lo que vimos anteriormente hace menos cuentas, solo una productoria) y tiene una mejor representación para palabras más frecuentes.

Es por esto que para nuestra implementación **se optó por utilizar SkipGram**; debido a que nuestro dataset de entrenamiento no es muy grande (nuestro vocabulario consiste en 60713 palabras) y necesitamos utilizar esta arquitectura que funciona mejor para pocos datos. Además, como mencionamos previamente, otra punto a favor de Skipgram es que nos permite representar mejor palabras “raras” o infrecuentes, y esto es una ventaja en IT ya que cada vez más cantidad de lenguajes y tecnologías van apareciendo, por lo que aunque una palabra aparezca pocas veces en nuestros corpus de texto, hay que tenerlas en cuenta y tratar de representarlas lo mejor posible.

El funcionamiento del modelo SkipGram para obtener nuestros word embeddings está detallado en la sección *Obteniendo nuestros Word Embeddings con Skipgram*. Previamente a esto, explicaremos el concepto de vectores de tipo *one hot encoding*, concepto necesario para entender el modelo Skipgram.

#### 4.5.4.5 One hot encoding.

*One hot encoding* es una forma de representación vectorial que consiste en representar a cada palabra como un vector único con un único elemento (o columna) siendo 1 y el resto de los elementos (o columnas) = 0. El producto interno de dos vectores de palabras diferentes siempre es 0, es decir, cada palabra siempre es *ortogonal* a todas las demás palabras[58].

Al igual que BoW y TF-IDF, one hot encoding también es una representación dispersa, de alta dimensionalidad y discreta: presentando todas las desventajas ya previamente detalladas en la sección *Desventajas BoW & TF-IDF*. Algo que se diferencia de BoW y TF-IDF es que estas son representaciones vectoriales para cada oración / documento de texto; en cambio one hot encoding, al igual que word embeddings, son representaciones vectoriales para cada *palabra* de la oración / documento de texto.

En la Figura 4.10 podemos observar la representación de 4 palabras como vectores one hot encoding. Mientras que en la Figura 4.11, considerando un tamaño de vocabulario de 1000, observamos la desventaja de ser una representación *dispersa* y de *alta dimensionalidad*: este vector tendrá 999 ceros y 1 solo uno.

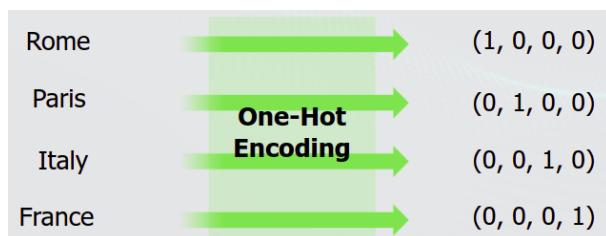


Figura 4.10: One Hot Encoding (1)[73].

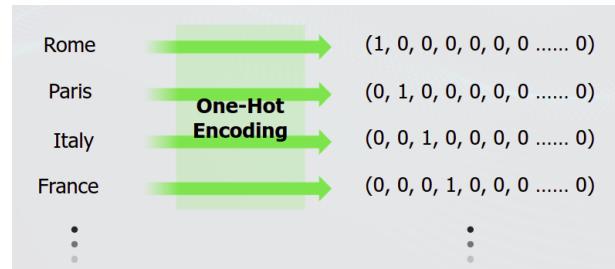


Figura 4.11: One Hot Encoding (2)[73].

En la Figura 4.12 se visualiza la otra desventaja de estos vectores: ser una representación *discreta*. De esta manera el vector que representa a la palabra no representa su significado ni permite capturar las relaciones entre las palabras, únicamente expresa su ubicación en un espacio N-dimensional (donde N es el tamaño del vocabulario). Como tenemos un vector para cada palabra, donde todos los vectores son todos distintos entre sí, cualquier par de vectores one hot es *ortogonal* (porque tienen un 1 en una dimensión distinta entre sí), por lo tanto no hay una noción natural de similitudes entre ellos: la palabra ‘Roma’ y ‘París’ son tan diferentes y tan parecidas como la palabra ‘Roma’ y ‘Baño’ (todas las palabras son igual de diferentes).

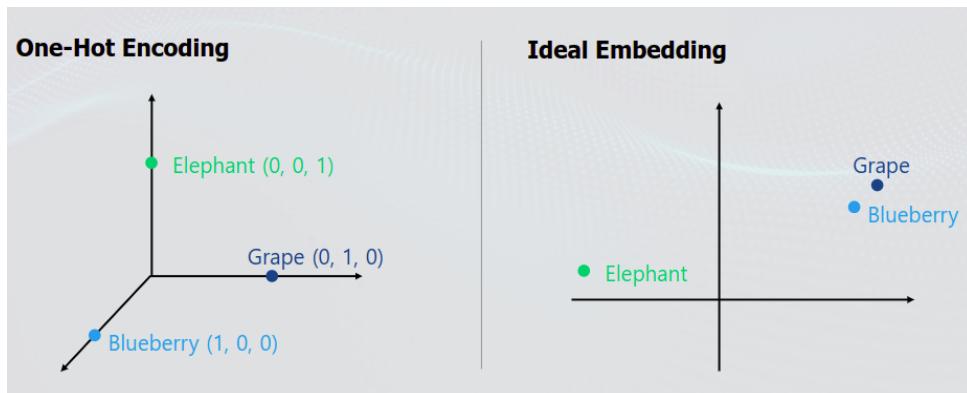


Figura 4.12: One Hot Encoding (3)[73].

Sin embargo, debido a que one hot encoding permite representar a nuestras palabras del vocabulario mediante vectores, aunque muy vagamente -ya que no permiten capturar las relaciones entre palabras-, estos vectores son utilizados para alimentar a nuestro modelo Word2vec (en la capa de entrada de la red neuronal) para obtener al final -luego del entrenamiento- nuestras representaciones densas (Word Embeddings).

#### 4.5.4.6 Obteniendo nuestros Word Embeddings con Skipgram.

Ya habiendo explicado previamente los conceptos básicos de redes neuronales y vectores de tipo one hot encoding, ahora explicaremos el modelo Skipgram, el cual utilizamos para entrenar y obtener nuestros Word Embeddings. Para esto, tomaremos como ejemplo la oración “The quick brown fox jumps over the lazy dog” como nuestro corpus de texto de ejemplo.

Como mencionamos previamente, el objetivo de Word2vec utilizando Skipgram es predecir cuál es la probabilidad que aparezcan todas las palabras de contexto a partir de la palabra central / dado que se conoce esta palabra central. Dicho de otra manera, la tarea principal en Skipgram consiste en construir un *modelo de lenguaje* que correctamente prediga las palabras de contexto dada la palabra central.

Un modelo de lenguaje es un modelo estadístico que trata de otorgar una distribución de probabilidad sobre secuencias de palabras; permitiendo así, por ejemplo, predecir cuál es la siguiente palabra. Dada una oración de  $m$  palabras, asigna una probabilidad  $Pr(w_1, w_2, \dots, w_m)$  a la oración completa. El objetivo del modelo de lenguaje es asignar probabilidades de forma que otorgue una alta probabilidad a oraciones “buenas” y bajas probabilidades a oraciones “malas”. Con “buenas” nos referimos a oraciones que son semántica o sintácticamente correctas; y por “malas” nos referimos a oraciones que son incorrectas semántica o sintácticamente o ambas<sup>41</sup>. De esta manera, para la oración “The cat jumped over the dog” tratará de asignar una probabilidad cercana a 1.0, mientras que para la oración “jumped over the the cat dog” asignará una probabilidad cercana a 0.0[71]. Este cálculo de probabilidades para Skipgram es el descrito anteriormente en la Fórmula 19.

Algo a destacar de los modelos de lenguajes es que tienen una gran *ventaja* sobre la mayoría de los otros modelos de aprendizaje automático. Esta ventaja es que tenemos un montón de datos de texto para entrenarlos: los podemos obtener de libros, artículos, contenido de millones de páginas web, etc. En otros modelos de aprendizaje automático, como en modelos de detección de imágenes, no tenemos “naturalmente” los datos, ya que generalmente necesitamos datos recopilados especialmente y que tienen un preprocessamiento de etiquetado manual.

Siguiendo con nuestro ejemplo, para nuestro corpus “The quick brown fox jumps over the lazy dog” utilizaremos un tamaño de ventana de contexto = 2. Nuestro primer objetivo es armar un *dataset* a partir del corpus para alimentar y entrenar al modelo, ya que como mencionamos anteriormente, el significado de las palabras vienen a partir de las *similitudes distribucionales*. Para esto realizamos una serie de iteraciones: en la Figura 4.13 podemos observar una iteración específica dentro del modelo Skipgram, donde a partir de la palabra central “jumped” (en azul), se intenta predecir/adivinar cada una de las palabras de contexto (en este caso “brown”, “fox”, “over” y “the”). Esto únicamente constituye un paso, Skipgram repite esto para cada palabra de nuestro corpus siendo la palabra central. Para esto *se desplaza la ventana de contexto*: vemos en la Figura 4.14 que para la siguiente iteración nos movemos una palabra a la derecha, cambiando de esta

---

<sup>41</sup>En el análisis sintáctico se analiza la sintaxis, que incluye la acción de dividir una oración en cada uno de sus componentes. En cambio, en el análisis semántico se busca entender el significado de la oración. Las palabras pueden tener múltiples significados, el objetivo en este caso es identificar el significado apropiado por medio del contexto de la oración.

manera nuestras palabras de contexto y nuestra palabra central (la cual pasa a ser “over”). Esta ventana se desplazará hasta recorrer todo el corpus de texto.

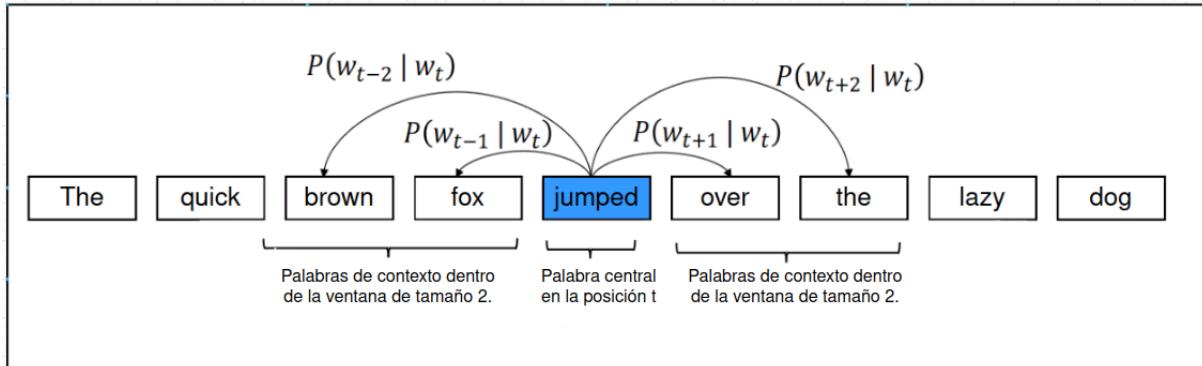


Figura 4.13: Skipgram: dada la palabra central se predicen las palabras de contexto.

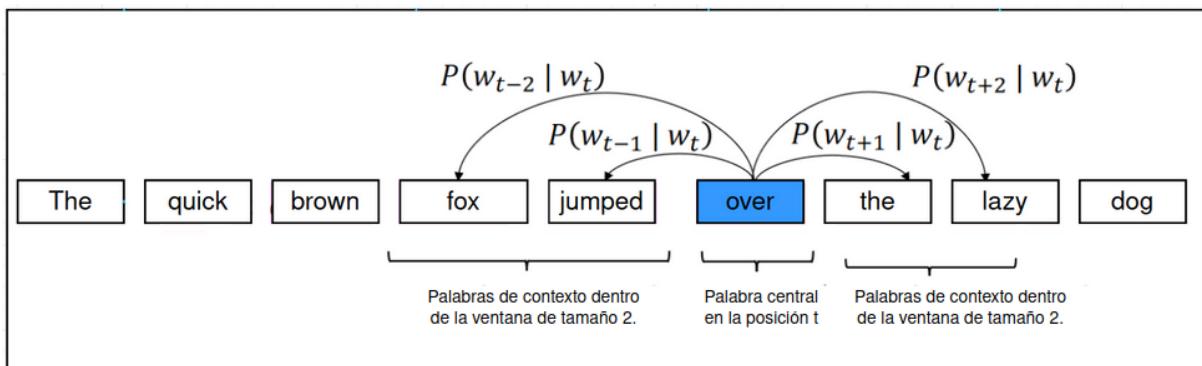


Figura 4.14: Skipgram: próxima iteración.

De esta manera se obtiene el *dataset* que será utilizado para entrenar Skipgram y obtener al final del entrenamiento nuestros embeddings. Explicado de una forma más general, podemos decir que para obtener nuestro dataset corremos una *ventana deslizante de contexto de tamaño C* sobre nuestro corpus de texto para obtener un set de  $2C + 1$  palabras que tendremos en consideración en cada paso. La palabra central en nuestra ventana es  $X$ , y las  $C$  palabras a cada lado de la palabra central son  $Y$ . Esto nos da  $2C$  puntos de datos. Un único punto de datos consiste en un par (*índice de la palabra central, índice de la palabra target o de contexto*). Luego deslizamos la ventana 1 palabra a la derecha del corpus y repetimos el proceso. De esta manera, deslizamos la ventana sobre todo nuestro corpus de texto para crear nuestro *dataset de entrenamiento*: Figura 4.15.

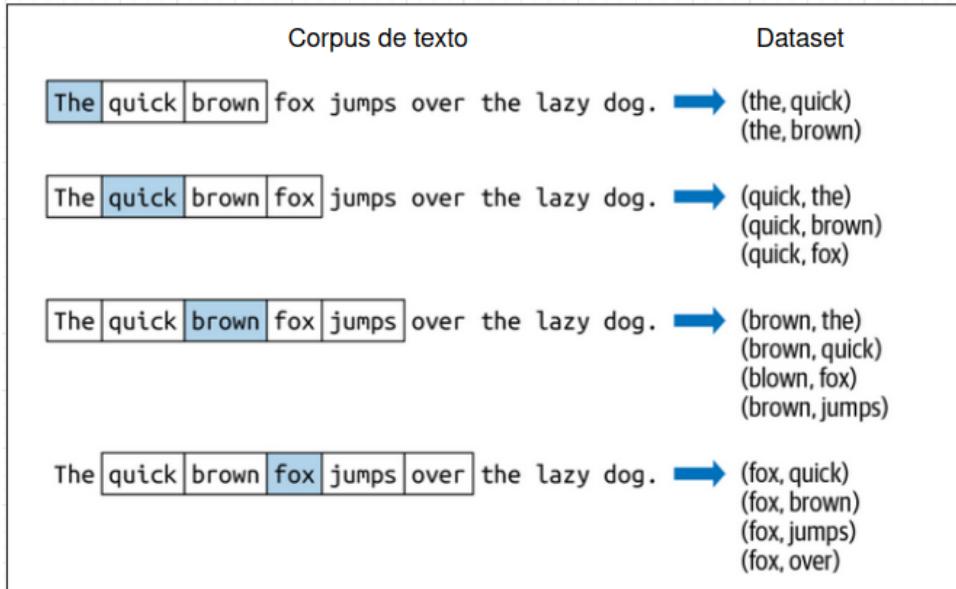


Figura 4.15: Preparando nuestro dataset para Skipgram (con  $C = 2$ )[71].

Como vemos en la imagen anterior, exceptuando los inicios y finales de nuestros corpus (donde las ventanas deslizantes no llegan a estar completas), podemos observar que dada una palabra central, la misma en el dataset va a aparecer  $2xC$  veces: por ejemplo, para la tercera iteración, brown aparecerá en el dataset  $2x2 = 4$  veces. Es por esto que con Skipgram se representan mejor las palabras infrecuentes / raras a comparación de CBOW: las palabras que aparecen pocas veces, en el dataset aparecen muchas veces.

En este dataset (Figura 4.15) la primera palabra de nuestro par (la palabra central) será tratada como la palabra de entrada a la red, y la segunda palabra de nuestro par (palabra de contexto) será tratada como la salida o posible resultado. Vemos que esto al final es una tarea de *clasificación*: dado que se sabe cual es esta palabra (input word) se debe predecir cuál palabra de contexto le sigue. de todas estas palabras de mi diccionario es la que sigue.

Por último, algo que puede generar curiosidad es: ¿qué sucede en el entrenamiento con las palabras que aparecen más de una vez en nuestro Corpus? Y ¿qué sucede con las palabras polisémicas<sup>42</sup>? Estas preguntas son respondidas en el Anexo: ver sección *Palabras repetidas en nuestro Corpus y palabras polisémicas*.

---

<sup>42</sup>Palabras que tienen más de un significado.

#### 4.5.4.7 Arquitectura del modelo Skipgram.

En la Figura 4.16 podemos observar: a la izquierda, la red neuronal de 2 capas utilizada para entrenar a nuestro modelo SkipGram, mientras que a la derecha, un “resumen” compactado de la misma utilizando el ejemplo de la iteración de la Figura 4.13. Observamos que nuestra arquitectura Skipgram tiene:

- Un vector de capa de entrada de  $V$  dimensiones -siendo  $V$  el tamaño de nuestro vocabulario-.
- Un vector de capa oculta de  $N$  dimensiones -siendo  $N$  el tamaño (dimensión) del embedding (de la matriz  $W$ )-.
- Un vector de salida también de  $V$  dimensiones.

Además, cabe destacar que la capa oculta *no posee una función de activación* y en la capa de salida se usa la *función softmax*.

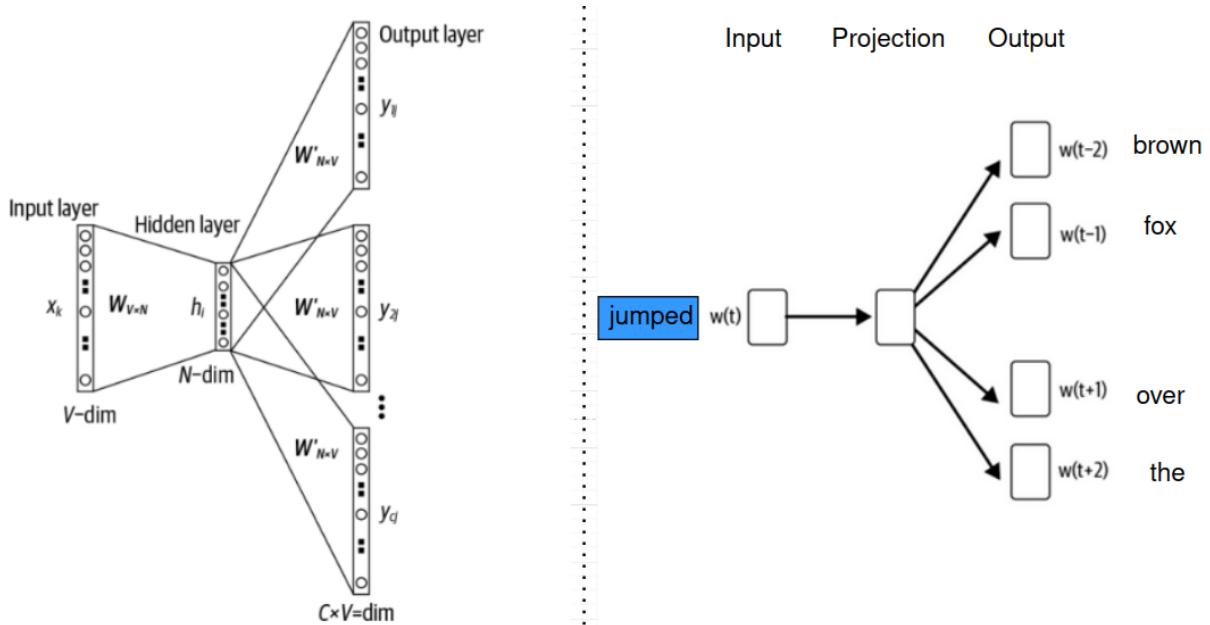


Figura 4.16: Arquitectura básica SkipGram[71].

En la Figura 4.17 podemos observar en mayor detalle la arquitectura de SkipGram. En este ejemplo se considera un tamaño de vocabulario = 6 ( $V$ ), un tamaño de embedding = 3 ( $N$ ) y un tamaño de ventana = 1 ( $C$ ). Como  $C = 1$ , tomaremos como palabras de contexto 1 palabra de cada lado de la palabra central.

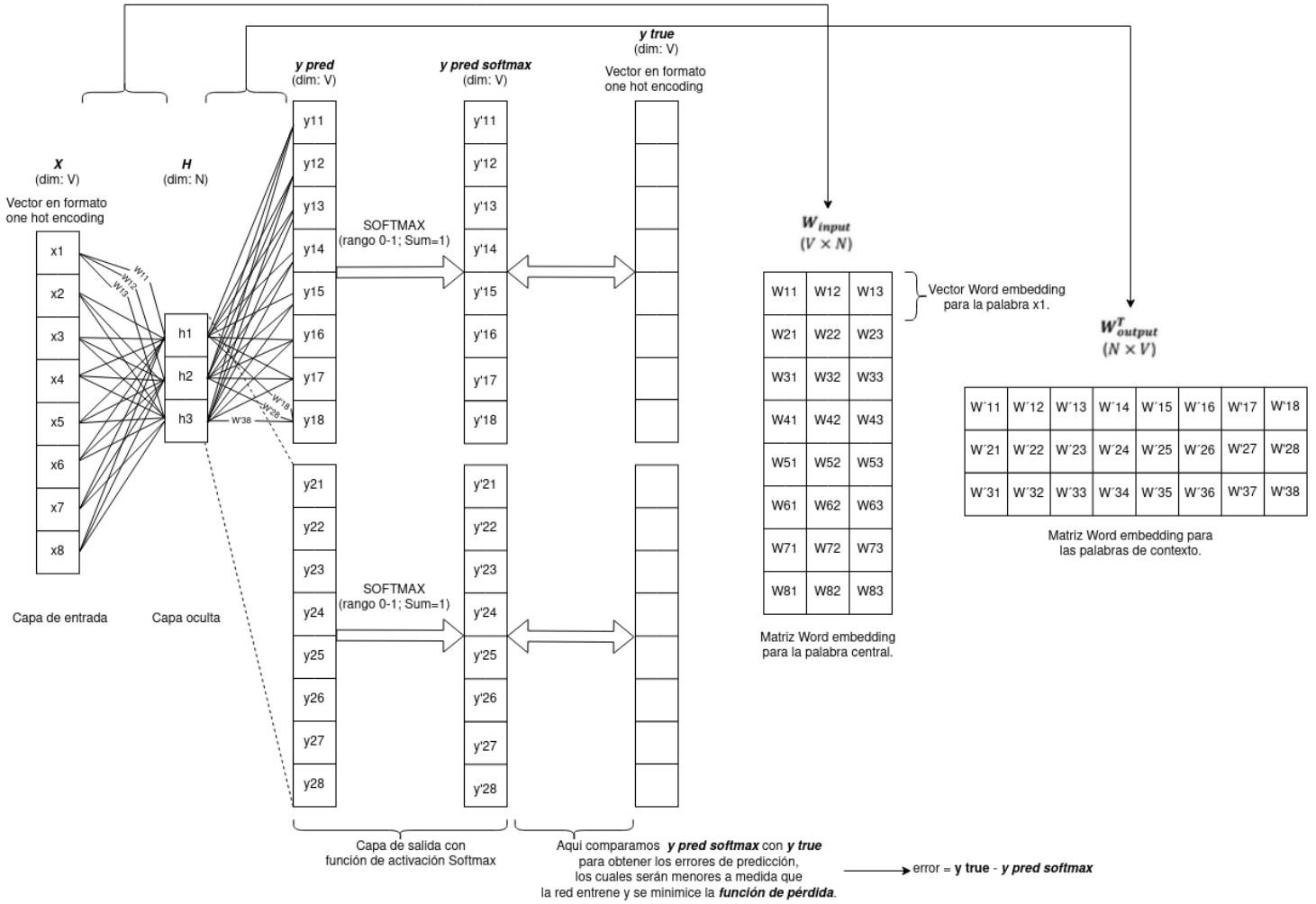


Figura 4.17: Arquitectura Skipgram con  $V = 8$ ,  $N = 3$ ,  $C = 1$ .

Debido a que  $C = 1$ , observamos 2 vectores de salida - $y_{pred}$ - e - $y_{pred softmax}$ . Sin embargo esto es simplemente para una mejor visualización, ya que la salida realmente para cada iteración es un único vector formada por 8 posiciones (o neuronas) debido a que nuestro modelo entrena con un par de palabras: primero (*central1, contexto1*) y luego (*central1, contexto2*) -considerando  $C = 1$ -.

El proceso de entrenamiento de esta red neuronal se explicará en detalle en la sección *Entrenamiento y función de costo con Softmax*. Lo que hay que tener en consideración, es que al final del mismo, *nuestro objetivo no es obtener la salida de la red; nuestro objetivo es extraer  $W$ , la cual será nuestra matriz de embedding que utilizaremos para representar los textos*. Por esta razón se considera que mediante Word2vec estamos realizando una “tarea falsa”, ya que realmente no estamos interesados en la predicción del modelo para la tarea en que fue entrenada, sino el subproducto (nuestros word embeddings) del modelo.

#### 4.5.4.8 Entrenamiento y función de costo con Softmax.

Como mencionamos previamente, la capa oculta *no posee una función de activación* y en la capa de salida se usa la *función softmax*. Además, las palabras que ingresan a la capa de entrada a lo largo de cada iteración de entrenamiento deben estar representadas en formato one hot encoding (ver sección *One hot encoding*).

Resumidamente, el entrenamiento de nuestro modelo Word2vec mediante la arquitectura Skipgram consiste en el ingreso de un par de palabras (*central, contexto*), donde la palabra central será puesta en la capa de entrada y la de contexto funcionará como target en la capa de salida. Luego mediante *forward propagation* se obtiene el valor de cada neurona de la capa de salida utilizando la función softmax. Este procedimiento anterior se repite hasta que todas las palabras de nuestro corpus hayan actuado como palabra central e ingresado a la red como un par (*central, contexto*). Al tener todas las salidas de cada una de las combinaciones palabra central y contexto, lo que se hace es comparar estas salidas predichas con las salidas reales correspondientes de la palabra de contexto / target, y utilizar *back propagation* para actualizar los parámetros de nuestro modelo (las matrices  $W$  y  $W'$ ) minimizando nuestra función de costo  $J$ . Este procedimiento que incluye forward propagation y back propagation se realizan *epoch* veces.

A continuación, describiremos en detalle este ciclo iterativo de procesos del entrenamiento. Como se trata de un modelo de red neuronal, este proceso general fue explicado anteriormente en la sección *Entrenamiento en una red neuronal*, solo que ahora adaptaremos la explicación para Skipgram. Al final de este ciclo de entrenamiento el objetivo es *extraer la matriz  $W$  (nuestra matriz de Word Embeddings)*.

El ciclo de entrenamiento utilizando Skipgram consiste en las siguientes fases:

1. **Inicialización aleatoria** por única vez de las matrices de pesos ( $W_{input}$  y  $W_{output}$  en nuestro caso), los cuales son los parámetros ( $\theta$ ) de nuestro modelo de red neuronal.
  2. **Forward propagation.** En este proceso se realizan los cálculos necesarios para obtener nuestras salidas -*y pred softmax*-.
- 2.1. *Forward propagation (capa de entrada a capa oculta):* En este paso se calcula el valor de cada neurona de la capa oculta. Para esto se utiliza la palabra central de entrada (la cual está codificada en formato one hot encoding) en conjunto con su fila correspondiente de la matriz de embedding  $W$  -de tamaño  $V \times N$ - (la cual representa los pesos de entrada de nuestra red). Como el vector de entrada únicamente es 1 para la palabra en cuestión (o dicho de otra forma, por cada iteración hay únicamente una neurona activa en la capa de entrada), entonces los valores de las neuronas de la capa oculta son directamente los valores de los pesos que corresponden a esa única neurona de la capa de entrada activa: por ejemplo, considerando que tenemos 3 neuronas en la capa oculta ( $N = 3$ ) y que nuestro vector de entrada es de 8 ( $V = 8$ ), nuestra primer entrada sería  $[1, 0, 0, 0, 0, 0, 0, 0]$ , por lo que el valor de la primera neurona de la capa oculta ( $h_1$ ) será de  $1 \times w_{11}$ ,  $h_2 = w_{12}$  y  $h_3 = w_{13}$ .
  - 2.2. *Forward propagation (capa oculta a capa de salida):* En este paso se calcula el valor de cada neurona de la capa de salida luego de pasar por la función de salida softmax (vector *y pred softmax*). Debido a que la capa oculta

no tiene función de activación, entonces los valores calculados en el paso 2.1 pasan directamente a la capa de salida. De esta manera, los valores de las neuronas de la capa de salida se calculan multiplicando el vector  $H$  que nos dió anteriormente con otra matriz  $W'$  -de tamaño  $N \times V$ -. Esto nos dará un vector  $y_{pred}$  de tamaño  $V$ , el cual alimentará a una función softmax para obtener una distribución de probabilidades sobre el espacio de nuestro vocabulario (- $y_{pred}$  softmax-).

Para cada una de las combinaciones entre palabras centrales y de contexto se realizan los pasos 2.1 y 2.2 hasta que todas las palabras de nuestro corpus hayan actuado como palabra central, obteniendo al final todas las salidas - $y_{pred}$  softmax- de cada una de las combinaciones palabra central y contexto.

3. **Cálculo de función de costo:** Luego de obtener las distribuciones - $y_{pred}$  softmax- (valor predicho) para cada combinación de palabras (*central, contexto*), estas son comparadas con los vectores de salida - $y_{true}$ - correspondientes (que tienen las etiquetas reales y al igual que los vectores de entrada están codificados en formato one hot encoding), computándose de esta manera el *error (o loss)* de clasificación (entre el valor predicho y el real) mediante el cálculo de la *función de costo (loss function)*  $J(\theta)$ : Fórmula 22 que veremos más adelante. Esta función de costo tiene en cuenta todas las palabras de nuestro texto  $T$ , y toma todos los pares correspondientes de palabras (*central, contexto*) dependiendo de nuestra ventana de contexto de tamaño  $m$ .
4. **Backpropagation:** Se aplica este proceso utilizando el algoritmo de optimización *descenso del gradiente* para poder ajustar / modificar las matrices  $W_{input}$  y  $W_{output}$  (que juntos forman  $\theta$ ) *con el objetivo de minimizar la función de costo*  $J(\theta)$  obtenida en el paso 3, y con ello minimizar el error de clasificación. No se detallará este procedimiento aquí porque no entra dentro de los objetivos de este trabajo. Sin embargo, lo que hay que tener en cuenta es que se utiliza la ecuación general 15 explicada anteriormente para actualizar los parámetros de nuestra red neuronal.
5. Volver al paso 2 hasta completar la cantidad de epochs seteados como *hiper-parámetro* de nuestro modelo.

Cabe destacar que este entrenamiento descrito tiene en cuenta el uso del *algoritmo de gradiente descendiente “básico”*. Con este método se calcula el error o función de costo luego de haber pasado todos los datos de entrenamiento a través de la red de una sola vez. De esta manera, luego de calcular forward propagation *para todos los datos de entrada*, se computa la función de costo y se aplica backpropagation realizando la actualización de los pesos. Todo este “bloque” de procedimiento se aplica por epoch. Por lo tanto, con este método de gradiente descendiente se calcula una función de costo y se realiza una sola actualización de los pesos por cada epoch.

Otra opción hubiera sido utilizar el algoritmo de *gradiente descendiente estocástico* (stochastic gradient descent, *SDG*). Con este método se calcula el error o función de costo luego de haber pasado solo un ejemplo de nuestros datos de entrenamiento a través de la red. De esta manera, por cada ejemplo de entrenamiento se calcula forward propagation, luego se computa la función de costo y se aplica backpropagation

realizando la actualización de pesos. Todo esto nuevamente se aplica por cada epoch. Si tuviéramos 100 ejemplos entonces calcularíamos 100 veces la función de costo y los pesos se actualizarian 100 veces. En la práctica este método resulta más rápido que el gradiente descendiente “básico”. Sin embargo, SDG no se utilizó en este proyecto debido a que por defecto no se encuentra la opción de setear el algoritmo de gradiente descendiente en la implementación de Word2vec de gensim, la cual se basa en el paper original de Word2vec[60] donde describe la función de costo mediante el uso de GD básico, teniendo en cuenta el entrenamiento de todas las palabras de nuestro texto  $T$  dentro de todas las ventanas de contexto. En caso de utilizar SDG, la función de costo debería expresarse individualmente por cada palabra de nuestro texto  $T$  teniendo en cuenta las combinaciones dentro de la ventana de contexto de tamaño  $m$ : Fórmula 23 que veremos más adelante.

Para comprender mejor las iteraciones de entrenamiento, se recomienda consultar el Anexo (sección *Ejemplo de obtención de Word Embeddings mediante skipgram y softmax*) donde se detalla un ejemplo concreto del entrenamiento del modelo Skip gram con función de salida softmax utilizando el algoritmo de gradiente descendiente “básico”. En este ejemplo se toma a la oración “The quick brown fox jumps over the lazy dog” como nuestro corpus de texto con el cual entrenará el algoritmo.

Como podemos observar, todo este entrenamiento tiene como objetivo minimizar la *función de costo* para realizar una correcta predicción. La *función de costo*  $J(\theta)$  de Skipgram se basa en la *función de verosimilitud*  $L(\theta)$ , la cual es prácticamente la que vimos en la Fórmula 19. La reescribimos como la Fórmula 21: aquí se calcula, para cada posición del texto  $t = 1, \dots, T$  (donde  $T$  es nuestro corpus de texto) y para cada ventana de contexto de tamaño fijo  $m$ , cuál es la probabilidad de que una palabra  $(w^{(t+j)})$  sea contexto de una palabra central  $(w^{(t)})$ , dada esa palabra  $(w^{(t)})$  y los parámetros del modelo  $(\theta)$  -que en nuestro caso son las matrices de pesos  $W$  y  $W'$ . De esta manera, con  $L(\theta)$  obtenemos la probabilidad de generar todas las palabras de contexto dada cualquier palabra central.

$$L(\theta) = \prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} | w^{(t)}; \theta) \quad (\text{Ver [72]}) \quad (21)$$

*Función de verosimilitud Skipgram.*

A  $L(\theta)$  se le aplica el logaritmo y se le coloca un menos por delante para que de esta manera sea minimizable, obteniendo así la función de costo a minimizar  $J(\theta)$  (Fórmula 22), la cual es la *verosimilitud logarítmica negativa media*. La mejor minimización de esta función la obtendremos cuando la probabilidad de que aparezca la palabra  $w^{(t+j)}$  dada  $w^{(t)}$  sea 1 (ya que el logaritmo de 1 es 0, siendo este el valor mínimo). Cuando el valor de la probabilidad es menor que 1 los valores se hacen muy negativos, pero al colocarle el menos se hacen muy positivos. Además, algo a tener en cuenta es que transformamos las *productorias* de la Fórmula 21 en *sumatorias* en la Fórmula 22; ya que le estamos aplicando el logaritmo a la función  $L(\theta)$ , y el logaritmo del producto es la suma de los logaritmos.

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} | w^{(t)}; \theta) \quad (\text{Ver [72]}) \quad (22)$$

*Función de costo Skipgram.*

Como mencionamos anteriormente, esta fórmula es teniendo en cuenta nuestra implementación con el algoritmo de gradiente descendiente “básico”. Si quisieramos implementar el algoritmo SGD, nuestra función de costo quedaría representada como la Fórmula 23, la cual se calcularía luego de realizar cada una de las iteraciones individuales de las palabras  $w(t)$  teniendo en cuenta cada posición de la ventana de contexto.

$$J(\theta; w^{(t)}) = - \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} | w^{(t)}; \theta) \quad (\text{Ver [72]}) \quad (23)$$

*Función de costo Skipgram utilizando SGD.*

Ahora, ¿cómo calculamos la probabilidad  $P(w^{(t+j)} | w^{(t)}; \theta)$ ? Vamos a omitir los parámetros de nuestro modelo ( $\theta$ ) para simplificar la notación. Utilizaremos dos vectores densos por cada palabra  $w$ :

- $v_w$ : vector de tamaño  $d$  para simbolizar cuando  $w$  es la palabra central  $w_c$ , donde  $d$  es el tamaño de palabras de nuestro vocabulario.
- $u_w$ : vector de tamaño  $d$  para simbolizar cuando  $w$  es una palabra de contexto  $w_o$ .

De esta manera, en la Fórmula 24 se describe cómo obtener la probabilidad de que  $w_o$  sea una palabra de contexto (con índice  $o$  en nuestra secuencia de textos T) dada una palabra central  $w_c$  (con índice  $c$  en el diccionario del corpus). El cálculo de la misma es similar a la que se usa cuando queremos calcular la probabilidad de salida de distintas clases en clasificación utilizando la función softmax (ver Fórmula 13) donde se calcula la probabilidad de todas las clases sumadas y por cada probabilidad se divide la misma, obteniendo así una *distribución* (ya que la suma de todas las probabilidades resultantes como salida dan 1).

$$P(w_o, w_c) = \frac{\exp(u_o^T v_c)}{\sum_{W \in V} \exp(u_W^T v_c)} \quad (\text{Ver [72]}) \quad (24)$$

*Probabilidad de que  $w_o$  sea una palabra de contexto dada una palabra central  $w_c$  (función de salida softmax).*

Teniendo en cuenta que:

- $u_o$  = vector de la palabra de contexto ( $o = \text{outside}/\text{contexto}$ ).
- $v_c$  = vector de la palabra central ( $c = \text{central}$ ).

- $V$  = nuestro vocabulario (palabras únicas de nuestro corpus de texto  $T$ ).
- En el denominador se realiza la sumatoria de todo el vocabulario de tamaño  $V$  para obtener la distribución de probabilidad.
- La exponencial se coloca para que siempre sea positiva (ya que las probabilidades tienen que ser siempre positivas).

#### 4.5.4.9 Optimizaciones: Muestreo Negativo.

Como podemos observar para el cálculo de  $P(w_o|w_c)$  (Fórmula 24) -la cual se realiza para obtener nuestra función de costo  $J(\theta)$  (Fórmula 22)- depende del tamaño de nuestro vocabulario  $V$  (ya que en el denominador de  $P(w_o|w_c)$  se realiza una sumatoria por cada palabra  $w$  perteneciente a  $V$ ). Debido a que nuestros vocabularios pueden ser miles o incluso millones de palabras, el costo de realizar el cálculo de los gradientes puede ser muy alto. Para reducir tal complejidad computacional y optimizar el cálculo de la función de costo explicada previamente se puede utilizar otro método de entrenamiento conocido como *muestreo negativo* (o *negative sampling*), el cual está descrito en el segundo Paper de Tomas Mikolov[64]. como una mejora al Skipgram tradicional con función de salida softmax.

Como podemos observar en la Figura 4.18, anteriormente utilizando softmax (parte izquierda de la imagen) teníamos un problema de *clasificación*, donde teníamos una palabra central (en este ejemplo “not”) como *entrada* y teníamos que clasificar cuál de todas las palabras del vocabulario era la más probable de que fuera la palabra de contexto de *salida* (en este ejemplo “thou”). El problema es que, como vimos anteriormente, la clasificación con softmax es un problema debido a su elevado costo de cálculo; con lo cual la idea es transformar esto a *otro problema de clasificación* pero usando la *regresión logística* (parte derecha de la imagen). De esta manera, mediante muestreo negativo se emplean nodos de regresión logística (es decir, con la función sigmoide -Fórmula 14-) con el objetivo de clasificar de la mejor manera posible, ejemplos positivos y negativos de nuestro conjunto de entrenamiento[63]. En este modelo de ejemplo nos dan como *entrada* las 2 palabras (palabra central: “not”, y palabra de contexto: “thou”) y a la *salida* nos fijamos cual es la probabilidad de que esas dos palabras estén juntas (en este caso arrojó como resultado “0.9”). En conclusión, pasamos de un problema de “Dada una palabra central ¿cuál de todas estas palabras es la palabra de contexto?” a “Dado un par de palabras (central, contexto) ¿cuál es la probabilidad de que estén juntas?”. Ahora nuestra salida no es una palabra, sino un número del 0 al 1: en lugar de usar la softmax, ahora usamos la sigmoide, la cual va de 0 a 1.



Figura 4.18: Comparación Skipgram con distintas funciones de salida: softmax (izquierda) y sigmoide (derecha)[74].

De esta manera, mediante muestreo negativo pasamos de usar la Fórmula 24 para calcular la probabilidad de la palabra  $w_o$  dada  $w_c$  (que representa la probabilidad  $P(w^{(t+j)}|w^{(t)}; \theta)$ ), a utilizar la Fórmula 25, la cual calcula la probabilidad de que la salida sea 1, dadas las 2 palabras  $w_c$  y  $w_o$ .

$$P(D = 1|w_c, w_o) = \sigma(u_o^T v_c) \quad (\text{Ver [72]}) \quad (25)$$

*Probabilidad de que la salida sea 1, dadas  $w_o$  y  $w_c$  (función de salida sigmoide).*

Reescribimos la fórmula descrita anteriormente para la función sigmoide (Fórmula 14) para adaptarla a nuestro caso: Fórmula 26.

$$\sigma(u_o^T v_c) = \frac{1}{1 + \exp(-(\sigma(u_o^T v_c)))} \quad (\text{Ver [72]}) \quad (26)$$

*Función sigmoide utilizada.*

En la Figura 4.19 podemos observar un ejemplo de construcción de datasets para utilizar como entrenamiento: usando como función de salida softmax (parte izquierda) y sigmoide (parte derecha). De esta manera pasamos de tener un dataset donde la entrada y salida son las dos palabras, a uno donde tengamos las dos palabras como features, y la etiqueta “target” es cual es el valor que debería arrojar el modelo.

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

Figura 4.19: Armando el datasets para distintas funciones de salida: softmax (izquierda), sigmoide (derecha)[74].

El problema del dataset descrito anteriormente (utilizando sigmoide) es que los ejemplos son siempre positivos: tenemos dos palabras y la respuesta/salida siempre es 1. Si dejamos todas las etiquetas en 1 el modelo rápidamente va a aprender que prediciendo 1 está en lo correcto y tendrá un 100 % de efectividad.

Aquí entran los ejemplos negativos (*muestreo negativo*): tenemos que agregar palabras que no vayan juntas para que el modelo aprenda que esas palabras no van juntas. De esta manera, mediante el muestreo negativo (Figura 4.20) lo que hacemos es que para cada ejemplo positivo agarramos *palabras al azar* del vocabulario mediante “random

*sampling*<sup>43</sup> (como “aaron” y “taco”), las metemos en el dataset y le colocamos etiqueta 0. Es decir, por cada ejemplo positivo estamos metiendo (en este caso) 2 negativos.

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

Figura 4.20: Armando el datasets para función de salida sigmoide usando negative sampling[74].

¿Cuántas palabras negativas ponemos por cada positiva?: esto es otro *hiper-parámetro* de nuestro modelo Word2vec.

Con esta optimización, transcribimos la Fórmula 25 anterior para describir la probabilidad de que la salida sea 1, dadas las 2 palabras  $w_c$  y  $w_o$ , a la Fórmula 27.

$$P(w^{(t+j)}|w^{(t)}) = P(D = 1|w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim P(w)}^K P(D = 0|w^{(t)}, w_k) \quad (\text{Ver [72]}) \quad (27)$$

Probabilidad de que  $w^{(t+j)}$  sea palabra de contexto de la palabra central  $w^{(t)}$  (función de salida sigmoide y uso de negative sampling).

Teniendo en cuenta que:

- $P(D = 1|w^{(t)}, w^{(t+j)})$  es la probabilidad de que la salida sea 1 cuando las dos palabras estén juntas  $w^{(t+j)}$  y  $w^{(t)}$ .
- $\prod_{k=1, w_k \sim P(w)}^K P(D = 0|w^{(t)}, w_k)$  es la probabilidad de que la salida sea 0 cuando las dos palabras estén juntas ( $w^{(t)}$  y la palabra central  $w_k$ , que es la que agarramos mediante random sampling). El  $K$  en esta ecuación define cuántas palabras negativas tenemos en cuenta.

De esta manera, observamos que la Fórmula 27, no depende de todo el vocabulario  $V$ , como sí sucedía en la Fórmula 24. La Fórmula 27 solo depende de las palabras negativas  $w_k$  que metamos: por cada ejemplo vamos a tener que iterar sobre esas  $K$  palabras en lugar de iterar sobre todo el diccionario *para calcular la función de pérdida* (Fórmula 22).  $K$  es mucho menor que el vocabulario  $V$ : generalmente estamos hablando de 5 a 10 ejemplos negativos por cada positivo, y 10 ya es mucho más pequeño que 1 millón (considerando 1 millón como el tamaño de nuestro vocabulario).

---

<sup>43</sup>En el *random sampling* lo más óptimo es agarrar palabras muy improbables de nuestro vocabulario para que nuestro modelo aprenda que no tiene que poner esas palabras.

#### 4.5.4.10 Entrenamiento y función de costo con Sigmoid.

El ciclo de entrenamiento del modelo Word2vec con función de salida sigmoidal y utilizando negative sampling es similar al descrito en la sección *Entrenamiento y función de costo con Softmax*. La diferencia es que la función de salida es sigmoidal en lugar de softmax, y la función de coste  $J$  será distinta; el resto de los pasos indicados anteriormente para el entrenamiento son los mismos.

Resumidamente, para realizar el entrenamiento de nuestro modelo Word2vec con función de salida sigmoidal y utilizando negative sampling, el primer paso es generar nuestras matrices  $W$  y  $W'$ : Figura 4.21. Para esto se generan los 2 vectores que vimos anteriormente ( $v_w$  -vector de la palabra central-, y  $u_w$  -vector de la palabra de contexto-) por cada palabra de nuestro vocabulario. Con estos vectores se crean dichas matrices: la *matriz de embedding* ( $W$ ) cuando la palabra es la central; y la *matriz de contexto* ( $W'$ ) cuando la palabra es de contexto. Ambas matrices tienen *el mismo tamaño*, por lo que tienen una representación de embedding para cada palabra de nuestro vocabulario. Las filas son las palabras de nuestro vocabulario y las columnas es el tamaño del embedding: otro *hiper-parámetro* de nuestro modelo Word2vec.

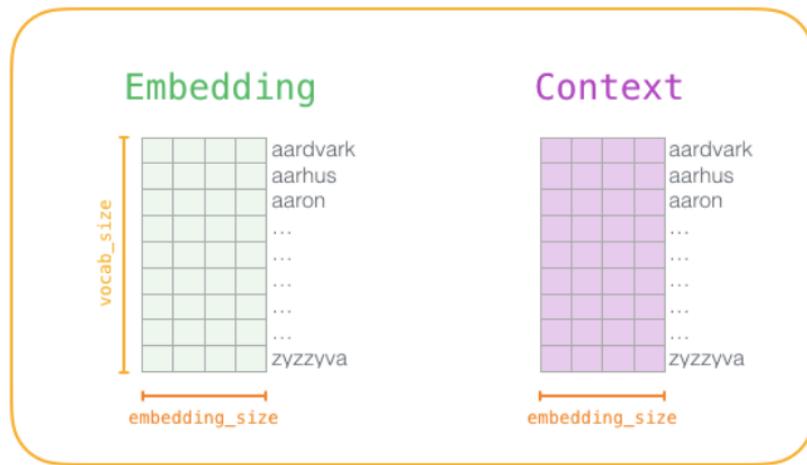


Figura 4.21: Primer paso entrenamiento Word2vec con función de salida sigmoidal y utilizando negative sampling[74].

Luego, viendo la Figura 4.22, observamos que el modelo agarra el primer grupo de palabras (en este caso son 3 filas: una fila es la palabra central junto con la palabra de contexto correcta -thou-, y las otras dos filas son la palabra central junto con las palabras de ejemplo negativo -aaron y taco-). Luego el modelo busca la palabra central en la matriz de embedding junto con todas las de contexto en la matriz de contexto, y las separa.

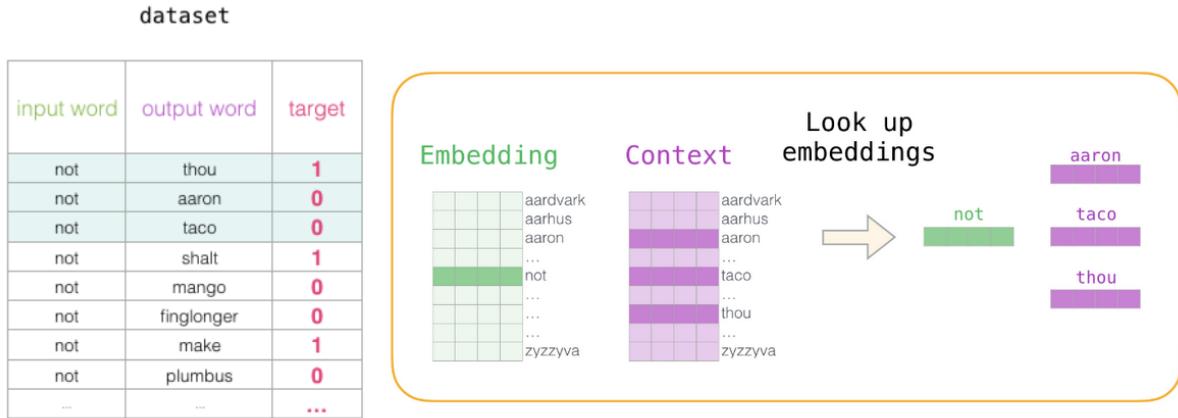


Figura 4.22: Obtención de vectores para el cálculo de forward propagation[74].

Teniendo las etiquetas de estas 3 salidas, lo que hace el modelo es calcular la salida mediante forward propagation, y por último aplicar la función de salida sigmoide. Esto se hace para *todos los ejemplos de entrenamiento*. Una vez finalizado el cálculo de las salidas calculamos el error total, los cuales surgen de la comparación de las salidas de la sigmoide con las etiquetas correspondientes aplicando la función de costo  $J$  descrita en la Fórmula 28. En esta función de costo observamos que ya no dependemos del tamaño de vocabulario  $T$ , sino que dependemos de nuestros  $k$  ejemplos negativos. Estos errores los usamos para modificar los embeddings de las matrices mediante backpropagation utilizando gradiente descendiente: Figura 4.23.

$$J(\theta) = -\log P(w^{(t+j)}|w^{(t)}) = -\log \sigma(u_{i_{t+j}}^T v_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log \sigma(-u_{h_k}^T v_{i_t}) \quad (28)$$

Función de costo Skipgram (función salida Sigmoide y uso de negative sampling)[72].



Figura 4.23: Forward y Back propagation[74].

Mientras se realiza el entrenamiento recorriendo epoch veces nuestro conjunto de datos, los embeddings continúan mejorando su representación para cada una de las palabras. Como mencionamos previamente, estos embeddings son entrenados para que las palabras que siempre están cerca, aparezcan cerca; y por lo tanto en el proceso de entrenamiento el modelo aprenda a codificar estas palabras para que se mantenga el significado de las mismas.

Una vez que el modelo finalizó el entrenamiento obtendremos nuestras 2 matrices / embeddings listas y lo que se hace es descartar la matriz de contexto  $W'$  y *usar la matriz de Embeddings de palabra central  $W$  como nuestros embeddings pre-entrenados* para representar a nuestras palabras.

#### 4.5.4.11 Desventajas Word2Vec.

Existen tres grandes desventajas de utilizar embeddings obtenidos mediante Word2vec, las cuales describiremos a continuación.

- La primer limitación[73] de Word2vec es que sus embeddings *no permiten trabajar con palabras desconocidas por nuestro vocabulario* (palabras *out-of-vocabulary, OVV*). De esta manera, si el embedding no contiene en su vocabulario una palabra, la misma simplemente no se tendrá en cuenta. Esta limitación puede ser solventada utilizando word embeddings obtenidos mediante el modelo *FastText*, el cual es una variante de Word2Vec creada por Facebook que si permite manejar palabras OOV. Esto ocurre debido a que Word2vec considera a una palabra como una unidad que no puede ser dividida, mientras que Fasttext considera una palabra como una combinación de sub-palabras. En FastText cada palabra es dividida en n-grams (o “sub-palabras”), los cuales están compuestos por n caracteres. El valor de n define cómo se divide cada palabra en cierto número de n-gramas. Por ejemplo, si  $n = 3$ , “apple” se divide en “app”, “ppl” y “ple”. Una vez que se obtengan los valores vectoriales de las sub-palabras, el valor vectorial de “apple” se convierte en la suma de los valores vectoriales de las sub-palabras. Podemos observar este ejemplo en la Figura 4.24.

Mediante esta técnica de entrenamiento FastText permite el manejo de palabras OOV. Por ejemplo, consideremos un caso en el que se cuentan en nuestro vocabulario de embedding las palabras “birth” (nacimiento) y “place” (lugar), pero no la palabra “birthplace” (lugar de nacimiento). En Word2vec, al no estar definido el valor del vector “birthplace”, entonces este no se podrá obtener de ninguna forma. En cambio, FastText puede calcular el valor del vector “birthplace” mediante la unión de las subpalabras “birth” y “place”, las cuales si están en nuestro vocabulario. Podemos observar este ejemplo en la Figura 4.25.

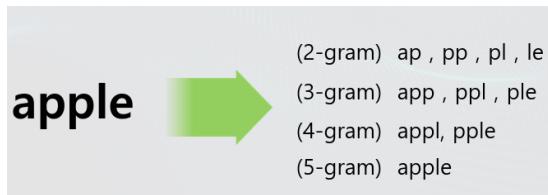


Figura 4.24: Representación de ‘apple’ mediante FastText[73].



Figura 4.25: Word2vec (izquierda) / FastText (derecha)[73].

- La segunda desventaja[73] de Word2vec es que *los resultados del embedding no son buenos frente a las palabras que tienen errores tipográficos (datos que presentan ruido)*. Esta limitación también puede ser solventada utilizando FastText. Por ejemplo, consideremos la palabra “apple” y “appple”, que tiene una p adicional como un error tipográfico. En Word2Vec, las dos palabras se consideran palabras completamente separadas. Sin embargo, para FastText, dado que las dos palabras tienen varios n-gramas idénticos, se considerarán palabras similares. Podemos observar este ejemplo en la Figura 4.26.

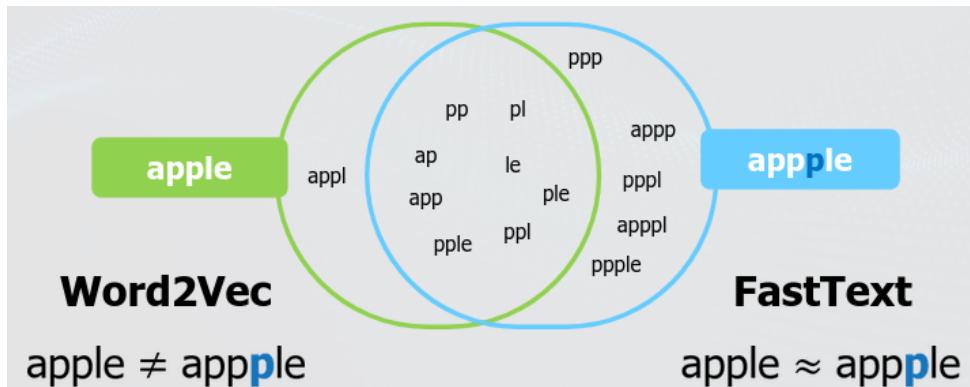


Figura 4.26: Word2vec / FastText[73].

Como mencionamos previamente, las limitaciones 1 y 2 pueden solventarse utilizando Fasttext. Sin embargo, en nuestra implementación no utilizamos este algoritmo debido a su mayor complejidad de cálculo, ya que para obtener el vector de una palabra en el proceso de entrenamiento tenemos que realizar la *suma de todos los vectores de subpalabras*.

- La tercer y más importante desventaja de usar embeddings obtenidos mediante Word2Vec[71], es que las relaciones construidas entre las palabras son puramente estadísticas y únicamente reflejan la proximidad en un espacio vectorial. Esto es una desventaja, ya que el objetivo de NLP justamente es recrear complejas relaciones que dejen de lado las representaciones estadísticas. De esta manera, muchas veces no nos es útil esta representación para representar palabras que se usan en distintos contextos. Esto es debido a que Word2vec obtiene un embedding que aprende un vector **fijo** para cada palabra. El inconveniente acá es que existen palabras con diversos significados, por ejemplo la palabra ‘queen’ puede ser usada en una oración de piezas de ajedrez o en una oración de tamaños de cama. El vector para la palabra ‘queen’ debería ser distinto dependiendo el contexto, pero en el caso de word2vec solo existe un único vector ‘queen’ que, en caso de haber sido entrenado con ambas oraciones, supondrá un significado ‘promedio’ entre los 2 contextos.

Esta tercer limitación se puede solventar mediante el uso de modelos de lenguaje más complejos basados en deep learning que permiten obtener embeddings contextuales (como *BERT*, o *ELMo*), permitiendo variar los embeddings dependiendo del contexto. Sin embargo, en nuestra implementación no utilizamos estos modelos debido a que el entrenamiento de los mismos requiere mucho mayor tiempo debido a su mayor complejidad de cálculo[52]. Otra razón por la cual no utilizamos estos modelos es porque tendríamos que realizar pasos extra para adaptar / transformar la salida de los mismos de modo que sean compatibles con la tarea que queremos realizar a posterior: obtener las similitudes de nuestras palabras en documentos/textos mediante el método Word Mover's Distance (WMD). Para utilizar WMD como técnica de medición de similitud, lo más simple es utilizar los embeddings obtenidos mediante Word2Vec, como está descrito en el paper de Kusner[65].

## 4.6 Obtención de las mediciones de similitud entre textos.

El siguiente paso de nuestra aplicación NLP consiste en definir y emplear una métrica de distancia que será utilizada como nuestra medida de similitud entre los vectores numéricos obtenidos del paso anterior (sean mediante BoW, TF-IDF o Embeddings), los cuales representan nuestros textos a comparar.

La medición de similitudes entre textos (text similarity measurement) es uno de los problemas más cruciales del *Procesamiento del Lenguaje Natural (NLP)*. Encontrar similitudes entre documentos se utiliza en varios dominios de NLP, tales como en sistemas de recomendación, information retrieval, automatic question answering, machine translation, dialogue systems, y document matching[4].

Una de las primeras aplicaciones de mediciones de similitud entre textos es quizás el modelo vectorial en la recuperación de información (information retrieval, IR), donde el documento más relevante para una consulta de entrada se determina clasificando los documentos de una colección en orden inverso a su similitud con la consulta dada. Las mediciones de similitud entre textos también se utilizaron en sistemas de retroalimentación de relevancia (relevance feedback) y de clasificación de texto (text classification). También se utilizaron para tareas de desambiguación del sentido de la palabra (word sense disambiguation) y, más recientemente, para resúmenes de texto (text summarization) y para métodos de evaluación en máquinas de traducción automática (machine translation)[67].

Para realizar esta tarea de obtener las mediciones de similitud entre nuestros textos utilizaremos dos métodos: uno más convencional (Cosine Similarity, representando previamente nuestros textos con TF-IDF) y un método más actual y novedoso (WMD o Word Mover's Distance, representando previamente nuestros textos mediante Embeddings). En la sección *¿Por qué decidimos utilizar Cosine similarity y WMD?* se detallan las razones por las cuales se decidieron elegir estos métodos; las cuales están justificadas en base al análisis realizado en *Maneras de medir la similitud entre textos*, el cual detalla las numerosas técnicas existentes para obtener similitudes entre textos.

Como mencionamos previamente, una vez que hayamos obtenido estas mediciones de similitud usando estos dos métodos, lo que haremos es combinar dichas mediciones e

ingresarlas como entradas a algoritmos de ML (en nuestro caso K-means, y luego KNN) para generar *un modelo de clasificación*. Lo que hará este modelo es, en base a los valores de similitud de los curriculums vitae de nuevos candidatos, lograr clasificar qué tan similares son dichos curriculums vitae con respecto a la descripción de un puesto de IT: similitud escasa, similitud media, similitud alta, similitud muy alta.

A continuación se describirán las técnicas existentes más populares para obtener mediciones de similitud entre textos, haciendo énfasis en las dos técnicas utilizadas en este proyecto.

#### 4.6.1 Maneras de medir la similitud entre textos.

Los textos pueden ser similares de dos maneras: *léxica* y *semánticamente*. Las palabras que componen el texto son similares léxicamente si tienen una secuencia de caracteres similar. En cambio, la similitud semántica determina la similitud entre textos basados en los significados de las palabras que la componen, en lugar de un macheo carácter por carácter. Las palabras son similares semánticamente si tienen algo en común, son opuestas entre sí, se usan de la misma manera, se usan en el mismo contexto y si una es un tipo de otra[66].

La similitud de texto no sólo da cuenta de la similitud semántica entre textos, sino que también considera una perspectiva más amplia que analiza las propiedades semánticas compartidas de dos palabras. Por ejemplo, las palabras 'King' y 'Man' pueden estar estrechamente relacionadas entre sí, pero no se consideran semánticamente similares, mientras que las palabras 'King' y 'Queen' si son semánticamente similares. Así, la similitud semántica puede considerarse como uno de los aspectos de la relación semántica. La relación semántica que incluye la similitud se mide en términos de distancia semántica, que es inversamente proporcional a la relación[4].

Existen muchas métricas para medir similitudes entre textos. Además, existen muchas técnicas para, previamente, representar vectorialmente a nuestros textos. Es por esto que podemos realizar un gran número de combinaciones entre estos métodos de vectorización y uso de métricas de similitud para llevar a cabo nuestra tarea de medición de similitudes entre textos. Esta misma separación fue tomada en cuenta por un estudio publicado en agosto de 2020[4], el cual provee un profundo análisis y conocimiento de las actuales técnicas para medir similitudes entre textos.

En la Figura 4.27 podemos observar un resumen de dicho estudio, en la cual el cálculo o medición de similitudes entre textos es descrita teniendo en cuenta los dos aspectos mencionados:

- *Distancias entre los textos (text distance)*: la cual describe la proximidad semántica de dos palabras de texto desde la perspectiva de la distancia.
- *Representaciones de los textos (text representation)*: la cual representa al texto como features o valores numéricos que pueden ser calculados.

Se han propuesto varias medidas y técnicas para medir la similitud semántica durante las últimas décadas. La mayoría de los académicos y estudios previos a este dividen los

métodos de medición de similitud de texto basados en corpus de textos (corpus based similarity) y basados en conocimiento (knowledge based similarity)[66]. Esta clasificación ignora los métodos de cálculo de distancias entre textos, y sólo considera los métodos de representación de textos. Es por esto que este estudio[4] hace una mayor extensión y subdivisión del sistema de clasificación. Adicionalmente, debido al desarrollo y auge de las redes neuronales en estos últimos años, surgieron varios métodos de representación y cálculos de distancias basándose en estas tecnologías, las cuales fueron agregadas a este estudio y permitieron mejorar enormemente el análisis semántico de las palabras y los textos: como por ejemplo Word Mover's Distance (WMD), Word2vec, Glove y BERT.

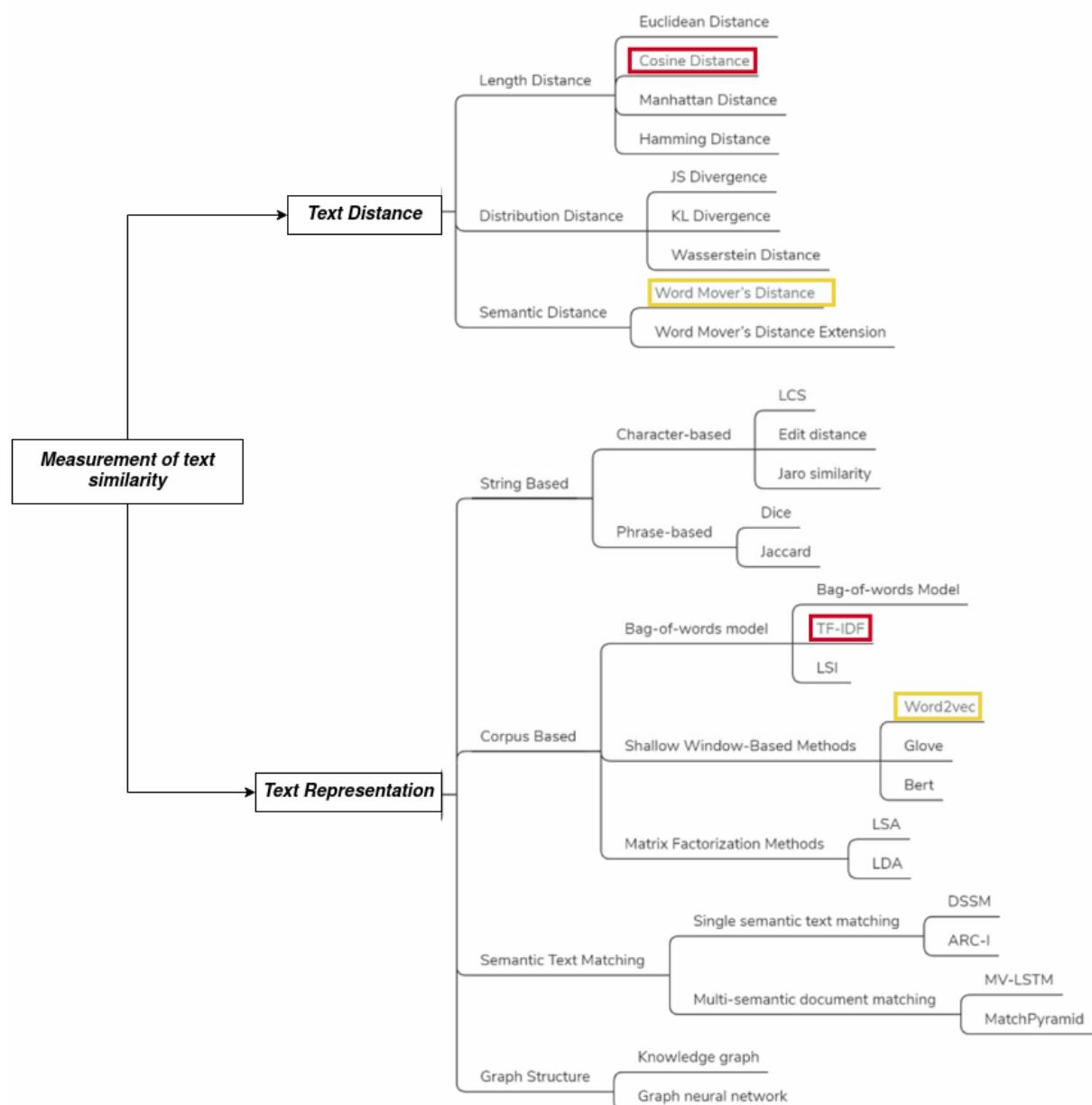


Figura 4.27: Medición de similitudes entre textos (remarcadas las técnicas utilizadas en la implementación)[4].

Como podemos observar en la Figura 4.27 ambos aspectos tienen subdivisiones: la *distancia entre los textos* (*text distance*) puede ser dividida en distancia de longitud (length distance), distancia de distribución (distribution distance), y distancia semántica (semantic distance); mientras que las *representaciones de los textos* (*text representation*) pueden ser divididas en representaciones basadas en cadenas de texto (string based), basadas en corpus de texto (corpus based), basadas en macheos/coincidencias semánticas de texto (semantic text matching), y basadas en estructuras de grafos<sup>44</sup> (graph structure based). La idea es la misma que expresamos a lo largo de este informe, luego de haber representado los textos vectorialmente se aplican las mediciones de distancias para obtener finalmente las medidas de similitud entre nuestros textos.

A continuación explicaremos más detalladamente en qué consisten estas subdivisiones de *distancia entre los textos* (*text distance*) y *representaciones de los textos* (*text representation*). No entraremos en detalle de cada una de las técnicas involucradas en cada categoría -a excepción de Word Mover's Distance y Cosine Distance- ya que no entra en los objetivos de este proyecto, pero si se explicará brevemente en qué consiste cada subdivisión.

Las tres maneras de medir las *distancias entre los textos* (*text distance*) son [4]:

1. *De acuerdo a la longitud* (*Length Distance*): Tradicionalmente, la similitud entre textos se evaluó midiendo la distancia de acuerdo a la longitud, utilizando de esta forma las características numéricas del texto para calcular la longitud de distancia del texto vectorial. Las 3 distancias más populares de esta categoría son la distancia euclídea (Euclidean Distance), la distancia del coseno (Cosine Distance) y la distancia de Manhattan (Manhattan Distance). Debido a que utilizaremos la distancia del coseno como métrica de similitud entre nuestros textos, la misma está descrita en la sección *Cosine Similarity*. Adicionalmente en la sección *Métrica de distancia a emplear* se detallan los cálculos para la distancia euclídea y distancia de Manhattan.
2. *De acuerdo a la distribución* (*Distribution Distance*): Existen dos problemas al usar las distancias basadas en longitud para calcular la similitud. El primero, es que solo es adecuado para problemas simétricos, como  $\text{Sim}(A, B) = \text{Sim}(B, A)$ , pero para que la pregunta Q recupere la respuesta A, la similitud correspondiente no es simétrica. En segundo lugar, existe el riesgo de utilizar la longitud y la distancia para juzgar la similitud sin conocer las características estadísticas de los datos[75]. La distancia de distribución se utiliza para comparar si los documentos provienen de la misma distribución, juzgando de esta manera la similitud entre los documentos según dicha distribución. La divergencia JS (JS Divergence) y la divergencia KL (KL Divergence) son actualmente los métodos más populares para investigar la distancia de distribución.
3. *De acuerdo a la semántica de los textos* (*Semantic distance*): Cuando no hay una palabra común en el texto, las similitudes obtenidas basándose en las medidas de distancias basadas en longitud o distribución pueden ser relativamente pequeñas, por lo que podemos considerar calcular la distancia a nivel semántico. La distancia

---

<sup>44</sup>Un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar y estudiar relaciones binarias entre elementos de un conjunto.

conocida como Word Mover's Distance (WMD) -también llamada distancia del transportador de palabras- es el principal método utilizado para determinar la distancia semántica[3], el cual será explicado en detalle en la sección *Word Mover's Distance (WMD)*.

Continuando con las distintas *maneras de representar nuestros textos* (*Text Representation*), algo a tener en cuenta es que mientras las representaciones basadas en cadenas de texto (String-Based) y los modelos basados en BoW -pertenecientes a las representaciones basadas en corpus de texto (Corpus-Based)- se basan principalmente en el análisis de *similitud léxica* entre palabras y textos; la *similitud semántica* en cambio, la cual es la que nos interesa abordar, abarca los métodos basados en ventanas poco profundas (Shallow Window-Based Methods) y basados en factorización de matrices (Matrix Factorization Methods) -pertenecientes a las representaciones basadas en corpus de texto (Corpus-Based)-, las representaciones basadas en macheo de macheos/coincidencias semánticas de texto (Semantic Text Matching) y las representaciones basadas en estructuras de grafos (Based on Graph Structure).

A continuación, explicaremos brevemente las divisiones involucradas en las *maneras de representar nuestros textos* (*Text Representation*)[4]:

1. *Representaciones basadas en cadenas de texto (String-Based)*: La ventaja de los métodos basados en cadenas es que son muy fáciles de implementar y los cálculos de similitud que se realizan sobre estos son muy simples de calcular. Las medidas de similitud de cadenas operan en secuencias de cadenas y composición de caracteres que miden la similitud o disimilitud (distancia) entre dos cadenas de texto para una coincidencia o comparación aproximada de cadenas. De acuerdo a la composición de su unidad básica (“basic unit”) se propuso sub-dividir en dos estas representaciones, las cuales explicaremos a continuación.
  - 1.1. *Representaciones basadas en caracteres (Character-Based)*: Un cálculo de similitud basado en caracteres se basa en la similitud entre los caracteres del texto para expresar la similitud entre los textos. Dentro de esta categoría se encuentran tres algoritmos: LCS (Longest Common Substring o subcadena común más larga), distancia de edición (Editing Distance) y similitud de Jaro (Jaro Similarity).
  - 1.2. *Representaciones basadas en frases (Phrase-Based)*: La diferencia entre este método y el método basado en caracteres, es que la unidad básica del método basado en frases es una palabra de frase (phrase word). Los métodos principales son los coeficientes DICE y Jaccard.
2. *Representaciones basadas en corpus de texto (Corpus-Based)*: Existe una diferencia importante entre los métodos basados en corpus y basados en cadenas: el método basado en corpus utiliza la información obtenida de grandes corpus de texto para calcular la similitud del texto; esta información puede ser una característica textual o una probabilidad de co-ocurrencia. Por su parte el enfoque basado en cadenas es una comparación de texto a nivel literal. En los estudios más recientes, el método basado en corpus se mide de tres maneras diferentes, las cuales detallaremos a continuación.

- 2.1. *Modelos basados en Bag-of-Words (Bag-of-Words Model)*: La idea básica del modelo BoW es representar el documento como una combinación de series de palabras sin considerar el orden en que aparecen las palabras en el documento. Los métodos basados en el modelo BoW incluyen principalmente BoW (explicado en la sección *Bag of Words (BoW)*), TF-IDF (explicado en la sección *TF-IDF*) y LSI (latent semantic indexing o indexación semántica latente).
  - 2.2. *Métodos basados en ventanas poco profundas (Shallow Window-Based Methods)*: Mejor conocidos como representaciones distribuidas (Distributed Representation), estos métodos difieren de los modelos basados en BoW en un aspecto importante: BoW y TF-IDF no permiten capturar la distancia semántica entre palabras. Utilizando métodos basados en Shallow Windows nos permiten entrenar y obtener vectores de palabras de baja dimensionalidad donde las palabras similares estén más cerca en distancias, resolviendo de esta manera los problemas de la alta dimensionalidad y falta de semántica presente en los modelos basados en BoW. Estas ventajas fueron explicadas en mayor detalle en la sección *Embeddings*. Un gran número de técnicas fueron desarrolladas para obtener estos vectores de palabras, por nombrar los tres principales métodos presentes en el estudio tenemos a los modelos Word2vec, glove y BERT. Word2vec es el modelo que utilizamos en este proyecto y fue detallado en la sección *Word2vec*.
  - 2.3-*Métodos basados en factorización de matrices (Matrix Factorization Methods)*: Los métodos de factorización matricial para generar representaciones de palabras de baja dimensionalidad tienen raíces que se remontan hasta LSA (Latent Semantic Analysis o análisis semántico latente). Estos métodos utilizan aproximaciones de bajo rango para descomponer matrices grandes que capturan información estadística sobre un corpus. El tipo particular de información capturada por dichas matrices varía según la aplicación. Los avances recientes en los métodos LSA facilitaron la investigación de los métodos LDA (Latent Dirichlet Allocation o asignación de dirichlet latente).
3. *Representaciones basadas en macheos/coincidencias semánticas de texto (Semantic Text Matching)*: Como mencionamos previamente, la similitud semántica determina la similitud entre el texto y el documento sobre la base de su significado en lugar de la coincidencia de carácter por carácter. Sobre la base de LSA, estas representaciones son extraídas mediante deep learning y permiten obtener una estructura semántica jerárquica, la cual está embebida en la consulta y el documento. Aquí el texto se codifica para extraer características, por lo que se obtiene una nueva expresión. Estas representaciones pueden ser divididas en Single Semantic Text Matching y Multi-Semantic Document Matching, cuyos métodos principales son visualizados en la Figura 4.27.
4. *Representaciones basadas en estructuras de grafos (Graph Structure-Based)*: Recientemente los grafos como una forma de datos de texto estructurado llamaron la atención de la investigación tanto académica como de la industria. Existen algoritmos basados en grafos para aprender y aplicar distribuciones semánticas en aplicaciones de NLP. La ventaja de la representación basada en grafos y el cálculo de la similitud de textos sobre estas representaciones, radica en que los vínculos

(o links) entre los nodos se establecen a través de los bordes (o edges) de las estructuras de grafos para obtener un mejor grado de similitud entre los nodos. Según los diferentes tipos de grafos, principalmente se basan en la representación de grafos de conocimiento (Knowledge Graph) y en la representación de redes neuronales basadas en grafos (Graph Neural Network).

#### 4.6.2 ¿Por qué decidimos utilizar Cosine similarity y WMD?

Como observamos en la sección *Maneras de medir la similitud entre textos*, existen numerosas técnicas para obtener las similitudes de nuestros textos, tanto de medición de distancias como de representación de los textos. En base al análisis anterior, para el sistema desarrollado en este proyecto se decidió utilizar una combinación entre Cosine similarity y WMD para realizar la medición de la distancia entre textos. A continuación explicaremos las razones por las cuales optamos por estos métodos.

- En el caso de *Cosine similarity*, se utilizó dicha técnica debido a que es una de las más tradicionales y ampliamente utilizadas para medir similitudes entre textos en varias aplicaciones[4, 5, 6, 7].
- En cambio, *Word Mover's Distance (WMD)* es una técnica más actual (2015)[3] que al momento del estudio publicado el 31 de Agosto del 2020[4] es la principal técnica utilizada para la medición semántica de la distancia entre textos.

A su vez, se decidieron utilizar las siguientes técnicas para representar vectorialmente los textos a comparar:

- Para *Cosine Distance* previamente utilizamos *TF-IDF* para representar vectorialmente nuestros textos. De igual forma esta elección es debido a TF-IDF que es una de las técnicas más tradicionales y ampliamente utilizadas para representar textos y usar en conjunto con cosine distance para medir similitudes entre textos[4].
- Para *Word Mover's Distance (WMD)* previamente utilizamos *Word embeddings* obtenidos de *Word2vec* para representar vectorialmente nuestros textos. Utilizamos el modelo Word2vec para obtener los embeddings y no otro (como por ejemplo Glove, Doc2Vec o BERT) debido a que en el paper original de WMD[3] se utilizan embeddings obtenidos mediante Word2vec, decidiendo de esta manera utilizar la misma base.

### 4.6.3 Cosine Similarity.

Al medir la *similitud del coseno*, en lugar de medir la distancia entre dos puntos -como ocurría en la distancia Euclídea o distancia Manhattan- esto se transforma a un problema donde se debe medir el coseno del ángulo correspondiente a estos dos puntos -representados como vectores- proyectados en un espacio multidimensional. En la Figura 4.28 podemos observar que un menor ángulo indica una mayor similitud. Mientras más pequeño sea el ángulo, más parecidos van a ser los vectores. Si el coseno nos da 1 es que son el mismo vector. Si los vectores son ortogonales, el coseno es 0, por lo que estos dos vectores no tendrán relación entre sí. Y cuando el valor nos da muy grande y negativo esto quiere decir que los vectores son opuestos. En nuestra implementación consideraremos a estos vectores como los textos que compararemos.

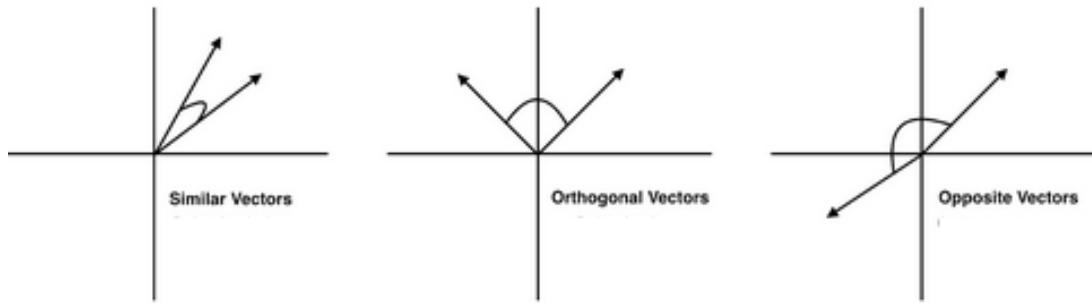


Figura 4.28: Obteniendo similitud del coseno.

De esta manera, considerando  $x$  e  $y$  como vectores a los cuales queremos calcular su similitud del coseno, su cálculo se obtiene mediante la Fórmula 29.

$$\text{CosSim}(x, y) = \cos(\theta) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \quad (\text{Ver [40]}) \quad (29)$$

*Similitud del coseno.*

Donde  $x = (x_1, x_2, \dots, x_n)$ ,  $y = (y_1, y_2, \dots, y_n)$  y  $n$  = dimensiones de los puntos.

Algo a tener en cuenta es que de la *similitud del coseno* deriva la *distancia del coseno*, o también llamada distancia angular, la cual se obtiene sustrayendo la similitud del coseno de 1. Considerando  $x$  e  $y$  como vectores a los cuales queremos calcular su distancia angular, su cálculo se obtiene mediante la Fórmula 30.

$$\text{CosDist}(x, y) = 1 - \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \quad (\text{Ver [40]}) \quad (30)$$

*Distancia del coseno.*

Donde  $x = (x_1, x_2, \dots, x_n)$ ,  $y = (y_1, y_2, \dots, y_n)$  y  $n$  = dimensiones de los puntos.

La similitud del coseno es uno de los principales métodos utilizados para medir la similitud entre dos textos  $x$  e  $y$ . Generalmente calcular la similitud del coseno para medir la similitud entre documentos de texto resulta mucho más eficiente que utilizar la distancia euclídea para el mismo fin[4].

Como mencionamos previamente, estos textos  $x$  e  $y$  están representados en forma de vectores. Si utilizamos la representación vectorial mediante el uso de TF-IDF, cada palabra en el texto definirá una dimensión en el espacio vectorial (siendo  $x_1, x_2, \dots, x_n$  las palabras del documento 1; e  $y_1, y_2, \dots, y_n$  las palabras del documento 2) y el valor TF-IDF de cada palabra corresponderá al valor en dicha dimensión. Algo que vale la pena recordar, es que utilizando TF-IDF la limitación que tenemos es que esta representación no cuenta con la habilidad de reconocer si las palabras que se comparan son semánticamente similares. Además, debido a que los valores de TF-IDF son únicamente valores positivos, entonces la similitud del coseno estará en el intervalo  $[0,1]$  y el ángulo entre los vectores no podrá ser mayor a  $90^\circ$ , por lo que podremos usar la distancia del coseno (Fórmula 30) para obtener nuestra similitud[68].

Por ejemplo, teniendo en cuenta la Figura 4.29 y considerando que  $C$ ,  $Q$  y  $D$  son nuestros textos a comparar -los cuales están representados en vectores de 2 dimensiones  $X1$  y  $X2$ - , podemos visualizar que calculando la similitud del coseno entre estos documentos, el menor ángulo que obtengamos será entre  $Q$  y  $D$ , por lo que estos serán los documentos más símiles considerando la la similitud del coseno como nuestra métrica de similitud.

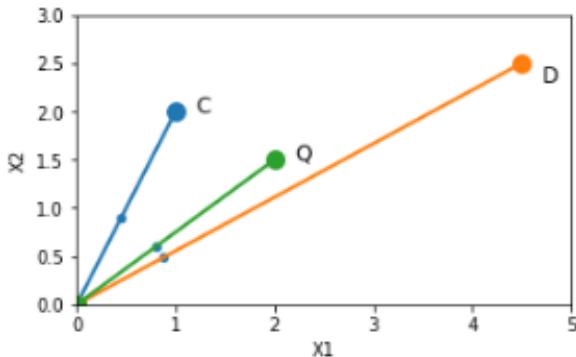


Figura 4.29: Similitud del coseno entre documentos[68].

#### 4.6.4 Word Mover's Distance (WMD).

La distancia conocida como *Word Mover's Distance (WMD)* -también llamada *distancia del transportador de palabras*- surgió en el año 2015[3] y es el principal método utilizado para determinar la similitud semántica entre textos[4]. En el cálculo de WMD una menor distancia WMD indica una mayor similitud. Anteriormente mencionamos que una de las limitaciones de Cosine Similarity era no tener la habilidad de reconocer si las palabras que compara son semánticamente similares. En cambio, WMD al tratarse de un algoritmo más complejo basado en mediciones de distancias entre *word embeddings*, si permite mantener las relaciones semánticas entre las oraciones para obtener las similitudes entre los documentos de texto. De esta forma WMD permite encontrar el costo mínimo necesario para que las palabras (representadas como word embeddings) de un documento “viajen” a la posición de todas las palabras del otro documento intentando encontrar el “vecino más cercano” para cada una de estas palabras. Por lo tanto, las colecciones de texto que comparten muchas palabras semánticamente similares deberían tener distancias más pequeñas que las colecciones de texto con palabras muy diferentes[70].

Sobre la base de representar el texto como un espacio vectorial, WMD utiliza el método de EMD (Earth Mover's Distance, originario en 1998)[76, 77]. para medir la distancia mínima requerida para que una palabra en un texto se mueva hacia una palabra en otro texto en el espacio semántico, y de esta manera minimizar el costo de transportar un texto A a un texto B[78]. Para calcular esta distancia mínima se utiliza la *distancia euclídea*. De esta manera, WMD es un método que utiliza la transportación sobre la base de los vectores de palabra (word embeddings), y su núcleo es la programación lineal[3].

Veamos un ejemplo[69]. Si bien word2vec es un enfoque sofisticado cuando se trata de generar word embeddings de calidad, estos vectores de palabras por sí solos no son suficientes para resolver a la tarea de comparar textos (los cuales están formados por diferentes word embeddings) y obtener la similitud entre ambos. Por ejemplo, consideremos estos dos textos / oraciones: “*My smart home should turn on my favorite music when I come to my home.*” y “*My smart home shall play my most favored songs when I arrive at my place*”. Vemos que las oraciones transmiten básicamente la misma información. Obteniendo los word embeddings de ambos textos y colocándolos en un espacio de embedding, observaremos que algunos de sus vectores estarán cerca, especialmente si las palabras son símiles (por ejemplo, los pares *< music, songs >* y *< come, arrive >* estarán cerca). La cercanía de los textos / oraciones completas, por otro lado, no se puede representar solo en el modelo word2vec. Para superar esta escasez, surgió Word Mover's Distance (WMD) como una medida de distancia basada en palabras (word-based) para textos / oraciones completas. Basado en la creación previa de word embeddings (como, por ejemplo, utilizando el modelo word2vec), la distancia entre dos documentos de texto *A* y *B* se describe como la distancia acumulada mínima que las palabras del documento *A* deben recorrer para coincidir exactamente con la nube de puntos del documento *B*. Con este método, WMD alcanza un alto grado de precisión estando completamente libre de hiper-parámetros y, por lo tanto, siendo fácil de usar.

Tomaremos otro ejemplo[70] para explicarlo gráfica y matemáticamente. Este es un ejemplo adaptado del paper original de WMD[3]. En laFigura 4.30 vemos un ejemplo de ilustración de las distancias euclidianas entre word embeddings. Algo a tener en cuenta es que en este ejemplo es un espacio vectorial de 2 dimensiones únicamente por motivos ilustrativos, usualmente los vectores de palabras tienen un tamaño de entre 50 a 500

dimensiones.

Cada cantidad (una por flecha) indica la contribución de ese par de palabras a la estadística general de WMD para esos dos documentos. Estas cantidades son el producto de dos números:  $c(i, j)$ , que es la distancia euclidiana entre las dos palabras en el espacio de embeddings n-dimensional -también llamado 'word travel cost'-, y un término de ponderación/peso  $T_{ij}$  que indica cuánto de  $i$  en un documento  $A$  debe viajar a la palabra  $j$  en el otro documento  $B$ . La contribución de la distancia entre "Chicago" e "Illinois" a WMD entre los documentos 1 y 2 es de 0,18, mientras que la contribución de la distancia entre "speaks" y "greets" es ligeramente mayor, 0,24: esto lo podemos visualizar en la Figura 4.31, en el cálculo de  $\text{WMD}(\text{doc1}, \text{doc2})$ . Vemos que al realizar este cálculo se toma el costo acumulado de mover todas las palabras en el *Documento1* a las ubicaciones de las palabras en el *Documento2*. Esta distancia WMD dió un resultado de 1.07, siendo menor que el resultado obtenido de  $\text{WMD}(\text{doc3}, \text{doc2})$ : 1,63. Esto nos indica que existe una mayor similitud entre los documentos 1 y 2 que entre los documentos 3 y 2.

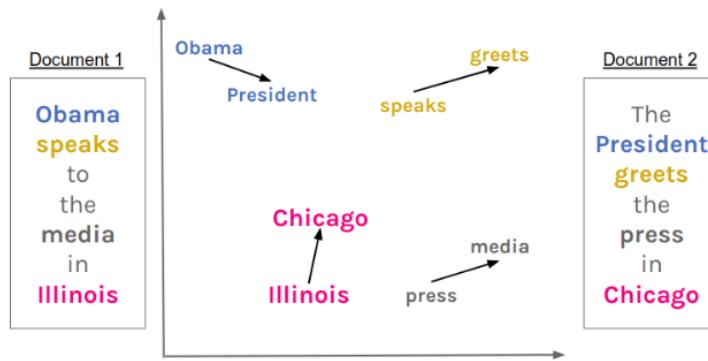


Figura 4.30: Ejemplo de ilustración de las distancias euclidianas entre word embeddings (2 dimensiones)[70].

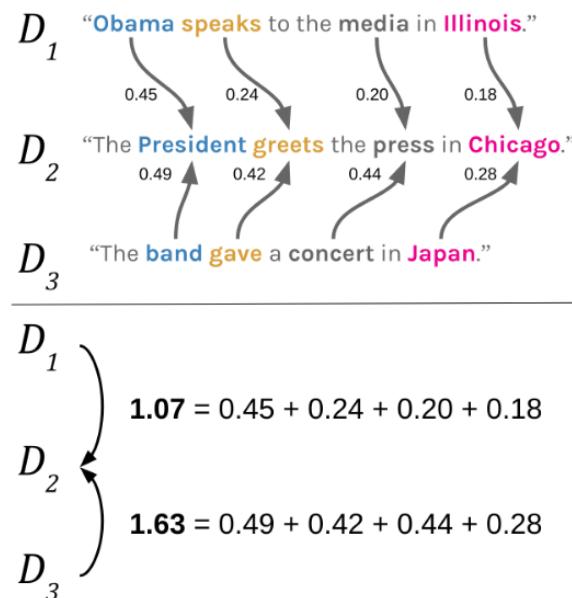


Figura 4.31: Ejemplo del cálculo de  $\text{WMD}(d1, d2)$  y  $\text{WMD}(d3, d2)$ [70].

Matemáticamente hablando, el algoritmo WMD puede ser descrito como un problema de optimización (Fórmula 31), donde se buscan los valores de una matriz T que minimizan “mover” un documento A, a otro documento B. De esta manera el mínimo resultado es la distancia de los dos documentos.

$$WMD_{ij} = \min_{T \geq 0} \sum_{i,j=1}^n T_{ij}c(i,j) \quad (\text{Ver [70]}) \quad (31)$$

*Cálculo de WMD.*

Donde:

- $i$ : cada una de las palabras de un documento  $A$ .
- $j$ : cada una de las palabras de un documento  $B$ .
- $T_{ij}$ : término de ponderación que indica cuánto de la palabra  $i$  en el documento  $A$  debe viajar a la palabra  $j$  en el otro documento  $B$ .
- $c(i,j)$ : costo asociado en “viajar” de una palabra a otra. Es la distancia euclidiana entre la palabra  $i$  y la palabra  $j$ , por lo que  $c(i,j) = \|x_i - x_j\|_2$  [3].

## 5 Implementación.

La implementación de este Sistema se trabajó en dos grandes partes:

### 1. Obtención del modelo de clasificación.

En esta primera parte se obtuvieron y preprocesaron datasets de Curriculum Vitae de distintos candidatos y descripciones de puestos de trabajo de IT publicados por distintas empresas, para luego ser comparados y obtener similitudes entre los textos utilizando las técnicas para medir distancias y obtener dichas similitudes (WMD y Cosine Similarity) y las técnicas de vectorización (TF-IDF y Word Embeddings).

Una vez obtenidas estas mediciones de similitud entre los Curriculum Vitae de los candidatos y las descripciones de los puestos laborales de IT, estos valores se utilizaron para alimentar un algoritmo de clustering K-means que a su vez, con sus datos de salida (4 clusters), alimentan a un modelo de clasificación KNN. Finalmente, con este modelo KNN logramos, en base a los valores de similitud de nuevos candidatos, clasificar qué tan similares son dichos candidatos con respecto a la descripción de un puesto de IT: similitud escasa, similitud media, similitud alta, similitud muy alta.

Estos análisis se realizaron en documentos de Jupyter Notebook utilizando Python; y sirvieron para evaluar el comportamiento del modelo de clasificación y los distintos algoritmos de medición de similitudes para luego ser utilizados en la siguiente etapa.

### 2. Integración al Sistema Web.

Etapa posterior a la primera parte. Una vez observado que los resultados fueron los esperables, lo que se hizo fue reutilizar las funciones que contenían la lógica de los distintos algoritmos utilizados junto con el modelo de clasificación KNN obtenidos previamente en la parte 1, para integrar todo esto en el sistema Web. Este sistema web está realizado en Django<sup>45</sup>, y cuenta con una base de datos relacional que contiene la información de los candidatos y reclutadores junto con los Curriculum Vitae y puestos que hayan cargado.

De esta manera, nuestro sistema cuenta con una interfaz gráfica permitiendo interactuar entre candidatos y reclutadores y, principalmente, permitiendo que el reclutador sea capaz de obtener un listado con los N candidatos más similes a un puesto determinado, y ordenados de mayor a menor de acuerdo a esta *similitud*. Dicha *similitud* representa el resultado obtenido de la clasificación por nuestro modelo KNN.

---

<sup>45</sup>Framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como modelo–vista–controlador.

## 5.1 Obtención del modelo de clasificación.

### 5.1.1 Introducción.

Como inicio definamos qué es un modelo. En nuestro caso, un modelo representa .... Este modelo se construyó en base a ... La implementación de este Sistema se realizó en Python...

### 5.1.2 Esquema.

Como podemos ver la figura 5.1 representa el procedimiento utilizado para la obtención de nuestro modelo de clasificación.

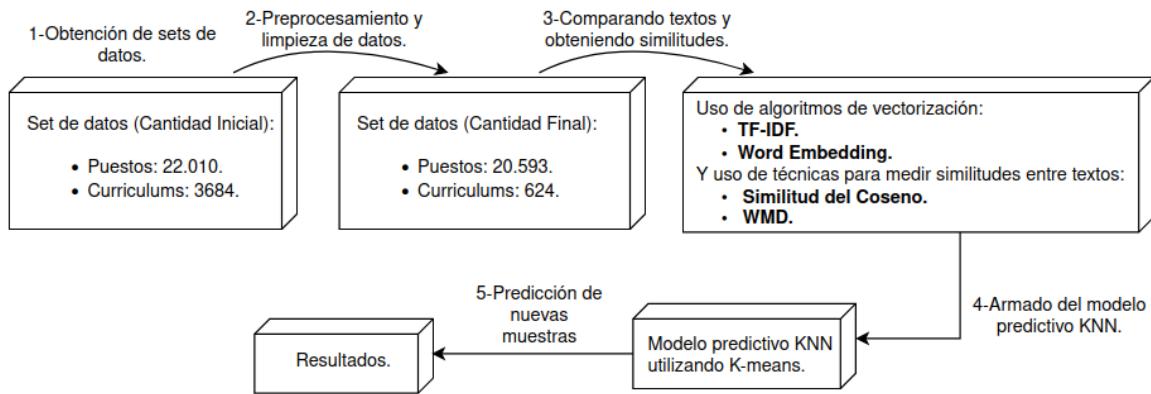


Figura 5.1: Pipeline Flow para la obtención del modelo de clasificación KNN

### **5.1.3 Obtención de sets de datos.**

En primer lugar debemos definir qué es un set o conjunto de datos. Un set o conjunto de datos es una tabla de una base de datos o, matemáticamente, una matriz estadística de datos. Cada columna de la tabla representa una variable del set de datos; y cada fila representa a un miembro determinado del mismo.

Para este Proyecto utilizamos dos grandes sets de datos que se obtuvieron mediante la recolección de distintos archivos alojados en la Web, los cuales estan descriptos a continuación.

#### **5.1.3.1 Curriculum Vitae.**

Los set de datos de Curriculum Vitae de los candidatos se obtuvieron de las siguientes fuentes:

1. 228 Curriculums en formato docx y posteriormente convertidos a pdf, obtenidos del sitio Kaggle<sup>46</sup>. Estos pdfs son candidatos de la India con experiencia en el rubro de IT.
2. 2484 Curriculums en formato CSV, obtenidos del sitio Kaggle<sup>47</sup>. Este CSV cuenta con curriculums vitae obtenidos del sitio web de postulación de trabajos 'livecareer.com'.
3. 962 Curriculums en formato CSV, obtenidos del sitio Kaggle<sup>48</sup>. Este CSV cuenta con curriculums vitae repartidos en distintas categorías de IT.
4. 10 Curriculums en formato PDF, los cuales los cuales fueron obtenidos como ejemplos mediante una recolección propia de distintos sitios web.

#### **5.1.3.2 Descripciones Puestos Laborales.**

Los set de datos de descripciones de puestos laborales se obtuvieron de las siguientes fuentes:

1. 22.000 descripciones en formato CSV; obtenido del sitio Kaggle<sup>49</sup>. El CSV cuenta con descripciones de puestos obtenidos del sitio web de USA de postulación de trabajos del rubro de IT 'Dice.com'.
2. 10 descripciones en formato CSV; obtenidas como ejemplos mediante una recolección propia del sitio Indeed<sup>50</sup> para puestos de trabajo de IT.

---

<sup>46</sup><https://www.kaggle.com/palaksood97/resume-dataset>

<sup>47</sup><https://www.kaggle.com/snehaanbhawal/resume-dataset>

<sup>48</sup><https://www.kaggle.com/gauravduttakiit/resume-dataset>

<sup>49</sup><https://www.kaggle.com/PromptCloudHQ/us-technology-jobs-on-dicecom>

<sup>50</sup><https://www.indeed.com/q-USA-jobs.html>

#### **5.1.4 Preprocesamiento de textos.**

Previamente a utilizar las técnicas para medir distancias y obtener similitudes entre textos (WMD y Cosine Similarity) y los algoritmos de aprendizaje (KNN y K-Means) necesitamos que los datos que comparemos e introduzcamos en los algoritmos estén lo más limpios posible; ya que de lo contrario las mismas podrían clasificar o predecir de forma errónea. Este análisis previo sobre los datos debe ser minucioso ya que puede haber valores incoherentes o absurdos.

El procedimiento para la Limpieza de los Curriculum Vitae y las descripciones de los puestos laborales fue el siguiente:

1. Convertimos todo a minúscula.
2. Eliminamos datos no relevantes para nuestros análisis (mails y páginas web).
3. Eliminamos signos de puntuación y carácteres especiales (incluyendo números).
4. Eliminamos stop words.
5. Eliminamos common words no relevantes para nuestros análisis.
6. Aplicamos lematización y Tokenización.
7. Eliminamos repetidos.
8. Obtenemos y usamos bi-gramas.

Luego de aplicar preprocesamiento y limpieza de datos nos quedarán los siguientes tamaños de nuestros datasets:

- 624 curriculums vitae de candidatos (en formato pdf y csv).
- 20593 descripciones de puestos de IT (en formato csv).

### 5.1.5 Cantidad final del set de datos y su uso en las distintas etapas.

El total de 624 curriculums vitae de candidatos y 20593 descripciones de puestos de IT que mencionamos previamente, serán utilizados para el entrenamiento y obtención de vectores mediante TF-IDF (para el posterior cálculo de Cosine Similarity) y para el entrenamiento de Word2Vec y obtención de los Word Embeddings (para el posterior cálculo de WMD).

Por otro lado, para calcular Cosine Similarity y WMD, para utilizarlos en K-means y para entrenar a nuestro algoritmo KNN, utilizaremos únicamente una porción de nuestros datasets:

1-Para el cálculo de Cosine Similarity y WMD:

- 301 curriculums vitae de candidatos.
- 201 descripciones de puestos de IT.

Nota: No obstante, al realizar los cálculos de distancias compararemos cada curriculum vitae con cada Job Description, obteniendo un dataframe total de 3131 filas con sus respectivos valores de WMD y Cosine Sim.

2-Para el uso de K-means y entrenamiento con KNN (eliminamos un curriculum vitae y una descripción de puesto IT que los utilizamos en '3-'):

- 300 curriculums vitae de candidatos.
- 200 descripciones de puestos de IT.

Nota: como se comentó previamente, nos quedarán 3000 filas / puntos para usar en K-means y entrenar KNN; llegando a representar estos 3000 puntos en un plano de 2 dimensiones.

3-Para la clasificación de nuevas muestras mediante KNN:

- 1 curriculum vitae de candidatos.
- 1 descripción de puesto de IT.

Nota: como se comentó previamente, nos quedarán 131 filas para clasificar.

¿Por qué utilizamos solo una porción de nuestros datasets?: Esto es debido a los drawbacks de WMD y KNN.

- WMD: posee una alta complejidad en el cálculo de la distancia, teniendo un tiempo de ejecución muy elevado. Como ejemplo, al correrlo localmente, el cálculo de WMD para 3131 filas tardó 7 horas; frente a los 3 segundos que tardó el cálculo de Cosine Similarity para la misma cantidad de filas.
- KNN: KNN es una gran opción para datasets pequeños con pocas variables de entrada; pero tiene problemas cuando la cantidad de entradas es muy grande. En

grandes dimensiones, los puntos que pueden ser similares pueden tener distancias muy grandes. Además, cada vez que se va a hacer una predicción con KNN, busca al vecino más cercano en el conjunto de entrenamiento completo. Por esto, se debe utilizar un dataset pequeño para que el clasificador KNN complete su ejecución rápidamente.

En conclusión, al utilizar solo una porción de nuestros datasets para obtener los distintos cálculos de distancias y entrenar KNN, el cálculo de WMD se podrá realizar en un tiempo finito, y nuestro clasificador KNN funcionará rápida y eficientemente al realizar predicciones.

## 5.2 Comparando textos y obteniendo similitudes.

FALTA

Previamente a utilizar Cosine Similarity y WMD para obtener las medidas de similitud entre los textos, se debe emplear alguna técnica de vectorización que permita representar las palabras de nuestros textos a un espacio vectorial. De esta forma Cosine Similarity y WMD podrán interpretarlos de la mejor manera. Como mencionamos previamente, como técnicas de vectorización se utilizarán TF-IDF y Word Embeddings.

### 5.2.1 TF-IDF & Cosine Similarity.

adgadaha

### **5.2.2 Word embeddings (Word2vec) & WMD.**

dagagad

### **5.2.2.1 Elección de hiper-parámetros Word2vec.**

adgadg

### **5.2.2.2 Word Mover's Distance (WMD).**

dagadga

### **5.3 Armado del modelo de clasificación KNN.**

**FALTA**

Una vez obtenidas estas mediciones de similitud entre los Curriculum Vitae de los candidatos y las descripciones de los puestos laborales de IT, estos valores se utilizarán para alimentar un algoritmo de clustering K-means que a su vez, con sus datos de salida (4 clusters), alimentarán a un modelo de clasificación KNN. Finalmente, con este modelo KNN lograremos, en base a los valores de similitud de nuevos candidatos, clasificar qué tan similares son dichos candidatos con respecto a la descripción de un puesto de IT: similitud escasa, similitud media, similitud alta, similitud muy alta.

## 5.4 Clasificación de nuevas muestras y resultados obtenidos.

FALTA

## 5.5 Integración al Sistema Web.

FALTA

Anteriormente lo que se hizo fue un análisis mediante documentos en Jupyter Notebooks para evaluar el comportamiento del modelo de clasificación y los distintos algoritmos de medición de similitudes.

Al observar que los resultados fueron los esperados, lo que se hizo en esta última etapa fue reutilizar las funciones que contenían la lógica de los distintos algoritmos utilizados junto con el modelo de clasificación KNN obtenidos en la fase previa, para integrar todo esto en el sistema Web.

Como mencionamos previamente, este sistema web está realizado en Django, y cuenta con una base de datos relacional que contiene la información de los candidatos y reclutadores junto con los Curriculum Vitae y puestos que hayan cargado.

De esta manera, nuestro sistema cuenta con una interfaz gráfica permitiendo interactuar entre candidatos y reclutadores y, principalmente, permitiendo que el reclutador sea capaz de obtener un listado los N candidatos más similares a un puesto determinado, y ordenados de mayor a menor de acuerdo a esta *similitud*. Dicha *similitud* representa el resultado obtenido de la clasificación por nuestro modelo KNN.

El sistema web contará con 2 tipos de usuario:

- Candidato: quienes cargarán en el sistema sus Curriculum Vitae y aplicarán a los distintos puestos disponibles.
- Reclutador: quienes cargarán en el sistema los puestos de trabajo que tengan disponibles y podrán consultar, entre otras cosas, un listado con los N candidatos más similares a un puesto determinado, y ordenados de mayor a menor de acuerdo a esta *similitud*.

### 5.5.1 Base de datos.

Nuestros datos los almacenaremos en una base de datos **FALTA definir cual**.

Para modelar y gestionar nuestros datos utilizamos el modelo relacional <sup>51</sup>.

**Sacar la mayoría del documento modelo-entidad-relacion-case-method-richard-barker.pdf**

Para comprender los datos que se almacenan en dicha base de datos, los representaremos utilizando un diagrama entidad relación <sup>52</sup>. Previamente a esto explicaremos los elementos del diagrama de entidad relación:

**FALTA PONER IMAGEN CON LOS ELEMENTOS:** Entidad rectángulo, Unión entre entidades

las líneas que puede ser obligatoria u opcional, cardinalidad son los 1:M / 1:1 / M:M

- Entidad: objeto concreto o abstracto que figura en nuestra base de datos. Por ejemplo una entidad puede ser un alumno, un cliente, una empresa, etc. Dentro de las entidades están los atributos, atributos principales o clave primaria (PK) y atributos foraneos o clave secundaria (FK). Las entidades que necesitamos para crear nuestra BD son: Candidato, Puesto, Reclutador y Candidato\_Puesto -entidad intermedia entre Candidato y Puesto-.
- Unión entre entidades: pueden ser obligatorias u opcionales. En nuestro diagrama nuestras uniones son todasopcionales, ya que el reclutador puede o no CARGAR un puesto, el candidato puede o no APLICAR a un puesto y a su vez el puesto puede o no ser aplicado por un candidato.
- Cardinalidad: Relación entre entidades o mapeo. La cardinalidad es el tipo de relación entre entidades. Observando la figura 5.2 y considerando que los rectángulos azules son una entidad y los naranjas son otra entidad observamos que pueden haber 4 tipos de cardinalidades posibles:
  1. Uno a uno: a cada entidad azul le corresponde solo una entidad naranja.
  2. Uno a muchos: a cada entidad azul le corresponde una o varias entidades naranjas.
  3. Muchos a uno: a cada entidad naranja le corresponde una o varias entidades azules.
  4. Muchos a muchos: las entidades azules pueden tener varias entidades naranjas y las entidades naranjas también pueden tener varias entidades azules.

---

<sup>51</sup>Una base de datos relacional es un conjunto de una o más tablas estructuradas en registros (líneas) y campos (columnas), que se vinculan entre sí por un campo en común.

<sup>52</sup>Un modelo entidad-relación es una herramienta para el modelo de datos, la cual facilita la representación de entidades de una base de datos.



Figura 5.2: Tipos de cardinalidad.

La cardinalidad entre nuestras entidades son:

- Entre Candidato y Puesto existe una relación muchos a muchos (M:M), ya que un candidato puede aplicar a M puestos y un puesto puede ser aplicado por M candidatos. Es por esto que se creó la tabla intermedia Candidato\_Puesto conllevando dos relaciones uno a muchos (1:M) con Puesto y Candidato.
- Entre Reclutador y Puesto existe una relación de uno a muchos (1:M), ya que un reclutador puede cargar M puestos, y un puesto pertenece a un solo reclutador.

En cuanto a las claves pueden haber dos tipos. Por un lado está la clave primaria o atributo principal (PK) es única y toda entidad debe tener la suya. Pueden haber múltiples PKs; estas se llaman PKs compuestas. Y por el otro está la clave secundaria o atributo foráneo (FK). Estas claves identifican a una entidad externa en otra, utilizándose para generar relaciones entre nuestras entidades. Si tenemos una clave FK en una entidad, significa que dicha clave FK es clave PK en otra entidad.

El diagrama de relación que utilizamos para nuestro trabajo lo observamos en la figura 5.3.

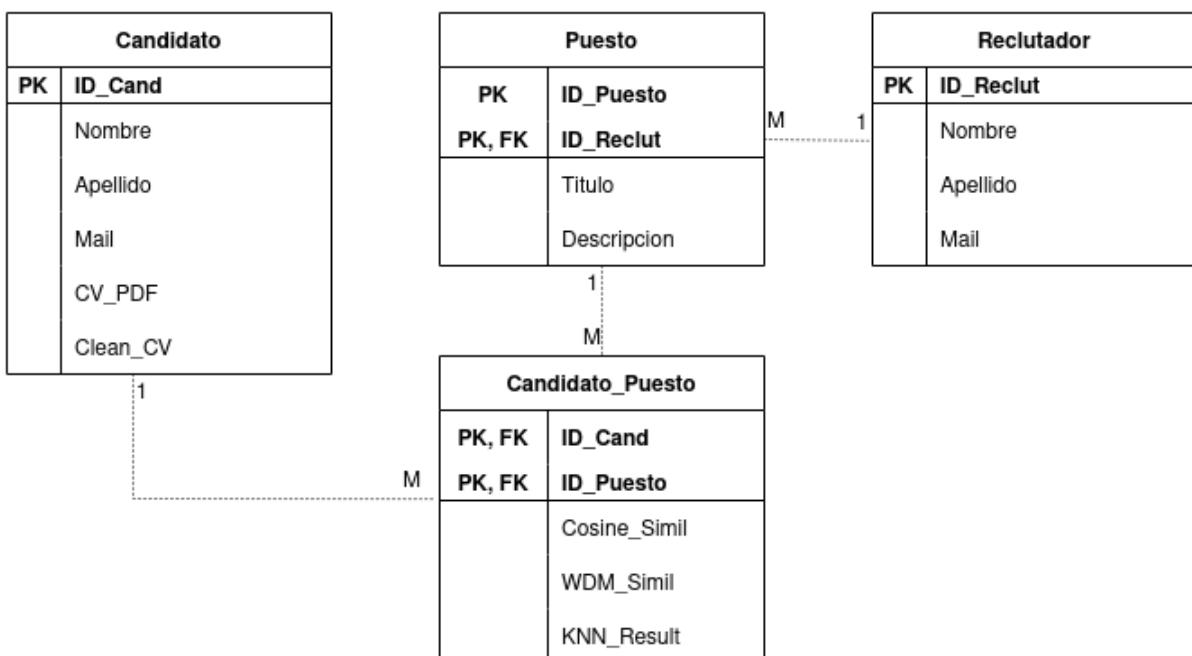


Figura 5.3: Diagrama de relación utilizado.

### 5.5.2 Secciones del sistema

Para registrarse o loguearse al sistema, se implementará la interfaz provista en la figura 5.4.

The figure displays a user interface for 'Login / Registration' with two main sections: 'Registration' and 'Login'.

**Registration Section:**

- Fields:** 'Usuario' (text input), 'Mail' (text input), and 'Contraseña' (text input).
- Checkboxes:** 'Candidato' (checked) and 'Reclutador'.
- Note:** '(en caso de ser Candidato):' followed by 'Cargar CV\*' (button) and 'Ningún CV Cargado'.
- Buttons:** 'Registrarse' (dark button).

**Login Section:**

- Fields:** 'Usuario / mail:' (text input) and 'Contraseña' (text input).
- Buttons:** 'Ingresar' (dark button).

Figura 5.4: Logueo y Registración.

ES UN BOCETO, FALTA PONER LA IMAGEN REAL

El Candidato tendrá acceso al menú indicado en la figura 5.5.

**Candidato**

---

### Mi Perfil

Nombre y apellido	<input style="width: 100%; height: 20px; border: 1px solid black;" type="text"/>
DNI	<input style="width: 100%; height: 20px; border: 1px solid black;" type="text"/>
Fecha nacimiento	<input style="width: 100%; height: 20px; border: 1px solid black;" type="text"/>
Teléfono	<input style="width: 100%; height: 20px; border: 1px solid black;" type="text"/>
Email	<input style="width: 100%; height: 20px; border: 1px solid black;" type="text"/>
Domicilio	<input style="width: 100%; height: 20px; border: 1px solid black;" type="text"/>

Actualizar datos
Cargar y analizar nuevo CV

---

### Puestos disponibles

ID Puesto	Puesto	Descripción Puesto	Ubicación
1	Programador Full Stack	Descripción larga	Buenos Aires
2	DB Engineer	Descripción larga	Buenos Aires
3	Data Scientist	Descripción larga	Buenos Aires

(se podrá filtrar/ordenar en cada columna)

Postularse

---

### Postulaciones

ID Puesto	Puesto	Descripción	Ubicación	Fecha Pos.
1	Programador Full Stack	Descripción	Buenos Aires	15/06/2021

(se podrá filtrar/ordenar en cada columna)

Figura 5.5: Vista del Candidato.

**ES UN BOCETO, FALTA PONER LA IMAGEN REAL**

Por su parte, el Reclutador tendrá acceso al menú indicado en la figura 5.6.

The diagram illustrates the Recruiter interface with several interconnected components:

- Lista puestos**: A table showing three job posts (Programador Frontend, DB Engineer, Data Scientist) with columns for ID Puesto, Puesto, Descripción Puesto, Keywords, and Ubicación. A trash can icon indicates deletion.
- Añadir Puesto**: A button to add new job posts.
- Candidatos para el Puesto X**: A table showing candidates for a selected position (e.g., Programador Frontend). Columns include Candidato, DNI, and Fecha Postulación. It includes a note: "(se podrá filtrar/ordenar en cada columna)".
- Postulaciones del Candidato**: A table showing applications made by a selected candidate (e.g., Federico Calonge). Columns include ID Puesto, Puesto, Descripción Puesto, Ubicación, and Fecha Pos. It includes a note: "(se podrá filtrar/ordenar en cada columna)".
- Top 3 mejores Candidatos para el Puesto X**: A bar chart comparing top candidates based on similarity to the selected position. The Y-axis shows Similitud (30%, 80%, 90%). The X-axis lists Candidates: Federico Calonge, Lucas Cabot, and Imanol Lew.
- Análisis candidatos por puesto**: A table showing the number of applicants per position. A note states: "Se selecciona un puesto dada una lista de puestos. Y se apreta en el botón 'Analizar'. Entonces se analizarán todos los candidatos que se postularon a dicho puesto." An "Analizar" button is present.
- Lista candidatos**: A table showing candidate details (Candidato, DNI, Fecha Nac., Teléfono, Email, Domicilio, CV). A note: "(se podrá filtrar/ordenar en cada columna)". A plus sign icon indicates addition.

Red arrows indicate navigation paths between the main sections: from the job list to the candidate list, from the candidate list to the analysis section, and from the analysis section to the top candidates chart.

Figura 5.6: Vista del Reclutador.

ES UN BOCETO, FALTA PONER LA IMAGEN REAL

### 5.5.3 Manejo de los datos.

FALTA

### 5.5.3.1 Modelado.

FALTA

### 5.5.3.2 Filtrado.

FALTA

### 5.5.3.3 Visualización.

FALTA

## 5.6 Pipeline Flow final del Sistema.

FALTA

Una vez que el reclutador dentro de la sección observada en la figura 5.6 haga click en ".^analizar", el sistema reflejará el pipeline indicado en la figura 5.7 para obtener como resultado un listado con los N candidatos más similes a un puesto determinado, y ordenados de mayor a menor de acuerdo a esta *similitud*. Dicha *similitud* representa el resultado obtenido de la clasificación por nuestro modelo KNN.

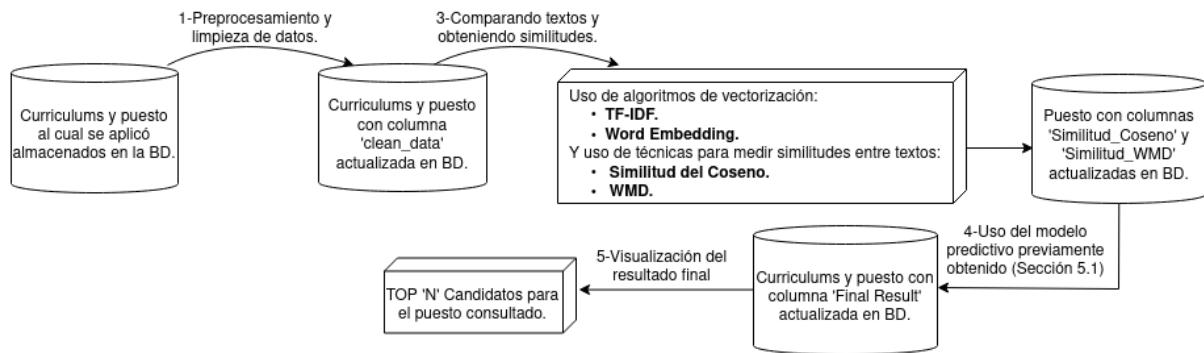


Figura 5.7: Pipeline Flow final del Sistema.

## **5.7 Caso de Uso.**

**FALTA**

## **5.8 Limitaciones del sistema.**

Las limitaciones del sistema son las siguientes:

- Solo acepta curriculums en formato PDF.
- Curriculums y Puestos de trabajo en idioma inglés.
- Curriculums y Puestos de trabajo de IT.

## 6 Conclusiones.

FALTA

Poner acá tambien los próximos pasos / mejoras.

## 7 Anexos.

### 7.1 Ejemplo de funcionamiento KNN.

A continuación detallaremos un ejemplo en el cual usaremos KNN para clasificación usando la distancia euclídea como métrica para calcular las distancias.

Como podemos ver en la Figura 7.1, el conjunto de datos está etiquetado: uno es un cuadrado azul y otro es un triángulo rojo. El círculo verde es el punto que necesitamos clasificar.

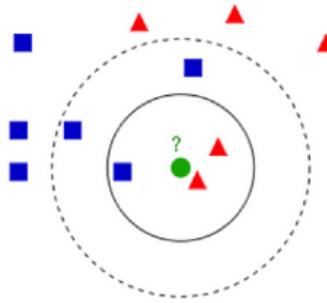


Figura 7.1: Ejemplo de Clasificación con KNN con  $K=3$  (círculo con linea sólida) y  $K=5$  (círculo con linea punteada), usando la distancia euclídea como métrica de cálculo de distancias[40].

Viendo la Tabla 8 observamos que:

- Si  $K = 1$ , tomamos únicamente el punto más cercano del círculo verde. Este punto es un triángulo rojo. Como es el único punto que vota, el punto verde a clasificar pertenece al triángulo rojo. De esta manera la probabilidad de que sea rojo es 1 y de que sea azul es 0.
- Si  $K = 3$ , entonces hay 2 triángulos rojos y 1 cuadrado azul más cercano al punto verde. Estos 3 puntos votan, por lo que el punto verde a clasificar pertenece al triángulo rojo. De esta manera la probabilidad de que sea rojo es  $2/3$  y de que sea azul es  $1/3$ .
- Si  $K = 5$ , entonces hay 2 triángulos rojos y 3 cuadrados azules más cercanos al punto verde. Estos 5 puntos votan, por lo que el punto verde a clasificar pertenece al cuadrado azul. De esta manera la probabilidad de que sea rojo es  $2/5$  y de que sea azul es  $3/5$ .

<b>K</b>	<b>P(Azul)</b>	<b>P(Rojo)</b>
1	0	1
3	$1/3$	$2/3$
5	$3/5$	$2/5$

Tabla 8: Probabilidad de pertenecer a cada clase para diferentes valores de k.

## 7.2 Ejemplo de funcionamiento K-means.

En esta sección detallaremos un ejemplo gráfico en el cual usaremos K-means para clusterizar nuestros datos hasta llegar a su convergencia.

En el ejemplo de la Figura 7.2, nuestros datos tienen 2 dimensiones. Definiendo previamente nuestro  $k=2$  (2 clusters), en el item (1) observamos dicha distribución de datos. Además, en este item se realiza la inicialización aleatoria de nuestros 2 centroides (triángulos rojo y amarillo). Todo esto involucra al **paso 1** del algoritmo k-means.

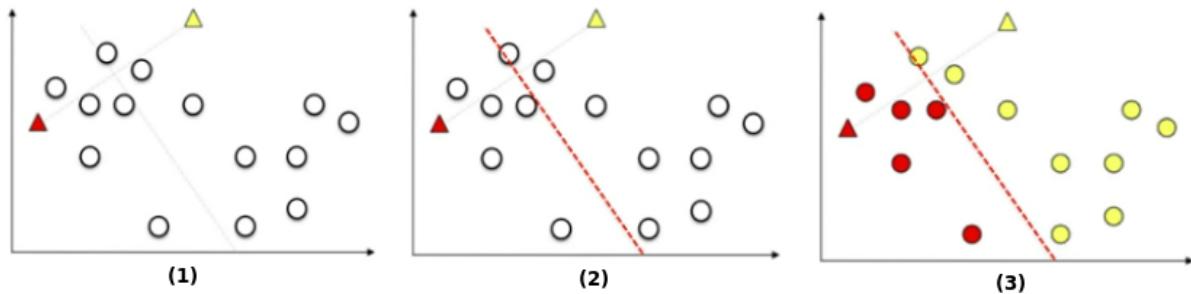


Figura 7.2: Algoritmo K-means.

En el **paso 2**, lo que se tiene que hacer es, para cada uno de nuestros puntos de datos, averiguar a cuál centroide se asignará, basándonos en la distancia más cercana al mismo. Al considerar la distancia euclíadiana para este objetivo, en (2) dibujamos una línea roja imaginaria que divide en iguales distancias nuestros 2 centroides: partiendo de esta línea la distancia a ambos centroides es la misma. De esta manera, cada punto ubicado a la derecha de la linea pertenecerá al cluster amarillo (por tener menor distancia al centroide triángulo amarillo), y cada punto ubicado a la izquierda de la linea pertenecerá al cluster rojo (por tener menor distancia al centroide triángulo rojo). Esta asignación la observamos en el item (3), que representa nuestra asignación inicial de nuestros puntos en los clusters.

La línea roja es simplemente una referencia utilizada para una mejor comprensión gráfica, pero internamente esta asignación de puntos se realiza en base al cálculo de las distancias euclidianas mediante la Fórmula 4. descrita anteriormente para el paso 2 de k-means.

Posteriormente pasamos al **paso 3** de k-means: la actualización de las posiciones de los centroides de los clusters. Lo que hacemos en (4) es mover los centroides al centro promedio de los puntos a los que se les asignaron. Para esto se utiliza la Fórmula 5 descrita anteriormente. Podemos observar las nuevas posiciones de los centroides en (5). De esta forma, finalizamos la primera iteración de nuestro algoritmo.

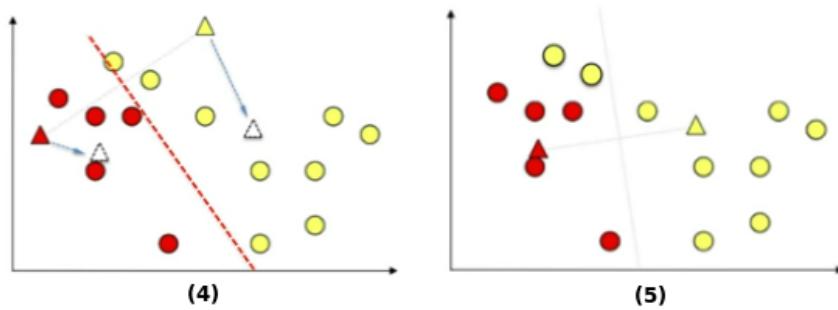


Figura 7.3: Algoritmo K-means.

En la próxima iteración, tenemos la misma disposición de los puntos, solo que ahora tenemos dos nuevas posiciones para nuestros centroides. Ignoramos las clases que se asignaron para los puntos de nuestros datos ya que esta era una asignación del paso anterior: ahora nuestros centroides cambiaron y la asignación de clases también lo hará. Nuevamente trazamos la linea imaginaria (6) y realizamos la nueva asignación: los únicos puntos que cambiaron con respecto al paso anterior son los 2 amarillos ubicados en la parte superior, los cuales pasan a ser rojos (7). Debido a que cambiaron de clase, entonces nuevamente vamos a actualizar las posiciones de nuestros centroides (8), quedándonos distribuidos finalmente como (9). De esta manera finalizamos la segunda iteración.

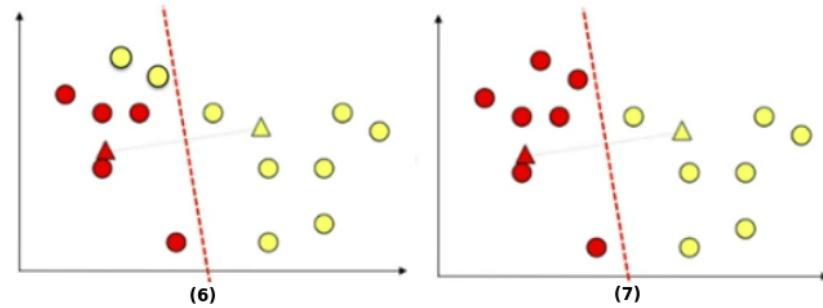


Figura 7.4: Algoritmo K-means.

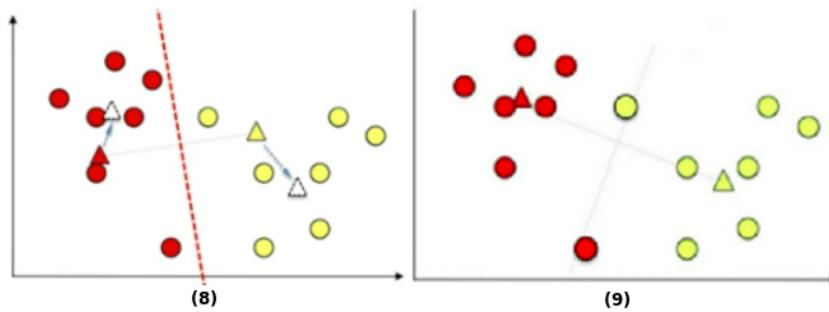


Figura 7.5: Algoritmo K-means.

Para la tercera iteración se vuelven a repetir los pasos 2 y 3: trazamos la línea de división de nuestros datos (10), realizamos la nueva asignación de puntos (11) y observamos que cambiaron 2 puntos de clases. Debido a que cambiaron de clase, entonces

re-ubicamos los centroides nuevamente (12), quedándonos finalmente como (13). De esta manera finalizamos la tercera iteración.

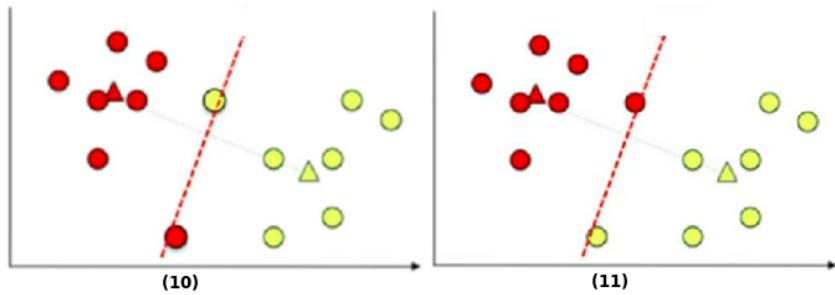


Figura 7.6: Algoritmo K-means.

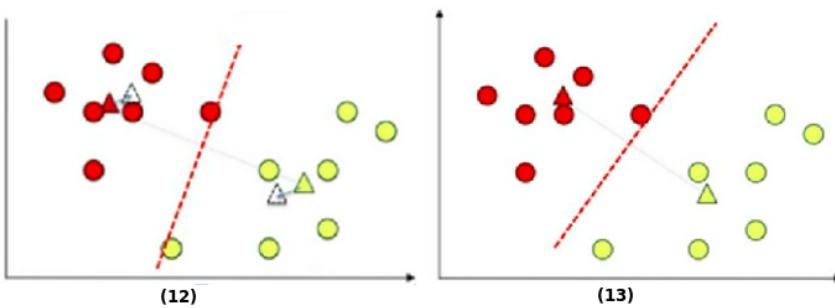


Figura 7.7: Algoritmo K-means.

Finalmente, en el ítem (13), si ahora intentamos asignar nuevamente nuestros puntos de datos, observamos que ninguno cambiaría de color/clase. Los puntos rojos ya están del lado izquierdo de la linea, y los puntos amarillos ya están del lado derecho de la linea. Esto significa que si seguimos iterando nada cambiará, por lo que llegamos a la **convergencia** de nuestro algoritmo.

### 7.3 Posición inicial de los centroides en K-means.

K-means, al ser un algoritmo que encuentra un mínimo local, es muy sensible a la posición inicial de los centroides. Una posible solución para no caer en un mínimo local malo es realizar varias ejecuciones de K-means con inicializaciones aleatorias para los centroides, idealmente ejecuciones en paralelo, y computar la función de coste  $J$  para cada resultado final quedándonos con la mínima.

La Figura 7.8 muestra un ejemplo donde hay 4 clusters y varios puntos aislados en un plano bidimensional. La Figura 7.9 muestra el resultado del mejor y peor costo. Se puede ver que el resultado del peor costo es realmente muy malo mientras que el resultado del mejor costo es lo que se esperaba obtener.

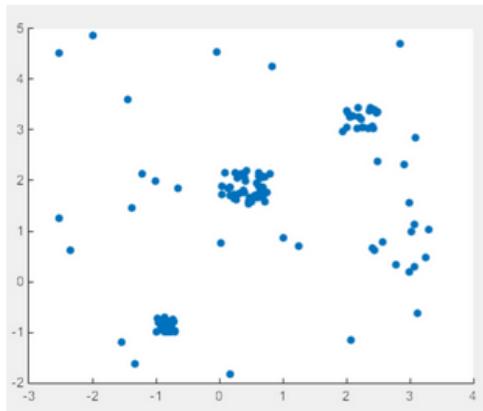


Figura 7.8: Ejemplo K-means. [10]

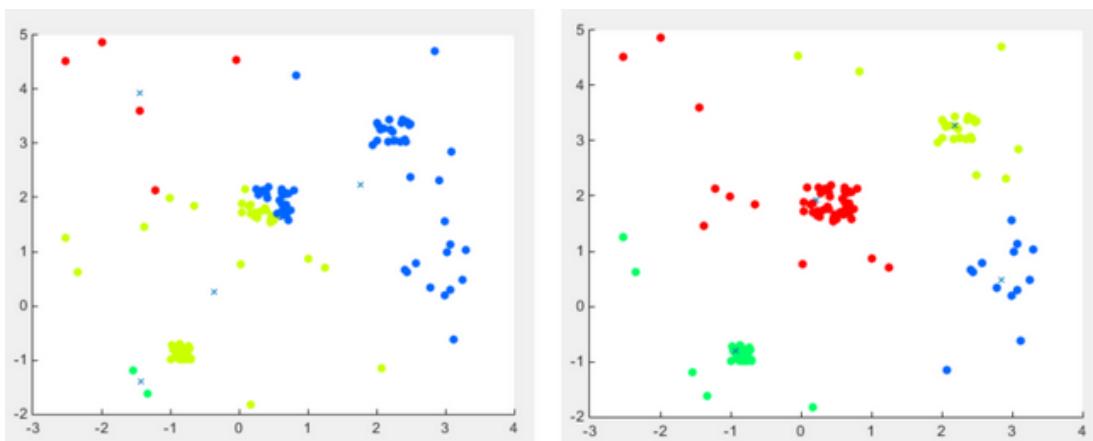


Figura 7.9: Ejemplo K-Means peor (izquierda) y mejor (derecha) costo. [10]

El gráfico de la Figura 7.10 muestra el resultado de la función costo para 100 ejecuciones de K-Means, con un  $k$  predefinido, y variando las inicializaciones de nuestros centroides. En el eje  $x$  se representan las 100 ejecuciones de k-means, y en el eje  $y$  encontramos la función de coste  $J$  obtenida para cada ejecución.

En la Figura 7.11 podemos observar un histograma representando la cantidad de veces en porcentajes (eje  $y$ ) que se observaron diferentes rangos de costo (eje  $x$ ). Como se puede ver, afortunadamente los costos bajos son amplia mayoría, es decir que únicamente con

algunas pocas inicializaciones desafortunadas K-means llega a un mínimo global muy malo. Esto quiere decir que una sola iteración de K-Means nos dará un buen resultado con una probabilidad alta y hacer un ciclo con unas pocas ejecuciones es más que suficiente para llegar a un excelente resultado.

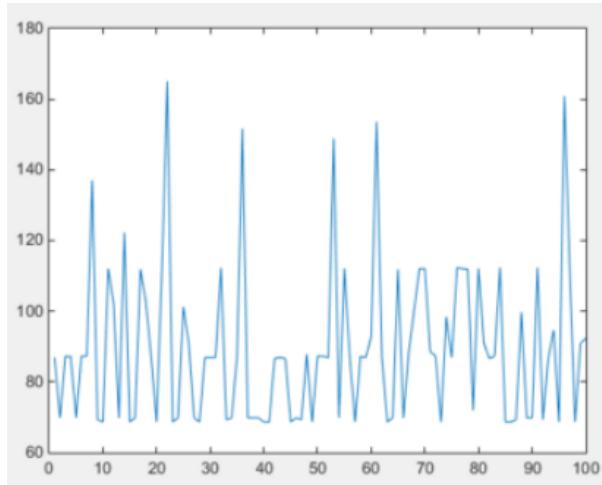


Figura 7.10: Costo en K-Means. [10]

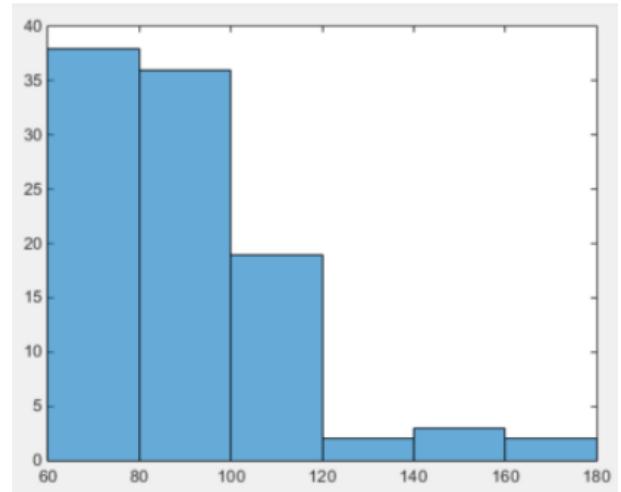


Figura 7.11: Histograma  $J$  en K-Means. [10]

## 7.4 Funciones de activación.

En la Figura 7.12 podemos observar una lista de las funciones de activación más comunes utilizadas en Redes Neuronales. Mientras que en la Figura 7.13 se detallan las expresiones matemáticas y los posibles rangos de valores de salida para dichas funciones de activación.

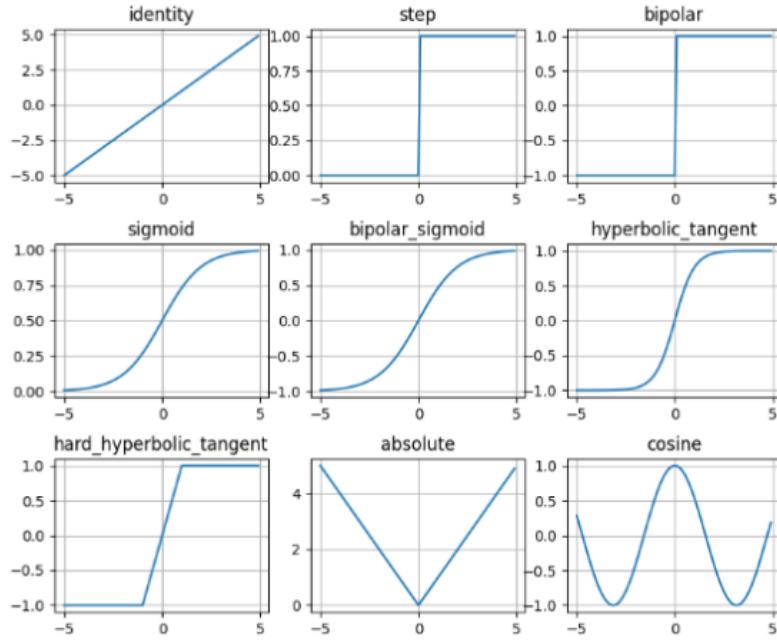


Figura 7.12: Funciones de activación más comunes[47].

Name	Expression	Range
Identity	$\text{id}(a) = a$	$(-\infty, +\infty)$
Step (Heavyside)	$T_{H \geq 0}(a) = \begin{cases} 0 & \text{if } a < 0 \\ 1 & \text{otherwise} \end{cases}$	{0, 1}
Bipolar	$B(a) = \begin{cases} -1 & \text{if } a < 0 \\ +1 & \text{otherwise} \end{cases}$	{-1, 1}
Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	(0, 1)
Bipolar sigmoid	$\sigma_B(a) = \frac{1-e^{-a}}{1+e^{-a}}$	(-1, 1)
Hyperbolic tangent	$\tanh(a)$	(-1, 1)
Hard hyperbolic tangent	$\tanh_H(a) = \max(-1, \min(1, a))$	[-1, 1]
Absolute value	$\text{abs}(a) =  a $	$[0, +\infty)$
Cosine	$\cos(a)$	[-1, 1]

Figura 7.13: Fórmulas y rangos de las funciones de activación más comunes<sup>53</sup>[47].

<sup>53</sup>a indica el resultado luego de haber aplicado la función de entrada dentro de la neurona artificial. Como mencionamos anteriormente, la función de entrada más común es la sumatoria de las entradas ponderadas.

## 7.5 Ejemplo de obtención de Word Embeddings mediante skipgram y softmax.

Tomaremos un ejemplo del libro de O’reilly Media[71] y explicaremos resumidamente los pasos que debemos realizar para obtener nuestros Word Embeddings utilizando el modelo Skip-gram con función de salida softmax y utilizando el algoritmo de gradiente descendiente básico.

El *corpus de texto* con el que nuestro algoritmo entrenará será la oración “The quick brown fox jumps over the lazy dog”. Previamente a que ingrese al modelo, este corpus debe ser tokenizado, obteniendo el siguiente *corpus tokenizado* : [“the”, “quick”, “brown”, “fox”, “jumps”, “over”, “the”, “lazy”, “dog”]. Nuestro *vocabulario*<sup>54</sup> será el siguiente: [“the”, “quick”, “brown”, “fox”, “jumps”, “over”, “lazy”, “dog”]. Observamos que el tamaño del *corpus tokenizado* es 9, mientras que el tamaño del *vocabulario* es 8: uno menos debido a la repetición de la palabra “the”.

Cabe mencionar que en este ejemplo no se preprocesaron los textos previamente a armar nuestro corpus tokenizado. Si se preprocesaran los mismos -como se realiza en nuestra implementación-, nuestro corpus tokenizado que será utilizado como entradas del modelo quedaría como [“quick”, “brown”, “fox”, “jump”, “lazy”, “dog”], y nuestro vocabulario, como no existe repetición de palabras, estarían formados por estas mismas 6 palabras / tokens.

Para el entrenamiento de nuestro modelo consideraremos los siguientes hiper-parámetros:

- Tamaño de ventana de contexto de 2. ( $C = 2$ ).
- Tamaño de embedding de 3. ( $N = 3$ ).

Lo primero que tenemos que hacer es armar nuestro dataset, el cual será utilizado como entrenamiento. Para mayor entendimiento, en la Figura 7.14 representaremos las palabras utilizadas en cada iteración para entrenar a nuestro modelo Skipgram. En este caso, como nuestro tamaño de ventana de contexto es 2, tomaremos para cada paso 2 palabras de cada lado de la palabra central como nuestras palabras de contexto: en el primer paso nuestra palabra central es “the” y nuestras palabras de contexto son “quick” y “brown”. En este caso no podemos tomar palabras a la izquierda, por esto únicamente tenemos 2 palabras de contexto; distinto es el caso del paso 3, donde tomamos como palabra central “brown” y sí podemos tomar las 2 palabras de contexto de ambos lados: “the”, “quick”, “fox”, “jumps”.

---

<sup>53</sup>El corpus tokenizado son todas las palabras de nuestro corpus que serán utilizadas para entrenar y evaluar nuestro modelo.

<sup>54</sup>El vocabulario es el set de palabras únicas -sin repetir- obtenidas de nuestro corpus tokenizado

Palabras utilizadas para entrenar Skipgram									Nº iteraciones necesarias
the	quick	brown	fox	jumps	over	the	lazy	dog	2
the	quick	brown	fox	jumps	over	the	lazy	dog	3
the	quick	brown	fox	jumps	over	the	lazy	dog	4
the	quick	brown	fox	jumps	over	the	lazy	dog	4
the	quick	brown	fox	jumps	over	the	lazy	dog	4
the	quick	brown	fox	jumps	over	the	lazy	dog	4
the	quick	brown	fox	jumps	over	the	lazy	dog	4
the	quick	brown	fox	jumps	over	the	lazy	dog	3
the	quick	brown	fox	jumps	over	the	lazy	dog	2

Figura 7.14: Armando dataset para Skipgram

En cada iteración el modelo entrenará con un *par* de palabras (central - de contexto). En la Figura 7.15 podemos observar el dataset que obtuvimos armándolo con las palabras de entrada (centrales) y palabras de salida (de contexto; o palabras “target”) que se utilizarán para entrenar a nuestro modelo Skipgram, junto a las iteraciones necesarias. Por ejemplo, para nuestro primer paso de la Figura 7.14 (primer fila), observamos que nuestra palabra central es “the” y nuestras palabras de contexto son “quick” y “brown”. Vemos que se forman 2 pares de palabras, por esto en la primera iteración entrenaremos con el par (“the”, “quick”) y en la segunda iteración usaremos el par (“the”, “brown”). Aquí lo que estamos haciendo esencialmente es capturar los pares de palabras de todo nuestro corpus para capturar el *contexto* y usarlo como información para entrenar nuestros word embeddings.

Entrada (palabra central)	Salida (palabras de contexto)	Iteración
the	quick	1
the	brown	2
quick	the	3
quick	brown	4
quick	fox	5
brown	the	6
brown	quick	7
brown	fox	8
brown	jumps	9
fox	quick	10
fox	brown	11
fox	jumps	12
fox	over	13
(...)	(...)	(...)
dog	the	29
dog	lazy	30

Figura 7.15: Dataset para Skipgram

Como mencionamos previamente, las palabras que ingresarán como entrada en nuestro modelo, presentarán un vector en formato one-hot encoding: Ver Figura 7.16. Recordemos que mediante one hot encoding se asigna un único código para cada única palabra.

Este vector tiene las características descritas a continuación.

- Su valor es 1 para la posición que tiene la palabra en nuestro vocabulario y 0 para las demás posiciones.
- Su tamaño es igual al número de palabras de nuestro vocabulario (en nuestro caso  $V = 8$ ).

<b>Entrada (palabra central)</b>	<b>One-hot encoding</b>	<b>Índice en el vocabulario</b>
the	[1,0,0,0,0,0,0]	1
quick	[0,1,0,0,0,0,0]	2
brown	[0,0,1,0,0,0,0]	3
fox	[0,0,0,1,0,0,0]	4
jumps	[0,0,0,0,1,0,0]	5
over	[0,0,0,0,0,1,0]	6
the	[1,0,0,0,0,0,0]	1
lazy	[0,0,0,0,0,1,0]	7
dog	[0,0,0,0,0,0,1]	8

Figura 7.16: Representación en one-hot encoding para cada una de las palabras de entrada

Teniendo en cuenta lo mencionado previamente, pasaremos a explicar gráficamente las iteraciones necesarias para que nuestro modelo Skipgram entrene, el cual podemos visualizar en la Figura 7.17.

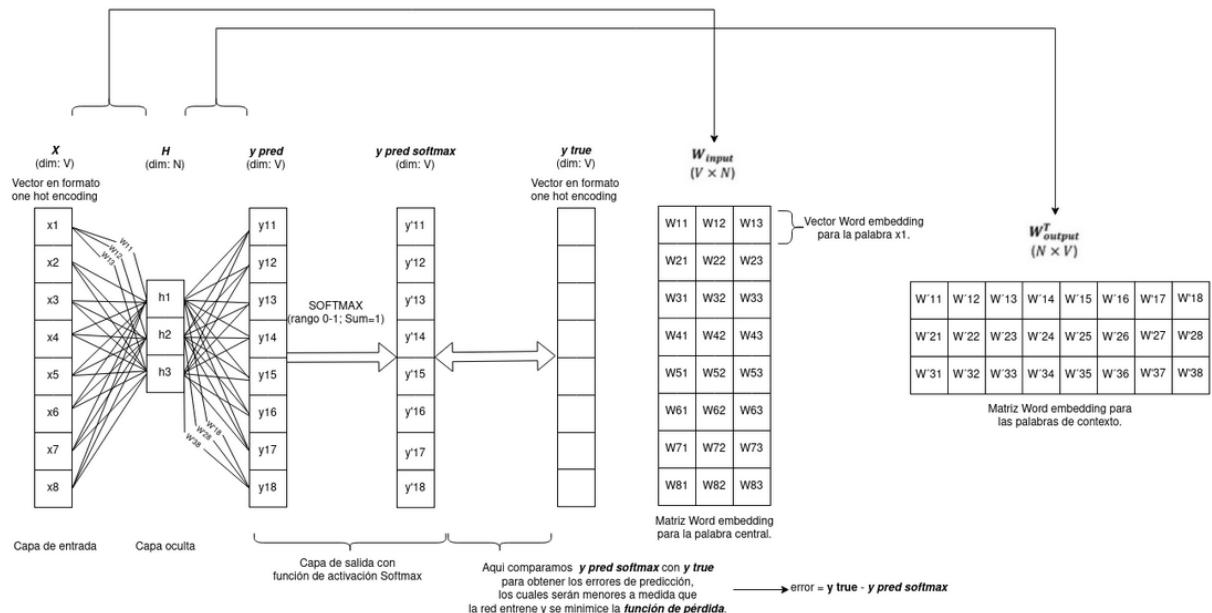


Figura 7.17: Arquitectura Skipgram con  $V=8$ ,  $N=3$ .

A continuación detallaremos las iteraciones de entrenamiento necesarias teniendo en cuenta la Figura 7.15.

1. El primer paso consiste en la inicialización aleatoria por única vez de las matrices de pesos, las cuales son los parámetros ( $\theta$ ) de nuestro modelo de red neuronal. En otras palabras, esto quiere decir que se llenan con valores numéricos las matrices  $W_{\text{input}}$  (o  $W$ ) y  $W_{\text{output}}$  (o  $W'$ ): los valores  $W_{11}$  a  $W_{83}$  y  $W'_{11}$  a  $W'_{38}$  respectivamente.
  2. Este paso consiste en la aplicación de forward propagation para obtener nuestra salida - $y_{\text{pred softmax}}$ . Este paso se divide en 2 sub-pasos: *Paso 2.1-Forward propagation (capa de entrada a capa oculta)* y *Paso 2.2-Forward propagation (capa oculta a capa de salida)*. Algo importante a tener en cuenta es que, debido a que estamos utilizando el algoritmo de gradiente descendiente básico y no SGD, en este paso N°2 tenemos que obtener los valores de  $y_{\text{pred softmax}}$  de todos nuestros pares de palabras (central, contexto) para luego, teniendo estos resultados, pasar al paso N° 3. Es por esto que, *este paso N°2 (y en consecuencia los sub-pasos 2.1 y 2.2) se realizan por cada una de las 30 iteraciones que vimos en la Figura 7.15* hasta obtener 30 vectores  $y_{\text{pred softmax}}$  (los cuales luego se compararán con los 30 vectores  $y_{\text{true}}$  correspondientes a la palabra de entrada de cada iteración).
- Iteración 1: Ingresa “the” como palabra central en la capa de entrada y se intenta predecir la palabra de contexto “quick” en la capa de salida. Ambas palabras son representadas como vectores one hot encoding (vectores  $X$  e  $y_{\text{true}}$  respectivamente). Esto lo observamos en la Figura 7.18. En la Figura 7.19 podemos observar un mayor detalle.

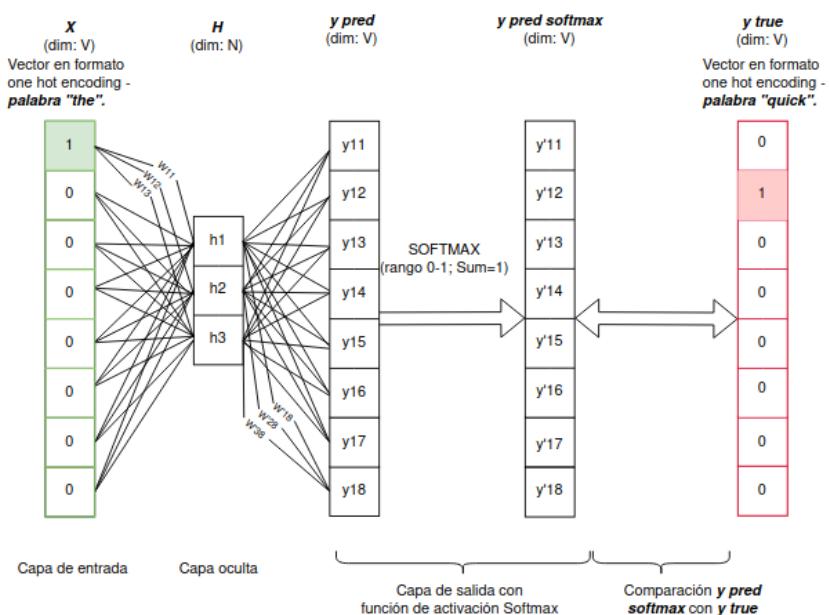


Figura 7.18: Primera iteración Skipgram del paso 2.

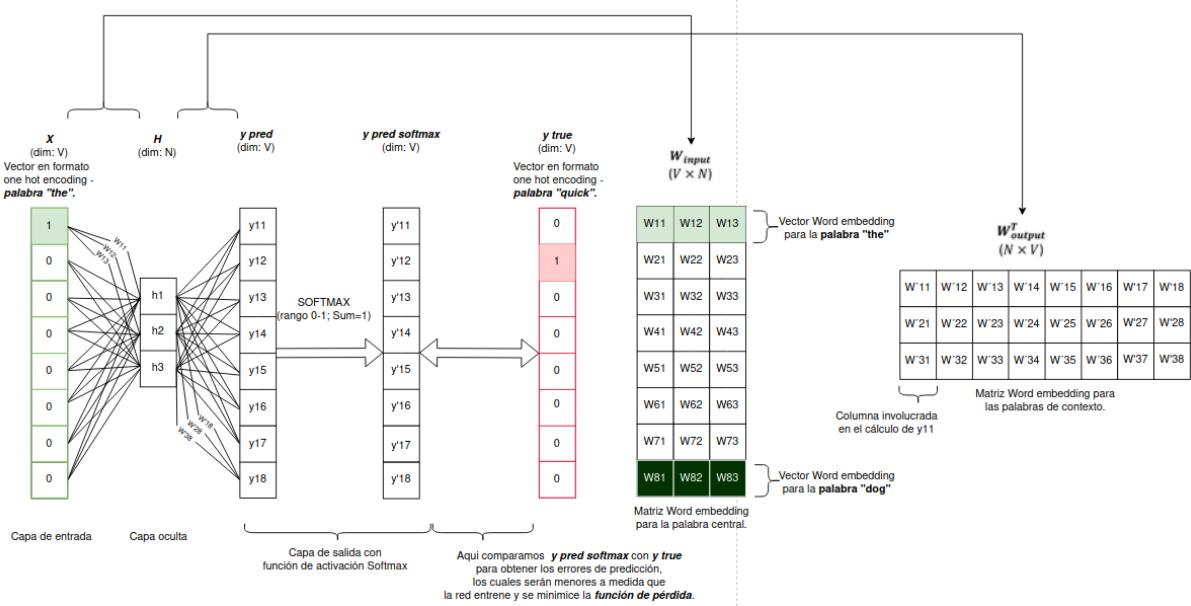


Figura 7.19: Primera iteración Skipgram del paso 2 en mayor detalle.

- *Paso 2.1 para la Iteración 1:* Forward propagation (capa de entrada a capa oculta): En este paso se calcula el valor de las neuronas  $h_1$ ,  $h_2$  y  $h_3$  de la capa oculta  $H$ . Para esto se utiliza la palabra central de entrada “the” en conjunto con su fila correspondiente de la matriz de embedding  $W$ : la cual observando la Figura 7.19 podemos ver que corresponde a los valores  $W_{11}$ ,  $W_{12}$  y  $W_{13}$ . Debido a que solo una posición del vector de entrada “the” es 1 ( $x_1$ ) y los demás son 0 ( $x_2$  a  $x_8$ ), entonces los valores  $W_{11}$ ,  $W_{12}$  y  $W_{13}$  son los valores que tomarán  $h_1$ ,  $h_2$  y  $h_3$  respectivamente). En 32 observamos el cálculo necesario para la obtención de  $h_1$ .

$$\begin{aligned}
 h_1 &= x_1.W_{11} + x_2.W_{21} + x_3.W_{31} + x_4.W_{41} + x_5.W_{51} \\
 h_1 &= x_1.W_{11} + 0 \\
 h_1 &= 1.W_{11} \\
 h_1 &= W_{11}
 \end{aligned} \tag{32}$$

Teniendo en cuenta este cálculo,  $h_2$  y  $h_3$  se calculan de la misma manera que  $h_1$ , siendo  $h_2 = W_{12}$  y  $h_3 = W_{13}$ . Entonces obtenemos  $H = (W_{11}, W_{12}, W_{13})$ .

- *Paso 2.2 para la Iteración 1:* Forward propagation (capa oculta a capa de salida): En este paso se calcula el valor de cada neurona de la capa de salida luego de pasar por la función de salida softmax (vector  $y_{pred softmax}$  de la Figura 7.19). Debido a que la capa oculta no tiene función de activación, entonces los valores calculados en el *paso 2.1* pasan directamente a la capa de salida. De esta manera, los valores de las neuronas de la capa de salida se calculan multiplicando el vector  $H$  que nos dió anteriormente con la otra matriz  $W'$  -de tamaño  $N \times V$ -. Esto nos dará un vector  $y_{pred}$  de tamaño

V. En 33 observamos el cálculo necesario para la obtención de  $y_{11}$  e  $y_{12}$ .

$$\begin{aligned} y_{11} &= h_1 \cdot W'_{11} + h_2 \cdot W'_{21} + h_3 \cdot W'_{31} \\ y_{12} &= h_1 \cdot W'_{12} + h_2 \cdot W'_{22} + h_3 \cdot W'_{32} \end{aligned} \quad (33)$$

Los cálculos de  $y_{13}$  a  $y_{18}$  siguen la misma lógica descrita en el cálculo 33. De esta manera, tendremos los valores para nuestro vector  $y \text{ pred}$ .

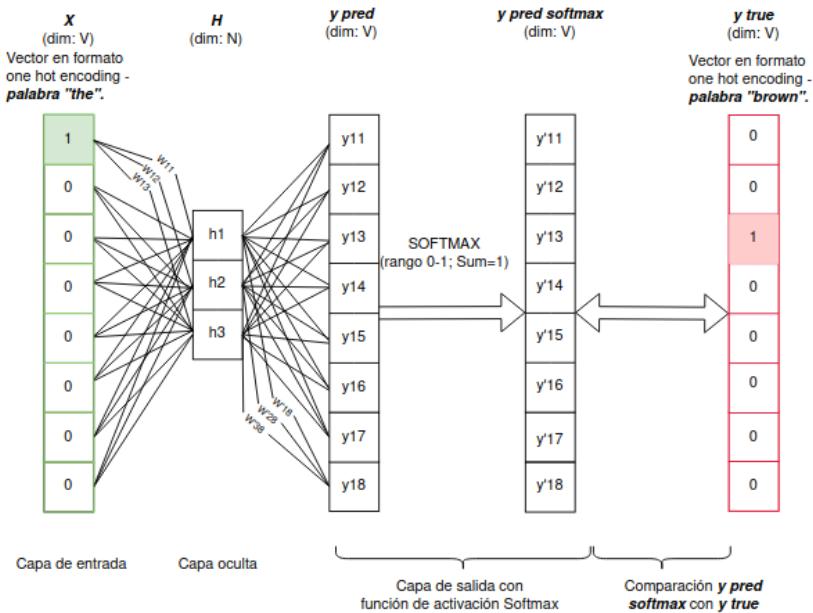
Por último, este vector alimentará a la función de salida softmax con el objetivo de obtener una distribución de probabilidades sobre el espacio de nuestro vocabulario - $y \text{ pred softmax}$ -, el cual se calculará, utilizando la Fórmula 14). Por ejemplo, en 34 observamos el cálculo necesario para la obtención de  $y'_{11}$  e  $y'_{12}$ .

$$y'_{11} = \sigma(y_{11}) = \frac{\exp(y_{11})}{1 + \exp(y_{11})} \quad (34)$$

$$y'_{12} = \sigma(y_{12}) = \frac{\exp(y_{12})}{1 + \exp(y_{12})}$$

Los cálculos de  $y'_{13}$  a  $y'_{18}$  siguen la misma lógica descrita en el cálculo 34. De esta manera, tendremos los valores para nuestro vector  $y \text{ pred softmax}$ .

- Iteración N°2: Ingresa nuevamente “the” como palabra central en la capa de entrada. Pero ahora se intenta predecir la palabra de contexto “brown” en la capa de salida. Nuevamente ambas palabras son representadas como vectores one hot encoding (vectores  $X$  e  $y\text{true}$  respectivamente): ver Figura 7.20.



- Se realizarán 28 iteraciones más hasta llegar a la iteración N°30, en la cual se ingresará como entrada la palabra central “dog” y como vector  $y$  true tendremos la representación en formato one hot encoding de la palabra “lazy”. De esta manera, al finalizar el paso N°2 tendremos 30 vectores  $y$  pred softmax y 30 vectores  $y$  true con los que comparar.
3. Luego de obtener las distribuciones - $y$  pred softmax- (valor predicho) para cada combinación de palabras (central, contexto), estas son comparadas con los vectores de salida - $y$  true- correspondientes (que tienen las etiquetas reales y al igual que los vectores de entrada están codificados en formato one hot encoding), computándose de esta manera el *error (o loss)* de clasificación (entre el valor predicho y el real) mediante el cálculo de la *función de costo*  $J(\theta)$  (Ver Fórmula 22).

En la Figura 7.21 podemos observar un ejemplo numérico de cómo podría quedar conformado nuestro vector  $y$  pred softmax (donde la suma de los valores dà 1) cuando el vector de entrada es, por ejemplo “the”, y cómo se realizaría la comparación contra el vector de salida  $y$  true cuando la palabra es “quick” para obtener los errores de clasificación. Vemos que el primer resultado de nuestro entrenamiento fue medianamente bueno, ya que en nuestro vector  $y$  pred softmax, en la posición  $y^{'12}$ , obtuvimos el mayor valor. Y esto es lo que queremos lograr, ya que si comparamos este valor con el vector  $y$  true en la misma posición, tiene un 1. La idea es que a medida que la red entrene y minimice la función de costo  $J$ , los valores de  $y$  pred softmax se acerquen cada vez más a los valores de  $y$  true: en nuestro caso que el valor 0,28 sea cada vez más cercanos al 1, y los otros 7 valores que sean cada vez más cercanos al 0.

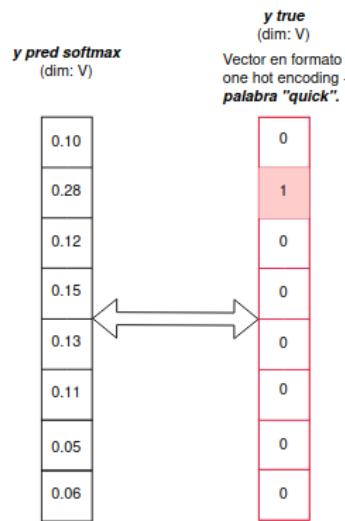


Figura 7.21: Comparando  $y$  pred softmax con  $y$  true.

- Este paso consiste en aplicar el proceso backpropagation utilizando el algoritmo de optimización *descenso del gradiente* para poder ajustar / modificar las matrices  $W_{input}$  y  $W_{output}$  con el objetivo de minimizar la función de costo  $J(\theta)$  obtenida en el paso 3, y con ello minimizar el error de clasificación.
- Se debe volver al paso 2 hasta completar la cantidad de epochs seteados como hiper-parámetro de nuestro modelo.

Finalmente, luego de haber iterado epoch veces sobre nuestro conjunto de entrenamiento y habiendo minimizado (idealmente) la función de costo J en cada iteración, lo que haremos es *obtener la matriz W* (de dimensión  $V \times N$ ), la cual será *nuestra matriz de embeddings*: Figura 7.22 Observamos que, para cada fila de nuestra matriz  $W$ , tendremos el valor del word embedding para la palabra que tenga el mismo índice en nuestro vocabulario: por ejemplo para “the” que tiene índice 1, la 1er fila de  $W$  será su representación en word embedding.

$W_{input}$ ( $V \times N$ )		
W11	W12	W13
W21	W22	W23
W31	W32	W33
W41	W42	W43
W51	W52	W53
W61	W62	W63
W71	W72	W73
W81	W82	W83

Matriz Word embedding  
para la palabra central.

Figura 7.22: Matriz  $W'$  de embeddings objetivo.

De esta manera, entrenamos nuestra red neuronal no con el objetivo de predecir la palabra de contexto a partir de la palabra central, sino con el objetivo de tomar de esa red neuronal su matriz de pesos  $W'$  ya entrenada y utilizarla como nuestra matriz de embeddings.

## 7.6 Palabras repetidas en nuestro Corpus y palabras polisémicas.

A continuación responderemos a dos preguntas muy interesantes: *¿qué sucede en el entrenamiento con las palabras que aparecen más de una vez en nuestro Corpus?* y *¿qué sucede con las palabras polisémicas*<sup>55</sup>?

Word2vec nos permite, en caso que la palabra aparezca  $N$  veces en nuestro corpus de texto, que el modelo aprenda los  $N$  contextos de esa palabra; o dicho de otra forma *permite aprender cuando se usa una misma palabra en diferentes contextos*. Además, si esa palabra tiene  $M$  significados (palabras polisémicas), lo que hará Word2vec es tener en cuenta estos  $M$  significados para armar un word embedding que “promedie” dichos significados.

En el ejemplo descrito en la sección del Anexo *Ejemplo de obtención de Word Embeddings mediante skipgram y softmax* esto ocurre con la palabra “the”, la cual ingresa al modelo 2 veces (siendo usada en 2 contextos distintos). En ese ejemplo, “the” es usada en contexto muy similares, por lo que su significado en este corpus también es similar, ya que ambas veces preceden a un adjetivo (“quick” y “lazy”).

Tomaremos un mejor ejemplo para explicar esto: consideremos que entrenamos nuestro algoritmo con el corpus de texto “She will park the car on the street so we can walk in the central park”. Luego de aplicar un pre-procesamiento y tokenización, nuestro *corpus tokenizado* quedará como: [“park”, “car”, “street”, “walk”, “central”, “park”]. Mientras que nuestro *vocabulario* será: [“park”, “car”, “street”, “walk”, “central”].

Al entrenar nuestro modelo con dicho *corpus tokenizado*, la palabra “park” ingresará al mismo 2 veces -con la misma representación en formato one-hot encoding: [1, 0, 0, 0, 0]-, ya que es usada en dos contextos diferentes. La diferencia con el anterior ejemplo, es que “park” en este corpus de texto tiene 2 significados muy distintos: el primero hace referencia a “estacionar” y el segundo a “parque”.

Al entrenar nuestro Word2vec con este *corpus tokenizado*, y considerando un tamaño de ventana de 1 ( $C = 1$ ), obtendremos lo siguiente:

- En las iteraciones 1 y 2 *el modelo aprenderá que “park” aparece cerca de “car”* (ya que en la iteración N°1 la palabra central / de entrada es “park” y la palabra de contexto / salida es “car”: y para la iteración N°2 la palabra central / de entrada es “car” y las palabras de contexto / salida son “park” y “street”).
- Mientras que en las iteraciones 5 y 6 *el modelo aprenderá que “park” además aparece cerca de “central”* (ya que en la iteración N°5 la palabra central / de entrada es “central” y las palabras de contexto / salida son “walk” y “park”; y para la iteración N°6 la palabra central / de entrada es “park” y la palabra de contexto/salida es “central”).

De esta manera, luego de realizar el entrenamiento, *implícitamente* nuestro modelo Word2vec aprenderá los 2 significados de “park” (debido al entrenamiento en distintos contextos en las iteraciones 1,2,5,6 previamente mencionadas). En conclusión,

---

<sup>55</sup>Footnote: Palabras que tienen más de un significado

obtendremos el vector word embedding para “park” que tendrá en cuenta los  $N$  contextos en lo que se usó dicha palabra y sus  $M$  distintos significados (en este caso  $N$  y  $M = 2$ ). Este vector permitirá representar razonablemente el *significado promedio* de “park” teniendo en cuenta sus 2 significados dentro del corpus (estacionar y parque); tendiendo a moverse en las coordenadas entre estos 2 significados.

## Referencias

- [1] Pinky Sitikhu, Kritish Pahi, Pujan Thapa, & Subarna Shakya. (2019, Octubre). *A Comparison of Semantic Similarity Methods for Maximum Human Interpretability*. IEEE International Conference on Artificial Intelligence for Transforming Business and Society. (pp. 1-4).
- [2] World Economic Forum. (2020, Octubre). *The Future of Jobs Report*. (pp. 29-31).
- [3] Matt Kusner, Yu Sun, Nicholas Kolkin, & Kilian Weinberger. (2015, Julio). *From word embeddings to document distances*. International Conference on Machine Learning. (pp. 957–966).
- [4] Jiapeng Wang, & Yihong Dong. (2020, Agosto). *Measurement of Text Similarity: A Survey*. Information 2020. 11(9). (pp. 1-8, 13,14).
- [5] Baoli Li, & Liping Han. (2013, Octubre). *Distance Weighted Cosine Similarity Measure for Text Classification*. IDEAL2013 Conference. (pp. 1,2).
- [6] Chunjie Luo, Jianfeng Zhan, Lei Wang, & Qiang Yang. (2017, Octubre). *Cosine Normalization: Using Cosine Similarity Instead of Dot Product in Neural Networks*. (pp. 1,2).
- [7] Dani Gunawan, C A Sembiring, & Mohammad Andri Budiman. (2018, Marzo). *The Implementation of Cosine Similarity to Calculate Text Relevance between Two Documents*. 2nd International Conference on Computing and Applied Informatics 2017. Journal of Physics Conference Series. 978(1). (pp. 1-2).
- [8] Derek S. Chapman, & Jane Webster. (2003, Junio-Septiembre). *The Use of Technologies in the Recruiting, Screening, and Selection Processes for Job Candidates*. International journal of selection and assessment. 11(2/3). (pp. 113-114, 117-119).
- [9] Pshdar Abdalla Hamza, Baban Jabbar Othman, Bayar Gardi, Sarhang Sorguli, Hassan Mahmood Aziz, Shahla Ali Ahmed, Bawan Yassin Sabir, Nechirwan Burhan Ismael, Bayad Jamal Ali, & Govand Anwar. (2021, Mayo-Junio). *Recruitment and Selection: The Relationship between Recruitment and Selection with Organizational Performance*. International journal of Engineering, Business and Management (IJEBM). 5(3). (pp. 1-6).
- [10] Luis Argerich, Natalia Golmar, Damián Martinelli, Martín Ramos Mejía, & Juan Andrés Laura. (2019, Enero). *75.06, 95.58 Organización de Datos*. Apunte del Curso Organización de Datos, Universidad de Buenos Aires, Facultad de Ingeniería. (pp. 4-8, 331, 332, 351, 352, 377-379, 387-389, 407, 429-431, 470-477).
- [11] Ladders Company. (2018). *Eye-Tracking Study*. (pp. 2,6).
- [12] Riza Tanaz Fareed, Sharadadevi Kaganurmath, & Rajath V. (2021, Agosto). *Resume Classification and Ranking using KNN and Cosine Similarity*. International Journal of Engineering Research & Technology (IJERT). 10(8). (pp. 192-194).

- [13] Senthil Kumaran V, & Annamalai Sankar. (2013, Mayo). *Towards an automated system for intelligent screening of candidates for recruitment using ontology mapping (EXPERT)*. International Journal of Metadata, Semantics and Ontologies. 8(1). (pp. 56-64).
- [14] Nuno Silva, & Joao Rocha. (2003). *Ontology Mapping for Interoperability in Semantic Web*, IADIS International Conference WWW/Internet (ICWI), Portugal. (pp. 1).
- [15] Wahiba Ben Abdessalem Karaa, & Nouha Mhimdi. (2011) *Using ontology for resume annotation*. International Journal of Metadata, Semantics and Ontologies. 6(3). (pp. 166-174).
- [16] Duygu Çelik, Askýn Karakas, Gülsen Bal, Cem Gültunca, Atilla Elçi, Basak Buluz, & Murat Can Alevli. (2013, Septiembre). *Towards an Information Extraction System Based on Ontology to Match Resumes and Jobs*. Computer Software and Applications Conference Workshops (COMPSACW), IEEE 37th. (pp. 333-338).
- [17] Frank Färber, Tim Weitzel, & Tobias Keim. (2003, Agosto). *An automated recommendation approach to selection in personnel recruitment*. 9th Americas Conference on Information Systems (AMCIS). (pp. 1-11).
- [18] Chirag Daryania, Gurneet Singh Chhabrab, Harsh Patel, Indrajeet Kaur Chhabrad, & Ruchi Patel. (2020). *An Automated Resume Screening System using Natural Language Processing and Similarity*. Topics In Intelligent Computing And Industry Design. 2(2). (pp. 99-103).
- [19] Juneja Afzal Ayub Zubeda, Momin Adnan Ayyas Shaheen, Gunduka Rakesh Narsayya Godavari, & Sayed ZainulAbideen Mohd Sadiq Naseem. (2016, Mayo). *Resume Ranking using NLP and Machine Learning*. Proyecto de Tesis para carrera de grado *Bachiller en Ingeniería*. School of Engineering and Technology Anjuman-I-Islam's Kalsekar Technical Campus. (pp. 1-3).
- [20] Jai Janyani, Kartik Agarwal, & Abhishek Sharma. (2018). *Automated Resume Screening System*. Proyecto de Tesis para carrera de grado *Bachiller en Tecnología*. Rajasthan Technical University. (pp. 5, 9-15).
- [21] V. V. Dixit, Trisha Patel, Nidhi Deshpande, & Kamini Sonawane. (2019, Abril). *Resume Sorting using Artificial Intelligence*. International Journal of Research in Engineering, Science and Management. 2(4). (pp. 423-425).
- [22] Dr. K.Satheesh, A.Jahnavi, L Aishwarya, K.Ayesha, G Bhanu Shekhar, & K.Hanisha. (2020). *Resume Ranking based on Job Description using Spacy NER model*. International Research Journal of Engineering and Technology (IRJET). 7(5). (pp. 74-77).
- [23] Paolo Montuschi, Valentina Gatteschi, Fabrizio Lamberti, Andrea Sanna, & Claudio Demartini. (2014, Septiembre-Octubre). *Job recruitment and job seeking processes: how technology can help*. IT Professional. 16(5). (pp. 41-49).
- [24] Leila Yahiaoui, Zizette Boufaïda, & Yannick Prié. (2006). *Semantic Annotation of Documents Applied to E-Recruitment*. SWAP 2006, the 3rd Italian Semantic Web Workshop. (pp. 1-6).

- [25] Rémy Kessler, Nicolas Béchet, Mathieu Roche, Juan Manuel Torres-Moreno, & Marc El-Bèze. (2012). *A hybrid approach to managing job offers and candidates*. Information Processing & Management. 48(6). (pp. 1124-1135).
- [26] Pradeep Kumar Roy, Sarabjeet Singh Chowdhary, & Rocky Bhatia. (2020). *A Machine Learning approach for automation of Resume Recommendation system*. International Conference on Computational Intelligence and Data Science (ICCIDIS). Procedia Computer Science. 167. (pp. 2318-2327).
- [27] Ioannis Paparrizos, B. Barla Cambazoglu, & Aristides Gionis. (2011). *Machine learned job recommendation*. 5th ACM Conference on Recommender Systems, ACM. (pp. 325-328).
- [28] Paul Resnick, & Hal R. Varian. (1997). *Recommender Systems*. Communications of the ACM40. (pp. 56-59).
- [29] M. Emre Celebi, Michael W. Berry, Azlinah Mohamed, & Bee Wah Yap. (2020). Libro *Supervised and Unsupervised Learning for Data Science*. (pp. 3,4,14,15).
- [30] Pinky Sodhi, Naman Awasthi, & Vishal Sharma. (2019, Enero). *Introduction to Machine Learning and Its Basic Application in Python*. In Proceedings of 10th International Conference on Digital Strategies for Organizational Success. (pp. 1354-1358).
- [31] Karthik Tangirala. (2011). *Semi-supervised and transductive learning algorithms for predicting alternative splicing events in genes*. Proyecto de Tesis para carrera de grado Master of Science. Kansas State University. (pp. 1-4).
- [32] Daniel Berra. (2018). *Cross-Validation*. Libro *Reference Module in Life Sciences*. Data Science Laboratory, Tokyo Institute of Technology. (pp. 1-5).
- [33] S. B. Kotsiantis, D. Kanellopoulos, & P. E. Pintelas. (2006). *Data Preprocessing for Supervised Learning*. International Journal of Computer Science. 1(1). (pp. 111,112,116).
- [34] Stephen Bradshaw, & Colm O'Riordan. (2018). *Evaluating Better Document Representation in Clustering with Varying Complexity*. In Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management 2018. 1. (pp. 199-201).
- [35] Julio-Omar Palacio-Niño, & Fernando Berzal. (2019, Mayo). *Evaluation Metrics for Unsupervised Learning Algorithms*. (pp. 1-6).
- [36] Nathalie Japkowicz. (2006, Mayo). *Why Question Machine Learning Evaluation Methods (An Illustrative Review of the Shortcomings of Current Methods)*. (pp. 1-5)
- [37] Alexei Botchkarev. (2018, Septiembre). *Performance Metrics (Error Measures) in Machine Learning Regression, Forecasting and Prognostics: Properties and Typology*. Interdisciplinary Journal of Information, Knowledge, and Management, 2019. 14. (pp. 1-8).
- [38] Haider Khalaf Jabbar, & Rafiqul Zaman Khan. (2014). *Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)*. (pp. 163-165)

- [39] Zhou Yong, Li Youwen, & Xia Shixiong. (2009, Marzo). *An Improved KNN Text Classification Algorithm Based on Clustering*. Journal of Computers. 4(3). (pp. 230, 233).
- [40] Haneen Arafat Abu Alfeilat, Ahmad B.A. Hassanat, Omar Lasassmeh, Ahmad S. Tarawneh, Mahmoud Bashir Alhasanat, Hamzeh S. Eyal Salman, & V.B. Surya Prasath. (2019, Diciembre). *Effects of Distance Measure Choice on K-Nearest Neighbor Classifier Performance: A Review*. Libro Big Data. 7(4). (pp. 1-9).
- [41] Boyang Li. (2018, Febrero). *An Experiment of K-Means Initialization Strategies on Handwritten Digits Dataset*. Intelligent Information Management. 10. (pp. 43-46).
- [42] Mengyao Cui. (2020). *Introduction to the K-Means Clustering Algorithm Based on the Elbow Method*. Accounting, Auditing and Finance. 1. (pp. 5-8).
- [43] Shraddha Shukla, & Naganna S. (2014). *A Review On K-means Data Clustering Approach*. International Journal of Information & Computation Technology. 4(17). (pp. 1847-1849, 1852, 1853).
- [44] Kodinariya, T.M. (2014). *Survey on Existing Method for Selecting Initial Centroids in K-Means Clustering*. International Journal of Engineering Development and Research. 4(2). (pp. 2865-2868).
- [45] David Arthur, & Sergei Vassilvitskii. (2007, Enero). *K-Means++: The Advantages of Careful Seeding*. SODA '07: Proceedings of the 18th annual ACM-SIAM symposium on Discrete algorithms. (pp. 1027-1035).
- [46] César Menacho Ch. (2013, Diciembre). *Modelos de regresión lineal con redes neuronales (Lineal regression models with neural networks)*. Anales Científicos. 75(2). (2014). (pp. 253-259).
- [47] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, & Roberto Prevete. (2021, Febrero). *A survey on modern trainable activation functions*. (pp. 1-25).
- [48] Damián Jorge Matich. (2001, Marzo). *Redes Neuronales: Conceptos Básicos y Aplicaciones*. Trabajo de Investigación. Cátedra Informática Aplicada a la Ingeniería de Procesos – Orientación I. Universidad Tecnológica Nacional, Facultad Regional Rosario. (pp. 4-28).
- [49] Juan José Montaño Moreno. (2002). *Redes Neuronales Artificiales aplicadas al Análisis de Datos*. Proyecto de Tesis Doctoral. Universitat de les illes balears, Mallorca. Facultad de Psicología. (pp. 17-47).
- [50] Ariel E. Repetur. (2019, Abril). *Redes Neuronales Artificiales*. Trabajo final para carrera Licenciatura en Ciencias Matemáticas. Universidad Nacional del Centro de la Provincia de Buenos Aires, Facultad de Ciencias Exactas. (pp. 5-19,41-43).
- [51] Priya B, Nandhini J.M, & Gnanasekaran T. (2021). *An Analysis of the Applications of Natural Language Processing in Various Sectors*. Smart Intelligent Computing and Communication Technology. (pp. 598-602).
- [52] Antoine Ly, Benno Uthayasooriyar, & Tingting Wang (2020, Octubre). *A survey on Natural Language Processing (NLP) & applications in insurance*. (pp. 1-29).

- [53] Bill Manaris. (1998, Febrero). *Natural Language Processing: A Human–Computer Interaction Perspective*. Advances in Computers. 47. (pp. 1-25).
- [54] Daniel W. Otter, Julian R. Medina, & Jugal K. Kalita. (2019, Diciembre). *A Survey of the Usages of Deep Learning for Natural Language Processing*. IEEE Transactions on neural networks and learning systems. (pp. 1-15).
- [55] N. Vasunthira Devi, & R. Ponnusamy. (2016, Julio). *A Systematic Survey of Natural Language Processing (NLP) Approaches in Different Systems*. International Journal of Computer Sciences and Engineering. 4(7). (pp. 192-197).
- [56] Benjaamin Bengfort, Rebecca Bilbro, & Tony Ojeda. (2018). Libro *Applied Text Analysis with Python. Enabling language-aware data products with machine learning*. O'reilly Media. (pp. 55-67).
- [57] Deepu S., Pethuru Raj., & S.Rajaraajeswari. (2016). *A Framework for Text Analytics using the Bag of Words (BoW) Model for Prediction*. International Journal of Advanced Networking & Applications (IJANA). (pp. 320-323).
- [58] Wenye Li, & Senyue Hao (2019, Noviembre). *Sparse lifting of dense vectors: unifying word and sentence representations*. (pp. 1-9).
- [59] Suhang Wang, Jiliang Tang, Charu Aggarwal, & Huan Liu. (2016, Octubre). *Linked Document Embedding for Classification*. (pp. 115-124).
- [60] Tomas Mikolov, Kai Chen, Greg Corrado, & Jeffrey Dean. (2013, Septiembre). *Efficient Estimation of Word Representations in Vector Space*. (pp. 1-10).
- [61] Eddy Muntina Dharma, Ford Lumban Gaol, Harco Leslie Hendric Spits Warnars, & Benfano Soewito. (2022, Enero). *The accuracy comparison among Word2vec, Glove, and Fasttext towards Convolution Neural Network (CNN) text classification*. Journal of Theoretical and Applied Information Technology. 100(2). (pp. 349-356).
- [62] Thomas Pickard. (2020, Diciembre). *Comparing word2vec and GloVe for Automatic Measurement of MWE Compositionality*. Joint Workshop on Multiword Expressions and Electronic Lexicons. (pp. 95-100).
- [63] Alejandro Cano Cos. (2020, Septiembre). *Aprendiendo vectores de palabras a partir de normas de asociación (Learning word embeddings from word association norms)*. Proyecto de Tesis para carrera de grado *Ingeniería Informática*. Universidad de Cantabria, Facultad de Ciencias. (pp. 11-31).
- [64] Tomas Mikolov, Ilya Sutskever, Kai Chen, Jeffrey Dean, & Greg Corrado. (2013, Octubre). *Distributed Representations of Words and Phrases and their Compositionality*. (pp. N-N).
- [65] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, & Kilian Q. Weinberger. (2015). *From Word Embeddings To Document Distances*. 32nd International Conference on Machine Learning. 37. (pp. 1-8).
- [66] Wael H. Gomaa, & Aly A. Fahmy. (2013, Abril). *A Survey of Text Similarity Approaches*. International Journal of Computer Applications. 68(13). (pp. 13-16).

- [67] Courtney Corley, & Rada Mihalcea. (2005, Junio). *Measuring the Semantic Similarity of Texts*. ACL Workshop on Empirical Modeling of Semantic Equivalence and Entailment. (pp. 13-17).
- [68] Pádraig Cunningham, & Sarah Jane Delany. (2020, Abril). *k-Nearest Neighbour Classifiers 2nd Edition (with Python examples)*. (pp. 1-19).
- [69] Kim Julian Gölle, Nicholas Ford, Patrick Ebel, Florian Brokhausen, & Andreas Vogelsang. (2020, Julio). *Topic Modeling on User Stories using Word Mover's Distance*. (pp. 1-8).
- [70] Dustin S. Stoltz, & Marshall A. Taylor. (2019). *Concept Mover's Distance: Measuring Concept Engagement via Word Embeddings in Texts*. Journal of Computational Social Science. (pp. 1-11).
- [71] Sowmya Vajjala, Bodhisattwa Majumder, Anuj Gupta, & Harshit Surana. (2020). Libro *Practical Natural Language Processing: A Comprehensive Guide to Building Real-World NLP Systems*. 6(17). O'reilly Media. (pp. 92-103).
- [72] Aston Zhang, Zachary C. Lipton, Mu Li, & Alexander J. Smola. (2020). Libro Online *Dive into Deep Learning*. Sitio Web consultado el 15 de Junio de 2022: <https://d2l.ai>. (Capítulo 14 - *Natural Language Processing: Pretraining*).
- [73] BACUDA Project, & CCF-Korea. (2022). Curso Online *Data Analytics – Advanced (HS Recommendation)*. Sitio Web consultado el 15 de Junio de 2022: <https://clikc.wcoomd.org/course/view.php?id=1652>.
- [74] Luciano Robino, Pablo Marinozi, & Matías Battocchia. (2021). Curso Online de Aprendizaje Profundo. Sitio Web consultado el 15 de Abril de 2022: <https://datitos.github.io/curso-aprendizaje-profundo/2021/>.
- [75] Michel Marie Deza, & Elena Deza. (2009). Libro *Encyclopedia of Distances*. Springer. 1. (pp. 1-583).
- [76] Andoni A., Indyk P., & Krauthgamer R. (2008, Enero). *Earth mover distance over high-dimensional spaces*. Symposium on Discrete Algorithms. (pp. 343–352).
- [77] Yossi Rubner, Carlo Tomasi, & Leonidas J. Guibas. (2000). *The Earth Mover's Distance as a metric for image retrieval*. International journal of computer vision. 40(2). (pp. 99-121).
- [78] Lingfei Wu, Ian E.H. Yen, Kun Xu, Fangli Xu, Avinash Balakrishnan, Pin-Yu Chen, Pradeep Ravikumar, & Michael J. Witbrock. (2018, Octubre). *Word mover's embedding: From word2vec to document embedding*. (pp. 1-9).