DESARROLLO DE UNA HERRAMIENTA PARA LA DETECCIÓN Y CLASIFICACIÓN DEL GRADO DE AMBIGÜEDAD EN REQUISITOS DE SOFTWARE MEDIANTE EL USO DE TÉCNICAS DE INTELIGENCIA ARTIFICIAL

Edgar Dario Nova Sanchez

Nota de Aceptación

Certificamos que el presente Trabajo de Grado satisface, en alcances y calidad, todos los requisitos que demanda un Trabajo de Grado de Maestría.

GLORIA INÉS ÁLVAREZ VARGAS

Director

SIMENA DINAS

Jurado 1

JORGE ANDRÉS COLLAZOS NARVAEZ

Jurado 2

Aprobado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana Cali, para optar el título de Magíster en Ingeniería de Software.

HERNÁN CAMILO ROCHA NIÑO

Decano Facultad de Ingeniería y Ciencias

JUAN CARLOS MARTÍNEZ ARIAS

Director Posgrados de Ingeniería y Ciencias

Santiago de Cali, 20 de Septiembre de 2021

Datos del Estudiante

24 de septiembre de 2021

Nombre Completo Édgar Darío Nova Sánchez

Correo Electrónico nova@javerianacali.edu.co, ragnarok540@gmail.com

Profesión Ingeniería de Sistemas, Universidad ICESI, 2013

Empresa Independiente, Desarrollador de Software

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería y Ciencias
Maestría en Ingeniería de Software

DESARROLLO DE UNA HERRAMIENTA PARA LA DETECCIÓN Y CLASIFICACIÓN DEL GRADO DE AMBIGÜEDAD EN REQUISITOS DE SOFTWARE MEDIANTE EL USO DE TÉCNICAS DE INTELIGENCIA ARTIFICIAL

Trabajo de grado para optar al título de Magister en Ingeniería de Software

Estudiante
Édgar Darío Nova Sánchez

Directora
Gloria Inés Álvarez Vargas

30 de julio de 2021

Índice

1. Abstract **3**

2. Resumen **4**

3. Introducción **5**

4. Marco Teórico **7**

4.1. Bases Teóricas 7
4.2. Trabajos Previos 22
4.3. Definición de Términos 23

5. Desarrollo **26**

5.1. Preparación de los Datos 26
5.2. Construcción de los Experimentos 34
5.3. Resumen de Resultados 53
5.4. Implementación de la Herramienta 53

6. Análisis de Resultados **59**

6.1. Análisis de los Experimentos 59
6.2. Análisis General 61

7. Conclusiones y Trabajo Futuro **62**

1. Abstract

The present work proposes the use of supervised learning techniques and weak supervision to classify three levels of ambiguity (low, moderate and high) in software requirements written in natural language. The prepared dataset contains 5.291 software requirements in English, that are labeled according to their syntactic ambiguity using the *link grammar* parser.

The classification performance of different learning models is evaluated, including *random forest* and convolutional neural networks. The best results are obtained using LSTM and GRU recurrent neural networks, with a *F1-Score* of 80 % in the low ambiguity class, 62 % in the moderate ambiguity class and 75 % in the high ambiguity class, and an *accuracy* between 69 % and 71 % of correct classifications of the level of ambiguity.

2. Resumen

El presente trabajo de grado propone el uso de técnicas de aprendizaje supervisado y supervisión débil para realizar la clasificación en tres grados de ambigüedad (baja, moderada y alta), de requisitos de software redactados en lenguaje natural. El conjunto de datos preparado contiene 5.291 requisitos redactados en inglés, que son etiquetados de acuerdo a su ambigüedad sintáctica usando el analizador de *link grammar*.

Se evalúa el desempeño de la clasificación con diferentes modelos de aprendizaje automático que incluyen *random forest* y redes neuronales convolucionales, entre otros. Los mejores resultados se obtienen con los modelos de redes neuronales recurrentes LSTM y GRU, con un *F1-Score* de 80 % en la clase ambigüedad baja, 62 % en la clase ambigüedad moderada y 75 % en la clase ambigüedad alta, y un *accuracy* entre 69 % y 71 % de clasificaciones correctas del grado de ambigüedad.

3. Introducción

El campo de la inteligencia artificial ha avanzado de manera acelerada en las últimas décadas, y ha hecho muchos aspectos de nuestra vida más simples. Como ejemplos de la aplicación de ésta tecnología están los sistemas de recomendación que usan servicios de entretenimiento como *Netflix* y *Spotify*, el reconocimiento de rostros que usan las redes sociales y las cámaras de los teléfonos móviles, las asistentes digitales como la de *Google*, *Alexa* (*Amazon*), *Siri* (*Apple*) o *Cortana* (*Microsoft*), así como otras aplicaciones menos visibles como la detección de fraude por parte de las instituciones financieras y el seguimiento y vigilancia que algunos gobiernos y empresas ejercen sobre sus ciudadanos y empleados, respectivamente.

Estos avances, sumados a la pandemia del Covid-19, que aceleró aún más la adopción de tecnología de todo tipo por parte de organizaciones e individuos (que ahora requieren de Internet para realizar sus actividades, sean laborales, de estudio, de entretenimiento, de compras y para comunicarse con sus seres queridos), hacen que sea necesario mejorar la relación que existe entre usuarios e ingenieros, y una manera de lograr esto puede ser ofreciéndole a los ingenieros de requisitos herramientas que automaticen parte de su proceso.

Existen dos aproximaciones a la especificación de requisitos, los métodos formales y los métodos no formales. Los primeros se fundamentan en el uso de un lenguaje matemático para especificar, desarrollar y verificar sistemas con rigurosidad y son usados en la industria aeronáutica y en el desarrollo de hardware, por ejemplo para verificar el correcto funcionamiento de los microprocesadores.

Si bien los métodos formales son muy completos, tienen la desventaja de ser crípticos e inaccesibles a la gran mayoría de interesados, de manera que no es sorprendente ver que los métodos no formales, basados en la especificación de requisitos en lenguaje natural, son los más usados en casi todas las industrias. La principal desventaja de esta aproximación es la ambigüedad inherente al lenguaje natural.

La ingeniería de requisitos es una de las primeras etapas del desarrollo de software, y entre más avanza un proyecto de este tipo, más costoso resulta resolver errores que surgen debido a los requisitos ambiguos [37]. El problema se evidencia aún más cuando se considera el hecho de que algunos sistemas cuentan con cientos de requisitos [26], y que en un proyecto de mediano o gran tamaño no es realizado por una o dos personas, sino por varios equipos de diferentes especialidades [37].

La ambigüedad aparece como problema cuando un mismo requisito puede ser interpretado de diferentes maneras por cada individuo, entonces es evidente que entre mayor sea el número de interesados que participen en un proyecto y entre más requisitos haya, el problema puede llegar a ser muy difícil de manejar.

Obtener requisitos de calidad es una de las tareas más complejas de la ingeniería de software, por consiguiente mejorar el entendimiento entre los interesados a través de requisitos más claros, precisos y exactos tiene gran potencial debido al ahorro de recursos y esfuerzo que se puede llegar a lograr.

Este proyecto contribuye a la identificación de los requisitos ambiguos mediante la aplicación de técnicas de inteligencia artificial, de manera que el ingeniero de requisitos cuente con una herramienta que le ayude mejorar la redacción de los requisitos de software en base a esta identificación.

En el capítulo 4 de este documento de trabajo de grado se presenta el marco teórico de referencia y los trabajos previos, en el capítulo 5 se detalla el desarrollo de las actividades del proyecto, en el capítulo 6 se analizan los resultados de los modelos y por último en el capítulo 7 se presentan las conclusiones.

4. Marco Teórico

En este capítulo se presentan las bases teóricas y los trabajos previos que sirven como base para la realización del proyecto. Se habla del proceso de ingeniería de requisitos, de la ambigüedad del lenguaje natural en el contexto de la lingüística, de las técnicas de inteligencia artificial que serán aplicadas, así como de los métodos para representar el texto para ser procesado por los algoritmos de aprendizaje automático.

4.1. Bases Teóricas

4.1.1. Ingeniería de Requisitos

La ingeniería de requisitos es el paso más crítico del ciclo de vida del desarrollo de software. Al ser una de las primeras etapas, se debe realizar de la mejor manera, entendiendo que el documento de especificación de requisitos de software es el insumo para los demás procesos del ciclo de vida del software según el estándar *12207* de la IEEE [19]: arquitectura, diseño, implementación, validación y verificación.

Según el estándar complementario *29148* de la IEEE [20], el proceso de ingeniería de requisitos consiste en identificar a los interesados, sean usuarios o clientes, transformar sus necesidades en requisitos mediante técnicas de educción tales como entrevistas, observación o construcción de prototipos, y posteriormente analizar, especificar, documentar, validar y comunicar estos requisitos.

Es claro que esta actividad requiere no sólo habilidades técnicas y conocimiento sobe el negocio y el contexto del desarrollo, sino habilidades en el lenguaje que le permitan a los ingenieros de requisitos establecer una canal de comunicación asertivo con sus clientes.

El estándar *830* de la IEEE [18], describe el contenido y los atributos de un buen documento de especificación de requisitos de software, supone que el resultado del proceso de la especificación es un documento no ambiguo y completo y menciona que

este documento debe:

1. Permitirle a los clientes describir de manera precisa lo que desean obtener de un producto de software.
2. Permitirle a los proveedores de software entender con exactitud lo que el cliente quiere.

Si bien estos objetivos son ideales, es deber del ingeniero de requisitos procurar que se cumplan.

4.1.2. Ambigüedad en el Lenguaje Natural

La ambigüedad es un fenómeno inherente al lenguaje natural, y puede ser definida como la posibilidad de una frase u oración de ser interpretada con más de un significado o intención.

El *Ambiguity Handbook* [2] presenta los siguientes tipos y subtipos de ambigüedad en el lenguaje natural:

- Léxica, cuando una palabra tiene varios significados.
    - Homonimia, cuando dos palabras diferentes tienen la misma representación fonética y escrita. Ej: "Banco" como institución financiera y como un tipo de asiento, en ocasiones sin espaldar.
    - Polisemia, cuando una palabra tiene diferentes significados relacionados, Ej: "Verde", puede hacer referencia al color verde, a algo inmaduro, joven, plantas, etc.
- Sintáctica, o ambigüedad estructural, cuando una secuencia de palabras dada puede tener más de una estructura gramatical.
    - Analítica, cuando el rol de las partes de una frase son ambiguos. Ej. "El profesor de historia de Francia" es ambiguo porque no es claro si el profesor es de origen francés o si es experto en la historia de Francia.

- Adjunta, cuando no hay claridad en a que parte de una oración hace referencia una frase preposicional. Ej. "Los policías le dispararon a los ladrones con armas", no es claro si los ladrones también estaban armados.
- Coordinada, cuando se usa más de una conjunción en una frase. Ej. "*I saw Peter and Paul and Mary saw me*", no es claro si vio a Pedro y Pablo o si Pablo y Maria lo vieron, esto usualmente se resuelve poniendo una coma en el lugar adecuado. En el español esta frase no seria igual de ambigua pues la conjugación de los verbos ayudaría a aclararla.
- Elíptica, ocurre cuando no es claro si se omite información en forma de elipsis. Ej. "Joe Biden conoce a un hombre mas rico que Donald Trump", no es claro si Biden conoce a alguien con mas dinero que Trump o si conoce a alguien con más dinero de los que Trump conoce. En este caso la elipsis es la palabra "conoce" que podría estar al final de la oración.

- Semántica, cuando una oración tiene más de una manera de ser leída en su contexto, si tener ambigüedad léxica o sintáctica.
  - Alcance, ocurre cuando hay un conflicto entre cuantificadores (todos, algunos, uno), además de la negación. Ej. "Todos los lingüistas prefieren una teoría", no es claro si todos prefieren la misma teoría o si cada uno prefiere una teoría diferente.
- Pragmática, ocurre cuando una oración tiene diferentes significados en el contexto en el que es usada.

4.1.3. Inteligencia Artificial

La inteligencia artificial es la inteligencia que puede mostrar una máquina, a diferencia de la inteligencia natural de los seres humanos u otros animales.

Se puede definir un agente inteligente como un sistema capaz de percibir su ambiente, y de llevar a cabo acciones que maximicen la posibilidad de lograr sus

objetivos con éxito [35].

En la literatura científica, el primer aporte sobre inteligencia artificial fue realizado por Turing en 1950 [43], y a lo largo de la historia de esta rama de la ciencia computacional han aparecido diferentes enfoques como los sistemas expertos, la lógica difusa, entre otros, así como grandes crisis en que no se avanzó por mucho tiempo, sin embargo desde el principio de la década de los 90, el aprendizaje automático y las redes neuronales han dado muestras de éxito gracias a los avances en hardware y en los algoritmos que cada vez se acercan más al modo de funcionamiento el cerebro biológico, y que han mostrado ser útiles en varias áreas académicas e industriales, como la visión por computador o los sistemas de recomendación.

4.1.4. Procesamiento de Lenguaje Natural

El procesamiento de lenguaje natural (PLN) es una rama de la inteligencia artificial que trata la interacción entre el lenguaje humano y los computadores, específicamente en como un programa de computador puede procesar y analizar grandes cantidades de datos en lenguaje natural.

Existen varios enfoques de PLN: basado en reglas (métodos simbólicos), basado en estadística, y basado en redes neuronales.

4.1.5. Aprendizaje Automático

El aprendizaje automático, o *machine learning* (ML), es una rama de la inteligencia artificial que trata del estudio de algoritmos que mejoran de manera automática a través de la experiencia [30].

Los algoritmos de ML construyen un modelo matemático a partir de datos de ejemplo, conocidos como datos de entrenamiento, para encontrar patrones, tomar decisiones, hacer clasificaciones o predicciones sin haber sido programados de manera explícita.

Existen tres grandes ramas del aprendizaje automático: Aprendizaje supervisado (se aplica cuando se tienen datos etiquetados y se quiere hacer clasificación o regresión), aprendizaje no supervisado (se aplica cuando los datos no están etiquetados pero se quieren encontrar patrones de agrupamiento) y aprendizaje por refuerzo (estudio de agentes inteligentes que realizan acciones en un ambiente para maximizar una recompensa).

4.1.6. Aprendizaje Supervisado

El aprendizaje supervisado consiste en aprender una función que convierta una entrada en una salida a partir de un conjunto de ejemplos de entradas y salidas [38]. Esta función se aprende a partir de un conjunto de datos de entrenamiento etiquetados, y se evalúa la calidad de la función aprendida con un conjunto de datos de prueba etiquetados [31].

Existen muchos algoritmos de aprendizaje supervisado, entre ellos se encuentran los árboles de decisión y las redes neuronales artificiales, y cada algoritmo tiene ventajas y desventajas, no existe una técnica que funcione mejor en todas las tareas de aprendizaje supervisado.

4.1.7. Supervisión Débil

La supervisión débil o *weak supervision* es el uso de fuentes limitadas, ruidosas, imprecisas o inexactas para etiquetar grandes cantidades de datos para realizar tareas de aprendizaje supervisado [50].

Esta técnica alivia la dificultad de obtener datos etiquetados a mano, dificultad que aparece debido al tamaño de los conjuntos de datos encontrados en la industria, la falta de expertos en el tema para etiquetar los datos, y el poco tiempo con el que se cuenta para preparar los conjuntos de datos [1]. Para el etiquetado automático de los requisitos de software se usó *Link Grammar* como fuente.

4.1.8. Link Grammar

Link Grammar es una teoría de la sintaxis, que construye relaciones entre pares de palabras, en lugar de construir una jerarquía de estructura de frase [42]. El analizador sintáctico de *Link Grammar* tiene la funcionalidad de encontrar un número de posibles "*linkages*" para cada frase que analiza, entonces se puede tomar este número como heurística para identificar que tan ambigua es una frase, entre más "*linkages*", más posibles interpretaciones de cada frase, o requisito de software en el contexto de este proyecto, como se puede observar en la figura 1.

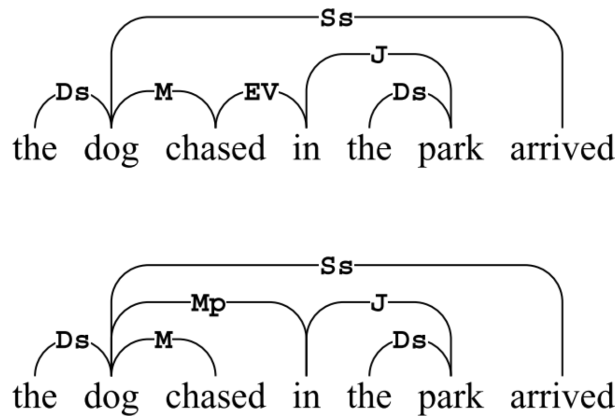

Figura 1: Frase con dos posibles "*linkages*", tomado de [42].

El significado de las etiquetas de los arcos de la figura 1 se presenta a continuación:

Ds Conecta un determinante con el sustantivo correspondiente.

Ss Conecta un verbo con el sustantivo correspondiente.

M/Mp Conecta un verbo usado como un adjetivo (participio) con el sustantivo correspondiente.

EV Conecta una adposición con un verbo.

J Conecta una adposición a su objeto.

La diferencia entre los dos “*linkages*” está en que la primera interpretación da un mayor énfasis sobre donde ocurre el verbo “perseguir”, mientras la segunda da más énfasis a la ubicación del sustantivo “perro”.

El número de “*linkages*” es usado para etiquetar cada requisito para tener una aproximación del nivel de ambigüedad sintáctica del mismo, con la ventaja de que el proceso de etiquetado se puede automatizar, pero con la limitación de no tener requisitos etiquetados por expertos humanos, sin embargo se considera que este etiquetado es aceptable y la dificultad está en definir los umbrales de “*linkages*” para clasificar el nivel de ambigüedad.

4.1.9. Representación del Texto

Los algoritmos de aprendizaje automático no pueden tratar directamente con los requerimientos de software, para aplicarlos, el texto debe ser previamente transformado en una representación numérica que puedan procesar [9]. A continuación se explican brevemente las técnicas aplicadas en el proyecto para lograr este objetivo.

4.1.9.1. Bag-of-Words La representación del texto usando *Bag-of-Words*, o bolsa de palabras, consiste en convertir cada documento en un vector en el que cada posición corresponde a una palabra del vocabulario, entonces si la palabra aparece en el documento se coloca el número de veces que aparece (es decir, su frecuencia), y cero en caso de que la palabra no aparezca [9, 21]. Por ejemplo, considerando las siguientes documentos:

1. *This is the first document.*
2. *This document is the second document.*

Este sería el resultado de aplicar el modelo de representación de bolsa de palabras:

	this	**is**	**the**	**first**	**document**	**second**
1	1	1	1	1	1	0
2	1	1	1	0	2	1

La debilidad de este modelo se encuentra en que al aplicarlo se pierde la estructura de los documentos ya que solo tiene en cuenta la frecuencia de las palabras que los componen. Otra desventaja es que cada vector tiene como dimensión la cantidad de términos del *corpus*, entonces si se tiene un vocabulario muy amplio y muchos documentos, puede que esta no sea la manera más eficiente de representarlos.

A pesar de estas debilidades, este modelo funciona muy bien en tareas que no dependen tanto del orden de las palabras, por ejemplo en el filtrado de correos no deseados (*spam*). Para mejorar el funcionamiento de *bag-of-words*, se tienen las técnicas *n-gram* y *tf-idf*.

4.1.9.1.1. n-gram Los n-gramas son secuencias de n palabras consecutivas que pueden ser usados para representar el texto en lugar de las palabras por sí solas [21]. Esta técnica es una mejora frente a la bolsa de palabras pues conserva el orden de las secuencias y por tanto ofrece información adicional a la frecuencia de las palabras. En base a los documentos de ejemplo del modelo de bolsa de palabras se tienen los siguientes bigramas y trigramas:

1. *This is, is the, the first, first document, This is the, is the first, the first document.*

2. *This document, document is, is the, the second, second document, This document is, document is the, is the second, the second document.*

Se puede observar que el modelo de bolsa de palabras consiste en usar únicamente unigramas para representar el texto.

4.1.9.1.2. tf-idf *Term frequency - inverse document frequency*, (frecuencia de término, frecuencia inversa de documento), es una técnica para darle una calificación de importancia a los términos dependiendo de que tan frecuentes son en los documentos [9, 21, 23], por ejemplo, si la palabra "*the*" es muy común, es decir, aparece en muchos documentos, de manera empírica se puede concluir que no ofrece tanta información como otras palabras menos frecuentes. Esta técnica aparece en el campo de búsqueda y recuperación de información, pero ha demostrado ser efectiva en clasificación de documentos de texto.

Esta técnica de representación se puede combinar con los n-gramas para ofrecer más datos a las técnicas de aprendizaje.

4.1.9.2. Word Embeddings La técnica *Word Embbeding* (encaje de palabras), permite representar palabras o frases mediante vectores de números reales que guardan correspondencia frente a sus significados [21, 34], por ejemplo, el vector de la palabra "*cat*" tendría una menor distancia frente a la palabra "*dog*" comparado con la palabra "*car*", como se observa en la figura 2.

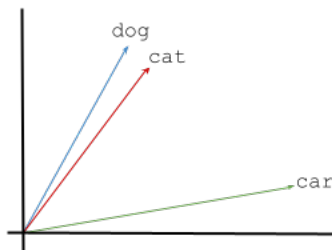

Figura 2: Representación bidimensional de palabras en un espacio vectorial.

Para obtener los vectores existen varios procedimientos, una opción es entrenar redes neuronales con grandes cantidades de texto (por ejemplo todo el contenido de la Wikipedia, sitios de noticias, artículos científicos, libros, etc.) para que aprendan el contexto de las palabras, por ejemplo, es muy probable encontrar frases como "*my*

pet is a cat" y "*my pet is a dog*", pero es muy improbable encontrar "*my pet is a car*", por tanto la red neuronal aprende la similitud entre las palabras y codifica cada palabra en un vector similar.

La ventaja de esta técnica es que reduce el número de dimensiones al momento de representar texto (con *Bag-of-Words* se tiene una dimensión por cada término en el vocabulario) entonces se mejora considerablemente el rendimiento en el aprendizaje. Una desventaja se encuentra con palabras que presentan ambigüedad léxica. Como ejemplos de *Word Embbeding* se tienen *GloVe* y *word2vec*.

4.1.9.2.1. GloVe *Global Vectors* es un modelo desarrollado en la universidad de Stanford para obtener vectores para representar palabras [34]. Se trata de un proyecto de código abierto y ofrece conjuntos de datos ya entrenados listos para ser usados en otras tareas de aprendizaje automático[1]. Los vectores ofrecidos se encuentran en versiones de 50, 100, 200 y 300 dimensiones.

4.1.9.2.2. word2vec Es un modelo desarrollado por investigadores de Google para obtener vectores para representar palabras o frases [28, 29]. La implementación usada en este proyecto es *Wikipedia2Vec* [46], y este proyecto de código abierto ofrece conjuntos de datos ya entrenados en 12 lenguajes y vectores entre 100 y 500 dimensiones[2].

4.1.10. Modelos de Aprendizaje Automático

A continuación se presentan de manera breve los algoritmos de aprendizaje supervisado basados en estadística para reconocimiento de patrones que fueron aplicados al conjunto de datos de requisitos. Para cada modelo se incluye un enlace a un video del canal de YouTube *StatQuest with Josh Starmer*, estos videos sirven como una excelente introducción a cada uno de estos temas.

[1]https://nlp.stanford.edu/projects/glove/

[2]https://wikipedia2vec.github.io/wikipedia2vec/pretrained/

4.1.10.1. Decision Tree Este modelo utiliza un árbol de decisión [38] como modelo de predicción para ir entre observaciones de atributos de un objeto (representadas en las ramas), hasta llegar a una conclusión sobre, por ejemplo, la clase del objeto (representado en las hojas)[3].

4.1.10.2. Random Forest Este modelo de aprendizaje de conjuntos de hipótesis (*ensemble learning*) trabaja usando un gran número de arboles de decisión generados a partir de un subconjunto aleatorio de atributos y datos de entrenamiento [38] . En el caso de tareas de clasificación, el resultado es la clase que la mayoría de los árboles selecciona. El resultado es un modelo menos propenso al *overfitting* y en general más efectivo en el momento de hacer clasificaciones de objetos que no están presentes en el conjunto de entrenamiento frente al uso de árboles de decisión[4].

4.1.10.3. Naïve Bayes Es una familia de clasificadores basados en el teorema de Bayes, que asumen independencia entre las características de las observaciones [38]. Estos modelos son muy eficientes, pues el entrenamiento toma tiempo lineal frente al número de variables de un problema de aprendizaje[5].

4.1.10.4. Logistic Regression El modelo de clasificación de regresión logística usa la función logística para modelar la predicción o clasificación, y a diferencia de Naïve Bayes, no asume que las características de una observación sean independientes, por tanto puede ser más efectivo en tareas de procesamiento de lenguaje natural [31]. No es tan eficiente como Naïve Bayes entonces puede no ser la mejor opción si se deben aprender un gran número de clases[6].

4.1.10.5. Support Vector Machine La máquina de vectores de soporte es un algoritmos de aprendizaje supervisado considerado un modelo de predicción eficiente

---
[3] https://www.youtube.com/watch?v=7VeUPuFGJHk
[4] https://www.youtube.com/watch?v=J4Wdy0Wc_xQ
[5] https://www.youtube.com/watch?v=O2L2Uv9pdDA
[6] https://www.youtube.com/watch?v=yIYKR4sgzI8

[38], y la manera en que consigue clasificar datos consiste en llevar las observaciones a dimensiones mayores donde las clases puedan ser separadas por un hiperplano [7].

4.1.10.6. K-Nearest Neighbors La idea de este modelo es que es probable que los puntos cercanos a una observación tengan características similares a la observación. Estos puntos cercanos se denominan vecinos, y el modelo clasifica la observación tomando la clase más común de los k vecinos más cercanos [38]. Como métrica para medir las distancias entre los puntos puede usarse la distancia euclidiana, aunque existen otras métricas como las distancias de *Mahalanobis* y *Hamming* que puede ser más apropiadas dependiendo de los datos que se están clasificando[8].

4.1.11. Redes Neuronales Profundas

Las técnicas de aprendizaje con redes neuronales profundas o *deep learning* aparecen como alternativa a los modelos de aprendizaje automático previos, y aprovechan de mejor manera conjuntos de datos cada vez más grandes gracias a los avances en la capacidad de computo del hardware moderno (*GPU*, *TPU*, etc.) para realizar el procesamiento en paralelo de estos grandes conjuntos de datos con resultados tales como los logros notables en visión por computador logrados con las redes convolucionales y el procesamiento de lenguaje natural con el modelo *Transformer*.

Estas mejoras se deben a que con el aprendizaje profundo se aprovechan las redes neuronales para realizar el proceso de extracción de características de los datos, este proceso usualmente debe ser realizado de manera manual, pues con los demás modelos sólo se puede automatizar el proceso de clasificación.

4.1.11.1. Redes Convolucionales Las redes neuronales convolucionales (*Convolutional Neural Networks*, CNN) son un tipo de red neuronal profunda usada principalmente para reconocimiento de imágenes que utiliza filtros o *kernels* que permiten reconocer patrones en imágenes, texto, series de tiempo, etc., con la ventaja de que

[7]https://www.youtube.com/watch?v=efR1C6CvhmE
[8]https://www.youtube.com/watch?v=HVXime0nQeI

los filtros hacen parte del aprendizaje y por tanto el proceso de extraer características de los datos es completamente automático.

Un ejemplo de una arquitectura de red convolucional se muestra en la figura 3.

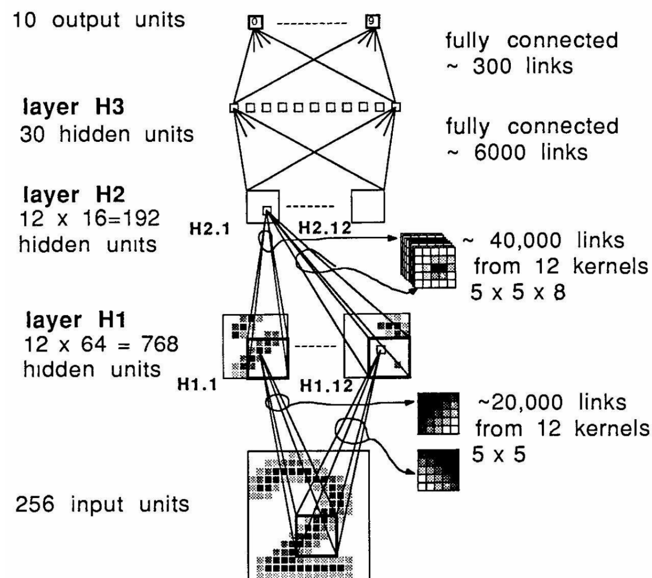

Figura 3: Arquitectura de una CNN para reconocer dígitos, tomado de [22].

Aunque estas redes han logrado resultados en clasificación de imágenes inicialmente con Yann LeCun *et al.* en 1989 [22], actualmente también son usadas para procesar texto, video y se pueden combinar con el aprendizaje por refuerzo en las llamadas *Deep Q-networks* o *DQN*.

4.1.11.2. Redes Recurrentes Las redes neuronales recurrentes (*Recurrent Neural Networks*, RNN) son un tipo de red neuronal profunda usada para procesar datos secuenciales de longitud variable, como texto, sonido y series de tiempo. Su principal diferencia con otras redes neuronales consiste en que las que las salidas de una neurona en el tiempo t es entregado a la misma neurona en el tiempo t+1, es decir,

aparecen conexiones de retroalimentación en estas redes. Existen varias iteraciones de las redes recurrentes, tales como las redes LSTM, las redes bidireccionales y las redes GRU.

4.1.11.2.1. Redes LSTM La arquitectura *long short-term memory* o LSTM, propuesta por Hochreiter y Schmidhuber en 1997 [16], es una mejora a las redes recurrentes pues solucionan en parte el problema de desvanecimiento del gradiente, que ocurre cuando las RNN trabajan con secuencias muy extensas.

4.1.11.2.2. Redes Bidireccionales La arquitectura de RNN bidireccionales consiste en combinar la salida de dos redes recurrentes, una que procesa la secuencia en cuestión de izquierda a derecha y otra que la procesa de derecha a izquierda, de esta manera se supone una mejora en el poder de predicción de la red pues tiene en cuenta no solo el contexto pasado sino además el futuro de cada elemento en la secuencia [39]. Esta arquitectura ha sido especialmente exitosa en combinación con las LSTM [14]. En la figura 4 se muestra un ejemplo de esta arquitectura[9].

4.1.11.2.3. Redes GRU La arquitectura *Gated Recurrent Unit* es una versión simplificada de las LSTM, que tiene el potencial de ser más adecuada para ciertos conjuntos de datos [4, 6].

4.1.11.3. Transformer La arquitectura *Transformer* es un modelo de aprendizaje profundo que adopta el "mecanismo de atención", que trabaja dando diferentes pesos a cada parte de la entrada de la red. A diferencia de las redes recurrentes, que deben procesar su entrada de manera secuencial, el modelo *Transformer* no requiere procesar las secuencias de esta manera pues el mecanismo de atención puede dar contexto a cada elemento de la secuencia de manera independiente. En la figura 5 se muestra un ejemplo de esta arquitectura adaptado a traducción de texto en inglés a alemán.

[9] https://www.tensorflow.org/text/tutorials/text_classification_rnn

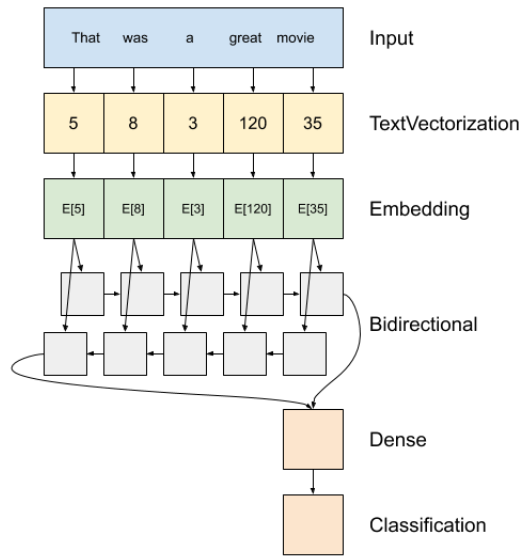

Figura 4: Arquitectura de una RNN bidireccional para clasificación de texto, tomado de la documentación de la librería *Keras*.

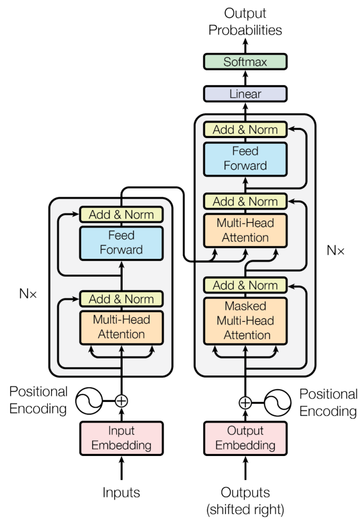

Figura 5: Arquitectura *Transformer* para traducción automática de texto, tomado de [44].

Este modelo es considerado el estado del arte en procesamiento de lenguaje natural, superando las redes LSTM y GRU [44].

4.2. Trabajos Previos

Los trabajos que se resumen a continuación tratan sobre la identificación de ambigüedad en requisitos de software escritos en lenguaje natural mediante el uso de técnicas de inteligencia artificial.

- En 2019 Riaz *et al.* [36], realizó una revisión de algunos de los principales avances en detección de requisitos de software ambiguos, entre los cuales se seleccionaron los siguientes:
    - En 2013 Génova *et al.* [12] propuso un marco de trabajo para evaluar y mejorar la calidad de requisitos textuales de manera automática. Este proyecto utilizó procesamiento de lenguaje natural basado en reglas y cálculo de indicadores a partir de características del texto, y similar al presente trabajo, identificaron tres niveles de calidad de los requisitos.
    - Entre 2010 y 2011 Yang *et al.* [47–49] estudió la ambigüedad en requisitos de software y propuso un marco de trabajo basado en aprendizaje automático para identificar ambigüedad nociva. En estos trabajos se aplicaron modelos tales como SVM, Naïve Bayes, regresión logística, y árboles de decisión. A diferencia del presente trabajo, no se usaron modelos de aprendizaje profundo.
    - Entre 2015 y 2016 Lucassen *et al.* [24,25] propuso un modelo para determinar la calidad de las historias de usuario, el formato de requisitos usado en las metodologías ágiles. Adicionalmente, desarrolló una herramienta que permite mejorar la calidad de las historias de usuario. A diferencia del presente trabajo, este proyecto utilizó procesamiento de lenguaje natural basado en reglas.
    - En 2017 Femmer *et al.* [10] propuso un método para encontrar errores en los requisitos de manera similar a las revisiones estáticas de código fuente.

A diferencia del presente trabajo, este proyecto utilizó procesamiento de lenguaje natural basado en reglas.

- En 2018 Osman *et al.* [32] realizó una comparación entre diferentes algoritmos de aprendizaje automático (Naïve Bayes, regresión logística, KNN y *random forest*), aplicados en la detección de ambigüedad en requisitos de software, con el objetivo de mejorar la calidad de los requisitos. A diferencia del presente trabajo, los requisitos fueron etiquetados por ingenieros en dos niveles de ambigüedad, y no se usaron modelos de aprendizaje profundo.

- En 2015 Shah *et al.* [40] hizo una comparación de diferentes herramientas para la resolución de ambigüedades en requisitos de software, teniendo en cuenta el uso de técnicas de inteligencia artificial. A diferencia del presente trabajo, el enfoque fue el procesamiento de lenguaje natural basado en reglas.

- En 2008 Berzins *et al.* [3] estudió el potencial de la aplicación del procesamiento de lenguaje natural en la ingeniería de requisitos. A diferencia del presente trabajo, sólo se evaluó el potencial de de estas herramientas.

4.3. Definición de Términos

Accuracy Métrica del porcentaje de clasificaciones correctas de un modelo de aprendizaje supervisado.

Bootstrap En el contexto del aprendizaje de conjuntos de hipótesis, es el uso de un subconjunto aleatorio de atributos y datos de entrenamiento obtenido a partir de los datos originales, con esto se espera reducir la posibilidad de *overfitting.*

Corpus Un corpus lingüistico es un conjunto amplio de ejemplos reales del uso de un lenguaje.

Dropout Es una técnica de regularización para reducir el *overfitting* en redes neuronales artificiales, consiste en filtrar de manera aleatoria un porcentaje de las conexiones de la red neuronal durante el proceso de entrenamiento [13].

Early Stopping Es una técnica de regularización para reducir el *overfitting* en redes neuronales artificiales, consiste en detener el entrenamiento de la red neuronal cuando el proceso termina de maximizar (o minimizar) una de las métricas del modelo.

Epoch Es el procesamiento de un conjunto de datos para el entrenamiento de un red neuronal, que normalmente debe ser repetido un número de veces para lograr un buen resultado en el proceso. Se puede usar la técnica *early stopping* para determinar el número adecuado de epocas para entrenar un modelo.

F1-Score Métrica que equivale a la media harmónica de *precision* y *recall*.

GPU Un *Graphics Processing Unit* es circuito especializado para presentar gráficos en una pantalla, y es útil en el contexto de las redes neuronales ya que permite realizar las operaciones matemáticas de las que dependen estos modelos de manera muy eficiente.

Kernel En el contexto de la redes neuronales convolucionales, son los filtros que se usan para aplicar la operación de convolución sobre las entradas. Estos filtros se aprenden en el proceso de entrenamiento de la red neuronal, de modo que la extracción de características se hace de manera automática con esta técnica.

One-hot-enconding Consiste en codificar una variable categórica en un vector de tamaño equivalente al número de categorías, de manera que el vector tenga un 1 en la categoría que le corresponde y 0 en las demás posiciones. El concepto está relacionado con la representación de texto usando *bag-of-words*, y en contexto de las redes neuronales se utiliza para codificar las etiquetas de manera que puedan ser usadas por la capa de salida de estas redes.

Overfitting El sobre-entrenamiento ocurre cuando un modelo de aprendizaje automático captura el ruido de un conjunto de datos en lugar de la estructura general de los mismos, de manera que el modelo resultante no será útil para clasificar observaciones futuras.

Pooling En el contexto de las redes neuronales, consiste en realizar una operación de submuestreo o compresión de la salida de una de las capas de la red antes de entregar esta salida a la siguiente capa.

Precision Métrica del porcentaje de clasificaciones correctas de una clase determinada.

Recall Métrica del porcentaje de elementos clasificados de manera correcta de una clase determinada.

ReLU *Rectified Linear Unit* es una función de activación para redes neuronales profundas que ha mostrado ser mejor que otras como la función logística sigmoidal y la tangente hiperbólica [15]. Estas funciones siguen siendo útiles en casos específicos como en las unidades de las redes LSTM.

Softmax Es una función usada en la capa final de las redes neuronales artificiales para convertir las salidas de la red en probabilidades. La probabilidad más alta (que puede ser determinada aplicando la función *Argmax*) puede ser interpretada como la clasificación de la red.

Stride En el contexto de la redes neuronales convolucionales, es el tamaño de paso que dan los filtros mientras se mueven a través de los datos de entrada para realizar su proceso.

TPU *Tensor Processing Unit* son circuitos diseñados específicamente para el aprendizaje automático con redes neuronales artificiales. Fueron creados por Google para ser usados con la librería *TensorFlow*.

5. Desarrollo

En este capítulo se detalla la manera cómo se construyeron y evaluaron los modelos de aprendizaje supervisado. El capítulo inicia describiendo la preparación de los datos usados, con especial atención a los métodos de etiquetado y representación del texto. A continuación se especifica como se construyeron los modelos y las condiciones generales en las que los experimentos fueron ejecutados. Luego se presentan los resultados obtenidos al evaluar los modelos. Por último se presentan los requerimientos funcionales y no funcionales de la herramienta que permitió ejecutar los experimentos, así como la arquitectura de la misma.

5.1. Preparación de los Datos

En esta sección se hace referencia al origen de los datos de entrenamiento y prueba aprovechados en el desarrollo del proyecto, así como de la manera en que estos requisitos de software fueron etiquetados de forma automática y de como se representan de modo que puedan ser procesados por los algoritmos de aprendizaje automático.

5.1.1. Corpus

Los requisitos de software usados para entrenar y evaluar los modelos fueron obtenidos de las siguientes fuentes:

- En 2017 Ferrari *et al.* [11] construyeron un conjunto de datos de requisitos obtenidos de fuentes públicas en Internet, y una parte de los documentos recolectados fueron convertidos al formato XML para facilitar su uso[10].

- En 2018 Shaukat *et al.* [41] construyeron un conjunto de datos de requisitos para predicción del riesgo en proyectos de software[11].

[10] http://fmt.isti.cnr.it/nlreqdataset/
[11] https://zenodo.org/record/1209601#.YN-4wBNKjmo

- En 2018 Cleland-Huang *et al.* [7] publicaron los requisitos de la herramienta *Dronology* en formato JSON para que fueran aprovechados en otros trabajos[12].
- En 2019 Dalpiaz *et al.* [8] construyeron un conjunto de datos de historias de usuario para detectar ambigüedad de terminología[13].

Todos los requisitos de software usados se encuentran en el idioma inglés.

5.1.1.1. Estadísticas del corpus En el cuadro 1 se muestra el resumen de las principales estadísticas del corpus.

Estadística	Valor
Palabras	109.789
Palabras Diferentes	6.386
Requisitos	5.291
Máximo de Palabras	67
Mínimo de Palabras	5
Promedio de Palabras	20,75

Cuadro 1: Estadísticas del Corpus.

En la figura 6 se muestra la distribución del número de palabras por requisito, es evidente que la mayoría de los requisitos se encuentra entre 10 y 35 palabras y son pocos los que tienen más de 40 palabras.

Las visualizaciones que se incluyen en este documento fueron creadas con las librerías *Matplotlib* [17] y *Seaborn* [45].

5.1.2. Etiquetado

Los requisitos fueron etiquetados de manera automática usando el analizador sintáctico de *Link Grammar*, como se explica en la sección 4.1.8, y en la figura 7 se

[12] https://dronology.info/datasets/
[13] https://data.mendeley.com/datasets/7zbk8zsd8y/1

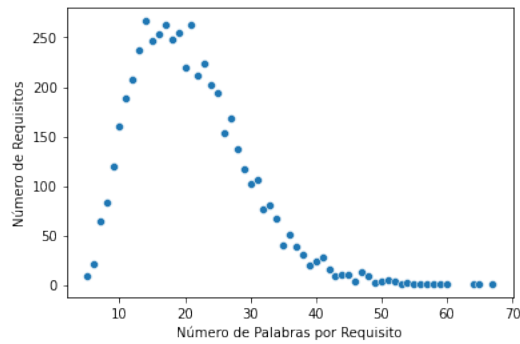

Figura 6: Distribución de palabras por requisito.

muestra la distribución de la cantidad de requisitos por rango de "*linkages*", estos rangos se toman en orden de magnitud, es decir, menor a 10, entre 10 y 100, etc.

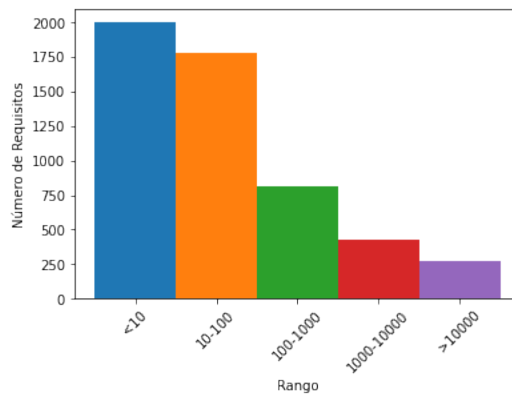

Figura 7: Distribución del número de requisitos en cada rango de "*linkages*".

En la figura 8 se muestran los rangos en porcentajes, y se observa que los dos primeros rangos tienen más de un tercio de los datos cada uno.

A continuación se procede a definir los umbrales para clasificar los requisitos en ambigüedad baja, moderada y alta. Se decide que menos de 11 "*linkages*" hacen que el requerimiento se considere de baja ambigüedad, entre 11 y 112 es ambigüedad

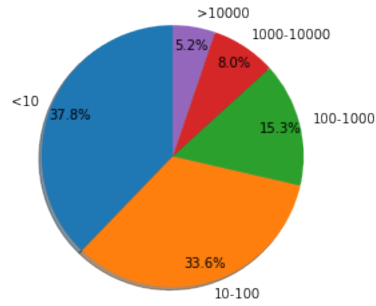

Figura 8: Porcentaje de requisitos en cada rango de "*linkages*".

moderada y más de 112 es ambigüedad alta.

Estos umbrales se definieron luego de analizar un número de requisitos y clasificarlos en estas categorías. En la figura 9 se muestra la distribución del número de requisitos en las tres categorías definidas.

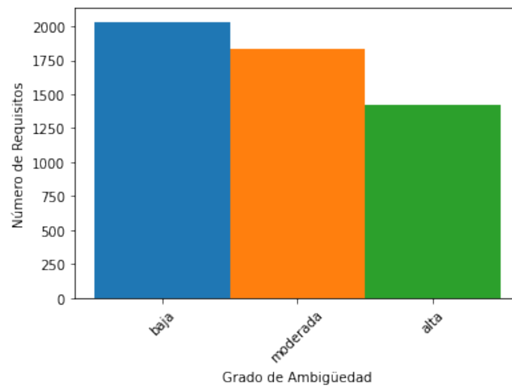

Figura 9: Distribución del número de requisitos en cada clase.

Por último en la figura 10 se muestran los porcentajes por clase, si bien las clases no están balanceadas, se considera que hay suficientes requisitos de cada clase para que los algoritmos de aprendizaje automático puedan obtener resultados.

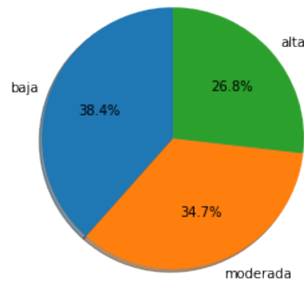

Figura 10: Porcentaje de requisitos en cada clase.

5.1.2.1. Ejemplos de los Requisitos A continuación se presentan tres ejemplos de cada una de las clases de requisito definidas en la sección 5.1.2, ambigüedad baja, moderada y alta.

5.1.2.1.1. Ambigüedad Baja

1. *As a moderator, I want to import stories from a spreadsheet, so that I don't have to copy and paste each individual story.*

2. *The product shall be able to handle up to 1500 simultaneous users.*

3. *The system shall retain existing HIPAA compliance capabilities.*

5.1.2.1.2. Ambigüedad Moderada

1. *As a collection curator, I want to have a summary of all remote replication status at collection level on my dashboard.*

2. *95 % of Adjusters and Collision Estimators shall find the product easy to use.*

3. *The Disputes System must provide different levels of access with regard to disputes case initiation and follow up actions.*

5.1.2.1.3. Ambigüedad Alta

1. *There must be at least two copies one to secure data as acquired and one to do assessment of data quality on line (this last copy preferably on removable media).*

2. *As a anonymous user, I want to see specific details on summits, so that I can learn more about the summit I'm interested in to see if it matches my interests, register for the event, and get day of knowledge to help me get to the location.*

3. *2 out of 3 Program Administrators or Nursing Staff Members shall successfully be able to use the system to manage the scheduling of classes and clinicals.*

5.1.3. Representación de los Requisitos

Como se mencionó en la sección 4.1.9, es necesario preparar los requisitos en lenguaje natural para que puedan ser procesados por los modelos de aprendizaje automático. Para las técnicas de aprendizaje que no están basadas en redes neuronales siempre se usa bolsa de palabras como representación, pero el uso de n-gramas y *tf-idf* es opcional, este proceso se muestra en la figura 11.

En la figura 12 se muestra el proceso de estos modelos basados en redes neuronales profundas, primero se tokenizan los textos y para el *word embedding* se debe elegir entre *GloVe* y *word2vec*. Además de la representación de los requisitos, las técnicas de aprendizaje profundo requieren que las etiquetas se presenten usando *one-hot encoding*.

Por último, los datos de los requisitos son separados en subconjuntos de entrenamiento y validación, tomando el 80 % de los datos para entrenamiento y 20 % para la validación.

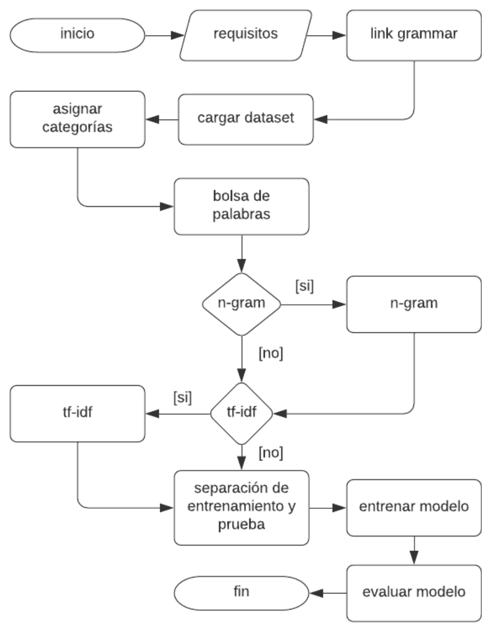

Figura 11: Diagrama de flujo de los modelos que no están basados en redes neuronales.

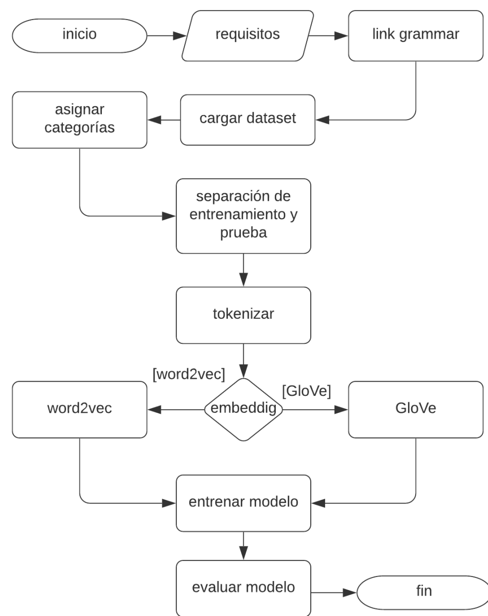

Figura 12: Diagrama de flujo de los modelos de aprendizaje profundo.

5.2. Construcción de los Experimentos

En esta sección se detallan la estimación de los parámetros de los modelos de aprendizaje. En las secciones 5.1.2 y 5.1.3 se explicó la manera como los requisitos son etiquetados y representados, y aquí se especifica cuales técnicas de representación aplican a cada modelo de aprendizaje automático, y en el caso de los modelos basados en redes neuronales, la arquitectura final seleccionada para cada uno de estos modelos.

5.2.1. Estimación de Parámetros de los Experimentos 1, 2, 3 y 4

En los primeros cuatro experimentos se busca la mejor representación del texto para cada uno de los modelos de aprendizaje que no están basados en redes neuronales, y como paso previo se realiza la estimación de algunos parámetros de cada uno de estos modelos como se presenta a continuación.

Decision Tree Se estimaron los parámetros `min_samples_leaf`, el mínimo número de ejemplos para tener un nodo hoja y `min_samples_split`, el mínimo número de ejemplos para dividir un nodo, y el mejor resultado se obtuvo con 1 y 2 respectivamente.

Random Forest Se estimó al parámetro `n_estimators`, el número de árboles que hace parte del bosque aleatorio y se encontró que 150 es un valor adecuado. Al igual que con los árboles de decisión se hizo la estimación de `min_samples_split` y el mejor valor fue 5. Por ultimo, al evitar usar el conjunto de *bootstrap* se mejoró el rendimiento de la modelo.

Naïve Bayes Los parámetros por defecto de este modelo fueron los que mejores resultados presentaron.

Logistic Regression Se estimó el parámetro `max_iter`, el número de iteraciones sobre los datos de entrenamiento y el valor seleccionado fue 1500.

K Nearest Neighbors Se estimó el parámetro del número de vecinos `n_neighbors` y `metric`, el mejor resultado se consiguió con 1 vecino y la distancia de *manhattan*.

Support Vector Machine Se estimó el parámetro `max_iter`, el número de iteraciones sobre los datos de entrenamiento y el valor seleccionado fue 1500.

5.2.2. Experimento 1: Representación de Bolsa de Palabras

En este experimento se ejecutan los algoritmos de aprendizaje que no están basados en redes neuronales usando la representación *bag-of-words* (unigramas). Ver cuadro 2.

bag-of-words		**F1-Score**		
	Accuracy	**Baja**	**Moderada**	**Alta**
Decision Tree	54	63	**45**	51
Random Forest	58	70	44	49
Naïve Bayes	49	58	41	47
Logistic Regression	**61**	**74**	44	**61**
K Nearest Neighbors	42	59	15	6
Support Vector Machine	60	73	41	**61**

Cuadro 2: Resultados de los modelos con los requisitos representados usando bolsa de palabras (unigramas).

5.2.3. Experimento 2: Representación TF-IDF

En este experimento se ejecutan los algoritmos de aprendizaje que no están basados en redes neuronales aplicando *tf-idf* a los unigramas. Ver cuadro 3.

tf-idf		**F1-Score**		
	Accuracy	**Baja**	**Moderada**	**Alta**
Decision Tree	55	64	43	55
Random Forest	**61**	**71**	**47**	**62**
Naïve Bayes	48	59	44	31
Logistic Regression	53	66	38	45
K Nearest Neighbors	49	62	40	34
Support Vector Machine	54	68	17	58

Cuadro 3: Resultados de los modelos aplicando *tf-idf* a la representación en bolsa de palabras de los requisitos.

5.2.4. Experimento 3: Representación N-GRAM

En este experimento se ejecutan los algoritmos de aprendizaje que no están basados en redes neuronales aplicando *n-gram*, usando unigramas, bigramas y trigramas. Ver cuadro 4.

n-gram		**F1-Score**		
	Accuracy	**Baja**	**Moderada**	**Alta**
Decision Tree	54	65	44	49
Random Forest	53	67	38	37
Naïve Bayes	52	60	45	49
Logistic Regression	**62**	**75**	**46**	**59**
K Nearest Neighbors	39	56	6	0
Support Vector Machine	**62**	**75**	45	**59**

Cuadro 4: Resultados de los modelos con los requisitos representados usando unigramas, bigramas y trigramas.

5.2.5. Experimento 4: Representación TF-IDF + N-GRAM

En este experimento se ejecutan los algoritmos de aprendizaje que no están basados en redes neuronales aplicando *tf-idf* a los unigramas, bigramas y trigramas. Ver cuadro 5.

tf-idf + n-gram		**F1-Score**		
	Accuracy	**Baja**	**Moderada**	**Alta**
Decision Tree	57	65	46	59
Random Forest	**65**	**74**	**54**	**67**
Naïve Bayes	48	61	43	21
Logistic Regression	47	62	35	14
K Nearest Neighbors	47	59	38	29
Support Vector Machine	55	68	26	57

Cuadro 5: Resultados de los modelos aplicando *tf-idf* a la representación de los requisitos usando unigramas, bigramas y trigramas.

5.2.6. Experimento 5: Red Neuronal Convolucional

Para construir la arquitectura de la red convolucional se procedió a determinar el número de capas y el tamaño del *kernel*. Se probaron hasta cuatro capas convolucionales con su respectivo tamaño de *kernel* y se seleccionaron 2 y 7 respectivamente, ver cuadro 6.

Para determinar el número de filtros se inició con 100 en cada capa y se fue disminuyendo hasta encontrar el mínimo número que ofrece los mejores resultados. Los valores seleccionados son 16 en la primera capa y 32 en la segunda capa, como se observa en el cuadro 7.

Por último el valor de *dropout* final se seleccionó a partir de las pruebas que se muestran en el cuadro 8, el valor seleccionado es 0.5.

CNN			**F1-Score**		
Capas	**Kernel**	**Accuracy**	**Baja**	**Moderada**	**Alta**
1	9	60	75	51	41
2	7	**67**	75	**58**	**69**
3	5	**67**	**76**	57	68
4	3	66	74	**58**	68

Cuadro 6: Selección del número de capas y el tamaño del kernel de la red convolucional. Las pruebas se ejecutaron con 100 filtros y *dropout* 0.5. El resultado seleccionado es de 2 y 7 respectivamente.

CNN				**F1-Score**	
Filtros C1	**Filtros C2**	**Accuracy**	**Baja**	**Moderada**	**Alta**
100	100	67	75	56	**70**
50	50	**68**	76	57	**70**
25	25	**68**	77	58	68
12	12	**68**	76	58	**70**
9	9	**68**	**78**	58	69
8	8	66	77	52	69
6	6	63	74	42	67
16	32	**68**	76	**60**	**70**
32	16	67	76	57	68

Cuadro 7: Selección del número de filtros de la red convolucional. Las pruebas se ejecutaron con *dropout* 0.5. El resultado seleccionado es de 16 para la primera capa y 32 para la segunda capa.

CNN			**F1-Score**	
Dropout	**Accuracy**	**Baja**	**Moderada**	**Alta**
0.3	65	73	56	68
0.5	**69**	**78**	**60**	**69**
0.7	64	75	43	67

Cuadro 8: Selección del valor de *dropout* de la red convolucional. El resultado seleccionado es 0.5.

Capa	**Salida**
Embedding	100 x 100
Dropout 0.5	100 x 100
Conv1D	32 x 16
Conv1D	9 x 32
GlobalMaxPooling1D	32
Dense	32
Dropout 0.5	32
Dense	3

Cuadro 9: Arquitectura de la red convolucional.

5.2.7. Experimento 6: Red Neuronal Recurrente

Para determinar el número de unidades que la capa recurrente se realizaron pruebas desde 100 unidades, disminuyendo el valor hasta obtener el mínimo valor que ofrece buenos resultados. El valor seleccionado fue 75 como se observa en el cuadro 10.

El valor de *dropout* final se seleccionó a partir de las pruebas que se muestran en el cuadro 11, el valor seleccionado es 0.3 pero se observa que 0.5 también es un valor adecuado.

RNN		**F1-Score**		
Unidades	**Accuracy**	**Baja**	**Moderada**	**Alta**
100	69	78	61	69
75	**70**	**79**	60	**73**
50	**70**	77	**64**	71
25	68	77	59	67

Cuadro 10: Selección del número de unidades de la capa recurrente. Las pruebas se ejecutaron con *dropout* 0.5. El resultado seleccionado es 75.

RNN		**F1-Score**		
Dropout	**Accuracy**	**Baja**	**Moderada**	**Alta**
0.1	67	74	57	70
0.3	**69**	**78**	**61**	68
0.5	68	75	59	**72**
0.7	48	63	13	43

Cuadro 11: Selección del valor de *dropout* de la red recurrente. El resultado seleccionado es 0.3.

Capa	**Salida**
Embedding	100 x 100
Dropout 0.3	100 x 100
SimpleRNN	100 x 75
GlobalMaxPooling1D	75
Dense	75
Dropout 0.3	75
Dense	3

Cuadro 12: Arquitectura de la red recurrente.

5.2.8. Experimento 7: Red Neuronal LSTM

Para determinar el número de unidades que la capa recurrente se realizaron pruebas desde 100 unidades, disminuyendo el valor hasta obtener el mínimo valor que ofrece buenos resultados. El valor seleccionado fue 50 como se observa en el cuadro 13.

Para la red LSTM se decidió no incluir las capas de *dropout* pues a partir de las pruebas que se muestran en el cuadro 14, el mejor resultado se obtuvo cuando estas capas no estaban presentes.

LSTM		**F1-Score**		
Unidades	**Accuracy**	**Baja**	**Moderada**	**Alta**
100	69	77	58	73
75	69	76	58	**75**
50	**71**	78	**62**	74
25	70	**79**	58	74

Cuadro 13: Selección del número de unidades de la capa recurrente LSTM. Las pruebas se ejecutaron con *dropout* 0.5. El resultado seleccionado es 50.

LSTM		**F1-Score**		
Dropout	**Accuracy**	**Baja**	**Moderada**	**Alta**
NA	**70**	**78**	56	**75**
0.1	69	76	**60**	74
0.3	69	75	59	74
0.5	69	**78**	58	73
0.7	69	77	57	73

Cuadro 14: Selección del valor de *dropout* de la red recurrente LSTM. El mejor resultado se obtuvo al no incluir las capas de *dropout*.

Capa	**Salida**
Embedding	100 x 100
LSTM	100 x 50
GlobalMaxPooling1D	50
Dense	50
Dense	3

Cuadro 15: Arquitectura de la red recurrente LSTM.

5.2.9. Experimento 8: Red Neuronal Bidireccional LSTM

Para determinar el número de unidades que la capa recurrente se realizaron pruebas desde 100 unidades, disminuyendo el valor hasta obtener el mínimo valor que ofrece buenos resultados. Como el mejor valor se obtuvo con 100, se incluyo una prueba con 125 unidades. El valor seleccionado fue 100 como se observa en el cuadro 13.

El valor de *dropout* final se seleccionó a partir de las pruebas que se muestran en el cuadro 17, el valor seleccionado es 0.5.

BLSTM		**F1-Score**		
Unidades	**Accuracy**	**Baja**	**Moderada**	**Alta**
125	**70**	**77**	**61**	72
100	**70**	**77**	60	**73**
75	69	76	58	**73**
50	69	76	60	**73**
25	68	73	59	**73**

Cuadro 16: Selección del número de unidades de la capa recurrente bidireccional LSTM. Las pruebas se ejecutaron con *dropout* 0.5. El resultado seleccionado es 100.

BLSTM		**F1-Score**		
Dropout	**Accuracy**	**Baja**	**Moderada**	**Alta**
0.3	68	**76**	58	71
0.5	**69**	73	**62**	72
0.7	67	69	59	**73**

Cuadro 17: Selección del valor de *dropout* de la red recurrente bidireccional LSTM. El resultado seleccionado es 0.5.

Capa	**Salida**
Embedding	100 x 100
Dropout 0.5	100 x 100
Bidirectional + LSTM	100 x 200
GlobalMaxPooling1D	200
Dense	100
Dropout 0.5	100
Dense	3

Cuadro 18: Arquitectura de la red recurrente bidireccional LSTM.

5.2.10. Experimento 9: Red Neuronal GRU

Para determinar el número de unidades que la capa recurrente se realizaron pruebas desde 100 unidades, disminuyendo el valor hasta obtener el mínimo valor que ofrece buenos resultados. El valor seleccionado fue 50 como se observa en el cuadro 19.

El valor de *dropout* final se seleccionó a partir de las pruebas que se muestran en el cuadro 20, el valor seleccionado es 0.5.

5.2.11. Experimento 9: Red Neuronal GRU

GRU		**F1-Score**		
Unidades	**Accuracy**	**Baja**	**Moderada**	**Alta**
100	**70**	78	**59**	72
75	**70**	**79**	56	73
50	**70**	78	**59**	74
25	69	74	58	**75**

Cuadro 19: Selección del número de unidades de la capa recurrente GRU. Las pruebas se ejecutaron con *dropout* 0.5. El resultado seleccionado es 50.

GRU		**F1-Score**		
Dropout	**Accuracy**	**Baja**	**Moderada**	**Alta**
0.3	**70**	**77**	61	72
0.5	**70**	73	**63**	**75**
0.7	38	55	0	0

Cuadro 20: Selección del valor de *dropout* de la red recurrente GRU. El resultado seleccionado es 0.5.

Capa	**Salida**
Embedding	100 x 100
Dropout 0.5	100 x 100
GRU	100 x 50
GlobalMaxPooling1D	50
Dense	50
Dropout 0.5	50
Dense	3

Cuadro 21: Arquitectura de las red recurrente GRU.

5.2.12. Experimento 10: Red Neuronal Transformer

Para determinar el número de bloques se hicieron pruebas con 1, 2 y 3 bloques, como se ve en el cuadro 22 el mejor resultado se obtuvo con 2 bloques. Luego se escogió el número de *Attention Heads* de cada bloque, y los mejores resultados se vieron con solo 1 en cada bloque, como se observa en el cuadro 23.

Para determinar el número de unidades que las capas ocultas se realizaron pruebas desde 100 unidades, disminuyendo el valor hasta obtener el mínimo valor que ofrece buenos resultados. El valor seleccionado fue 75 como se observa en el cuadro 24.

El valor de *dropout* final se seleccionó a partir de las pruebas que se muestran en el cuadro 25, el valor seleccionado es 0.3 pero se observa que 0.1 también es un valor adecuado.

Transformer		**F1-Score**		
Bloques	**Accuracy**	**Baja**	**Moderada**	**Alta**
1	69	77	58	**73**
2	**70**	77	**62**	**73**
3	68	**78**	58	69

Cuadro 22: Selección del número de bloques de la red *Transformer*. Las pruebas se ejecutaron con *dropout* 0.1 y 2 *Attention Heads*. El resultado seleccionado es 2.

Transformer			**F1-Score**		
Heads B1	**Heads B2**	**Accuracy**	**Baja**	**Moderada**	**Alta**
1	1	69	**77**	**62**	69
2	2	69	76	59	**72**
3	3	69	76	60	71

Cuadro 23: Selección del número de *Attention Heads* de la red *Transformer*. Las pruebas se ejecutaron con *dropout* 0.1. El resultado seleccionado es de 1 en ambos bloques.

Transformer			**F1-Score**	
Unidades	**Accuracy**	**Baja**	**Moderada**	**Alta**
100	**70**	77	59	73
75	**70**	**78**	**61**	71
50	69	74	**61**	**74**
25	69	76	59	**74**

Cuadro 24: Selección del número de unidades de la red *Transformer*. Las pruebas se ejecutaron con *dropout* 0.1. El resultado seleccionado es 75.

Transformer			**F1-Score**	
Dropout	**Accuracy**	**Baja**	**Moderada**	**Alta**
NA	68	77	58	69
0.1	**69**	77	**61**	69
0.3	**69**	**78**	**61**	69
0.5	68	75	59	**72**

Cuadro 25: Selección del valor de *dropout* de la red *Transformer*. El resultado seleccionado es 0.3.

Capa	**Salida**
Token and Position Embedding	100 x 100
Transformer Block	100 x 100
GlobalAveragePooling1D	100
Dropout 0.3	100
Dense	75
Dropout 0.3	75
Dense	3

Cuadro 26: Arquitectura de las red *Transformer*.

5.2.13. Experimento 11: Comparación de Redes Neuronales

Se realizaron pruebas de los modelos de redes neuronales profundas con los parámetros definidos en los experimentos anteriores usando las dos representaciones de las palabras como vectores, como se muestra en los cuadros 27 para *GloVe*, y 28 para *word2vec*.

GLOVE		**F1-Score**		
	Accuracy	**Baja**	**Moderada**	**Alta**
CNN	68	**76**	57	71
RNN	**69**	75	59	**74**
LSTM	**69**	**76**	61	71
BLSTM	**69**	75	61	72
GRU	**69**	75	**63**	69
Transformer	**69**	**76**	61	**74**

Cuadro 27: Resultados de los modelos de redes neuronales profundas aplicando *GloVe* a la representación de los requisitos.

WORD2VEC		**F1-Score**		
	Accuracy	**Baja**	**Moderada**	**Alta**
CNN	67	76	60	67
RNN	69	78	57	71
LSTM	69	78	58	73
BLSTM	69	76	**61**	72
GRU	**71**	**80**	58	**74**
Transformer	69	76	60	72

Cuadro 28: Resultados de los modelos de redes neuronales profundas aplicando *word2vec* a la representación de los requisitos.

5.3. Resumen de Resultados

En el cuadro 29 se resumen los mejores resultados de cada uno de los modelos evaluados. Para los modelos que no están basados en redes neuronales se incluye la representación de los requisitos usada, pero no se incluye para los modelos de redes neuronales profundas pues no se considera que sea un parámetro que afecte de manera considerable los resultados.

		F1-Score		
	Accuracy	**Baja**	**Moderada**	**Alta**
DT (tf-idf+n-gram)	57	65	46	59
RF (tf-idf+n-gram)	65	74	54	67
NB (n-gram)	52	60	45	49
LR (n-gram)	62	75	46	59
KNN (tf-idf)	49	62	40	34
SVM (n-gram)	62	75	45	59
CNN	69	78	60	69
RNN	69	78	61	68
LSTM	70	78	56	**75**
BLSTM	69	73	**62**	72
GRU	**71**	**80**	58	74
Transformer	69	78	61	69

Cuadro 29: Resumen de los mejores resultados de todos los modelos evaluados.

5.4. Implementación de la Herramienta

La herramienta desarrollada tiene el propósito de analizar el corpus de requerimientos, transformarlo en la representación adecuada que requiere cada modelo de aprendizaje, realizar el proceso de aprendizaje con el conjunto de datos de entrenamiento y por último reportar las métricas de rendimiento de los modelos con el conjuntos de datos de prueba.

A continuación se presentan los requisitos funcionales y no funcionales de la herramienta desarrollada. Los requerimientos funcionales se presentan en el formato

de historia de usuario, reflejando la naturaleza iterativa de este proyecto. Al final de la sección se especifican los detalles de la implementación, las funciones y clases de las librerías que fueron aprovechadas.

5.4.1. Requisitos Funcionales

1. Cómo desarrollador, quiero etiquetar los requisitos de manera automática.
2. Cómo desarrollador, quiero cargar el conjunto de datos de requisitos a clasificar.
3. Cómo desarrollador, quiero definir los umbrales para clasificar requisitos en ambigüedad baja, moderada o alta.
4. Cómo desarrollador, quiero identificar como se encuentran balanceados los datos en las tres clases definidas.
5. Cómo desarrollador, quiero representar el texto de los requisitos de tal manera que pueda ser procesado por los algoritmos de aprendizaje automático seleccionados.
6. Cómo desarrollador, quiero separar los datos en subconjuntos de entrenamiento y pruebas.
7. Cómo desarrollador, quiero aplicar técnicas de aprendizaje automático con los datos de los requisitos.
8. Cómo desarrollador, quiero aplicar técnicas de aprendizaje profundo con los datos de los requisitos.
9. Cómo desarrollador, quiero evaluar el rendimiento de las diferentes técnicas aplicadas y de esta manera encontrar cuales son las más adecuadas para clasificar la ambigüedad.

5.4.2. Requisitos no Funcionales

1. La herramienta debe usar tecnologías de ciencia de datos estándar y de código abierto.

2. La herramienta debe ser de código abierto, para que otros investigadores puedan replicar los experimentos y construir a partir de estos.

5.4.3. Arquitectura de la Herramienta

Para llevar a cabo el proyecto se utilizaron las siguientes herramientas:

GitHub Repositorio de código que usa el sistema de control de versiones *git*, se utilizó para almacenar y publicar tanto el código de la herramienta como el conjunto de datos de requisitos[14].

Google Colab Plataforma de desarrollo en línea para proyectos de ciencia de datos ofrecida por Google que permite ejecutar de manera interactiva el lenguaje de programación Python así como todo el ecosistema de librerías de inteligencia artificial, aprendizaje automático, manipulación y visualización de datos que a surgido al rededor del lenguaje[15].

5.4.4. Detalles de la Implementación

La versión de Python con la que cuenta Google Colab es la 3.7.11, la libreria *Keras* se encuentra en la versión 2.4.3, *Tensorflow* en la 2.5.0, *scikit-learn* en la 0.22.2, *Pandas* en la 1.1.5, *matplotlib* en la 3.2.2, y *Seaborn* en la 0.11.1.

El código fuente de los experimentos se encuentra en el repositorio de Github que aparece en el pie de página[16]. En este repositorio tambien se incluye el conjunto de datos de requisitos en el archivo `tags.txt`.

[14] https://github.com/
[15] https://colab.research.google.com/
[16] https://github.com/Ragnarok540/pdg

A continuación se hace referencia a las funciones y clases de las librerías que fueron aprovechadas para implementar los experimentos.

5.4.4.1. Preparación de los Datos

train-test split La función `sklearn.model_selection.train_test_split` fue aprovechada para separar el corpus en los subconjuntos de entrenamiento y pruebas.

corpus Se usó la función `pandas.read_csv` de la librería *Pandas* [27], además de otras funciones, para cargar en memoria y preparar los datos del corpus.

one-hot enconding Se usó `sklearn.preprocessing.LabelBinarizer` para codificar las etiquetas de los requisitos para los modelos de redes neuronales.

5.4.4.2. Representación del Texto

bag-of-words La clase `sklearn.feature_extraction.text.CountVectorizer` de la librería *scikit-learn* [33], permite obtener los vectores para la representación de bolsa de palabras.

n-gram La clase `sklearn.feature_extraction.text.CountVectorizer` tiene el parámetro opcional `ngram_range` que permite seleccionar el rango de n-gramas que se desea generar. Por defecto la clase genera unigramas.

tf-idf Para convertir los vectores de bolsa de palabras en la representación *tf-idf* se uso la clase `sklearn.feature_extraction.text.TfidfTransformer`.

word embedding La clase `keras.layers.Embedding` de la librería *Keras* [5].

5.4.4.3. Modelos

Decision Tree Se usó la clase `sklearn.tree.DecisionTreeClassifier` para implementar el modelo.

Random Forest Se usó la clase `sklearn.ensemble.RandomForestClassifier` para implementar el modelo.

Naïve Bayes Se usó la clase `sklearn.naive_bayes.MultinomialNB` para implementar el modelo. La librería tiene otras versiones de este modelo, y esta es la más adecuada para clasificar texto.

Logistic Regression y SVM Para implementar ambos modelos se usó la clase `sklearn.linear_model.SGDClassifier` .

KNN Se usó la clase `sklearn.neighbors.KNeighborsClassifier` para implementar el modelo.

Redes Neuronales Se usó la clase `keras.models.Sequential`.

Capas Convolucionales Se usó la clase `keras.models.Conv1D`.

Capas Recurrentes Se usó la clase `keras.layers.SimpleRNN`.

Capas Recurrentes LSTM Se usó la clase `keras.layers.LSTM`.

Capas Recurrentes GRU Se usó la clase `keras.layers.GRU`.

Capas Bidireccionales Se usó la clase `keras.layers.Bidirectional`.

Capas Dropout Se usó la clase `keras.layers.Dropout`.

Modelo Transformer Se usó la clase `keras.layers.MultiHeadAttention`[17].

Capas Ocultas Se usó la clase `keras.models.Dense` con la función de activación *ReLU* para las capas ocultas de clasificación.

Capas de Salida Se usó la clase `keras.models.Dense` con la función de activación *softmax* para las capas de salida de clasificación.

Capas de Pooling Se usó las clase `keras.layers.GlobalMaxPooling1D` para hacer *pooling* en las redes neuronales convolucionales y recurrentes, y la clase `keras.layers.GlobalAveragePooling1D` para el modelo *transformer*.

[17]https://keras.io/examples/nlp/text_classification_with_transformer/

5.4.4.4. Entrenamiento de los Modelos

Early Stopping Al entrenar los modelos se usó `keras.callbacks.EarlyStopping` para aplicar la técnica *early stopping*. La variable que se monitorizó fue el *recall* de validación y se ejecutó con un valor de 5 *epochs* en el parámetro `patience`. Gracias a la aplicación de esta técnica, se encuentra el número óptimo de *epochs* para el proceso de entrenamiento, que es diferente para cada modelo.

Estimación de Parámetros Para realizar la estimación de parámetros de cada uno de los modelos que no están basados en redes neuronales se aprovechó la clase `sklearn.model_selection.GridSearchCV`.

5.4.4.5. Evaluación de los Modelos

Matriz de Confusión Se usó `sklearn.metrics.confusion_matrix` para visualizar el desempeño de los modelos.

Reporte de Clasificación Se usó `sklearn.metrics.classification_report` para obtener las métricas de clasificación *accuracy* y el *F1-Score* de cada clase.

6. Análisis de Resultados

En esta sección se analizan los resultados de construcción de los modelos en la sección 5.2, y la evaluación de los modelos que se presentó en la sección 5.3.

6.1. Análisis de los Experimentos

6.1.1. Experimentos 1, 2, 3 y 4: Modelos de Aprendizaje Automático

Para los modelos que no están basados en redes neuronales es evidente como cada una de las posibles representaciones del texto afecta significativamente los resultados de la clasificación de requisitos. La representación *bag-of-words* por si sola no es la mejor para ninguna de las técnicas probadas. El uso de *tf-idf* mejora el rendimiento de los árboles de decisión, *random forest* y k-vecinos más cercanos, pero disminuye para las demás técnicas. La aplicación de *n-gram* mejora el rendimiento de los clasificadores lineales: *Naïve Bayes*, regresión logística y máquina de vectores de soporte, y disminuye para las otras. El mejor resultado obtenido se encuentra al combinar las dos técnicas de representación como entrada para *random forest*, con 65 % de *accuracy*.

6.1.2. Experimento 5: Red Neuronal Convolucional

Aunque las redes neuronales convolucionales son usadas principalmente para clasificar imágenes, el resultado de este experimento muestra que se puede lograr un rendimiento comparable con los de las redes neuronales recurrentes para clasificar texto. La dificultad con esta aproximación consiste en determinar la arquitectura de la red y los parámetros de la misma, es decir, el número de capas convolucionales, si se debe aplicar *pooling* entre estas capas, el tamaño del *kernel*, el número de filtros de cada capa, el porcentaje de *dropout*, etc. El mejor resultado obtenido con esta técnica fue un 69 % de *accuracy*.

6.1.3. Experimento 6: Red Neuronal Recurrente

Comparada con las redes neuronales convolucionales, la arquitectura de las redes recurrentes es mucho más sencilla. Para realizar la clasificación sólo se usó una capa de unidades recurrentes, se determinó el número de unidades y el porcentaje de *dropout* y se obtuvieron resultados entre 69 % y 70 % de *accuracy*.

6.1.4. Experimento 7: Red Neuronal LSTM

Este tipo de red neuronal recurrente da resultados comparables con los del experimento 6, usando menos unidades que las del experimento mencionado, y se observa que no requiere capas de *dropout* para obtener resultados entre 69 % y 71 % de *accuracy*.

6.1.5. Experimento 8: Red Neuronal Bidireccional LSTM

La red neuronal bidireccional LSTM obtuvo resultados similares a los experimentos 6 y 7, sin embargo, para lograrlo, requiere más unidades y el tiempo de entrenamiento y prueba y casi cuatro veces más alto que el de los experimentos mencionados. Al final se obtuvieron resultados entre 69 % y 70 % de *accuracy*.

6.1.6. Experimento 9: Red Neuronal GRU

Los experimentos de este tipo de red recurrente también dieron resultados similares a los experimentos 6 y 7, el número de unidades fue el mismo que se uso para las redes LSTM, pero si se justificó el uso de capas de *dropout*. Al final se obtuvieron resultados entre 69 % y 70 % de *accuracy*.

6.1.7. Experimento 10: Red Neuronal Transformer

Similar al experimento 5, este tipo de red tiene muchos parámetros que se deben tener en cuenta al momento de definir la arquitectura del modelo (número de bloques *transformer*, número de *attention heads* de cada bloque, etc.). Esto, sumado al hecho que la librería aún no implementa la arquitectura directamente hace que sea compleja

su aplicación. Los resultados del experimento son similares a los obtenidos con las demás técnicas, entre 69 % y 70 % de *accuracy*.

6.1.8. Experimento 11: Comparación de Redes Neuronales

En este experimento se comparó el rendimiento de todos los modelos de redes neuronales profundas implementados usando las dos opciones de representación disponibles. Si bien no hay diferencias significativas en la métrica *accuracy*, el promedio del *F1-Score* de la clase ambigüedad baja es 77 % para *GloVe* comparado con 76 % para *word2vec*, y el promedio del *F1-Score* de la clase ambigüedad moderada es 60 % para *word2vec* comparado con 59 % para *GloVe*. El promedio del *F1-Score* es el mismo para la clase ambigüedad alta, 72 %.

6.2. Análisis General

Con el presente trabajo se muestra que aplicar técnicas de aprendizaje supervisado puede ayudar a detectar la ambigüedad en requisitos de software, sin embargo no resuelve el problema por completo, el sistema propuesto se limita a indicarle al ingeniero de requisitos los puntos donde debe prestar atención para mejorar la redacción y la calidad en general de los requisitos.

Los resultados obtenidos con las diferentes redes neuronales se encuentran en un rango de *accuracy* muy similar, entre 69 % y 71 %, este hecho puede ser interpretado como si este fuera el límite que se puede lograr con el tamaño del conjunto de datos actual y la manera en que fueron etiquetados. Agregar más requisitos al conjunto de datos y ajustar los umbrales de etiquetado podría ser un punto de partida para un trabajo futuro.

Para finalizar, se recomienda el uso de redes neuronales recurrentes LSTM para clasificar los requisitos de acuerdo a su ambigüedad, debido a que con esta arquitectura relativamente sencilla se lograron los mejores resultados.

7. Conclusiones y Trabajo Futuro

Con los experimentos ejecutados de este proyecto, se logró clasificar el nivel de ambigüedad de requisitos de software con un *accuracy* entre 69 % y 71 % con el uso de redes neuronales recurrentes, LSTM y GRU.

Se lograron resultados comparables con las redes convolucionales y *transformer*, pero la complejidad de implementar estos modelos, en especial *transformer*, hace que esta aproximación no sea la más adecuada para los datos en cuestión, al menos con las herramientas que actualmente ofrecen las liberías.

Aunque se esperaría que para el problema de la ambigüedad en requisitos las redes recurrentes bidireccionales fueran especialmente efectivas, este no fue el caso, el uso de esta técnica no mejoró los resultados sino que los modelos obtenidos fueron mucho más lentos en el proceso de entrenamiento.

No se encontraron diferencias entre el uso de *word2vec* y *GloVe* en la representación de texto, de modo que se recomienda el uso de *GloVe* pues es más accesible que la alternativa, ya que los vectores ya entrenados pueden ser descargados directamente en la página web de los desarrolladores. Para *word2vec* no se encontraron vectores ya entrenados en 50 dimensiones, de modo que fue necesario usar como mínimo 100 dimensiones y realizar un procesamiento adicional de los archivos para que siguieran el formato del archivo de *GloVe* y pudieran ser comparados.

De los algoritmos que no están basados en redes neuronales el mejor resultado se consiguió con *random forest*, con 65 % de *accuracy*, y se hace necesario usar tanto *tf-idf* como *n-gram* en la representación del texto para lograr este resultado.

La principal dificultad encontrada en el proyecto fue la falta de datos etiquetados. Aunque se encontraron suficientes requisitos de software, se hizo necesario buscar una manera de etiquetarlos. Se encontró que el uso de *link grammar* como fuente para

etiquetar de manera automática, y de esta manera aplicar supervisión débil fue efectivo para obtener resultados interesantes.

Como trabajo futuro se podría ampliar la base de datos de requisitos de software, además de validar con un gran número de ingenieros de software expertos para optimizar los umbrales para determinar el nivel de ambigüedad de los requisitos, así como realizar pruebas con diferentes números de clases que las frente a las tres usadas en este proyecto.

Referencias

[1] Stephen Bach, Daniel Rodriguez, Yintao Liu, Chong Luo, Haidong Shao, Cassandra Xia, Souvik Sen, Alex Ratner, Braden Hancock, Houman Alborzi, Rahul Kuchhal, Chris Ré, and Rob Malkin. Snorkel DryBell: A Case Study in Deploying Weak Supervision at Industrial Scale. In *Procceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 362–375, 2019.

[2] Daniel Berry, Erik Kamsties, and Michael Krieger. *From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity: A Handbook*. University of Waterloo, 2003.

[3] Valdis Berzins, Craig Martell, Luqi, and Paige Adams. Innovations in Natural Language Document Processing for Requirements Engineering. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5320 LNCS:125–146, 2008.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734, 2014.

[5] François Chollet. Keras - https://keras.io, 2015.

[6] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. In *NIPS 2014 Deep Learning and Representation Learning Workshop*, pages 1–9, 2014.

[7] Jane Cleland-Huang, Michael Vierhauser, and Sean Bayley. Dronology: An Incubator for Cyber-Physical Systems Research. In *2018 ACM/IEEE 40th Inter-*

national Conference on Software Engineering: New Ideas and Emerging Results, pages 109–112, 2018.

[8] Fabiano Dalpiaz, Ivor van der Schalk, Sjaak Brinkkemper, Fatma Başak Aydemir, and Garm Lucassen. Detecting Terminological Ambiguity in User Stories: Tool and Experimentation. *Information and Software Technology*, 110:3–16, 2019.

[9] Ronen Feldman and James Sanger. *The Text Mining Handbook: Advanced Approaches in Analizing Unstructured Data*. Cambridge University Press, 2007.

[10] Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid Quality Assurance with Requirements Smells. *Journal of Systems and Software*, 123:190–213, 2017.

[11] Alessio Ferrari, Giorgio Oronzo Spagnolo, and Stefania Gnesi. PURE: A Dataset of Public Requirements Documents. In *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference RE 2017*, pages 502–505, 2017.

[12] Gonzalo Génova, José M. Fuentes, Juan Llorens, Omar Hurtado, and Valentín Moreno. A Framework to Measure and Improve the Quality of Textual Requirements. *Requirements Engineering*, 18(1):25–41, 2013.

[13] Xavier Glorot, Antoine Borde, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[14] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3697 LNCS:799–804, 2005.

[15] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors. *Computing Research Repository (CoRR)*, pages 1–18, 2012.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[17] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science and Engineering*, 9(3):90–95, 2007.

[18] IEEE. 830 - Recommended Practice for Software Requirements Specifications. Technical report, 1998.

[19] ISO/IEC/IEEE. Systems and Software Engineering — Software Life Cycle Processes. *ISO/IEC/IEEE 12207:2017*, 2017.

[20] ISO/IEC/IEEE. Systems and Software Engineering — Life Cycle Processes — Requirements Engineering. *ISO/IEC/IEEE 29148:2018*, 2018.

[21] Dan Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, 3rd edition, 2021.

[22] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 1989.

[23] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 3rd edition, 2019.

[24] Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E.M. Van Der Werf, and Sjaak Brinkkemper. Forging High-Quality User Stories: Towards a Discipline for Agile Requirements. *2015 IEEE 23rd International Requirements Engineering Conference, RE 2015 - Proceedings*, pages 126–135, 2015.

[25] Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E.M. van der Werf, and Sjaak Brinkkemper. Improving Agile Requirements: The Quality User Story Framework and Tool. *Requirements Engineering*, 21(3):383–403, 2016.

[26] Qiao Ma. The Effectiveness of Requirements Prioritization Techniques for a Medium to Large Number of Requirements: A Systematic Literature Review. *Auckland University of Technology*, 2009.

[27] Wes McKinney. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference*, 1:56–61, 2010.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *1st International Conference on Learning Representations ICLR 2013 - Workshop Track Proceedings*, pages 1–12, 2013.

[29] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems*, pages 1–9, 2013.

[30] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[31] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2nd edition, 2018.

[32] Mohd Hafeez Osman and Mohd Firdaus Zaharin. Ambiguous Software Requirement Specification Detection: An Automated Approach. In *2018 IEEE/ACM 5th International Workshop on Requirements Engineering and Testing*, pages 33–40, 2018.

[33] Fabian Pedregosa, Ron Weiss, Matthieu Brucher, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard

Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2011.

[34] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1532–1543, 2014.

[35] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

[36] Muhammad Qasim Riaz, Wasi Haider Butt, and Saad Rehman. Automatic Detection of Ambiguous Software Requirements: An Insight. In *5th International Conference on Information Management, ICIM 2019*, pages 1–6. IEEE, 2019.

[37] Robert B Rowen. Software Project Management Under Incomplete and Ambiguous Specifications. *IEEE Transactions on Engineering Management*, 1990.

[38] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[39] Mike Schuster and Kuldip K. Paliwal. Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[40] Unnati S. Shah and Devesh C. Jinwala. Resolving Ambiguities in Natural Language Software Requirements. *ACM SIGSOFT Software Engineering Notes*, 2015.

[41] Zain Shaukat, Rashid Naseem, and Muhammad Zubair. A Dataset for Software Requirements Risk Prediction. In *Proceedings - 21st IEEE International Conference on Computational Science and Engineering, CSE 2018*, pages 112–118. IEEE, 2018.

[42] Daniel Sleator and Davy Temperley. Parsing English with a Link Grammar. In *Third International Workshop on Parsing Technologies*, 1993.

[43] Alan Turing. Computing Machinery and Intelligence. *MIND: A Quarterly Review of Psychology and Philosophy*, pages 433–460, 1950.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[45] Michael Waskom. Seaborn: Statistical Data Visualization. *Journal of Open Source Software*, 6(60), 2021.

[46] Ikuya Yamada, Akari Asai, Jin Sakuma, Hiroyuki Shindo, Hideaki Takeda, Yoshiyasu Takefuji, and Yuji Matsumoto. Wikipedia2Vec: An Efficient Toolkit for Learning and Visualizing the Embeddings of Words and Entities from Wikipedia. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 23–30, 2020.

[47] Hui Yang, Anne de Roeck, Vincenzo Gervasi, Alistair Willis, and Bashar Nuseibeh. Analysing Anaphoric Ambiguity in Natural Language Requirements. *Requirements Engineering*, 16(3):163–169, 2011.

[48] Hui Yang, Anne De Roeck, Alistair Willis, and Bashar Nuseibeh. A Methodology for Automatic Identification of Nocuous Ambiguity. *The 23rd International Conference on Computational Linguistics (Coling 2010), 23-27 Aug 2010, Beijing, China.*, (January), 2010.

[49] Hui Yang, Alistair Willis, Anne De Roeck, and Bashar Nuseibeh. Automatic Detection of Nocuous Coordination Ambiguities in natural language requirements. *ASE'10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 53–62, 2010.

[50] Zhi-Hua Zhou. A Brief Introduction to Weakly Supervised Learning. *National Science Review*, 5(1):44–53, 2018.