MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

FACTGUARD: UN ESCUDO INTELIGENTE CONTRA LA DESINFORMACIÓN

Autor: Sergio Cuenca Núñez

Director: Miguel Ángel Sanz Bobi

Madrid, 19 de mayo de 2025

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

FactGuard: Un Escudo Inteligente Contra la Desinformación

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas

en el curso académico 2024/25 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información

que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: *Sergio Cuenca Núñez* Fecha: 19/05/2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado por SANZ BOBI MIGUEL ANGEL - ***6599** el día 19/05/2025 con un certificado emitido por AC FNMT Usuarios

Fdo.: *Miguel Ángel Sanz Bobi* Fecha: 19/05/2025

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

FACTGUARD: UN ESCUDO INTELIGENTE CONTRA LA DESINFORMACIÓN

Autor: Sergio Cuenca Núñez

Director: Miguel Ángel Sanz Bobi

Madrid, 19 de mayo de 2025

FACTGUARD: UN ESCUDO INTELIGENTE CONTRA LA DESINFORMACIÓN

Autor: Cuenca Núñez, Sergio.
Director: Sanz Bobi, Miguel Ángel.
Entidad Colaboradora: ICAI – (Universidad Pontificia Comillas).

RESUMEN DEL PROYECTO

El proyecto está enfocado en el desarrollo de **FactGuard**, una aplicación centrada en la identificación de *fake news* y la verificación de afirmaciones mediante la consulta de fuentes contrastadas. La finalidad no es sólo detectar si una noticia se trata de un bulo, sino proporcionar explicaciones y referencias que ayuden a contextualizar la información.

Los resultados muestran una gran precisión alcanzada por los modelos de detección de noticias falsas y una correcta integración de la API de *Google FactCheck Claim Search* para la verificación de declaraciones. Todo ello enmarcado en una aplicación que cuenta con una interfaz clara, coherente y accesible, y un servicio de base de datos encargado de almacenar la información de los usuarios.

Palabras clave: *Desinformación*, *Detección de Fake News*, *Fact-checking*, *Sistema Dual de Análisis, Predicción Multiclase, Umbral de Confianza, Interfaz de Usuario Accesible*

1. Introducción

La difusión de información falsa se está convirtiendo en un problema cada vez mayor en la sociedad digital. Debido al drástico aumento de la información disponible en línea, a los usuarios les resulta cada vez más difícil distinguir entre información veraz y falsa. La toma de decisiones políticas, económicas y de salud pública se ve significativamente afectada por este fenómeno, influyendo en la percepción del público general sobre eventos importantes.

2. Definición del Trabajo

Ello justifica la necesidad de una herramienta que no solo identifique con precisión las noticias falsas, sino que también incorpore métodos de verificación fiables y proporcione una explicación clara de los resultados.

Ante esta problemática y con el propósito de abordar la lucha contra la desinformación, la propuesta del proyecto se centra en la creación de **FactGuard**, una aplicación que integra enfoques de ML (*Machine Learning*), DL (*Deep Learning*) y NLP (*Natural Language Processing*).

3. Descripción del Sistema

FactGuard está formado por un *backend* desarrollado en *Python* para la identificación de *fake news* y la verificación de declaraciones con la API de *Google FactCheck Claim Search*. La gestión de la aplicación, incluyendo la autenticación y la interfaz de usuario (*frontend*), se ha construido con `Node.js` y `Chakra UI`.

La aplicación se divide en dos grandes funcionalidades principales:

- **FactGuard Detect**: Módulo de detección de noticias falsas, desarrollado mediante algoritmos de ML y DL. Se han implementado modelos tradicionales como Árboles de Decisión, *Random Forest* y *XGBoost*, así como modelos avanzados, como BERT y arquitecturas neuronales LSTM que permiten capturar mejor los patrones contextuales y relaciones semánticas.

- **FactGuard Verify**: Herramienta de verificación de declaraciones, para contrastar información mediante la API de *Google FactCheck Claim Search*. Este módulo permite a los usuarios consultar si una afirmación ha sido verificada con anterioridad por fuentes confiables, proporcionando contexto y enlaces a las verificaciones realizadas por organismos especializados en *fact-checking*.

Además de proporcionar explicaciones contextualizadas y de la mera detección, el objetivo final es fomentar el pensamiento crítico y ayudar a los usuarios a tomar decisiones informadas sobre las fuentes en las que confían.

4. Resultados

Los resultados del módulo de FactGuard Detect, mostrados en la Tabla 1, muestran los valores de *accuracy* junto con el tiempo aproximado de entrenamiento de cada modelo empleado.

Modelo	*Accuracy*	Tiempo de Entrenamiento
CART	0.94	~46 min
Random Forest	0.97	~13 min
XGBoost	0.98	~28 min
LSTM	0.97	~12 min
BERT	0.99	~188 min

Tabla 1. Comparación de Modelos Empleados en FactGuard Detect

Los modelos BERT y *XGBoost* se sitúan a la cabeza, mostrando un rendimiento sobresaliente. LSTM también ofrece resultados muy sólidos, siendo el modelo con menor tiempo de entrenamiento entre los de aprendizaje profundo. *Random Forest* combina estabilidad y rapidez, logrando métricas competitivas.

A pesar de su sencillez, el árbol de decisión CART presentó el segundo mayor tiempo de entrenamiento y obtuvo el peor rendimiento global. Finalmente, aunque BERT alcanza una precisión cercana a la perfección, el coste computacional del modelo es muy elevado.

Por su parte, la introducción del módulo de FactGuard Verify venía dada por la necesidad de complementar el sistema de detección automática con evidencia externa. Algunos de los beneficios que consolidan su valor añadido son:

1. Acceso a evidencia verificable y contextual: El sistema no se limita a mostrar si una afirmación es verdadera o falsa, presentando enlaces a fuentes verificadoras reconocidas, la fecha de la revisión y el medio que la realizó.

2. Priorización de resultados relevantes: Dado que la API de *Google* puede devolver varios resultados para una misma afirmación, el filtrado inteligente ordena las evidencias recuperadas por fecha, mostrando primero las más recientes.

3. Gestión eficiente de casos sin evidencia disponible: En aquellos casos en los que no se encuentra ninguna evidencia relevante, el sistema devuelve un mensaje claro explicando la situación.

La UI ha sido diseñada para facilitar al usuario las funcionalidades del sistema, de forma que la interacción sea fluida, clara y comprensible, intentándose adaptar a la experiencia y necesidades de distintos perfiles. Desde las primeras etapas del desarrollo se priorizó la simplicidad en la navegación, la coherencia visual y la claridad en la presentación de los distintos módulos ofrecidos. La Figura 1 muestra el panel de control de la aplicación.

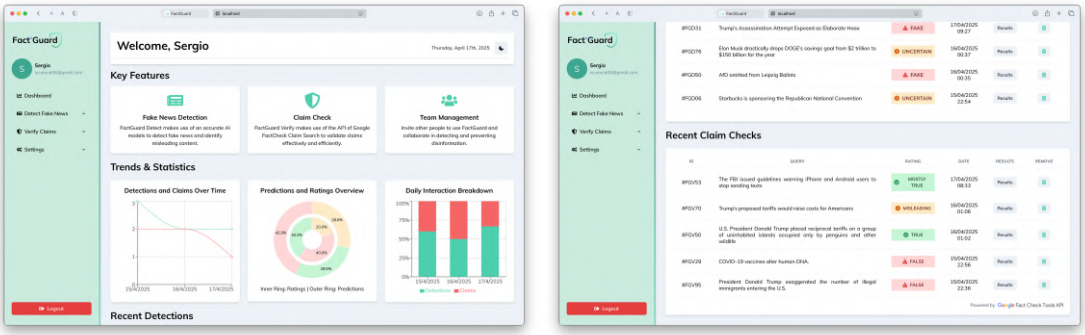


Figura 1. Interfaz del Panel de Usuario de FactGuard

5. Conclusiones

El desarrollo de FactGuard ha permitido abordar el fenómeno de la desinformación desde dos perspectivas: la detección automatizada de *fake news*, y la verificación de afirmaciones mediante fuentes externas contrastadas. El sistema ha cumplido con los objetivos que se planteaban, ofreciendo una herramienta robusta y accesible.

No obstante, FactGuard no solo representa una plataforma al servicio de la lucha contra la desinformación, sino que también cuenta con una base sólida sobre la que construir nuevas funcionalidades más completas y adaptadas al contexto cambiante del consumo informativo digital.

FACTGUARD: A SMART SHIELD AGAINST DISINFORMATION

Author: Cuenca Núñez, Sergio.
Supervisor: Sanz Bobi, Miguel Ángel.
Collaborating Entity: ICAI – (Comillas Pontifical University).

ABSTRACT

The project focuses on the development of **FactGuard**, an application centered on the identification of fake news and the verification of claims by consulting contrasting sources. The aim is not only to detect whether a news item is a hoax, but also to provide explanations and references to help contextualize the information.

The results show a high accuracy achieved by the fake news detection models and a seamless integration of the Google FactCheck Claim Search API for the verification of statements. All this is framed in an application that has a clear, coherent, and accessible interface, and a database service responsible for storing user information.

Keywords: *Disinformation*, *Fake News Detection*, *Fact-checking*, *Dual Analysis System*, *Multiclass Prediction*, *Confidence Threshold*, *Accessible User Interface*

1. Introduction

The spread of false information is becoming an increasing problem in the digital society. Due to the dramatic increase in the amount of information available online, users are finding it increasingly difficult to distinguish between truthful and false information. Political, economic, and public health decision-making is significantly affected by this phenomenon, influencing the general public's perception of important events.

2. Project Definition

This justifies the need for a tool that not only accurately identifies fake news but also incorporates reliable verification methods and provides a clear explanation of the results.

Faced with this problem and with the purpose of addressing the fight against disinformation, the project proposal focuses on the creation of **FactGuard**, an application that integrates ML (Machine Learning), DL (Deep Learning), and NLP (Natural Language Processing) approaches.

3. System description

FactGuard consists of a backend developed in Python for fake news identification and statement verification with the Google FactCheck Claim Search API. The management of the application, including authentication and frontend, has been built with `Node.js` and `Chakra UI`.

The application is divided into two main functionalities:

- **FactGuard Detect**: Fake news detection module, developed using ML and DL algorithms. Traditional models such as Decision Trees, Random Forest, and XGBoost have been implemented, as well as advanced models such as BERT and LSTM neural architectures that allow for better capture of contextual patterns and semantic relationships.
- **FactGuard Verify**: A statement verification tool to verify information using the Google FactCheck Claim Search API. This module allows users to query whether a claim has been previously verified by trusted sources, providing context and links to verifications performed by specialized fact-checking agencies.

In addition to providing contextualized explanations and mere detection, the ultimate goal is to encourage critical thinking and help users make informed decisions about the sources they trust.

4. Results

The results of the FactGuard Detect module, shown in Table 1, represent the accuracy together with the approximate training time for each model used.

Model	Accuracy	Training Time
CART	0.94	~46 min
Random Forest	0.97	~13 min
XGBoost	0.98	~28 min
LSTM	0.97	~12 min
BERT	0.99	~188 min

Table 1. Comparison of Models Used in FactGuard Detect

The BERT and XGBoost models come out on top, showing outstanding performance. LSTM also offers solid results, being the model with the shortest training time among the DL models. Random Forest combines stability and speed, achieving competitive metrics.

Despite its simplicity, the CART decision tree presented the second-longest training time and obtained the worst overall performance. Finally, although BERT achieves near-perfect accuracy, the computational cost of the model is very high.

The introduction of the FactGuard Verify module was prompted by the need to complement the automatic detection system with external evidence. Some of the benefits that consolidate its added value are:

1. Access to verifiable and contextual evidence: The system is not limited to showing whether a statement is true or false, presenting links to recognized verification sources, the date of the review, and the media that carried it out.

2. Prioritization of relevant results: Since the Google API can return multiple results for the same claim, intelligent filtering sorts the retrieved evidence by date, displaying the most recent first.

3. Efficient handling of cases with no available evidence: In those cases where no relevant evidence is found, the system returns a clear message explaining the situation.

The UI has been designed to provide the user with the system's functionalities so that the interaction is fluid, clear, and understandable. It tries to adapt to the experience and needs of different profiles. From the early stages of development, priority was given to simplicity in navigation, visual coherence, and clarity in the presentation of the different modules offered. Figure 1 shows the app's main dashboard.

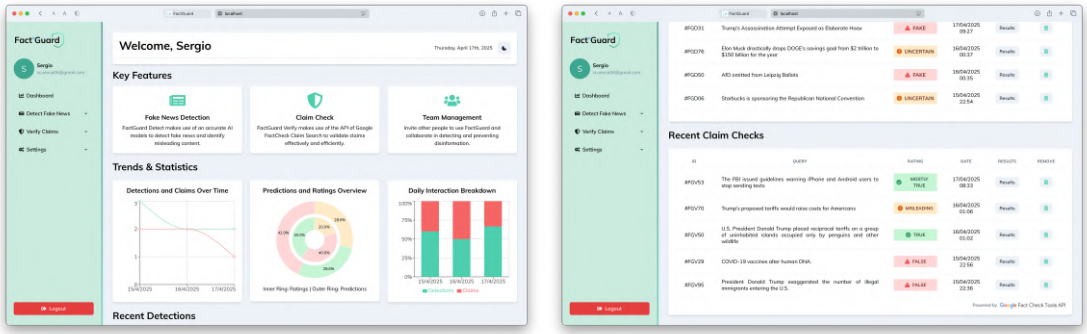


Figura 1. User Interface of FactGuard's Dashboard

5. Conclusions

The development of FactGuard has made it possible to address the phenomenon of disinformation from two perspectives: the automated detection of fake news and the verification of claims through verified external sources. The system has met its objectives, offering a robust and accessible tool.

However, FactGuard not only represents a platform in service of the fight against disinformation but also has a solid base on which to build new, more complete functionalities adapted to the changing context of digital information consumption.

Índice de la Memoria

Índice de Figuras

Índice de Tablas

Índice de Listados

Lista de Acrónimos

MIT: *Massachusetts Institute of Technology*

ML: *Machine Learning*

DL: *Deep Learning*

NLP: *Natural Language Processing*

SVM: *Support Vector Machines*

GPU: *Graphics Processing Unit*

TPU: *Tensor Processing Unit*

RNN: *Recurrent Neural Networks*

CNN: *Convolutional Neural Networks*

BERT: *Bidirectional Encoder Representations from Transformers*

RoBERTa: *Robustly Optimized BERT Pre-training Approach*

GPT: *Generative Pre-trained Transformer*

ID3: *Iterative Dichotomiser 3*

CART: *Classification and Regression Trees*

GBDT: *Gradient Boosting Decision Trees*

LSTM: *Long-Short Term Memory*

MLM: *Masked Language Model*

NSP: *Next Sentence Prediction*

NLTK: *Natural Language Toolkit*

TF-IDF: *Term Frequency-Inverse Document Frequency*

TF: *Term Frequency*

IDF: *Inverse Document Frequency*

API: *Application Programming Interface*

WSGI: *Web Server Gateway Interface*

I/O: *Input/Output*

JWT: *JSON Web Token*

MVCC: *Multiversion Concurrency Control*

ACID: Atomicidad, Consistencia, Aislamiento y Durabilidad

DOM: *Document Object Model*

JSX: *JavaScript XML*

3HAN: *Three-level Hierarchical Attention Network*

GRU: *Gated Recurrent Units*

1D-CNN: *One-dimensional CNN*

MDF: *Dynamic Fusion Model*

UEM: *Uncertainty Estimation Module*

GAT: *Graph Attention Network*

DFN: *Dynamic Fusion Network*

RTE: *Recognizing Textual Entailment*

LLM: *Large Language Model*

NLI: *Natural Language Inference*

EDA: *Exploratory Data Analysis*

UI: *User Interface*

NMT: *Neural Machine Translation*

ODS: Objetivos de Desarrollo Sostenible

ONU: Organización de las Naciones Unidas

Capítulo 1. INTRODUCCIÓN

Actualmente, con el auge de Internet y la gran popularidad de las redes sociales la información se propaga a una velocidad sin precedentes. La creciente innovación digital que se ha experimentado en la última década ha permitido a la sociedad estar en contacto permanente con datos y noticias en tiempo real. Ello ha cambiado significativamente la forma en la que las personas deciden informarse, abriendo la puerta a las denominadas *fake news*. Este término, que en español podría ser traducido como noticias falsas, está causando una distorsión en la percepción de la realidad, el debilitamiento de la confianza en las instituciones y la polarización de la opinión pública.

Estas noticias, que contienen información errónea, inventada o inexacta, afectan a múltiples círculos y dimensiones, desde la política a la economía, pasando por la salud y la cultura. En el contexto político, por ejemplo, pueden llegar a interferir en el proceso electoral, influyendo en el sentido de voto (Cugler et al., 2024). Este suceso también se observa en el ámbito de la sanidad, donde la difusión de información errónea puede perjudicar el bienestar público. Ello se evidenció durante la pandemia de COVID-19, cuando se propagaron ciertas creencias e informaciones, como el uso de ciertos medicamentos, alimentos o sustancias para prevenir la infección o la atribución del virus a teorías conspirativas (Rocha et al., 2023).

La facilidad con la que este tipo de noticias se propagan tiene que ver, en gran parte, con el algoritmo utilizado en buscadores y redes sociales, donde se premia el hecho de que un contenido sea viral en lugar de comprobar su veracidad. Ello se puede corroborar con la Figura 1, donde más de la mitad de los encuestados mencionan que las redes sociales constituyen el medio en el que se encuentran la mayor parte de las noticias falsas. Asimismo, estudios llevados a cabo por el MIT (*Massachusetts Institute of Technology*) en la red social X, conocida anteriormente como *Twitter*, señalan que las noticias en las que abunda la

desinformación se extendían 6 veces más rápido que aquellas que tenían contenido verídico (Dizikes, 2018). En este sentido, esta red social ya está implementando mecanismos para prevenir la proliferación de las *fake news*, con funcionalidades como las llamadas "Notas de la Comunidad" (X, 2024). Se tratan de aclaraciones en *tweets* publicados que pretenden agregar contexto de manera colaborativa a publicaciones potencialmente engañosas.

Figura 1. Principales Fuentes de Fake News en España

Fuente: (40dB, 2024)

El presente trabajo busca abordar dicha problemática desarrollando una herramienta efectiva para detectar y prevenir la propagación de noticias falsas. Sin embargo, la detección general de patrones y características comunes en las *fake news* no siempre es suficiente. Es en este punto donde las verificaciones de hechos (*fact-checking*) desempeñan un papel esencial, no solo comprobando si una noticia es auténtica o engañosa, sino también verificando la exactitud de declaraciones concretas, proporcionando contexto y evidencia que respalden la valoración. Esto hace que el sistema sea crucial para luchar contra la desinformación en todos los terrenos.

De esta manera, el proyecto propone fusionar estas dos perspectivas complementarias en un solo uso realizando una aplicación cuyo objeto no solo sea ayudar a reducir la propagación de noticias falsas en diversas áreas, sino que también ofrezca una herramienta valiosa para el análisis crítico de la información.

1.1 Motivación del Proyecto

Unos de los problemas más importantes a los que se enfrenta la sociedad actual en la era de la información es la proliferación de las noticias falsas. Aunque muchas organizaciones han desarrollado herramientas para la identificación de *fake news*, estas presentan limitaciones significativas en su capacidad de detección y verificación. En la mayoría de los casos, las soluciones existentes se enfocan en temas específicos, y no tienen la flexibilidad necesaria para tratar la extensa variedad de contenidos que se difunden a través de Internet. Asimismo, sólo proporcionan resultados binarios ofreciendo explicaciones poco exhaustivas y sin ningún marco o contexto (A.B. et al., 2023).

Según la Figura 2, un porcentaje elevado de los adultos en diferentes países expresaron su preocupación por la veracidad de las noticias en Internet, con Brasil (84,0%), Sudáfrica (71,6%) y Estados Unidos (67,5%) encabezando la lista. Por su parte, en España el 65,1% de los encuestados manifestaron su inquietud por este fenómeno, lo que evidencia la necesidad de desarrollar y encontrar soluciones efectivas para contrarrestar la proliferación de las *fake news*.

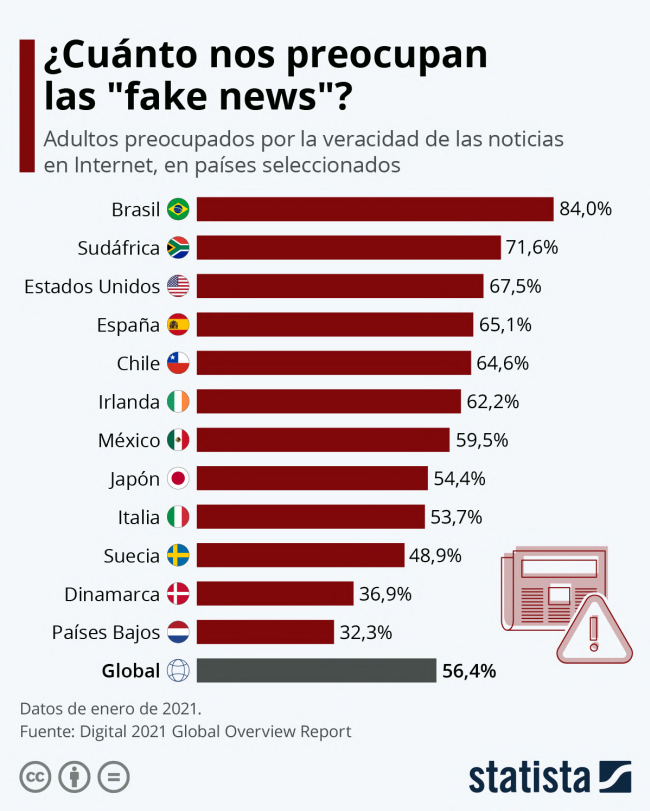


Figura 2. Preocupación Global por las Fake News en Distintos Países

Fuente: (We Are Social & Hootsuite, 2021)

A partir de esta problemática, la motivación principal del presente trabajo es desarrollar una aplicación centrada en la identificación y verificación de información engañosa. La finalidad no es sólo detectar si una noticia se trata de un bulo, sino que también pretende proporcionar explicaciones y referencias que ayuden a contextualizar la información.

La innovación de esta plataforma radica en su capacidad para analizar noticias de diversas temáticas y proporcionar un mecanismo de verificación de afirmaciones. Este sistema de *fact-checking* permite comprobar la veracidad de declaraciones recientes que pueden no estar registradas en otras bases de datos estáticas. Ello no sólo beneficia a usuarios que desean verificar la autenticidad de la información en redes sociales y otros medios, sino que también resulta útil para periodistas, educadores y analistas que requieren herramientas

fiables para combatir la desinformación. Además de proporcionar explicaciones contextualizadas y de la mera detección, el objetivo final es fomentar el pensamiento crítico y ayudar a los usuarios a tomar decisiones informadas sobre las fuentes en las que confían.

Capítulo 2. DEFINICIÓN DEL TRABAJO

2.1 JUSTIFICACIÓN

La difusión de información falsa se está convirtiendo en un problema cada vez mayor en la sociedad digital. Debido al drástico aumento de la información disponible en línea, a los usuarios les resulta cada vez más difícil distinguir entre información veraz y falsa. Como se ha mencionado con anterioridad, la toma de decisiones políticas, económicas y de salud pública se ve significativamente afectada por este fenómeno, influyendo en la percepción del público general sobre eventos importantes.

Ello justifica la necesidad de una herramienta que no solo identifique con precisión las noticias falsas, sino que también incorpore métodos de verificación fiables y proporcione una explicación clara de los resultados. Ante esta problemática y para abordar el creciente problema de la desinformación, la propuesta del trabajo se centra en la creación de **FactGuard**, una aplicación que integra enfoques de ML (*Machine Learning*), DL (*Deep Learning*) y NLP (*Natural Language Processing*).

2.2 OBJETIVOS

El objetivo principal se encuentra en la creación de la aplicación **FactGuard**, formada por un *backend* desarrollado en *Python* para la identificación de *fake news* y la verificación de declaraciones con la API de *Google FactCheck Claim Search*. Se ha decidido optar por este lenguaje dado que proporciona un amplio abanico de librerías y marcos de trabajo útiles para el desarrollo de algoritmos de detección y NLP. Por su parte, la gestión de la aplicación, incluyendo la autenticación y la interfaz de usuario, se construirá con `Node.js` y `Chakra UI`.

La aplicación se divide en dos grandes funcionalidades principales:

- **FactGuard Detect**: Módulo de detección de noticias falsas, desarrollado mediante algoritmos de ML y DL. Se han implementado modelos tradicionales como Árboles de Decisión, *Random Forest* y *XGBoost*, así como modelos avanzados, como BERT y arquitecturas neuronales LSTM que permiten capturar mejor los patrones contextuales y relaciones semánticas. Ello se realiza con el fin de realizar una comparación de la precisión otorgada por los diferentes modelos.

- **FactGuard Verify**: Herramienta de verificación de declaraciones, para contrastar información mediante la API de *Google FactCheck Claim Search*. Este módulo permite a los usuarios consultar si una afirmación ha sido verificada con anterioridad por fuentes confiables, proporcionando contexto y enlaces a las verificaciones realizadas por organismos especializados en *fact-checking* (Google Fact Check Tools API, 2023).

2.3 METODOLOGÍA

Para alcanzar el objetivo general del proyecto, se ha establecido la siguiente metodología que cumple con los siguientes objetivos secundarios:

1. Desarrollo del módulo **FactGuard Detect**, para identificar de manera automática noticias falsas.

    - Implementar y entrenar los modelos mencionados haciendo uso del conjunto de datos *ISOT Fake News Dataset* (Ahmed et al., 2017).

    - Perfeccionar los modelos finales para conseguir una precisión adecuada en cada uno de ellos, y que su capacidad de generalización sea alta para la detección de noticias falsas de diversas temáticas.

2. Incorporación del segundo módulo, **FactGuard Verify**, para la comprobación detallada de afirmaciones y declaraciones.

   - Implementar la API de *Google FactCheck Claim Search* en la plataforma para obtener las verificaciones de hechos realizadas por fuentes fiables.

   - Desarrollar funcionalidades que permitan al usuario introducir declaraciones o noticias y obtener los resultados pertinentes.

3. Integración de **FactGuard Detect** y **FactGuard Verify** en la aplicación final.

   - Crear una interfaz de usuario dinámica e intuitiva con la que el usuario tenga acceso a ambas funciones.

4. Comprobar el rendimiento general de la aplicación **FactGuard** en su conjunto.

   - Realizar pruebas rigurosas para evaluar la exactitud, rapidez y utilidad de la aplicación en contextos reales.

Para lograr estos objetivos, FactGuard debe incorporar un conjunto de herramientas que serán detalladas en los siguientes apartados. No obstante, antes de profundizar en las tecnologías específicas, es importante presentar una visión general del enfoque adoptado donde se la estructura de la aplicación y el flujo de información a través de los distintos componentes que la forman.

En la Figura 24, se introduce un diagrama de arquitectura general que ilustra la relación entre el módulo de detección desarrollado en *Python*, el *backend* de `Node.js` donde se encuentra la base de datos y el *frontend*. Por su parte, en la Figura 25 se muestra una arquitectura análoga para el módulo de verificación, en la que se detalla la conexión de la aplicación con la API externa de *fact-checking*.

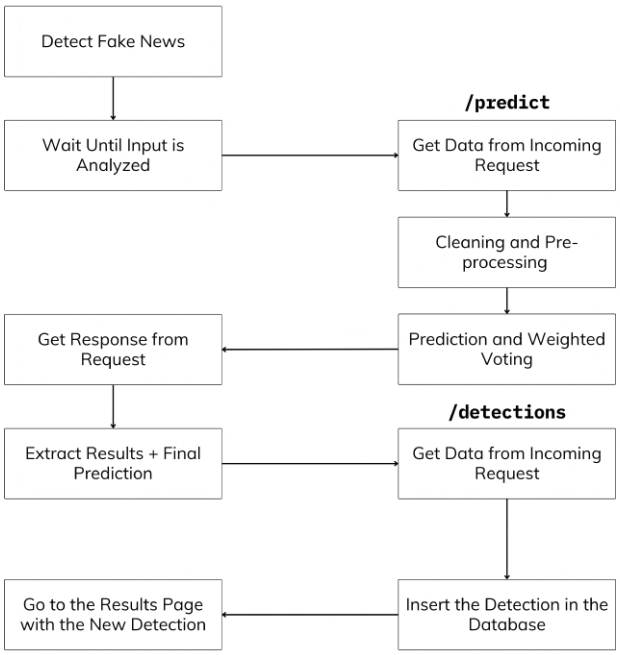


Figura 3. Flujo de Ejecución General de FactGuard Detect

Fuente: Elaboración Propia

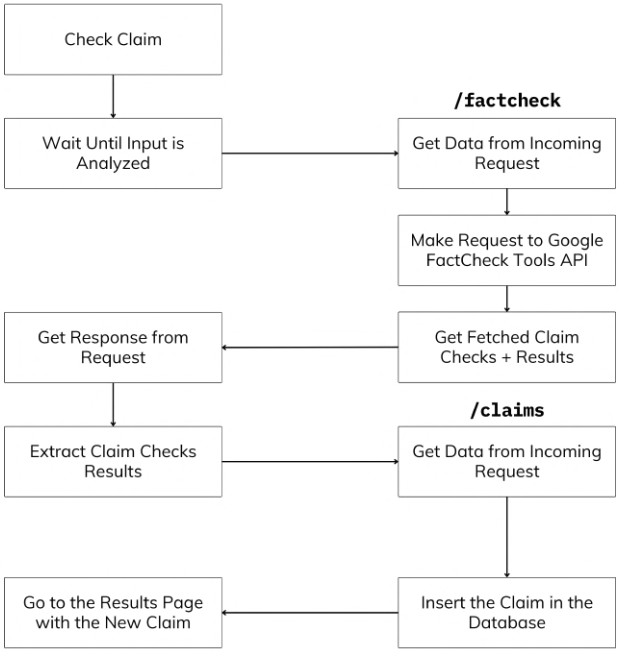


Figura 4. Flujo de Ejecución General de FactGuard Verify

Fuente: Elaboración Propia

2.4 PLANIFICACIÓN

Respecto a la gestión del proyecto, se implementará el marco de trabajo SCRUM, adaptado para maximizar la eficiencia en el contexto de este desarrollo. Se realizarán reuniones mensuales para realizar los controles y *checkpoints* necesarios, establecer nuevas responsabilidades, confirmar aquellas que se han finalizado y corregir los errores señalados.

El cronograma estimado se muestra en el Diagrama de *Gantt* de la Figura 5.

	Octubre				Noviembre				Diciembre				Enero				Febrero				Marzo				Abril			
Análisis y Preparación																												
Exploración del Conjunto de Datos y Herramientas a Utilizar																												
Revisión Bibliográfica y Marco Conceptual																												
Desarrollo, Entrenamiento y Validación de los Modelos de Detección																												
Integración de la API de Google FactCheck																												
Construcción y Despliegue de la Aplicación Final																												
Pruebas y Validación																												
Redacción del TFM																												

Figura 5. Diagrama de Gantt

Fuente: Elaboración Propia

Capítulo 3. DESCRIPCIÓN DE LAS TECNOLOGÍAS

En este capítulo se presentan las principales tecnologías y *frameworks* utilizados en el desarrollo de FactGuard. Se abordarán tanto las herramientas de utilizadas para la detección de *fake news* y *fact-checking*, así como las librerías para el procesamiento de datos y lenguaje natural, y las tecnologías utilizadas en el desarrollo del *frontend*/*backend* de la aplicación.

3.1 DETECCIÓN DE FAKE NEWS Y FACT-CHECKING

3.1.1 HERRAMIENTAS PARA APRENDIZAJE AUTOMÁTICO Y PROFUNDO

3.1.1.1 Scikit-Learn

Desarrollada inicialmente por David Cournapeau como parte del proyecto *Google Summer of Code* en 2007 y ampliada más tarde por un equipo de desarrolladores, `scikit-learn` es una librería de código abierto en *Python* que proporciona herramientas para el análisis de datos y la modelización predictiva (Pedregosa et al., 2011). Se trata de una de las bibliotecas más utilizadas para tareas relacionadas con el aprendizaje supervisado y no supervisado, incluyendo clasificación, regresión y *clustering*. Destaca por su sencillez, facilidad de uso y compatibilidad con otras librerías como `NumPy` y `SciPy`.

Incluye implementaciones de varios algoritmos de ML, tales como Máquinas de Soporte Vectorial o SVM (*Support Vector Machines*), Árboles de Decisión, *Random Forest*, *Gradient Boosting* (como XGBoost), *K-Means* y DBSCAN. Asimismo, también incluye funciones para la estandarización, normalización y transformación de datos, lo que facilita la preparación de los diferentes conjuntos de datos antes del entrenamiento de los modelos. Por último, proporciona métricas y técnicas de validación cruzada para evaluar el rendimiento de los modelos desarrollados de manera robusta (Pedregosa et al., 2011).

Esta biblioteca goza de gran popularidad en la industria como en la investigación académica, siendo citada en numerosas publicaciones de carácter científico, adoptada y ampliamente utilizada en diferentes tareas como la detección de fraude, predicción de tendencias y clasificación de textos.

3.1.1.2 Tensorflow

`Tensorflow` es un *framework* de código abierto creado por *Google Brain*, lanzado en 2015 como una evolución de su antecesor *DistBelief*. Posee una arquitectura flexible y escalable lo que ha convertido a esta herramienta en una de las utilizadas en la implementación de modelos de DL, facilitando el desarrollo de redes neuronales artificiales en contextos de investigación y en aplicaciones comerciales (Abadi et al., 2016).

Este marco de trabajo está concebido para optimizar el cálculo en paralelo y repartir las cargas de trabajo en varios procesadores y aceleradores, como GPUs (*Graphics Processing Units*) y TPUs (*Tensor Processing Units*). Incorpora un sistema de grafos computacionales que permite definir modelos de manera modular, haciendo que las operaciones matemáticas complejas que se requieren en el entrenamiento de redes neuronales se ejecuten de una forma más eficiente (Abadi et al., 2016).

Por otro lado, `tensorflow` incorpora otras herramientas como `tensorboard`, que ofrece visualizaciones interactivas para depuración y evaluación del desempeño de modelos, y `tensorflowLite`, que permite la implementación de modelos optimizados para dispositivos móviles y otros sistemas. Gracias a su versatilidad, se ha empleado en tareas de NLP, visión computacional y reconocimiento de patrones en grandes volúmenes de datos.

3.1.1.3 Transformers

En el ámbito de NLP, los *Transformers* representan una de las innovaciones más destacadas puesto que han revolucionado la forma en la que los modelos adquieren representaciones contextuales del lenguaje. Introducidos en 2017 por Vaswani et al., estos

modelos sustituyen las arquitecturas de redes neuronales como las RNN (*Recurrent Neural Networks*) y CNN (*Convolutional Neural Networks*) utilizadas en tareas de NLP, al introducir una estructura basada en mecanismos de autoatención. La Figura 6 ilustra este mecanismo, que permite capturar dependencias a largo plazo en los textos, mejorando significativamente el modelado del contexto lingüístico (Vaswani et al., 2017).

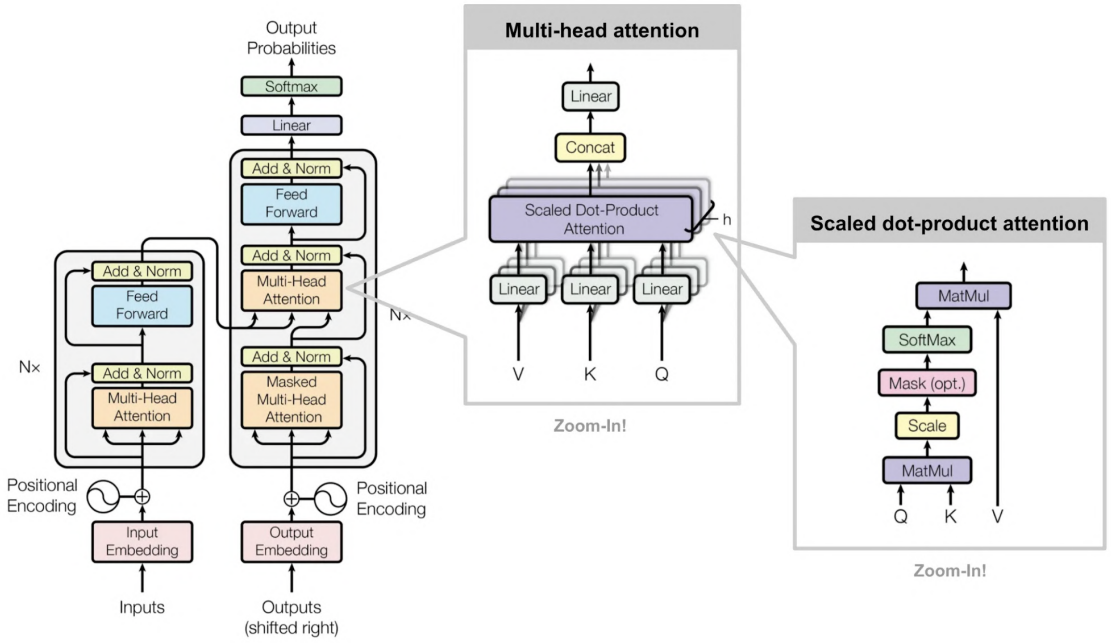


Figura 6. Arquitectura del Transformer y Mecanismo de Autoatención

Fuente: (Vaswani et al., 2017)

La evolución y adopción que han experimentado estos modelos han sido impulsadas por la comunidad de código abierto, destacando la biblioteca `transformers` de *Hugging Face*. Esta ofrece implementaciones optimizadas de modelos ya entrenados como BERT (*Bidirectional Encoder Representations from Transformers*), RoBERTa (*Robustly Optimized BERT Pretraining Approach*) y GPT (*Generative Pre-trained Transformer*). Se ha comprobado que poseen un desempeño destacado en tareas de clasificación de texto, respuesta a preguntas, análisis de sentimientos y generación de lenguaje natural, alcanzando resultados similares al rendimiento humano en varios *benchmarks* (Vaswani et al., 2017).

Esta habilidad para capturar vínculos semánticos y sintácticos en grandes cantidades de información ha facilitado su uso en otros sistemas como la traducción automática, asistentes virtuales y herramientas para de verificación de declaraciones.

3.1.2 ALGORITMOS PARA CLASIFICACIÓN SUPERVISADA

3.1.2.1 Decision Tree Classifier

Un árbol de decisión (*Decision Tree*) es un modelo de aprendizaje supervisado utilizado principalmente para tareas de clasificación y regresión. Fue introducido por J.R. Quinlan en 1986 con la creación del algoritmo ID3 (*Iterative Dichotomiser 3*), que evolucionó posteriormente a versiones más sofisticadas y avanzadas como C4.5 y CART (*Classification and Regression Trees*).

Se trata de un modelo que representa un conjunto de decisiones en una estructura jerárquica. Tal y como se muestra en la Figura 7, cada nodo interno simboliza una pregunta basada en una característica del conjunto de datos, las ramas indican las posibles respuestas, y las hojas representan la predicción final. La elección de características para la división se lleva a cabo a través de métricas como la ganancia de información (*Information Gain*) en el caso del criterio de entropía, o la reducción del coeficiente de Gini (Quinlan, 1986).

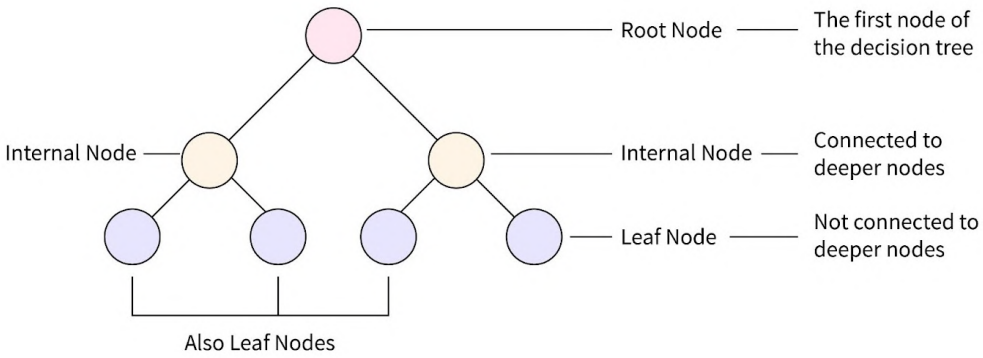


Figura 7. Estructura de un Árbol de Decisión

Fuente: (Singh, 2024)

La entropía evalúa la pureza de un nodo en términos de clasificación. Un nodo puro tiene entropía cero, lo que significa que todos los datos en ese nodo pertenecen a la misma clase (Quinlan, 1986). Esto se puede expresar como:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2 p_i$$

donde:

- $H(S)$ representa la entropía del conjunto de datos S.
- c es el número total de clases posibles.
- p_i es la proporción de elementos de la clase i dentro del conjunto de datos.

La ganancia de información mide la reducción de la entropía obtenida al dividir un nodo en función de una característica A, y se expresa como:

$$IG(S, A) = H(S) - \sum_{v \in V} \frac{|S_v|}{|S|} H(S_v)$$

donde:

- $IG(S, A)$ es la ganancia de información de la característica A.
- V es el conjunto de valores posibles de A.
- S_v representa el subconjunto de S donde la característica A toma el valor v.

Por otro lado, el índice de Gini mide la posibilidad de que una muestra sea categorizada de manera equivocada si se selecciona de manera aleatoria según la distribución de la etiqueta en el nodo (Quinlan, 1986). Se define como:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

donde p_i es la proporción de la clase i en el nodo actual. Cuanto menor sea el valor del índice de Gini, mayor será la pureza del nodo.

Este algoritmo destaca por su interpretabilidad, pues facilita la visualización de las reglas de clasificación aprendidas y por su fácil implementación. Además, el coste computacional de este modelo es muy bajo en comparación con otros modelos más complejos. No obstante, pueden llevar al sobreajuste (*overfitting*) en caso de que se desarrollen árboles profundos con muchos nodos. Para ello, se deben implementar mecanismos de poda (*pruning*), con el objeto de evitar la sensibilidad a datos ruidosos.

3.1.2.2 Random Forest

Random Forest es un algoritmo de agrupación (*ensemble*) basado en múltiples árboles de decisión. Fue desarrollado inicialmente por Leo Breiman en 2001, y mejora la precisión obtenida por un solo árbol a través del método de *bagging* (*Bootstrap Aggregating*). Gracias a su robustez y precisión se ha convertido en una de las herramientas más empleadas en clasificación y regresión.

El funcionamiento de este modelo está basado en el entrenamiento de varios árboles de decisión a partir de distintos subconjuntos del conjunto de datos principal y combinar sus predicciones con el objetivo de reducir la varianza y evitar el *overfitting*. Para ello, se hace uso del muestreo *bootstrap*, donde el subconjunto de datos seleccionado para entrenar cada árbol es seleccionado aleatoriamente con reemplazo, introduciendo diversidad en la construcción del modelo (Breiman, 2001).

A diferencia de un árbol de decisión tradicional, cada nodo en *Random Forest* elige el mejor atributo entre un subconjunto aleatorio de m características, lo que reduce la correlación entre los árboles y mejora la generalización (Breiman, 2001). Esta selección se lleva a cabo con la finalidad de reducir la correlación entre los árboles y mejorar su habilidad para generalizar. El número óptimo de características por nodo suele definirse como:

$$m = \sqrt{M} \text{ (para clasificación)}$$

$$m = \frac{M}{3} \text{ (para regresión)}$$

$$\text{con } m < M$$

donde:

- m es el número de características seleccionadas en cada nodo.
- M es el número total de características en el conjunto de datos.

Una vez entrenados los múltiples árboles de decisión, el modelo final realiza la predicción combinando los resultados de todos los árboles individuales. En el caso de la clasificación, las predicciones se evalúan por votación mayoritaria, tomando la clase más votada entre todos los árboles:

$$\hat{y} = \text{mode}\{T_1(x), T_2(x), \dots, T_k(x)\}$$

donde $T_k(x)$ representa la predicción del árbol k para la muestra x.

Por otro lado, en tareas de regresión se realiza el promedio de todas las predicciones obtenidas.

$$\hat{y} = \frac{1}{K}\sum_{k=1}^{K} T_k(x)$$

donde K es el número total de árboles y $T_k(x)$ es la predicción del árbol k.

Todo este proceso de entrenamiento y resultado de la predicción se observa en la Figura 8, donde se ilustra el algoritmo que sigue el modelo de *Random Forest*. Este enfoque se distingue por su robustez frente a datos ruidosos o valores atípicos y su capacidad para manejar grandes conjuntos de datos o de alta dimensionalidad. Gracias a su resistencia al

overfitting, es adecuado para actividades donde los patrones de clasificación pueden ser complejos y ruidosos (Breiman, 2001).

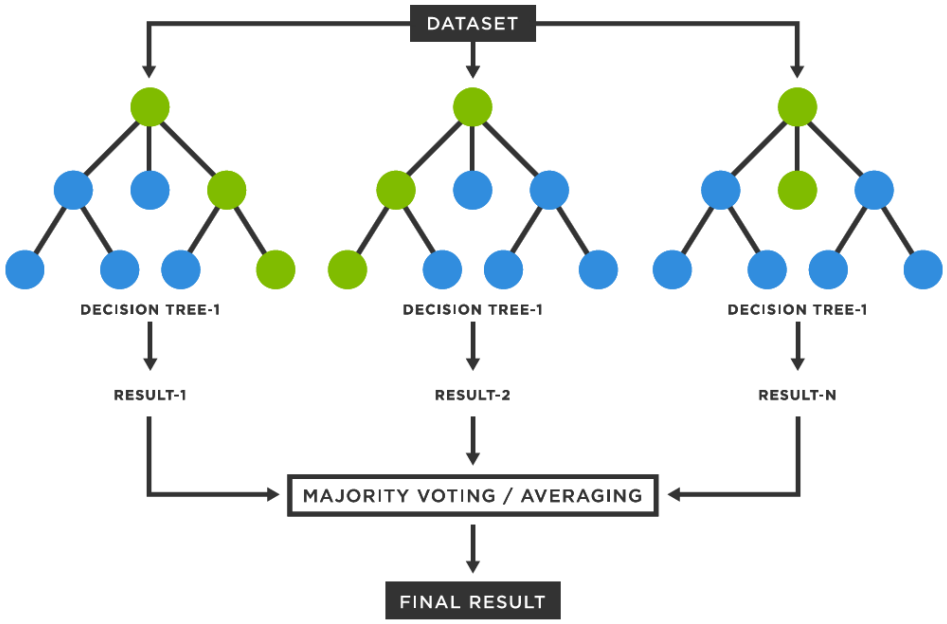


Figura 8. Algoritmo de Random Forest

Fuente: (Gunay, 2023)

Sin embargo, también posee algunas limitaciones, como un mayor consumo de memoria al tener que calcular las predicciones de múltiples árboles simultáneamente o la dificultad en la interpretación debido a la combinación de múltiples modelos (Breiman, 2001).

3.1.2.3 XGBoost

XGBoost (*Extreme Gradient Boosting*) es un algoritmo de aprendizaje supervisado basado en la técnica de *boosting*, introducido por Tianqi Chen en 2016. Se trata de una implementación optimizada del algoritmo de GBDT (*Gradient Boosting Decision Trees*), donde se mejora significativamente el rendimiento en tareas de clasificación y regresión mediante el ajuste iterativo de errores residuales (Chen & Guestrin, 2016). Se ha establecido

como uno de los instrumentos más potentes en la predicción y clasificación de datos estructurados, dada su elevada eficacia computacional y su gran capacidad de generalización.

Este algoritmo, que sigue el principio de *Gradient Boosting*, funciona mediante la construcción secuencial de árboles de decisión, donde el objetivo de cada árbol nuevo es corregir los errores cometidos por los anteriores. Ello se consigue mediante la optimización iterativa de una función de pérdida $L(y, \hat{y})$, donde y representa el valor real y $\hat{y}$ la predicción realizada (Chen & Guestrin, 2016). La Figura 9 muestra la estructura de este procedimiento, en la que se muestra la corrección gradual de los errores residuales a medida que se incorporan nuevos árboles al modelo.

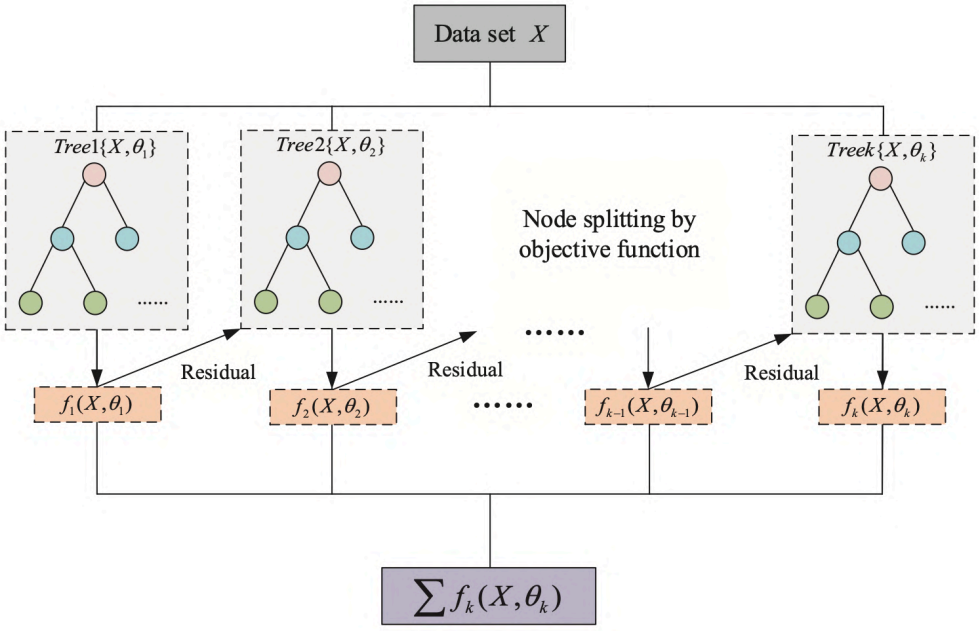


Figura 9. Funcionamiento de XGBoost con Corrección de Errores Residuales

Fuente: (Guo et al., 2020)

En cada interacción t, se construye un nuevo árbol que ajusta los residuos r_t de las predicciones anteriores mediante la derivada negativa de $L(y, \hat{y})$ (Chen & Guestrin, 2016):

$$r_t = -\frac{\partial L(y, \hat{y})}{\partial \hat{y}}$$

El modelo se actualiza de la siguiente forma:

$$F_t(x) = F_{t-1}(x) + \eta h_t(x)$$

donde:

- $F_t(x)$ es la predicción del modelo en la iteración t.
- $h_t(x)$ es el nuevo árbol ajustado a los residuos.
- η es la tasa de aprendizaje (*learning rate*), que controla la contribución del nuevo árbol.

A diferencia de otros métodos, *XGBoost* optimiza el proceso de *boosting* a través del uso de regularizaciones L1 y L2, evitando de esta manera el *overfitting* y mejorando su capacidad de generalización (Chen & Guestrin, 2016). La función de pérdida regularizada utilizada en *XGBoost* se expresa como:

$$L(\theta) = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \lambda \|w\|_1 + \alpha \|w\|_2^2$$

donde:

- λ es el parámetro de regularización L1, que controla la cantidad de características seleccionadas, eliminando aquellas con menor peso.
- α es el parámetro de regularización L2, que penaliza los pesos elevados de los parámetros para evitar que el modelo tenga *overfitting*.

En lugar de expandir los nodos de los árboles hasta que no haya mejora, este modelo reduce la contribución de cada nuevo árbol (*shrinkage*) y realiza un muestreo de columnas

(*column sampling*) para evitar el *overfitting*. Asimismo, la selección automática de características por nodo, donde el número óptimo es calculado al igual que en *Random Forest*, y la partición eficiente de datos lo hace altamente efectivo en tareas con grandes conjuntos de datos (Chen & Guestrin, 2016). Ello potencia el rendimiento frente a otros procedimientos fundamentados en árboles de decisión.

Una de las principales ventajas que otorga *XGBoost* es su mayor velocidad en comparación con métodos de *boosting*, gracias a su paralelización y optimización de memoria. Debido a que el cálculo de las divisiones de los nodos se realiza en paralelo, se consigue reducir drásticamente el tiempo de entrenamiento, en lugar de construir cada árbol de manera secuencial.

Por otro lado, requiere un ajuste fino (*fine-tuning*) de parámetros como la tasa de aprendizaje η, el número de árboles T y la profundidad de los árboles d, para evitar un posible bajo rendimiento o el *overfitting* a los datos de entrenamiento.

3.1.2.4 LSTM

Las redes neuronales de memoria a corto y largo plazo o LTSM (*Long Short-Term Memory*) fueron introducidas en 1997 por Hochreiter y Schmidhuber como una variante de las RNN. Las redes LSTM se diseñaron para tratar el problema del desvanecimiento del gradiente (*vanishing gradient*), que afectaba a las RNN tradicionales. En estas redes, los gradientes se propagan hacia atrás utilizando la regla de la cadena en la retropropagación (*backpropagation algorithm*), pero en secuencias largas tendían a volverse extremadamente pequeños, impidiendo la actualización efectiva de los pesos de la red (Hochreiter & Schmidhuber, 1997). Este problema se puede formular como:

$$\frac{\partial E}{\partial W} = \prod_{t=1}^{T} \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial E}{\partial h_T}$$

donde:

- E es la función de error de la red.
- W representa los pesos de la red.
- h_t es el estado oculto en el tiempo t.

Cuando la derivada parcial de h_t respecto a h_{t-1} es inferior a 1, la multiplicación repetida provoca que los gradientes tiendan a cero, limitando su capacidad para aprender dependencias a largo plazo (Hochreiter & Schmidhuber, 1997).

Las redes LSTM superan dicha limitación mediante un mecanismo de control basado en puertas que regulan el flujo de información dentro de cada celda de memoria. De esta manera, pueden aprender patrones en secuencias de datos largas y recordar información relevante durante períodos prolongados (Hochreiter & Schmidhuber, 1997).

Cada celda cuenta con un estado (*cell state*), que funciona como un canal de memoria que facilita el almacenamiento de la información a lo largo del tiempo, y un estado oculto (*hidden state*), que representa la salida de la celda en cada paso de tiempo. Para gestionar estos estados, se emplean tres tipos de puertas (Hochreiter & Schmidhuber, 1997).

La Puerta de Entrada (*Input Gate*) se encarga de regular la cantidad de nueva información que debe almacenarse en la celda de memoria, permitiendo actualizar su estado con información relevante. La activación de la puerta de entrada se define como:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

y la memoria candidata que se almacenará en la celda es calculada como:

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

donde:

- i_t es la activación de la puerta de entrada en el tiempo t.

- x_t es la entrada en el tiempo t.
- h_{t-1} es el estado oculto de la celda en el tiempo anterior.
- W_i, U_i, b_i son los parámetros de entrenamiento.
- σ es la función sigmoide, que produce una probabilidad entre 0 y 1, y tanh es la tangente hiperbólica.

Por su parte, la Puerta de Olvido (*Forget Gate*) controla qué información del estado de la celda deben ser eliminada, garantizando que solo se conserve aquella información que sea más importante a lo largo del tiempo. Se expresa como:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

y la actualización del estado de la celda se define como:

$$C_t = f_t C_{t-1} + i_t \tilde{C_t}$$

donde:

- f_t es la activación de la puerta de olvido.
- C_{t-1} es el estado de la celda en el tiempo anterior.
- C_t es el nuevo estado de la celda.

Por último, la Puerta de Salida (*Output Gate*) es la que establece cuánta información del estado de la celda debe ser transmitida a la capa subsiguiente de la red. Se calcula como:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

y el estado oculto de la celda se actualiza como:

$$h_t = o_t \tanh(C_t)$$

donde h_t es la salida final de la celda en el tiempo t.

Este proceso se ilustra en la Figura 10, donde se representa el flujo de información dentro de una celda LSTM. Se observa cómo las puertas de entrada (1), olvido (2) y salida (3) regulan el estado de la celda y la propagación de la información a lo largo de la secuencia. Asimismo, se muestra la interacción entre el estado de celda C_t, el estado oculto h_t y las funciones de activación sigmoide y tangente hiperbólica que modulan el flujo de datos.

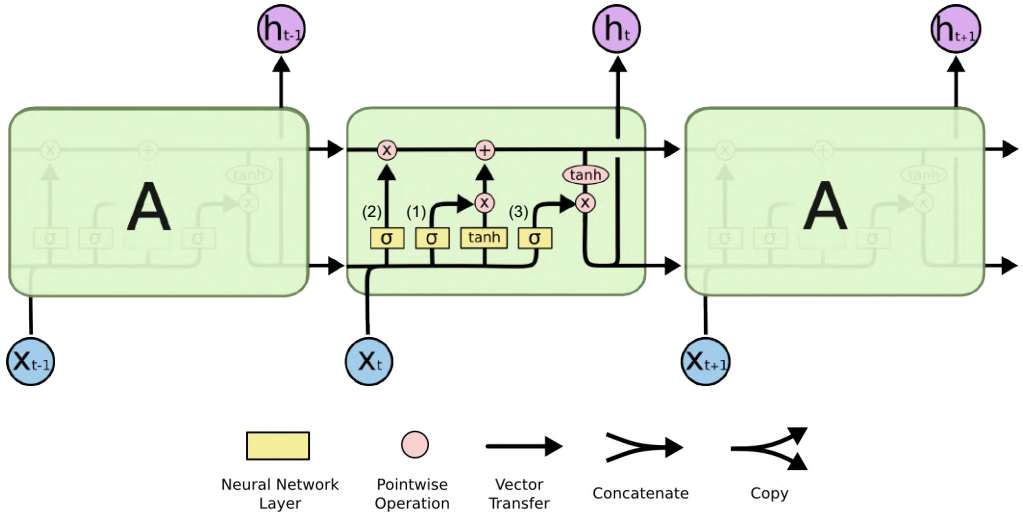


Figura 10. Estructura Interna de una Celda LSTM

Fuente: (Olah, 2015)

Este mecanismo de control permite a la red aprender que cantidad o tipo de información debe recordar y cuál debe olvidar, mejorando significativamente la capacidad de modelar relaciones de largo plazo en datos secuenciales.

Ello hace que las LSTM sean capaces de modelar relaciones temporales complejas en series temporales y procesamiento de lenguaje natural, siendo ampliamente utilizadas en estos campos. Si bien resuelven el principal problema que afecta a las RNN tradicionales, su arquitectura con múltiples puertas y cálculos adicionales introduce un mayor coste

computacional. Esto requiere tiempos de entrenamiento más largos y un ajuste de hiperparámetros más preciso para alcanzar un rendimiento óptimo.

3.1.2.5 BERT

Presentado en 2019 por Devlin et al., BERT es un modelo de aprendizaje profundo basado en la arquitectura de *Transformers*, desarrollado por Google AI. La innovación principal de este modelo radica en su capacidad para capturar relaciones bidireccionales dentro de los textos, lo que mejora significativamente el rendimiento en tareas de NLP.

A diferencia de otras arquitecturas, como las redes LSTM, o modelos tradicionales que procesaban el texto de manera unidireccional (de izquierda a derecha o de derecha a izquierda), BERT utiliza múltiples capas de autoatención (*self-attention*) para modelar las dependencias dentro de los textos. Ello le permite tener una comprensión más profunda del contexto al considerar tanto la información anterior como posterior a una palabra en una oración (Devlin et al., 2019). Esto se evidencia en la Figura 11, donde se visualiza su estructura basada en *Transformers* y cómo maneja la información mediante las capas de autoatención.

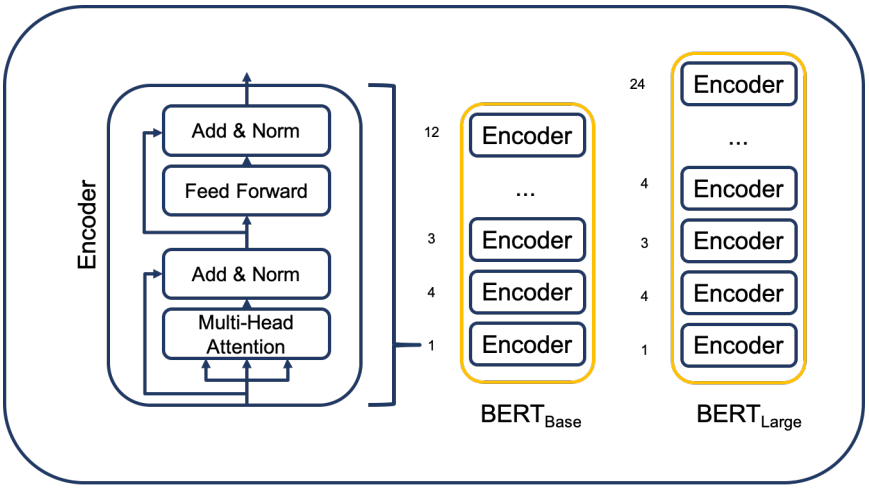


Figura 11. Arquitectura General de BERT

Fuente: (Evtimov et al., 2020)

El mecanismo de autoatención, también conocido como *Scaled Dot-Product Attention*, se define como:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

donde:

- Q (*queries*), K (*keys*) y V (*values*) son matrices de entrada que representan las palabras en la secuencia de texto.
- d_k es la dimensión de las claves, utilizada para escalar la operación y evitar gradientes inestables.
- QK^T representa la similitud entre las palabras de la oración, permitiendo que el modelo capture dependencias a larga distancia.

Para poder comprender el contexto bidireccional en una oración, BERT divide las palabras en unidades más pequeñas mediante un proceso llamado *WordPiece Tokenization*, asignando a cada subpalabra un *embedding* de 768 dimensiones. La Figura 12 ilustra este proceso, mostrando cómo las palabras de entrada se dividen en *tokens* y son representadas en un espacio de mayor dimensión.

El entrenamiento de este modelo se divide en dos tareas principales. Por un lado, MLM (*Masked Language Model*) es la parte encargada de enmascarar aleatoriamente algunas de las palabras dentro del texto de entrada y entrenar el modelo para predecirlas basándose en el resto del contexto. Ello permite a BERT aprender representaciones de palabras en función de su significado en diferentes contextos. El objetivo de la función es maximizar la probabilidad del modelo de predecir la palabra correcta entre todas las palabras del vocabulario (Devlin et al., 2019).

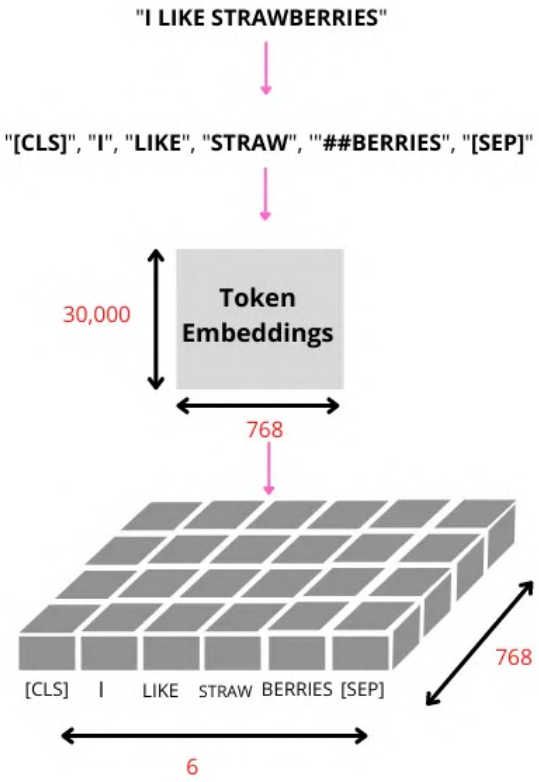


Figura 12. Tokenización y Representación de Palabras en BERT

Fuente: (Siva, 2021)

La probabilidad de una palabra enmascarada w_t se calcula mediante la función de predicción:

$$P(w_t|w_1,\dots,w_{t-1},w_{t+1},\dots,w_n) = \frac{e^{h_t^T W}}{\sum_{w' \in V} e^{h_t^T W_{w'}}}$$

donde:

- h_t es la representación contextual de la palabra enmascarada.
- W es la matriz de pesos del modelo.
- V es el vocabulario del modelo.

Este proceso se puede observar en la Figura 13, donde el modelo está ocultando el 15% de los *tokens* en la entrada y predice las palabras enmascaradas basándose en su contexto.

Por otro lado, el módulo de NSP (*Next Sentence Prediction*) recibe dos oraciones y es el encargado de predecir si la segunda oración sigue lógicamente a la primera. Este mecanismo mejora el rendimiento en tareas que requieren la comprensión de relaciones entre oraciones dentro de un mismo contexto o párrafo (Devlin et al., 2019).

Esto se puede modelar como un problema de clasificación binaria:

$$P(S_2|S_1) = \sigma(W_s[CLS] + b_s)$$

donde:

- S_1 y S_2 son las oraciones de entrada.
- $[CLS]$ es el token especial de inicio, que resume la información de la oración.
- W_s y b_s son los parámetros de la capa de clasificación.
- σ es la función sigmoide.

La Figura 13 representa esta tarea, donde BERT predice si la segunda oración sigue lógicamente a la primera, utilizando el token especial $[CLS]$ para la clasificación.

Una de las principales ventajas de este modelo es el que el entrenamiento se ha realizado con texto sin etiquetar, lo que permite realizar aprendizaje por transferencia (*transfer learning*) a otros modelos o tareas de NLP, como clasificación de texto, traducción automática y respuesta a preguntas.

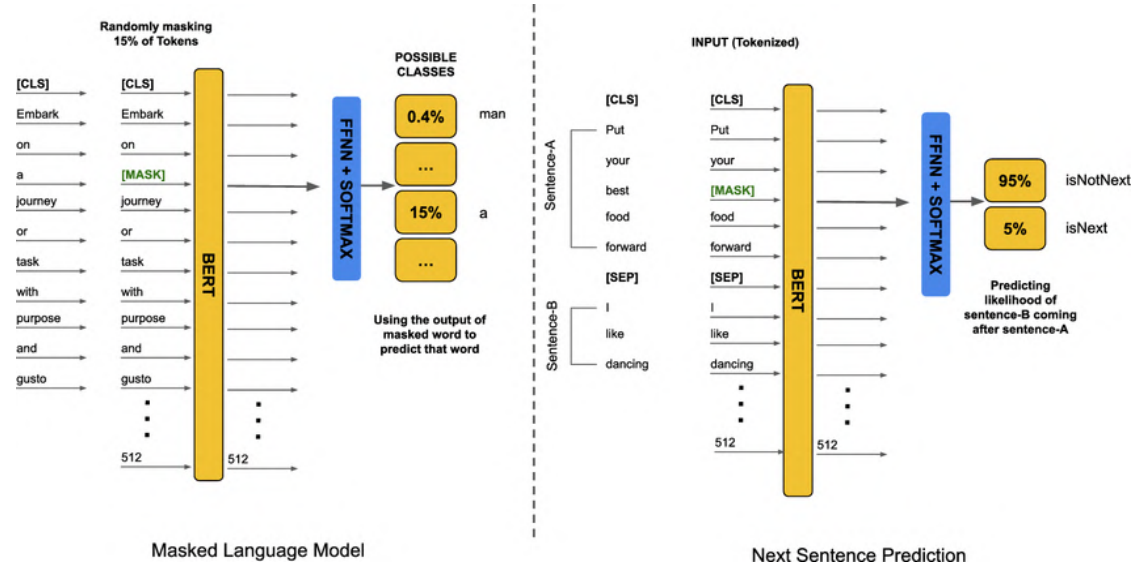


Figura 13. MLM y NLP en BERT

Fuente: (Barros et al., 2023)

BERT ha sido optimizado en múltiples versiones y tamaños según diferentes necesidades computacionales, surgiendo nuevas versiones del modelo (RoBERTa o DistilBERT). Sin embargo, al igual que todos los modelos que poseen una estructura y entrenamiento complejos, BERT tiene un alto coste computacional debido su gran número de parámetros. Asimismo, puede ser difícil de interpretar debido a su arquitectura distribuida, al no proporcionar reglas explícitas de decisión como otros modelos más simples.

3.1.3 Bibliotecas para Procesamiento de Lenguaje Natural

3.1.3.1 RE

La librería `re` de *Python* es un módulo utilizado para el manejo de expresiones regulares (*regular expressions*) que proporciona herramientas para la búsqueda, coincidencia y manipulación de patrones en cadenas de texto. Su implementación sigue el estándar definido por las expresiones regulares de *Perl*, lo que convierte a esta biblioteca en una herramienta versátil y flexible para tareas relacionadas con el procesamiento de texto.

Las expresiones regulares facilitan la realización de operaciones avanzadas de búsqueda y sustitución en grandes cantidades de texto, siendo ampliamente utilizadas en procesos como la tokenización, normalización y limpieza de datos textuales en NLP. Esta biblioteca ofrece métodos esenciales como `search()`, `match()`, `findall()`, `sub()`, y `split()`, que permiten manipular patrones en los textos que se pretenden analizar.

En aplicaciones de NLP, la utilización de expresiones regulares es esencial para la extracción de entidades, normalización de textos y preprocesamiento de corpus lingüísticos. Estudios anteriores han destacado su importancia en la segmentación de texto, identificación de entidades mencionadas y filtrado de ruido en datos lingüísticos no estructurados (Feldman & Sanger, 2006).

Una de las principales ventajas es su capacidad para gestionar grandes cantidades de texto de forma óptima, reduciendo el tiempo de procesamiento en comparación a otros métodos iterativos convencionales. No obstante, las expresiones regulares, por su sintaxis compacta y expresividad, pueden resultar complicadas de entender en patrones complejos. Para minimizar este inconveniente, es recomendable el uso de herramientas de depuración de expresiones regulares en entornos de desarrollo.

3.1.3.2 NLTK

Introducida inicialmente por Steven Bird y Edward Loper, `NLTK` (*Natural Language Toolkit*) es una de las librerías más empleadas para NLP en *Python*, siendo adoptada e implementada en diferentes áreas de investigación y usos prácticos.

Una de sus funciones más destacadas es la tokenización y segmentación de texto, que posibilita la segmentación de documentos en oraciones o palabras a través de algoritmos como `word_tokenize()` y `sent_tokenize()`, facilitando la extracción de información a partir de textos sin procesar. Además, posee métodos de *stemming* y lematización, que reducen las palabras a su raíz o forma base respectivamente. Para ello, emplea algoritmos

como `PorterStemmer` y `WordNetLemmatizer`, que mejoran la normalización del texto y permiten un análisis más eficiente. Otra funcionalidad destacada es el etiquetado gramatical (*POS tagging*), que asigna categorías gramaticales a las palabras dentro de una oración, facilitando el análisis sintáctico y semántico (Bird, 2006).

Se ha evidenciado que `NLTK` es un instrumento eficaz para la enseñanza y desarrollo de sistemas de NLP, principalmente gracias a su accesibilidad y amplia documentación (Perkins, 2010). No obstante, en comparación con otras bibliotecas más recientes como `spaCy` o `Transformers`, puede tener limitaciones en términos de eficiencia computacional, especialmente en actividades donde se analicen grandes cantidades de texto.

El uso de `NLTK` ha sido fundamental en aplicaciones como análisis de sentimientos, detección de *spam* y traducción automática. Gracias a su flexibilidad y compatibilidad con modelos avanzados, se ha posibilitado su integración en flujos de trabajo de procesamiento de texto en múltiples disciplinas.

3.1.3.3 TF-IDF

TF-IDF (*Term Frequency-Inverse Document Frequency*) es un indicador utilizado en la recuperación de información y la minería de textos para determinar la importancia de una palabra en un documento dentro de un conjunto de ellos. Esta técnica pondera la frecuencia de un término en un documento y la ajusta en función de la frecuencia de la palabra en el conjunto de documentos, facilitando la identificación de términos relevantes que definen el contenido de un documento (Salton et al., 1975).

El valor de TF-IDF se obtiene mediante la frecuencia del término o TF (*Term Frequency*) y la frecuencia inversa de documentos o IDF (*Inverse Document Frequency*). TF mide la frecuencia con la que un término t aparece en un documento d (Salton et al., 1975). La forma más común de calcularlo es:

$$\mathrm{TF}(t,d) = \frac{f_{t,d}}{\max\{f_{t',d}: t' \in d\}}$$

donde:

- $f_{t,d}$ es el número de veces que el término t aparece en el documento d.
- $\max\{f_{t',d}: t' \in d\}$ es la máxima frecuencia de cualquier término en el mismo documento.

Por su parte, IDF evalúa la importancia del término en el conjunto de documentos, reduciendo el peso de términos comunes y aumentando el de términos raros (Salton et al., 1975):

$$\mathrm{IDF}(t,D) = \log\left(\frac{N}{|d \in D: t \in d|}\right)$$

donde:

- N es el número total de documentos en el corpus.
- $|d \in D: t \in d|$ es el número de documentos que contienen el término t.

El TF-IDF es calculado entonces como:

$$\mathrm{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D)$$

Se trata de una herramienta considerablemente utilizada en motores de búsqueda y sistemas de recuperación de información para ponderar términos y mejorar la relevancia de los resultados. Sin embargo, presenta algunas limitaciones, sobre todo en contextos donde los términos importantes pueden aparecer con mucha frecuencia en varios documentos, lo que reduce su habilidad para poder diferenciar correctamente entre documentos. Asimismo, no captura relaciones semánticas entre palabras ni considera el orden de las palabras.

3.1.4 GOOGLE FACTCHECK CLAIM SEARCH

Google Fact Check Tools es una API (*Application Programming Interface*) que permite acceder y gestionar datos relacionados con el *fact-checking*. Facilita la búsqueda y publicación de información sobre afirmaciones verificadas.

La API permite buscar a través de dichas afirmaciones que han sido objeto de *fact-checking*, proporcionando detalles como el texto de la afirmación, el autor, la fecha y los resultados de la verificación. Las organizaciones pueden utilizar la API para agregar datos estructurados (*ClaimReview Markup*) a sus artículos de verificación, lo que mejora la visibilidad y accesibilidad de estas verificaciones (Google Fact Check Tools API, 2023).

El principal beneficio que otorga es el acceso unificado para consultar y publicar datos de *fact-checking*, facilitando la integración en otras aplicaciones o entornos de desarrollo. Además, dado que permite la publicación de datos estructurados, la API ayuda a que las verificaciones sean más fácilmente identificables y accesibles para el público general.

3.2 GESTIÓN DE LA APLICACIÓN

3.2.1 FLASK

`Flask` es un *framework* de código abierto escrito en *Python*, diseñado para facilitar el desarrollo de aplicaciones web de forma ligera, modular y extensible. Creado por Armin Ronacher en 2010 como parte del proyecto Pocoo, se caracteriza por seguir un enfoque minimalista, proporcionando solo aquellas herramientas estrictamente necesarias para la inicialización de un servidor web. Este enfoque permite dejar a elección del desarrollador la inclusión de componentes adicionales (Grinberg, 2018).

Su núcleo está construido sobre `Werkzeug`, una biblioteca WSGI (*Web Server Gateway Interface*), y utiliza `Jinja2`, un motor de plantillas que permite separar la lógica de negocio

de la capa de presentación. Esta estructura favorece la escritura de código limpio, comprensible y fácilmente escalable, adaptándose correctamente tanto a aplicaciones simples como a estructuras más complejas mediante la organización en *Blueprints*.

Desde un punto de vista computacional, `Flask` hace uso del modelo síncrono de *Python*, en el que cada solicitud HTTP se procesa en un hilo independiente. Esta estrategia es eficiente para aplicaciones de *backend* centradas en procesamiento de datos o integración con modelos de ML, donde el volumen de solicitudes concurrentes no es muy elevado.

Una de sus mayores ventajas radica en su simplicidad y flexibilidad, haciéndolo adecuado para proyectos de investigación o despliegue de modelos de IA. Asimismo, su integración nativa con librerías de aprendizaje automático/profundo de *Python* (como `tensorflow`, `scikit-learn` o `transformers`) lo convierten en una opción habitual para la implementación de servicios en entornos de producción ligera.

3.2.2 NODE.JS

Desarrollado por Ryan Dahl en 2009 y fundamentado en el motor V8 de Google, `Node.js` es un entorno de ejecución de JavaScript de código abierto y multiplataforma que facilita la ejecución de código fuera del navegador web. Su diseño se enfoca en un modelo de entrada/salida o I/O (*input/output*) asíncrono centrado en eventos, idóneo para aplicaciones que necesitan el manejo simultáneo de varias conexiones. A diferencia de modelos de servidores basados en hilos, `Node.js` emplea un solo hilo con un ciclo de eventos que facilita la ejecución de operaciones de I/O sin interrupciones, mejorando así el desempeño en aplicaciones con múltiples usuarios (Tilkov & Vinoski, 2010).

El modelo de ejecución de `Node.js` está basado en una arquitectura I/O impulsada por eventos (*event-driven*), sin bloquear las operaciones en ejecución mientras esperan una respuesta. Matemáticamente, la gestión de eventos se basa en el modelo de programación reactiva, en el que las funciones de *callback* se activan al finalizar una operación de I/O.

Esto posibilita gestionar al mismo tiempo un gran número de conexiones sin la necesidad de crear varios hilos, evitando la sobrecarga de memoria y los problemas de sincronización que afectan a otros modelos multihilo tradicionales (Tilkov & Vinoski, 2010).

Una de las características más importantes de este entorno de ejecución es su ecosistema de paquetes gestionado por `npm` (*Node Package Manager*), que ofrece acceso a una amplia gama de librerías. Ello permite la creación modular de aplicaciones y brinda a los desarrolladores la posibilidad de utilizar soluciones ya existentes para agilizar el desarrollo del *software*.

3.2.2.1 Express.js

`Express.js` es una infraestructura de aplicaciones web diseñada para trabajar con `Node.js` de forma mínima y flexible que proporciona una interfaz modular para la construcción de aplicaciones web escalables. Desde su lanzamiento, se ha consolidado en múltiples entornos de desarrollo gracias a su integración con el ecosistema de paquetes de `Node.js` y compatibilidad con arquitecturas de microservicios y desarrollo basado en RESTful APIs (OpenJS Foundation).

Una de las principales características de este marco de aplicaciones es su sistema de enrutamiento, con el que es posible definir reglas específicas sobre cómo una aplicación debe interactuar con las solicitudes HTTP en distintos puntos finales. Ello le permite gestionar de forma modular las peticiones entrantes y sus respectivas respuestas. Su arquitectura basada en *middleware* facilita la implementación de funcionalidades como validación de datos, autenticación, manipulación de cabeceras y control de errores (OpenJS Foundation)

`Express.js` tiene la capacidad de integrarse con otras plantillas y bases de datos, convirtiéndolo en una opción adecuada y versátil para el desarrollo de aplicaciones en múltiples dominios. Asimismo, dado que no impone una estructura rígida en este desarrollo, se pueden diseñar arquitecturas personalizadas según las necesidades específicas del

proyecto. Esta flexibilidad lo ha convertido en una de las herramientas más utilizadas en este contexto.

3.2.2.2 JSON Web Token

JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una método compacto e independiente para la transmisión segura de datos entre entidades como un objeto JSON. Este estándar facilita la verificación de la autenticidad de los datos y la integridad de la información enviada (Jones et al., 2015).

La estructura de un objeto JWT consta de tres partes: el encabezado (*header*), el contenido (*payload*) y la firma (*signature*). El primero de ellos especifica el algoritmo de firma empleado y el tipo de *token*, mientras que el segundo contiene las reclamaciones (*claims*), que son afirmaciones respecto a una entidad (generalmente, el usuario) y datos adicionales. La firma se utiliza para confirmar que la persona que entrega el *token* es quien dice ser y para garantizar que el mensaje no fue modificado durante la transmisión del mismo (Jones et al., 2015).

Dentro del marco de la autenticación, cuando un usuario inicia sesión correctamente, se le devuelve un JWT. Este *token* debe ser enviado al cliente a través de un método seguro, por ejemplo, una cookie HTTP-only. Cuando el usuario desea acceder a una ruta o recurso seguro, necesita enviar el JWT, generalmente en el encabezado HTTP de autorización empleando el esquema *Bearer*. Este es un procedimiento de autenticación sin estado, dado que la sesión del usuario no se guarda en el servidor (Jones et al., 2015).

3.2.3 Base de Datos

3.2.3.1 SQLite

SQLite es un sistema de gestión de bases de datos relacional de código abierto, conocido por su ligereza, sencillez y eficacia en ambientes con recursos limitados. Desarrollado por D.

Richard Hipp, se caracteriza por no necesitar un servidor independiente, lo que simplifica su integración directa en aplicaciones, en particular en dispositivos móviles, navegadores web y sistemas embebidos (Hipp et al., 2016).

Una de las principales características de SQLite es su implementación de control de concurrencia mediante un modelo de bloqueo basado en archivos, lo que posibilita que varios procesos lean al mismo tiempo, a pesar de que las escrituras se lleven a cabo de manera secuencial. Aunque SQLite no aplica un modelo integral de control de concurrencia multiversión (MVCC), garantiza la observancia de los principios ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad) a través de la utilización de transacciones y un sistema de registro. Asimismo, SQLite proporciona por defecto un nivel de aislamiento de transacciones serializable, asegurando que las operaciones simultáneas se administren de forma segura en la mayoría de los escenarios de uso (Hipp et al., 2016).

3.3 Tecnologías Web y de Interfaz de Usuario

3.3.1 React

`React.js` es una librería de código abierto de JavaScript diseñada para crear interfaces de usuario, en particular para aplicaciones de una sola página, mantenida por Meta y una comunidad de desarrolladores y empresas. El objetivo principal con el que ha sido desarrollada es simplificar la creación de interfaces de usuario dinámicas e interactivas, centrándose en la creación de componentes reutilizables que gestionan su propio estado.

El uso del Virtual DOM (*Document Object Model*) es una de sus principales características. `React.js` crea una representación virtual en la memoria en lugar de modificar el DOM del navegador directamente. `React.js` actualiza el DOM virtual cada vez que cambia el estado de un componente y luego lo compara con el DOM real para aplicar

solo los cambios necesarios. Al reducir las operaciones de manipulación directa del DOM, este método aumenta significativamente el rendimiento.

Además, `React.js` introduce JSX (*JavaScript XML*), una extensión que permite crear estructuras similares a HTML dentro del código JavaScript, facilitando la creación de componentes visuales mientras se mejora la legibilidad y la facilidad de mantenimiento del código. Aunque el uso de JSX no es obligatorio, se ha convertido en un procedimiento estándar para los desarrolladores que trabajan con esta biblioteca.

3.3.2 Chakra UI

`Chakra UI` es una librería de componentes de interfaz de usuario de `React` que facilita la creación de interfaces accesibles y estilizadas de manera consistente. Proporciona una colección de componentes modulares y personalizables que cumplen con los estándares de accesibilidad, permitiendo a desarrolladores crear aplicaciones con una experiencia de usuario coherente y de alta calidad sin necesidad de diseñar y codificar cada componente.

Utiliza un sistema de diseño basado en temas y plantillas, lo que permite definir estilos globales y mantener la coherencia visual en la aplicación desarrollada. Asimismo, `Chakra UI` se integra fácilmente con otras herramientas y bibliotecas dentro del ecosistema de `React`, lo que facilita su adopción en otros proyectos que utilicen esta librería. Su diseño modular permite importar solo los componentes necesarios, lo que contribuye a optimizar el rendimiento de la aplicación global.

3.3.3 Fetch API

La `Fetch API` es una interfaz de JavaScript que reemplaza al antiguo objeto `XMLHttpRequest` y permite realizar consultas HTTP rápidas y sencillas. La obtención y el envío de datos en aplicaciones web se simplifican con esta API al ofrecer un método flexible y adaptable para interactuar con los recursos de la red.

El uso de promesas (*promises*) en la `Fetch API` es una de sus principales ventajas, dado que facilita la gestión de actividades asíncronas y mejora la legibilidad del código. Al realizar una solicitud con `fetch()`, se devuelve una promesa con un objeto `Response` una vez que el servidor responde. A diferencia de los *callbacks* tradicionales, esta función permite encadenar métodos `then()` para procesar la respuesta y manejar mejor los errores. La sintaxis básica para realizar una solicitud con la `Fetch API` es la siguiente:

```
fetch('https://factguard.api.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error in the request: ' + response.status);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Listado 1. Ejemplo de Llamada a una API REST usando `Fetch API`

Fuente: Elaboración Propia

`Fetch API` no rechaza la promesa en respuestas HTTP con códigos de error (como `404` o `500`). Es por ello que es necesario verificar manualmente la propiedad `ok` del objeto `Response` para manejar correctamente los casos de error.

Capítulo 4. Estado de la Cuestión

La gran evolución experimentada en modelos y algoritmos de ML y DL ha permitido una mejora considerable en las técnicas utilizadas para la identificación de las *fake news*, desde modelos más sencillos a métodos mixtos y más sofisticados. Ello ha permitido un mejor manejo de la complejidad del lenguaje utilizado en este tipo de noticias y ha incrementado la precisión de los sistemas de detección.

4.1 Soluciones para la Detección de Fake News

Con el fin de proporcionar una comprensión clara y estructurada de las distintas soluciones presentes, la Figura 14 muestra un esquema general que sintetiza las principales familias de modelos para la detección de noticias falsas que se detallarán a lo largo de este apartado.

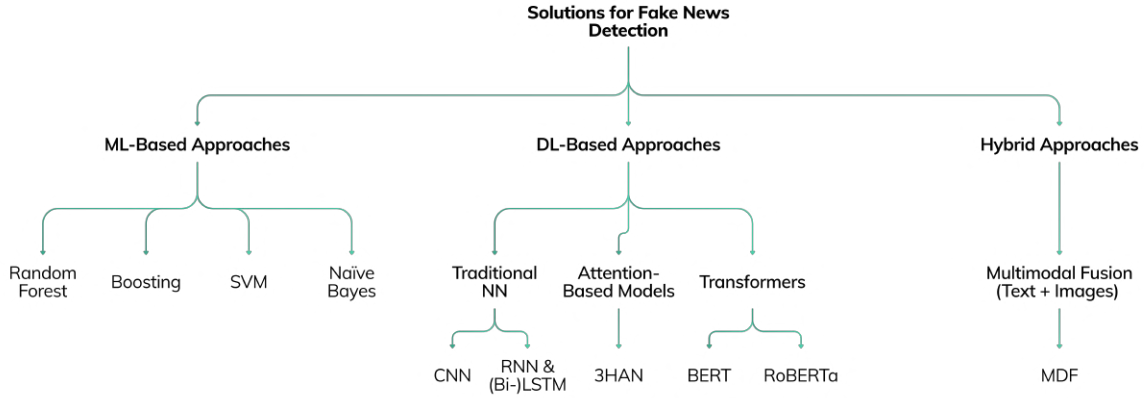


Figura 14. Esquema General de Soluciones tecnológicas para la Detección de Fake News

Fuente: Elaboración Propia

4.1.1 ALGORITMOS DE APRENDIZAJE AUTOMÁTICO

Las técnicas de aprendizaje automático han desarrollado un papel fundamental en la detección automática de noticias falsas, especialmente en el ámbito de las redes sociales, donde ejercen un efecto considerable. Investigaciones recientes muestran una visión general de los modelos utilizados y los problemas que surgen a raíz de la implementación en dicho campo (Shu et al., 2017; Zhou & Zafarani, 2020).

La detección de *fake news* empleando algoritmos de ML se ha apoyado enormemente en características extraídas del contenido escrito. Estos atributos incluyen elementos lingüísticos como el estudio y el conteo de palabras, las frecuencias de n-gramas (secuencia de n elementos consecutivos extraída de un texto) y el análisis de sentimientos (*sentiment analysis*), empleados como entrada para algoritmos de clasificación supervisados como SVM, *Naïve Bayes* y *Random Forest*. Esto se observa en la Figura 15, donde se aplican técnicas de NLP para segmentar, limpiar y extraer características relevantes del texto y utilizarlas como entrada para los modelos mencionados.

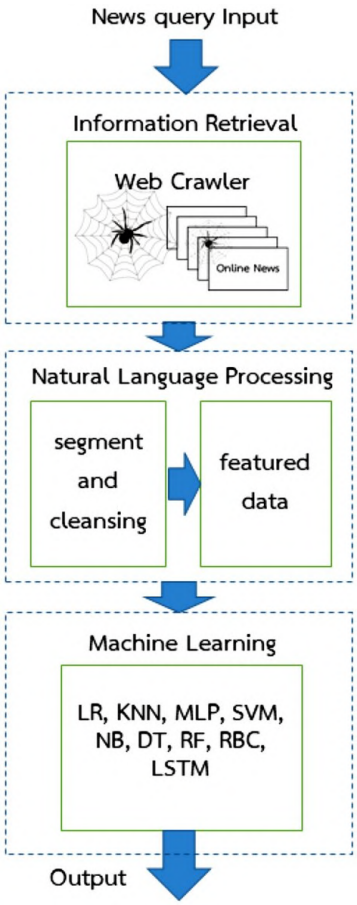


Figura 15. Flujo de Procesamiento para la Detección de Fake News con NLP y ML

Fuente: (Meesad, 2021)

Estos modelos son desarrollados basándose bajo la premisa de que las *fake news*, dada su naturaleza de engaño intencionado, poseen rasgos estilísticos diferentes, como un lenguaje más persuasivo o sensacionalista (Zhou & Zafarani, 2020) y un tono emocional determinado en los titulares diseñado para incitar un mayor número de clics (*clickbait*) (Shu et al., 2017). Esto se manifiesta mediante el uso de determinados rasgos léxicos (aparición de palabras únicas, longitud de las frases) y sintácticos (aplicación de puntuación, segmentos del discurso). Asimismo, algunos algoritmos también examinan rasgos más específicos como la cantidad de referencias externas o la longitud media de los párrafos.

Pese a las ventajas que poseen estos enfoques, hay algunas restricciones presentes. Primeramente, la gran dependencia en el uso de características obtenidas de manera manual crea un sesgo considerable y requiere de una tarea compleja de ingeniería de datos (Zhou & Zafarani, 2020). Ello disminuye la escalabilidad de los modelos cuando se enfrentan a grandes volúmenes de datos no estructurados.

El desempeño de estos modelos se ve afectado por la calidad de la información de entrada. Los datos que contengan información incompleta, ruidosa y muy variada dificultan el desarrollo de modelos robustos, así como aquellos datos que contengan noticias falsas relacionadas con nuevos sucesos o eventos emergentes que no están recogidos en bases de datos existentes (Shu et al., 2017).

Por otro lado, también existen otros métodos más adecuados para tratar la difusión de noticias falsas en las redes sociales, donde factores como los patrones de propagación y las interacciones sociales pueden ser indicadores clave (Zhou & Zafarani, 2020). Ello se observa en la Figura 16, donde la estructura de publicación y el comportamiento de los usuarios que interactúan con las *fake news* siguen rutas de propagación diferentes en comparación con las noticias verdaderas.

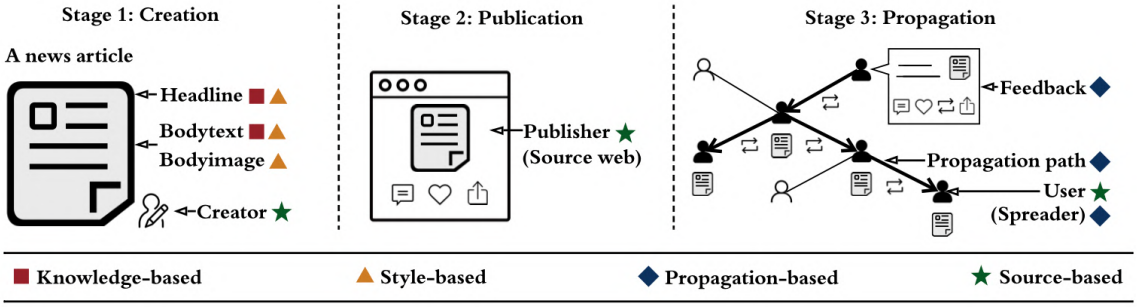


Figura 16. Proceso de Creación, Publicación y Propagación de Fake News

Fuente: (Zhou & Zafarani, 2020)

4.1.2 REDES NEURONALES Y MODELOS PROFUNDOS

La introducción de modelos basados en redes neuronales profundas ha transformado el campo de la detección automática de *fake news*, facilitando la captura de patrones más complejos y contextuales del lenguaje que previamente no se podían obtener con métodos convencionales de ML. Ello no solo ha posibilitado una mayor precisión a la hora de identificar noticias engañosas, sino que también ha aumentado la capacidad de gestionar datos no estructurados procedentes de redes sociales (*tweets*, *posts*).

En 2017, se presentó la Red de Atención Jerárquica de Tres Niveles o 3HAN (*Three-level Hierarchical Attention Network*), un modelo jerárquico que emplea diferentes niveles y capas de atención para analizar las palabras, oraciones y titulares en un artículo (Singhania et al., 2017). Para ello, utiliza TF-IDF para capturar la importancia de las palabras en el marco de cada noticia y GRU (*Gated Recurrent Units*) como arquitectura para modelar las dependencias a largo plazo de los textos analizados. Tal y como se ilustra en la Figura 17, en las capas inferiores se emplean un codificador de palabras y uno de oraciones con mecanismos de atención, para destacar aquellos términos relevantes y agrupa la información a nivel de oración respectivamente. La clasificación final se realiza mediante una función de activación sigmoide que, en función del codificador de titulares y cuerpo del artículo, genera un vector representativo de la noticia.

Este método se basa en la asignación de pesos diferenciados a cada una de las partes de un artículo según su importancia, lo que permite al modelo concentrarse en los aspectos más significativos de cada artículo para categorizarlo como verdadero o falso. Con una precisión alcanzada del 96.77%, se trata de una de las alternativas más eficaces para la identificación automatizada de noticias falsas.

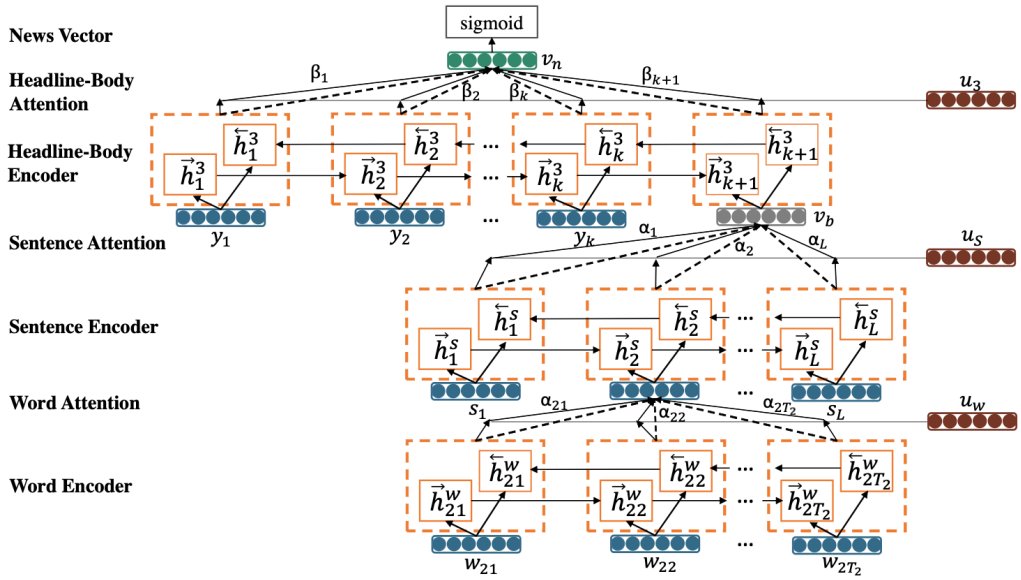


Figura 17. Modelo 3HAN y su Estructura Jerárquica de Atención

Fuente: (Singhania et al., 2017)

Posteriormente, se hicieron otros estudios utilizando otras arquitecturas como las CNNs y LSTMs, donde hicieron uso de estos modelos junto con LSTM bidireccionales (Bi-LSTM) para examinar la relación entre el titular y el cuerpo de las noticias (Abedalla et al., 2020).

Dado que las CNN tienen la capacidad de identificar características locales significativas de un texto, estas redes se emplearon para detectar aquellos patrones peculiares encontrados en los titulares y contenidos de las *fake news*, como combinaciones de palabras que podrían sugerir similitudes o discrepancias contextuales. Por su parte, las LSTM se utilizaron para registrar dependencias a largo plazo en las secuencias de texto, con el objeto de entender el contexto global de todo el artículo. Asimismo, los autores implementaron redes Bi-LSTM para poder manejar secuencias de textos en ambas direcciones y dotar al modelo de la capacidad de tomar en cuenta tanto las relaciones previas como las subsiguientes en las secuencias de texto y así potenciar el entendimiento del contexto.

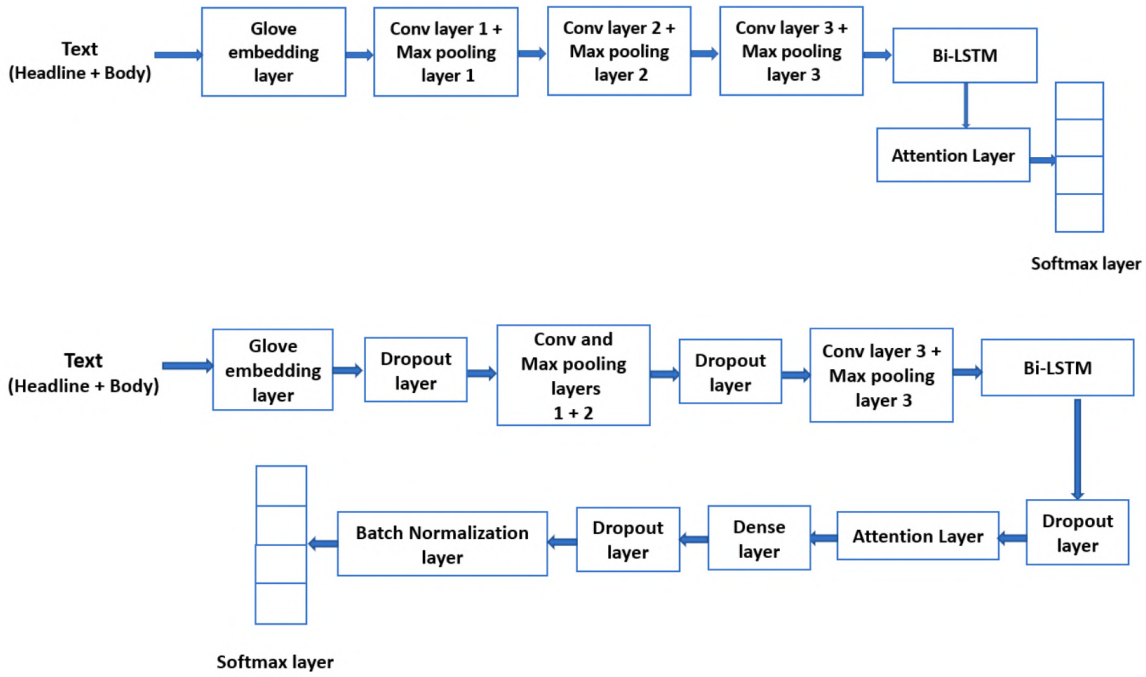


Figura 18. Combinación de Redes CNN y Bi-LSTM

Fuente: (Singhania et al., 2017)

La combinación de estas técnicas, que resulta en la arquitectura mostrada en la Figura 18, permitió alcanzar una precisión final del 71.2% (Abedalla et al., 2020). Aunque es inferior a la realizada por otras investigaciones, es relevante dado que el conjunto de datos empleado constaba de múltiples categorías de etiquetas, lo que demostró la efectividad de modelos de aprendizaje profundo para hacer frente a problemas de clasificación más detallada en la detección de noticias falsas.

Más allá del uso de redes neuronales, la aparición de los *Transformers* ha supuesto un avance adicional en lo que respecta a las tareas centradas en NLP. Un claro ejemplo de ello es BERT, caracterizado por su arquitectura bidireccional basada en un mecanismo de autoatención que permite comprender el contexto completo de una oración considerando tanto las palabras anteriores como posteriores (Devlin et al., 2019). Por su parte, RoBERTa mejora el entrenamiento de BERT mediante el aumento del tamaño de los datos de

entrenamiento y un mejor ajuste de los hiperparámetros del modelo, lo que le ha permitido obtener un mejor desempeño en ciertas tareas y conjuntos de datos, en comparación con BERT (Liu et al., 2019).

A diferencia de Bi-LSTM, que también procesa texto en ambos sentidos, el mecanismo de autoatención de los *Transformers* no se basa en la transmisión secuencial de información. BERT y RoBERTa procesan todas las palabras de manera paralela y determinan las conexiones entre todas las palabras de una oración al mismo tiempo, lo que facilita la comprensión de un contexto más integral y exacto.

En este contexto, el modelo *FakeBERT* (Kaliyar et al., 2021), combina las ventajas de BERT con las habilidades de las CNN unidimensionales o 1D-CNN (*One-dimensional CNN*), con el propósito de obtener tanto representaciones semánticas como relaciones de larga distancia en las noticias analizadas. La elección detrás de 1D-CNN se basa en su capacidad para capturar patrones locales y n-gramas en secuencias de texto, lo que resulta de gran importancia a la hora de reconocer rasgos estilísticos y contextuales (Kaliyar et al., 2021).

Como se muestra en la Figura 19, al emplear varios filtros con distintos tamaños de núcleo, filtros, capas de convolución y de agrupación-máxima (*max-pooling*), las 1D-CNN son capaces de reconocer varias estructuras sintácticas y semánticas, complementando las representaciones contextuales bidireccionales de BERT que se introducen al modelo. De esta forma, es posible gestionar las complejidades y ambigüedades del lenguaje natural, factores clave en lo que se refiere a la identificación de *fake news*, presentes en textos estructurados y no estructurados a gran escala.

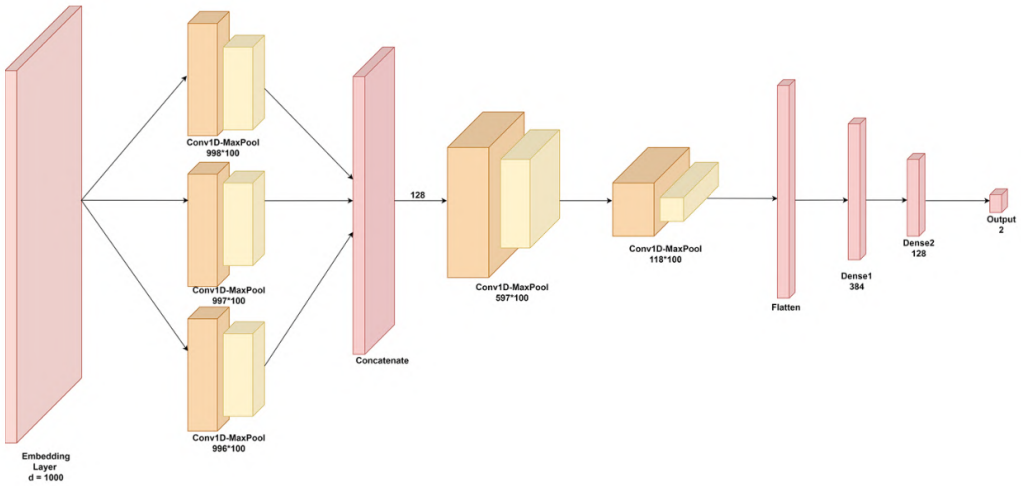


Figura 19. Estructura del Modelo FakeBERT

Fuente: (Kaliyar et al., 2021)

Los experimentos realizados con *FakeBERT* demostraron una precisión final del 98.90%, lo que supone un avance considerable en relación con otros modelos de ML. Utilizando el mismo conjunto de datos se compara el rendimiento con otros algoritmos, como *Multinomial Naïve Bayes*, *Random Forest* y *Decision Tree*, que lograron precisiones del 89.97% al 73.65%. Por otro lado, arquitecturas de DL de redes neuronales clásicas, como CNN y LSTM, alcanzaron precisiones del 97.55% y 92.70%. Si bien se tratan de resultados óptimos, el estudio resalta la eficacia de *FakeBERT* al fusionar las representaciones bidireccionales de BERT con la habilidad de las CNN para recoger características locales y globales de los datos, últimamente ofreciendo un modelo sólido para la tarea de la identificación de noticias falsas (Kaliyar et al., 2021).

Pavlov y Mirceva también estudiaron la aplicación de modelos entrenados BERT y RoBERTa para la detección de *fake news* vinculadas a datos de la pandemia de COVID-19. Mediante el uso del *dataset* de ConstraintAI, que comprende 10,700 muestras balanceadas entre noticias verdaderas y falsas (Felber, 2021), los modelos fueron ajustados utilizando

transfer learning, que permitió que aprovechar modelos pre-entrenados en otros conjuntos de datos, ajustándolos al campo particular en el cual centraron su estudio.

El modelo BERT empleado (`covid-twitter-bert-v2`) (Müller et al., 2023) fue entrenado únicamente en un conjunto de *tweets* vinculados a COVID-19, mientras que el modelo RoBERTa (`twitter-roberta-base-sentiment`) (Loureiro et al., 2022) se entrenó con cerca de 58 millones de *tweets* generales y, posteriormente, modificado para enfocarse en tareas de *sentiment analysis*. El propósito de la investigación consistió en utilizar las ventajas del preentrenamiento específico de BERT centrado en datos de COVID-19 y realizar una comparación con RoBERTa, que tenía un conocimiento de los datos más general (Pavlov & Mirceva, 2022).

Después del *fine-tunning*, los resultados mostraron una leve superioridad de BERT frente a RoBERTa en todas las métricas de evaluación, precisión (98.31 % en comparación con 97.52 %), *F1-score* (98.31 % en comparación con 97.52 %), *recall* (98.83 % en comparación con 98.21 %) y el Coeficiente de Correlación de *Matthew* (0.9663 en comparación con 0.9504). Los autores concluyen que la ventaja de BERT es debida al entrenamiento previo en *tweets* específicos de COVID-19, lo que facilitó la gestión del contexto por parte del modelo, en detrimento de RoBERTa, cuyo preentrenamiento fue más amplio y menos centrado en el conjunto empleado (Pavlov & Mirceva, 2022).

4.1.3 Enfoques Híbridos de Clasificación

Dada la variedad de los datos en los que se presentan las noticias (desde vídeos hasta imágenes y *posts* publicados en redes sociales), también han surgido alternativas eficaces que son resultado de la fusión de distintas arquitecturas y métodos para afrontar las complejidades de los datos multimodales. Un modelo que sobresale en este enfoque es el Modelo de Fusión Dinámica o MDF (*Dynamic Fusion Model*), centrado en fusionar la información proveniente de distintos formatos, como texto e imágenes, empleando un marco

dinámico que trata tanto la incertidumbre como el ruido existente en la información variada (Lv et al., 2024). Se encuentra formado por tres módulos: el Módulo de Estimación de Incertidumbre o UEM (*Uncertainty Estimation Module*), la Red de Grafos de Atención o GAT (*Graph Attention Network*) y la Red de Fusión Dinámica o DFN (*Dynamic Fusion Network*).

El UEM captura la incertidumbre dependiendo del formato de la información, sea texto o imagen, a través de un sistema de atención multi-cabeza, garantizando representaciones sólidas ante el ruido y la ambigüedad. Ello se realiza asignando pesos a las representaciones de texto e imagen, modelando las características de cada uno en un espacio gaussiano. Por otro lado, la GAT se encarga de modelar las interacciones entre los diferentes formatos, otorgando pesos dinámicos a texto e imágenes mediante un método basado en grafos para recoger las relaciones complejas que a menudo no son evidentes en un análisis unimodal. Finalmente, la DFN combina todas estas representaciones mediante la teoría de evidencia de *Dempster-Shafer*, otorgando grados de confianza a cada modalidad y modificando su impacto en la predicción final de acuerdo a su calidad y pertinencia (Lv et al., 2024).

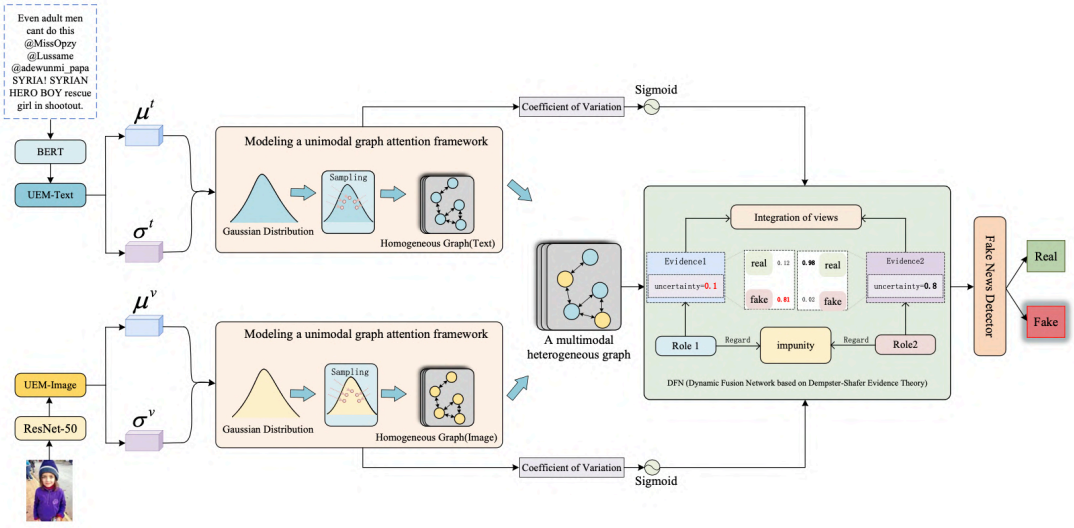


Figura 20. Integración Multimodal en el Modelo MDF

Fuente: (Lv et al., 2024)

La complejidad de esta arquitectura, mostrada en la Figura 20, permite al modelo gozar de una capacidad de adaptación frente a datos multimodales heterogéneos. En la evaluación realizada en conjuntos de datos públicos de X y Weibo los resultados fueron destacables, con una precisión del 94.7% y un *F1-score* del 96.2%. No obstante, la complejidad del modelo hace que su generalización a otros contextos o dominios fuera de los datos mencionados pueda requerir modificaciones en su arquitectura y parámetros. Asimismo, la sensibilidad a datos incompletos, ruidosos o irrelevantes puede afectar en la generación de representaciones precisas incluso teniendo los mecanismos de estimación de incertidumbre (Lv et al., 2024).

4.2 Soluciones para la Prevención de Fake News

4.2.1 Sistemas de Fact-Checking Automatizados

En 2014, Vlachos y Riedel reconocieron los principales problemas de los sistemas de *fact-checking*, recalcando la importancia del contexto en el que se encuentran las noticias, del tiempo en el proceso de comprobación y la necesidad de emplear múltiples fuentes para comprobar la veracidad de las declaraciones. Los autores enfatizaron que tanto la ambigüedad del lenguaje empleado en las *fake news* y la carencia de bases de datos verificadas representaban importantes limitaciones en este campo (Vlachos & Riedel, 2014).

Estas restricciones llevaron a la necesidad de incrementar la robustez y la eficiencia de estos sistemas de comprobación, incorporando diversas fuentes de información y mejorando su capacidad de entendimiento del lenguaje natural. Gracias al desarrollo de nuevas técnicas y la introducción de nuevos algoritmos más eficientes y modelos con mayor precisión, las herramientas de *fact-checking* automatizadas se han convertido en instrumentos de gran relevancia para luchar contra la desinformación y la difusión de noticias falsas, ofreciendo mayor rapidez y precisión que procedimientos manuales. El uso de modelos de ML y NLP permiten a los sistemas analizar distintas declaraciones y contrastarlas con bases de datos fiables para, de esta forma, demostrar su autenticidad.

Un ejemplo destacado es *ClaimBuster*, un sistema *end-to-end* que identifica declaraciones para categorizarlas en función de su importancia con el propósito de comprobar su veracidad (Hassan et al., 2017).

Como se observa en la Figura 21, el módulo *Claim Monitor* recopila declaraciones provenientes de diversas fuentes, para que el *Claim Spotter* pueda analizar el contenido y filtrar aquellas afirmaciones que deban ser verificadas. A continuación, el *Claim Matcher* compara las declaraciones detectadas con bases de datos de verificaciones previas, y si se encuentra una coincidencia, se genera un documento con los resultados. En caso contrario, el *Claim Checker* consulta otras bases y fuentes web para intentar proporcionar información adicional. El modelo base consiste en un clasificador SVM donde se utilizó un conjunto de datos etiquetado manualmente, con afirmaciones categorizadas en alta, media y baja importancia para el *fact-checking*. Estas etiquetas permiten al modelo distinguir a una declaración verificable de una que no lo es.

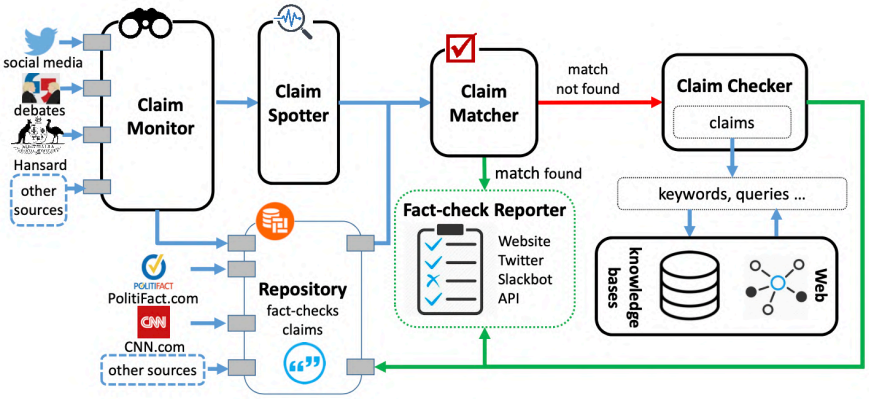


Figura 21. Flujo de Trabajo de ClaimBuster

Fuente: (Hassan et al., 2017)

En términos de rendimiento, la herramienta alcanzó una precisión del 79% en la detección de afirmaciones que podrían ser verificadas dentro de discursos políticos (Hassan

et al., 2017). Asimismo, su capacidad para integrar instrumentos de búsqueda en bases de datos de hechos posibilita la automatización parcial del proceso de verificación.

Por otro lado, FEVER es un conjunto de datos creado específicamente para la extracción y verificación de hechos contra otras fuentes, como Wikipedia (Thorne et al., 2018). En este *dataset* se incluyen 185,445 declaraciones creadas a partir de modificaciones manuales de oraciones recogidas, y clasificadas en tres categorías: *supported*, *refuted* y *notenoughinfo*.

Gracias a FEVER, se hace posible el desarrollo de DOMLIN, un modelo destinado para realizar la tarea de comprobación automática de afirmaciones basado en ese conjunto (Stammbach, 2019). Este sistema incorpora un proceso que incluye recolección de evidencia, filtrado e identificación de implicación textual o RTE (*Recognizing Textual Entailment*). Durante la primera fase, el sistema recupera oraciones significativas mediante un modelo de recuperación que se basa en búsquedas contextuales en documentos y vínculos hipertextuales. Luego, un modelo BERT mide la relevancia de las oraciones recuperadas para que, finalmente, el módulo de RTE categorice las declaraciones en las etiquetas mencionadas con anterioridad (Stammbach, 2019).

Un año más tarde, se desarrolla la evolución de DOMLIN, lo que condujo a la creación de DOMLIN++, una versión mejorada que incluyó avances en las etapas de recuperación de evidencia y clasificación (Stammbach & Ash, 2020). En esta versión se actualizó el modelo BERT a RoBERTa en lo que se refiere a tareas de filtrado y RTE, y se incorporó un nuevo módulo para la recuperación de documentos basado en la similitud TF-IDF. Las nuevas implementaciones para combinar afirmaciones y evidencias condujeron a un mejor desempeño del sistema, logrando una precisión del 77.48% (Stammbach & Ash, 2020).

En este contexto, se crea e-FEVER, una ampliación de DOMLIN++ que integra explicaciones generadas automáticamente a través del modelo de lenguaje de gran tamaño o LLM (*Large Language Model*) GPT-3. Este sistema hace uso del aprendizaje de pocos

ejemplos o instancias (*few-shot learning*) de GPT-3 para generar resúmenes abstractivos que vinculan la declaración con la evidencia obtenida. Estas explicaciones no solo potencian la interpretación del sistema, sino que también fortalecen su capacidad para clasificar las declaraciones en las respectivas etiquetas de manera más exacta. En las pruebas realizadas, e-FEVER consiguió una precisión cercana a 75% si se usaban como entradas la declaración y un resumen de la misma, mientras que si solo se usaba la declaración disminuía al 60% (Stammbach & Ash, 2020).

Posteriormente, en 2022, se desarrolla un sistema semiautomatizado llamado *FacTeR-Check* con objetivos como el seguimiento de la desinformación y el estudio de su difusión en las redes sociales. Utilizando tecnologías de NLP y modelos multilingües, como XLM-RoBERTa, contrastaron y examinaron declaraciones en distintos idiomas, prestando atención a la detección de similitudes semánticas y vínculos contextuales (Martín et al., 2022).

El sistema, cuya arquitectura se exhibe en la Figura 22, está compuesto por un módulo para evaluar la similitud semántica usando un conjunto de modelos *Transformer* para producir representaciones vectoriales de las declaraciones. Ello facilita el cálculo de métricas como la similitud coseno (*cosine simmilarity*) entre las afirmaciones procesadas y aquellas que han sido verificadas previamente. Se trata de una fase crucial para detectar patrones y vínculos que sean de utilidad para corroborar o desmentir las nuevas declaraciones confirmadas (Martín et al., 2022).

Una vez identificadas las declaraciones similares, un modelo de inferencia de lenguaje natural o NLI (*Natural Language Inference*) establece si hay una relación de implicación, contradicción o neutralidad entre las afirmaciones para poder determinar si esta declaración respalda, contradice o no está vinculada con un rumor ya existente. Asimismo, el sistema incorpora una funcionalidad específica para la recuperación automatizada de datos en redes sociales, creado específicamente para plataformas como X. Este módulo produce consultas

que se basan en términos extraídos de las declaraciones y recupera aquellas publicaciones vinculadas, lo que facilita el seguimiento de la evolución de la información en tiempo real (Martín et al., 2022).

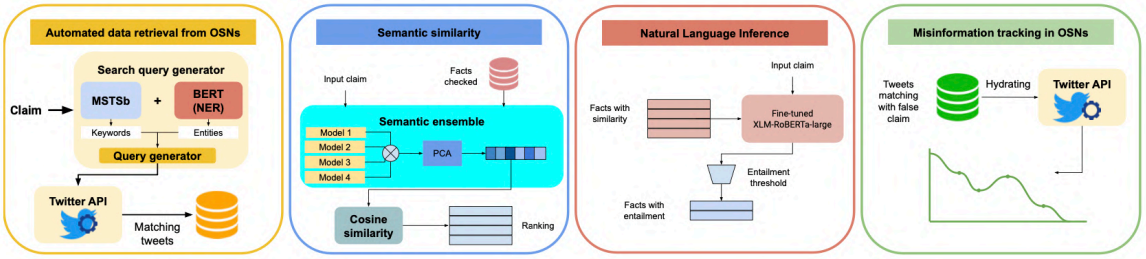


Figura 22. Componentes del Sistema FacTeR-Check

Fuente: (Martín et al., 2022)

Evaluado con datos relacionados con rumores acerca de la pandemia COVID-19 en español, las pruebas revelaron una gran precisión obtenida por *FacTeR-Check* tanto en español como en inglés, recalcando su habilidad para manejar datos en múltiples idiomas. Algunas de las limitaciones del sistema se encuentran en la dependencia de bases de datos basadas en declaraciones comprobadas anteriormente, lo que puede limitar la efectividad del sistema ante noticias falsas o rumores completamente nuevos. Además, el *pipeline* de verificación en el que se basa el sistema está restringido información textual y no contempla otros formatos de datos como imágenes, vídeos o audios (Martín et al., 2022).

Por último, cabe mencionar el modelo de FACT-GPT, creado por Choi y Ferrara, que emplea LLMs como GPT-4 y GPT-3.5-Turbo para automatizar la etapa de comparación de declaraciones en el proceso de *fact-checking*. Este sistema también tiene como objetivo establecer si la información que circula por redes sociales respalda, contradice o es neutral con respecto a afirmaciones previamente desmentidas (Choi & Ferrara, 2024). Para ello, el sistema produce datos artificiales que han sido creados específicamente para tareas de comparación a través de LLMs, incluyendo información sobre *tweets* generados a partir de afirmaciones comprobadas de bases como *Google Fact Check Tools* y *PolitiFact*. Como se

representa en la Figura 23, el proceso está basado en la recopilación de afirmaciones desacreditadas por otros sistemas de *fact-checking*, que luego sirven de base para la generación de publicaciones sintéticas mediante modelos de lenguaje como GPT.

Estas publicaciones se combinan con ejemplos reales extraídos de redes sociales, permitiendo la creación de un conjunto de datos sintético empleado para entrenar un modelo más pequeño y ajustado específicamente para esta tarea. Las declaraciones son categorizadas por parejas en tres grupos, respaldo (*entailment*), contradicción (*contradiction*) y neutral. De esta forma, un par se clasifica como respaldo si la veracidad de una afirmación A implica la verdad de otra B, neutral si la veracidad de A no afirma ni niega la verdad de B y, contradicción si la veracidad de A lleva a la conclusión de que la afirmación descrita B es falsa (Choi & Ferrara, 2024).

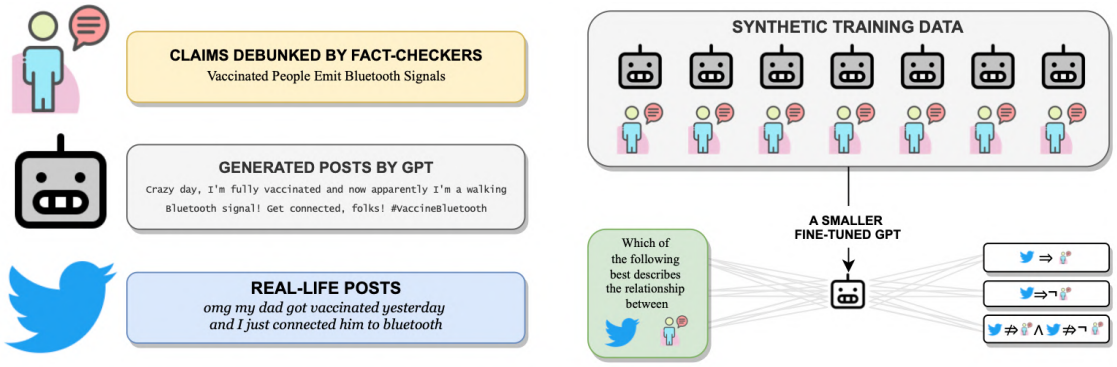


Figura 23. Generación de Datos Sintéticos en el Modelo FACT-GPT

Fuente: (Martín et al., 2022)

Durante las pruebas realizadas, se detectó que el modelo de GPT-3 entrenado con *datasets* creados con GPT-4 consiguió una precisión del 73%, superando a las métricas obtenidas por otros modelos como GPT-3.5-Turbo y LLaMA-2. Sin embargo, como en el resto de herramientas, este modelo se encuentra con limitaciones en lo que respecta a categorizar declaraciones erróneamente como contradictorias, restringiendo su capacidad para identificar refutaciones (Choi & Ferrara, 2024).

Capítulo 5. Sistema Desarrollado

En este capítulo se presenta la descripción técnica del sistema desarrollado en el marco del presente trabajo. Dado que FactGuard se encuentra dividido en dos módulos funcionales complementarios, se expone el desarrollo seguido para cada uno de ellos.

Asimismo, se explicará la arquitectura completa de la aplicación que engloba tanto la interfaz de usuario como el *backend* implementado.

5.1 Visión General

5.1.1 Análisis de Requisitos

Con el objetivo de guiar el diseño e implementación de FactGuard, se llevó a cabo una identificación detallada de los requisitos funcionales y no funcionales. Esta clasificación ha permitido asegurar el cumplimiento de los objetivos del proyecto y garantizar tanto la calidad técnica del desarrollo como la experiencia del usuario final.

5.1.1.1 Requisitos Funcionales

En cuanto a los requisitos funcionales del sistema, la aplicación debe ser capaz de:

Código	Descripción
RF1	Permitir el registro de nuevos usuarios y autenticación mediante credenciales (email y contraseña).
RF2	Generar y validar los *tokens* de sesión (JWT) entre la aplicación y los servidores del *backend* para garantizar el acceso seguro a las funcionalidades principales.
RF3	Conceder acceso al perfil de usuario para consultar y actualizar los datos personales, así como gestionar de forma segura el cambio de contraseña.
RF4	Permitir introducir y analizar noticias mediante el módulo de detección de *fake news* (**FactGuard Detect**).

RF5	Posibilitar al usuario la configuración de un umbral mínimo de confianza para que el sistema clasifique una noticia como verdadera o falsa.
RF6	Ofrecer un módulo de verificación de afirmaciones a través de la API de *Google FactCheck Claim Search* (**FactGuard Verify**).
RF7	Mostrar los resultados de cada análisis de manera clara. En el caso de FactGuard Detect: las predicciones por modelo, confianza y resultado final. En FactGuard Verify: veredicto, fuente verificadora y URL de la evidencia.
RF8	Permitir la consulta del historial de detecciones y verificaciones realizadas por cada usuario, posibilitándole la capacidad de eliminar cualquier análisis previo desde el panel de su perfil.
RF9	Ofrecer una sección de estadísticas y tendencias de uso personalizadas, siempre y cuando exista información suficiente.
RF10	Permitir la gestión de los usuarios, revisión de estadísticas del sistema y supervisión del historial de detecciones y verificaciones desde el panel de administración, incluyendo la posibilidad de eliminar cualquier registro cuando sea necesario.

Tabla 1. Requisitos Funcionales de FactGuard

Fuente: Elaboración Propia

5.1.1.2 Requisitos No Funcionales

Por su parte, se han definido una serie de requisitos no funcionales que afectan a la calidad, rendimiento, seguridad y escalabilidad de la aplicación:

Código	Descripción
RNF1	El tiempo de respuesta medio de FactGuard ante una predicción o una verificación no puede superar los cinco segundos en condiciones normales.
RNF2	La arquitectura debe ser modular y facilitar la separación de tareas y responsabilidades entre los distintos servicios ofrecidos.
RNF3	La interfaz debe ser clara, intuitiva y compatible con navegadores de escritorio, móviles y *tablets*.
RNF4	Las contraseñas deben guardarse de forma segura mediante técnicas de *hash*, y el sistema debe poder garantizar la integridad de los *tokens* JWT.
RNF5	El diseño del sistema debe hacer posible la incorporación futura de nuevos modelos, idiomas o fuentes de verificación sin afectar al núcleo funcional.

RNF6	El código debe seguir las buenas prácticas de desarrollo de *software* tales como modularidad, documentación, validación de *inputs* y gestión de errores.
RNF7	El acceso a las principales funcionalidades de la aplicación debe estar restringido solo a usuarios registrados mediante autenticación previa.
RNF8	El sistema debe validar la entrada de datos del usuario para prevenir errores en el almacenamiento y evitar inserciones duplicadas.

Tabla 2. Requisitos No Funcionales de FactGuard

Fuente: Elaboración Propia

5.1.2 CARACTERÍSTICAS RELEVANTES

FactGuard presenta una serie de características clave que distinguen el sistema de otras soluciones y que representan los principales logros técnicos y funcionales de la aplicación desarrollada. Estas funcionalidades no solo cumplen los objetivos y requisitos definidos previamente, sino que consolidan la propuesta de valor del proyecto como una plataforma robusta, práctica y orientada a la veracidad informativa.

5.1.2.1 Sistema Dual de Análisis: Detección y Verificación

FactGuard combina en una única herramienta dos enfoques complementarios para el tratamiento de la desinformación. Por un lado, la clasificación automática de noticias con el objeto de averiguar si son *fake* mediante modelos de ML y DL (FactGuard Detect); y por otro, la recuperación de afirmaciones verificadas desde fuentes externas oficiales y reconocidas (FactGuard Verify). Esta integración permite al usuario no solo detectar posibles falsedades, sino también obtener respaldo contrastado cuando existe información previa publicada.

5.1.2.2 Predicción Multiclase con Cinco Modelos Entrenados y Umbral de Confianza Configurable

FactGuard Detect no se limita a los resultados arrojados por un único modelo, sino que ejecuta de forma simultánea cinco algoritmos distintos: *Decision Tree*, *Random Forest*, *XGBoost*, LSTM y BERT. Las predicciones de cada uno de estos modelos se muestran de forma desagregada, permitiendo al usuario comparar cómo varía la evaluación según la arquitectura empleada. Además, el sistema incorpora un umbral de confianza configurable, que permite establecer un nivel mínimo de certeza para que una predicción sea considerada válida. En el caso de que ningún modelo supere el umbral establecido, FactGuard Detect mostrará la clasificación como *incierta*.

El veredicto sobre la noticia se calcula mediante una estrategia de votación ponderada entre todos los modelos que han emitido predicciones válidas. Cada uno contribuye con un peso proporcional a su rendimiento (medido mediante *F1 Score*), permitiendo reducir la dependencia de un único enfoque, obtener una mejor fiabilidad del análisis y aportar una mayor transparencia al proceso de detección.

5.1.2.3 Interfaz Accesible y Centrada en el Usuario

La aplicación presenta una interfaz clara e intuitiva, compuesta por una página de inicio (*Home*) donde se muestran los principales beneficios y funcionalidades de FactGuard, así como un acceso directo al registro o inicio de sesión. Una vez el usuario se ha autenticado, puede acceder a su perfil personal (*Profile*) organizado en distintas pestañas: *Dashboard*, *Detect Fake News*, *Verify Claims*, y *Settings*.

Cada uno de los módulos implementados ha sido diseñado para guiar al usuario de forma sencilla y estructurada en el proceso de análisis, presentando los resultados de manera detallada. Esto permite una consulta del historial personalizado y ofrece opciones de

configuración como el ajuste del umbral de confianza en FactGuard Detect o la elección del idioma para comprobación de afirmaciones en FactGuard Verify.

5.1.2.4 Gestión Segura de Usuarios. Almacenamiento Personalizado

FactGuard incorpora un sistema completo de autenticación y gestión de usuarios que garantiza tanto la seguridad como la personalización de la aplicación. El proceso de autenticación está basado en tokens JWT**,** validando cada sesión de forma segura y eficiente. Asimismo, todas las contraseñas de los usuarios se almacenan mediante técnicas de cifrado con *hash*, siguiendo buenas prácticas de seguridad.

Cada usuario dispone de su panel personal desde el que puede consultar su historial de detecciones y verificaciones, permitiéndolo revisar resultados anteriores, eliminarlos o realizar nuevas consultas. Todo ello no solo mejora la experiencia de uso de cara al usuario final, sino que facilita la trazabilidad de los datos de cada uno de ellos sin poner en riesgo la privacidad de los datos personales.

5.2 ARQUITECTURA DE FACTGUARD

La arquitectura ha sido diseñada con un enfoque modular, seguro y escalable, con el fin de garantizar una separación de responsabilidades, facilitar el mantenimiento del sistema y permitir futuras extensiones.

5.2.1 ESTRUCTURA GENERAL

FactGuard se apoya en una arquitectura de doble servidor definida en tres niveles fundamentales:

- Nivel de Presentación (*frontend*): Compuesto por la interfaz gráfica mostrada al usuario final y con la que interactúa en toda la aplicación. Esta capa se encarga de recoger las entradas (*inputs*), mostrar los resultados de los distintos análisis (*results*)

y gestionar la navegación (*navigate*) entre módulos. Está construida con tecnologías web estándar y conectada mediante peticiones HTTP a los servidores del *backend*.

- Nivel de Aplicación (*backend*): Formado por dos servidores independientes que colaboran de forma coordinada:
  - Un servidor de aplicación y base de datos, desarrollado en `Node.js`, encargado de las funciones de autenticación de usuarios, gestión de cuentas y almacenamiento de resultados en una base de datos SQLite.
  - Un servidor de procesamiento desarrollado en *Python* con `Flask`, que actúa como motor de procesamiento, ejecutando los modelos de ML/DL para la detección de *fake news* y de realizar las verificaciones externas a través de la API pública de *Google FactCheck Claim Search*.
- Nivel de Datos: Gestionado a través de SQLite, permite almacenar localmente la información estructurada relacionada con usuarios, detecciones, verificaciones y configuración individual. Se trata de una solución simple, eficiente y fácilmente integrable en entornos de desarrollo y pruebas.

La arquitectura resultante responde a un modelo cliente-servidor distribuido, en el que la comunicación entre componentes se realiza a través de peticiones HTTP autenticadas mediante *tokens* JWT. Además de mejorar la seguridad del sistema, permite escalar cada uno de los servidores de forma independiente, facilitando un posible despliegue futuro.

5.2.2 Flujo de Datos y Comunicación

El flujo de datos en FactGuard se articula en torno a las dos grandes funcionalidades de la aplicación. A continuación, se describen los flujos de ambos módulos manera independiente.

5.2.2.1 Flujo de FactGuard Detect

El proceso se inicia cuando el usuario introduce el título y contenido de la noticia a analizar en la pestaña *Detect Fake News* desde la interfaz web. El flujo completo, expuesto en la Figura 24, consta de las siguientes etapas:

1. Validación de Entradas en el *Frontend*: Antes de enviar la solicitud, se comprueba que los campos obligatorios no se encuentren vacíos. En caso de que falte alguno de ellos, se mostrará un mensaje de error correspondiente al código `` `HTTP 403 Bad Request` `` al pulsar el botón de detección.

2. Envío de Datos al Servidor de Modelos: En caso de que los campos de entrada sean válidos, el *frontend* envía una solicitud POST al *endpoint* `` `/predict` `` del servidor `` `Flask` ``. Dicha solicitud incluye el título y el contenido de la noticia, así como el umbral de confianza ajustado por el usuario.

3. Autenticación en el Servidor `` `Flask` ``: Al recibir la solicitud en el *endpoint*, el servidor comprueba la presencia y la validez del *token* de autorización en la cabecera. Si la cabecera no está presente, se rechaza la solicitud con un código `` `HTTP 403 Forbidden` ``. Si el token no se encuentra, es inválido o ha expirado, la respuesta emitida es `` `HTTP 401 Unauthorized` ``, indicando un problema de autenticación.

4. Preprocesamiento del Texto: Una vez recibida la petición y extraídos los campos del cuerpo de esta, el servidor ejecuta una función encargada de realizar el preprocesamiento del texto, incluyendo técnicas de limpieza léxica, tokenización, lemmatización y eliminación de ruido. Esto se realiza para normalizar el contenido y transformarlo en una entrada adecuada para los distintos modelos.

5. Vectorización y Tokenización: El texto preprocesado se adapta al formato requerido por cada uno de los modelos empleados en el sistema:

- Para los modelos de ML (*Decision Tree*, *Random Forest* y *XGBoost*), se transforma el texto en vectores numéricos mediante la técnica TF-IDF.

- Para el modelo LSTM, se realiza la tokenización del texto y la aplicación de *padding* para ajustar todas las secuencias a una longitud uniforme.

- Para el modelo BERT, se utiliza un *tokenizer* preentrenado que genera los tensores de entrada adecuados.

6. Ejecución de Modelos: Una vez preparadas las entradas, se ejecutan los cinco modelos integrados. Cada uno de ellos procesa los datos de forma independiente y devuelve las probabilidades de que el contenido sea calificado como verdadero o falso, generando una predicción individual.

7. Aplicación del Umbral de Confianza: Tras obtener las predicciones, se comparan las probabilidades asignadas a las clases y se identifica la de mayor valor. Si dicha probabilidad no supera el umbral de confianza definido por el usuario, la predicción de ese modelo se marca como incierta.

8. Agregación de Resultados: Para llegar al resultado final, se aplica una estrategia de votación ponderada en función del *F1-score* que cada modelo ha obtenido durante la fase de validación, otorgando mayor peso a aquellos modelos con mejor rendimiento. Después, se suma la confianza de cada predicción para cada clase (*True*, *Fake* o *Uncertain*), y se selecciona la tenga mayor puntuación total tenga como el resultado final del análisis.

9. Envío de Resultados al *Frontend*: El servidor `Flask` devuelve una respuesta que incluye el veredicto global, las predicciones individuales de cada modelo, sus respectivas probabilidades y los niveles de confianza.

10. Comprobación de la Respuesta del Servidor: Al recibir la respuesta, el *frontend* lleva a cabo una verificación del estado de la petición. En el caso de que los resultados no estén correctamente estructurados o se encuentren incompletos, el servidor responde con un mensaje de error con código `HTTP 400 Bad Request`, informando al usuario de que la detección no ha podido completarse correctamente.

11. Almacenamiento en Base de Datos: Tras recibir y validar los resultados de la detección, el *frontend* vuelve a realizar otra solicitud POST, pero en esta ocasión al servidor de `Node.js` a través del *endpoint* `/detections`. Esta petición incluye todos los datos generados por el análisis, así como el *token* JWT del usuario autenticado. Tal y como ocurre con el servidor `Flask`, se realizan todas las verificaciones pertinentes. En caso de que la cabecera no se esté presente o haya problemas con el *token*, se devuelve la solicitud con los mismos códigos de error.

12. Verificación de Duplicados: El servidor consulta la base de datos para comprobar si ya existe una entrada con el mismo contenido (título y texto) previamente registrada por el usuario en cuestión. Si se identifica una detección idéntica, se interrumpe la operación y se devuelve un código `HTTP 409 Conflict`.

13. Inserción y Redirección a la Vista de Resultados: En caso de no detectarse duplicados, se procede a insertar la nueva detección en la base de datos. Si se produce algún error durante la escritura, el servidor responde con un código `HTTP 500 Internal Server Error`. Si la operación se realiza correctamente, el *frontend* redirige al usuario a la página de resultados, donde podrá consultar el análisis almacenado recientemente.

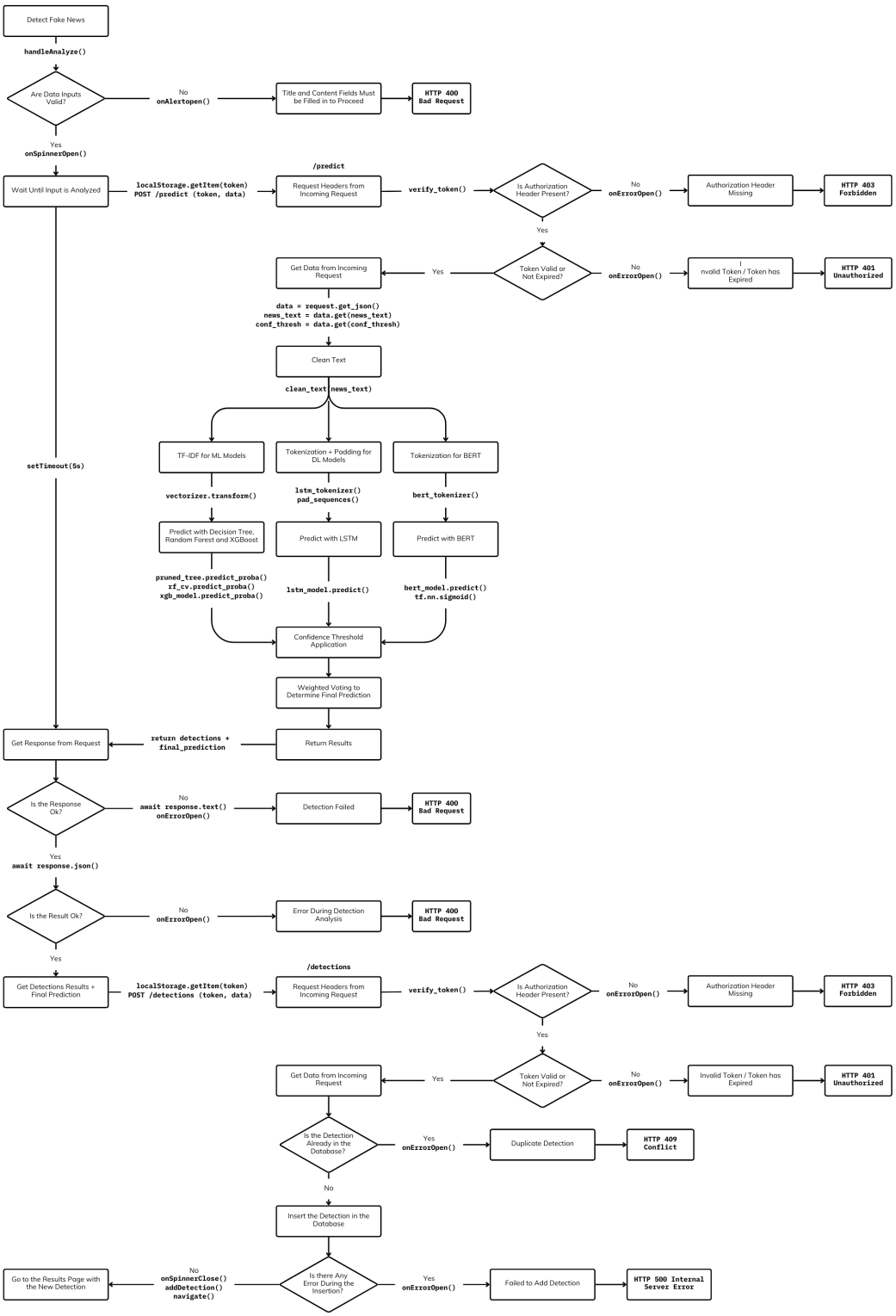


Figura 24. Flujo de Ejecución de FactGuard Detect con Gestión de Errores

Fuente: Elaboración Propia

5.2.2.2 Flujo de FactGuard Verify

Similarmente, el proceso se inicia cuando el usuario selecciona el idioma e introduce el contenido de la afirmación a verificar en la pestaña *Verify Claims* desde la interfaz web. El flujo completo, expuesto en la Figura 25, consta de las siguientes etapas:

1. Validación de Entradas en el *Frontend*: Antes de enviar la solicitud, se comprueba que los campos obligatorios no se encuentren vacíos. En caso de que falte alguno de ellos, se mostrará un mensaje de error correspondiente al código `` `HTTP 403 Bad Request` `` al pulsar el botón de verificación.

2. Envío de Datos al Servidor de Verificación: En caso de que los campos de entrada sean válidos, el *frontend* envía una solicitud POST al *endpoint* `` `/factcheck` `` del servidor `` `Flask` ``. Dicha solicitud incluye el contenido de la consulta y el idioma seleccionado por el usuario.

3. Autenticación en el Servidor `` `Flask` ``: Al recibir la solicitud en el *endpoint*, el servidor comprueba la presencia y la validez del *token* de autorización en la cabecera. Si la cabecera no está presente, se rechaza la solicitud con un código `` `HTTP 403 Forbidden` ``. Si el token no se encuentra, es inválido o ha expirado, la respuesta emitida es `` `HTTP 401 Unauthorized` ``, indicando un problema de autenticación.

4. Comunicación con *Google FactCheck Tools API*: Una vez recibida la petición y extraídos los campos del cuerpo de esta, el servidor genera una solicitud externa a la API de verificación de afirmaciones de Google (`` `factchecktools.googleapis.com` ``). Para ello, se emplean como parámetros de entrada la consulta introducida por el usuario y el idioma seleccionado.

5. Procesamiento de la Respuesta: El servidor `` `Flask` `` devuelve una respuesta que incluye los textos de las afirmaciones verificadas, la valoración emitida, el medio

verificador (*publisher*) y el enlace correspondiente. Esta información se envía al cliente como una respuesta estructurada.

6. Comprobación de la Respuesta del Servidor: Al recibir la respuesta, el *frontend* lleva a cabo una verificación del estado de la petición. En el caso de que los resultados no estén correctamente estructurados o se encuentren incompletos, el servidor responde con un mensaje de error con código `` `HTTP 400 Bad Request` ``. Este mensaje informa al usuario de que la verificación no ha podido completarse correctamente o que no existen coincidencias válidas con la consulta realizada.

7. Almacenamiento en Base de Datos: Tras recibir y validar los resultados de la verificación, el *frontend* vuelve a realizar otra solicitud POST, pero en esta ocasión al servidor de `` `Node.js` `` a través del *endpoint* `` `/claims` ``. Esta petición incluye todos los datos generados por el análisis, así como el *token* JWT del usuario autenticado. Tal y como ocurre con el servidor `` `Flask` ``, se realizan todas las verificaciones pertinentes. En caso de que la cabecera no se esté presente o haya problemas con el *token*, se devuelve la solicitud con los mismos códigos de error.

8. Verificación de Duplicados: El servidor consulta la base de datos para comprobar si ya existe una entrada con el mismo contenido previamente registrada por el usuario en cuestión. Si se identifica una verificación idéntica, se interrumpe la operación y se devuelve un código `` `HTTP 409 Conflict` ``.

9. Inserción y Redirección a la Vista de Resultados: En caso de no detectarse duplicados, se procede a insertar la nueva verificación en la base de datos. Si se produce algún error durante la escritura, el servidor responde con un código `` `HTTP 500 Internal Server Error` ``. Si la operación se realiza correctamente, el *frontend* redirige al usuario a la página de resultados, donde podrá consultar el análisis almacenado recientemente.

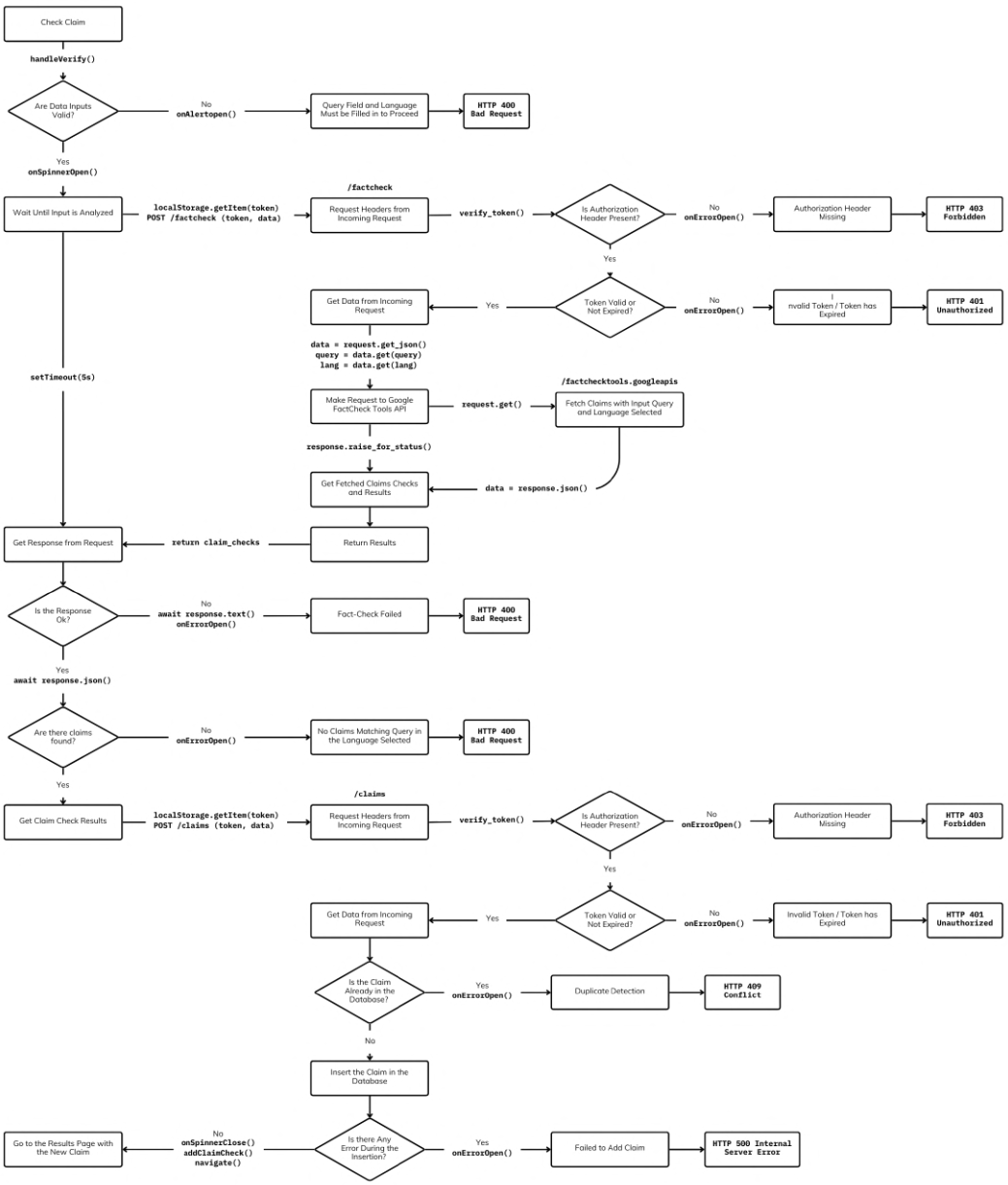


Figura 25. Flujo de Ejecución de FactGuard Verify con Gestión de Errores

Fuente: Elaboración Propia

5.2.2.3 Resumen de Flujos de Ejecución

Para facilitar la comprensión global del funcionamiento de ambos módulos, se presenta la Tabla 3 que resume los pasos principales del flujo de ejecución tanto en *FactGuard*

Detect como en *FactGuard Verify*. Esta visión permite identificar con claridad las similitudes y diferencias entre ambos procesos.

N.º	Etapa	FactGuard Detect	FactGuard Verify
1	**Validación de Entradas**	Verifica que el título y contenido no estén vacíos.	Verifica que existan el campo de idioma y la consulta.
2	**Envío de Datos al Servidor `Flask`**	Se envía una solicitud POST al *endpoint* `/predict`	Se envía una solicitud POST al *endpoint* `/factcheck`.
3	**Autenticación** (Servidor `Flask`)	Comprueba cabecera y *token* JWT. Devuelve `403` o `401` si no es válido.	Mismo proceso: Cabecera y *token*. Devuelve `403` o `401` si falla.
4	**Procesamiento Interno**	Ejecuta cinco modelos (DT, RF, XGB, LSTM, BERT) y limpia el texto.	Realiza una llamada a la API de *Google FactCheck* con el idioma y la consulta.
5	**Aplicación de Lógica Interna**	Aplica umbral de confianza y votación ponderada para llegar al veredicto.	Procesa y estructura las afirmaciones devueltas por la API.
6	**Comprobación de Respuesta** (*frontend*)	Verifica estructura y campos de la respuesta. Si falla, muestra `400`.	Igual: Valida el formato o ausencia de resultados. Si falla, `400`.
7	**Envío de Resultados al *Backend* `Node.js`**	POST al *endpoint* `/detections` con los resultados y el *token*.	POST al *endpoint* `/claims` con los resultados y el *token*.
8	**Autenticación** (Servidor `Node.js`)	Verifica el *token*. Devuelve `403` o `401` si no es válido.	Mismo proceso: Cabecera y *token*. Devuelve `403` o `401` si falla.
9	**Verificación de Duplicados**	Revisa si ya existe una detección igual (título + contenido). En ese caso, devuelve `409`.	Comprueba si ya existe la misma consulta del usuario. Si existe, devuelve `409`.
10	**Inserción** (SQLite) **y Redirección a Resultados**	Inserta en la base de datos. Si hay error, `500`. Si no, se redirige a vista de resultados.	Igual: Inserta en base de datos. Si falla, `500`. Si no, se redirige a vista de resultados.

Tabla 3. Flujo de Ejecución de FactGuard

Fuente: Elaboración Propia

5.3 DESARROLLO DEL MÓDULO DE DETECCIÓN DE FAKE NEWS (FACTGUARD DETECT)

5.3.1 PREPROCESAMIENTO DE DATOS

Para garantizar que los modelos de detección trabajen con información limpia, estructurada y relevante es necesario realizar un procesamiento previo del texto. Este proceso se desarrolló en dos fases principales. Por un lado, un EDA (*Exploratory Data Analysis*) para seleccionar las variables más importantes y comprender la naturaleza del conjunto de datos y; por otro, una transformación del texto para preparar las entradas que se deben emplear en el entrenamiento de los distintos modelos.

5.3.1.1 Análisis Exploratorio Inicial

Como se comentó previamente en la Sección 2.3 del Capítulo 2. , el conjunto de datos utilizado es el *ISOT Fake News Dataset*, el cual está distribuido en dos archivos independientes según la veracidad de las noticias. Cada entrada incluye información clasificada en las siguientes columnas: `title`, `text`, `subject` y `date`. Para unificar el conjunto y facilitar su procesamiento, ambos archivos fueron combinados en uno único, añadiendo la columna `target` como variable objetivo, donde el valor `0` indica que la noticia es falsa, y `1` que es verdadera.

Aunque el *dataset* proporciona tanto el título como el cuerpo completo de las noticias (`text`), se optó por trabajar únicamente con este último ya que el contenido completo del artículo ofrecía una mayor profundidad semántica, lo que facilitaría la detección de patrones asociados a *fake news*. Por otro lado, el resto de campos (`subject` y `date`) tampoco fueron considerados relevantes para esta tarea y fueron descartados durante el preprocesamiento.

Durante esta fase también se llevó a cabo un primer proceso de limpieza básica del cuerpo de las noticias. Por ejemplo, las noticias verdaderas incluían encabezados con nombres de ciudades entre paréntesis que señalaban la ubicación del reportaje, pero como no aportaban valor informativo tuvieron que eliminarse. También se quitaron espacios iniciales/finales en blanco, signos de puntuación, caracteres especiales y se aplicó normalización a minúsculas para reducir la dimensionalidad del vocabulario.

Posteriormente, el texto fue tokenizado, es decir, segmentado en unidades léxicas básicas (*tokens*), generalmente palabras, para facilitar su análisis. A continuación, se aplicó un proceso de lemmatización, mediante el cual cada *token* fue reducido a su forma canónica o lema (por ejemplo, "*running*", "*ran*" y "*runs*" pasan todos a ser "*run*"), con el objetivo de reducir la variabilidad morfológica del lenguaje sin perder el significado original. Asimismo, se eliminaron las palabras vacías (*stopwords*), términos muy frecuentes pero que no aportan información discriminativa para la tarea de clasificación. En este grupo de palabras se suelen incluir artículos, pronombres, algunos verbos auxiliares y preposiciones.

Todo ello se traduce en la función `clean_text()`, mostrada en el Listado 2, que permitió simplificar el vocabulario y centrar el análisis en las palabras con mayor carga semántica.

```
def clean_text(text):
    # Remove extra whitespaces
    text = re.sub(r'\s+', ' ', text, flags=re.I)

    # Remove special characters
    text = re.sub(r'\W', ' ', str(text))

    # Remove single characters
    text = re.sub(r'\s+[a-zA-Z]\s+', ' ', text)

    # Remove not alphabetical characters
    text = re.sub(r'[^a-zA-Z\s]', ' ', text)

    # Convert to lowercase
    text = text.lower()
```

```
    # Lemmatization
    tokens = word_tokenize(text)
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in
stop_words]

    # Removal of Stop Words
    tokens = [word for word in tokens if len(word) > 3]

    return tokens
```

Listado 2. Función De Limpieza Léxica utilizada en el Preprocesamiento

Fuente: Elaboración Propia

Tras aplicar esta función de limpieza, se realizó una representación gráfica de la distribución y composición del conjunto de datos, con el objeto de poder identificar posibles patrones antes del entrenamiento de los modelos.

En primer lugar, tal y como se muestra en los gráficos superiores de la Figura 26, el *dataset* presenta una distribución relativamente equilibrada entre noticias reales y falsas, con una ligera mayoría de las últimas (52,5 %). Esta proporción es importante para el entrenamiento de modelos supervisados, ya que evita el gran problema del desequilibrio de clases, que podría sesgar los resultados del clasificador.

Asimismo, se analizó la distribución temática del contenido. En el caso de las *fake news*, los temas más comunes están relacionados con categorías genéricas como *News* y *Politics*, seguidos por etiquetas ideológicas (*left-news*) y referencias institucionales (*Government News* o *US_News*). En comparación, las noticias verdaderas presentan una clasificación más estructurada, centrada principalmente en *PoliticsNews* y *WorldNews*. Esta diferencia en las temáticas entre noticias reales y *fake* sugiere que ciertos contextos o ámbitos podrían estar más expuestos a la desinformación, lo que refuerza la necesidad de un sistema que pueda detectar patrones más allá de simples palabras clave.

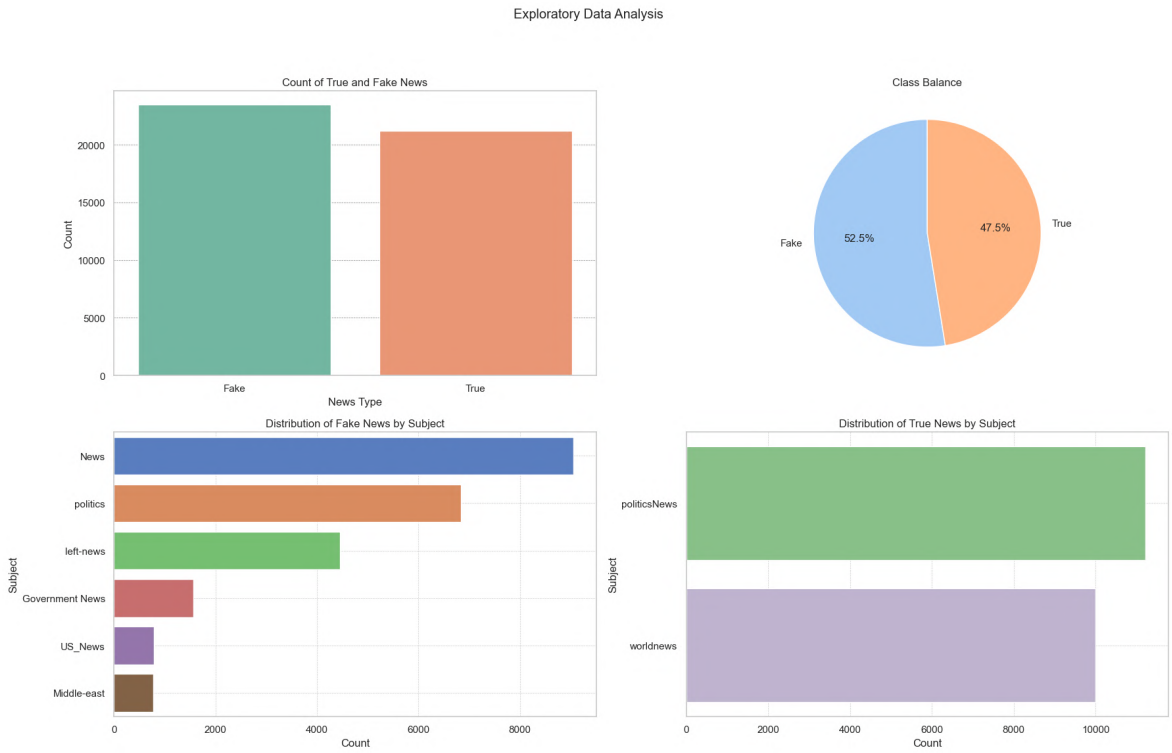


Figura 26. Análisis Exploratorio Inicial del Conjunto de Datos

Fuente: Elaboración Propia

5.3.1.2 División de Conjuntos y Vectorización

Una vez realizado todo el preprocesamiento requerido, se procedió a dividir el conjunto de datos en grupos específicos para cada tipo de modelo. En todos los casos se aplicó una división estratificada que mantuviera la proporción de clases original.

Para los modelos de ML tradicionales, solo se realizó una partición del 80 % para entrenamiento y 20 % para *test*. Sin embargo, para los modelos basados en DL (LSTM y BERT), también se hizo una segunda partición sobre el conjunto de entrenamiento para obtener un subconjunto de validación. Esta separación permite evaluar la capacidad de generalización de los modelos y su rendimiento durante el entrenamiento, lo que facilita el ajuste de hiperparámetros y evita el *overfitting*. Este proceso se exhibe en el Listado 3.

```
# Split data for ML, LSTM and BERT
X = data['text_cleaned']
y = data['target']
X_bert = data['text_cleaned'].to_numpy()
y_bert = data['target'].to_numpy().astype('int32')

# Split for ML models
X_train_ml, X_test_ml, y_train_ml, y_test_ml = train_test_split(
    X, y, test_size = 0.2, stratify = y, random_state = 42
)

# Split for LSTM model
X_train_dl, X_test_dl, y_train_dl, y_test_dl = train_test_split(
    texts_lstm, y, test_size = 0.2, stratify = y, random_state = 42
)
X_train_dl, X_valid_dl, y_train_dl, y_valid_dl = train_test_split(
    X_train_dl, y, test_size = 0.2, stratify = y_train_dl, random_state = 42
)

# Split for BERT model
X_train_bert, X_test_bert, y_train_bert, y_test_bert = train_test_split(
    X_bert, y_bert, test_size = 0.2, stratify = y_bert, random_state = 42
)
X_train_bert, X_valid_bert, y_train_bert, y_valid_bert = train_test_split(
    X_train_bert, y_train_bert, test_size = 0.2, stratify = y_train_bert,
random_state = 42
)
```

Listado 3. División del Conjunto de Datos para Modelos ML, LSTM y BERT

Fuente: Elaboración Propia

Para representar los textos como vectores numéricos adecuados para cada arquitectura, se aplicaron diferentes técnicas de vectorización según el tipo de modelo. En el caso de los clasificadores tradicionales (*Decision Tree*, *Random Forest* y *XGBoost*), se utilizó TF-IDF para reducir el impacto de las palabras más frecuentes y resaltar aquellas que aportan más información.

Por su parte, como el modelo LSTM requiere como entrada secuencias de números enteros, se utilizó una tokenización basada en índices de palabras (*word index*), y un proceso de *padding* para unificar la longitud de todas las secuencias al tamaño máximo permitido. Ello facilita el entrenamiento del modelo al permitir procesar las muestras en forma de lotes

(*batches*). En el caso de BERT, se empleó un tokenizador preentrenado de la arquitectura `bert-base-uncased`, que genera estructuras específicas requeridas por el modelo, como los *inputs IDs* y las *attention masks*. Además, se estableció un *padding* y truncado automáticos para que cada entrada respetara las restricciones de longitud impuestas por el *transformer*.

Este proceso, reflejado en el Listado 4, garantizó que cada modelo recibiera las entradas en el formato esperado, respetando las especificaciones de cada arquitectura.

```
# TF-IDF Vectorization for ML models
vectorizer = TfidfVectorizer(max_features = 5000) # Limit the number of features
for faster training
X_train_ml = vectorizer.fit_transform(X_train_ml)
X_test_ml = vectorizer.transform(X_test_ml)

# Tokenization and Padding for LSTM
lstm_tokenizer = Tokenizer()
lstm_tokenizer.fit_on_texts(X_train_dl)
word_idx = lstm_tokenizer.word_index
X_toke_train = lstm_tokenizer.texts_to_sequences(X_train_dl)
X_toke_valid = lstm_tokenizer.texts_to_sequences(X_valid_dl)
X_toke_test = lstm_tokenizer.texts_to_sequences(X_test_dl)

max_sequence_length = 150 # Define max sequence length for padding
size = len(word_idx) # Define the vocabulary size
X_train_dl = pad_sequences(X_toke_train, maxlen = max_sequence_length)
X_valid_dl = pad_sequences(X_toke_valid, maxlen = max_sequence_length)
X_test_dl = pad_sequences(X_toke_test, maxlen = max_sequence_length)

# Tokenization for BERT
bert_name = "bert-base-uncased"
bert_tokenizer = AutoTokenizer.from_pretrained(bert_name, padding = "max_length",
do_lower_case = True)

X_tokenized_bert_train = bert_tokenizer(X_train_bert.tolist(), padding = True,
truncation = True, return_tensors = "tf").input_ids
X_tokenized_bert_valid = bert_tokenizer(X_valid_bert.tolist(), padding = True,
truncation = True, return_tensors = "tf").input_ids
X_tokenized_bert_test = bert_tokenizer(X_test_bert.tolist(), padding = True,
truncation = True, return_tensors = "tf").input_ids
```

Listado 4. Vectorización y Preparación de Entradas para Modelos ML, LSTM y BERT

Fuente: Elaboración Propia

5.3.2 ARQUITECTURA DE LOS MODELOS EMPLEADOS

En esta sección se describen las arquitecturas seleccionadas para los diferentes modelos de clasificación utilizados en el módulo de FactGuard Detect. Aunque la descripción y el funcionamiento de estos algoritmos ya fue introducido en el Capítulo 2. , aquí se detalla la manera en la que se han integrado específicamente dentro del flujo del proyecto y cómo se han adaptado para el tratamiento del texto tras el preprocesamiento realizado previamente.

Se abordan tanto los clasificadores tradicionales, como aquellos basados en aprendizaje profundo, destacando sus configuraciones particulares, capas empleadas y parámetros relevantes.

5.3.2.1 Decision Tree Classifier

La arquitectura implementada para el modelo de árbol de decisión corresponde a un clasificador que utiliza el algoritmo CART, el cual parte del texto ya vectorizado y opera sobre un conjunto de hasta 5.000 características. Para ello, se utilizó la implementación de `DecisionTreeClassifier` de la librería `scikit-learn` con la especificación de dos parámetros importantes:

- `class_weight='balanced'`: Para mitigar posibles sesgos en la predicción que puedan darse debido al ligero desequilibrio de clases observado en el EDA.
- `criterion='gini'`: Para medir la calidad de una partición en cada nodo del árbol, favoreciendo las divisiones que generen subconjuntos con una mayor pureza.

La arquitectura inicial del modelo, junto con la configuración de los parámetros mencionados, se muestra en el Listado 5.

```
# CART Architecture
cart_model = DecisionTreeClassifier(
            max_depth = 10,
            criterion = 'gini',
```

```
            class_weight = 'balanced',
            random_state = 42
          )
```

Listado 5. Arquitectura Inicial de CART

Fuente: Elaboración Propia

5.3.2.2 Random Forest

El modelo de *Random Forest* se realizó con el clasificador `RandomForestClassifier`, también de la biblioteca `scikit-learn`, utilizando como entrada el conjunto de datos vectorizado mediante TF-IDF. Al igual que anteriormente, se estableció el parámetro que permite el uso de una configuración balanceada (`class_weight='balanced'`) para compensar las pequeñas diferencias que existen en el número de observaciones por clase. Además, se habilitó otro parámetro propio de las arquitecturas *ensemble* basadas en *bagging*:

- `oob_score=True`: Para obtener una estimación del rendimiento del modelo durante el entrenamiento, sin necesidad de reservar una parte del conjunto para validación.

Nuevamente, la arquitectura inicial de este clasificador, junto con la configuración de los parámetros mencionados, se muestra en el Listado 6.

```
# Random Forest Architecture
rf_model = RandomForestClassifier(
            n_estimators = None,   # To be set via GridSearchCV
            criterion    = None,   # To be set via GridSearchCV
            max_depth    = None,   # To be set via GridSearchCV
            max_features = None,   # To be set via GridSearchCV
            oob_score    = True,
            n_jobs       = -1,
            class_weight = 'balanced',
            random_state = 42
          )
```

Listado 6. Arquitectura Inicial de Random Forest

Fuente: Elaboración Propia

5.3.2.3 XGBoost

Para el modelo de XGBoost se utilizó la clase `XGBClassifier` de la librería `xgboost`. Como en los modelos anteriores, la entrada está constituida por el conjunto de datos vectorizado mediante TF-IDF con 5.000 características. La arquitectura se diseñó atendiendo a dos nuevos parámetros:

- `objective = 'binary:logistic'`: Para definir la función de pérdida logística.
- `scale_pos_weight = negative_instances/positive_instances`: Para atender a la ligera descompensación entre clases. Este parámetro ajusta la penalización de los errores cometidos sobre la clase minoritaria.

La configuración de estos parámetros que definen la arquitectura inicial del modelo se observa en el Listado 7.

```
# XGBoost Architecture
positive_instances = sum(y_train_ml == 1)
negative_instances = sum(y_train_ml == 0)
ratio = negative_instances / positive_instances

xgb = xgb.XGBClassifier(
            objective = 'binary:logistic',
            scale_pos_weight = negative_instances/positive_instances
    )
```

Listado 7. Arquitectura Inicial de XGBoost

Fuente: Elaboración Propia

5.3.2.4 LSTM

En el caso del modelo LSTM, se emplearon las bibliotecas de `keras` y `tensorflow` para definir una arquitectura flexible y modular que se adaptara al procesamiento de secuencias textuales. Ahora, la entrada estaba formada por secuencias de longitud fija, establecidas en 150 *tokens*. Este valor fue definido tras observar la distribución de longitudes

en los textos de las noticias, con el objetivo de retener la mayor de información relevante sin comprometer el rendimiento ni introducir ruido por *padding* excesivo.

A continuación, se detallan todas las capas incorporadas en la arquitectura del modelo, expuesta en el Listado 8.

```
# Model Parameters
maxlen = 150
embed_size = 100

# LSTM Architecture
input = Input(shape=(maxlen,))
learning_rate = 0.0001
x = Embedding(size + 1, embed_size)(input)
x = Dropout(0.5)(x)
x = LSTM(150,return_sequences=True)(x)
x = Dropout(0.5)(x)
x = GlobalMaxPooling1D()(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(2, activation='softmax')(x)

lstm_model = Model(input, x)
```

Listado 8. Arquitectura Inicial de LSTM

Fuente: Elaboración Propia

Las secuencias de entrada se transformaron en representaciones densas mediante una capa `Embedding` de dimensión 100, convirtiendo cada índice de palabra en un vector denso para su entrenamiento. Aunque dicho valor es inferior a la longitud máxima de las secuencias, se seleccionó para mantener un equilibrio entre capacidad de representación semántica y eficiencia computacional. A continuación, se aplicó una capa `Dropout` con una tasa del 50%, para reducir el riesgo de *overfitting* durante el entrenamiento. Como esta capa desactiva aleatoriamente una parte de las unidades en cada iteración, se fuerza al modelo a no depender de patrones específicos, permitiendo obtener una mejor generalización.

La arquitectura sigue con una capa `LSTM` de 150 unidades, diseñada para capturar dependencias a largo plazo dentro de la secuencia textual. Como esta capa se configuró con el parámetro `return_sequences=True`, se guarda la información de cada paso temporal y se transmite a la siguiente capa, permitiendo representar estructuras lingüísticas más complejas. Posteriormente, una operación de `GlobalMaxPooling1D` reduce la dimensionalidad de la salida `LSTM`, capturando la señal más relevante sin perder información significativa.

Por último, se añadió una capa `Dense` de 64 unidades con una función de activación `ReLU`, que actúa como un transformador no lineal antes de la etapa de salida. Al ser una tarea de clasificación binaria, la capa `Dense` de salida debe estar compuesta por dos neuronas y activación `softmax` para dar como resultado la probabilidad de pertenencia a cada una de las clases (noticia verdadera o falsa).

5.3.2.5 BERT

Para el último enfoque se eligió al modelo BERT, cargado a través de la librería `transformers` de *Hugging Face*. Concretamente, se utilizó la clase `TFAutoModelForSequenceClassification`, configurada para una tarea de clasificación binaria mediante el parámetro `num_labels = 2`.

```
# BERT Architecture
bert_model = TFAutoModelForSequenceClassification.from_pretrained(bert_name,
num_labels = 2)
```

Listado 9. Arquitectura Inicial de BERT

Fuente: Elaboración Propia

Como se confirma con el Listado 9, no fue necesario construir manualmente una arquitectura de red neuronal ni introducir modificaciones sobre las capas internas del modelo. Más bien, se realizó un proceso de *fine-tuning* en el que se reutilizó el tokenizador

preentrenado así como su estructura original, ajustando únicamente los pesos a las características del conjunto de datos empleado. Dada la gran precisión de este modelo en datos textuales, no eran necesarios cambios estructurales para obtener un buen rendimiento.

5.3.3 ENTRENAMIENTO Y AJUSTE DE HIPERPARÁMETROS

Una vez definidos los modelos, el siguiente paso es el entrenamiento de los mismos, optimizando su rendimiento mediante técnicas de ajuste de hiperparámetros. Para los modelos tradicionales, se empleó `GridSearchCV` para identificar la combinación óptima de valores y maximizar el rendimiento de los modelos. Por su parte, en los de aprendizaje profundo se definieron manualmente parámetros como el número de épocas, tamaño del *batch* o función de pérdida.

5.3.3.1 Decision Tree Classifier

Inicialmente, se definió una búsqueda que incluía distintos valores para la profundidad máxima del árbol (`max_depth`), número mínimo de muestras por partición (`min_samples_split`) y ganancia mínima de impureza (`min_impurity_decrease`). A partir de los resultados arrojados por esta exploración inicial, se ajustaron los intervalos de estos valores en base a los resultados obtenidos, ejecutando una segunda búsqueda más precisa mostrada en el Listado 10.

```
# Adapted grid based on result from initial grid search
param_grid = {
    'max_depth': [10, 12, 15, 18],
    'min_samples_split': [5, 10, 15, 20],
    'min_impurity_decrease': [0.00005, 0.00008, 0.0001, 0.0003, 0.0005],
}
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state = 42), param_grid,
cv = 10, n_jobs = -1)
gridSearch.fit(X_train_ml, y_train_ml)
```

Listado 10. Búsqueda de Hiperparámetros mediante `GridSearchCV` sobre CART

Fuente: Elaboración Propia

Como resultado de esta nueva búsqueda, se obtuvo una puntuación de validación cruzada de 0.9339, con los siguientes hiperparámetros óptimos:

- `max_depth = 18`
- `min_samples_split = 10`
- `min_impurity_decrease = 5e-05`

Esta combinación permitió construir un árbol lo suficientemente profundo para capturar patrones relevantes, sin incurrir en *overfitting*.

5.3.3.2 Random Forest

En este caso, para optimizar el rendimiento del modelo, se combinó la búsqueda con validación cruzada con una estrategia de validación `RepeatedKFold` (5 particiones, 3 repeticiones). La combinación de hiperparámetros, que incluía, entre otros, el número de árboles (`n_estimators`) y el número máximo de características consideradas en cada división (`max_features`), se muestra en el Listado 11. Esta configuración permitió explorar de forma robusta el espacio de hiperparámetros y reducir la varianza en los resultados.

```
# Parameter grid for CV
param_grid = {'n_estimators': [20, 25, 50, 100, 150],
              'max_features': [5, 10, 15, 20, 25],
              'max_depth'   : [1, 5, 10, 15, 20],
              'criterion'   : ['gini', 'entropy']
             }

# Grid search by CV
grid = GridSearchCV(
        estimator  = RandomForestClassifier(random_state = 42, class_weight =
'balanced'),
        param_grid = param_grid,
        scoring    = 'accuracy',
        n_jobs     = - 1,
        cv         = RepeatedKFold(n_splits = 5, n_repeats = 3, random_state =
42),
        refit      = True,
        verbose    = 0,
```

```
        return_train_score = True
       )

grid.fit(X = X_train_ml, y = y_train_ml)
```

Listado 11. Búsqueda de Hiperparámetros mediante `GridSearchCV` y `RepeatedKFold` sobre Random Forest

Fuente: Elaboración Propia

La mejor configuración encontrada obtuvo una precisión del 96.8% durante la validación cruzada, mejorando notablemente respecto al árbol de decisión CART:

- `'criterion': 'entropy'`
- `'max_depth': 20`
- `'max_features': 25`
- `'n_estimators': 150`

Una vez seleccionada la mejor combinación, el modelo se entrenó de forma definitiva sobre el todo el conjunto de entrenamiento.

5.3.3.3 XGBoost

En la estructura final de *XGBoost* se evaluaron distintas combinaciones de hiperparámetros clave como la tasa de aprendizaje (`eta`), la profundidad máxima de los árboles (`max_depth`), el peso mínimo de los nodos hoja (`min_child_weight`), la regularización mediante `gamma`, y la fracción de columnas utilizada por árbol (`colsample_bytree`). El proceso de ajuste, reflejado en el Listado 12, empleó una partición de 3 pliegues, y el criterio de optimización fue el valor medio (`mean_test_score`).

```
# Parameter grid for CV
parameters = {
    "eta": [0.15, 0.2, 0.25],
```

```
    "max_depth": [6, 7, 8],
    "min_child_weight": [1, 2],
    "gamma": [0.1, 0.2],
    "colsample_bytree": [0.3, 0.5],
    "n_estimators": [30, 50]
}

grid = GridSearchCV(xgb, parameters, n_jobs = 4, cv = 3)
grid.fit(X = X_train_ml, y = y_train_ml)
```

Listado 12. Búsqueda de Hiperparámetros mediante `GridSearchCV` sobre XGBoost

Fuente: Elaboración Propia

Esta combinación óptima de parámetros consiguió una precisión media de validación del 97.84%, mejorando aún más la alcanzada por *Random Forest*. Las mejores condiciones se obtuvieron con los siguientes valores:

- `'colsample_bytree': 0.3`
- `'eta': 0.25`
- `'gamma': 0.1`
- `'max_depth': 8`
- `'min_child_weight': 2`
- `'n_estimators': 50`

Nuevamente, con estos hiperparámetros seleccionados, se procedió a entrenar el modelo final sobre el conjunto completo de entrenamiento.

5.3.3.4 LSTM

El modelo fue entrenado utilizando el optimizador `Adam`, con una tasa de aprendizaje inicial de 0.0001 y función de pérdida `categorical_crossentropy`, pues se trata de la adecuada para problemas de clasificación con salidas codificadas en `one-hot`. El

entrenamiento de la red se realizó en 15 épocas sobre el conjunto de entrenamiento y validado de forma continua con el conjunto de validación separado. Ello se muestra en el Listado 13.

```
# Target preparation
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train_dl)
y_valid_encoded = label_encoder.fit_transform(y_valid_dl)
y_test_encoded = label_encoder.transform(y_test_dl)

y_train_one_hot = tf.keras.utils.to_categorical(y_train_encoded)
y_valid_one_hot = tf.keras.utils.to_categorical(y_valid_encoded)
y_test_one_hot = tf.keras.utils.to_categorical(y_test_encoded)

# Optimizer and loss function
optimizer = Adam(learning_rate = learning_rate)
lstm_model.compile(optimizer = optimizer, loss = 'categorical_crossentropy',
metrics = ['accuracy'])

# Train the model
lstm_history = lstm_model.fit(X_train_dl, y_train_one_hot, epochs = 15,
validation_data=(X_valid_dl, y_valid_one_hot))
```

Listado 13. Preparación de Etiquetas y Entrenamiento de LSTM

Fuente: Elaboración Propia

Durante las primeras épocas, el modelo mostró una mejora significativa tanto en la precisión como en la pérdida. A partir de la sexta época, el rendimiento se estabilizó, logrando una precisión de validación por encima del 97 % en varias iteraciones. Esta evolución, representada en la Figura 27, evidencia una clara convergencia en la fase de entrenamiento y una diferencia mínima entre las métricas de entrenamiento y validación. Esto sugiere un gran ajuste del modelo, sin signos aparentes de *overfitting* a pesar del aumento progresivo en la precisión.

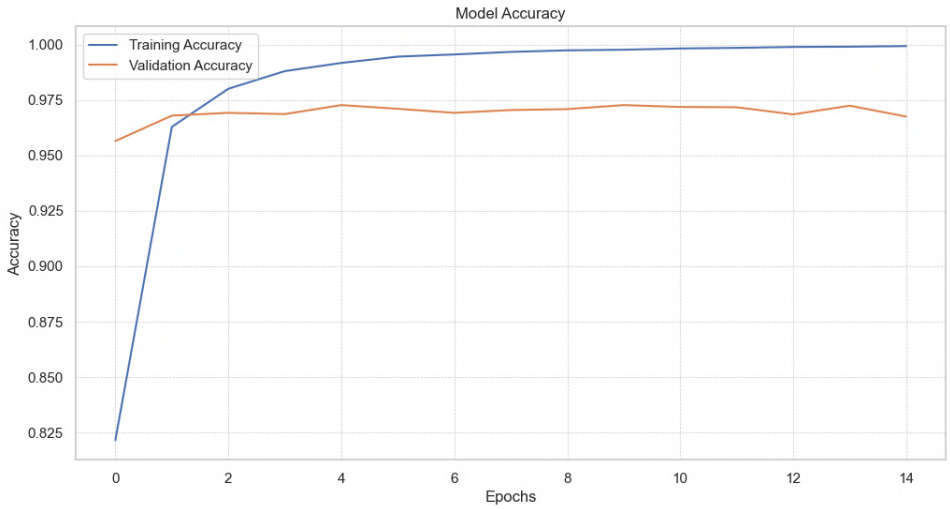


Figura 27. Evolución de la Precisión de LSTM durante Entrenamiento y Validación

Fuente: Elaboración Propia

En cuanto a la función de pérdida de la Figura 28, se aprecia un descenso progresivo en los datos de entrenamiento, alcanzando valores prácticamente nulos. La pérdida en validación, aunque presenta ligeras oscilaciones durante las cinco últimas épocas, se mantiene en niveles similares, corroborando el ajuste adecuado del modelo.

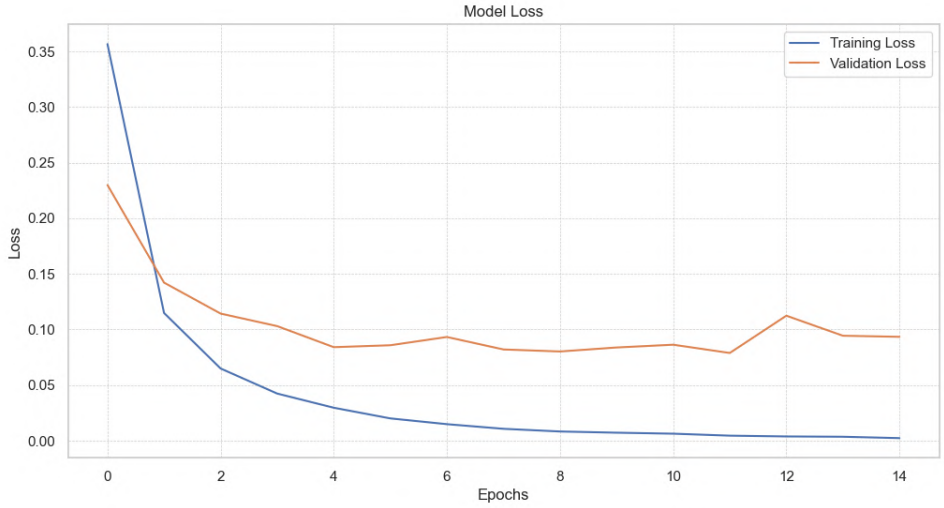


Figura 28. Evolución de la Función de Pérdida de LSTM

Fuente: Elaboración Propia

5.3.3.5 BERT

Para la fase de entrenamiento, se empleó el optimizador `AdamWeightDecay` con una tasa de aprendizaje de 2e-5 y un coeficiente de decaimiento de pesos (`weight_decay_rate`) de 0.01, valores habitualmente eficaces en *fine-tuning* sobre modelos preentrenados. La función de pérdida seleccionada fue `SparseCategoricalCrossentropy (from_logits=True)`, dada la configuración del modelo con dos neuronas de salida para clasificación binaria. Como métrica principal se utilizó la `accuracy`, reflejando el porcentaje de predicciones acertadas sobre el conjunto de validación. La configuración de estos parámetros se muestra con más detalle en el Listado 14.

A diferencia del resto de algoritmos empleados, el entrenamiento de BERT se realizó durante sólo tres épocas, debido a la gran cantidad de tiempo que supone la ejecución de modelos como este. No obstante, ello no impidió que el rendimiento alcanzara niveles prácticamente excelentes.

```
bert_model.compile(
    optimizer = AdamWeightDecay(learning_rate = 2e-5, weight_decay_rate = 0.01),
    loss = SparseCategoricalCrossentropy(from_logits = True),
    metrics = ["accuracy"]
)

bert_history = bert_model.fit(
    train_ds,
    validation_data = valid_ds,
    epochs = 3,
    batch_size = 16
)
```

Listado 14. Fine-tuning y Entrenamiento de BERT

Fuente: Elaboración Propia

Durante las tres épocas que comprenden esta etapa, BERT mostró un comportamiento progresivo en términos de precisión. En la primera de ellas se alcanzó una *accuracy* del 95.78 %, valor que se vio incrementado hasta un 99.13% en la segunda época. La precisión

en validación tampoco se queda corta, siguiendo una tendencia ascendente al lograr un 99.40 % en la última época. Tal y como se observa en la Figura 29, las curvas de entrenamiento y validación sugieren un ajuste robusto y eficaz.

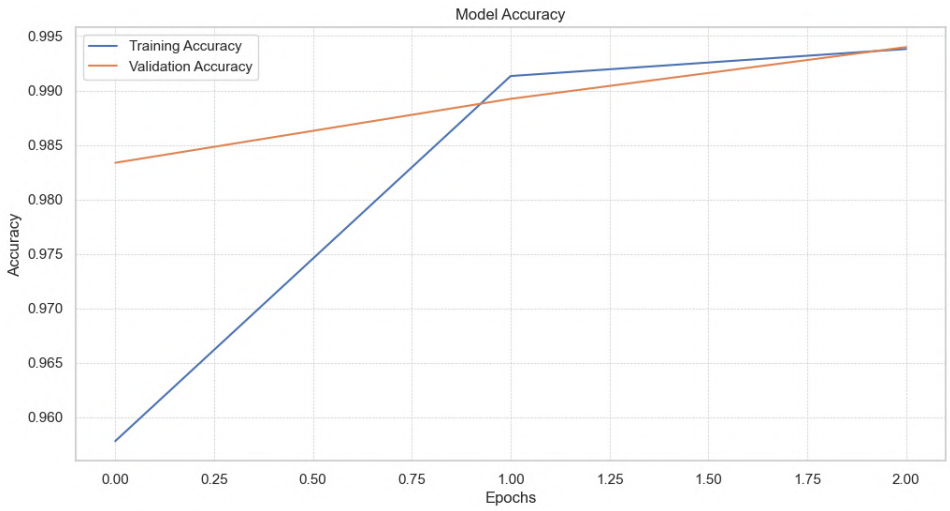


Figura 29. Evolución de la Precisión de BERT durante Entrenamiento y Validación

Fuente: Elaboración Propia

En cuanto a la función de pérdida, representada en la Figura 30, se nota una clara reducción en los valores del conjunto de entrenamiento, que pasan de 0.1078 en la primera época a 0.0194 al final del entrenamiento. Asimismo, en validación la pérdida se mantiene baja y estable a lo largo de las tres épocas, con un valor final de 0.0228. Este comportamiento verifica la solidez del modelo y su capacidad de generalización con un número reducido de épocas, debido en gran parte a la utilización de una arquitectura preentrenada.

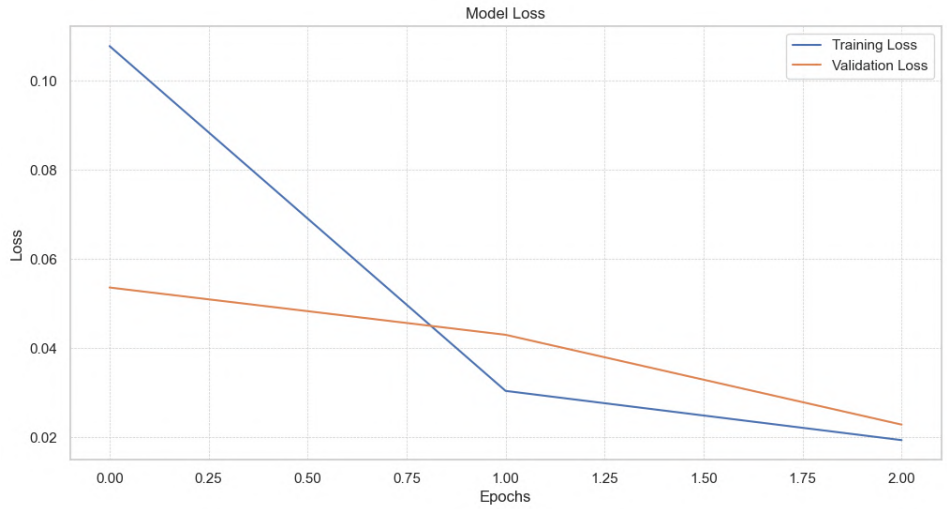


Figura 30. Evolución de la Función de Pérdida de BERT

Fuente: Elaboración Propia

5.3.4 Evaluación de la Precisión Obtenida

Una vez entrenados todos los modelos con los hiperparámetros óptimos, se evaluó el rendimiento de cada uno de ellos tanto en el conjunto de entrenamiento como en el de *test*. Las métricas utilizadas incluyeron la matriz de confusión y el informe de clasificación, los cuales proporcionan una visión completa del comportamiento del modelo frente a ambas clases.

5.3.4.1 Decision Tree Classifier

En el conjunto de prueba, la matriz de confusión de la Figura 31 reveló que el modelo fue capaz de clasificar correctamente 4.355 verdaderos negativos y 4.029 verdaderos positivos, mientras que los errores se concentraron en 341 falsos positivos y 213 falsos negativos. Ello corresponde con una precisión del 93.8%, lo que representa una tasa de error mínima e indica el buen equilibrio del modelo entre sensibilidad y especificidad.

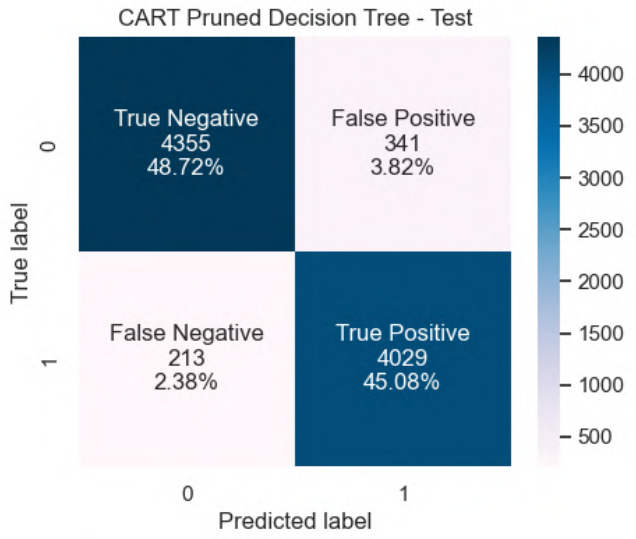


Figura 31. Matriz de Confusión en Test de CART

Fuente: Elaboración Propia

Asimismo, el informe de clasificación refuerza esta interpretación, dado que el árbol podado logró una exactitud global del 94%, con un comportamiento similar en ambas clases. Según la Tabla 4, el modelo obtuvo una *precision* del 95% para la clase 0 (*fake news*) y del 92% para la clase 1 (noticias reales), mientras que el *recall* fue del 93% y 95% respectivamente. Estos valores dan lugar a un *F1-score* de 0.94 para ambas clases, reflejando un desempeño notable para identificar tanto noticias falsas como verdaderas.

	Precision	***Recall***	***F1-Score***	***Support***
0	0.95	0.93	0.94	4696
1	0.92	0.95	0.94	4242
Accuracy			0.94	8938
Macro Avg	0.94	0.94	0.94	8938
Weighted Avg	0.94	0.94	0.94	8938

Tabla 4. Informe de Clasificación de CART

Fuente: Elaboración Propia

Los valores agregados del informe (*macro average* y *weighted average*) también marcan un *F1-score* de 0.94, lo que confirma la robustez del modelo y la ausencia de sesgo hacia una de las clases.

5.3.4.2 Random Forest

La evaluación sobre el conjunto de *test* mostró una mejora significativa en todas las métricas respecto al árbol de decisión CART. Concretamente, se identificaron correctamente 4.516 verdaderos negativos y 4.143 verdaderos positivos. Por el contrario, el modeló clasifico 180 ejemplos como falsos positivos (noticias falsas que eran realmente verdaderas), y 99 falsos negativos (*fake news* que no fueron identificadas como tal). La distribución de la Figura 32 implica una precisión del 96.88% y un error muy reducido, lo que se traduce en un rendimiento balanceado entre precisión y sensibilidad.

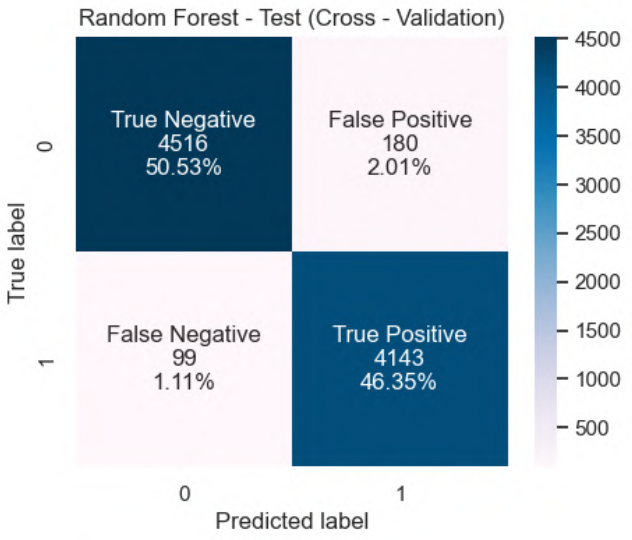


Figura 32. Matriz de Confusión en Test de Random Forest

Fuente: Elaboración Propia

Por otro lado, los resultados del informe de clasificación corroboran esta evaluación positiva. El modelo de *Random Forest* alcanzó una exactitud global del 97%, con métricas balanceadas en ambas clases. En la Tabla 5 se observa cómo la *precision* obtenida asciende

al 98% para la clase 0 y al 96% para la clase 1, mientras que el *recall* fue del 96% y 98% respectivamente. Este comportamiento también se refleja en los valores del *F1-score*, que alcanzaron un 0.97 para ambas clases, confirmando la capacidad del modelo para detectar patrones relevantes sin *overfitting* a una clase específica.

	Precision	***Recall***	***F1-Score***	***Support***
0	0.98	0.96	0.97	4696
1	0.96	0.98	0.97	4242
Accuracy			0.97	8938
Macro Avg	0.97	0.97	0.97	8938
Weighted Avg	0.97	0.97	0.97	8938

Tabla 5. Informe de Clasificación de Random Forest

Fuente: Elaboración Propia

Además, los valores agregados del informe muestran un *F1-score* de 0.97, lo que recalca la robustez del clasificador incluso en presencia de ligeros desequilibrios de clase.

5.3.4.3 XGBoost

Los resultados en *test* de *XGBoost* fueron notablemente positivos. Como se observa en la matriz de confusión de la Figura 33, el modelo logró identificar correctamente 4.585 verdaderos negativos y 4.176 verdaderos positivos, mientras que los errores se limitaron solo a 111 falsos positivos y 66 falsos negativos. Ello se traduce en una tasa de precisión del 98.02%, es decir, un margen de error extremadamente reducido.

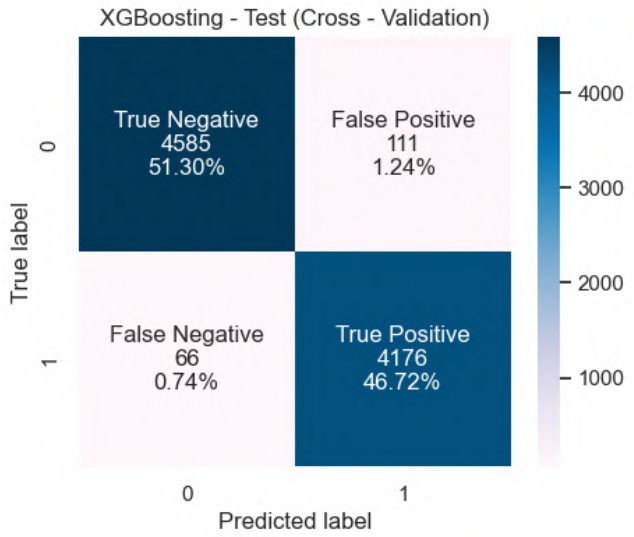


Figura 33. Matriz de Confusión en Test de XGBoost

Fuente: Elaboración Propia

El informe de clasificación de la Tabla 6 complementa este análisis con métricas detalladas para ambas clases. La clase 0 (*fake news*) alcanzó una *precision* del 99% y un *recall* del 98%, mientras que la clase 1 (noticias reales) obtuvo valores de 97% y 98% respectivamente. El *F1-score* fue de 0.98 para ambas clases, reflejando una capacidad casi perfecta para distinguir entre contenido veraz y desinformación.

	Precision	***Recall***	***F1-Score***	***Support***
0	0.99	0.98	0.98	4696
1	0.97	0.98	0.98	4242
Accuracy			0.98	8938
Macro Avg	0.98	0.98	0.98	8938
Weighted Avg	0.98	0.98	0.98	8938

Tabla 6. Informe de Clasificación de XGBoost

Fuente: Elaboración Propia

Asimismo, los promedios macro y ponderado también situaron el *F1-score* en 0.98, lo que no solo confirma el rendimiento sobresaliente del modelo, sino también su estabilidad en términos de equilibrio de clases.

5.3.4.4 LSTM

La evaluación sobre el conjunto de prueba arrojó resultados muy positivos. La matriz de confusión de la Figura 34 mostró 4.615 verdaderos negativos y 4.048 verdaderos positivos, con apenas 81 falsos positivos y 194 falsos negativos. Estos resultados se traducen en una precisión final del 96.92%, ligeramente inferior a *XGBoost*, pero consolidando a la arquitectura neuronal como uno de los modelos más sólidos del sistema.

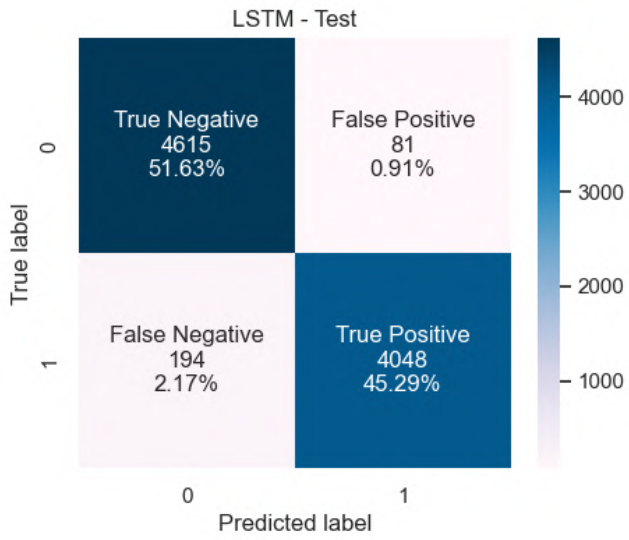


Figura 34. Matriz de Confusión en Test de LSTM

Fuente: Elaboración Propia

El informe de clasificación presentado en la Tabla 7 refuerza la solidez de LSTM. Los valores obtenidos muestran un rendimiento balanceado entre ambas clases. La clase 0 alcanzó un *recall* del 0.98 y una *precision* del 0.96, mientras que la clase 1 obtuvo una *precision* del 0.98 y un *recall* del 0.95. Ello demuestra la alta sensibilidad del modelo para detectar noticias reales e identificar correctamente las falsas.

	Precision	***Recall***	***F1-Score***	***Support***
0	0.96	0.98	0.97	4696
1	0.98	0.95	0.97	4242
Accuracy			0.97	8938
Macro Avg	0.97	0.97	0.97	8938
Weighted Avg	0.97	0.97	0.97	8938

Tabla 7. Informe de Clasificación de LSTM

Fuente: Elaboración Propia

Además, las métricas agregadas muestran una *accuracy* global del 97 %, y valores coincidentes en las medias macro y ponderada, lo que sugiere que el modelo no está sesgado hacia ninguna clase. Ello conlleva a concluir que la arquitectura LSTM mantiene una alta sensibilidad para identificar noticias reales sin sacrificar la precisión en la detección de *fake news*.

5.3.4.5 BERT

Finalmente, en lo que respecta a la evaluación de BERT en el conjunto de prueba, el rendimiento obtenido por parte de este modelo fue notablemente superior al del resto, siendo el clasificador más preciso en la detección de noticias falsas. La matriz de confusión, reflejada en la Figura 35, muestra una distribución casi perfecta, con 4.627 verdaderos negativos y 4.237 verdaderos positivos clasificados correctamente. El modelo alcanzó una precisión global del 99.17%, con una pérdida de validación final de 0.0249, lo que confirma su estabilidad y capacidad de generalización.

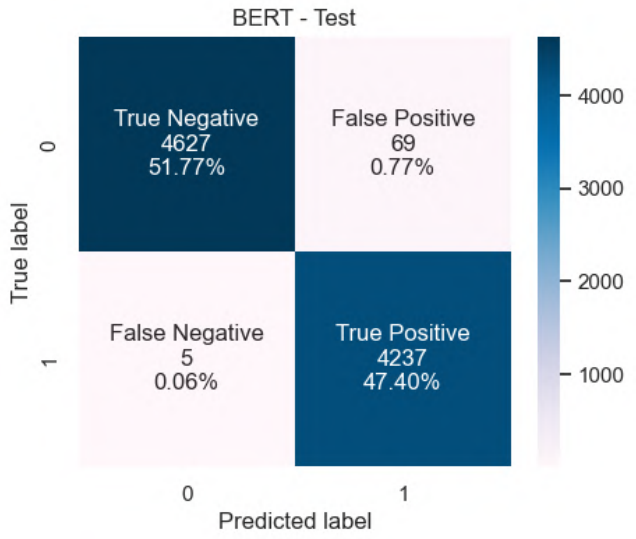


Figura 35. Matriz de Confusión en Test de BERT

Fuente: Elaboración Propia

Tal y como se recoge en la Tabla 8, BERT alcanzó un rendimiento prácticamente perfecto sobre el conjunto de prueba. La clase 0 (noticias reales) obtuvo una *precision* excepcional (1.00) y un *recall* del 0.99, mientras que la clase 1 (*fake news*) presentó un *recall* del 1.00 y una *precision* del 0.98. Esta combinación sugiere que el modelo cuida la sensibilidad sin comprometer la precisión, convirtiéndolo en una opción altamente robusta.

	Precision	***Recall***	***F1-Score***	***Support***
0	1	0.99	0.99	4696
1	0.98	1	0.99	4242
Accuracy			0.99	8938
Macro Avg	0.99	0.99	0.99	8938
Weighted Avg	0.99	0.99	0.99	8938

Tabla 8. Informe de Clasificación de BERT

Fuente: Elaboración Propia

La *accuracy* global del 99 %, junto con las medias macro y ponderada, confirma que el modelo generaliza excepcionalmente bien. Este resultado es consistente con la evolución observada en el entrenamiento y demuestra que, a pesar del bajo número de iteraciones, el modelo fue capaz de converger de manera eficaz y sin signos de *overfitting*.

5.3.5 Lógica de Inferencia y Agregación de Resultados

5.3.5.1 Votación Ponderada

Con el objetivo de evitar que el sistema dependa del criterio de un único modelo, FactGuard Detect implementa un mecanismo de votación ponderada, donde cada clasificador contribuye a la decisión final según su rendimiento obtenido. A diferencia de una votación mayoritaria simple donde todos los modelos tendrían el mismo peso en el voto, esta estrategia tiene en cuenta el *F1-score* de cada modelo como peso relativo, asignando una mayor influencia a aquellos con un equilibrio demostrado entre *precision* y *recall*.

Esta lógica se implementa en el servidor `Flask` mediante una función específica, `predict_news()`, mostrada en el Listado 15, que se encarga de coordinar todo el flujo de detección y predicción de resultados.

```
def predict_news(news_text, confidence_threshold):
    tokens = clean_text(news_text)
    cleaned_text = ' '.join(tokens)

    tfidf = vectorizer.transform([cleaned_text])
    lstm_input = pad_sequences(lstm_tokenizer.texts_to_sequences([cleaned_text]),
maxlen=150)
    bert_input = bert_tokenizer(cleaned_text, return_tensors="tf",
padding="max_length", truncation=True, max_length=512)

    dt = pruned_tree.predict_proba(tfidf)[0]
    rf = rf_cv.predict_proba(tfidf)[0]
    xgb = xgb_model.predict_proba(tfidf)[0]
    lstm = lstm_model.predict(lstm_input)[0]
    bert =
tf.nn.softmax(bert_model.predict(dict(bert_input))['logits'][0]).numpy()

    preds = {
```

```
        "Decision Tree": dt,
        "Random Forest": rf,
        "XGBoost": xgb,
        "LSTM": lstm,
        "BERT": bert,
    }

    f1_scores = {
        "Decision Tree": 0.9357,
        "Random Forest": 0.9674,
        "XGBoost": 0.9792,
        "LSTM": 0.9671,
        "BERT": 0.9936,
    }

    results, votes = [], {"True": 0, "Fake": 0, "Uncertain": 0}
    for model, probs in preds.items():
        fake, true = probs[0] * 100, probs[1] * 100
        pred = "True" if true >= fake else "Fake"
        if max(true, fake) < confidence_threshold:
            pred = "Uncertain"
        results.append({
            "Model": model,
            "True Probability": f"{true:.2f}%",
            "Fake Probability": f"{fake:.2f}%",
            "Prediction": pred
        })
        votes[pred] += f1_scores.get(model, 1.0)

    return {
        "success": True,
        "detections": results,
        "final_prediction": max(votes, key=votes.get)
    }
```

Listado 15. Función de Predicción con Votación Ponderada

Fuente: Elaboración Propia

5.3.5.2 Proceso de Clasificación

Cuando se recibe una nueva noticia para clasificar, el contenido se limpia con las mismas estrategias que las realizadas en el preprocesamiento (tokenización, lematización, y eliminación de ruido) y se adapta al formato esperado por cada modelo. Posteriormente, cada uno de estos devuelve las probabilidades que representan el grado de certeza con el

que predice si la noticia es verdadera o falsa y, en caso de no alcanzar el umbral de confianza establecido, la predicción se clasifica como *Uncertain*.

Las predicciones individuales de cada modelo se combinan para la votación, donde los votos se ponderan por el *F1-score* asociado a cada arquitectura. Esto permite que los modelos más fiables influyan con mayor fuerza en la decisión. Por último, se calcula la suma total de votos ponderados a favor de cada clase (*True*, *Fake* o *Uncertain*) y se escoge aquella que acumule la mayor puntuación. En caso de empate, se opta por el valor *Uncertain*, priorizando la prudencia en contextos ambiguos.

5.4 Desarrollo del Módulo de Fact-Checking (FactGuard Verify)

5.4.1 Integración con la API de Google FactCheck Tools

La implementación del módulo FactGuard Verify con la API de *Google FactCheck Claim Search* se ha llevado a cabo mediante una función propia dentro del servidor `Flask`, encargado del preprocesamiento de las entradas y la ejecución de los modelos descritos con anterioridad. En este contexto, el servidor ahora también actúa como intermediario entre la afirmación introducida por el usuario y los resultados obtenidos desde el *endpoint* oficial de la API de *Google*.

Esta API ofrece una funcionalidad de búsqueda de afirmaciones verificadas o *fact checks* a través del recurso `claims:search`, que permite recuperar información relacionada con revisiones hechas por entidades verificadoras que han sido acreditadas previamente. Para utilizarla, es necesario disponer de una clave (`API_key`) otorgada por *Google Cloud Console* y una serie de parámetros como el texto de la consulta (`query`) y el idioma en el que se realizará la búsqueda (`languageCode`).

La integración de este módulo se ha realizado a través de la función `search_fact_check_claims()`, expuesta en el Listado 16. El propósito de la función es realizar la gestión del envío de parámetros, el tratamiento de errores y la transformación de la respuesta en un formato estructurado. En concreto, se procesan los campos de la respuesta enviada por la API y se devuelven los resultados más relevantes en un JSON.

```
def search_fact_check_claims(API_key, query, language_code="en"):
    url = "https://factchecktools.googleapis.com/v1alpha1/claims:search"
    params = {"key": API_key, "query": query, "languageCode": language_code}

    try:
        response = requests.get(url, params=params)
        claims = response.json().get("claims", [])
        results = []

        for claim in claims:
            review = claim.get("claimReview", [{}])[0]
            results.append({
                "Claim": claim.get("text", "No text available"),
                "Claimant": claim.get("claimant", "Unknown"),
                "Publisher": review.get("publisher", {}).get("name", "Unknown"),
                "Rating": review.get("textualRating", "No rating"),
                "Date": review.get("reviewDate", "No date"),
                "URL": review.get("url", "No URL")
            })

        if results:
            return {"success": True, "data": sorted([r for r in results if
r["Date"] != "No date"], key=lambda x: x["Date"].split("T")[0],
reverse=True)[:3]}
        else:
            return {
                "success": False,
                "message": f"No claims matching '{query}' were found in
{LANGUAGE_MAP.get(language_code, language_code)}."
            }

    except requests.exceptions.RequestException as e:
        return {"success": False, "error": str(e)}
```

Listado 16. Función de Consulta a la API de Google FactCheck Claim Search

Fuente: Elaboración Propia

5.4.2 LÓGICA DE VERIFICACIÓN Y RECUPERACIÓN DE EVIDENCIA

Una vez obtenida la respuesta por parte de la API, FactGuard Verify aplica una lógica de verificación orientada a filtrar y organizar los resultados para facilitar su comprensión al usuario final. Esta lógica permite identificar aquellos *fact-checks* que resulten más relevantes, fiables y actualizados en relación con la afirmación introducida.

5.4.2.1 Tratamiento de los Resultados Devueltos por la API

Los resultados se iteran y se procesan para cada *fact-check* con el objeto de extraer el texto original de la afirmación (`text`), el posible autor de dicha afirmación (`claimant`), y la revisión más destacada asociada a ella (`claimReview`). De esta última se obtiene el nombre de la entidad verificadora (`publisher`), la calificación o veredicto emitido (`textualRating`), la fecha en la que se realizó (`reviewDate`) y el enlace a la fuente completa (`url`). En caso de que alguno de estos campos no esté presente, se informa al usuario de la no existencia de los mismos.

5.4.2.2 Criterio de Filtrado y Ordenación

Puesto que la API puede devolver múltiples verificaciones para una misma afirmación o relacionadas con temas similares, se ha optado por limitar el número de resultados mostrados a un máximo de tres, priorizando aquellas evidencias más recientes y contextualmente relevantes.

Asimismo, al tratarse de una funcionalidad conectada con un servicio externo, se ha incluido un distintivo "*Powered by Google Fact Check Tools*" para que en caso de que el usuario desee consultar más información, pueda hacerlo visitando directamente el motor de búsqueda oficial donde se muestran todos los resultados.

5.4.2.3 Gestión de Errores o Afirmaciones sin Evidencia

La función también incorpora un mecanismo de gestión de resultados negativos. En aquellos casos en los que no haya coincidencias válidas para la afirmación consultada, el sistema responderá con un mensaje indicando que no se han encontrado *fact-checks* relacionados, con un mensaje de sugerencia para reformular la afirmación o intentarlo en otro idioma.

5.4.2.4 Construcción del Objeto de Salida

Por último, los resultados seleccionados son devueltos al *frontend* en un formato JSON, donde se encuentran los elementos de la verificación descritos previamente. Esta información se presenta al usuario, permitiéndole consultar de forma transparente y detallada la evidencia disponible para la afirmación introducida.

5.5 Interfaz del Usuario (Frontend)

La interfaz de usuario o UI (*User Interface*) ha sido diseñada con el objeto de ofrecer una presentación visual coherente y accesible, implementando una navegación sencilla entre los distintos módulos. La UI permite elegir entre dos modos de visualización (modo claro y modo oscuro), que se adaptan automáticamente a la configuración del sistema definida en el dispositivo empleado, aunque el usuario también puede alternarlos manualmente según sus propias preferencias o entornos de uso.

5.5.1 Diseño

La estructura de la aplicación se organiza a partir de una página de inicio (*Home*), expuesta en la Figura 36, en la que se muestran las principales funcionalidades y beneficios de FactGuard, con una sección para captar la atención del usuario e invitarle a registrarse. El proceso de registro puede hacerse desde el botón de *Sign Up* en la parte superior derecha

o mediante el *call-to-action* destacado en color verde. Por otro lado, los usuarios que disponen de una cuenta pueden autenticarse en la aplicación pulsando el botón de *Login*.

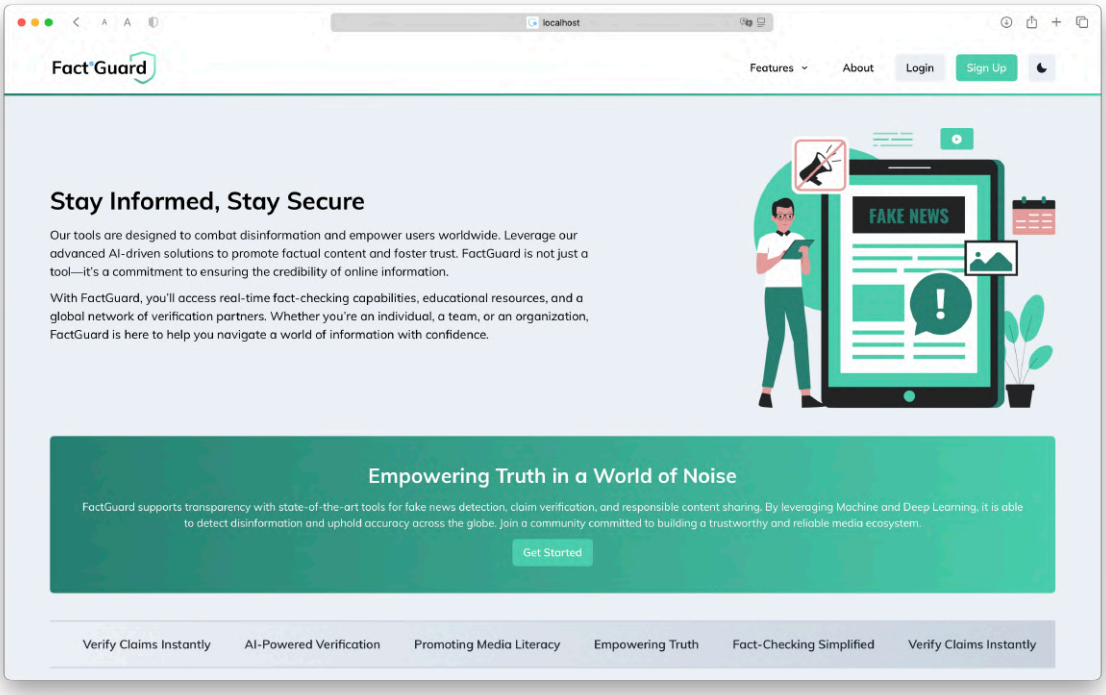


Figura 36. Interfaz de la Página de Inicio de FactGuard

Fuente: Elaboración Propia

Una vez autenticado, el usuario accede al perfil personal mostrado en la Figura 37, formado por una interfaz distribuida en distintas pestañas, a las que se puede acceder en la barra lateral o superior, según la resolución de pantalla. En el *Dashboard* principal se muestra una descripción de las funcionalidades junto con una serie de gráficos que reflejan el uso de FactGuard y las tendencias del usuario. Como se puede observar, en caso de no haber realizado ninguna interacción con la aplicación, estas estadísticas se muestran bloqueadas. No obstante, si el usuario hubiera realizado detecciones o verificaciones, se mostrarían las cinco más recientes junto con su respectivo resultado en las secciones de *Recent Detections* y *Recent Claim Checks*.

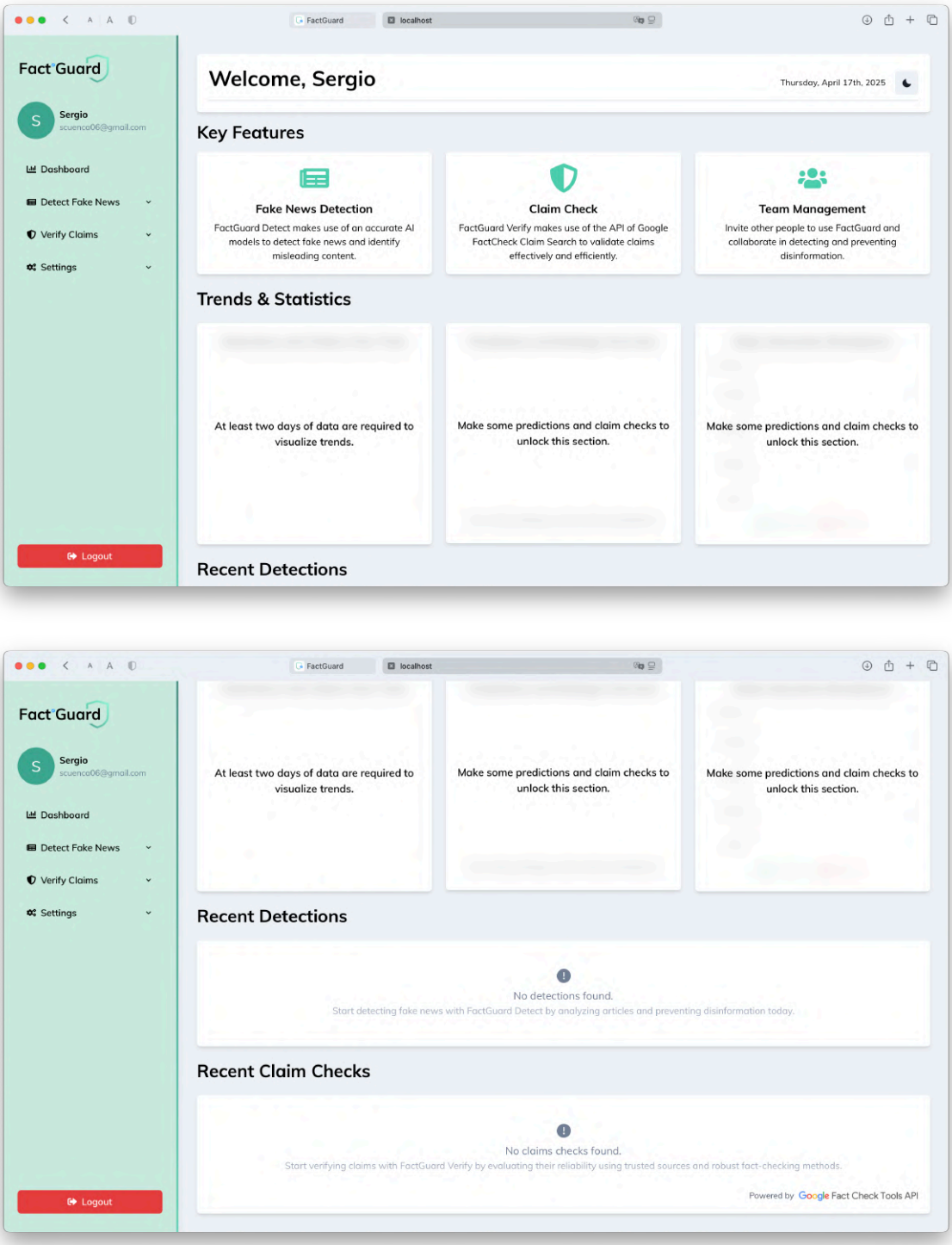


Figura 37. Interfaz del Perfil de Usuario de FactGuard (Sin Información)

Fuente: Elaboración Propia

5.5.2 FUNCIONALIDADES OFRECIDAS

5.5.2.1 Registro e Inicio de Sesión

Antes de acceder a las principales funcionalidades del sistema, el usuario debe registrarse o iniciar sesión. Para ello, se ha diseñado una interfaz de autenticación que cuenta con los campos básicos y botones que permiten completar el proceso de forma rápida. Los formularios con los que se encuentra el usuario se ilustran en la Figura 38 y en la Figura 39. Realizada la autenticación, el usuario es redirigido automáticamente al perfil privado donde puede empezar a utilizar el sistema.

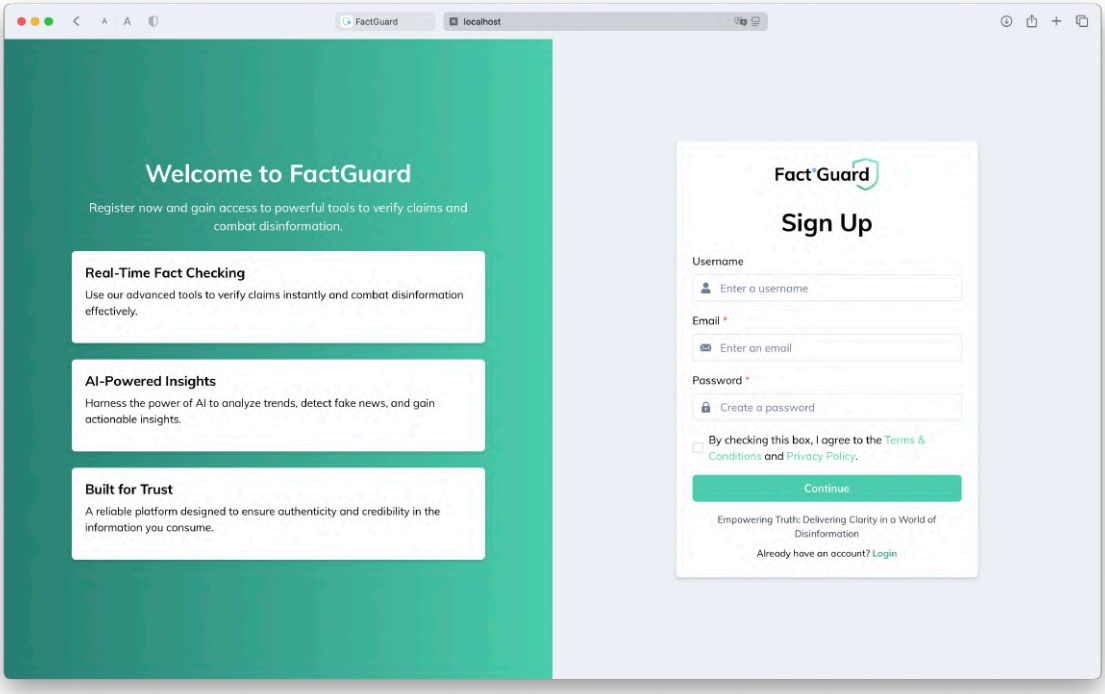


Figura 38. Interfaz de Registro de Usuario de FactGuard

Fuente: Elaboración Propia

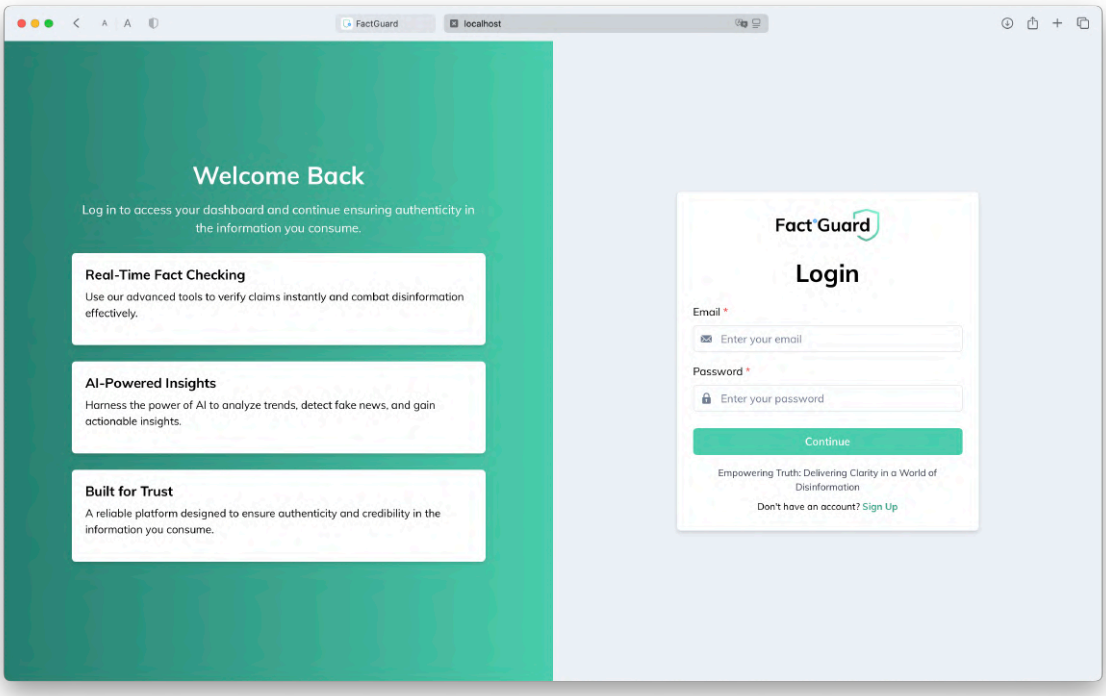


Figura 39. Interfaz de Inicio de Sesión de Usuario de FactGuard

Fuente: Elaboración Propia

5.5.2.2 Dashboard

Como se ha mencionado previamente, el panel de control ofrece una visión general del uso de FactGuard. La primera imagen de la Figura 40 corresponde a la parte superior del panel, donde destaca la sección de tendencias y estadísticas. Aquí se incluyen tres gráficos principales: un gráfico de líneas temporal de las detecciones y verificaciones realizadas, un gráfico de anillos que compara la distribución de veredictos y predicciones, y un gráfico de barras apiladas que representa la proporción diaria de interacciones.

En la segunda imagen, se observa la parte inferior del panel, donde se encuentran los bloques *Recent Detections* y *Recent Claim Checks*. En estas tablas el usuario puede revisar las cinco últimas noticias y afirmaciones que ha analizado recientemente, junto con su clasificación, fecha y opción de visualizar los resultados detallados o eliminarlos.

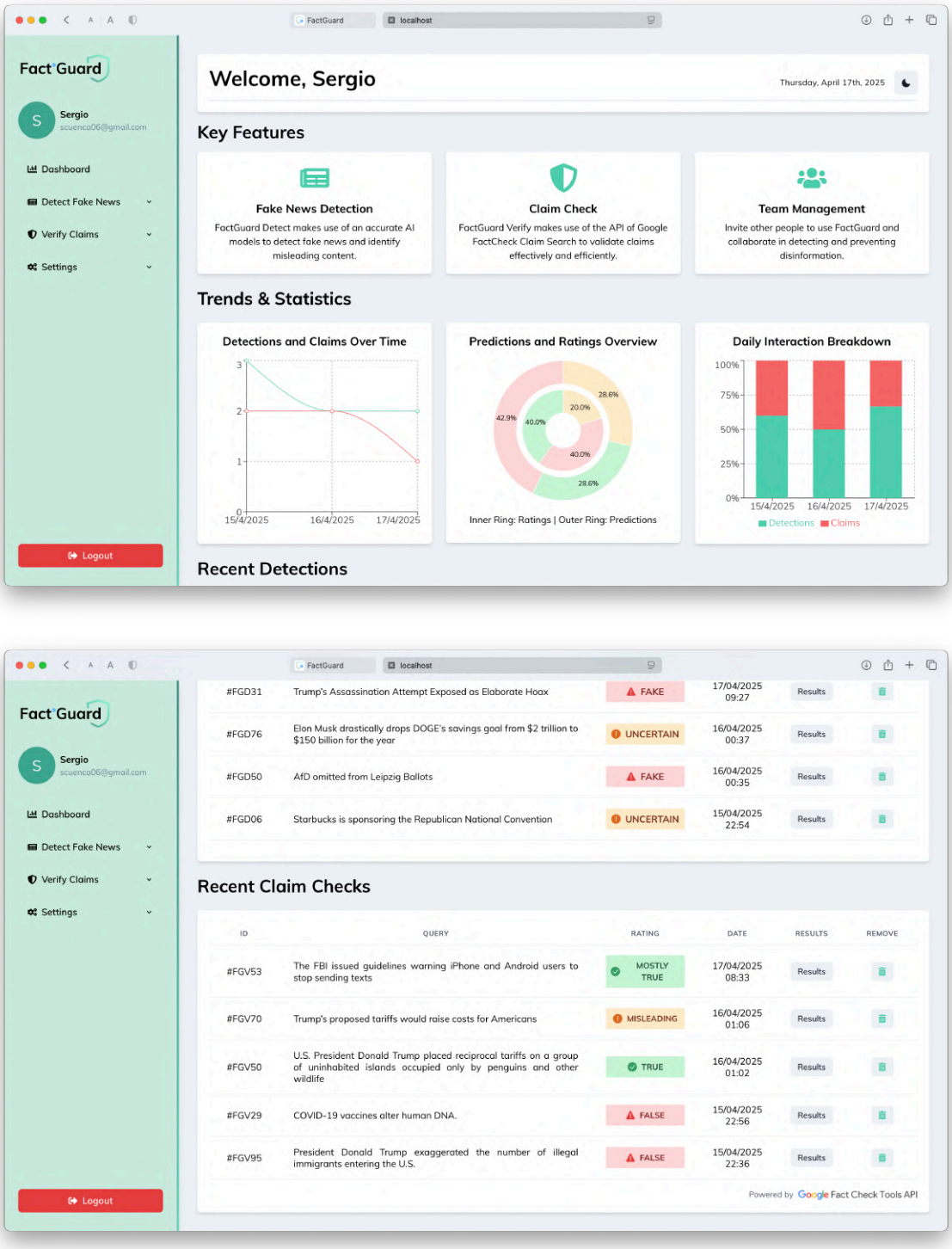


Figura 40. Interfaz del Perfil de Usuario de FactGuard

Fuente: Elaboración Propia

5.5.2.3 Detect Fake News, Detection Results & My News Detections

El análisis de *fake news* comienza en la pestaña expuesta en la Figura 41, donde se incluyen dos campos de texto para que el usuario introduzca el título del artículo y el contenido de la noticia a evaluar. Asimismo, también se muestra un control deslizante (*slider*) para que el usuario pueda ajustar el umbral de confianza aplicado en la clasificación.

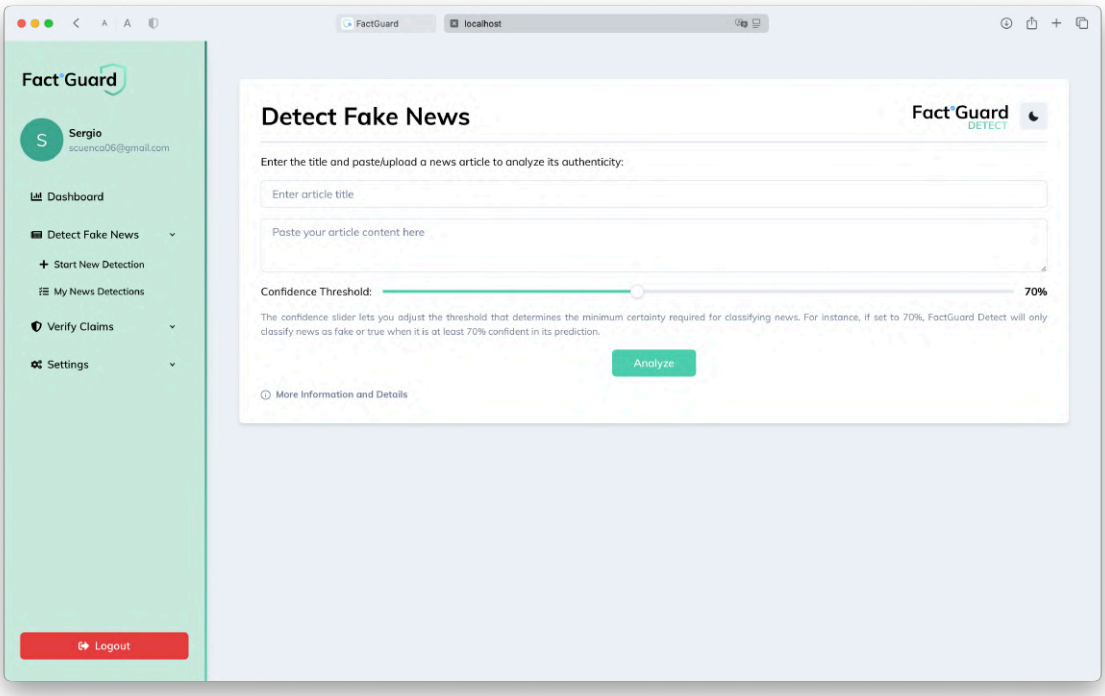
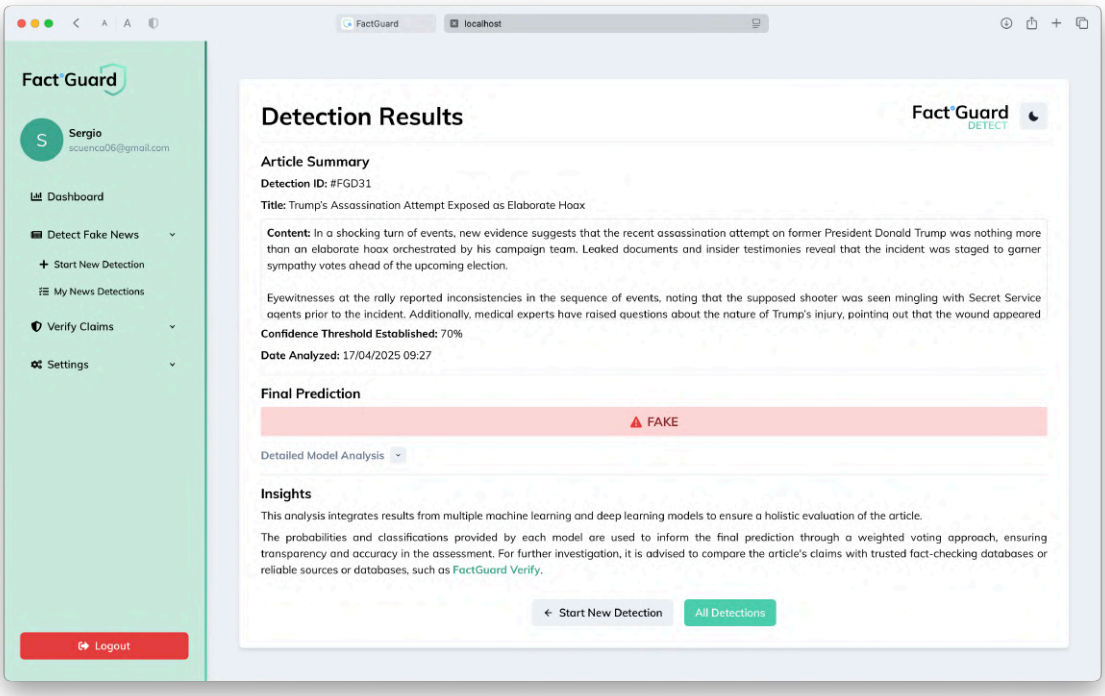


Figura 41. Interfaz del Módulo de Detección (FactGuard Detect)

Fuente: Elaboración Propia

Tras el procesamiento, FactGuard Detect redirige al usuario a la página de resultados representada en la Figura 42 que incluye toda la información del artículo analizado, así como el veredicto al que se ha llegado mediante la votación ponderada de las predicciones de todos los modelos. El resultado se presenta dentro de una tarjeta visualmente destacada con el objeto de facilitar la comprensión del resultado de forma inmediata.



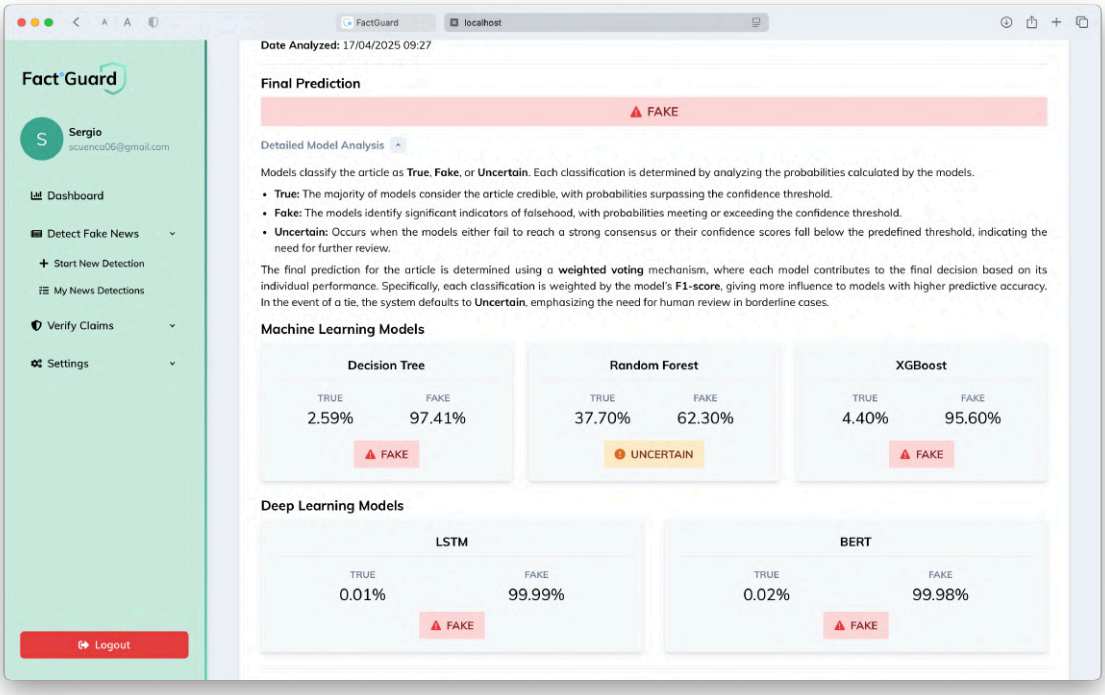


Figura 42. Interfaz de los Resultados de FactGuard Detect

Fuente: Elaboración Propia

En caso de que el usuario quiera conocer con más detalle cómo se ha llegado al resultado final, puede hacer clic en la flecha que despliega el apartado *Detailed Model Analysis*. En ella, se explica el criterio utilizado para llegar al veredicto, así como las predicciones individuales y las probabilidades asignadas por cada uno de los modelos empleados. Cada modelo se presenta en una tarjeta independiente, lo que facilita la compresión de sus aportaciones para el usuario.

Tras la visualización de los resultados, el usuario puede optar por realizar una nueva detección o consultar el historial de análisis en la pestaña *My News Detections*, expuesta en la Figura 43. Esta sección ofrece una vista resumida de todas las noticias evaluadas, junto con la predicción final asignada a cada una. El usuario puede acceder a los resultados detallados de cada análisis o eliminar aquellos que considere. También se da la opción de seleccionar múltiples noticias mediante las casillas y eliminarlas de forma conjunta.

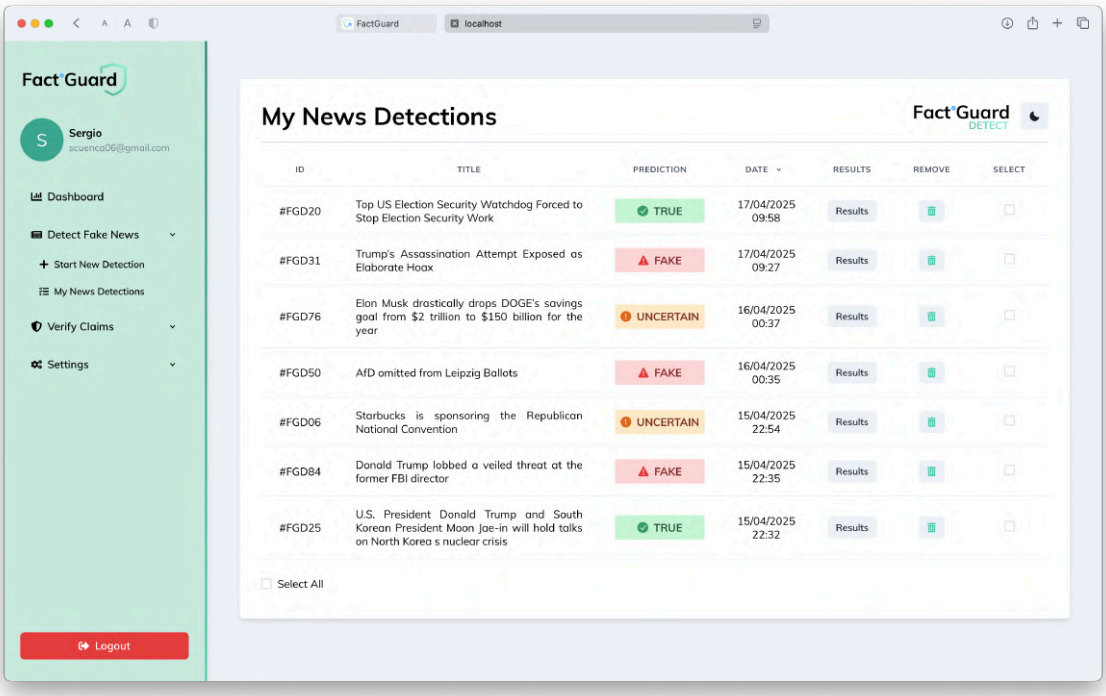


Figura 43. Interfaz de las Detecciones Realizadas con FactGuard Detect

Fuente: Elaboración Propia

5.5.2.4 Verify Claims, Claim Check Results & My Claim Checks

Similarmente, el análisis para verificar afirmaciones comienza en la pestaña expuesta en la Figura 44, donde se incluye un menú desplegable (*dropdown*) para seleccionar el idioma y un campo para introducir la afirmación que el usuario desea contrastar, ya sea una declaración política o una frase controvertida. Además, se indica que esta funcionalidad se basa en un servicio externo controlado por la API de *Google FactCheck Claim Search*.

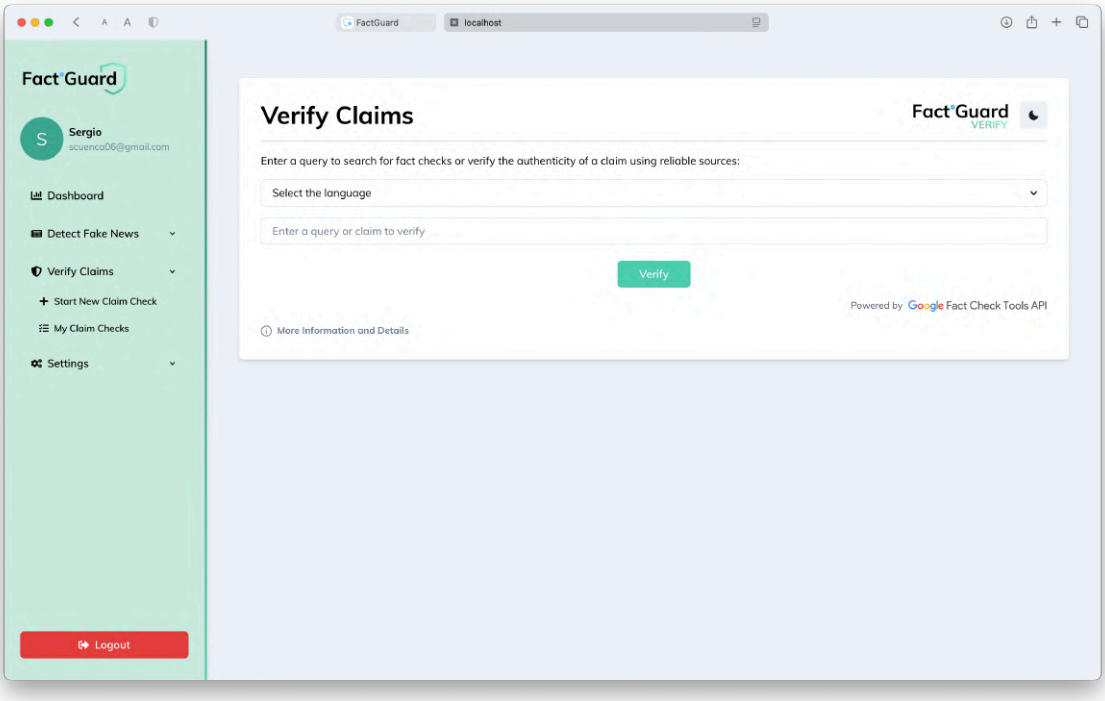


Figura 44. Interfaz del Módulo de Verificación (FactGuard Verify)

Fuente: Elaboración Propia

Tras obtener los resultados de la API, FactGuard Verify redirige al usuario a la página de resultados representada en la Figura 45 que incluye toda la información de la declaración analizada, así como los *fact-checks* encontrados. Cada una de estas evidencias se muestra en una tarjeta independiente que contiene el texto verificado, el nombre del medio verificador, el veredicto asignado y un enlace directo a la fuente oficial.

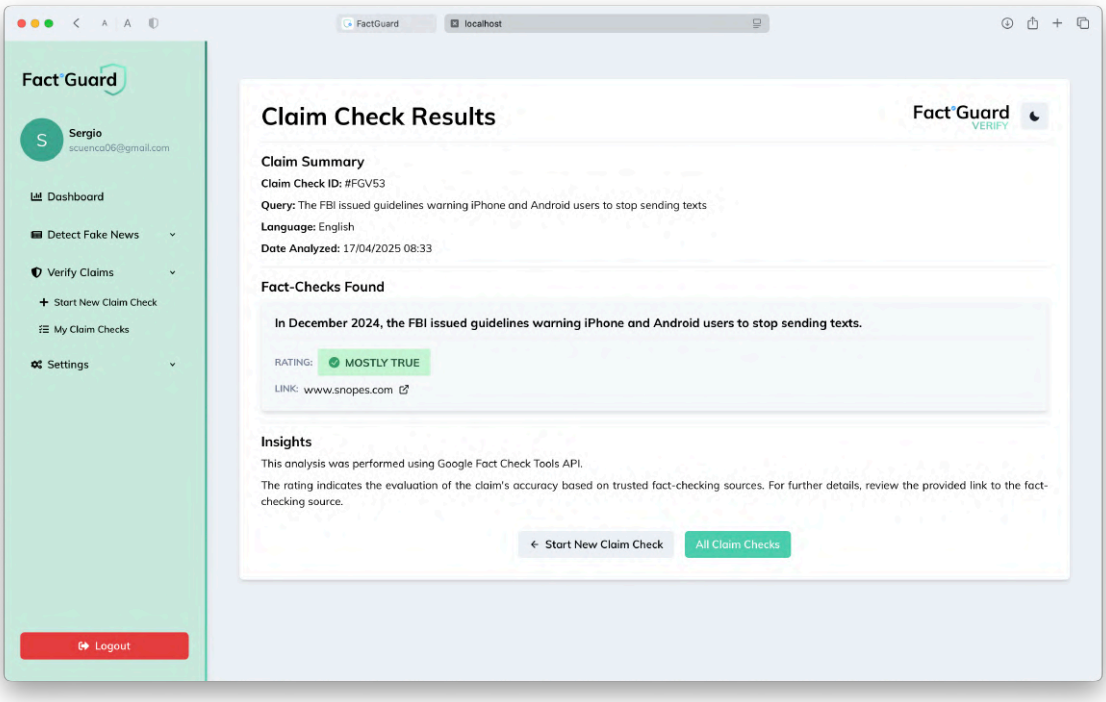


Figura 45. Interfaz de los Resultados de FactGuard Verify

Fuente: Elaboración Propia

Tras la visualización de los resultados, el usuario puede optar por realizar una nueva verificación o consultar el historial de análisis en la pestaña *My Claim Checks*, expuesta en la Figura 46Figura 43. Esta sección ofrece una vista resumida de todas las declaraciones evaluadas, junto con el *rating* asignado a cada una. El usuario puede acceder a los resultados detallados de cada análisis o eliminar aquellos que considere. También se da la opción de seleccionar múltiples declaraciones mediante las casillas y eliminarlas de forma conjunta.

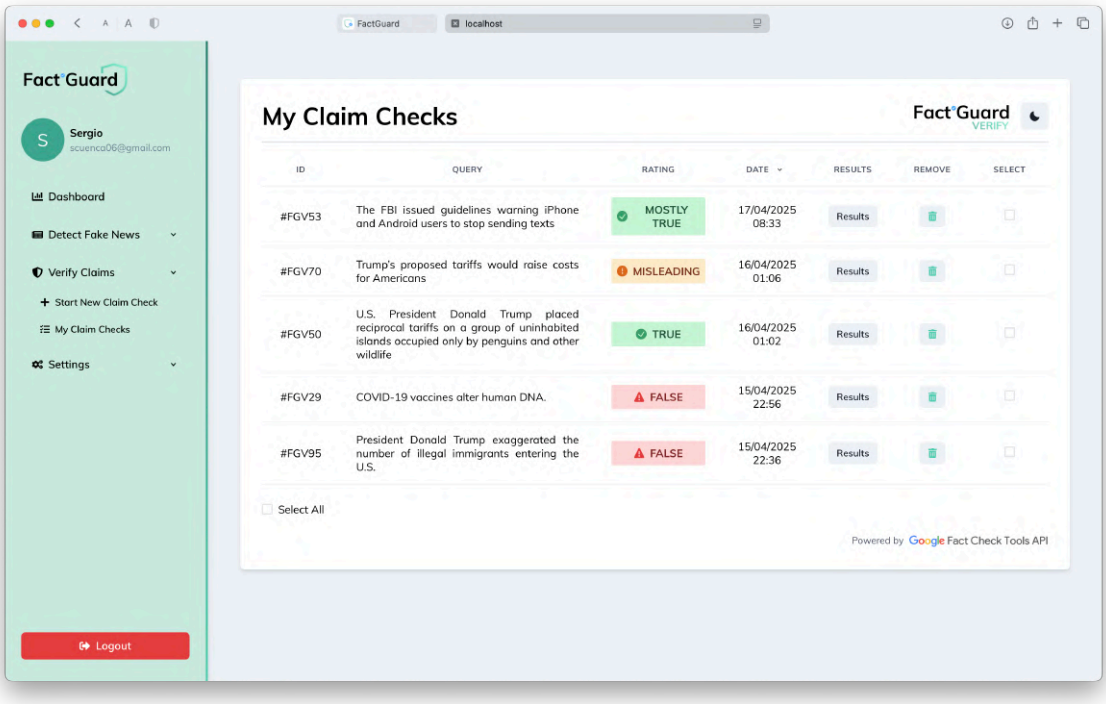


Figura 46. Interfaz de las Verificaciones Realizadas con FactGuard Verify

Fuente: Elaboración Propia

5.5.2.5 Settings

La sección de ajustes, ilustrada en la Figura 47, está destinada a la configuración y gestión del perfil personal. El usuario puede modificar sus credenciales (nombre, correo electrónico, contraseña) o eliminar su cuenta. La creación de una nueva contraseña sigue las mismas restricciones que en el registro y, además, se muestra en la interfaz el nivel de seguridad de la misma.

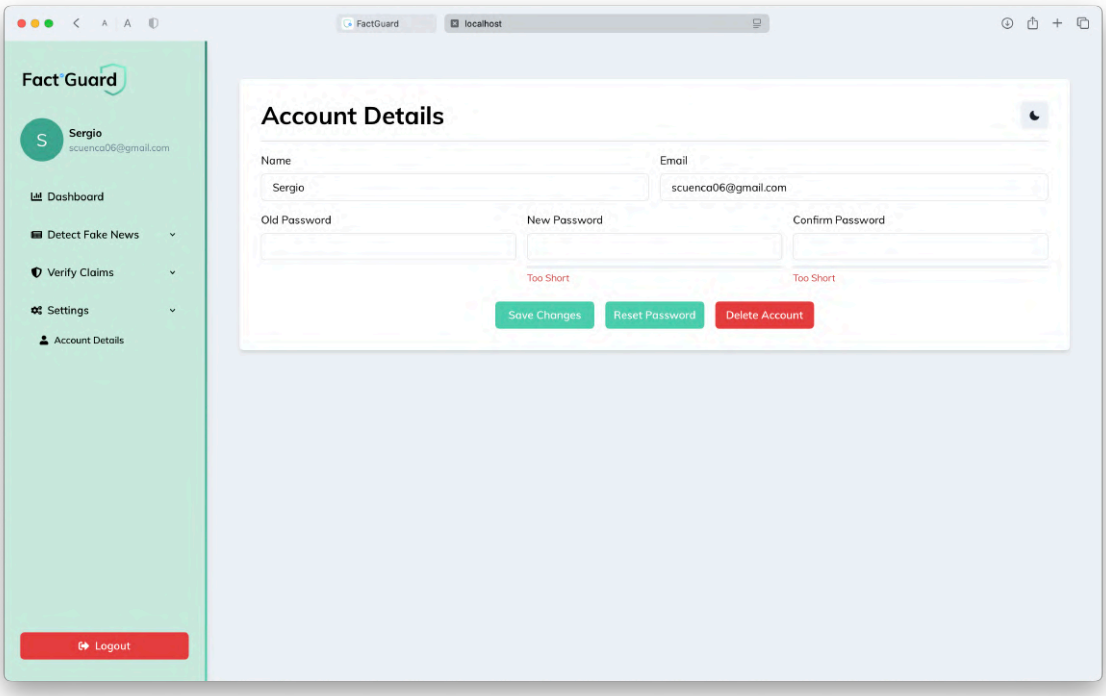


Figura 47. Interfaz de los Ajustes de FactGuard

Fuente: Elaboración Propia

5.6 Servidor de Base de Datos y Servicios Auxiliares (Backend)

La última sección está dedicada a la descripción de los componentes principales que conforman la arquitectura del *backend*, desarrollado en `Node.js` utilizando el *framework* `Express.js`. Este servidor es el encargado de proporcionar las rutas protegidas para la autenticación, la gestión de cuentas y la persistencia de las interacciones realizadas. A continuación, se describen los componentes principales que conforman su arquitectura.

5.6.1 Estructura del Servidor y Rutas

El servidor del *backend* expone una API REST organizada en torno a los principales bloques funcionales del sistema. Todas las rutas se encuentran protegidas mediante

autenticación JWT y un *middleware* que controla el acceso en función del rol del usuario (`admin` o `user`).

La Tabla 9 detalla las rutas implementadas, agrupadas por categoría.

	Endpoint	***HTTP Method***	**Descripción**	***Auth***	***Role***
Autenticación y Configuración	`/signup`	`POST`	Registra un nuevo usuario y devuelve un *token* de autenticación.	**No**	`N/A`
	`/login`	`POST`	Inicia sesión y genera un token JWT. Actualiza `last_access`.	**No**	`N/A`
	`/check-email`	`GET`	Comprueba si un correo ya está registrado en *SignUp*. Si lo está, se redirige al usuario al *Login*.	**No**	`N/A`
	`/check-login-email`	`GET`	Verifica si un correo está registrado en el *Login*. Si lo no está, se redirige al *SignUp*.	**No**	`N/A`
	`/account-update`	`POST`	Actualiza el nombre y correo del usuario autenticado.	**Sí**	`user`
	`/reset-password`	`POST`	Permite cambiar la contraseña mediante validación de la anterior.	**Sí**	`user`
	`/delete-account`	`DELETE`	Elimina la cuenta del usuario autenticado y sus datos asociados (detecciones y verificaciones).	**Sí**	`user`
	`/profile`	`GET`	Recupera el perfil del usuario autenticado.	**Sí**	`user`
Gestión de FactGuard Detect	`/detections`	`GET`	Recupera todas las detecciones. *Admins* reciben todas; usuarios, solo las propias.	**Sí**	`user/admin`
	`/detections/:id`	`GET`	Consulta una detección específica. Filtrado por `user_id` salvo si es *admin*.	**Sí**	`user/admin`
	`/detections`	`POST`	Registra una nueva detección, evitando duplicados.	**Sí**	`user`
	`/detections/:id`	`DELETE`	Elimina una detección concreta. *Admins* pueden borrar cualquiera; usuarios, solo las propias.	**Sí**	`user/admin`

Gestión de FactGuard Verify	`/claims`	`GET`	Recupera todas las verificaciones. *Admins* reciben todas; usuarios, solo las propias	**Sí**	`user/admin`
	`/claims/:id`	`GET`	Consulta una detección específica. Filtrado por `` `user_id` `` salvo si es *admin*.	**Sí**	`user/admin`
	`/claims`	`POST`	Registra una nueva verificación (máximo 3 evidencias).	**Sí**	`user`
	`/claims/:id`	`DELETE`	Elimina una verificación concreta. *Admins* pueden borrar cualquiera; usuarios, solo las propias.	**Sí**	`user/admin`
Rutas Administrativas	`/users`	`GET`	Muestra todos los usuarios con rol `` `user` ``, ordenados por último acceso.	**Sí**	`admin`
	`/users/:id`	`DELETE`	Elimina un usuario específico y todos sus datos asociados.	**Sí**	`admin`
	`/admin/profile`	`GET`	Muestra información del perfil administrativo y métricas agregadas del sistema.	**Sí**	`admin`

Tabla 9. Rutas del Servidor de FactGuard

Fuente: Elaboración Propia

5.6.2 ALMACENAMIENTO DE DATOS E INTERACCIONES

La persistencia de datos en FactGuard se gestiona mediante una base de datos SQLite, elegida por su facilidad a la hora de integrarla en entornos de desarrollo local y su ligereza. En ella se almacena tanto la información de los usuarios como todas las interacciones realizadas a través de los módulos *Detect* y *Verify*. La conexión se realiza directamente desde el servidor *backend*, utilizando el módulo `` `sqlite3` ``.

La estructura principal de la base de datos se compone de tres tablas:

- `` `users` ``: Almacena la información básica de todos los usuarios (`` `id` ``, `` `username` ``, `` `email` ``, `` `password` ``, `` `role` ``, `` `last_access` ``). Este último se actualiza automáticamente cada vez que un usuario inicia sesión o se registra por primera vez,

permitiendo ordenar a los usuarios por actividad reciente desde el panel de administración.

- `detections`: Registra las predicciones y clasificaciones realizadas por FactGuard Detect. Incluye un identificador personalizado (`FGDXX`), el identificador del usuario (`user_id`), el título (`title`) y contenido analizado (`content`), la predicción final (`final_prediction`), la confianza asignada (`confidence`), y los resultados desglosados por modelo: nombres (`models`), predicciones (`predictions`), probabilidades verdaderas (`true_probabilities`) y falsas (`fake_probabilities`), junto con la fecha (`date`). Los resultados generados por los modelos son almacenados en campos JSON.

- `claims`: Almacena las verificaciones realizadas por FactGuard Verify. Contiene un identificador personalizado (`FGVXX`), el identificador del usuario (`user_id`), la afirmación original introducida por el usuario (`query`), y los resultados devueltos por la API de *Google*, que incluyen el texto de la afirmación contrastada (`claims`), el veredicto asignado (`ratings`) y enlace (`links`) para los tres *fact-checks* más recientes. También, se registran el idioma de la búsqueda (`language`) y la fecha (`date`). Los *arrays* de evidencias, valoraciones y enlaces también se almacenan como JSON.

La Figura 48 expone las relaciones entre las tablas, que se mantienen mediante una clave foránea (`user_id`) que vincula cada interacción con su respectivo usuario. Además, se aplican una serie de restricciones para evitar duplicados innecesarios: en `detections`, por título y contenido; en `claims`, por consulta. Antes de cada inserción, el servidor comprueba que no exista ya un registro idéntico para el mismo usuario.

Además, el *backend* incluye funciones auxiliares para generar identificadores personalizados y reiniciar las secuencias de autoincremento (`sqlite_sequence`) tras la

eliminación de registros en la tabla de usuarios. Esto permite mantener la coherencia interna en los identificadores que se asignan a estos.

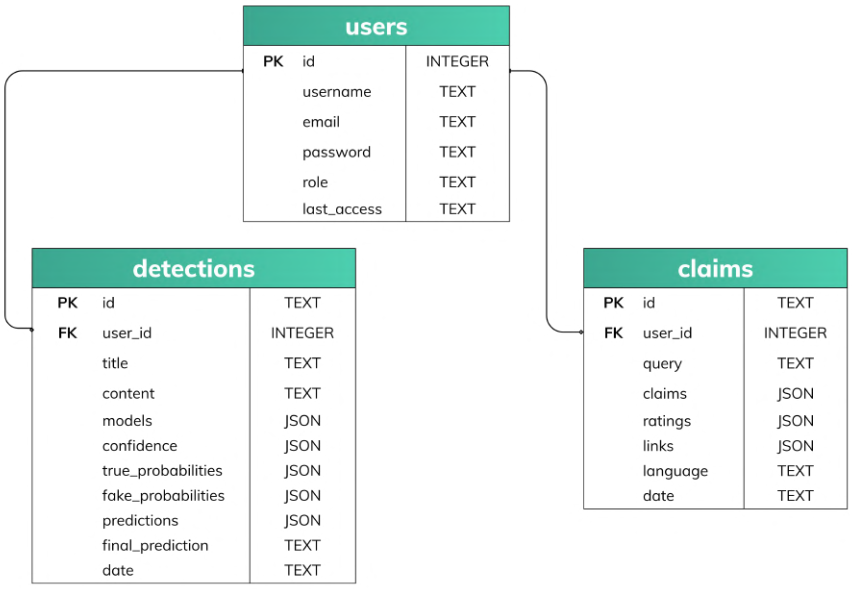


Figura 48. Diagrama Entidad-Relación (ERD) de la Base de Datos de FactGuard

Fuente: Elaboración Propia

Por último, los *endpoints* permiten consultar tanto todas las interacciones de un usuario como los resultados específicos de cada entrada. En el caso de los administradores, se habilita la posibilidad de acceder a todas las entradas registradas sin restricciones por `user_id`.

5.6.3 Seguridad y Gestión de Usuarios

5.6.3.1 Autenticación mediante JWT

Como se ha ido comentando a lo largo del capítulo, la arquitectura de seguridad de FactGuard está basada en un modelo de autenticación sin estado mediante *tokens* JWT, lo que permite verificar la identidad del usuario en cada solicitud sin necesidad de mantener sesiones persistentes en el servidor.

Cuando el usuario completa con éxito el proceso de registro o inicia sesión en la aplicación, se genera un *token* único que encapsula el id del usuario y su rol, con una validez limitada (una hora). Este token debe ser incluido en la cabecera de autorización de todas las solicitudes a rutas protegidas, siguiendo el formato `` `Authorization: `Bearer ${token}` ``. Su uso es obligatorio para acceder a todos los *endpoints* implementados, tanto los del servidor de `` `Node.js` `` que se describe en esta sección, como en los del servidor `` `Flask` `` que maneja la funcionalidad de los módulos principales de detección y verificación.

Para controlar el acceso, ambos servidores incorporan un *middleware* de verificación (`` `verifyToken()` ``), que se encarga de:

- Comprobar la existencia y validez del *token*.
- Decodificar su contenido y asociarlo al usuario.
- Denegar el acceso si el *token* no está presente, está caducado o sea inválido.

Con este *middleware*, presentado en el Listado 17, se garantiza que únicamente los usuarios autenticados o los administradores puedan acceder a datos personales, realizar acciones sobre ellos o utilizar las funcionalidades del sistema.

```
// Middleware to verify token
const verifyToken = (req, res, next) => {
  const token = req.headers.authorization?.split(" ")[1];
  if (!token) return res.status(403).json({ error: "No token provided" });

  jwt.verify(token, SECRET_KEY, (err, decoded) => {
    if (err) return res.status(401).json({ error: "Unauthorized" });
    req.user = decoded;
    next();
  });
};
```

Listado 17. Middleware de Verificación Implementado en el Servidor de `Node.js`

Fuente: Elaboración Propia

5.6.3.2 Control de Acceso y Gestión de Roles

Además, como ya se mostraba en la Tabla 9, se ha implementado un sistema de roles diferenciados que permite condicionar el acceso a ciertas rutas. Por ejemplo, las operaciones de visualización y eliminación de usuarios, así como el acceso al panel administrativo (`/admin/profile`), están restringidas solo a los administradores.

Como se puede observar en la Figura 49 y en la Figura 50, el administrador cuenta con una UI específica que le permite acceder a un panel de control con métricas agregadas del sistema (usuarios, detecciones y verificaciones), gestionar la eliminación de cuentas y/o consultar la actividad reciente de los usuarios registrados.

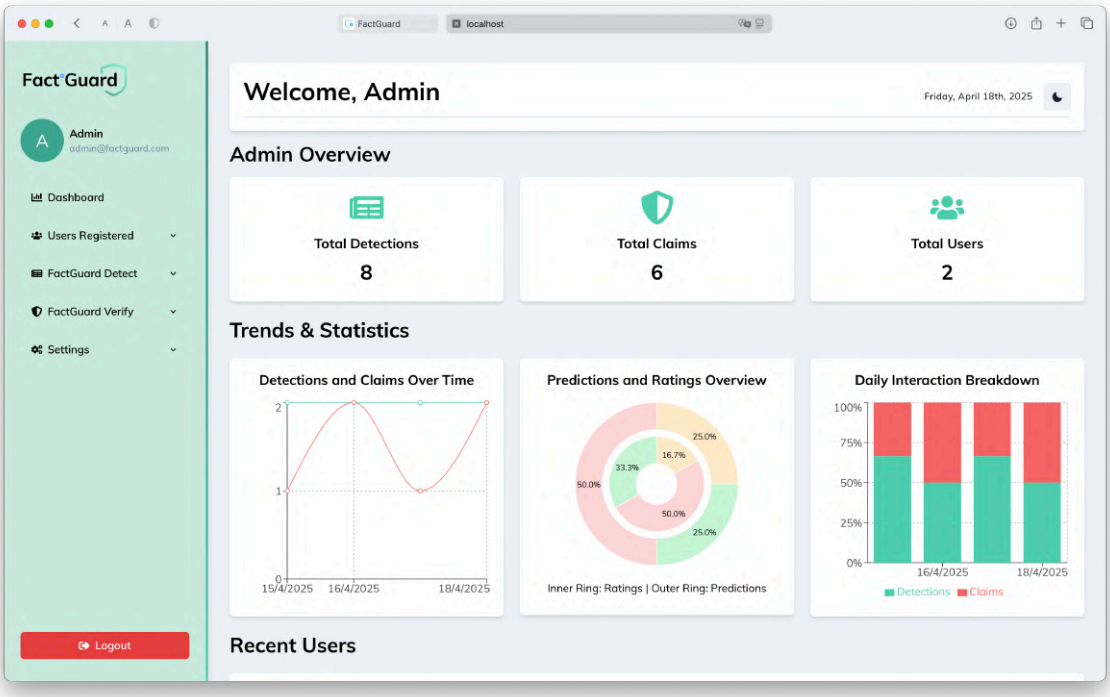
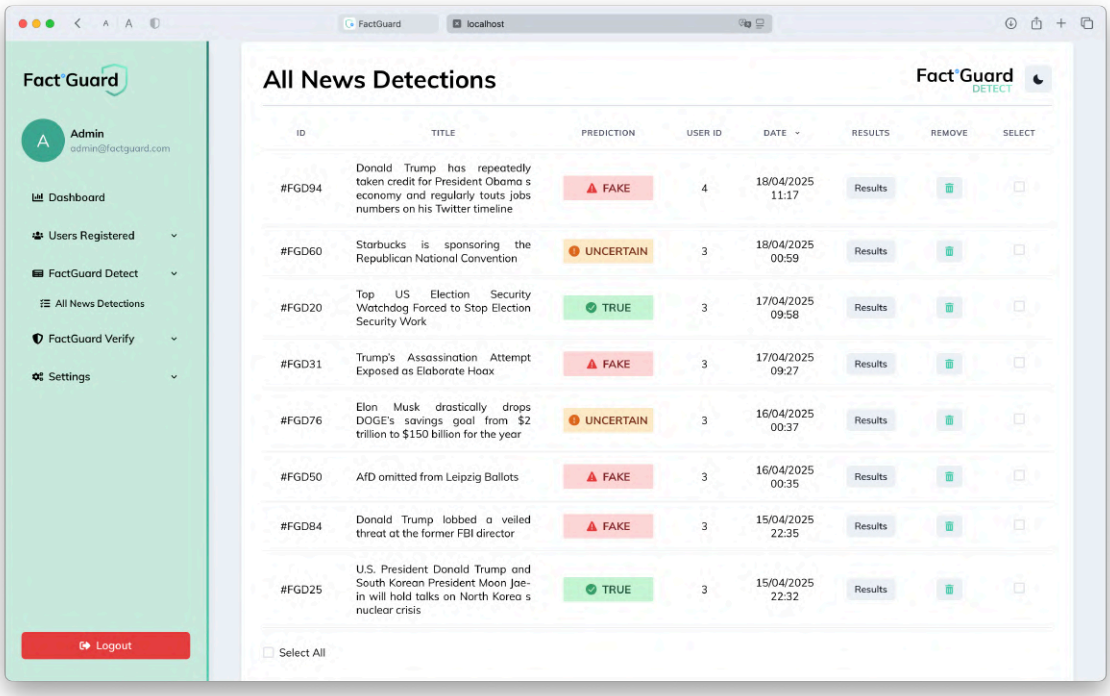


Figura 49. Interfaz del Panel de Administrador de FactGuard

Fuente: Elaboración Propia



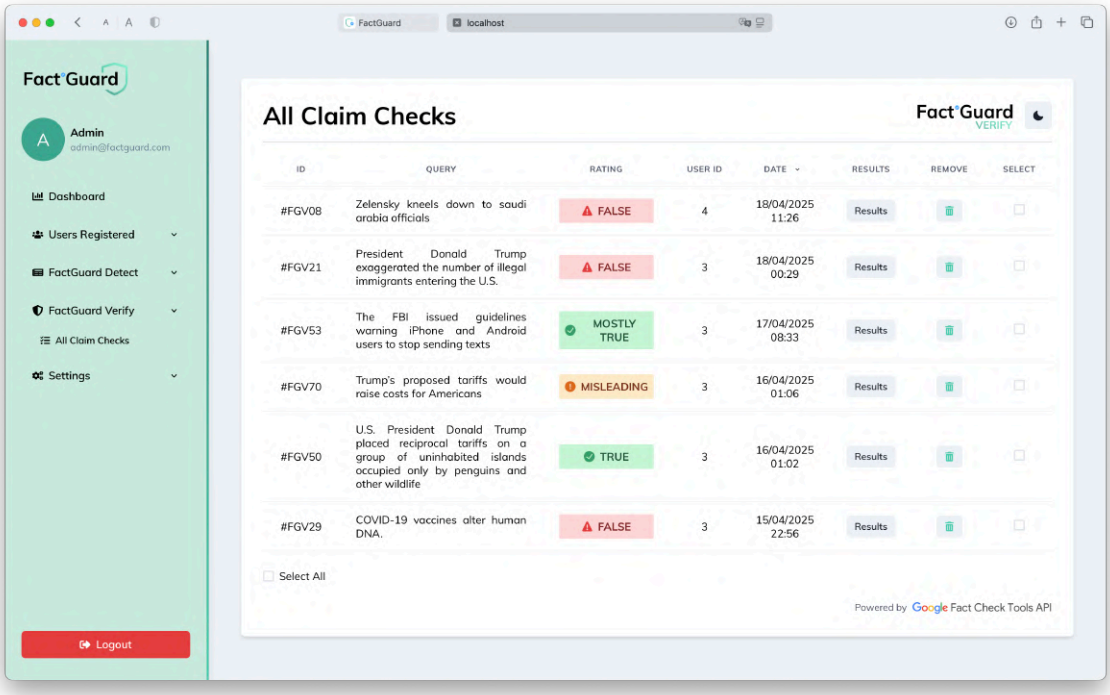


Figura 50. Interfaz de las Detecciones y Verificaciones en el Panel de Administrador

Fuente: Elaboración Propia

5.6.3.3 Validación de Credenciales y Gestión del Perfil

En cuanto a la validación de credenciales, se utiliza el módulo `bcryptjs` para gestionar las contraseñas de forma segura. Durante el registro, las contraseñas se almacenan de forma cifrada, y en el inicio de sesión se comparan utilizando técnicas de *hashing*. Así, el servidor nunca almacena ni manipula contraseñas en texto plano. Este proceso se ilustra en el Listado 18, donde se expone una versión reducida de las rutas correspondientes.

```
// Sign Up Route (hash only)
app.post("/signup", (req, res) => {
  const { email, password } = req.body;
  const hashedPassword = bcrypt.hashSync(password, 10);
  db.run("INSERT INTO users (email, password) VALUES (?, ?)", [email,
hashedPassword]);
  res.json({ message: "User registered" });
});

// Login Route (comparison only)
app.post("/login", (req, res) => {
  const { email, password } = req.body;
  db.get("SELECT * FROM users WHERE email = ?", [email], (err, user) => {
    if (!user || !bcrypt.compareSync(password, user.password)) {
      return res.status(401).json({ error: "Invalid credentials" });
    }
    res.json({ message: "Login successful" });
  });
});
```

Listado 18. Cifrado y Verificación de Contraseñas (Registro e Inicio de Sesión)

Fuente: Elaboración Propia

Por otra parte, el sistema incorpora otras medidas para la gestión segura del perfil:

- Actualización de nombre y correo mediante autenticación.
- Cambio de contraseña con verificación de la contraseña actual.
- Eliminación completa de la cuenta, con el borrado de todas las detecciones y verificaciones asociadas.

Capítulo 6. Análisis de Resultados

Este capítulo expone un análisis crítico y detallado de los resultados obtenidos tras el desarrollo e implementación de FactGuard. A partir de los módulos desarrollados y las funcionalidades descritas en el capítulo anterior, se evalúa el rendimiento del sistema desde una perspectiva técnica, funcional y de experiencia de usuario.

6.1 Evaluación de FactGuard Detect

El módulo de FactGuard Detect es uno de los pilares fundamentales del sistema, dado que se encarga de clasificar mediante los modelos entrenados las noticias introducidas por el usuario según su veracidad. En esta sección se analizan los resultados obtenidos por cada uno de los modelos, así como el impacto del sistema de votación ponderada en la mejora del rendimiento general.

6.1.1 Comparación Global de los Modelos de Detección

Una vez entrenados y evaluados todos los modelos, se procede a realizar un análisis global para evaluar su rendimiento relativo y justificar la estrategia empleada en FactGuard Detect. En la Tabla 10 se recogen los valores de *accuracy*, *precision*, *recall* y *F1-score* para ambas clases, junto con el tiempo aproximado de entrenamiento de cada modelo. Ello permite observar no solo el comportamiento de cada clasificador, sino también su eficiencia computacional y equilibrio en la clasificación binaria.

Modelo	*Accuracy*	*Precision*		*Recall*		*F1-Score*		Tiempo de Entrenamiento
		0	1	0	1	0	1	
CART	0.94	0.95	0.92	0.93	0.95	0.94	0.94	**~46 min**
Random Forest	0.97	0.98	0.96	0.96	0.98	0.97	0.97	**~13 min**

XGBoost	0.98	0.99	0.97	0.98	0.98	0.98	0.98	**~28 min**
LSTM	0.97	0.96	0.98	0.98	0.95	0.97	0.97	**~12 min**
BERT	0.99	0.99	0.99	0.99	0.99	0.99	0.99	**~188 min**

Tabla 10. Comparación de Modelos Empleados en FactGuard Detect

Fuente: Elaboración Propia

Desde una perspectiva de eficacia predictiva, los modelos BERT y *XGBoost* se sitúan a la cabeza, con *F1-scores* de 0.99 y 0.98 respectivamente, mostrando un rendimiento sobresaliente y equilibrado en ambas clases. LSTM también ofrece resultados muy sólidos, con un *F1-score* de 0.96 y una *accuracy* cercana al 98%, siendo además el modelo con menor tiempo de entrenamiento entre los de aprendizaje profundo (únicamente 12 minutos). Por su parte, *Random Forest* combina estabilidad y rapidez, logrando métricas competitivas (*F1-score* de 0.97) en apenas 13 minutos.

A pesar de su sencillez, el árbol de decisión CART presentó el segundo mayor tiempo de entrenamiento (46 minutos) y obtuvo el peor rendimiento global (*F1-score* de 0.94). Finalmente, aunque BERT alcanza una precisión cercana a la perfección, el coste computacional del modelo es muy elevado (aproximadamente 188 minutos), siendo el modelo más exigente en términos de recursos.

6.1.2 Ventajas del Enfoque Híbrido y de Votación Ponderada

La implementación de una estrategia híbrida basada en la detección de *fake news* con varios modelos de ML y DL, junto con un sistema de votación ponderada, aporta a la aplicación una serie de beneficios significativos en términos de robustez, fiabilidad y

representatividad de los resultados. Entre las principales ventajas de FactGuard Detect, cabe destacar las siguientes:

1. Mayor estabilidad frente a errores individuales: Al considerar las predicciones de múltiples modelos, se reducen los efectos de posibles errores puntuales cometidos por alguno de ellos. Esta diversidad de enfoques permite obtener una predicción final más estable y menos sensible a desviaciones extremas.

2. Ponderación proporcional al desempeño real: La ponderación de los votos en función del *F1-score* individual garantiza que la contribución de cada modelo se ajusta a su capacidad predictiva demostrada. De esta forma, se da una mayor importancia e influencia en la votación a aquellos clasificadores que han demostrado un rendimiento más equilibrado y preciso durante la fase de evaluación.

3. Decisión final más informada y representativa: La inclusión de distintas perspectivas en el proceso de decisión final permite obtener un resultado más completo y objetivo. Dado que el sistema no depende de un único modelo, se puede ofrecer una visión más global, lo que se traduce en un análisis más fiable de la veracidad de cada noticia procesada.

4. Presentación clara e integrada de la información: Los resultados devueltos se organizan y presentan en una estructura sencilla y comprensible, facilitando la interpretación sin necesidad de conocimientos técnicos. Esto hace que la funcionalidad sea accesible para una amplia variedad de usuarios.

6.2 Evaluación de FactGuard Verify

La otra gran funcionalidad del sistema está constituida por el módulo de FactGuard Verify, que permite verificar afirmaciones concretas introducidas por el usuario mediante la consulta de fuentes contrastadas por entidades verificadoras reconocidas. La introducción

de este componente venía dada por la necesidad de complementar el sistema de detección automática con evidencia externa, proporcionando al usuario un análisis completo, contrastado y confiable.

6.2.1 VENTAJAS DEL MÓDULO DE VERIFICACIÓN EXTERNA

Si bien no se ha realizado una evaluación cuantitativa del rendimiento del módulo debido a la naturaleza externa y verificada de las fuentes empleadas, es posible destacar una serie de beneficios funcionales que consolidan su valor añadido dentro del sistema.

1. Acceso a evidencia verificable y contextual: El sistema no se limita a mostrar si una afirmación es verdadera o falsa. Junto al veredicto, se presentan enlaces a fuentes verificadoras reconocidas, la fecha de la revisión y el medio que la realizó. Esto permite al usuario consultar directamente la fuente original si desea ampliar la información, haciendo el proceso más transparente.

2. Priorización de resultados relevantes: Dado que la API de Google puede devolver varios resultados para una misma afirmación, el filtrado inteligente ordena las evidencias recuperadas por fecha, mostrando primero las más recientes. De este modo se evita sobrecargar al usuario con información poco actual o repetitiva, y se mejora la claridad de la respuesta.

3. Gestión eficiente de casos sin evidencia disponible: En aquellos casos en los que no se encuentra ninguna evidencia relevante, el sistema devuelve un mensaje claro explicando la situación. Además, se sugiere reformular la afirmación o cambiar el idioma de búsqueda, para que el usuario pueda intentarlo de nuevo sin quedar bloqueado.

6.3 EVALUACIÓN DE LA INTERFAZ DE USUARIO

La UI ha sido diseñada para facilitar al usuario las funcionalidades del sistema, de forma que la interacción sea fluida, clara y comprensible, intentándose adaptar a la experiencia y necesidades de distintos perfiles. Desde las primeras etapas del desarrollo se priorizó la simplicidad en la navegación, la coherencia visual con la elección de colores y disposición de elementos y la claridad en la presentación de los distintos módulos ofrecidos.

Esto hace posible que usuarios con distintos niveles de experiencia tecnológica puedan interactuar con el sistema sin dificultades. Todos los formularios incluyen validaciones y mensajes de ayuda, de forma que el sistema proporcione retroalimentación en caso de errores, ausencia de resultados o acciones exitosas. Al mismo tiempo, gracias a la inclusión de explicaciones claras y accesibles, se facilita la comprensión de los resultados incluso para aquellos que no están familiarizados con el funcionamiento interno de los modelos de detección de *fake news*.

El diseño estructural de la plataforma contempla una separación funcional entre módulos, manteniendo una estética uniforme en todas las vistas y una disposición lógica de los elementos, lo que contribuye a una navegación fluida. Las funcionalidades están claramente identificadas y distribuidas en pestañas diferenciadas, accesibles desde un perfil personalizado una vez el usuario se autentica. Asimismo, el administrador tiene control absoluto sobre todos los datos y métricas agregadas, facilitando el control sobre usuarios e interacciones realizadas en la aplicación.

Asimismo, la posibilidad de elegir entre los distintos modos de visualización permite adaptar el entorno de trabajo a las condiciones de cada usuario, mejorando la accesibilidad y reduciendo la fatiga visual. En conjunto, la interfaz cumple un papel esencial en la propuesta de valor de FactGuard, facilitando el acceso a funcionalidades complejas a través de una experiencia de uso intuitiva y eficiente.

Capítulo 7. Conclusión y Trabajos Futuros

En este último capítulo se extraen las principales conclusiones a las que se han llegado, así como una exposición de las limitaciones detectadas y posibles mejoras. Se proponen distintas líneas de investigación futura que podrían ampliar o mejorar las capacidades del sistema, tanto en términos técnicos como funcionales.

7.1 Limitaciones y Posibles Mejoras

Aunque el sistema ha cumplido con los objetivos y requisitos planteados, y se han llegado a resultados satisfactorios, FactGuard aún presenta ciertas limitaciones que derivan de algunos aspectos técnicos y de infraestructura que no han podido ser resueltas en el momento de la redacción de este proyecto.

7.1.1 Restricciones en el Despliegue y Uso de Modelos de Gran Tamaño

Uno de los principales problemas encontrados durante la fase final del desarrollo ha sido el despliegue de la aplicación, que no ha podido completarse debido a los altos requerimientos de almacenamiento y memoria de los modelos de aprendizaje profundo.

Arquitecturas complejas como BERT requieren una cantidad considerable de espacio en disco y memoria, lo que excede y es muy superior a los límites establecidos por la mayoría de plataformas gratuitas de despliegue o ejecución de servicios (véase *Vercel* o *Render*). Ello imposibilita completamente la posibilidad de ofrecer un sistema accesible al público general y a usuarios que no se encuentren en la red local desde donde se ejecuta la aplicación. Sin recurrir a opciones de pago o infraestructuras específicas, se trata de una barrera técnica que representa un obstáculo significativo.

Como posible mejora, se plantea la optimización de los modelos entrenados, sacrificando la precisión obtenida, así como la reducción en el número de algoritmos empleados y la utilización de modelos más ligeros como DistilBERT, que ofrece un mejor equilibrio entre rendimiento y eficiencia computacional. No obstante, estas modificaciones no garantizan que el espacio de los modelos se reduzca lo suficiente como para ajustarse a las limitaciones impuestas, por lo que se deberían externalizar aquellos procesos computacionalmente exigentes mediante servicios escalables en la nube.

7.1.2 Limitaciones con la Base de Datos Empleada

Durante el desarrollo del proyecto, se ha utilizado SQLite como sistema de gestión de base de datos debido a sus ventajas inherentes como sencillez, portabilidad y facilidad de integración con entornos de desarrollo ligeros. Ello ha permitido avanzar rápidamente en fases preliminares y realizar varias pruebas sin necesidad de hacer configuraciones complejas.

No obstante, SQLite presenta limitaciones importantes en caso de querer realizar un despliegue más ambicioso. Si bien puede ser adecuado en determinados entornos de producción con baja concurrencia de usuarios, no está optimizado para aplicaciones que requieren accesos múltiples simultáneos, almacenamiento en la nube o una gestión avanzada de usuarios y sesiones.

Por tanto, de cara a una implementación real orientada al público general, sería recomendable migrar a un sistema de base de datos más robusto, como PostgreSQL o MongoDB, o incluso mirar servicios gestionados en la nube (*Firebase*, *Supabase* o *Amazon RDS*) que permitan una mayor escalabilidad, seguridad y rendimiento. Por otro lado, en caso de no querer utilizar un servicio web, también podría considerarse la opción de un servidor local con mejores prestaciones para entornos autoalojados.

7.1.3 CARENCIAS EN LA GESTIÓN DE RECUPERACIÓN DE CONTRASEÑAS

Actualmente, FactGuard no incluye un mecanismo para recuperar o restablecer contraseñas olvidadas debido a la ausencia de un servidor de correo configurado para el envío de códigos de verificación o enlaces de reseteo. En caso de que un usuario olvidara su contraseña, el proceso de restablecimiento debería de hacerse manualmente mediante sentencias SQL sobre la base de datos, lo cual limita la escalabilidad y la comodidad de la aplicación en un entorno real de uso.

Aunque no se trata de una limitación que afecta al núcleo funcional del sistema, sí que supone una barrera para la experiencia de usuario en escenarios reales. En versiones futuras, sería recomendable incorporar esta funcionalidad mediante servicios de correo transaccional seguros (como *SendGrid*, *Mailgun* o similares).

7.2 LÍNEAS DE INVESTIGACIÓN FUTURAS

A partir del sistema desarrollado, se identifican una serie de líneas de investigación futura con posible potencial para ampliar el alcance y mejorar el impacto de FactGuard en su lucha contra la desinformación.

7.2.1 INCORPORACIÓN DE FEEDBACK ACTIVO POR PARTE DEL USUARIO

Una posible evolución de la aplicación sería la incorporación de un sistema de retroalimentación que permitiera a los usuarios valorar los resultados obtenidos por los módulos de detección y verificación. Esta funcionalidad estaría basada en un canal de comunicación entre el usuario y FactGuard, permitiendo señalar posibles errores, inexactitudes o casos dudosos en tiempo real.

En aquellos casos en los que se descubra que FactGuard Detect ha realizado una clasificación incorrecta, o que FactGuard Verify ha devuelto verificaciones poco relevantes, el usuario podría señalar el resultado como no válido. De manera voluntaria, también se le ofrecería la opción de aportar información adicional en la que exprese la razón de su desacuerdo, como enlaces, capturas de pantalla o argumentos que contradigan los resultados del sistema.

Desde el punto de vista técnico, esta nueva funcionalidad permitiría una construir una nueva tabla en la base de datos dedicada a este tipo de interacciones. Dicha tabla estaría compuesta en su mayoría por los casos fallidos o conflictivos que podrían ser utilizados para reentrenar o ajustar los modelos en versiones posteriores.

Asimismo, la incorporación de *feedback* en tiempo real abriría la puerta a la implementación de nuevos algoritmos o sistemas de aprendizaje semi-supervisados, en los que el modelo pueda incorporar de forma dinámica los datos aportados por usuario reales. Ello no solo mejoraría notablemente la calidad del conjunto de entrenamiento, sino que permitiría adaptar el comportamiento de FactGuard a contextos de noticias actuales o cambiantes.

7.2.2 Mejora del Tratamiento Multilingüe

Otra posible línea de investigación consistiría en ampliar las capacidades multilingües del sistema, especialmente en lo que se refiere a la detección de *fake news* en otros idiomas distintos del inglés. Actualmente, el sistema está limitado al uso de modelos entrenados con un conjunto de datos en inglés, lo que limita la aplicación del sistema en contextos donde se utilicen otros idiomas.

Para abordar esto, se pueden explorar diversas estrategias. Una de ellas sería el reentrenamiento de los modelos integrados utilizando otros conjuntos de datos específicos en español u otros idiomas, como el corpus LIAR-ES o FAKENEWS-SP. Otra de las

opciones que se puede implementar es la incorporaración de modelos multilingües preentrenados, como mBERT (*Multilingual BERT*), XLM-RoBERTa, o *DistilmBERT-multilingual*, que han sido objeto de entrenamiento con textos en múltiples idiomas y aportan una mayor generalización fuera del inglés.

Por otro lado, también podría considerarse el uso de técnicas de NMT (*Neural Machine Translation*) como un paso previo a la detección. Ello permitiría reutilizar y aplicar los modelos ya entrenados en inglés a contenidos introducidos en otros idiomas. Sin embargo, esta aproximación estaría basada en un enfoque más inexacto, dado que puede conllevar la pérdida de ciertos matices textuales o lingüísticos característicos de las *fake news* o errores de traducción que podrían afectar a la precisión del sistema.

7.2.3 EXPANSIÓN DE LA VERIFICACIÓN CON MÚLTIPLES FUENTES CONTRASTADAS

FactGuard Verify está basado exclusivamente en la conexión e integración con la API de *Google FactCheck Claim Search*. Si bien se trata de una herramienta que permite acceder a gran cantidad de verificaciones realizadas por entidades y fuentes reconocidas, la dependencia de una única API limita la cobertura temática y geográfica del sistema. En aquellos contextos donde las verificaciones publicadas no están contempladas en ciertos idiomas o dominios específicos, el sistema no devuelve coincidencias válidas.

Una línea de investigación futura se centraría en la ampliación y diversificación de fuentes de verificación utilizadas, con el objetivo de mejorar aún más la fiabilidad y el alcance de FactGuard Verify. Esta ampliación podría llevarse a cabo mediante la incorporación de bases de datos de *fact-checking* locales, medios independientes, o incluso plataformas académicas que publiquen repositorios de afirmaciones verificadas.

Un caso potencial de gran interés sería la integración con las Notas de la Comunidad de X, un mecanismo de verificación en el que son los propios usuarios de la red social los que

redactan verificaciones, aclaraciones o contextualizaciones sobre *posts* en los que se redactan *fake news*. Esta funcionalidad, relativamente nueva, ha ganado creciente relevancia como herramienta de verificación ágil y se encuentra respaldada por el público general, siendo utilizada para desmentir afirmaciones virales. Su incorporación podría ofrecer al usuario una visión más cercana al comportamiento real en redes sociales, así como acceso a verificaciones dinámicas y de rápida respuesta.

En lo que respecta a nivel técnico, esta evolución requeriría el diseño de una arquitectura que permita consultar a múltiples APIs o fuentes externas de forma eficiente, integrando los resultados en una estructura común, similar a como ya se realiza en FactGuard Detect. Asimismo, se podrían establecer diferentes criterios de calidad o de prioridad para las fuentes elegidas, con el propósito de evitar conflictos entre verificaciones y asegurar la fiabilidad de los resultados que se presentan al usuario.

7.3 CONSIDERACIONES FINALES

El desarrollo de FactGuard ha permitido abordar el fenómeno de la desinformación desde dos perspectivas: la detección automatizada de *fake news* mediante modelos de ML y DL, y la verificación de afirmaciones mediante fuentes externas contrastadas. A lo largo del proyecto, se ha diseñado e implementado una arquitectura funcional y modular que combina precisión técnica, claridad en la interfaz y transparencia en la presentación de resultados.

El sistema ha cumplido con los objetivos que se planteaban, ofreciendo una herramienta robusta y accesible que permite a los usuarios identificar contenidos que pueden ser objeto de *fake news* y contrastar aquellas afirmaciones que deseen con evidencia verificada. La estrategia de votación ponderada entre modelos, la integración con la API de *Google FactCheck Claim Search*, y el diseño de una interfaz clara constituyen algunos de los logros más relevantes del proyecto.

No obstante, en este último capítulo también se han identificado algunas de las limitaciones relacionadas con el despliegue, la escalabilidad de la base de datos, la gestión de contraseñas o la dependencia de ciertos servicios externos. Las líneas de investigación que se proponen pretenden hacer de la aplicación una plataforma más dinámica, multilingüe, conectada a fuentes diversas y con capacidad para incorporar retroalimentación de los usuarios.

FactGuard no solo representa una herramienta al servicio de la lucha contra la desinformación, sino que también cuenta con una base sólida sobre la que construir nuevas funcionalidades más completas y adaptadas al contexto cambiante del consumo informativo digital. En definitiva, este proyecto pone de manifiesto el potencial de combinar IA, verificación automatizada y diseño centrado en el usuario para afrontar uno de los retos más relevantes de la sociedad actual.

Capítulo 8. BIBLIOGRAFÍA

40dB. (2024). El "desorden democrático" en españa. *El País, Cadena SER,* https://ep00.epimg.net/infografias/encuestas40db/2024/09-barometro/2024_09_barometro_democracia.pdf

A.B., A., Kumar, S. D. M., & Chacko, A. M. (2023). A systematic survey on explainable AI applied to fake news detection. *Engineering Applications of Artificial Intelligence, 122*, 106087. https://doi.org/10.1016/j.engappai.2023.106087

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., . . . Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

Abedalla, A., Al-Sadi, A., & Abdullah, M. (2020). A closer look at fake news detection. *ICAAI '19: Proceedings of the 3rd International Conference on Advances in Artificial Intelligence*, 24–28. https://doi.org/10.1145/3369114.3369149

Ahmed, H., Traore, I., & Saad, S. (2017). Detection of online fake news using N-gram analysis and machine learning techniques. *Traore, I., Woungang, I., Awad, A. (Eds)*

Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments. ISDDC 2017, 10618, 127–138. https://doi.org/10.1007/978-3-319-69155-8_9

Barros, T. S., Pires, C. E. S., & Nascimento, D. C. (2023). Leveraging BERT for extractive text summarization on federal police documents. *Knowledge and Information Systems, 65*(11), 4873–4903. https://doi.org/10.1007/s10115-023-01912-8

Bird, S. (2006). NLTK: The natural language toolkit. *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, 69–72. https://aclanthology.org/P06-4018.pdf

Breiman, L. (2001). Random forests. *Machine Learning, 45*, 5–32. https://doi.org/10.1023/A:1010933404324

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. https://doi.org/10.1145/2939672.2939785

Choi, E. C., & Ferrara, E. (2024). FACT-GPT: Fact-checking augmentation via claim matching with LLMs. *WWW '24: Companion Proceedings of the ACM Web Conference 2024*, 883–886. https://doi.org/10.1145/3589335.3651504

Cugler, E., Silva, M., & Vaz, J. C. (2024). *How disinformation and fake news impact public policies?: A review of international literature*

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 1*, 4171–4186. https://doi.org/10.18653/v1/N19-1423

Dizikes, P. (2018). Study: On twitter, false news travels faster than true stories. *MIT News,* https://news.mit.edu/2018/study-twitter-false-news-travels-faster-true-stories-0308

Evtimov, R., Falli, M., & Maiwald, A. &. (2020). Anti social online behaviour detection with BERT. *Seminar Information Systems (WS19/20),* https://humboldt-wi.github.io/blog/research/information_systems_1920/bert_blog_post/

Felber, T. (2021). Constraint 2021: Machine learning models for COVID-19 fake news detection shared task. *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI-21)*, arXiv preprint arXiv:2101.03717. https://doi.org/10.48550/arXiv.2101.03717

Feldman, R., & Sanger, J. (2006). The text mining handbook: Advanced approaches in analyzing unstructured data. *Cambridge University Press,* https://doi.org/10.1017/CBO9780511546914

Google Fact Check Tools API. (2023). Fact check tools API. https://newsinitiative.withgoogle.com/es-es/resources/trainings/google-fact-check-tools/

Grinberg, M. (2018). Flask web development: Developing web applications with python. https://flaskbook.com

Gunay, D. (2023). Random forest. *Medium,* https://medium.com/@denizgunay/random-forest-af5bde5d7e1e

Guo, R., Zhao, Z., Wang, T., Liu, G., Zhao, J., & Gao, D. (2020). Degradation state recognition of piston pump based on ICEEMDAN and XGBoost. *Applied Sciences, 10*(18), 6593. https://doi.org/10.3390/app10186593

Hassan, N., Zhang, G., Arslan, F., Caraballo, J., Jimenez, D., Gawsane, S., Hasan, S., Joseph, M., Kulkarni, A., Nayak, A., Kumar, Sable, V., Li, C., & Tremayne, M. (2017). ClaimBuster: The first-ever end-to-end fact-checking system. *Proceedings of the VLDB Endowment, 10*(12), 1945–1948. https://doi.org/10.14778/3137765.3137815

Hipp, D. R., Kennedy, D., & Mistachkin, J. (2016). SQLite: A software library that implements a selfcontained, serverless, zero-configuration, transactional SQL database engine. *SQLite Consortium,* https://sqlite.org/

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Jones, M. B., Bradley, J., & Sakimura, N. (2015). JSON web token (JWT). *RFC Editor*, 30. https://doi.org/10.17487/RFC7519

Kaliyar, R. K., Goswami, A., & Narang, P. (2021). FakeBERT: Fake news detection in social media with a BERT-based deep learning approach. *Multimedia Tools and Applications, 80*(8), 11765–11788. https://doi.org/10.1007/s11042-020-10183-2

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). RoBERTa: A robustly optimized BERT pretraining approach., arXiv preprint arXiv:1907.11692. https://doi.org/10.48550/arXiv.1907.11692

Loureiro, D., Barbieri, F., Neves, L., Espinosa Anke, L., & Camacho-Collados, J. (2022). TimeLMs: Diachronic language models from twitter. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 251–260. https://doi.org/10.18653/v1/2022.acl-demo.25

Lv, H., Yang, W., Wei, F., Peng, J., & Geng, H. (2024). MDF: A dynamic fusion model for multi-modal fake news detection ⋆. https://doi.org/10.48550/arXiv.2406.19776

Martín, A., Huertas-Tato, J., Huertas-García, Á, Villar-Rodríguez, G., & Camacho, D. (2022). FacTeR-check: Semi-automated fact-checking through semantic similarity and natural language inference. *Knowledge-Based Systems, 251*, 109265. https://doi.org/10.1016/j.knosys.2022.109265

Meesad, P. (2021). Thai fake news detection based on information retrieval, natural language processing and machine learning. *SN Computer Science, 2*(6), 425. https://doi.org/10.1007/s42979-021-00775-6

Müller, M., Salathé, M., & Kummervold, P. E. (2023). COVID-twitter-BERT: A natural language processing model to analyse COVID-19 content on twitter. *Frontiers in Artificial Intelligence, 6*, 1023281. https://doi.org/10.3389/frai.2023.1023281

Olah, C. (2015). Understanding LSTM networks. *Colah's Blog,* https://colah.github.io/posts/2015-08-Understanding-LSTMs/

OpenJS Foundation.Express. fast, unopinionated, minimalist web framework for node.js. https://expressjs.com

Pavlov, T., & Mirceva, G. (2022). COVID-19 fake news detection by using BERT and RoBERTa models. *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, 312–316. https://doi.org/10.23919/MIPRO55190.2022.9803414

Pedregosa, F., Michel, V., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Cournapeau, D., Varoquaux, G., Gramfort, A., Thirion, B., Passos, A., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research, 12*, 2825–2830. https://doi.org/10.48550/arXiv.1201.0490

Perkins, J. (2010). Python text processing with NLTK 2.0 cookbook. *PACKT Publishing,*

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81–106. https://doi.org/10.1007/BF00116251

Rocha, Y. M., De Moura, G. A., Desidério, G. A., De Oliveira, C. H., Lourenço, F. D., & De Figueiredo Nicolete, L. D. (2023). The impact of fake news on social media and its influence on health during the COVID-19 pandemic: A systematic review. *Journal of Public Health, 31*, 1007–1016. https://doi.org/10.1007/s10389-021-01658-z

Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM, 18*(11), 613–620. https://doi.org/10.1145/361219.361220

Shu, K., Sliva, A., Wang, S., Tang, J., & Liu, H. (2017). Fake news detection on social media: A data mining perspective. *ACM SIGKDD Explorations Newsletter, 19*(1), 22–36. https://doi.org/10.1145/3137597.3137600

Singh, A. (2024). Decision tree in machine learning. *Applied Roots,* https://www.appliedaicourse.com/blog/decision-tree-in-machine-learning/

Singhania, S., Fernandez, N., & Rao, S. (2017). 3HAN: A deep neural network for fake news detection. *Neural Information Processing. ICONIP 2017. Lecture Notes in Computer Science, 10635*, 572–581. https://doi.org/10.1007/978-3-319-70096-0_59

Siva, G. (2021). BERT — bidirectional encoder representations from transformer. *Medium,* https://gayathri-siva.medium.com/bert-bidirectional-encoder-representations-from-transformer-8c84bd4c9021

Stammbach, D. (2019). Team DOMLIN: Exploiting evidence enhancement for the FEVER shared task. *Proceedings of the Second Workshop on Fact Extraction and VERification (FEVER)*, 105–109. https://doi.org/10.18653/v1/D19-6616

Stammbach, D., & Ash, E. (2020). E-FEVER: Explanations and summaries for automated fact checking. *Proceedings of the 2020 Truth and Trust Online (TTO 2020)*, 32–43. https://doi.org/10.3929/ethz-b-000453826

Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing, 14*(6), 80–83. https://doi.org/10.1109/MIC.2010.145

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł, & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems, 30*, 5998–6008. https://doi.org/10.48550/arXiv.1706.03762

Vlachos, A., & Riedel, S. (2014). Fact checking: Task definition and dataset construction. *Proceedings of the ACL 2014 Workshop on Language Technologies and Computational Social Science*, 18–22. https://doi.org/10.3115/v1/W14-2508

We Are Social & Hootsuite. (2021). Digital 2021 global overview report. https://wearesocial.com/uk/blog/2021/01/digital-2021-uk/

X. (2024). About community notes on X. https://help.x.com/en/using-x/community-notes

Zhou, X., & Zafarani, R. (2020). A survey of fake news: Fundamental theories, detection methods, and opportunities. *ACM Computing Surveys (CSUR), 53*(5), 1–40. https://doi.org/10.1145/3395046

ANEXO I: ALINEACIÓN CON LOS ODS

La aplicación que se pretende implementar, que tiene como propósito la identificación de las *fake news* y prevenir su difusión, se encuentra en sintonía con los Objetivos de Desarrollo Sostenible (ODS) impulsados por la Organización de las Naciones Unidas (ONU):

- Objetivo 4. Educación de calidad: FactGuard asegura una educación justa y de alta calidad permitiendo a los estudiantes y docentes contar con una herramienta para detectar información errónea, inventada o inexacta en aquellos proyectos educativos que pretendan elaborar, promoviendo el desarrollo del pensamiento crítico. La aplicación permite a los usuarios corroborar la autenticidad de referencias empleadas, minimizando la posibilidad de difundir información falsa en contextos educativos.

- Objetivo 9. Industria, innovación e infraestructura: La implementación de una herramienta como FactGuard fomenta el progreso tecnológico al hacer uso de herramientas complejas como modelos de ML y DL, y APIs como la de *Google FactCheck Claim Search* en un único sistema. La metodología que sigue el proyecto promueve la creación de infraestructuras digitales robustas que permiten extender los beneficios de herramientas como la IA para el uso general.

- Objetivo 16. Paz, justicia e instituciones sólidas: Representa la principal motivación de FactGuard, al ofrecer una herramienta avanzada para disminuir la cantidad de desinformación que circula por diversos medios digitales y redes sociales. Las *fake news* y los bulos se han constituido como uno de los mayores peligros en el ámbito de la información digital, afectando tanto a los ciudadanos como a las instituciones públicas. Ello influye gravemente en la estabilidad social al fomentar la polarización entre diferentes sectores de la sociedad, disminuyendo la confianza entre los ciudadanos y sus gobernantes.