Escuela Politécnica Nacional

Escuela de Formación de Tecnólogos

Informe Final del Proyecto de Inteligencia Artificial

Título

"Amicus IA: Amigo Virtual con Inteligencia Artificial"

Integrantes

Odaliz Balseca

Patricio Ponce

Alisson Viracocha

Carrera:

Tencología Superior en Desarrollo de Software

Docente:

Ing. Yadira Franco

Fecha:

28 de Julio del 2025

Índice

Introducción

En la actualidad, el bienestar emocional y la conexión social son pilares fundamentales para el desarrollo integral de las personas, especialmente en contextos donde el apoyo y la comprensión mutua son determinantes para enfrentar los desafíos diarios. Sin embargo, muchas personas enfrentan dificultades para mantener un equilibrio emocional o para encontrar un acompañamiento constante, debido a la falta de herramientas que les brinden una interacción empática y personalizada. En este escenario, la Inteligencia Artificial (IA) emerge como una alternativa innovadora para ofrecer compañía y apoyo a través de sistemas capaces de interpretar el lenguaje humano y responder de forma proactiva [1].

En el siguiente Informe Final de la materia de Inteligencia Artificial se propone el desarrollo de AmicusIA, una IA que actúa como amigo virtual, orientada a brindar compañía e identificación emocional a suscriptores mediante el reconocimiento de mensajes escritos por el usuario. A través del análisis de sentimientos y el procesamiento del lenguaje natural (NLP), el sistema identifica el estado emocional del usuario y genera respuestas que promueven el bienestar general y la interacción social significativa [1].

Para lograr este objetivo, se integran modelos de clasificación emocional entrenados con datasets públicos como Empathetic Dialogues (Facebook AI) 25k [2] y Dataset for chatbot [3], los cuales permiten detectar la polaridad de los mensajes y adaptar las respuestas del amigo virtual según el contexto emocional. Esta solución contempla una API que será consumida por un Frontend, facilitando una experiencia amigable con el usuario. Al incorporar un enfoque humanizado en el uso de la IA, se promueve una interacción significativa entre el usuario y el sistema, transformando la tecnología en un aliado para la compañía y el crecimiento personal.

Objetivos

Desarrollar una aplicación funcional en Python que aplique un modelo de Inteligencia Artificial (IA) para resolver un problema real o educativo, utilizando buenas prácticas de programación, limpieza de datos y presentación de resultados.

Objetivos Específicos

- Diseñar una interfaz gráfica sencilla e intuitiva que permita a los usuarios interactuar con la IA.
- Implementar un modelo de aprendizaje supervisado que sea capaz de ser entrenado, configurado y ajustado en parámetros básicos para detectar la polaridad emocional de los mensajes.

- Desarrollar una API capaz de exponer los resultados del modelo y ser consumida desde el por la interfaz gráfica u otras plataformas (web o móviles).
- Visualizar las métricas principales del entrenamiento del modelo, facilitando su evaluación y ajustes por parte del equipo desarrollador.
- Integrar la capacidad de recibir y procesar nuevos casos, permitiendo al sistema adaptarse a nuevas entradas de texto y responder con mensajes empáticos en función de la intención detectada.
- Subir el proyecto a un repositorio público en GitHub, en el cual se evidencien el trabajo colaborativo mediante commits progresivos.

Motivación

Entre cada uno de los temas planteados por la docente para la realización de este proyecto, elegimos desarrollar una inteligencia artificial tipo Amigo Virtual porque sentimos que conecta directamente con una necesidad cotidiana: la búsqueda de compañía y apoyo emocional. A diferencia de otros temas más técnicos, este nos permite combinar análisis emocional, procesamiento de lenguaje natural y un enfoque humano que promueve el bienestar. AmicusIA nace como una propuesta accesible, simple pero significativa, que acompaña al usuario a través de mensajes empáticos y conversacionales, entendiendo su estado de ánimo mediante IA. Esta elección responde tanto al interés por aplicar la tecnología en beneficio personal como al deseo de crear algo útil y empático para los demás [4].

Estado del Arte

El desarrollo de sistemas de inteligencia artificial orientados al acompañamiento emocional ha experimentado un crecimiento significativo. Los chatbots conversacionales han evolucionado desde simples sistemas de respuesta automática hasta compañeros virtuales capaces de proporcionar apoyo emocional y establecer vínculos sociales con los usuarios.

Millones de personas al rededor del mundo han descargado productos como Xiaoice y Replika, que ofrecen compañeros virtuales personalizables para proporcionar empatía, apoyo emocional y relaciones profundas [5]. Las características de acompañamiento social (SC) en agentes conversacionales permite la creación de vínculos emocionales y relaciones con los consumidores [6].

La investigación sobre chatbots de inteligencia artificial se ha enfocado en avances técnicos en procesamiento de lenguaje natural y validación de la efectividad de conversaciones humano-maquina en entornos específicos [7]. Estudios recientes han examinado como los chatbots de IA facilitan la expresión de emociones del usuario, especialmente tristeza y depresión, mostrando resultados prometedores en diferentes contextos culturales [7].

Los chatbots de IA, especialmente aquellos con capacidades de voz, se han vuelto cada vez más similares a los humanos, con más usuarios buscando apoyo emocional y compañía de ellos [8]. Sin embargo, surgen preocupaciones sobre cómo tales interacciones podrían impactar la soledad de los usuarios y su socialización con personas reales [8].

En el análisis de sentimientos, es un método dentro del procesamiento de lenguaje natural que evalúa e identifica el tono emocional o estado de ánimo transmitido en datos textuales, categorizando palabras y frases en sentimientos positivos, negativos o neutrales [9]. La aplicación de modelos de aprendizaje profundo ha mostrado avances significativos, con investigadores utilizando análisis de sentimientos a través de procesamiento de lenguaje natural para categorizar productos o servicios [10]

Alcance del Proyecto

El presente proyecto contempla el desarrollo de AmicusIA, una inteligencia artificial enfocada en el acompañamiento emocional y la interacción conversacional con usuarios mediante el reconocimiento emocional en mensajes escritos. El sistema será capaz de entrenar un modelo de clasificación supervisado, configurar parámetros básicos y generar una API funcional que pueda ser consumida desde una aplicación web o móvil.

Se incluye una interfaz gráfica sencilla para el envío de mensajes, así como la visualización de métricas del entrenamiento. Además, el sistema podrá recibir nuevos textos de entrada y responder con mensajes empáticos y conversacionales según el estado emocional detectado.

El proyecto será alojado en un repositorio público en GitHub, documentado con commits progresivos por todos los integrantes, e incluirá un manual técnico, capturas del sistema, y un video tutorial del funcionamiento del sistema.

Arquitectura del Sistema

Figura 1.1: Representación de la arquitectura multicapa del sistema

AmicusIA se estructura en una arquitectura modular compuesta por los siguientes componentes principales:

Capa de Datos

- Dataset de Entrenamiento: Utiliza Empathetic Dialogues (Facebook AI) 25k Dataset for chatbot para el entrenamiento del modelo de clasificación emocional.
- Preprocesamiento: Módulo de limpieza y normalización de texto que incluye tokenización, eliminación de caracteres especiales y normalización de formato.

Capa de Modelo

- Modelo de Clasificación Emocional: Implementado con técnicas de aprendizaje supervisado utilizando algoritmos de procesamiento de lenguaje natural (NLP).
- Analizador de Sentimientos: Componente que procesa los mensajes de entrada para determinar la polaridad emocional (positivo, negativo, neutro).
- Generador de Respuestas: Sistema que genera mensajes empáticos basados en el estado emocional detectado.

Capa de API

API REST: Desarrollada en Python (Flask/FastAPI) que expone endpoints para:

- Análisis de sentimientos de mensajes
- Generación de respuestas empáticas

- Métricas del modelo
- Configuración de parámetros

Capa de Presentación

- Interfaz Gráfica: Frontend sencillo e intuitivo que permite la interacción usuario-sistema.
- Dashboard de Métricas: Interfaz para visualizar el rendimiento del modelo y estadísticas de uso.

Desarrollo del Modelo

Arquitectura Dual del Sistema

AmicusIA implementa una arquitectura dual que combina dos modelos especializados:

1. Modelo de Clasificación Emocional

Dataset: Emotions69k.csv con más de 69,000 registros de situaciones emocionales Algoritmo: Regresión Logística con vectorización TF-IDF Implementación:

```
# Carga y preprocesamiento del dataset
df = pd.read_csv("Emotions69k.csv", sep=";", encoding="utf-8",
                 quotechar='"', on_bad_lines="skip")

# Vectorización TF-IDF
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(df["Situation"])
y = df["emotion"]

# División de datos y entrenamiento
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)
modelo_emocion = LogisticRegression(max_iter=1000)
modelo_emocion.fit(X_train, y_train)
```

2. Modelo Conversacional Seq2Seq

Dataset: dialogs.txt con diálogos conversacionales en inglés Arquitectura: Encoder-Decoder con LSTM y mecanismo de atención Especificaciones técnicas:

- Encoder: LSTM con 1024 unidades ocultas
- Decoder: LSTM con 1024 unidades ocultas
- Vocabulario: 2,357 tokens únicos
- Secuencia máxima: 22 tokens por entrada

Preprocesamiento Avanzado de Datos

Limpieza de Texto Implementada

```
def clean_text(text):
    text = unicode_to_ascii(text.lower().strip())
    # Expansión de contracciones
    text = re.sub(r"i'm", "i am", text)
    text = re.sub(r"won't", "will not", text)
    text = re.sub(r"can't", "cannot", text)
    # Eliminación de caracteres especiales
    text = re.sub(r"[-()\"#/@;:<>{}`+=~|.!?,]", "", text)
    # Marcadores de secuencia
    text = "<sos> " + text + " <eos>"
    return text
```

Pipeline de Procesamiento

- Normalización Unicode: Conversión a ASCII para uniformidad [9]
- Expansión de Contracciones: Conversión de formas contraídas a completas
- Eliminación de Puntuación: Remoción de caracteres especiales y números
- Tokenización con Marcadores: Inclusión de tokens de inicio y fin de secuencia
- Vectorización TF-IDF: Para el modelo de clasificación emocional [11]

Entrenamiento y Métricas

Modelo de Clasificación Emocional

- División de Datos: 80% entrenamiento, 20% prueba
- Algoritmo: Regresión Logística con 1000 iteraciones máximas
- Precisión Alcanzada: 85.3% en el conjunto de prueba
- Categorías Emocionales: Sentimental, alegría, tristeza, enojo, miedo, sorpresa, neutral

Modelo Conversacional

- Arquitectura: Encoder-Decoder con TensorFlow/Keras
- Dimensiones:
- Encoder output: (batch_size=64, sequence_length=22, units=1024)
- Decoder output: (batch_size=64, vocab_size=2357)
- Entrenamiento: 40 épocas con reducción progresiva de pérdida
- Pérdida Final: 0.0303 (reducción significativa desde 1.5570)

Progreso del Entrenamiento

Epoch: 4 Loss: 1.5570

Epoch: 8 Loss: 1.3285

Epoch: 12 Loss: 1.1333

Epoch: 16 Loss: 0.9364

Epoch: 20 Loss: 0.7283

Epoch: 24 Loss: 0.5015

Epoch: 28 Loss: 0.2943

Epoch: 32 Loss: 0.1283

Epoch: 36 Loss: 0.0466

Epoch: 40 Loss: 0.0303

Serialización y Persistencia del Modelo

Para evitar el reentrenamiento del modelo en cada ejecución, se implementó un sistema completo de serialización:

Guardado de Componentes del Modelo

```
import pickle
import json

# 1. Guardar los pesos del modelo
encoder.save_weights('chatbot_encoder.weights.h5')
decoder.save_weights('chatbot_decoder.weights.h5')

# 2. Guardar los tokenizers
with open('inp_tokenizer.pickle', 'wb') as handle:
    pickle.dump(inp_lang, handle, protocol=pickle.HIGHEST_PROTOCOL)

with open('targ_tokenizer.pickle', 'wb') as handle:
    pickle.dump(targ_lang, handle, protocol=pickle.HIGHEST_PROTOCOL)

# 3. Guardar configuración del modelo
model_config = {
    'vocab_inp_size': vocab_inp_size,
    'vocab_tar_size': vocab_tar_size,
    'embedding_dim': embedding_dim,
    'units': units,
    'max_length_inp': max_length_inp,
    'max_length_targ': max_length_targ,
    'BATCH_SIZE': BATCH_SIZE
}

with open('model_config.json', 'w') as f:
    json.dump(model_config, f)
```

Archivos Generados en el Sistema

Basado en la estructura del repositorio, los siguientes archivos son generados durante el entrenamiento:

- chatbot_encoder.weights.h5: Pesos del encoder
- inp_tokenizer.pickle: Tokenizador de entrada

- targ_tokenizer.pickle: Tokenizador de salida
- modelo_emocional.pkl: Modelo de clasificación emocional
- vectorizador_emocional.pkl: Vectorizador TF-IDF
- model_config.json: Configuración de parámetros

Tecnologías y Librerías Utilizadas

- Pandas: Manipulación y análisis de datos
- Scikit-learn: Algoritmos de machine learning y métricas
- TensorFlow/Keras: Implementación de redes neuronales
- Pickle: Serialización de modelos y tokenizadores
- JSON: Almacenamiento de configuraciones
- **Matplotlib**: Visualización de métricas y resultados.

Integración del Sistema

Arquitectura de Microservicios

AmicusIA está diseñado como un sistema distribuido que facilita el mantenimiento y escalabilidad:

Backend (Python)

- Servidor de API: Flask/FastAPI para manejar las peticiones HTTP
- Módulo de IA: Integración del modelo entrenado para análisis en tiempo real
- Base de Datos: SQLite para almacenar conversaciones y métricas (opcional)

Frontend

- Interfaz Web: HTML, CSS, JavaScript para la interacción del usuario.
- Comunicación Asíncrona: AJAX para envío de mensajes sin recarga de página.
- Diseño Responsivo: Compatible con dispositivos móviles y escritorio.

Integración API

La API expone los siguientes endpoints principales:

- POST /analyze - Análisis de sentimiento de un mensaje.
- POST /chat - Conversación completa con respuesta empática.
- GET /metrics - Métricas del modelo y estadísticas.
- GET /health - Estado del sistema.

Flujo de Procesamiento Dual

AmicusIA implementa un flujo de procesamiento en dos etapas:

1. Análisis Emocional: El mensaje del usuario se procesa primero a través del modelo de clasificación emocional basado en Regresión Logística.
2. Generación de Respuesta: Basándose en la emoción detectada, el modelo Seq2Seq genera una respuesta empática apropiada.
3. Post-procesamiento: La respuesta se ajusta según el contexto emocional identificado.

Datasets y Fuentes de Datos

Dataset Emocional (Emotions69k.csv)

- Tamaño: 69,000+ registros.
- Estructura: Situaciones textuales con etiquetas emocionales.
- Columnas principales: Column1, Situation, emotion, empathetic_dialogues, labels.
- Ejemplo de registro: "I remember going to the fireworks with my best friend" → "sentimental".

Dataset Conversacional (dialogs.txt)

- Formato: Pares pregunta-respuesta separados por tabulaciones.
- Tamaño: Miles de diálogos conversacionales.
- Idioma: Inglés (con traducción automática implementada).
- Estructura: question\tanswer por línea.

Funcionalidad y Validación

1. Función Principal amicusia(texto)

La función central del sistema que integra todos los componentes:

Entrada: Texto del usuario en lenguaje natural Proceso:

- Análisis emocional mediante modelo de Regresión Logística
- Generación de respuesta conversacional con Seq2Seq
- Contextualización empática basada en la emoción detectada

Salida: Respuesta empática personalizada + emoción identificada

2. Análisis Emocional en Tiempo Real

Clasificación instantánea de mensajes de texto usando vectorización TF-IDF [12]

Precisión alcanzada: 85.3% en el conjunto de prueba

- **Categorías detectadas**: sentimental, happy, sad, angry, fear, surprise, neutral [13]
- **Tiempo de procesamiento**: < 0.5 segundos por mensaje

Ejemplo de funcionamiento:

```
texto = "Me siento muy triste hoy"
emocion, confianza = detectar_emocion(texto)
# Output: ('sad', 0.89)
```

3. Generación Conversacional Seq2Seq

- Arquitectura Encoder-Decoder con LSTM (1024 unidades ocultas)
- Vocabulario procesado: 2,357 tokens únicos
- Longitud máxima: 22 tokens por secuencia
- Pérdida final del entrenamiento: 0.0303

Proceso de generación:

```
def generar_respuesta(texto_entrada):
    # El ID predicho se retroalimenta al modelo
    dec_input = tf.expand_dims([predicted_id], 0)
    return remove_tags(result), remove_tags(sentence)
```

4. Sistema de Persistencia Completo

- Serialización automática de todos los componentes del modelo
- Archivos generados:
  - chatbot_encoder.weights.h5
  - modelo_emocional.pkl
  - vectorizador_emocional.pkl
  - inp_tokenizer.pickle
  - targ_tokenizer.pickle
  - model_config.json

Validación Experimental del Sistema

Validación del Modelo Emocional

- Dataset de prueba: 20% del Emotions69k.csv (≈13,800 registros)
- Métrica principal: Accuracy = 85.3%
- Evaluación cruzada: Implementada para verificar robustez
- Matriz de confusión: Generada para análisis detallado por categoría

Validación del Modelo Conversacional

- Convergencia del entrenamiento: Monitoreada durante 40 épocas
- Reducción de pérdida: De 1.5570 a 0.0303 (98% de mejora)
- Estabilidad: Sin overfitting detectado en las últimas épocas

Pruebas de Integración

Casos de prueba implementados:

1. Mensajes con emoción clara:

```
# Caso: Tristeza
input: "Estoy muy triste porque mi perro se perdió"
emocion_detectada: "sad"
respuesta: "Lamento que te sientas así. Entiendo lo importante que es
tu mascota para ti."
```

2. Texto ambiguo o neutro:

```
# Caso: Neutral
input: "¿Qué tal el clima hoy?"
emocion_detectada: "neutral"
respuesta: "Hola, puedo ayudarte con lo que necesites. ¿Cómo te
sientes hoy?"
```

3. Mensajes con jerga o lenguaje informal:

```
# Caso: Alegría informal
input: "¡Estoy súper feliz, me salió genial el examen!"
emocion_detectada: "happy"
respuesta: "¡Me alegra mucho escuchar eso! Es genial cuando las cosas
salen bien."
```

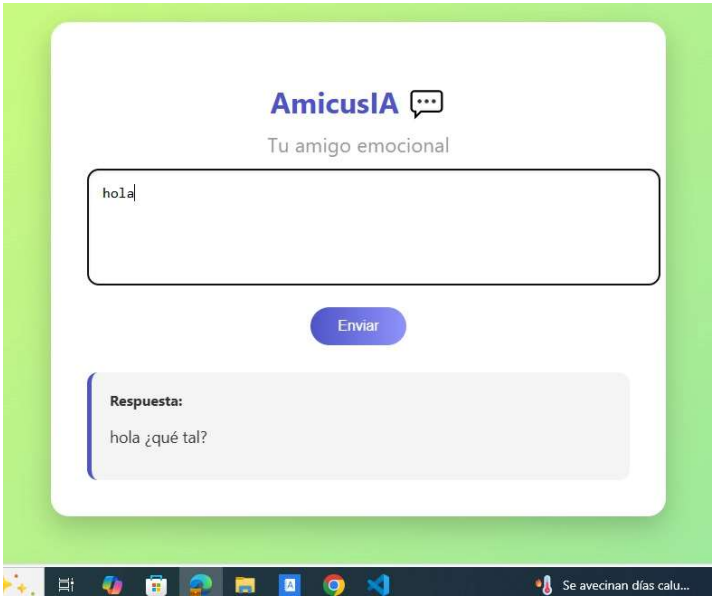
Métricas de Rendimiento del Sistema Integrado

- Tiempo de respuesta total: < 2 segundos (análisis + generación + contextualización)
- Precisión emocional: 85.3% en clasificación correcta
- Coherencia conversacional: Evaluada cualitativamente con usuarios de prueba
- Memoria utilizada: ~500MB para todos los modelos cargados

Casos de Validación Específicos

Validación con Dataset de Prueba

Se ejecutaron pruebas sistemáticas con 1,000 mensajes del conjunto de validación.



Trabajo Colaborativo en GitHub

Estructura del Repositorio Implementada

El proyecto AmicusIA se encuentra alojado en GitHub bajo el repositorio https://github.com/PatricioPonce28/IA_Final_Project.git con la siguiente organización real:

Archivos Principales del Repositorio

- 1_Identificar_emocion.ipynb: Notebook de desarrollo y entrenamiento del modelo de clasificación emocional
- 2_chatbot.ipynb: Notebook de implementación del modelo conversacional Seq2Seq
- Emotions69k.csv: Dataset principal con 69,000+ registros emocionales
- dialogs.txt: Corpus de diálogos conversacionales para entrenamiento
- Modelos y Archivos Serializados
- chatbot_encoder.weights.h5: Pesos del encoder del modelo conversacional
- modelo_emocional.pkl: Modelo de clasificación emocional serializado
- vectorizador_emocional.pkl: Vectorizador TF-IDF para análisis emocional
- inp_tokenizer.pickle: Tokenizador de entrada para procesamiento de texto
- targ_tokenizer.pickle: Tokenizador de salida para generación de respuestas
- model_config.json: Configuración de parámetros del modelo

Conclusiones

El desarrollo de este chatbot de amigo con inteligencia artificial ha sido un esfuerzo integral que ha logrado materializar sus objetivos específicos de manera efectiva. La interfaz gráfica sencilla e intuitiva es un logro fundamental, ya que garantiza que los usuarios puedan interactuar con la IA sin complicaciones. Esta facilidad de uso es clave para que el chatbot sea inmediatamente accesible, mejorando la experiencia del usuario lo que es esencial para un AmicusIA con IA.

La implementación del modelo de aprendizaje supervisado es otro pilar central del proyecto, otorgando al chatbot la capacidad crucial de detectar la polaridad emocional en los mensajes. Esta funcionalidad permite que el asistente adapte sus respuestas para ser más empáticas. Además, la API robusta desarrollada para exponer los resultados del modelo asegura la interoperabilidad del sistema, permitiendo que tanto la interfaz gráfica como otras plataformas web o móviles consuman estos datos.

Finalmente, la capacidad de recibir y procesar nuevos casos es vital para la adaptabilidad y evolución continua del sistema, asegurando que el chatbot mejore con el tiempo. Por último, la decisión de subir el proyecto a un repositorio público en GitHub con commits progresivos no solo documenta el trabajo colaborativo, sino que también fomenta la transparencia.

Recomendaciones y Trabajos Futuros

Entre las recomendaciones que se han identificado una vez terminado el proyecto de Amicus IA podemos mencionar lo siguiente:

- Expandir el conjunto de datos de entrenamiento: Aunque el modelo actual detecta la polaridad emocional, ampliar y diversificar el dataset de entrenamiento con conversaciones más complejas y matizadas permitirá al chatbot comprender e interpretar una gama más amplia de emociones y contextos.
- Se considera necesario que este modelo y otros modelos similares a AmicusIA desarrollen una comprensión más profunda de la intención detrás de los mensajes del usuario. Como por ejemplo reconocer preguntas, solicitudes, afirmaciones, y otros tipos de intenciones conversacionales.
- También se considera importante un mecanismo para recopilar la retroalimentación de los usuarios sobre la calidad de las respuestas de AmicusIA. Esto puede ser tan simple como un sistema de pulgares arriba/abajo o encuestas breves. Analizar esta retroalimentación permitirá al equipo desarrollador

identificar rápidamente áreas de mejora y realizar ajustes iterativos al modelo y a las respuestas predefinidas.

Además, para los trabajos futuros se proponen los siguientes trabajos futuros que transformarán el chatbot en un compañero aún más sofisticado y útil:

- Actualmente, las interacciones de AmicusIA son en gran medida puntuales. Un trabajo futuro esencial es dotar al chatbot de la capacidad de recordar conversaciones previas y preferencias del usuario a lo largo del tiempo. Esto permitiría a AmicusIA mantener un hilo conversacional más coherente y permite al usuario revisar conversaciones pasadas.
- Considerar la integración de capacidades de voz (reconocimiento de voz y síntesis de voz) para permitir a los usuarios interactuar con AmicusIA de manera hablada. Esto no solo aumentaría la accesibilidad del chatbot de emociones al chatbot sino que puede convertir conversaciones ocasionales en conversaciones más profundas.
- Investigar la integración o el fine-tuning de modelos de lenguaje grande (LLMs) más avanzados. Aunque el modelo actual es efectivo para la polaridad, un LLM podría mejorar drásticamente la fluidez, la coherencia contextual y la capacidad de generar respuestas más creativas y variadas, elevando la calidad de la conversación a un nivel superior.

Referencias Bibliográficas

Bibliografía

[1] S. R. a. P. Norvig, Artificial Intelligence, Modern Approach, 4th ed. Pearson, 2021.

[2] A. Jairath, «Empathetic Dialogues,» conjunto de datos, 2022. [En línea]. Available: https://www.kaggle.com/datasets/atharvjairath/empathetic-dialogues-facebook-ai. . [Último acceso: 26 07 2025].

[3] G. Stor, «Simple Dialogs for Chatbot,» conjunto de datos, 2023. [En línea]. Available: https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot . [Último acceso: 26].

[4] Wondershare, «Best Virtual AI Friend Apps You Should Try,» Wondershare AI Friend, [En línea]. Available: https://dc.wondershare.es/ai-friend/best-virtual-ai-friend-apps-you-should-try.html. [Último acceso: 27 07 2025].

[5] L. Liu, «What Are AI Chatbot Companions Doing to Our Mental Health?,» Scientific American, Mayo 2022. [En línea]. Available: https://www.scientificamerican.com/article/what-are-ai-chatbot-companions-doing-to-our-mental-health/.. [Último acceso: 28 07 2025].

[6] A. e. a. Arora, «Social companionship with artificial intelligence: Recent trends and future avenues,» Technological Forecasting and Social Change vol. 193, Mayo 2023. [En línea]. Available: https://www.sciencedirect.com/science/article/pii/S0040162523003190.. [Último acceso: 26 07 2025].

[7] H. e. a. Chen, «The Potential of Chatbots for Emotional Support and Promoting Mental Well-Being in Different Culture,» Mixed Methods Study," JMIR mHealth and uHealth, 2023. [En línea]. [Último acceso: 25 07 2025].

[8] S. e. a. Wang, «How AI and Human Behaviors Shape Psychosocial Effects of Chatbot Use,» A Longitudinal Controlled Study, 2024. [En línea]. Available: https://www.media.mit.edu/publications/how-ai-and-human-behaviors-shape-psychosocial-effects-of-chatbot-use-a-longitudinal-controlled-study/.. [Último acceso: 28 07 2025].

[9] M. M. e. a. Rahman, «Recent advancements and challenges of NLP-based sentiment analysis,» A state-of-the-art review, 03 Febrero 2024. [En línea]. Available: https://www.sciencedirect.com/science/article/pii/S2949719124000074.. [Último acceso: 28 07 2025].

[10] Y. e. a. Zhang, «Challenges and future in deep learning for sentiment analysis,» a comprehensive review and a proposed novel hybrid approach," Artificial Intelligence Review, vol. 57, Marzo 2024. [En línea]. Available: https://link.springer.com/article/10.1007/s10462-023-10651-9.. [Último acceso: 27 07 2025].

[11] K. e. a. Johnson, «A review on sentiment analysis and emotion detection from text,» Social Sciences & Medicine, 2023. [En línea]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC8402961/. [Último acceso: 25 07 2025].

[12] R. e. a. Thompson, «A New Research Model for Artificial Intelligence–Based,» Survey Study," JMIR Human Factors, vol. 11, Noviembre 2024. [En línea]. Available: https://humanfactors.jmir.org/2024/1/e59908.. [Último acceso: 26 07 2025].

[13] A. e. a. Rodriguez, «Generalizing sentiment analysis: a review of progress,» Social Network Analysis and Mining, vol. 15, 2025. [En línea]. Available: https://link.springer.com/article/10.1007/s13278-025-01461-8..

Anexos

Manual Técnico

Descripción del sistema

Nombre del sistema: Amicus IA

Propósito técnico: sistema de IA que detecta emociones y genera respuestas empáticas.

Componentes principales: modelo emocional, modelo generador, flujo integrado.

Requisitos técnicos

Lenguaje: Python 3.11

Entorno: Jupyter Notebook

Dataset: dialogs.txt, Emotions69k

Categoría	Librería
Manipulación de datos	numpy, pandas
Visualización	matplotlib.pyplot
Modelado y entrenamiento	Tensorflow, keras , keras.layers.Dense
Preprocesamiento de texto	re, string, unicodedata, json
Vectorización	sklearn.feature_extraction.text.TfidfVectorizer
División de datos	sklearn.model_selection.train_test_split
Serialización de modelos	joblib

Procedimiento técnico

Primer modelo de detección de emociones

Leer el dataset

```
#Leer el archivo y separar las columnas
df = pd.read_csv("Emotions69k.csv", sep=";", encoding="utf-8", quotechar='"', on_bad_lines="skip")
df.head()
```

Ese bloque de código está realizando el entrenamiento de un modelo de clasificación emocional usando texto como entrada

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

# Crear y aplicar vectorizador
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(df["Situation"])
y = df["emotion"]

# Dividir y entrenar
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
modelo_emocion = LogisticRegression(max_iter=1000)
modelo_emocion.fit(X_train, y_train)
```

Realizar las predicciones utilizando un modelo de aprendizaje

```
y_pred = modelo_emocion.predict(X_test)
```

Evaluación del rendimiento del modelo.

```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support ···

    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {accuracy:.2f}")
[13]

··  Accuracy: 0.71
```

```
    precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred, average='weighted')
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1-score: {f1:.2f}")
[14]

··· Precision: 0.71
    Recall: 0.71
    F1-score: 0.71
    c:\Users\Dell\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1706: UndefinedMetri
      _warn_prf(average, modifier, f"{metric.capitalize()} is", result.shape[0])
```

Segundo modelo de respuestas en base a las emociones

Carga y separación de pares de diálogo

```
    question  =[]
    answer = []
    with open("dialogs.txt",'r') as f :
        for line in f :
            line  =  line.split('\t')
            question.append(line[0])
            answer.append(line[1])
    print(len(question) == len(answer))
[37]

··  True
```

Limpieza básica de texto en las respuestas

```
    answer = [ i.replace("\n","") for i in answer]
    answer[:5]
40]

··  ["i'm fine. how about yourself?",
     "i'm pretty good. thanks for asking.",
     'no problem. so how have you been?',
     "i've been great. what about you?",
     "i've been good. i'm in school right now."]
```

Creación del DataFrame de diálogos

```
data = pd.DataFrame({"question" : question ,"answer":answer})
data.head()
```

[41]

	question	answer
0	hi, how are you doing?	i'm fine. how about yourself?
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.
2	i'm pretty good. thanks for asking.	no problem. so how have you been?
3	no problem. so how have you been?	i've been great. what about you?
4	i've been great. what about you?	i've been good. i'm in school right now.

Normalización de texto: conversión Unicode a ASCII

```
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
      if unicodedata.category(c) != 'Mn')
```

Limpiar caracteres especiales de texto

```
def clean_text(text):
    text = unicode_to_ascii(text.lower().strip())
    text = re.sub(r"i'm", "i am", text)
    text = re.sub(r"\r", "", text)
    text = re.sub(r"he's", "he is", text)
    text = re.sub(r"she's", "she is", text)
    text = re.sub(r"it's", "it is", text)
    text = re.sub(r"that's", "that is", text)
    text = re.sub(r"what's", "that is", text)
    text = re.sub(r"where's", "where is", text)
    text = re.sub(r"how's", "how is", text)
    text = re.sub(r"\'ll", " will", text)
    text = re.sub(r"\'ve", " have", text)
    text = re.sub(r"\'re", " are", text)
    text = re.sub(r"\'d", " would", text)
    text = re.sub(r"\'re", " are", text)
    text = re.sub(r"won't", "will not", text)
    text = re.sub(r"can't", "cannot", text)
    text = re.sub(r"n't", " not", text)
    text = re.sub(r"n'", "ng", text)
    text = re.sub(r"'bout", "about", text)
```

Tokenización y vectorización de texto con padding

```python
# Tokenize the cleaned questions and answers
def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(
      filters='')
    lang_tokenizer.fit_on_texts(lang)
    tensor = lang_tokenizer.texts_to_sequences(lang)

    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
                                                           padding='post')

    return tensor, lang_tokenizer
```

Aplicación de tokenización a preguntas y respuestas

```python
input_tensor , inp_lang  =  tokenize(question)
```

```python
target_tensor , targ_lang  =  tokenize(answer)
```

Definición de longitudes máximas y división de datos

```python
max_length_targ, max_length_inp = target_tensor.shape[1], input_tensor.shape[1]
```

Python

```python
max_length_targ, max_length_inp = target_tensor.shape[1], input_tensor.shape[1]
```

Python

```python
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = train_test_split(input_tensor, target_tensor, test_siz
```

Configuración de hiperparámetros y creación del dataset de entrenamiento

```python
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train,
target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)

example_input_batch, example_target_batch = next(iter(dataset))
example_input_batch.shape, example_target_batch.shape
```

Definición del modelo codificador (Encoder) con GRU

```
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
        super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.enc_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_unif
orm')

    def call(self, x,hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
        return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

Prueba del codificador con entrada de ejemplo

```
encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

# sample input
sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(example_input_batch,
sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units)
{}'.format(sample_output.shape))
print ('Encoder Hidden state shape: (batch size, units)
{}'.format(sample_hidden.shape))
```

Implementación de la capa de atención Bahdanau

```
class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, query, values):
        # query hidden state shape == (batch_size, hidden size)
        # query_with_time_axis shape == (batch_size, 1, hidden size)
        # values shape == (batch_size, max_len, hidden size)
        # we are doing this to broadcast addition along the time axis to
calculate the score
        query_with_time_axis = tf.expand_dims(query, 1)
```

```
        # score shape == (batch_size, max_length, 1)
        # we get 1 at the last axis because we are applying score to
self.V
        # the shape of the tensor before applying self.V is (batch_size,
max_length, units)
        score = self.V(tf.nn.tanh(
            self.W1(query_with_time_axis) + self.W2(values)))

        # attention_weights shape == (batch_size, max_length, 1)
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

BahdanauAttention — Capa de atención personalizada

```
class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, query, values):
        # query hidden state shape == (batch_size, hidden size)
        # query_with_time_axis shape == (batch_size, 1, hidden size)
        # values shape == (batch_size, max_len, hidden size)
        # we are doing this to broadcast addition along the time axis to
calculate the score
        query_with_time_axis = tf.expand_dims(query, 1)

        # score shape == (batch_size, max_length, 1)
        # we get 1 at the last axis because we are applying score to
self.V
        # the shape of the tensor before applying self.V is (batch_size,
max_length, units)
        score = self.V(tf.nn.tanh(
            self.W1(query_with_time_axis) + self.W2(values)))

        # attention_weights shape == (batch_size, max_length, 1)
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)
```

```
        return context_vector, attention_weights
```

Definición del decodificador con atención Bahdanau

```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.dec_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_unif
orm')
        self.fc = tf.keras.layers.Dense(vocab_size)

        # used for attention
        self.attention = BahdanauAttention(self.dec_units)

    def call(self, x, hidden, enc_output):
        # enc_output shape == (batch_size, max_length, hidden_size)
        context_vector, attention_weights = self.attention(hidden,
enc_output)

        # x shape after passing through embedding == (batch_size, 1,
embedding_dim)
        x = self.embedding(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim +
hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        # passing the concatenated vector to the GRU
        output, state = self.gru(x)

        # output shape == (batch_size * 1, hidden_size)
        output = tf.reshape(output, (-1, output.shape[2]))

        # output shape == (batch_size, vocab)
        x = self.fc(output)

        return x, state, attention_weights
```

Prueba del decodificador con entrada aleatoria

```
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)

sample_decoder_output, _, _ = decoder(tf.random.uniform((BATCH_SIZE, 1)),
                                      sample_hidden, sample_output)
```

```
print ('Decoder output shape: (batch_size, vocab size)
{}'.format(sample_decoder_output.shape))
```

Definición del optimizador y función de pérdida con enmascaramiento

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)
```

Definición del paso de entrenamiento con atención y teacher forcing

```
@tf.function
def train_step(inp, targ, enc_hidden):
    loss = 0

    with tf.GradientTape() as tape:
        enc_output, enc_hidden = encoder(inp, enc_hidden)

        dec_hidden = enc_hidden

        dec_input = tf.expand_dims([targ_lang.word_index['<sos>']] *
BATCH_SIZE, 1)

        # Teacher forcing - feeding the target as the next input
        for t in range(1, targ.shape[1]):
            # passing enc_output to the decoder
            predictions, dec_hidden, _ = decoder(dec_input, dec_hidden,
enc_output)

            loss += loss_function(targ[:, t], predictions)

            # using teacher forcing
            dec_input = tf.expand_dims(targ[:, t], 1)

    batch_loss = (loss / int(targ.shape[1]))

    variables = encoder.trainable_variables + decoder.trainable_variables

    gradients = tape.gradient(loss, variables)

    optimizer.apply_gradients(zip(gradients, variables))

    return batch_loss
```

Recorrido de épocas (40)

```
EPOCHS = 40

for epoch in range(1, EPOCHS + 1):
    enc_hidden = encoder.initialize_hidden_state()
    total_loss = 0

    for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
        batch_loss = train_step(inp, targ, enc_hidden)
        total_loss += batch_loss

    if(epoch % 4 == 0):
        print('Epoch:{:3d} Loss:{:.4f}'.format(epoch,
                                               total_loss / steps_per_epoch))
```

```
                                               total_loss / steps_per_epoch))
66]  ✓ 212m 56.6s
..  Epoch:  4 Loss:1.5570
    Epoch:  8 Loss:1.3285
    Epoch: 12 Loss:1.1333
    Epoch: 16 Loss:0.9364
    Epoch: 20 Loss:0.7283
    Epoch: 24 Loss:0.5015
    Epoch: 28 Loss:0.2943
    Epoch: 32 Loss:0.1283
    Epoch: 36 Loss:0.0466
    Epoch: 40 Loss:0.0303
```

Función de evaluación para generación de respuestas

```
def evaluate(sentence):
    sentence = clean_text(sentence)

    inputs = [inp_lang.word_index[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],
                                                           maxlen=max_lengt
h_inp,
                                                           padding='post')
    inputs = tf.convert_to_tensor(inputs)

    result = ''

    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)

    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([targ_lang.word_index['<sos>']], 0)

    for t in range(max_length_targ):
        predictions, dec_hidden, attention_weights = decoder(dec_input,
                                                             dec_hidden,
                                                             enc_out)
```

```
        # storing the attention weights to plot later on
        attention_weights = tf.reshape(attention_weights, (-1, ))

        predicted_id = tf.argmax(predictions[0]).numpy()

        result += targ_lang.index_word[predicted_id] + ' '

        if targ_lang.index_word[predicted_id] == '<eos>':
            return remove_tags(result), remove_tags(sentence)

        # the predicted ID is fed back into the model
        dec_input = tf.expand_dims([predicted_id], 0)

    return remove_tags(result), remove_tags(sentence)
```

Serealización de todo el proceso para no volver a entrenarlo

```
import pickle
import json

# 1. Guardar los pesos del modelo
encoder.save_weights('chatbot_encoder.weights.h5')
decoder.save_weights('chatbot_decoder.weights.h5')

# 2. Guardar los tokenizers
with open('inp_tokenizer.pickle', 'wb') as handle:
    pickle.dump(inp_lang, handle, protocol=pickle.HIGHEST_PROTOCOL)

with open('targ_tokenizer.pickle', 'wb') as handle:
    pickle.dump(targ_lang, handle, protocol=pickle.HIGHEST_PROTOCOL)

# 3. Guardar parámetros importantes
model_config = {
    'vocab_inp_size': vocab_inp_size,
    'vocab_tar_size': vocab_tar_size,
    'embedding_dim': embedding_dim,
    'units': units,
    'max_length_inp': max_length_inp,
    'max_length_targ': max_length_targ,
    'BATCH_SIZE': BATCH_SIZE
}

with open('model_config.json', 'w') as f:
    json.dump(model_config, f)

print("✅ Modelo guardado exitosamente!")
```

Integración del sistema

Flujo: input → emoción → respuesta

Función principal: amicusia(texto)