BITÁCORA

12/10/2024
Intento número 1.

```
import requests
import json

# Define parameters for the request
params = {"category": "Marketing",
         "limit": 5,
        "candidate_required_location": "USA"}

url = "https://remotive.com/api/remote-jobs"

# Make the request with parameters
response = requests.get(url, params=params)

# Check if the request was successful
if response.status_code == 200:
    # Parse the JSON response
    jobs = response.json().get('jobs', [])

    # Print the jobs in a readable format
    print(json.dumps(jobs, indent=4))
else:
    print(f"Error: {response.status_code} - {response.text}")
```

Hasta este punto logramos que la api “remotive” nos devuelva valores, logramos que funcione y obtuvimos respuesta.

Api: https://github.com/remotive-com/remote-jobs-api?tab=readme-ov-file
https://jobicy.com/jobs-rss-feed

14/10/2024

Intentamos implementar una segunda opción para que la API nos traiga datos.

```
import requests

response = requests.get(f'https://remotive.com/api/remote-jobs?limit=5')
jobs = response.json()
jobs = jobs.get("jobs",[])

for job in jobs:
  print(f"{job.get('job_type')} - {job.get('title')} - {job.get('candidate_required_location')}")
```

Esto nos devolvía datos limpios, sin embargo, no es lo que buscábamos en un principio. Buscamos otras APIS.

15/10/2024
Implementamos Jobicy.

https://jobicy.com/jobs-rss-feed

Fue un intento que no funcionó, debido a que jobicy no estaba actualizado. Sin embargo no lo desestimamos porque nos faltaba investigar.

31/10
Implementación del primer bot sin apis de por medio.

```
TOKEN="8084299282:AAHfyGZ-eLEODzj2-Si-9RukLZiu9NZDDhA"
import telebot

# Coloca tu token de bot aquí
bot = telebot.TeleBot(TOKEN)

# Comando /start
@bot.message_handler(commands=['hola'])
def send_welcome(message):
    bot.reply_to(message, "¡Hola! Soy tu bot para buscar laburo. Mandame un mensaje para ver cómo respondo.")

# Comando /ayuda
@bot.message_handler(commands=['ayuda'])
def send_help(message):
    bot.reply_to(message, "Escribí cualquier cosa y te responderé. Usa /Hola para empezar.")

# Manejo de mensajes de texto
@bot.message_handler(func=lambda message: True)
def echo_all(message):
    bot.reply_to(message, f"Dijiste: {message.text}")

# Iniciar el bot
bot.polling()
```

El bot funcionaba, sin embargo no lograba el objetivo ya que no nos traía la información, utilizamos chat gpt para poder implementarlo.

7/11/2024

Intento de combinar BOT + API.

```
import requests
response = requests.get(f'https://remotive.com/api/remote-jobs?limit=5')
jobs = response.json()
jobs = jobs.get("jobs",[])

for job in jobs:
  print(f"{job.get('job_type')} - {job.get('title')} - {job.get('candidate_required_location')}")


TOKEN="8084299282:AAHfyGZ-eLEODzj2-Si-9RukLZiu9NZDDhA"
import telebot

bot = telebot.TeleBot(TOKEN)

# Comando /start
@bot.message_handler(commands=['hola'])
def send_welcome(message):
    bot.reply_to(message, "¡Hola! Soy tu bot para buscar laburo. Mandame un mensaje para ver cómo respondo.")

# Comando /help
@bot.message_handler(commands=['ayuda'])
def send_help(message):
    bot.reply_to(message, "Escribí cualquier cosa y te responderé. Usa /Hola para empezar.")

# Manejo de mensajes de texto
@bot.message_handler(func=lambda message: True)
def echo_all(message):
    bot.reply_to(message, f"Dijiste: {message.text}")

# Iniciar el bot
bot.polling()
```

Se repetían los errores, no pudimos optimizar el bot, se nos complicó la unión de las APIS. No nos traía resultados, sin embargo corría y no pudimos juntar Remotive con Jobicy.

Con la poca información que teníamos, pensamos en cómo unir las APIS.

Pensamos utilizar PANDAS para poder unirlos con tablas. El resultado no estaba mal pero no era funcional para nuestro objetivo.

Ejemplo de pd

```
import pandas as pd

# Datos de ambas APIs
api1_data = [
   {
      "id": 123,
      "url": "https://remotive.com/remote-jobs/product/lead-developer-123",
      "title": "Lead Developer",
      "company_name": "Remotive",
      "company_logo": "https://remotive.com/job/123/logo",
      "category": "Software Development",
      "job_type": "full_time",
      "publication_date": "2020-02-15T10:23:26",
      "candidate_required_location": "Worldwide",
      "salary": "$40,000 - $50,000",
      "description": "The full HTML job description here",
   }
]

api2_data = [
   {
      "id": 111985,
      "url": "https://jobicy.com/jobs/111985-senior-devsecops-engineer",
      "jobSlug": "111985-senior-devsecops-engineer",
      "jobTitle": "Senior DevSecOps Engineer",
      "companyName": "CrossFit",
                                                                  "companyLogo":
"https://jobicy.com/data/server-nyc0409/galaxy/mercury/2021/12/6966c9ea04a0e651f69f113
113a45e27.png",
      "jobIndustry": ["Software Engineering"],
      "jobType": ["full-time"]
   }
]

# Convertir a DataFrames
df_api1 = pd.DataFrame(api1_data)
df_api2 = pd.DataFrame(api2_data)
```

Utilizar PD para poder concatenar la información de ambas APIS.

```
import requests
import pandas as pd

# URLs de las APIs
api_url_1 = "https://jobicy.com/api/v2/remote-jobs"
api_url_2 = "https://remotive.com/api/remote-jobs"

# Obtener datos de la primera API
response_1 = requests.get(api_url_1)
data_api_1 = response_1.json().get("jobs", [])

# Obtener datos de la segunda API
response_2 = requests.get(api_url_2)
data_api_2 = response_2.json().get("jobs", [])

# Convertir datos a DataFrames de Pandas
df_api_1 = pd.DataFrame(data_api_1)
df_api_2 = pd.DataFrame(data_api_2)

# Renombrar columnas en df_api_1 para que coincidan con df_api_2 donde sea posible
df_api_1 = df_api_1.rename(columns={
    "jobTitle": "title",
    "companyName": "company_name",
    "companyLogo": "company_logo",
    "jobIndustry": "category",
    "jobType": "job_type"
})

# Seleccionar columnas de interés para simplificar el dataframe combinado
columns_of_interest = [
    "id", "url", "title", "company_name", "company_logo", "category",
    "job_type", "publication_date", "candidate_required_location", "salary", "description"
]
df_api_1 = df_api_1[columns_of_interest]
df_api_2 = df_api_2[columns_of_interest]

# Concatenar los dos dataframes
df_combined = pd.concat([df_api_1, df_api_2], ignore_index=True)

# Mostrar los primeros registros combinados
print(df_combined.head())
```

14/11/2024

En un giro de tuerca, intentamos implementar Google Cloud Talent. (Acá nos demoramos más de la mitad del trabajo ya que la API no nos servía y no lo pudimos utilizar)

```
from google.cloud import talent_v4beta1


def complete_query(project_id, tenant_id, query):
    """Complete job title given partial text (autocomplete)"""

    client = talent_v4beta1.CompletionClient()

    # project_id = 'Your Google Cloud Project ID'
    # tenant_id = 'Your Tenant ID (using tenancy is optional)'
    # query = '[partially typed job title]'

    if isinstance(project_id, bytes):
        project_id = project_id.decode("utf-8")
    if isinstance(tenant_id, bytes):
        tenant_id = tenant_id.decode("utf-8")
    if isinstance(query, bytes):
        query = query.decode("utf-8")

    parent = f"projects/{project_id}/tenants/{tenant_id}"

    request = talent_v4beta1.CompleteQueryRequest(
        parent=parent,
        query=query,
        page_size=5,  # limit for number of results
        language_codes=["en-US"],  # language code
    )
    response = client.complete_query(request=request)
    for result in response.completion_results:
        print(f"Suggested title: {result.suggestion}")
        # Suggestion type is JOB_TITLE or COMPANY_TITLE
        print(
                                                                f"Suggestion type:
{talent_v4beta1.CompleteQueryRequest.CompletionType(result.type_).name}"
        )
```

SALE ERROR.

INTENTO 2 DE GOOGLE. 14/11/2024

```python
import os
import requests
from google.cloud import talent
import pandas as pd
from tabulate import tabulate
import random
import telebot
from telebot.types import ReplyKeyboardMarkup, KeyboardButton

# Configurar el token del bot de Telegram
BOT_TOKEN = '7916035380:AAGeAhkPYL1i0K8e_FOks05Q1a0YeMnvRy4'
bot = telebot.TeleBot(BOT_TOKEN)

# Configurar las credenciales de Google Cloud
os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = 'client_secret_81853400008-04t9d57h9dghf9amsshijk9jnvsbd6nk.apps.googleusercontent.com (1).json'

# Definir los rubros
RUBROS = ["Tecnología", "Marketing", "Finanzas", "Recursos Humanos", "Ventas", "Diseño", "Educación", "Otro"]

def get_google_jobs(query, location, limit=5):
    client = talent.JobServiceClient()
    parent = f"projects/{os.environ['GOOGLE_CLOUD_PROJECT']}/tenants/{os.environ['GOOGLE_CLOUD_TENANT']}"

    job_query = talent.JobQuery(query=query, location_filters=[talent.LocationFilter(address=location)])
    request = talent.SearchJobsRequest(
        parent=parent,
        job_query=job_query,
        job_view=talent.JobView.JOB_VIEW_FULL,
        limit=limit
    )

    response = client.search_jobs(request=request)
    return [job.job for job in response.matching_jobs]

def get_jobicy_jobs(query, location, limit=5):
    api_url = "https://jobicy.com/api/v2/remote-jobs"
    params = {"query": query, "limit": limit, "location": location}
    response = requests.get(api_url, params=params)
    if response.status_code == 200:
        return response.json().get("jobs", [])
    else:
        print(f"Error al obtener datos de Jobicy: {response.status_code}")
        return []

def format_google_job(job):
    return {
        "title": job.title,
        "company": job.company_display_name,
        "location": job.addresses[0] if job.addresses else "Remote",
        "url": job.application_info.uris[0] if job.application_info.uris else "",
        "source": "Google Job Search API"
    }

def format_jobicy_job(job):
    return {
        "title": job.get("title"),
        "company": job.get("company_name"),
        "location": job.get("candidate_required_location", "Remote"),
        "url": job.get("url"),
        "source": "Jobicy API"
    }

def get_combined_jobs(query, location):
    google_jobs = get_google_jobs(query, location, limit=5)
    jobicy_jobs = get_jobicy_jobs(query, location, limit=5)

    formatted_google_jobs = [format_google_job(job) for job in google_jobs]
    formatted_jobicy_jobs = [format_jobicy_job(job) for job in jobicy_jobs]

    all_jobs = formatted_google_jobs + formatted_jobicy_jobs
    random.shuffle(all_jobs)

    return all_jobs[:10]

@bot.message_handler(commands=['start', 'help'])
def send_welcome(message):
    bot.reply_to(message, "Bienvenido! Por favor, ingresa el país donde buscas trabajo.")
    bot.register_next_step_handler(message, ask_rubro)

def ask_rubro(message):
    global user_location
    user_location = message.text
    markup = ReplyKeyboardMarkup(row_width=2)
    markup.add(*[KeyboardButton(rubro) for rubro in RUBROS])
    bot.reply_to(message, "Selecciona el rubro en el que buscas trabajo:", reply_markup=markup)
    bot.register_next_step_handler(message, process_job_query)

def process_job_query(message):
    rubro = message.text
    if rubro not in RUBROS:
        bot.reply_to(message, "Por favor, selecciona un rubro válido.")
        return

    bot.reply_to(message, f"Buscando trabajos de {rubro} en {user_location}. Por favor, espera...")
    jobs = get_combined_jobs(rubro, user_location)

    if not jobs:
        bot.reply_to(message, "Lo siento, no se encontraron trabajos para tu consulta.")
        return

    response = f"Top 10 puestos laborales de {rubro} en {user_location}:\n\n"
    for i, job in enumerate(jobs, 1):
        response += f"{i}. {job['title']}\n"
        response += f"   Empresa: {job['company']}\n"
        response += f"   Ubicación: {job['location']}\n"
        response += f"   Fuente: {job['source']}\n"
        response += f"   URL: {job['url']}\n\n"

    bot.reply_to(message, response)
    bot.reply_to(message, "¿Quieres buscar en otro rubro o país? Envía /start para comenzar de nuevo.")

# Iniciar el bot
bot.polling()
```

No funcionaba, no trae opciones. Google Talent no se podía utilizar, era demasiado difuso y no daba órdenes claras, no se podía traer ningún trabajo y el bot se rompía cada vez que se lo llamaba.

21/11/2024

Volvimos con la Api remotive, implementamos Gemini para poder mejorarlo y funcionó muy bien, lo único que había que pulirlo un poco más.

```
import telebot
import requests
import google.generativeai as genai
import os
from telebot import types

# Configurar las llaves API de Google Gemini y Telegram
try:
    os.environ["API_KEY"] = "AIzaSyBmEsrcoQO36YNfPGizkdjIo1IVqaPzHUg"
    genai.configure(api_key=os.environ["API_KEY"])
except Exception as e:
    print(f"Error al configurar la API de Gemini: {e}")
    exit()

# Crear el modelo de Gemini
try:
    model = genai.GenerativeModel('gemini-1.5-flash-latest')
except Exception as e:
    print(f"Error al crear el modelo de Gemini: {e}")
    exit()

# Token de tu bot de Telegram
TOKEN = "7972576195:AAFApKCt351YIzZzb3yjRkGCrqZVCGzAHrU"
bot = telebot.TeleBot(TOKEN)

# URL de la API de Remotive
API_URL = "https://remotive.com/api/remote-jobs"

# Diccionario de áreas de trabajo y localidades disponibles para los botones
AREAS_DE_TRABAJO = ['Engineering', 'Design', 'Product', 'Sales', 'Marketing']
LOCALIDADES = ['Remote', 'United States', 'Europe', 'Asia', 'Africa']

# Variables para almacenar la selección de búsqueda
seleccion_localidad = None
seleccion_area_trabajo = None

# Función que hace la búsqueda de empleos y muestra los primeros 5 resultados
def buscar_empleos(area_trabajo, localidad):
```

```
    params = {
        'category': area_trabajo,
        'location': localidad
    }

    try:
        response = requests.get(API_URL, params=params)
        response.raise_for_status()  # Lanza un error si la respuesta no es exitosa (status 200)
        data = response.json()
        jobs = data.get('jobs', [])
        if jobs:
            result = ""
            for i, job in enumerate(jobs[:5]):
                result += f"{i + 1}. Título: {job['title']}\n"
                result += f"   Empresa: {job['company_name']}\n"
                result += f"   Localidad: {job['candidate_required_location']}\n"
                result += f"   Enlace: {job['url']}\n"
                result += "-" * 5 + "\n"
            return result
        else:
            return "No se encontraron empleos para esta búsqueda.\n"
    except requests.exceptions.RequestException as e:
        print(f"Error al hacer la solicitud a la API de Remotive: {e}")
        return "Hubo un error al conectar con la API de empleos. Intenta de nuevo.\n"

# Función para interactuar con Gemini
def hablar_con_workie(pregunta):
    try:
        respuesta = model.generate_content(pregunta)
        return respuesta.text
    except Exception as e:
        print(f"Error al generar contenido con Gemini: {e}")
        return "Hubo un error al interactuar con Workie. Intenta de nuevo.\n"

# Función que muestra los botones de opciones después de mostrar los resultados
def mostrar_opciones(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)

    # Agregar botones con las opciones
    markup.add(types.KeyboardButton("Hablar con Workie"))
    markup.add(types.KeyboardButton("Ver 5 trabajos más"))
    markup.add(types.KeyboardButton("Finalizar conversación"))

    bot.send_message(
        message.chat.id,
        "¿Qué te gustaría hacer ahora?",
        reply_markup=markup
    )
```

```
# Función que maneja el comando /start y muestra los botones de selección de localidad
@bot.message_handler(commands=['start'])
def enviar_bienvenida(message):
    # Crear un teclado con botones de localidades
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    localidad_buttons = [types.KeyboardButton(localidad) for localidad in LOCALIDADES]
    markup.add(*localidad_buttons)

    bot.send_message(
        message.chat.id,
        "¡Bienvenido! Elige una localidad para buscar empleos:",
        reply_markup=markup
    )

    # Guardamos el paso actual (localidad) para usarlo en la siguiente fase
    bot.register_next_step_handler(message, handle_localidad_selection)

# Función que maneja la selección de localidad
def handle_localidad_selection(message):
    global seleccion_localidad
    seleccion_localidad = message.text

    # Crear un teclado con botones de áreas de trabajo
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    area_buttons = [types.KeyboardButton(area) for area in AREAS_DE_TRABAJO]
    markup.add(*area_buttons)

    bot.send_message(
        message.chat.id,
            f"Has seleccionado la localidad: {seleccion_localidad}. Ahora selecciona un área de
trabajo:",
        reply_markup=markup
    )

    # Guardamos la localidad seleccionada para usarla después
    bot.register_next_step_handler(message, handle_area_selection)

# Función que maneja la selección de área de trabajo
def handle_area_selection(message):
    global seleccion_area_trabajo
    seleccion_area_trabajo = message.text

    # Realizar la búsqueda de empleos con la localidad y el área de trabajo seleccionados
    result = buscar_empleos(seleccion_area_trabajo, seleccion_localidad)
    bot.send_message(message.chat.id, result)

    # Mostrar opciones para continuar la conversación
```

```
    mostrar_opciones(message)

# Función que maneja las opciones seleccionadas por el usuario
@bot.message_handler(func=lambda message: message.text in ["Hablar con Workie", "Ver
5 trabajos más", "Finalizar conversación"])
def manejar_opciones(message):
    if message.text == "Hablar con Workie":
        # Hablar con Workie usando Gemini
        bot.send_message(message.chat.id, "¡Hola! ¿En qué puedo ayudarte hoy?")
        bot.register_next_step_handler(message, procesar_pregunta_workie)

    elif message.text == "Ver 5 trabajos más":
        # Mostrar otros 5 trabajos con la misma búsqueda
        result = buscar_empleos(seleccion_area_trabajo, seleccion_localidad)
        bot.send_message(message.chat.id, result)

        # Mostrar opciones nuevamente
        mostrar_opciones(message)

    elif message.text == "Finalizar conversación":
        # Finalizar la conversación
        bot.send_message(message.chat.id, "Gracias por usar el bot. ¡Hasta pronto!")
        return

# Función para procesar preguntas a Workie (Gemini)
def procesar_pregunta_workie(message):
    pregunta = message.text
    respuesta = hablar_con_workie(pregunta)
    bot.send_message(message.chat.id, respuesta)

    # Mostrar opciones para continuar la conversación
    mostrar_opciones(message)

# Inicia el bot
try:
    print("Iniciando bot de Telegram...")
    bot.polling()
except Exception as e:
    print(f"Error al iniciar el bot: {e}")
```

22/11/2024

El siguiente código era casi definitivo, solo que nos fallaba una opción que era "Hablar con Workie" que era interactuar con la IA de Gemini.

```
import telebot
import requests
import google.generativeai as genai
import os
from telebot import types

# Configurar las llaves API de Google Gemini y Telegram
try:
    os.environ["API_KEY"] = "AIzaSyBmEsrcoQO36YNfPGizkdjIo1IVqaPzHUg"
    genai.configure(api_key=os.environ["API_KEY"])
except Exception as e:
    print(f"Error al configurar la API de Gemini: {e}")
    exit()

# Crear el modelo de Gemini
try:
    model = genai.GenerativeModel('gemini-1.5-flash-latest')
except Exception as e:
    print(f"Error al crear el modelo de Gemini: {e}")
    exit()

# Token de tu bot de Telegram
TOKEN = "7972576195:AAFApKCt351YIzZzb3yjRkGCrqZVCGzAHrU"
bot = telebot.TeleBot(TOKEN)

# URL de la API de Remotive
API_URL = "https://remotive.com/api/remote-jobs"

# Diccionario de áreas de trabajo y localidades disponibles para los botones
AREAS_DE_TRABAJO = ['Engineering', 'Design', 'Product', 'Sales', 'Marketing']
LOCALIDADES = ['Remote', 'United States', 'Europe', 'Asia', 'Africa']

# Variables para almacenar la selección de búsqueda
seleccion_localidad = None
seleccion_area_trabajo = None
```

```
# Variables para almacenar los trabajos mostrados
trabajos_mostrados = []
trabajos_mostrados_totales = []  # Para almacenar todos los trabajos mostrados

# Función que hace la búsqueda de empleos y muestra los primeros 5 resultados
def buscar_empleos(area_trabajo, localidad, mostrar_nuevos=True):
    global trabajos_mostrados, trabajos_mostrados_totales
    params = {
        'category': area_trabajo,
        'location': localidad
    }

    try:
        response = requests.get(API_URL, params=params)
        response.raise_for_status()  # Lanza un error si la respuesta no es exitosa (status 200)
        data = response.json()
        jobs = data.get('jobs', [])

        # Filtramos para mostrar trabajos que no hayan sido mostrados previamente
        if mostrar_nuevos:
            nuevos_trabajos = [job for job in jobs if job not in trabajos_mostrados_totales]
        else:
            nuevos_trabajos = jobs

        if nuevos_trabajos:
            result = ""
            for i, job in enumerate(nuevos_trabajos[:5]):
                result += f"{i + 1}. Título: {job['title']}\n"
                result += f"   Empresa: {job['company_name']}\n"
                result += f"   Localidad: {job['candidate_required_location']}\n"
                result += f"   Enlace: {job['url']}\n"
                result += "-" * 5 + "\n"
            # Actualizamos la lista de trabajos mostrados
            trabajos_mostrados_totales.extend(nuevos_trabajos[:5])
              trabajos_mostrados = trabajos_mostrados_totales  # Guardamos todos los trabajos
mostrados
            return result
        else:
            return "No se encontraron nuevos empleos para esta búsqueda.\n"
    except requests.exceptions.RequestException as e:
        print(f"Error al hacer la solicitud a la API de Remotive: {e}")
        return "Hubo un error al conectar con la API de empleos. Intenta de nuevo.\n"

# Función para interactuar con Gemini
def hablar_con_workie(pregunta):
    try:
        respuesta = model.generate_content(pregunta)
        return respuesta.text
```

```
    except Exception as e:
        print(f"Error al generar contenido con Gemini: {e}")
        return "Hubo un error al interactuar con Workie. Intenta de nuevo.\n"

# Función que muestra las opciones después de una respuesta de Workie
def mostrar_opciones_workie(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    markup.add(types.KeyboardButton("Seguir hablando con Gemini"))
    markup.add(types.KeyboardButton("Buscar nuevos empleos"))
    markup.add(types.KeyboardButton("Finalizar el workbot"))

    bot.send_message(
        message.chat.id,
        "¿Qué te gustaría hacer ahora?",
        reply_markup=markup
    )

# Función que muestra las opciones de búsqueda después de obtener resultados de trabajos
def mostrar_opciones(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    markup.add(types.KeyboardButton("Hablar con Workie"))
    markup.add(types.KeyboardButton("Buscar más empleos"))
    markup.add(types.KeyboardButton("Realizar una búsqueda nueva"))
    markup.add(types.KeyboardButton("Finalizar el workbot"))

    bot.send_message(
        message.chat.id,
        "¿Qué te gustaría hacer ahora?",
        reply_markup=markup
    )

# Función que maneja el comando /start y muestra los botones de selección de localidad
@bot.message_handler(commands=['start'])
def enviar_bienvenida(message):
    # Crear un teclado con botones de localidades
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    localidad_buttons = [types.KeyboardButton(localidad) for localidad in LOCALIDADES]
    markup.add(*localidad_buttons)

    bot.send_message(
        message.chat.id,
        "¡Bienvenido! Elige una localidad para buscar empleos:",
        reply_markup=markup
    )

    # Guardamos el paso actual (localidad) para usarlo en la siguiente fase
    bot.register_next_step_handler(message, handle_localidad_selection)
```

```
# Función que maneja la selección de localidad
def handle_localidad_selection(message):
    global seleccion_localidad
    seleccion_localidad = message.text

    # Crear un teclado con botones de áreas de trabajo
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    area_buttons = [types.KeyboardButton(area) for area in AREAS_DE_TRABAJO]
    markup.add(*area_buttons)

    bot.send_message(
        message.chat.id,
            f"Has seleccionado la localidad: {seleccion_localidad}. Ahora selecciona un área de
trabajo:",
        reply_markup=markup
    )

    # Guardamos la localidad seleccionada para usarla después
    bot.register_next_step_handler(message, handle_area_selection)

# Función que maneja la selección de área de trabajo
def handle_area_selection(message):
    global seleccion_area_trabajo
    seleccion_area_trabajo = message.text

    # Realizar la búsqueda de empleos con la localidad y el área de trabajo seleccionados
    result = buscar_empleos(seleccion_area_trabajo, seleccion_localidad)
    bot.send_message(message.chat.id, result)

    # Mostrar opciones para continuar la conversación
    mostrar_opciones(message)

# Función que maneja las opciones seleccionadas por el usuario
@bot.message_handler(func=lambda message: message.text in ["Hablar con Workie",
"Buscar más empleos", "Realizar una búsqueda nueva", "Finalizar el workbot"])
def manejar_opciones(message):
    if message.text == "Hablar con Workie":
        # Iniciar conversación con Workie
        bot.send_message(message.chat.id, "¡Hola! ¿En qué puedo ayudarte hoy?")
        bot.register_next_step_handler(message, procesar_pregunta_workie)

    elif message.text == "Buscar más empleos":
            # Mostrar otros 5 trabajos con la misma búsqueda, sin repetir los ya mostrados
anteriormente
                    result = buscar_empleos(seleccion_area_trabajo, seleccion_localidad,
mostrar_nuevos=True)
        bot.send_message(message.chat.id, result)
```

```
        # Mostrar opciones nuevamente
        mostrar_opciones(message)

    elif message.text == "Realizar una búsqueda nueva":
        # Volver a empezar la búsqueda
        bot.send_message(message.chat.id, "¡Qué bueno! ¿En qué localidad?")

        # Crear el teclado de localidades para elegir nuevamente
                                markup = types.ReplyKeyboardMarkup(resize_keyboard=True,
one_time_keyboard=True)
        localidad_buttons = [types.KeyboardButton(localidad) for localidad in LOCALIDADES]
        markup.add(*localidad_buttons)

                bot.send_message(message.chat.id, "Elige una localidad para comenzar.",
reply_markup=markup)
        bot.register_next_step_handler(message, handle_localidad_selection)

    elif message.text == "Finalizar el workbot":
        # Finalizar la conversación
        bot.send_message(message.chat.id, "Gracias por usar el bot. ¡Hasta pronto!")
        return

# Función para procesar preguntas a Workie (Gemini)
def procesar_pregunta_workie(message):
    pregunta = message.text
    respuesta = hablar_con_workie(pregunta)
    bot.send_message(message.chat.id, respuesta)

    # Mostrar opciones para seguir conversando o finalizar
    mostrar_opciones_workie(message)

# Inicia el bot
try:
    print("Iniciando bot de Telegram...")
    bot.polling()
except Exception as e:
    print(f"Error al iniciar el bot: {e}")
```

24/11/2024
CÓDIGO FINAL

```
import telebot
import requests
import google.generativeai as genai
import os
from telebot import types

# Configurar las llaves API de Google Gemini y Telegram
try:
    os.environ["API_KEY"] = "AIzaSyBmEsrcoQO36YNfPGizkdjIo1IVqaPzHUg"
    genai.configure(api_key=os.environ["API_KEY"])
except Exception as e:
    print(f"Error al configurar la API de Gemini: {e}")
    exit()

# Crear el modelo de Gemini
try:
    model = genai.GenerativeModel('gemini-1.5-flash-latest')
except Exception as e:
    print(f"Error al crear el modelo de Gemini: {e}")
    exit()

# Token de tu bot de Telegram
TOKEN = "7972576195:AAFApKCt351YIzZzb3yjRkGCrqZVCGzAHrU"
bot = telebot.TeleBot(TOKEN)

# URL de la API de Remotive
API_URL = "https://remotive.com/api/remote-jobs"

# Diccionario de áreas de trabajo y localidades disponibles para los botones
AREAS_DE_TRABAJO = ['Engineering', 'Design', 'Product', 'Sales', 'Marketing']
LOCALIDADES = ['Remote', 'United States', 'Europe', 'Asia', 'Africa']

# Variables para almacenar la selección de búsqueda
seleccion_localidad = None
seleccion_area_trabajo = None

# Variables para almacenar los trabajos mostrados
trabajos_mostrados = []
trabajos_mostrados_totales = []  # Para almacenar todos los trabajos mostrados

# Función que hace la búsqueda de empleos y muestra los primeros 5 resultados
def buscar_empleos(area_trabajo, localidad, mostrar_nuevos=True):
    global trabajos_mostrados, trabajos_mostrados_totales
    params = {
        'category': area_trabajo,
```

```
        'location': localidad
    }

    try:
        response = requests.get(API_URL, params=params)
        response.raise_for_status()  # Lanza un error si la respuesta no es exitosa (status 200)
        data = response.json()
        jobs = data.get('jobs', [])

        # Filtramos para mostrar trabajos que no hayan sido mostrados previamente
        if mostrar_nuevos:
            nuevos_trabajos = [job for job in jobs if job not in trabajos_mostrados_totales]
        else:
            nuevos_trabajos = jobs

        if nuevos_trabajos:
            result = ""
            for i, job in enumerate(nuevos_trabajos[:5]):
                result += f"{i + 1}. Título: {job['title']}\n"
                result += f"   Empresa: {job['company_name']}\n"
                result += f"   Localidad: {job['candidate_required_location']}\n"
                result += f"   Enlace: {job['url']}\n"
                result += "-" * 5 + "\n"
            # Actualizamos la lista de trabajos mostrados
            trabajos_mostrados_totales.extend(nuevos_trabajos[:5])
            trabajos_mostrados = trabajos_mostrados_totales  # Guardamos todos los trabajos
mostrados
            return result
        else:
            return "No se encontraron nuevos empleos para esta búsqueda.\n"
    except requests.exceptions.RequestException as e:
        print(f"Error al hacer la solicitud a la API de Remotive: {e}")
        return "Hubo un error al conectar con la API de empleos. Intenta de nuevo.\n"

# Función para interactuar con Gemini
def hablar_con_workie(pregunta):
    try:
        respuesta = model.generate_content(pregunta)
        return respuesta.text
    except Exception as e:
        print(f"Error al generar contenido con Gemini: {e}")
        return "Hubo un error al interactuar con Workie. Intenta de nuevo.\n"

# Función que muestra las opciones después de una respuesta de Workie
def mostrar_opciones_workie(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    markup.add(types.KeyboardButton("Seguir hablando con Workie"))
    markup.add(types.KeyboardButton("Buscar nuevos empleos"))
```

```
    markup.add(types.KeyboardButton("Finalizar el workbot"))

    bot.send_message(
        message.chat.id,
        "¿Qué te gustaría hacer ahora?",
        reply_markup=markup
    )

# Función que muestra las opciones de búsqueda después de obtener resultados de
trabajos
def mostrar_opciones(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    markup.add(types.KeyboardButton("Hablar con Workie"))
    markup.add(types.KeyboardButton("Buscar más empleos"))
    markup.add(types.KeyboardButton("Realizar una búsqueda nueva"))
    markup.add(types.KeyboardButton("Finalizar el workbot"))

    bot.send_message(
        message.chat.id,
        "¿Qué te gustaría hacer ahora?",
        reply_markup=markup
    )

# Función que maneja el comando /start y muestra los botones de selección de localidad
@bot.message_handler(commands=['start'])
def enviar_bienvenida(message):
    # Crear un teclado con botones de localidades
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    localidad_buttons = [types.KeyboardButton(localidad) for localidad in LOCALIDADES]
    markup.add(*localidad_buttons)

    bot.send_message(
        message.chat.id,
        "¡Bienvenido! Elige una localidad para buscar empleos:",
        reply_markup=markup
    )

    # Guardamos el paso actual (localidad) para usarlo en la siguiente fase
    bot.register_next_step_handler(message, handle_localidad_selection)

# Función que maneja la selección de localidad
def handle_localidad_selection(message):
    global seleccion_localidad
    seleccion_localidad = message.text

    # Crear un teclado con botones de áreas de trabajo
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True, one_time_keyboard=True)
    area_buttons = [types.KeyboardButton(area) for area in AREAS_DE_TRABAJO]
```

```
    markup.add(*area_buttons)

    bot.send_message(
        message.chat.id,
        f"Has seleccionado la localidad: {seleccion_localidad}. Ahora selecciona un área de
trabajo:",
        reply_markup=markup
    )

    # Guardamos la localidad seleccionada para usarla después
    bot.register_next_step_handler(message, handle_area_selection)

# Función que maneja la selección de área de trabajo
def handle_area_selection(message):
    global seleccion_area_trabajo
    seleccion_area_trabajo = message.text

    # Realizar la búsqueda de empleos con la localidad y el área de trabajo seleccionados
    result = buscar_empleos(seleccion_area_trabajo, seleccion_localidad)
    bot.send_message(message.chat.id, result)

    # Mostrar opciones para continuar la conversación
    mostrar_opciones(message)

# Función que maneja las opciones seleccionadas por el usuario
@bot.message_handler(func=lambda message: message.text in ["Hablar con Workie",
"Buscar más empleos", "Realizar una búsqueda nueva", "Finalizar el workbot"])
def manejar_opciones(message):
    if message.text == "Hablar con Workie":
        # Iniciar conversación con Workie
        bot.send_message(message.chat.id, "¡Hola! ¿En qué puedo ayudarte hoy?")
        bot.register_next_step_handler(message, procesar_pregunta_workie)

    elif message.text == "Buscar más empleos":
        # Mostrar otros 5 trabajos con la misma búsqueda, sin repetir los ya mostrados
anteriormente
        result = buscar_empleos(seleccion_area_trabajo, seleccion_localidad,
mostrar_nuevos=True)
        bot.send_message(message.chat.id, result)

        # Mostrar opciones nuevamente
        mostrar_opciones(message)

    elif message.text == "Realizar una búsqueda nueva":
        # Volver a empezar la búsqueda
        bot.send_message(message.chat.id, "¡Qué bueno! ¿En qué localidad?")

        # Crear el teclado de localidades para elegir nuevamente
```

```
        markup = types.ReplyKeyboardMarkup(resize_keyboard=True,
one_time_keyboard=True)
        localidad_buttons = [types.KeyboardButton(localidad) for localidad in LOCALIDADES]
        markup.add(*localidad_buttons)

        bot.send_message(message.chat.id, "Elige una localidad para comenzar.",
reply_markup=markup)
        bot.register_next_step_handler(message, handle_localidad_selection)

    elif message.text == "Finalizar el workbot":
        # Finalizar la conversación
        bot.send_message(message.chat.id, "Gracias por usar el bot. ¡Hasta pronto!")
        return

# Función para procesar preguntas a Workie (Gemini)
def procesar_pregunta_workie(message):
    pregunta = message.text
    respuesta = hablar_con_workie(pregunta)
    bot.send_message(message.chat.id, respuesta)

    # Mostrar opciones para seguir conversando o finalizar
    mostrar_opciones_workie(message)

# Función que maneja la opción de seguir hablando con Workie
@bot.message_handler(func=lambda message: message.text == "Seguir hablando con
Workie")
def seguir_hablando_con_workie(message):
    # Responder con la frase de continuación
    bot.send_message(message.chat.id, "¡Genial! ¿En qué más te puedo ayudar?")
    bot.register_next_step_handler(message, procesar_pregunta_workie)

# Inicia el bot
try:
    print("Iniciando bot de Telegram...")
    bot.polling()
except Exception as e:
    print(f"Error al iniciar el bot: {e}")




# Función para interactuar con Gemini (con reintentos y manejo de tiempo)
def hablar_con_workie(pregunta, usuario_id):
    try:
        if usuario_id not in historial_conversacion:
            historial_conversacion[usuario_id] = []

        # Limitar el historial a 3 interacciones
```

```
        if len(historial_conversacion[usuario_id]) >= 3:
            historial_conversacion[usuario_id].pop(0)

        # Añadir la pregunta al historial
        historial_conversacion[usuario_id].append(f"Usuario: {pregunta}")

        # Crear el contexto para la conversación
        contexto = "\n".join(historial_conversacion[usuario_id])

        # Intentar obtener la respuesta de Gemini con un límite de tiempo
        timeout = 10  # Tiempo máximo de espera en segundos
        start_time = time.time()
        while time.time() - start_time < timeout:
            try:
                respuesta = model.generate_content(contexto)
                # Añadir la respuesta de Workie al historial
                historial_conversacion[usuario_id].append(f"Workie: {respuesta.text}")
                return respuesta.text
            except Exception as e:
                print(f"Error al interactuar con Gemini: {e}")
                time.sleep(2)  # Intentar nuevamente después de 2 segundos
        # Si no se recibe respuesta en el tiempo límite
        return "Lo siento, parece que Workie está teniendo problemas para responder. Intenta de nuevo más tarde. 🕒"
    except Exception as e:
        print(f"Error al generar contenido con Gemini: {e}")
        return "Hubo un error al interactuar con Workie. Intenta de nuevo más tarde. 😞"
```

Acá el bot funciona correctamente, filtra bien, busca de nuevo, se interactúa con Gemini, da recomendaciones y está totalmente correcto.

Se quiere seguir implementando mejoras a futuro.