

LLM w użyciu – automatyczne podsumowywanie konwersacji email

Kto nie słyszał o rewolucji w AI dziejącej się na naszych oczach? Znalezienie w branży IT takiej osoby wydaje się dziś niemożliwe. Obecnie AI kojarzy się jednak przede wszystkim z usługami chmurowymi dostarczanych przez takie firmy jak OpenAI lub Midjourney, co rodzi uzasadnione obawy dotyczące bezpieczeństwa przy powierzaniu im danych wrażliwych. W tym artykule przedstawię przykład użycia samodzielnie hostowanego LLM (ang. Large Language Model – Duży Model Językowy) do ułatwienia konsumpcji dużych ilości wiadomości przekazywanych przez e-mail, np. w środowisku korporacyjnym.

I ZARYS PROJEKTU

Ostateczną intencją projektu jest utworzenie programu, który przyspieszy codzienną organizację skrzynki pocztowej, na którą spływają wiadomości z różnych list mailingowych, notyfikacje z serwisów takich jak ZenDesk czy GitHub oraz bezpośrednie konwersacje e-mailowe, jednocześnie nie narażając danych wrażliwych na wyciek i/lub złamanie NDA.

Cel ten chcę osiągnąć, wykorzystując LLM od francuskiej firmy Mistral AI o nazwie kodowej *Mistral-7B-Instruct-v0.2* (modele typu *instruct* przeznaczone są do czatów, modele „zwykłe” dokończają podany tekst). Do zdobycia treści z poczty użyję API Gmail, z którym będę się komunikował za pomocą języka Ruby. Aby uzyskać zamierzony efekt – bezpieczeństwo danych wrażliwych – wszystko będę uruchamiał albo na moim własnym sprzęcie, albo na sprzęcie, nad którym mam kontrolę.

Hosting będzie się odbywał lokalnie, z użyciem serwera o parametrach przedstawionych poniżej (najważniejsza jest karta graficzna z odpowiednio dużą ilością pamięci RAM):

- » Procesor i7-12900K
- » 128GB DDR5 RAM
- » GeForce RTX 4090 24GB VRAM

I Serwowanie LLM

Aby udostępniać Mistral-7B, wykorzystamy gotowy obraz Dockera dostarczony przez jego twórców [0]. W dokumentacji proponują oni wywołanie długiej komendy wiersza poleceń (Listing 0), ale zdecydowanie wygodniejszym rozwiązaniem jest wykorzystanie narzędzia docker-compose. Plik konfiguracyjny dla docker-compose dostosowany do mojej maszyny testowej znajduje się w Listingu 1.

Listing 0. Wywołanie Dockera wg. dokumentacji Mistral (źródło: [0])

```
docker run --gpus all \
-e HF_TOKEN=$HF_TOKEN -p 8000:8000 \
ghcr.io/mistralai/mistral-src/vllm:latest \
--host 0.0.0.0 \
--model mistralai/Mistral-7B-Instruct-v0.2
```

Listing 1. Plik docker-compose.yml

```
version: '3'
services:
  mistral-vllm:
    image: ghcr.io/mistralai/mistral-src/vllm:latest
    command:
      - "--host"
      - "0.0.0.0"
      - "--model"
      - "mistralai/Mistral-7B-Instruct-v0.2"
    environment:
      - HF_TOKEN=[token z Hugging Face]
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
    ports:
      - "3001:8000"
```

Możemy teraz uruchomić serwis z modelem Mistral za pomocą komendy docker compose up -d. Aby sprawdzić jego działanie, użyjemy komendy docker compose logs -f, która wyświetli logi danej instancji Dockera. Przykład poprawnego działania znajduje się w Listingu 2.

Listing 2. Przykładowe poprawne uruchomienie usługi

```
> docker compose up -d
[+] Running 1/1
✓ Container mistral-mistral-vllm-1 Started 0.1s
> docker compose logs -f
mistral-vllm-1 | The HF_TOKEN environment variable set,
logging to Hugging Face.
[...]
mistral-vllm-1 | Login successful
[...]
tokenizer_config.json: 100%|██████████|
1.46k/1.46k [00:00<00:00, 3.25MB/s]
tokenizer.model: 100%|██████████|
493k/493k [00:00<00:00, 4.69MB/s]
tokenizer.json: 100%|██████████|
1.80M/1.80M [00:00<00:00, 3.22MB/s]
special_tokens_map.json: 100%|██████████|
72.0/72.0 [00:00<00:00, 192kB/s]
model-00003-of-00003.safetensors: 100%|██████████|
4.54G/4.54G [02:02<00:00, 37.0MB/s]
model-00001-of-00003.safetensors: 100%|██████████|
4.94G/4.94G [02:03<00:00, 39.9MB/s]
model-00002-of-00003.safetensors: 100%|██████████|
5.00G/5.00G [02:06<00:00, 39.5MB/s]
[...]
```

```
mistral-vllm-1 | INFO: Started server process [1]
mistral-vllm-1 | INFO: Waiting for application startup.
mistral-vllm-1 | INFO: Application startup complete.
mistral-vllm-1 | INFO: Uvicorn running on
http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Kontener dockera serwuje API zgodne z specyfikacją OpenAI [1], przynajmniej w zakresie, który nas interesuje na potrzeby tego projektu – będziemy w nim używać tylko API `chat.completions`.

I OBSŁUGA POCZTY

Google Workspace, a więc i Gmail, jest *de facto* standardem w biznesie, tak przynajmniej wynika z mojego doświadczenia. Dlatego też użyję Gmail API do pobierania poczty. Dokumentacja API dostarczona przez Google jest dość ogólnikowa [2] i rozrzucona po wielu różnych miejscach, ale wystarczy odrobina syndromu sztokholmskiego, aby przez nią przebrnąć.

I OAuth

Dostęp do danych wrażliwych przez API od Google wymaga autoryzacji za pomocą protokołu OAuth. Do jego obsługi użyjemy biblioteki `Signet`, używając trybu z użyciem kodu autoryzacyjnego.

Listing 3. Implementacja klasy `GoogleOAuth`

```
class GoogleOAuth
  REDIRECT_URI = 'urn:ietf:wg:oauth:2.0:oob'

  def initialize creds_fn = 'creds.json'
    @creds_fn = creds_fn
    x = begin
      JSON.parse(
        File.read(@creds_fn),
        symbolize_names: true
      ).fetch(:installed, {})
    rescue
      print "Client ID: "
      client_id = STDIN.gets.chomp
      print "Client Secret: "
      client_secret = STDIN.gets.chomp
      data = {
        installed: {
          client_id: client_id,
          client_secret: client_secret,
        }
      }
      File.write @creds_fn, JSON.pretty_generate(data)
      data[:installed]
    end

    @client_id = x[:client_id]
    @client_secret = x[:client_secret]
    @refresh_token = x[:refresh_token]
  end

  def get_oauth_token
    unless @refresh_token
      get_refresh_token
    end

    client = HTTPClient.new

    url = 'https://www.googleapis.com/oauth2/v3/token'
    params = {
      client_id: @client_id,
      client_secret: @client_secret,
      refresh_token: @refresh_token,
      grant_type: 'refresh_token'
    }

    param_str = params.map{ "#{_1}=#{"#{_2}" } }.join('&')
    final_url = "#{url}?#{param_str}"
```

```
    r = HTTPClient.post final_url, ''

    ret = JSON.parse(r.body)
    ret['access_token']

  end

  def get_auth_token
    client = Signet::OAuth2::Client.new(
      authorization_uri:
        'https://accounts.google.com/o/oauth2/auth',
      token_credential_uri:
        'https://oauth2.googleapis.com/token',
      client_id: @client_id,
      client_secret: @client_secret,
      scope: [
        'https://www.googleapis.com/auth/gmail.modify'
      ],
      redirect_uri: 'urn:ietf:wg:oauth:2.0:oob',
      access_type: 'offline',
    )

    puts "Go to #{client.authorization_uri} and " +
      "paste the code:"
    auth_code = STDIN.gets

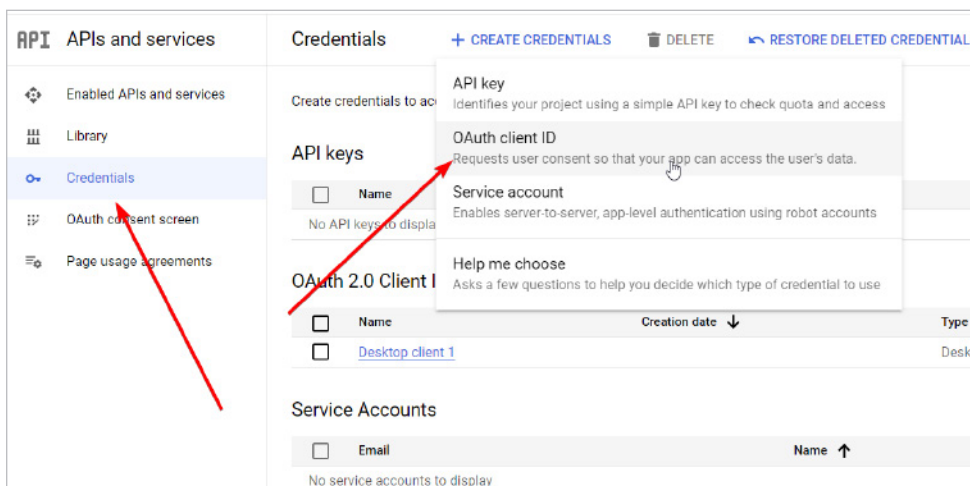
  end

  def get_refresh_token
    auth_code = get_auth_token
    client = Signet::OAuth2::Client.new(
      token_credential_uri:
        'https://www.googleapis.com/oauth2/v3/token',
      code: auth_code,
      client_id: @client_id,
      client_secret: @client_secret,
      redirect_uri: 'urn:ietf:wg:oauth:2.0:oob',
      grant_type: 'authorization_code'
    )
    client.fetch_access_token!
    @refresh_token = client.refresh_token

    data = JSON.parse File.read(@creds_fn)
    data['installed']['refresh_token'] = @refresh_token
    File.write @creds_fn, JSON.pretty_generate(data)
    @refresh_token
  end
end
```

W tym miejscu warto zauważyć, że w przypadku niezalezienia odpowiednich informacji w pliku json użytkownik będzie odpytany o wartości `client_id` i `client_secret`. Są to wartości, które uzyskamy po utworzeniu projektu w *Google Cloud Console* [3], a następnie po wygenerowaniu danych poświadczających. Na przykładzie projektu *Mail Helper* (Rysunek [0]) dodawanie nowego klienta przedstawiono na Rysunku 1. *Client secret* można uzyskać po wejściu w szczegóły dodanego klienta.

Rysunek 0. Zakładanie projektu *Mail Helper*



Rysunek 1. Tworzenie client ID

I Gmail API

Obsługa API w naszym przypadku sprowadzi się do pozyskania listy wiadomości (`users.messages.list`) [4] oraz danych konkretnej wiadomości (`users.messages.get`) [5]. Klasa Gmail, której użyjemy do komunikacji z API Google, znajduje się w Listingu 4.

Listing 4. Klasa Gmail do obsługi API Gmail

```
class Gmail
  def initialize google
    @google = google
    @c = HTTPClient.new
    renew_token
  end

  def renew_token
    @auth_token = @google.get_oauth_token
  end

  def api_call api, method: 'GET', query: nil, body: nil,
               user_id: 'me'
    tries = 0
    begin
      url = "https://gmail.googleapis.com/gmail/v1/" +
            "users/#{user_id}/#{api}"
      headers = {
        'Authorization': "Bearer #{@auth_token}",
        'Content-Type': 'application/json',
      }
      r = @c.request method, url, query, body, headers
      if r.code >= 400
        puts "Error #{r.code}: #{r.body}"
        raise '401' if r.code == 401
        return nil
      end
      JSON.parse r.body, symbolize_names: true
    rescue RuntimeError => e
      tries += 1
      if e.message == '401' and tries <= 3
        renew_token
        retry
      end
    end
  end
end
```

I KOMUNIKACJA Z LLM

Protokół oferowany przez Mistral AI jest zgodny z OpenAI, wobec czego użyjemy biblioteki `ruby-openai`. Na jej podstawie zbudujemy prostą klasę pozwalającą na rozmowę z LLM – na kształt ChatGPT (nie należy jednak oczekiwać podobnej jakości).

Listing 5. Klasa AIChat

```
class AIChat
  attr_reader :client
  attr_accessor :temperature
  attr_accessor :messages
  attr_accessor :system_prompt

  def initialize(client, temperature: 0.7)
    @client = client
    @temperature = temperature
    @system_prompt = "The following is a conversation " +
      "with an AI assistant. The assistant is " +
      "helpful, creative, clever, very friendly, " +
      "and is very laconic in his answers.\n\n"
    self.reset
  end

  def reset
    @messages = []
  end

  def chat prompt, stream: false
    if @messages.empty?
      @messages << {
        role: "user",
        content: system_prompt + prompt
      }
    else
      @messages << {
        role: "user",
        content: prompt
      }
    end

    message = ""
    if stream
      old_stream = stream
      stream = proc do |chunk|
        delta = chunk.dig("choices", 0, "delta")
        if delta and delta["content"]
          message += delta["content"]
          old_stream.call delta["content"]
        end
      end
    end

    response = @client.chat( parameters: {
      model: MODEL,
      messages: @messages,
      temperature: @temperature,
      stream: stream,
    })
    message = response
      .dig("choices", 0, "message", "content")
      .strip unless stream
    @messages << { role: "assistant", content: message }
    return message
  end
end
```

W celach testowych można zaimplementować prostego konsolowego czatbota (Listing 6). Użycie parametru `stream` pozwala na podgląd generowanych treści na żywo. Przykład z użycia tego czatbota znajduje się na Rysunku 2.

Listing 6. Implementacja czatbota w oparciu o model językowy oraz kod z Listingu 5

```
if __FILE__ == $0
  chat = AIChat.new(client)

  loop do
    print "You: ".gray
    prompt = gets.chomp
    break if prompt == "quit"
    print "\nBot: ".light_green
    stream = proc do |chunk|
      print chunk.green
    end
    response = chat.chat(prompt, stream: stream)
    puts response unless stream
    puts "\n\n"
  end
end
```

Kod zwrócony przez bota można sprawdzić w serwisie o nazwie godbolt.org [6]. Serwis ten to interaktywne narzędzie do weryfikacji kodu binarnego generowanego przez kompilatory oraz do sprawdzania jego wykonania. Wygenerowana funkcja jest zasadniczo poprawna, choć wymagałaby zmian przed oddaniem go na produkcję. Niemniej jednak jako przykład działania LLM jest znakomita.

I ANALIZA TREŚCI E-MAILI

Mamy już wszystkie elementy potrzebne do złożenia w całość naszego projektu. Zakładając, że narzędzie będzie używane codziennie, skupię się na sprawdzeniu e-maili z ostatnich 24 godzin, które nie mają statusu odczytane.

Listing 7. Funkcja `get_messages`

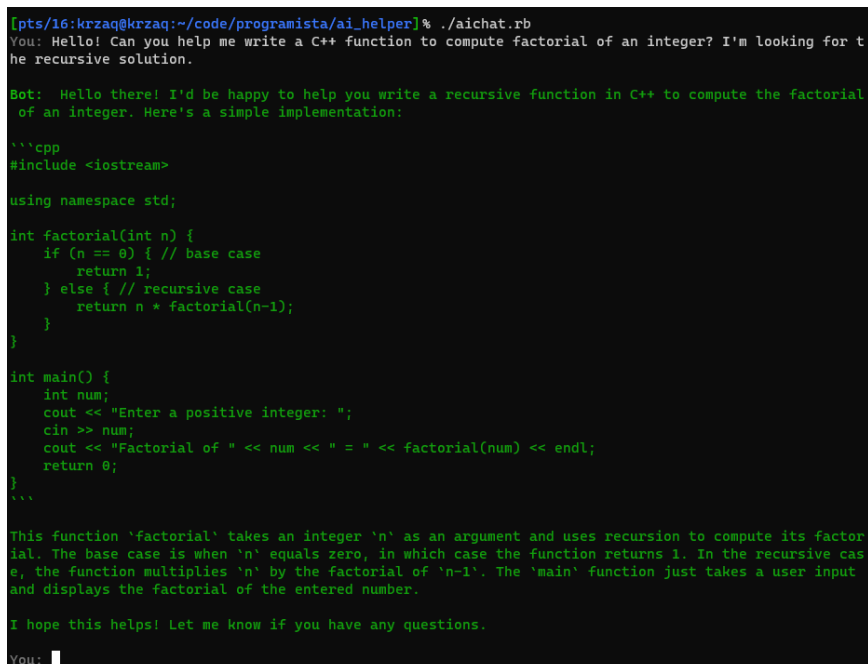
```
def get_messages
  time_threshold = (Time.now - 24 * 60 * 60).to_i
  result = M.api_call 'messages', query: {
    q: "in:inbox is:unread after:#{time_threshold}",
  }
  result[:messages]
end
```

Następnie dla każdej wiadomości ekstrahujemy jej treść, a dalej prosimy LLM o pomoc w jej interpretacji. Na wynik znaczący wpływ ma zapytanie (ang. *prompt*), które zostaje przekazane do naszej instancji Mistrala – *Prompt engineering* to dość dynamicznie rozwijająca się branża.

Po pozyskaniu streszczenia danych prosimy jeszcze o informację, czy wiadomość jest dla nas w jakikolwiek sposób interesująca. W moim przypadku nieistotne są np. przypomnienia o spotkaniach lub wiadomości marketingowe. Jeśli wiadomość powinna zwrócić naszą uwagę, pokazujemy jej tytuł i streszczenie użytkownikowi.

Listing 8. Kod wiążący treść e-maila z AI

```
get_messages[0..].each do |message|
  message_id = message[:id]
  result = M.api_call "messages/#{message_id}",
    query: { format: 'full' }
  subject = result[:payload][:headers]
    .find{ |x| x[:name] == 'Subject' }[:value]
  sender = result[:payload][:headers]
    .find{ |x| x[:name] == 'From' }[:value]
  message_text = if result[:payload][:parts]
    result[:payload][:parts].map do |part|
      msg, mime_type = parse_message_part part,
        message: message
    end.join("\n")
  else
    msg, mime_type =
      parse_message_part result[:payload],
        message: message
    msg
  end
end
```



```
[pts/16:krzak@krzak:~/code/programista/ai_helper]% ./aichat.rb
You: Hello! Can you help me write a C++ function to compute factorial of an integer? I'm looking for the recursive solution.

Bot: Hello there! I'd be happy to help you write a recursive function in C++ to compute the factorial of an integer. Here's a simple implementation:

'''cpp
#include <iostream>

using namespace std;

int factorial(int n) {
  if (n == 0) { // base case
    return 1;
  } else { // recursive case
    return n * factorial(n-1);
  }
}

int main() {
  int num;
  cout << "Enter a positive integer: ";
  cin >> num;
  cout << "Factorial of " << num << " = " << factorial(num) << endl;
  return 0;
}
'''

This function 'factorial' takes an integer 'n' as an argument and uses recursion to compute its factorial. The base case is when 'n' equals zero, in which case the function returns 1. In the recursive case, the function multiplies 'n' by the factorial of 'n-1'. The 'main' function just takes a user input and displays the factorial of the entered number.

I hope this helps! Let me know if you have any questions.

You: 
```

Rysunek 2. Przykład użycia czatbota z Listingu 6

