Zadania kwalifikacyjne — Budowanie Agentów AI

Wstęp

Zadania kwalifikacyjne służą dwóm celom: po pierwsze, pozwalają nam sprawdzić, czy masz podstawowe umiejętności potrzebne na warsztatach; po drugie, są okazją do nauki konceptów, które będą przydatne podczas warsztatów. Każde zadanie zawiera linki do materiałów, które pomogą Ci poznać potrzebne pojęcia.

Nie musisz rozwiązać wszystkiego idealnie — chodzi o to, byś pokazał/a, że potrafisz samodzielnie szukać informacji i pisać kod. Jeśli utkniesz, skorzystaj z podanych linków.

Odnośnie użycia AI. Choć warsztaty nazywają się "Budowanie Agentów AI", to nie oznacza, że możesz użyć AI do ich rozwiązania. Wszystkie te zadania AI rozwiąże bez problemu i użycie go do tego celu mija się z celem. AI jest świetnym narzędziem do nauki i może działać jako bardziej zawansowana przeglądarka internetu. I jeżeli będzie ono użyte w ten sposób może pómóć w nauce. Naprzykład:

- "Co to jest logit w kontekście llm-ow?" - *Dobre pytanie pozwalające się czegoś nauczyć*
- Wklejenie całego zadania w lla - *Zły pomysł, który uniemożliwi Ci naukę i zastąpi twoje procesy myślowe*

Zadanie 1: Podstawy działania LLM — prawdopodobieństwo tokenów

Załóżmy, że nasz model językowy ma bardzo mały słownik tokenów i dostał prompt: "Moim zwierzakiem jest". Poniżej przedstawiono logity dla każdego tokenu:

Token	Logit (z)
"kot"	4.0
"pies"	3.5
"ptak"	1.0
"chomik"	1.0
"ryba"	0.5
"wąż"	-1.0
"jest"	-2.0
"zwierzakiem"	-3.0
"Moim"	-5.0
<eos>	-10.0

Oblicz:

a) Prawdopodobieństwo każdego tokenu przy odpowiednio dobranych temperaturach (0.5, 1.0, 2.0).

b) Prawdopodobieństwo każdego tokenu przy **temperature = 0.9** i **top-k = 2**

c) Prawdopodobieństwo każdego tokenu przy **temperature = 2.0** i **top-p = 0.9**

d) Krótko wyjaśnij (1–3 zdania) jak zmiany parametrów temperature, top-k i top-p wpływają na rozkład prawdopodobieństw tokenów.

Przydatne linki:

- Jak działają LLM — wprowadzenie: https://stanford-cs324.github.io/winter2022/lectures/introduction/
- Generowanie tekstu llm-em — wyjaśnienie: https://huggingface.co/blog/how-to-generate
- Tokenizacja: https://huggingface.co/docs/tokenizers/quicktour

Zadanie 2: Funkcje, słowniki i JSON w Pythonie

Część A — Proste słowniki

Oto odpowiedź z API modelu językowego:

```
api_response = {
  "id": "chatcmpl-9kX2mP4vL8nQ1rY3sT6uW0zA",
  "object": "chat.completion",
  "created": 1717029234,
  "model": "gpt-4o-mini",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Stolica Polski to Warszawa.",
        "refusal": null
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 28,
    "completion_tokens": 6,
    "total_tokens": 34,
    "prompt_tokens_details": {
      "cached_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0
    }
  },
  "system_fingerprint": "fp_4a8c2e1b9d"
}
```

Napisz funkcje:

a) `get_assistant_message(response)` — zwraca treść odpowiedzi asystenta (string "Stolica Polski to Warszawa.") z podanego słownika.

b) `get_total_tokens(response)` — zwraca liczbę całkowitą `total_tokens` z sekcji `usage`.

c) `get_all_roles(response)` — zwraca listę ról ze wszystkich elementów w `choices` (np. `["assistant"]`).

Część B — Zagnieżdżony JSON i function calling

```
import json

raw_json = '''
{
  "id": "chatcmpl-9mK4pR7vL2nQ8sY1xT3uW6zA",
  "object": "chat.completion",
  "created": 1717031582,
  "model": "gpt-4o-mini",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": null,
        "refusal": null,
        "tool_calls": [
          {
            "id": "call_8bF2mK9pL1nQ4rY7sT0uW3xZ",
            "type": "function",
            "function": {
              "name": "calculator",
              "arguments": "{\"expression\": \"2 + 2\"}"
            }
          },
          {
            "id": "call_3aC5dE7fG9hI1jK2lM4nO6pQ",
            "type": "function",
            "function": {
              "name": "search",
              "arguments": "{\"query\": \"pogoda Warszawa\"}"
            }
          }
        ]
      },
      "logprobs": null,
      "finish_reason": "tool_calls"
    }
  ],
  "usage": {
    "prompt_tokens": 48,
```

```
    "completion_tokens": 36,
    "total_tokens": 84,
    "prompt_tokens_details": {
      "cached_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0
    }
  },
  "system_fingerprint": "fp_7d3e9c1a5b"
}
'''
```

Napisz funkcje:

d) `parse_api_response(raw_json_string)` — parsuje powyższy JSON (jako string) i zwraca słownik Pythona.

e) `get_tool_names(parsed_response)` — zwraca listę nazw narzędzi wywołanych przez model (np. `["calculator", "search"]`).

f) `get_tool_arguments(parsed_response, tool_name)` — zwraca argumenty (jako słownik) wywołania narzędzia o podanej nazwie. Jeśli narzędzie nie zostało wywołane, zwraca `None`.
Uwaga: pole `"arguments"` jest stringiem w formacie JSON — musisz je dodatkowo sparsować!

g) `count_tool_calls(parsed_response)` — zwraca liczbę wywołań narzędzi w odpowiedzi.

Przydatne linki:

- Słowniki w Pythonie: https://docs.python.org/3/tutorial/datastructures.html#dictionaries
- Funkcje w Pythonie: https://docs.python.org/3/tutorial/controlflow.html#defining-functions
- Moduł `json`: https://docs.python.org/3/library/json.html
- Zagnieżdżone struktury danych: https://www.geeksforgeeks.org/python/python-nested-dictionary/

Zadanie 3: Komunikacja z API przez `requests`

Wykorzystaj darmowe API **JSONPlaceholder** (https://jsonplaceholder.typicode.com/), które symuluje REST API.

Napisz skrypt, który:

a) Wysyła żądanie GET pod adres `https://jsonplaceholder.typicode.com/posts/1` i wypisuje kod statusu HTTP odpowiedzi.

b) Parsuje odpowiedź jako JSON i wypisuje tytuł postu (pole `"title"`).

c) Wysyła żądanie POST pod adres `https://jsonplaceholder.typicode.com/posts` z następującym ciałem (JSON):

```
{
    "title": "Mój post",
    "body": "Treść mojego posta",
    "userId": 1
}
```

Wypisze kod statusu i `"id"` nowo utworzonego postu z odpowiedzi.

d) Obsłuży przypadek błędu — wyślij żądanie GET pod nieistniejący adres, np. `https://jsonplaceholder.typicode.` i użyj `raise_for_status()` z blokiem `try/except`, aby przechwycić błąd i wypisać przyjazny komunikat.

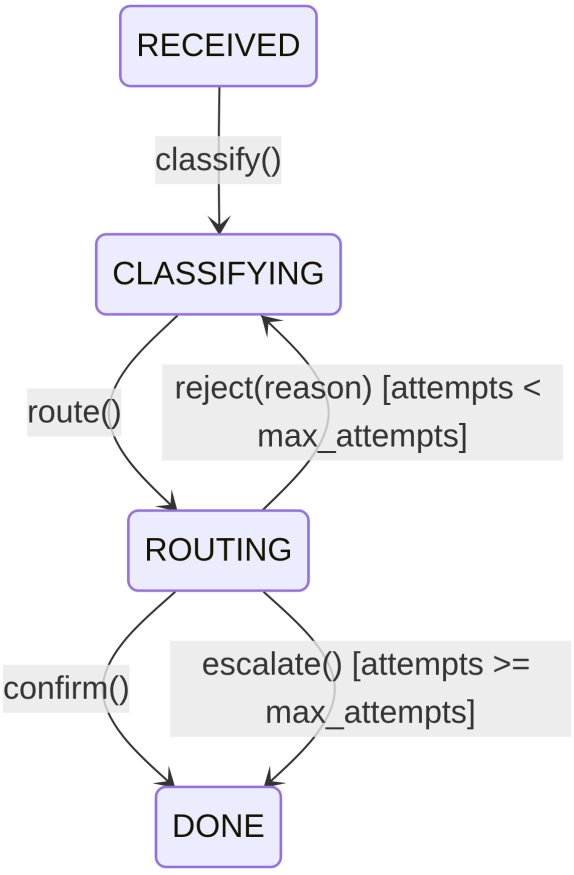
Przydatne linki:

- Biblioteka `requests`: https://requests.readthedocs.io/en/latest/user/quickstart/
- Kody statusu HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Zadanie 4: Maszyna stanowa agenta — triage

Agent triage klasyfikuje zgłoszenia użytkowników i kieruje je do odpowiedniego zespołu. Zespół może odrzucić zgłoszenie — wtedy agent klasyfikuje ponownie, mając dostęp do powodu odrzucenia. Stan maszyny jest implikowany przez kolejność wywołań akcji. Stan agenta:

```
{
  "text": "pay",
  "attempts": 0,
  "max_attempts": 2,
  "intent": "",
  "team": "",
  "feedback": [],
  "done": False
}
```



Akcje i ich wpływ na stan:

Akcja	Przejście	Zmiana stanu
`classify()`	RECEIVED → CLASSIFYING	ustawia `intent` na podstawie `text` i `feedback`
`route()`	CLASSIFYING → ROUTING	ustawia `team` na podstawie `intent` (z mapowania poniżej)
`confirm()`	ROUTING → DONE	ustawia `done=True`

Akcja	Przejście	Zmiana stanu
`reject(reason)`	ROUTING → CLASSIFYING	zwiększa `attempts` o 1, dodaje `reason` do `feedback`
`escalate()`	ROUTING → DONE	ustawia `done=True`, `team="escalation"`

Mapowanie intencji → zespół: `"payment"` → `"billing"`, `"refund"` → `"billing"`, `"support"` → `"tech"`, `"other"` → `"general"`.

Prosty algorytm klasyfikacji (heurystyka słów kluczowych):

```
KEYWORDS = {
    "payment": ["pay", "płatność", "faktura", "invoice"],
    "refund": ["refund", "zwrot", "oddanie"],
    "support": ["help", "pomoc", "błąd", "error", "nie działa"],
}

def classify(text, feedback):
    if feedback:
        search_in = " ".join(feedback).lower()
        for intent, keywords in KEYWORDS.items():
            if any(kw in search_in for kw in keywords):
                return intent
    search_in = text.lower()
    for intent, keywords in KEYWORDS.items():
        if any(kw in search_in for kw in keywords):
            return intent
    return "other"
```

a) Agent startuje ze stanem jak wyżej (`text="pay"`, `max_attempts=2`). Jaki będzie stan po sekwencji: `classify() → route() → confirm()`? Wypisz pełny stan po każdej akcji.

b) Agent startuje z `text="pay"`, `max_attempts=2`. Po `classify() → route()` zespół billing odrzuca zgłoszenie: `reject("to nie jest płatność, to pytanko techniczne")`. Agent klasyfikuje ponownie z nowym feedbackiem i tym razem ustala `intent="support"`. Wypisz pełny stan po każdej akcji. Ile wynosi `attempts` na końcu?

c) Agent startuje z `text="???"`, `max_attempts=1`. Po `classify() → route()` zespół odrzuca zgłoszenie: `reject("źle skierowane")`. Co się stanie — agent spróbuje ponownie, czy eskaluje? Uzasadnij na podstawie grafu i wartości `attempts`/`max_attempts`.

d) Zaprojektuj lepszy algorytm klasyfikacji (np. ważenie słów kluczowych, obsługa negacji, łączenie `text` + `feedback`). Opisz, dlaczego Twoje podejście jest lepsze.

Przydatne linki:

- Maszyny stanowe — wprowadzenie: https://en.wikipedia.org/wiki/Finite-state_machine
- Maszyny stanowe w Pythonie (tutorial): https://python-course.eu/applications-python/finite-state-machine.php
- Typy z `TypedDict`: https://docs.python.org/3/library/typing.html#typing.TypedDict

---

Powodzenia! Pamiętaj — jeśli czegoś nie wiesz, podane linki to dobre miejsce na start. Na warsztatach nie będziemy uczyć się podstaw Pythona od zera, ale chętnie pomogę, jeśli masz pytania do zadań.