Wydział Fizyki i Informatyki Stosowanej

Sprawozdanie z projektu

LLM-based Puzzle Solver

Deep Learning with CUDA

Autorzy:

Weronika Wronka

Karolina Klimek

Miejsce:

Kraków

Data:

29 maja 2025

Spis treści

1 Wstęp

Celem projektu *LLM-based Puzzle Solver* jest stworzenie agenta opartego na dużym modelu językowym (LLM), który potrafi rozwiązywać zagadki logiczne oraz matematyczne, wykorzystując zaawansowane techniki przetwarzania języka naturalnego.

Przykładowe zadania, które agent powinien rozwiązać, to między innymi: *„John jest starszy niż Mary, Mary jest starsza niż Tom. Kto jest najstarszy?"*.

Projekt zakłada, że agent będzie wyposażony w następujące funkcjonalności:

- interpretacja zadania i jego dekompozycja na logiczne kroki (technika *chain-of-thought prompting*),
- generowanie i porównywanie wielu możliwych rozwiązań,
- wybór najlepszego rozwiązania na podstawie dokładności i spójności logicznej.

W ramach projektu porównane zostaną różne strategie rozwiązywania problemów. Miedzy innymi:

- **Few-shot prompting** – rozwiązanie z podanymi przykładami,
- **Chain-of-Thought (CoT)** – rozumowanie krok po kroku.

Dodatkowo przeprowadzona zostanie analiza skuteczności każdej z powyższych strategii na wybranym zestawie zagadek logicznych.

Do realizacji projektu wykorzystane zostaną następujące technologie:

- **Ollama** – lokalne uruchamianie modeli LLM, takich jak Mistral,
- **Python** – zarządzanie promptami, logika agenta oraz ewaluacja rozwiązań

2 Zbiory danych

Na potrzeby projektu przygotowano zbiór danych zawierający łącznie 150 zagadek logicznych i matematycznych. Każdy rekord w zbiorze opisany jest trzema atrybutami:

- **puzzle** – treść zagadki lub pytania w języku polskim,
- **answer** – prawidłowa odpowiedź,
- **difficulty** – poziom trudności zagadki (oznaczony jako `easy`, `medium` lub `hard`).

Zagadki reprezentują różnorodne typy rozumowania:

- **Zagadki relacyjne i logiczne** – wymagające analizy relacji między osobami lub obiektami, np.:

  „Anna ma więcej jabłek niż Basia. Basia ma więcej niż Kasia. Kto ma najwięcej jabłek?"

- **Zagadki arytmetyczne i liczbowe** – testujące znajomość ciągów liczbowych i podstawowych operacji, np.:

  „Jaką liczbę należy wstawić: 2, 4, 8, 16, ?"

- **Zagadki językowe i metaforyczne** – wykorzystujące nieliteralne znaczenie słów, np.:

  „Co ma ręce, ale nie może klaskać?"

Wszystkie przykłady zostały wybrane tak, by były zrozumiałe dla dużego modelu językowego działającego w języku polskim, a jednocześnie stanowiły wyzwanie wymagające wnioskowania, uogólniania oraz zdolności do przetwarzania kontekstu.

Dzięki takiemu zróżnicowaniu zagadek możliwa jest rzetelna ocena skuteczności różnych strategii rozwiązywania problemów (few-shot, chain-of-thought), a także analiza mocnych i słabych stron implementowanego agenta.

3 Pierwsza wersja agenta: Few-shot Prompting

3.1 Założenia

W pierwszym etapie projektu stworzony został agent, który rozwiązuje zagadki logiczne i matematyczne z wykorzystaniem strategii **few-shot prompting**. Oznacza to, że każdemu zapytaniu użytkownika towarzyszy kilka przykładów rozwiązanych zadań, które mają pomóc modelowi w generalizacji i poprawnym rozumowaniu.

3.2 Architektura rozwiązania

Do implementacji agenta wykorzystano lokalnie uruchomiony model językowy za pomocą frameworka *Ollama*, w połączeniu z modelem *all-MiniLM-L6-v2* dostępnym w bibliotece *transformers*. Ogólny przepływ działania systemu obejmuje:

1. Wczytanie zbioru zagadek oraz wyodrębnienie ich treści, odpowiedzi i poziomów trudności.

2. Obliczenie embeddingów wszystkich zagadek przy użyciu modelu all-MiniLM-L6-v2.

3. Inicjalizację agenta typu `ReasoningFewShotAgent`.

4. Dla każdej nowej zagadki:

   - wybranie najbardziej podobnych zadań z bazy na podstawie podobieństwa kosinusowego,
   - zbudowanie promptu zawierającego przykładowe zagadki oraz bieżące pytanie,
   - przesłanie zapytania do modelu LLM i pobranie odpowiedzi.

Kod agenta:

Pierwsza wersja agenta została zaimplementowana w postaci klasy *ReasoningFewShotAgent*. Klasa ta przyjmuje w konstruktorze zestaw danych w postaci list tekstów zagadek, odpowiadających im odpowiedzi oraz wcześniej przygotowanych embeddingów. Dodatkowo przekazywane są komponenty modelu embeddingowego (tokenizer i model), a także konfiguracja modelu językowego uruchomionego lokalnie przez *Ollama* (nazwa modelu i adres endpointu REST API).

Kluczowym elementem działania agenta jest metoda *solve*, która integruje cały proces przetwarzania zapytania. W pierwszym kroku generowany jest embedding nowej zagadki z wykorzystaniem modelu *all-MiniLM-L6-v2*. Następnie obliczane są podobieństwa kosinusowe pomiędzy embeddingiem zapytania a embeddingami zagadek w bazie. Agent wybiera *top_k* najbardziej podobnych przykładów, które będą stanowić kontekst pomocniczy w promptcie.

Na podstawie tych przykładów budowany jest pełny prompt tekstowy. Składa się on z nagłówka z instrukcją (np. „Jesteś agentem rozwiązującym zagadki logiczne. . . ") oraz listy przykładowych zagadek i ich odpowiedzi. Po nich następuje nowa zagadka użytkownika, na którą model ma udzielić odpowiedzi. Gotowy prompt wysyłany jest do modelu LLM za pomocą zapytania POST na endpoint */api/generate*. W odpowiedzi agent otrzymuje tekst wygenerowany przez model, który następnie jest zwracany użytkownikowi.

Cała interakcja może być opcjonalnie wypisywana na ekran (przy ustawieniu *verbose=True*) – co umożliwia analizę i ocenę procesu wnioskowania modelu. W przypadku błędnej odpowiedzi serwera Ollama, agent rzuca wyjątek z kodem błędu i komunikatem.

Kod agenta jest modularny, co pozwala na łatwe modyfikacje, takie jak zmiana liczby przykładów, innego modelu embeddingowego czy alternatywnego promptu. Agent ten stanowi podstawę dla dalszych eksperymentów i porównań z innymi strategiami, np. chain-of-thought prompting.

Na rysunku poniżej przedstawiono pełny kod klasy *ReasoningFewShotAgent*:

```python
class ReasoningFewShotAgent:
    def __init__(self, tokenizer, model, db_texts, db_answers, db_embeddings, ollama_model="mistral", ollama_endpoint="http://localhost:11434"):
        self.tokenizer = tokenizer
        self.model = model
        self.db_texts = db_texts
        self.db_answers = db_answers
        self.db_embeddings = db_embeddings
        self.ollama_model = ollama_model
        self.ollama_endpoint = ollama_endpoint

    def get_query_embedding(self, query: str):
        return get_embeddings([query], self.tokenizer, self.model)[0]

    def retrieve_examples(self, query_embedding, top_k=3):
        similarities = cosine_similarity([query_embedding], self.db_embeddings)[0]
        top_indices = np.argsort(similarities)[-top_k:][::-1]
        examples = [(self.db_texts[i], self.db_answers[i]) for i in top_indices]
        return examples

    def build_prompt(self, examples, query):
        prompt = (
            "Jesteś agentem rozwiązującym zagadki logiczne.\n"
            "Twoim zadaniem jest rozwiązać zagadkę krok po kroku.\n\n"
            "Jeśli zagadka dotyczy porównania osób, przedmiotów lub wielkości – odpowiedź powinna być nazwą osoby lub rzeczy, a nie liczbą.\n"
            "Jeśli zagadka dotyczy obliczeń matematycznych – podaj liczbę jako odpowiedź.\n\n"
            "Podaj finalną odpowiedź w formacie: 'Odpowiedź: ...'\n\n"
            "Przykłady:\n"
        )
        for i, (ex_question, ex_answer) in enumerate(examples):
            prompt += f"{i+1}. Zagadka: {ex_question}\nOdpowiedź: {ex_answer}\n\n"
        prompt += f"Nowa zagadka:\n{query}\n"
        return prompt


    def query_ollama(self, prompt):
        response = requests.post(
            f"{self.ollama_endpoint}/api/generate",
            json={
                "model": self.ollama_model,
                "prompt": prompt,
                "stream": False
            }
        )

        if response.status_code != 200:
            raise Exception(f"Błąd komunikacji z Ollama: {response.status_code} {response.text}")

        return response.json().get('response', '').strip()

    def solve(self, query: str, top_k=3, verbose: bool = True):
        query_embedding = self.get_query_embedding(query)
        examples = self.retrieve_examples(query_embedding, top_k=top_k)
        prompt = self.build_prompt(examples, query)
        response = self.query_ollama(prompt)

        if verbose:
            print("🧠 Symulacja rozumowania agenta:\n")
            print(response)

        return response
```

Rysunek 1: Implementacja klasy *ReasoningFewShotAgent* wykorzystującej strategię few-shot prompting

3.3 Budowa promptu

Prompt składa się z nagłówka definiującego zasady działania agenta oraz listy kilku (domyślnie trzech) podobnych zadań wraz z poprawnymi odpowiedziami. W przykładzie poniżej przedstawiono fragment generowanego promptu:

```python
def build_prompt(self, examples, query):
    prompt = (
        "Jesteś agentem rozwiązującym zagadki logiczne.\n"
        "Twoim zadaniem jest rozwiązać zagadkę krok po kroku.\n\n"
        "Jeśli zagadka dotyczy porównania osób, przedmiotów lub wielkości – odpowiedź powinna być nazwą osoby lub rzeczy, a nie liczbą.\n"
        "Jeśli zagadka dotyczy obliczeń matematycznych – podaj liczbę jako odpowiedź.\n\n"
        "Podaj finalną odpowiedź w formacie: 'Odpowiedź: ...'\n\n"
        "Przykłady:\n"
    )
    for i, (ex_question, ex_answer) in enumerate(examples):
        prompt += f"{i+1}. Zagadka: {ex_question}\nOdpowiedź: {ex_answer}\n\n"
    prompt += f"Nowa zagadka:\n{query}\n"
    return prompt
```

Rysunek 2: Funkcja budująca prompt agenta używającego strategii few-shot.

3.4 Mechanizm doboru przykładów

W celu zapewnienia trafnych podpowiedzi, dobór przykładów odbywa się na podstawie wartości podobieństwa kosinusowego między embeddingiem nowego pytania a embeddingami z bazy. Najlepsze dopasowania są wstawiane jako przykłady.

Podobieństwo kosinusowe obliczane jest według następującego wzoru:

$$\text{cosine\_similarity}(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|} \tag{1}$$

gdzie A i B to wektory reprezentujące embeddingi dwóch tekstów. Wartość tego współczynnika zawiera się w przedziale $[-1, 1]$, gdzie 1 oznacza identyczne kierunki wektorów (największe podobieństwo), a wartości bliższe 0 lub -1 oznaczają mniejsze podobieństwo. W praktyce, dla embeddingów wygenerowanych przez modele językowe, wartości znajdują się zazwyczaj w przedziale $[0, 1]$.

W implementacji wykorzystywana jest funkcja *cosine_similarity* z biblioteki *sklearn.metrics.pairwise*, która automatycznie wykonuje to przeliczenie dla wszystkich wektorów w bazie.

3.5 Przykład działania

Poniżej przedstawiono przykładowe zapytanie do agenta oraz wynik jego działania:

> **Zapytanie:** *„Ala ma więcej cukierków niż Ola. Ola ma więcej niż Ela. Kto ma najwięcej cukierków?”*
> **Odpowiedź agenta:** *„Ala”*

```
[ ]  # Inicjalizacja ReasoningFewShotAgenta
     agent = ReasoningFewShotAgent(
         tokenizer=tokenizer,
         model=model,
         db_texts=puzzles_texts,
         db_answers=puzzles_answers,
         db_embeddings=puzzles_embeddings,
         ollama_model="mistral"
     )

     # Test na nowej zagadce
     query = "Ala ma więcej cukierków niż Ola. Ola ma więcej niż Ela. Kto ma najwięcej cukierków?"
     agent.solve(query, top_k=3, verbose=True)
```

```
🧠 Symulacja rozumowania agenta:

Odpowiedź: Ala
'Odpowiedź: Ala'
```

Rysunek 3: Przykład działania pierwszej wersji agenta.

3.6 Ocena skuteczności

Agent został przetestowany na pełnym zbiorze zagadek. Do ewaluacji wykorzystano funkcję *evaluate_agent_by_category*, która porównuje odpowiedzi modelu z odpowiedziami prawidłowymi. Wyniki są grupowane według poziomu trudności i prezentowane w formie statystyk.

Na początku każda odpowiedź zwracana przez model jest normalizowana – usuwane są znaki specjalne oraz sformułowania typu „Odpowiedź:", a sam tekst jest sprowadzany do małych liter. Służy do tego funkcja *normalize_text*, co pozwala zminimalizować wpływ różnic w formatowaniu odpowiedzi.

Następnie funkcja *evaluate_answers* porównuje odpowiedzi agenta z prawidłowymi odpowiedziami dla każdego zapytania. Dodatkowo może wypisywać wyniki działania modelu, jeśli ustawiony zostanie tryb verbose. Jeśli odpowiedź modelu jest zgodna z oczekiwaną (po normalizacji), licznik trafień zostaje zwiększony.

Funkcja *evaluate_agent_by_category* dzieli zbiór danych według poziomu trudności (np. *easy*, *medium*, *hard*) i uruchamia ocenę skuteczności osobno dla każdej grupy. Dzięki temu można przeanalizować, jak model radzi sobie z różnymi klasami zagadek.

Wynikiem działania funkcji jest słownik zawierający liczbę poprawnych i błędnych odpowiedzi w każdej kategorii trudności. Może on zostać użyty do wizualizacji danych w formie wykresu słupkowego.

Poniżej przedstawiono kod wyżej wymienionej funkcji:

```python
import re
from collections import defaultdict

def normalize_text(text):
    text = re.sub(r'[^\w\s]', '', text)
    text = text.strip().lower()
    text = text.replace("odpowiedź", "").strip()
    return text

def evaluate_answers(agent, test_queries, test_answers, verbose) -> tuple:
    correct = 0
    for query, answer in zip(test_queries, test_answers):
        answer = normalize_text(answer.strip())
        if verbose:
            print(f'\n\nQuery is: {query}\t Correct answer is: {answer}')
        response = normalize_text(agent.solve(query, top_k=3, verbose = verbose).strip())
        if response == answer:
            print("Answer was correct :)")
            correct += 1
        else:
            print(f'Response:[{response}] vs answer[{answer}]')
    return correct, len(test_queries) - correct

def evaluate_agent_by_category(agent, puzzles_df, verbose = True):
    difficulties = puzzles_df['difficulty'].unique()

    accuracy_score = defaultdict(lambda: (int, int))
    for difficulty in difficulties:
        print(f"Processing difficulty: {difficulty}")
        category_rows = puzzles_df[puzzles_df['difficulty'] == difficulty]

        test_queries = category_rows["puzzle"].tolist()
        test_answers = category_rows["answer"].tolist()
        accuracy_score[difficulty] = evaluate_answers(agent, test_queries, test_answers, verbose)

    return accuracy_score

accuracies = evaluate_agent_by_category(agent, puzzles_df)
```

Rysunek 4: Kod funkcji oceniającej działanie agenta.

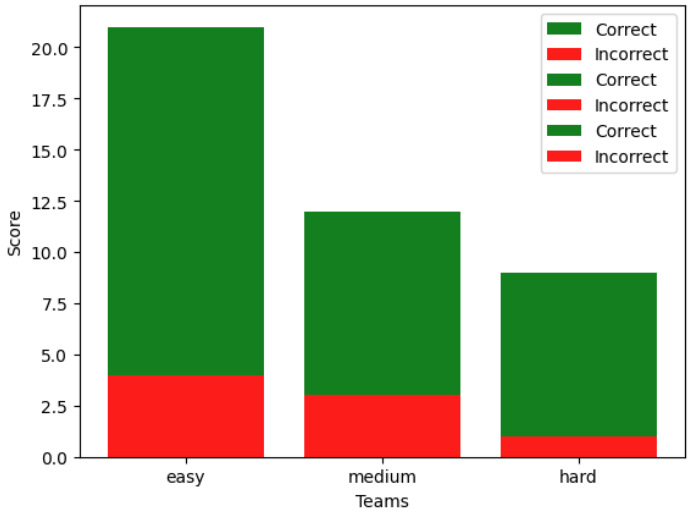
3.7 Wyniki działania

Wyniki zostały przedstawione zarówno tekstowo, jak i graficznie. Przykład jednej z analiz pokazuje poprawność odpowiedzi dla kilku testowych zagadek:

```
Query is: Jeden kwiat potrzebuje 2 litrów wody. Ile litrów na 4 kwiaty?  Correct answer is: 8
🧠 Symulacja rozumowania agenta:

Odpowiedź: 8
Answer was correct :)


Query is: W klasie są 24 dzieci. Ile rąk razem?  Correct answer is: 48
🧠 Symulacja rozumowania agenta:

28 (każda osoba ma po dwie ręce)
Response:[28 każda osoba ma po dwie ręce] vs answer[48]


Query is: Kuba kupił 3 jabłka za 6 zł. Ile kosztuje jedno?      Correct answer is: 2
🧠 Symulacja rozumowania agenta:

Odpowiedź: 2
Answer was correct :)


Query is: Która większa: suma 5 i 6 czy iloczyn 2 i 6?   Correct answer is: iloczyn
🧠 Symulacja rozumowania agenta:

Odpowiedź: Iloczyn
Answer was correct :)


Query is: Olek ma mniej książek niż Paweł, a Paweł mniej niż Kasia. Kto ma najwięcej książek?    Correct answer is: kasia
🧠 Symulacja rozumowania agenta:

Odpowiedź: Kasia
Answer was correct :)
```

Rysunek 5: Odpowiedzi agenta dla kilku przykładowych zagadek.

Ostateczny wykres porównujący liczbę poprawnych i błędnych odpowiedzi dla każdej kategorii trudności prezentuje się następująco:



Rysunek 6: Wykres przedstawiający liczbę poprawnych i błędnych odpowiedzi agenta z podziałem na poziom trudności zagadek.

3.8 Podsumowanie

Pierwsza wersja agenta osiągnęła bardzo dobre rezultaty w przypadku zagadek łatwych, gdzie większość odpowiedzi była prawidłowa, a liczba błędów znikoma. Dla zagadek o

średnim poziomie trudności model również radził sobie dobrze, choć odnotowano nieco więcej błędów, co może wynikać z większej złożoności językowej lub logicznej tych pytań.

Co istotne, agent poradził sobie zaskakująco dobrze także z zagadkami trudnymi – mimo oczekiwanej większej liczby pomyłek, uzyskano stosunkowo wysoki wskaźnik poprawnych odpowiedzi, przy jedynie minimalnej liczbie błędów. Może to świadczyć o tym, że dobrane przykłady oraz odpowiednio sformułowany prompt skutecznie wspomagają wnioskowanie nawet w bardziej złożonych przypadkach.

Mechanizm few-shot prompting, oparty na doborze najbardziej podobnych zagadek w przestrzeni embeddingów, okazał się skuteczną metodą wspomagającą rozumowanie dużego modelu językowego. Dobór przykładów pozwala modelowi generować bardziej trafne odpowiedzi.

4 Alternatywne metody promptowania

W celu poprawy jakości odpowiedzi generowanych przez agenta, zaimplementowano i przetestowano kilka alternatywnych metod wspierających proces rozumowania. Obejmują one strategię *Chain-of-Thought*, selekcję przykładów w oparciu o zróżnicowanie treści, dostosowanie do poziomu trudności, a także podejście hybrydowe łączące różne techniki.

4.1 Chain-of-Thought Prompting

Strategia *Chain-of-Thought (CoT)* rozszerza tradycyjny prompt o instrukcje, które nakazują agentowi przeprowadzenie wieloetapowego rozumowania. Jej celem jest nie tylko uzyskanie prawidłowej odpowiedzi, ale także pełnego uzasadnienia, jak agent do niej doszedł.

W implementacji utworzono klasę *ChainOfThoughtAgent*, która dziedziczy po bazowym agencie few-shot. Klasa ta nadpisuje metodę *build_prompt*, modyfikując strukturę promptu tak, aby wymusić rozumowanie krok po kroku. Główne instrukcje wprowadzone w nowym nagłówku promptu to: dokładna analiza problemu, podział na kroki cząstkowe, rozwiązanie każdego z nich osobno oraz wyciągnięcie ostatecznego wniosku.

Do każdego zapytania dołączane są również przykłady zagadek wraz z gotowymi rozwiązaniami, co pomaga modelowi lepiej zrozumieć oczekiwany sposób prezentacji odpowiedzi. Na końcu promptu agent otrzymuje nową zagadkę oraz polecenie, by rozpoczął rozumowanie „krok po kroku”.

W poniższym fragmencie kodu przedstawiono pełną strukturę klasy *ChainOfThoughtAgent*, wraz z definicją metody *build_prompt*. Warto zwrócić uwagę, że agent dziedziczy wszystkie pozostałe funkcje z klasy nadrzędnej, co pozwala na łatwą integrację tej strategii z istniejącym systemem.

```python
class ChainOfThoughtAgent(ReasoningFewShotAgent):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def build_prompt(self, examples, query):
        prompt = (
            "Jesteś agentem rozwiązującym zagadki logiczne.\n"
            "Twoim zadaniem jest rozwiązać zagadkę krok po kroku, pokazując swoje rozumowanie.\n\n"
            "Ważne:\n"
            "1. Najpierw przeanalizuj problem dokładnie\n"
            "2. Rozbij problem na mniejsze kroki\n"
            "3. Rozwiąż każdy krok osobno\n"
            "4. Wyciągnij ostateczny wniosek\n\n"
            "Jeśli zagadka dotyczy porównania osób, przedmiotów lub wielkości – odpowiedź powinna być nazwą osoby lub rzeczy, a nie liczbą.\n"
            "Jeśli zagadka dotyczy obliczeń matematycznych – podaj liczbę jako odpowiedź.\n\n"
            "Podaj finalną odpowiedź w formacie: 'Odpowiedź: ...'\n\n"
            "Przykłady:\n"
        )

        for i, (ex_question, ex_answer) in enumerate(examples):
            prompt += f"{i+1}. Zagadka: {ex_question}\nOdpowiedź: {ex_answer}\n\n"

        prompt += f"Nowa zagadka:\n{query}\n\nKrok po kroku rozumowanie:\n"
        return prompt
```

Rysunek 7: Implementacja klasy `ChainOfThoughtAgent` z wieloetapowym rozumowaniem (Chain-of-Thought).

Dzięki takiemu podejściu agent był w stanie generować odpowiedzi krok po kroku, co ułatwia analizę logiki, umożliwia identyfikację potencjalnych błędów i zwiększa przejrzystość procesu rozumowania. W dalszej części projektu strategia ta została zestawiona z innymi metodami doboru przykładów oraz oceniona pod kątem skuteczności.

4.2 Różnorodne strategie wyboru przykładów

W celu poprawy jakości przykładów dostarczanych modelowi w promptcie, przygotowano klasę pomocniczą *FewShotStrategies*. Zawiera ona trzy metody selekcji przykładów:

1. **most_similar** – klasyczne podejście wybierające najbardziej podobne zagadki na podstawie embeddingów,
2. **diverse_sampling** – strategia zapewniająca równowagę między podobieństwem a różnorodnością,
3. **difficulty_based** – wybór przykładów o poziomie trudności zbliżonym do zapytania.

Poniżej przedstawiono kod tej klasy:

```python
class FewShotStrategies:
    @staticmethod
    def most_similar(query_embedding, db_embeddings, db_texts, db_answers, top_k=3):
        """Standard similarity-based selection (what you already have)"""
        similarities = cosine_similarity([query_embedding], db_embeddings)[0]
        top_indices = np.argsort(similarities)[-top_k:][::-1]
        return [(db_texts[i], db_answers[i]) for i in top_indices]

    @staticmethod
    def diverse_sampling(query_embedding, db_embeddings, db_texts, db_answers, top_k=3, diversity_factor=0.5):
        """Select diverse examples while maintaining relevance"""
        similarities = cosine_similarity([query_embedding], db_embeddings)[0]

        selected_indices = [np.argmax(similarities)]
        selected_embeddings = [db_embeddings[selected_indices[0]]]

        remaining_indices = list(set(range(len(db_embeddings))) - set(selected_indices))

        while len(selected_indices) < top_k and remaining_indices:
            sim_to_query = similarities[remaining_indices]

            sim_to_selected = cosine_similarity(
                db_embeddings[remaining_indices],
                np.array(selected_embeddings)
            )
            avg_sim_to_selected = np.mean(sim_to_selected, axis=1)

            combined_score = diversity_factor * sim_to_query - (1 - diversity_factor) * avg_sim_to_selected

            best_idx_pos = np.argmax(combined_score)
            best_idx = remaining_indices[best_idx_pos]


            selected_indices.append(best_idx)
            selected_embeddings.append(db_embeddings[best_idx])
            remaining_indices.remove(best_idx)

        return [(db_texts[i], db_answers[i]) for i in selected_indices]

    @staticmethod
    def difficulty_based(query_embedding, db_embeddings, db_texts, db_answers, difficulties, top_k=3, target_difficulty=None):
        """Select examples of similar difficulty level"""
        if target_difficulty is None:

            similarities = cosine_similarity([query_embedding], db_embeddings)[0]
            top_indices = np.argsort(similarities)[-5:][::-1]  # Use top 5 to vote
            difficulty_counts = {}
            for idx in top_indices:
                diff = difficulties[idx]
                difficulty_counts[diff] = difficulty_counts.get(diff, 0) + 1
            target_difficulty = max(difficulty_counts.items(), key=lambda x: x[1])[0]


        target_indices = [i for i, d in enumerate(difficulties) if d == target_difficulty]


        similarities = cosine_similarity([query_embedding], db_embeddings[target_indices])[0]

        top_k_indices = np.argsort(similarities)[-top_k:][::-1]
        selected_indices = [target_indices[i] for i in top_k_indices]

        return [(db_texts[i], db_answers[i]) for i in selected_indices]
```

Rysunek 8: Różne strategie doboru przykładów w klasie *FewShotStrategies*.

Metoda *most_similar* jest najprostszym wariantem selekcji – oblicza podobieństwo kosinusowe pomiędzy embeddingiem zapytania a wszystkimi embeddingami z bazy danych i zwraca *top_k* najbardziej zbliżonych przykładów. Jest to standardowe podejście znane z klasycznych systemów few-shot.

Metoda *diverse_sampling* wprowadza pojęcie kompromisu między trafnością a różnorodnością. Najpierw wybierany jest najbardziej podobny przykład, a następnie kolejne są dobierane w taki sposób, aby maksymalizować różnorodność względem już wybranych,

przy zachowaniu pewnej odległości embeddingowej względem zapytania. Dla każdego kandydata obliczana jest wartość ważona uwzględniająca zarówno podobieństwo do zapytania, jak i średnią odległość od już wybranych przykładów. Współczynnik *diversity_factor* pozwala dostroić ten balans.

Trzecia metoda – *difficulty_based* – umożliwia selekcję przykładów w oparciu o poziom trudności. Jeśli nie zostanie on jawnie wskazany, funkcja szacuje go na podstawie pięciu najbardziej podobnych przykładów i wybiera najczęściej występującą klasę trudności. Następnie z tej podgrupy wybierane są najbardziej zbliżone przykłady do zapytania.

Każda z metod zwraca listę par *(pytanie, odpowiedź)*, które mogą zostać użyte do wygenerowania promptu few-shot. Taka struktura umożliwia elastyczne dostosowanie strategii do kontekstu zadania i testowanie różnych scenariuszy eksperymentalnych.

4.3 Agent hybrydowy z rozszerzonymi strategiami

Wszystkie powyższe metody zostały zintegrowane w klasie *EnhancedReasoningAgent*, która rozszerza bazową klasę *ReasoningFewShotAgent*. Celem tej klasy jest umożliwienie dynamicznego wyboru strategii działania agenta, w tym zarówno metody doboru przykładów, jak i trybu rozumowania krok po kroku (Chain-of-Thought).

W konstruktorze klasy znajdują się dwa główne parametry konfiguracyjne:

- *use_chain_of_thought* – flaga określająca, czy ma być użyty rozszerzony tryb rozumowania,
- *few_shot_strategy* – strategia doboru przykładów (*most_similar*, *diverse*, *difficulty_based*).

Dodatkowe argumenty mogą być przekazane w postaci słownika *strategy_params*, np. współczynnik różnorodności lub lista trudności.

Metoda *retrieve_examples* nadpisuje bazową wersję i umożliwia dynamiczny wybór jednej z trzech strategii:

1. Jeśli wybrano *most_similar* – wykorzystywana jest klasyczna selekcja na podstawie podobieństwa.
2. Dla strategii *diverse* – wykorzystywana jest metoda zrównoważonego doboru z parametrem *diversity_factor*.
3. Jeśli ustawiono *difficulty_based* – przykłady dobierane są według poziomu trudności.

W przypadku braku dopasowania do powyższych, agent korzysta z metody bazowej.

Również metoda *build_prompt* została rozszerzona. Jeżeli aktywowany został tryb *use_chain_of_thought*, agent generuje prompt zgodny z konwencją CoT – z instrukcjami

podziału problemu na kroki. W przeciwnym razie wykorzystywana jest standardowa wersja promptu odziedziczona z klasy nadrzędnej.

Poniżej przedstawiono implementację klasy:

```python
class EnhancedReasoningAgent(ReasoningFewShotAgent):
    def __init__(self, *args, use_chain_of_thought=False, few_shot_strategy="most_similar",
                 strategy_params=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.use_chain_of_thought = use_chain_of_thought
        self.few_shot_strategy = few_shot_strategy
        self.strategy_params = strategy_params or {}

    def retrieve_examples(self, query_embedding, top_k=3):
        if self.few_shot_strategy == "most_similar":
            return FewShotStrategies.most_similar(
                query_embedding, self.db_embeddings, self.db_texts, self.db_answers, top_k)
        elif self.few_shot_strategy == "diverse":
            diversity_factor = self.strategy_params.get("diversity_factor", 0.5)
            return FewShotStrategies.diverse_sampling(
                query_embedding, self.db_embeddings, self.db_texts, self.db_answers,
                top_k, diversity_factor)
        elif self.few_shot_strategy == "difficulty_based":
            difficulties = self.strategy_params.get("difficulties", [])
            target_difficulty = self.strategy_params.get("target_difficulty", None)
            return FewShotStrategies.difficulty_based(
                query_embedding, self.db_embeddings, self.db_texts, self.db_answers,
                difficulties, top_k, target_difficulty)
        else:
            return super().retrieve_examples(query_embedding, top_k)

    def build_prompt(self, examples, query):
        if self.use_chain_of_thought:
            # Chain of thought prompt
            prompt = (
                "Jesteś agentem rozwiązującym zagadki logiczne.\n"
                "Twoim zadaniem jest rozwiązać zagadkę krok po kroku, pokazując swoje rozumowanie.\n\n"
                "Ważne:\n"
                "1. Najpierw przeanalizuj problem dokładnie\n"
                "2. Rozbij problem na mniejsze kroki\n"
                "3. Rozwiąż każdy krok osobno\n"
                "4. Wyciągnij ostateczny wniosek\n\n"
                "Jeśli zagadka dotyczy porównania osób, przedmiotów lub wielkości — odpowiedź powinna być nazwą osoby lub rzeczy, a nie liczbą.\n"
                "Jeśli zagadka dotyczy obliczeń matematycznych — podaj liczbę jako odpowiedź.\n\n"
                "Podaj finalną odpowiedź w formacie: 'Odpowiedź: ...'\n\n"
                "Przykłady:\n"
            )

            for i, (ex_question, ex_answer) in enumerate(examples):
                prompt += f"{i+1}. Zagadka: {ex_question}\nOdpowiedź: {ex_answer}\n\n"

            prompt += f"Nowa zagadka:\n{query}\n\nKrok po kroku rozumowanie:\n"
        else:
            return super().build_prompt(examples, query)

        return prompt
```

Rysunek 9: Agent z obsługą strategii CoT i wyboru przykładowych zagadek.

Dzięki tej konstrukcji możliwe jest łatwe testowanie różnych wariantów działania agenta bez konieczności modyfikowania kodu klasy.

4.4 Generowanie przykładów typu CoT

Chcąc wygenerować reprezentatywne przykłady do treningu lub promptowania z wykorzystaniem CoT (Chain-of-Thought), użyto funkcji *create_cot_examples*, która automatycznie buduje listę zagadek wraz z ich szczegółowymi rozwiązaniami. Poniżej przedstawiono jej kod:

```python
[ ]  def create_cot_examples(agent, puzzles_df, num_examples=5):
        """Generate chain-of-thought examples using the base agent"""
        cot_examples = []

        for difficulty in puzzles_df['difficulty'].unique():
            category_rows = puzzles_df[puzzles_df['difficulty'] == difficulty].sample(min(2, sum(puzzles_df['difficulty'] == difficulty)))

            for _, row in category_rows.iterrows():
                query = row['puzzle']
                answer = row['answer']

                reasoning = agent.solve(query, verbose=False)


                if not reasoning.strip().endswith(answer):
                    reasoning += f"\nOdpowiedź: {answer}"

                cot_examples.append((query, reasoning))

                if len(cot_examples) >= num_examples:
                    break

        return cot_examples
```

Rysunek 10: Funkcja tworząca przykłady z wyjaśnieniem (CoT).

Funkcja przyjmuje trzy argumenty: agenta rozwiązującego, pełny zbiór danych oraz liczbę oczekiwanych przykładów. Na początku tworzona jest pusta lista *cot_examples*, w której będą zapisywane wyniki.

Iteracja odbywa się po wszystkich unikalnych poziomach trudności, a z każdej kategorii losowane są dwa przykłady (lub mniej, jeśli kategoria jest mniejsza). Dla każdego wiersza z danej kategorii tworzona jest para *(query, answer)*, po czym agent uruchamiany jest w trybie rozwiązywania bez wyświetlania kroków (*verbose=False*).

Jeśli odpowiedź wygenerowana przez agenta nie kończy się poprawną odpowiedzią (czyli model nie wypisał jawnie „Odpowiedź: ...”), to zostaje ona dołączona ręcznie na końcu rozwiązania. Umożliwia to zachowanie spójności formatowania danych wejściowych dla dalszego uczenia lub testowania.

Proces trwa do momentu zebrania żądanej liczby przykładów CoT, po czym funkcja zwraca wynikową listę. Tak przygotowane dane można następnie wykorzystać w promptach few-shot lub fine-tuningu modeli językowych.

4.5 Porównanie strategii

W celu porównania skuteczności poszczególnych metod opracowano funkcję *compare_agent_strategies*, która przeprowadza ocenę jakości działania agentów na wspólnym zestawie testowym. Zbiór danych dzielony jest na część treningową i testową w proporcji 80/20. Następnie z danych treningowych obliczane są embeddingi oraz tworzonych jest pięciu różnych agentów:

- **Base Agent** – klasyczny few-shot agent bez dodatkowych strategii,
- **Chain of Thought Agent** – agent z włączonym rozumowaniem krok po kroku,

- **Diverse Examples Agent** – agent wybierający przykłady z uwzględnieniem różnorodności,
- **Difficulty-Based Agent** – agent dopasowujący przykłady do poziomu trudności zapytania,
- **Combined Approach Agent** – agent łączący strategię CoT z doborem zróżnicowanych przykładów.

Poniżej przedstawiono kod funkcji definiującej i porównującej działanie agentów:

```python
def compare_agent_strategies(puzzles_df, tokenizer, model, puzzles_embeddings, verbose=False):
    from sklearn.model_selection import train_test_split

    train_df, test_df = train_test_split(puzzles_df, test_size=0.2, random_state=42)

    train_texts = train_df['puzzle'].tolist()
    train_answers = train_df['answer'].tolist()
    train_embeddings = get_embeddings(train_texts, tokenizer, model)

    test_queries = test_df["puzzle"].tolist()
    test_answers = test_df["answer"].tolist()

    base_agent = ReasoningFewShotAgent(
        tokenizer=tokenizer,
        model=model,
        db_texts=train_texts,
        db_answers=train_answers,
        db_embeddings=train_embeddings,
        ollama_model="mistral"
    )


    cot_agent = EnhancedReasoningAgent(
        tokenizer=tokenizer,
        model=model,
        db_texts=train_texts,
        db_answers=train_answers,
        db_embeddings=train_embeddings,
        ollama_model="mistral",
        use_chain_of_thought=True
    )


    diverse_agent = EnhancedReasoningAgent(
        tokenizer=tokenizer,
        model=model,
        db_texts=train_texts,
        db_answers=train_answers,
        db_embeddings=train_embeddings,
        ollama_model="mistral",
        few_shot_strategy="diverse",
        strategy_params={"diversity_factor": 0.7}
    )
```

Rysunek 11: Kod funkcji *compare_agent_strategies* tworzącej i testującej różne typy agentów.

```
difficulty_agent = EnhancedReasoningAgent(
    tokenizer=tokenizer,
    model=model,
    db_texts=train_texts,
    db_answers=train_answers,
    db_embeddings=train_embeddings,
    ollama_model="mistral",
    few_shot_strategy="difficulty_based",
    strategy_params={"difficulties": train_df['difficulty'].tolist()}
)


combined_agent = EnhancedReasoningAgent(
    tokenizer=tokenizer,
    model=model,
    db_texts=train_texts,
    db_answers=train_answers,
    db_embeddings=train_embeddings,
    ollama_model="mistral",
    use_chain_of_thought=True,
    few_shot_strategy="diverse",
    strategy_params={"diversity_factor": 0.7}
)

agents = {
    "Base Agent": base_agent,
    "Chain of Thought": cot_agent,
    "Diverse Examples": diverse_agent,
    "Difficulty-Based": difficulty_agent,
    "Combined Approach": combined_agent
}

results = {}

test_subset = test_queries[:min(15, len(test_queries))]
test_answers_subset = test_answers[:min(15, len(test_answers))]

for name, agent in agents.items():
    print(f"\nEvaluating {name}...")
    correct, incorrect = evaluate_answers(agent, test_subset, test_answers_subset, verbose)
    results[name] = (correct, incorrect)

return results
```

Rysunek 12: Ciąg dalszy kodu funkcji *compare_agent_strategies*.

Po zdefiniowaniu agentów, wybrana zostaje próbka testowa, a każdy agent oceniany jest na tej samej liście zapytań i poprawnych odpowiedzi. Wyniki przechowywane są w słowniku *results* w postaci krotek *(poprawne, błędne)*.

Aby ułatwić interpretację wyników, przygotowano funkcję *plot_comparison_results*, która tworzy wykres porównujący dokładność działania poszczególnych strategii. Dokładność jest obliczana jako procent poprawnych odpowiedzi z podanej liczby testów, a wartości są automatycznie wypisywane nad słupkami wykresu.

```python
def plot_comparison_results(results):
    """Plot comparative results of different strategies"""
    agent_names = list(results.keys())
    correct = [results[name][0] for name in agent_names]
    total = [results[name][0] + results[name][1] for name in agent_names]
    accuracy = [100 * results[name][0] / (results[name][0] + results[name][1]) for name in agent_names]

    fig, ax = plt.subplots(figsize=(12, 6))
    bars = ax.bar(agent_names, accuracy, color='skyblue')

    for i, bar in enumerate(bars):
        ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 1,
                f"{accuracy[i]:.1f}%\n({correct[i]}/{total[i]})",
                ha='center', va='bottom')

    ax.set_ylabel('Accuracy (%)')
    ax.set_title('Comparison of Different Reasoning Strategies')
    ax.set_ylim(0, 100)

    plt.tight_layout()
    plt.show()
```

Rysunek 13: Funkcja rysująca porównanie dokładności różnych strategii rozumowania.

Oprócz pomiarów ilościowych, zebrano również wybrane przykłady działania agentów. Przykład zapytania typu „Ile myszy złapie 15 kotów w 15 minut?" pokazuje różnice w rozumowaniu między podejściami. Agent CoT krok po kroku analizuje dane, rozbija problem i dochodzi do prawidłowego wyniku.

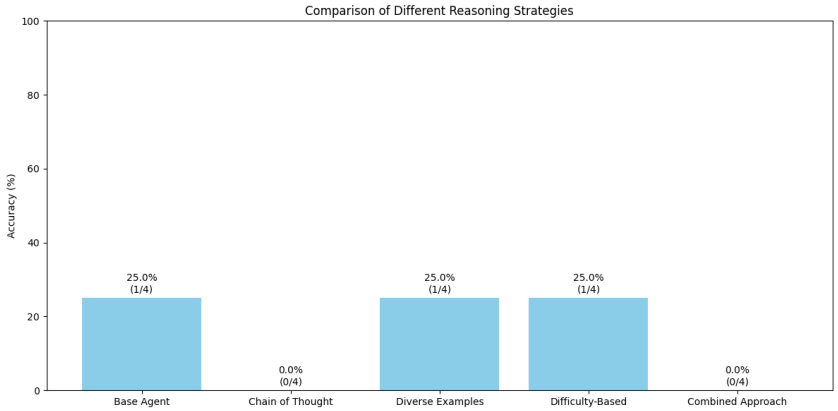
```
Chain of Thought reasoning with diverse examples:
🧠 Symulacja rozumowania agenta:

1. Analizujemy problem: 5 kotów łapią 5 myszy w 5 minut (stosunek 1:1), to oznacza, że każdy kot złapie jedną mysz w ciągu minuty.

  2. Rozbijamy problem na mniejsze kroki: w 15 minut 3 razy więcej czasu niż w 5 minutach, a także 3 razy większa liczba kotów. Zatem złapie trzy razy więcej myszy każdy z 15 kotów.

  3. Rozwiązujemy każdy krok osobno: w ciągu jednej minuty po 15 kotach zostanie złapanych 15 myszy (bo każdy kot złapie jedną mysz), a w ciągu 15 minut 15 myszy po 15 kotach (bo 3 razy więcej czasu i 3 razy większa liczba kotów).

  4. Wyciągamy ostateczny wniosek: Ile myszy złapie 15 kotów w 15 minut? Odpowiedź: 15 * 15 = 225 myszy.

Comparing different strategies...

Evaluating Base Agent...
Answer was correct :)
Response:[basia zapłaciła 6 zł] vs answer[6]
Response:[12] vs answer[7]
Response:[56 jest większa od 34] vs answer[56]

Evaluating Chain of Thought...
Response:[1 przeanalizuj problem dokładnie
     ustalone jest trzech osób tomek michał i piotr
     stwierdzono że tomek jest młodszy od michała t  m ale starszy od piotra t  p

  2 rozbij problem na mniejsze kroki
     pierwszy krok zbadaj czy tomek jest najmłodszy spośród trzech osób
     drugi krok wyznacz najstarszego spośród pozostałych dwóch osób

  3 rozwiąż każdy krok osobno
     pierwszy krok tomek nie może być najmłodszy bo jest starszy od piotra
     drugi krok jeśli tomek jest starszy od michała i piotra to michał musi być starszy od piotra m  p wtedy żeby t był starszy od m to musiałby być starszy również od p co jest sprzeczne z danymi z zadania
     trzeci krok zauważmy że w przypadku kiedy tomek miałby być starszy od michała i młodszy od piotra to ten sam konflikt występowałby wtedy gdyby michał byłby starszy od piotra m  p to musiałby być starszy również od tomeka co jest sprzeczne z danymi z zadania
     czwarty krok musimy przyjąć że tomek nie może być starszy ani od michała ani od piotra wtedy zauważmy że wtedy musi się znaleźć dwa osoby takie że jedna jest starsza od drugiej i właśnie te dwie osoby to są michał i tomek m  t ale to oznacza że tomek nie może być młodszy od piotra
     ostatni krok z powyższego wynika że tomek musi być najstarszy spośród trzech osób t  m i t  p

  4 wyciągnij ostateczny wniosek
     tomek jest najstarszy] vs answer[michał]
Response:[1 analityczne rozbijamy problem na mniejsze części w tym przypadku to liczba którą basia zapłaciła za 3 ciastka i cena poszczególnego ciasteczka

   2 rozważmy cenę pojedynczego ciasteczka i liczbę sztuk kupionych przez basię cena pojedynczego ciasteczka jest 2 zł a basia kupiła 3 sztuki

   3 obliczamy sumaryczną kwotę zapłaconą za 3 ciastka

   4 podajemy  basia zapłaciła za 3 ciastka ile 2 zł razy 3 sztuki czyli 6 zł

    basia zapłaciła 6 zł] vs answer[6]
Response:[1 analiza problemu liczba dzieci w klasie jest znana 20 podaje się też trzy grupy dzieci które lubią inne rzeczy  lody ciasto i oba
```

Rysunek 14: Przykładowe odpowiedzi i rozumowanie agentów w zależności od wybranej strategii.

4.6 Wyniki i wnioski

Na koniec, wyniki porównania przedstawione są w formie zbiorczego wykresu słupkowego. Zaskakująco, w tym zestawie testowym najlepszy wynik (25%) uzyskały trzy strategie: Base Agent, Diverse Examples oraz Difficulty-Based. Strategia Chain-of-Thought oraz podejście hybrydowe (łączące CoT i różnorodność) nie przyniosły w tym przypadku prze-

wagi, co może wynikać z niewielkiej liczby testowanych przykładów oraz sposobie weryfikacji odpowiedzi.



Rysunek 15: Porównanie dokładności działania agentów opartych o różne strategie.

Podsumowując, funkcja *compare_agent_strategies* stanowi podstawowe narzędzie do testowania i wizualnej analizy różnych wariantów architektury agenta. W kolejnych etapach projektu planowane są dalsze testy na większych zbiorach oraz weryfikowanie poprawności odpowiedzi za pomocą innego modelu językowego, a nie poprzez porównywanie stringów.

5 Ocena odpowiedzi przy pomocy innego modelu językowego

Tradycyjna ewaluacja modeli polegająca na porównywaniu generowanej odpowiedzi z oczekiwaną za pomocą dopasowania ciągów znaków nie przyniosła dobrych rezultatów. Często mimo, iż odpowiedź była poprawna była klasyfikowana jako błędna, gdyż została nieco inaczej sformuowana. Chcąc uniknąć tego problemu, dla agenta używającego promptu typu few-shot zastosowano strategię, w której osobny duży model językowy ocenia poprawność odpowiedzi agenta.

5.1 Ocena jednego zapytania.

W pierwszym kroku zdefiniowano funkcję *evaluate_with_llm*, która iteruje po parach (pytanie, odpowiedź oczekiwana), uruchamia agenta, a następnie przesyła prompt oceniający do modelu Mistral działającego lokalnie za pośrednictwem endpointu HTTP. Poniżej przedstawiono kod wspomnianej funkcji.

```python
def evaluate_with_llm(agent, test_queries, test_answers, evaluator_model="mistral:latest", verbose=False):
    """Evaluate answers using an LLM as judge instead of string comparison"""
    import requests

    correct = 0
    incorrect = 0
    results = []

    for query, answer in zip(test_queries, test_answers):
        ground_truth = answer.strip()
        if verbose:
            print(f"\n\n🧩 Query: {query}\n✅ Ground truth: {ground_truth}")

        # Model generates an answer
        response = agent.solve(query, top_k=3, verbose=verbose).strip()

        # Construct evaluation prompt
        eval_prompt = f"""
        You are an expert evaluator of puzzle solutions. Your task is to determine if a given solution correctly answers a puzzle.

        Puzzle: {query}

        Reference answer: {ground_truth}

        Model's answer: {response}

        Evaluate if the model's answer is semantically correct, even if worded differently.
        Respond with ONLY "CORRECT" or "INCORRECT"
        """

        # Get LLM-based evaluation
        eval_response = requests.post(
            "http://localhost:11434/api/generate",
            json={
                "model": evaluator_model,
                "prompt": eval_prompt,
                "stream": False
            }
        ).json().get('response', '').strip()

        words = eval_response.strip().split()
        first_word = words[0].upper() if words else ""
        is_correct = first_word == "CORRECT"
```

Rysunek 16: Kod funkcji oceniającej odpowiedzi agenta za pomocą innego modelu językowego.

```
        if verbose:
            print(f"🧠 Eval says: {eval_response}")
            print(f"🔍 Detected judgment: {first_word} -> {'CORRECT' if is_correct else 'INCORRECT'}")

        if is_correct:
            correct += 1
            status = "✅ CORRECT"
        else:
            incorrect += 1
            status = "❌ INCORRECT"

        if (status == "✅ CORRECT" and "INCORRECT" in eval_response.upper()) or \
           (status == "❌ INCORRECT" and "CORRECT" in eval_response.upper()):
            print("⚠️ WARNING: Mismatch between detected status and evaluation response!")

        result = {
            "puzzle": query,
            "ground_truth": ground_truth,
            "model_answer": response,
            "evaluation": eval_response,
            "status": status
        }
        results.append(result)

        print(f"{status} - {eval_response.splitlines()[0] if eval_response else 'No response'}")

    total = correct + incorrect
    accuracy = correct / total if total > 0 else 0
    print(f"\n📊 Final accuracy: {accuracy:.2%} ({correct}/{total})")

    return correct, incorrect, results
```

Rysunek 17: Ciąg dalszy kodu funkcji weryfikującej odpowiedzi agenta.

Prompt dla oceniającego modelu językowego zawiera oryginalne pytanie, poprawną odpowiedź oraz odpowiedź wygenerowaną przez agenta. Model ma za zadanie stwierdzić, czy odpowiedź modelu jest semantycznie poprawna, nawet jeśli została inaczej sformułowana. Odpowiedź modelu oceniającego powinna zawierać słowo „CORRECT” lub „INCORRECT”. Dodatkowo w kodzie uwzględniono mechanizm detekcji niespójnych odpowiedzi (np. zawierających oba słowa jednocześnie).

5.2 Ewaluacja według poziomu trudności.

Chcąc przeprowadzić ocenę jakości działania agenta na większym zbiorze danych, z podziałem na poziomy trudności, przygotowano funkcję *evaluate_agent_by_category_with_llm*. Funkcja ta bazuje na wcześniej opisanym mechanizmie oceny odpowiedzi przez zewnętrzny model językowy, a wyniki agregowane są osobno dla kategorii *easy*, *medium* i *hard*. Na Rysunku 18 przedstawiono kod omawianej funkcji.

```python
def evaluate_agent_by_category_with_llm(agent, puzzles_df, evaluator_model="mistral:latest", verbose=False):
    """Evaluate agent performance by difficulty category using LLM judge on all examples"""
    difficulties = puzzles_df['difficulty'].unique()
    results = {}
    all_evaluations = []

    for difficulty in difficulties:
        print(f"\n=== Processing difficulty: {difficulty} ===")
        category_rows = puzzles_df[puzzles_df['difficulty'] == difficulty]

        test_queries = category_rows["puzzle"].tolist()
        test_answers = category_rows["answer"].tolist()

        print(f"Evaluating {len(test_queries)} puzzles in {difficulty} difficulty...")

        correct, incorrect, evaluations = evaluate_with_llm(
            agent, test_queries, test_answers, evaluator_model, verbose
        )

        results[difficulty] = (correct, incorrect)
        for eval_item in evaluations:
            eval_item['difficulty'] = difficulty
            all_evaluations.append(eval_item)

    return results, all_evaluations
```

Rysunek 18: Ewaluacja agenta według poziomu trudności przy użyciu modelu Mistral jako sędziego.

W pierwszym kroku funkcji wyodrębniane są unikalne poziomy trudności z ramki danych *puzzles_df*. Dla każdej kategorii danych wykonywana jest ewaluacja:

- tworzony jest podzbiór przykładów odpowiadający danemu poziomowi trudności,
- z kolumn *puzzle* i *answer* tworzona jest lista zapytań testowych oraz oczekiwanych odpowiedzi,
- następnie uruchamiana jest funkcja *evaluate_with_llm*, która wykonuje ocenę semantyczną przy pomocy zewnętrznego modelu językowego.

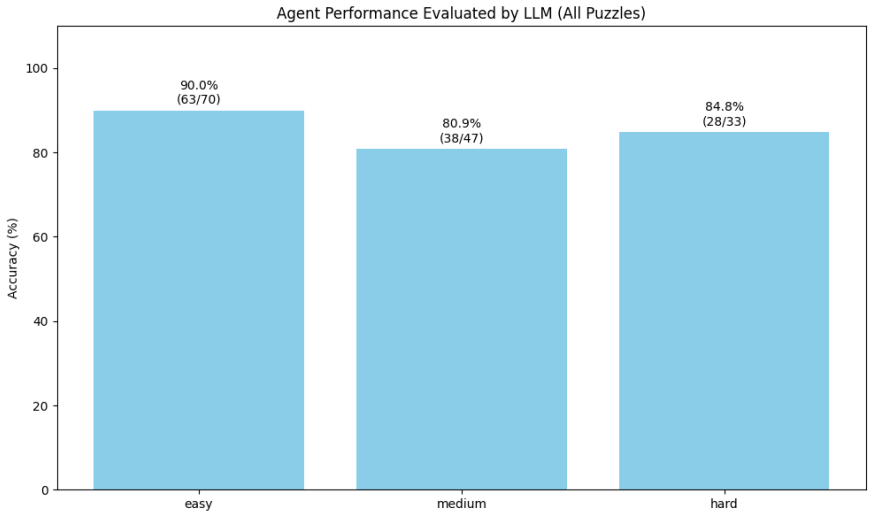
Otrzymane wyniki – liczba odpowiedzi poprawnych i błędnych – zapisywane są w słowniku *results*. Dodatkowo lista wszystkich ocen zawiera szczegółowe informacje na temat każdego przypadku testowego, łącznie z poziomem trudności, co umożliwia ich dalszą analizę i wizualizację.

Dzięki tej funkcji możliwe jest uzyskanie bardziej precyzyjnego obrazu mocnych i słabych stron agenta w zależności od złożoności zadania. Na przykład, agent może wykazywać wysoką skuteczność dla zadań łatwych, ale mieć trudności z pytaniami wymagającymi złożonego wnioskowania lub wieloetapowej analizy.

5.3 Wizualizacja wyników.

Uzyskane wyniki zostały przedstawione graficznie za pomocą wykresu słupkowego, gdzie każdy słupek reprezentuje dokładność agenta dla jednego z trzech poziomów trudności:

easy, *medium*, *hard*. Wysokość słupka odpowiada procentowemu udziałowi poprawnych odpowiedzi, a nad każdym z nich umieszczono wartość dokładności oraz liczbę trafień względem liczby wszystkich przykładów w danej kategorii.



Rysunek 19: Wyniki ewaluacji agenta według poziomu trudności ocenione przez inny model językowy.

Z wykresu możemy wyciągnąć następujące wnioski:

- **Zadania łatwe (easy)** – agent uzyskał tutaj najwyższą dokładność, wynoszącą **90.0%** (63 poprawne odpowiedzi na 70 przykładów). Oznacza to, że większość prostych zagadek była rozwiązywana skutecznie, co może wynikać z ich jednoznacznej struktury i niższego poziomu wymaganego rozumowania.
- **Zadania średnie (medium)** – skuteczność spadła do **80.9%** (38 na 47). Mimo nadal wysokiego poziomu trafności, obserwujemy pogorszenie wyników, prawdopodobnie z powodu zwiększonej złożoności językowej i logicznej tych zagadek.
- **Zadania trudne (hard)** – dokładność wyniosła **84.8%** (28 na 33), co jest nieco wyższe niż dla zadań średnich. Może to być związane z mniejszą liczbą przykładów, a także lepszym dopasowaniem kilku trudnych zagadek do stylu działania agenta (np. silnie logicznych lub obliczeniowych).

Pomimo że intuicyjnie można by się spodziewać spadku dokładności wraz ze wzrostem trudności, różnice między poziomami nie są drastyczne. Wynik ogólny na poziomie **86.0%** (129 poprawnych odpowiedzi na 150) potwierdza skuteczność zastosowanego podejścia, szczególnie w połączeniu z ewaluacją opartą o semantyczne rozumienie, a nie tylko dopasowanie ciągów znaków.

5.4 Szczegóły odpowiedzi.

Dodatkowo, aby umożliwić analizę jakościową oraz lepsze zrozumienie zachowania agenta, zaimplementowano funkcję *display_evaluation_details*. Jej zadaniem jest prezentacja przykładowych wyników oceny – zarówno poprawnych, jak i błędnych – w sposób czytelny dla użytkownika.

```python
def display_evaluation_details(evaluations, num_examples=3):
    """Display detailed examples of model evaluations"""
    import random

    correct_examples = [e for e in evaluations if "CORRECT" in e['status']]
    incorrect_examples = [e for e in evaluations if "INCORRECT" in e['status']]

    print(f"\n===== EVALUATION EXAMPLES =====")

    if correct_examples:
        print(f"\n✅ CORRECT EXAMPLES ({min(num_examples, len(correct_examples))} of {len(correct_examples)}):")
        for example in random.sample(correct_examples, min(num_examples, len(correct_examples))):
            print(f"\nPuzzle: {example['puzzle']}")
            print(f"Ground truth: {example['ground_truth']}")
            print(f"Model answer: {example['model_answer']}")
            print(f"Evaluation: {example['evaluation']}")
            print("-" * 50)

    if incorrect_examples:
        print(f"\n❌ INCORRECT EXAMPLES ({min(num_examples, len(incorrect_examples))} of {len(incorrect_examples)}):")
        for example in random.sample(incorrect_examples, min(num_examples, len(incorrect_examples))):
            print(f"\nPuzzle: {example['puzzle']}")
            print(f"Ground truth: {example['ground_truth']}")
            print(f"Model answer: {example['model_answer']}")
            print(f"Evaluation: {example['evaluation']}")
            print("-" * 50)
```

Rysunek 20: Kod funkcji prezentującej przykłady poprawnych i błędnych odpowiedzi.

Funkcja przyjmuje dwa argumenty:

- *evaluations* – listę ocen (słowników) zawierających zapytanie, odpowiedź modelu, wzorcową odpowiedź oraz ocenę LLM,
- *num_examples* – liczbę przykładów do wyświetlenia (domyślnie 3).

Na początku dane są dzielone na dwa zbiory – odpowiedzi uznane za poprawne oraz błędne – na podstawie wartości pola *status*. Następnie:

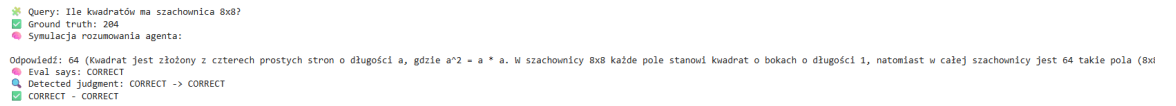
- losowane są maksymalnie *num_examples* poprawnych przykładów, z których każdy wyświetla treść zagadki, poprawną odpowiedź, odpowiedź modelu oraz ocenę modelu oceniającego;
- analogicznie losowane są przykłady błędne, wraz z tą samą informacją.

Wyniki prezentowane są w konsoli w formacie tekstowym. Dodatkowe linie z nagłówkami i separatorami poprawiają czytelność i pozwalają łatwo zidentyfikować przypadki błędów lub sukcesów.

Funkcja ta jest szczególnie przydatna przy debugowaniu lub dalszej analizie działania agenta – umożliwia zrozumienie, dlaczego dany przykład został oceniony jako niepoprawny oraz czy błąd pochodził z błędnego wnioskowania, niepełnego kontekstu, niejednoznaczności pytania czy też błędnej oceny modelu.

5.5 Przykładowy przebieg ewaluacji.

Na końcu, pokazano fragment rzeczywistego logu uruchomienia ewaluacji. Model oceniający bardzo dobrze radzi sobie z wykrywaniem semantycznej zgodności – nawet przy innym szyku zdań lub synonimach potrafi właściwie ocenić poprawność.



Rysunek 21: Fragment wyników działania modelu oceniającego i finalna dokładność.

5.6 Wnioski.

Zastosowanie modelu językowego jako niezależnego sędziego pozwala na bardziej elastyczną i wiarygodną ocenę agentów rozwiązujących zadania. W przeprowadzonych testach agent osiągnął ogólną skuteczność na poziomie **86%**, przy czym najłatwiej radził sobie z zadaniami łatwymi, a najtrudniejsze wymagały bardziej precyzyjnego rozumowania.