Dokumentacja Projektu: Prompt Optimizer

Data: 30.11.2025 **Nazwa Drużyny:** GPU Enjoyers

---

1. Zdefiniowanie problemu

Kontekst ekologiczny

Sektor IT, a w szczególności centra danych obsługujące modele sztucznej inteligencji, odpowiada za rosnące zużycie energii. Szacuje się, że do 2040 roku oprogramowanie może generować nawet 14% globalnych emisji CO2. Każde zapytanie do modelu LLM (Large Language Model) zużywa energię proporcjonalną do liczby przetwarzanych tokenów (słów/części słów).

Zidentyfikowany problem

Użytkownicy modeli językowych często tworzą nieefektywne prompty zawierające:

- Zbędne zwroty grzecznościowe ("Dzień dobry", "Proszę", "Pozdrawiam").
- Nadmiarowy kontekst, który nie wpływa na jakość odpowiedzi.
- Powtórzenia.

Model LLM przetwarza te "śmieciowe" dane, zużywając moc obliczeniową GPU i energię bez wnoszenia wartości merytorycznej.

Proponowane rozwiązanie

Stworzyliśmy **Prompt Optimizer** – warstwę pośrednią, która automatycznie optymalizuje prompty przed wysłaniem ich do modelu docelowego. Rozwiązanie wykorzystuje NLP do usuwania szumu komunikacyjnego i kondensacji treści.

Klient docelowy i korzyści biznesowe

- **Klient:** Firmy SaaS i startupy technologiczne, które intensywnie wykorzystują płatne API w swoich produktach (np. chatboty obsługi klienta, asystenci kodowania).
- **Korzyści:**
    1. **Ekologia:** Mniejszy ślad węglowy dzięki redukcji liczby tokenów przetwarzanych przez energochłonne modele w data center.
    2. **Oszczędność:** Redukcja kosztów operacyjnych (API providerzy rozliczają się za liczbę tokenów wejściowych).
    3. **Wydajność:** Krótszy czas przetwarzania zapytania (mniejsza latencja).

---

2. Analiza i przygotowanie danych

Wykorzystane zbiory danych:

1. **Nvidia HelpSteer**
    - **Źródło:** HuggingFace (https://huggingface.co/datasets/nvidia/HelpSteer).

    - **Zastosowanie:** Testowanie algorytmów na danych technicznych. Zbiór zawiera prompty i odpowiedzi dotyczące programowania. Pozwolił zweryfikować, czy optymalizacja promptu nie usuwa kluczowych fragmentów kodu lub zmiennych.
2. **Conversational Interaction Dataset**
    - **Źródło:** Kaggle, wiele małych datasetów (m. in https://www.kaggle.com/datasets/thedevastator/dailydialog-unlock-the-conversation-potential-in).
    - **Charakterystyka:** Zbiór typowych interakcji czatowych zawierający powitania, pożegnania i *small talk*.
    - **Cel:** Wytrenowanie klasyfikatora do rozpoznawania intencji "social" vs "content".

Przetwarzanie danych

- **Czyszczenie:** Usunięcie znaczników i fragmentów kodu z datasetów.
- **Tokenizacja:** Podział tekstu na zdania w celu analizy ich wagi semantycznej.
- **Analiza:** Zidentyfikowano, że średnio 15-20% tokenów w zapytaniach konwersacyjnych to "szum" (stopwords, zwroty grzecznościowe), co stanowi potencjał optymalizacyjny.

Generowanie treningowego zbioru danych

Cel i etykiety

Skrypt generuje zbalansowany zbiór krótkich fraz rozmówkowych i „treści merytorycznej”, klasyfikowanych do czterech etykiet:

- Label 0 – Greetings (powitania)
- Label 1 – Thanks (podziękowania)
- Label 2 – Goodbyes (pożegnania)
- Label 3 – Others/Hard (pozostałe treści, w tym „tricky” i zwykłe prompty)

Zbiór jest budowany tak, aby liczba przykładów dla Label 3 była zbliżona do łącznej liczby elementów klas 0–2 (balans klas).

Główne etapy:

1. Generowanie wariantów fraz dla etykiet 0–2 (różne wielkości liter oraz prosta interpunkcja dla krótkich fraz).
2. Dodanie bogatej listy przykładów dla etykiety 3 (`TRICKY_PHRASES`).
3. Opcjonalne doładowanie danych zewnętrznych do Label 3 z lokalnych plików CSV (patrz „Dane wejściowe – opcjonalne”).
4. Scalenie i deduplikacja dokładnych duplikatów (case-sensitive).
5. Zapis gotowego zbioru i wygenerowanie wykresu rozkładu etykiet.

Funkcje kluczowe:

- `generate_variants(phrase, label)` – wytwarza proste warianty danej frazy.
- `load_external_data(target_count=None)` – wczytuje dodatkowe przykłady Label 3 z plików CSV w bieżącym katalogu roboczym.
- `build_final_dataset()` – buduje kompletny, zbalansowany dataset i zwraca listę `(text, label)`.
- `plot_distribution(counts)` – zapisuje wykres rozkładu klas do pliku `dataset_distribution.png`.

Wyniki

Skrypt zapisuje w bieżącym katalogu roboczym:

- `categorized_phrases.csv` – plik CSV z kolumnami: `text`, `label`.
- `dataset_distribution.png` – wykres rozkładu liczby rekordów w poszczególnych klasach.

---

3. Zastosowanie modeli uczenia maszynowego

Architektura rozwiązania:

```
model.train()
for epoch in range(epochs):
    total_loss = 0

    for i, batch in enumerate(train_loader):
        optimizer.zero_grad()

        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )

        loss = outputs.loss
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if i % 100 == 0:
            print(f"Batch {i}, Loss: {loss.item():.4f}")

    avg_loss = total_loss / len(train_loader)
    print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")

    model.eval()

    all_preds = []
    all_labels = []

    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)
```

```
            outputs = model(input_ids, attention_mask=attention_mask)
            logits = outputs.logits

            # Predictions for this batch
            batch_preds = torch.argmax(logits, dim=1)

            # Collect all predictions and labels
            all_preds.extend(batch_preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
```

Krok A: Klasyfikacja intencji (BERT)

Wykorzystano model **BERT (Bidirectional Encoder Representations from Transformers)**, poddany procesowi fine-tuningu na zbiorze konwersacyjnym.

- **Funkcja:** Klasyfikacja wieloklasowa każdego zdania w prompcie.
- **Logika:** Jeśli zdanie jest klasyfikowane jako "Greeting/farewell/gratitude" -> odpowiedz hardcode'owaną odpowiedzią. Jeśli "prompt" -> przejdź dalej.
- **Dlaczego BERT?** Doskonale rozumie kontekst. Odróżnia powitania, pożegnania i podziękowania od elementów kodu czy istotnych nazw własnych, czego nie robią proste filtry oparte na słowach kluczowych.

Krok B: Ekstraktywna sumaryzacja (TextRank)

Dla długich promptów zastosowano algorytm **TextRank**.

Fragment kodu

```
def summarize(self, input_text, summary_percentage=0.7):
        """
        Generates the compressed summary.
        """
        original_sentences, cleaned_sentences =
self._clean_and_process_sentences(input_text)

        if not original_sentences or all(not s for s in cleaned_sentences):
            return "Cannot summarize empty or non-text content."

        total_original_words = sum(len(word_tokenize(s)) for s in
original_sentences)
        target_word_count = int(total_original_words * summary_percentage)

        if target_word_count == 0:
            target_word_count = 1

        if len(original_sentences) == 1:
            return self._handle_single_sentence_case(original_sentences,
cleaned_sentences, target_word_count)
```

```
        try:
            sentence_vectors = self.vectorizer.fit_transform(cleaned_sentences)
        except ValueError:
            return "The document contains no significant words after cleaning."

        similarity_matrix = cosine_similarity(sentence_vectors)

        graph = nx.from_numpy_array(similarity_matrix)
        scores = nx.pagerank(graph)

        ranked_sentences = {i: scores[i] for i in range(len(original_sentences))}

        sorted_ranks = sorted(ranked_sentences.items(),
                              key=lambda item: item[1],
                              reverse=True)

        # --- WORD-BASED SELECTION, COMPRESSION & RECONSTRUCTION ---
        selected_sentences_by_index = {}
        current_word_count = 0
        for index, score in sorted_ranks:
            original_sentence = original_sentences[index]

            compressed_sentence = self._compress_sentence(original_sentence)
            compressed_length = len(compressed_sentence.split())

            if current_word_count + compressed_length <= target_word_count:
                selected_sentences_by_index[index] = compressed_sentence
                current_word_count += compressed_length
            else:
                break

        if not selected_sentences_by_index and sorted_ranks:
            top_index = sorted_ranks[0][0]
            compressed = self._compress_sentence(original_sentences[top_index])
            compressed_words = compressed.split()
            if len(compressed_words) > target_word_count:
                selected_sentences_by_index[top_index] = '
'.join(compressed_words[:target_word_count])
            else:
                selected_sentences_by_index[top_index] = compressed

        final_summary_parts = []

        for index in sorted(selected_sentences_by_index.keys()):
            final_summary_parts.append(selected_sentences_by_index[index])

        return " ".join(final_summary_parts)
```

- **Funkcja:** Budowa grafu, gdzie wierzchołkami są zdania, a krawędziami ich podobieństwo semantyczne.

- **Działanie:** Wybierane są tylko zdania o najwyższej randze (najważniejsze dla kontekstu), tworząc skróconą wersję tekstu.
- **Zaleta:** TextRank jest algorytmem "lekkim" obliczeniowo w porównaniu do generowania podsumowań przez LLM, co wpisuje się w ideę *Green AI*.

Krok C: Porównanie rozwiązań

Porównaliśmy nasze podejście hybrydowe (BERT + TextRank) z brakiem preprocessingu.

- *Brak preprocessingu:* Wolne, zużywa dużo tokenów.
- *Prompt Optimizer:* Zachowuje sens semantyczny (zgodność cosine similarity > 0.9) przy redukcji tokenów o średnio 25%.

---

4. Prezentacja rozwiązania i wnioski

Sposób działania (Workflow)

1. Użytkownik wysyła zapytanie: "*Cześć, mam problem z kodem, mógłbyś zerknąć? Oto on: [Długi Kod]. Z góry dzięki!*"
2. **Prompt Optimizer API** przetwarza tekst:
    - BERT usuwa: "*Cześć, mam problem z kodem, mógłbyś zerknąć?*" oraz "*Z góry dzięki!*".
    - TextRank analizuje kod/opis i kondensuje go, jeśli jest zbyt rozwlekły.
3. Do LLM (np. GPT-4) trafia tylko: "*Analiza błędu: [Długi Kod]*".
4. LLM zwraca poprawną odpowiedź, zużywając mniej tokenów.

Wnioski końcowe

Stworzony prototyp udowadnia, że można pogodzić wysoką jakość odpowiedzi systemów AI z dbałością o środowisko. Projekt spełnia założenia hackathonu:

1. Rozwiązuje problem ekologiczny (zużycie energii przez Data Center).
2. Wykorzystuje zbiory danych i modele AI (BERT, TextRank).
3. Jest gotowy do wdrożenia jako usługa biznesowa B2B.

Bibliografia

- https://economictimes.indiatimes.com/magazines/panache/do-you-say-please-to-chatgpt-sam-altman-reveals-how-much-electricity-your-manners-cost-to-openai/articleshow/120455018.cms?utm_source=chatgpt.com&from=mdr
- https://www.techi.com/sam-altman-ai-politeness-costs-energy/?utm_source=chatgpt.com
- https://www.sciencenews.org/article/ai-energy-carbon-emissions-chatgpt?utm_source=chatgpt.com
- https://www.kaggle.com/datasets/thedevastator/dailydialog-unlock-the-conversation-potential-in
- https://aclanthology.org/W04-3252/
- https://arxiv.org/abs/1810.04805

---

5. Dodatkowa dokumentacja techniczna — mapowanie kodu i szybki start

Poniższa sekcja dodaje bezpośrednie odniesienia do plików źródłowych i instrukcje uruchomienia zachowując całą treść powyżej.

Szybki start (Windows / PowerShell)

1. Utwórz środowisko i zainstaluj zależności:

```
py -3 -m venv .venv
.\.venv\Scripts\Activate.ps1
pip install -r requirements.txt
```

2. (Opcjonalne) Wygeneruj dataset:

```
python .\datasets\dataset_gen.py
```

   Wyniki: datasets\categorized_phrases.csv i dataset_distribution.png.
3. Uruchom demo (Streamlit lub skrypt):

```
python .\main.py
```

   Uwaga: main.py używa zmiennych środowiskowych (np. GEMINI_API_KEY) oraz lokalnych artefaktów model/tokenizer (foldery "model" i "tokenizer") — sprawdź katalog projektu.

Repo layout — pliki kluczowe

- main.py
  - Streamlit UI, orchestration pipeline, generate_content_with_retry (retry/backoff), pomiary energii (codecarbon).
- utils\classifier.py
  - Klasa Classifier, ładowanie modelu DistilBERT, metoda predict().
- utils\textrank.py
  - TextRankSummarizer: _ensure_nltk_resources(), summarize(), _compress_sentence().
- utils\model.py
  - Alternatywna klasa Evaluation pokazująca lokalne wywołanie LLM
- datasets\dataset_gen.py
  - Funkcje generate_variants, load_external_data, build_final_dataset, plot_distribution.
- POC\
  - Notebooki i skrypty eksperymentalne (fine-tuning, pomiary energii).

Mapowanie komponentów do implementacji (krótko)

- Klasyfikacja intencji
  - Plik: utils/classifier.py
  - Model: lokalny folder "model" i "tokenizer" (ładowane z DistilBERT).
  - Cel: wykryć powitania/podziękowania/pożegnania i uniknąć wysyłania ich do drogiego LLM.

- Ekstraktywna sumaryzacja (TextRank + kompresja)
    - Plik: utils/textrank.py
    - Działanie: TF-IDF -> cosine_similarity -> PageRank (networkx) -> POS-based compression.
    - Zasoby NLTK: _ensure_nltk_resources() pobiera 'punkt', 'stopwords', 'averaged_perceptron_tagger' przy imporcie.
- Wywołanie LLM
    - Lokalnie: utils/model.py -> Evaluation.run_LLM()
    - Zdalnie: main.py -> generate_content_with_retry() (Gemini API, potrzebny GEMINI_API_KEY).
- Orkiestracja + pomiary
    - main.py -> Evaluation.run_optimized_LLM(), run_processing_pipeline(), kod z codecarbon do zbierania emisji.

Krótka ścieżka wykonywania (trace)

1. main.py otrzymuje raw prompt.
2. Classifier (utils/classifier.py) ocenia fragmenty; jeśli social → krótka odpowiedź bez LLM.
3. Jeśli „prompt/content" → TextRankSummarizer.summarize() (utils/textrank.py).
4. Skrócony prompt wysyłany do LLM (lokalnie lub Gemini API).
5. Wynik oraz metryki energii/pomiarów zapisywane/wyświetlane.