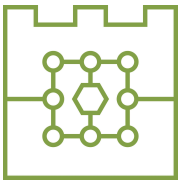
Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji



Bartosz Piechnik

Numer albumu: 139450

Udoskonalenie generowania tekstu przez duże modele językowe z wykorzystaniem technik RAG i LoRA

Improving text generation by large language models using RAG and LoRA techniques

Praca inżynierska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem:
dr Radosław Kycia

Kraków, 2025

1. WSTĘP

Duże modele językowe (LLM) zrewolucjonizowały w ostatnich latach rynek dając masę narzędzi które pomagają nie tylko w nauce ale także w życiu codziennym. Modele te uczone są na ogromnych zbiorach danych tekstowych aby odpowiedzi były jak najbardziej podobne do tych ludzkich. Niestety są one podatne na halucynację tzn. generowanie tekstu tylko po to aby zadowolić użytkownika i dopełnić odpowiedzi mimo tego, że zwracany tekst nie jest zgodny z prawdą. Wraz z rozwojem tej dziedziny, powstały metody które próbują minimalizować prawdopodobieństwo halucynacji, RAG (retrieval augmented generation) [6] i LoRA (Low – Rank Adaptation) [2]. Niniejsza praca zbada efektywność poszczególnych metod, porównując je w konkretnych dziedzinach, najlepszy z modeli zostanie wykorzystany do stworzenia prostego interfejsu użytkownika który pozwoli na integrację z modelem na lokalnej maszynie użytkownika zapewniając małe opóźnienie w generowaniu.

2. CEL ZAKRES PRACY I METODYKA

Cel

Celem niniejszej pracy jest utworzenie kodu pozwalającego na szczegółowe zbadanie efektywności różnych metod minimalizujących występowanie halucynacji w dużych modelach językowych (Large Language Models - LLM). W szczególności uwaga zostanie skupiona na dwóch podejściach: RAG (Retrieval-Augmented Generation), które łączy generowanie treści z mechanizmami wyszukiwania informacji, oraz LoRA (Low-Rank Adaptation), umożliwiającym dostosowywanie dużych modeli za pomocą efektywnych i lekkich modyfikacji. Analiza ta ma na celu wyłonienie najbardziej skutecznego podejścia do redukcji halucynacji w specyficznym zastosowaniu, jakim jest generowanie krótkich odpowiedzi na pytania użytkowników. Wybrana metoda, uznana za najefektywniejszą, zostanie wykorzystana do zaprojektowania i zaimplementowania prostego w obsłudze interfejsu użytkownika. Interfejs ten pozwoli na intuicyjną i praktyczną interakcję z modelem działającym na maszynie lokalnej, zapewniając jednocześnie niskie opóźnienie w generowaniu odpowiedzi.

Zakres

Zakres pracy obejmuje kompleksowe badania literaturowe oraz praktyczne eksperymenty związane z dużymi modelami językowymi (LLM). Na początkowym etapie zostanie przeprowadzony przegląd dostępnych materiałów naukowych dotyczących architektury i funkcjonowania LLM, z naciskiem na proces ich trenowania na rozległych zbiorach danych tekstowych. W tej części szczególny nacisk zostanie położony na zrozumienie podstawowych mechanizmów działania modeli oraz ich ograniczeń, w tym skłonności do generowania halucynacji. Omówione zostanie samo zjawisko halucynacji – jego przyczyny, mechanizmy powstawania oraz wpływ na wiarygodność i jakość generowanych odpowiedzi.

W kolejnym kroku szczegółowo scharakteryzowane zostaną wybrane metody minimalizujące halucynacje. Zostanie przeanalizowana metodologia działania RAG, która integruje generowanie tekstu z modułami wyszukiwania informacji, oraz LoRA, która wprowadza lekkie adaptacje do dużych modeli w celu dostosowania ich do specyficznych zadań. Dla obu metod zostaną przedstawione wyniki analiz literaturowych, obejmujące ich skuteczność w różnych scenariuszach zastosowań oraz ograniczenia techniczne.

W części empirycznej pracy zaplanowano implementację modeli opartych na podejściach RAG i LoRA w środowisku testowym. Eksperymenty obejmą zastosowanie tych modeli na specjalnie przygotowanych zbiorach danych reprezentujących różne dziedziny tematyczne. Wyniki zostaną ocenione pod kątem dokładności, spójności oraz wiarygodności generowanych odpowiedzi, z uwzględnieniem redukcji halucynacji. Na podstawie wyników eksperymentów zostanie wyłoniony najlepszy model, który następnie posłuży jako baza dla stworzenia klienckiej aplikacji. Aplikacja ta umożliwi użytkownikowi interakcję z modelem w przyjaznym, intuicyjnym środowisku, działającym w pełni lokalnie.

Metodyka

Prace nad projektem obejmują zarówno teoretyczne analizy, jak i praktyczne eksperymenty. Na początkowym etapie przeprowadzono przegląd literatury naukowej dotyczącej dużych modeli językowych (LLM), ze szczególnym uwzględnieniem zjawiska halucynacji – jego przyczyn, mechanizmów oraz wpływu na jakość generowanych odpowiedzi. Następnie wybrano dwa podejścia do minimalizacji halucynacji: RAG (Retrieval-Augmented Generation) oraz LoRA (Low-Rank Adaptation). Obie metody zostały szczegółowo przeanalizowane pod kątem ich mechanizmów działania, skuteczności i ograniczeń.

W części empirycznej zaimplementowano i przetestowano modele oparte na RAG i LoRA w środowisku testowym, wykorzystując specjalnie przygotowane zbiory danych. Eksperymenty skupiały się na ocenie dokładności, spójności oraz wiarygodności generowanych odpowiedzi w różnych scenariuszach tematycznych.

3. DUŻE MODELE JĘZYKOWE (LLMS)

3.1 Wprowadzenie do modeli językowych

Duży model językowy (LLM - Large Language Model) to zaawansowany algorytm sztucznej inteligencji, który specjalizuje się w przetwarzaniu i generowaniu języka naturalnego. Dzięki szkoleniu na ogromnych zbiorach danych tekstowych, takie modele są w stanie rozumieć, interpretować i tworzyć tekst w sposób naturalny i płynny.

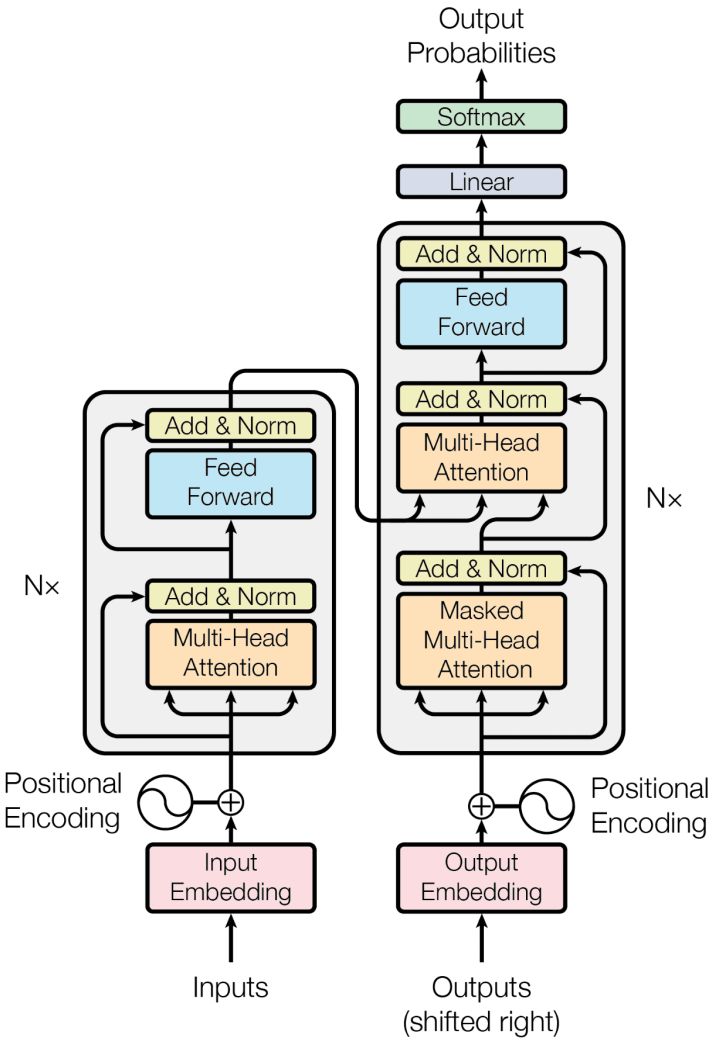
Jednym z kluczowych atutów dużych modeli językowych jest ich skalowalność, która pozwala na efektywne przetwarzanie i generowanie tekstu w różnych kontekstach. LLMy mogą tworzyć różnorodne treści – od prostych zdań, przez rozbudowane artykuły, aż po skomplikowane fragmenty kodu czy kreatywne formy, takie jak poezja. Ważnym aspektem ich działania jest umiejętność rozumienia kontekstu, co pozwala im generować spójne i sensowne odpowiedzi na zapytania użytkowników.

Modele te znajdują szerokie zastosowanie w różnych dziedzinach. Mogą być wykorzystywane w chatbotach, takich jak ChatGPT [10] czy Gemini [11], które umożliwiają prowadzenie płynnych rozmów z użytkownikami. Są też używane w tłumaczeniach maszynowych, gdzie zapewniają bardziej precyzyjne i naturalne tłumaczenia. Ponadto, LLMy służą do automatycznego generowania treści, takich jak opisy produktów, raporty czy artykuły, oraz wspierają działanie wirtualnych asystentów, takich jak Siri czy Google Assistant, ułatwiając codzienne interakcje z technologią.

3.2 Architektura

3.2.1 Transformer

Podstawą większości dużych modeli językowych jest architektura Transformera, zaprezentowana w pracy "Attention Is All You Need" [1]. Transformer przedstawiony na rys. 3.1 został zaprojektowany jako uniwersalna architektura sieci neuronowej do przetwarzania sekwencji, co sprawia, że nadaje się idealnie do zadań takich jak tłumaczenie maszynowe czy generowanie tekstu.



Rys. 3.1. Architektura Transformera (Źródło: [1])

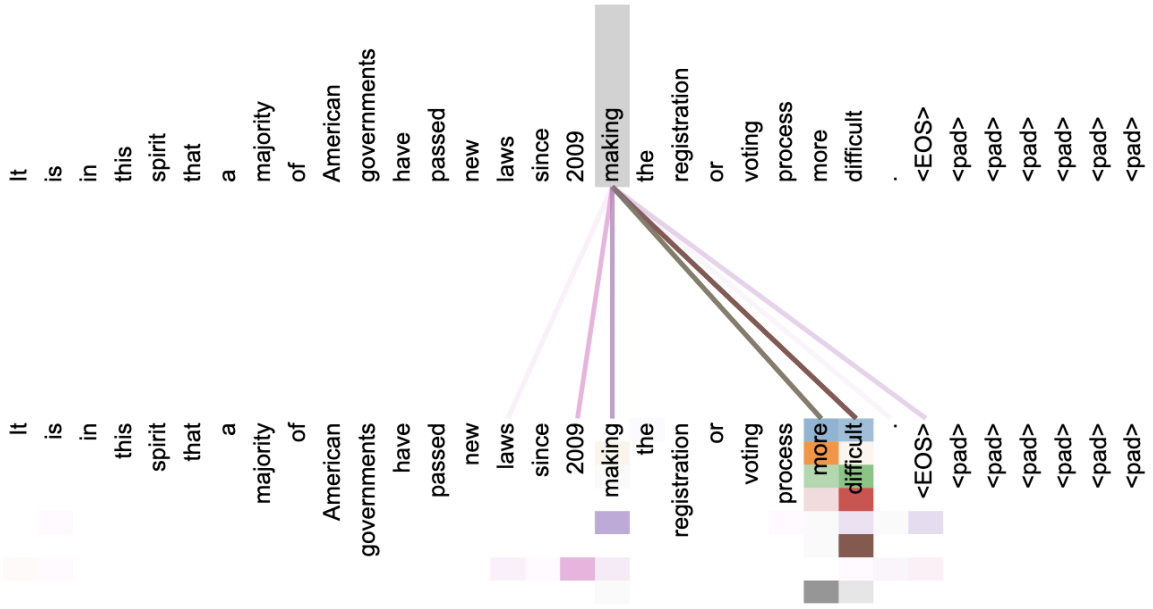
Model ten opiera się na dwóch głównych komponentach: koderze (encoder) i dekoderze (decoder). Koder przetwarza wejściową sekwencję tekstu i generuje jej reprezentację, podczas gdy dekoder przekształca tę reprezentację w wyjściowy tekst, generowany token po tokenie. Kluczową zaletą architektury Transformer jest eliminacja rekurencji, co pozwala na równoległe przetwarzanie danych, zwiększając efektywność trenowania i przetwarzania tekstu.

3.2.2 Mechanizm samo-uwagi (self-attention)

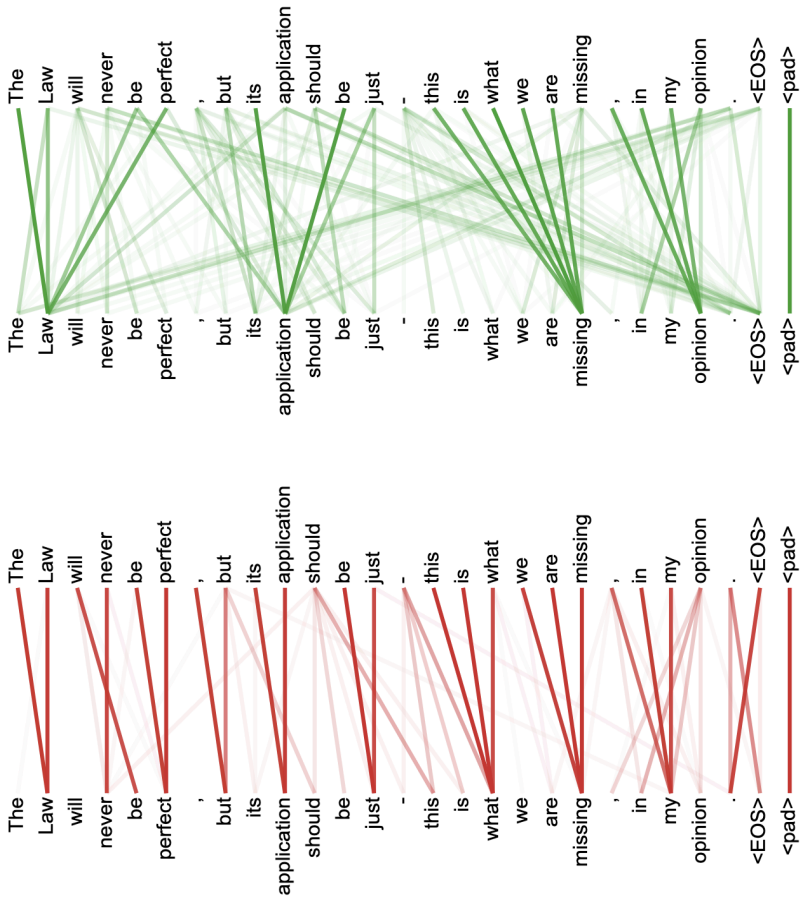
Mechanizm samo-uwagi [1] umożliwia modelowi analizowanie zależności między różnymi elementami tekstu w ramach całej sekwencji. Dzięki temu każdy fragment tekstu jest interpretowany w kontekście innych, co pozwala na uchwycenie zarówno lokalnych, jak i globalnych powiązań w danych wejściowych.

Działanie mechanizmu można streścić w kilku krokach: każdy element sekwencji jest oceniany pod kątem jego znaczenia w stosunku do innych, a następnie tworzone są nowe

reprezentacje, które odzwierciedlają ten kontekst jak przedstawiono na rys. 3.2.. Mechanizm samo-uwagi wprowadza również wariant Multi-Head Attention, który pozwala analizować dane z różnych perspektyw, co znacząco zwiększa zdolności modelu do wychwytywania bardziej złożonych zależności, ponieważ każda z ‘głów’ mechanizmu Multi-Head Attention może inaczej interpretować semantykę zdania. Na rys. 3.3 przedstawiono jak różne ‘głowy’ przypisują inne wagi tym samym słowom w zdaniu.



Rys. 3.2. Wizualizacja reprezentacji wag mechanizmu self-attention (Źródło: [1])



Rys. 3.3. Wizualizacja wag słów dla różnych ‘głów’ mechanizmu Multi-Head Attention (Źródło: [1])

3.3 Praktyczne aspekty dużych modeli językowych

3.3.1 Skalowanie i rozmiar modeli

Jednym z najbardziej charakterystycznych elementów LLM jest ich rozmiar – liczone w miliardach parametrów. Większa liczba parametrów pozwala na uchwycenie bardziej subtelnych niuansów językowych, ale wiąże się również z większym zapotrzebowaniem na zasoby obliczeniowe. Modele takie jak GPT-3 [7] czy GPT-4 [12] wymagają zaawansowanego sprzętu zarówno do trenowania, jak i do działania w czasie rzeczywistym, co ogranicza ich zastosowanie w środowiskach o niskiej mocy obliczeniowej.

3.3.2 Wyzwania związane z LLM

Pomimo ich imponujących możliwości, duże modele językowe nie są pozbawione wad. Jednym z głównych problemów są tzw. halucynacje – sytuacje, w których model generuje odpowiedzi, które są przekonujące, ale niezgodne z faktami. Przyczyny tego zjawiska tkwią w sposobie trenowania modeli, które mogą nadmiernie polegać na statystycznych zależnościach w danych treningowych. Innym wyzwaniem jest odpowiedzialne wykorzystanie modeli – ich skalę można potencjalnie wykorzystać do tworzenia dezinformacji czy innych nieetycznych zastosowań.

3.3.3 Perspektywy rozwoju

W miarę jak technologia LLM się rozwija, coraz większy nacisk kładziony jest na tworzenie bardziej efektywnych i dostępnych modeli. Innowacje takie jak LoRA (Low-Rank Adaptation) czy techniki kompresji pozwalają na dostosowywanie dużych modeli do konkretnych zadań przy mniejszym zapotrzebowaniu na zasoby. Kolejnym kierunkiem jest integracja modeli z zaawansowanymi systemami wyszukiwania informacji, takimi jak RAG (Retrieval-Augmented Generation), co pozwala na bardziej precyzyjne i wiarygodne odpowiedzi.

Te postępy sprawiają, że duże modele językowe mają przed sobą szerokie perspektywy zastosowań, od wspierania nauki i pracy, po rozwój rozrywki i technologii codziennego użytku.

4. ZJAWISKO HALUCYNACJI W DUŻYCH MODELACH JĘZYKOWYCH

Zjawisko halucynacji (czasami używany jest również termin złudzenia SI) w dużych modelach językowych (Large Language Models, LLM) to sytuacja, w której model generuje informacje, które są nieprawdziwe, niespójne z rzeczywistością lub całkowicie wymyślone. Problem ten wynika z natury modeli językowych, które opierają swoje działanie na analizie wzorców w danych treningowych, a nie na rzeczywistym „rozumieniu" kontekstu. Chociaż LLM zostały zaprojektowane do generowania tekstu w sposób przypominający komunikację ludzką, ich odpowiedzi mogą czasem być mylące, szczególnie gdy brakuje im odpowiedniego kontekstu lub kiedy dane wejściowe są niejasne. Halucynacje mogą mieć różne formy, od drobnych błędów po całkowicie nieprawdziwe informacje, co sprawia, że ich zrozumienie i minimalizacja są kluczowe w rozwoju technologii opartej na sztucznej inteligencji.

4.1 Rodzaje halucynacji

Halucynacje w modelach językowych można podzielić na kilka głównych typów. Pierwszym z nich są halucynacje faktograficzne, które występują, gdy model generuje fałszywe informacje dotyczące faktów. Na przykład model może podać, że wynalazcą teorii względności był Isaac Newton, co jest oczywistym błędem. Kolejnym rodzajem są halucynacje strukturalne, w których generowany tekst jest logicznie niespójny lub pozbawiony sensu, na przykład zdanie „Rzeka jest niebieskim ptakiem, który śpiewa pod wodą". Inną kategorią są halucynacje powiązań, w których model sugeruje nieprawdziwe relacje między elementami, jak na przykład twierdzenie, że „Marilyn Monroe była żoną Winstona Churchilla", co wynika z błędnego łączenia kontekstów.

4.2 Przyczyny halucynacji

Halucynacje są efektem kilku kluczowych ograniczeń modeli językowych. Jednym z nich jest brak dostępu do aktualnych informacji, ponieważ modele, które nie mają wbudowanego mechanizmu przeszukiwania zewnętrznych baz danych, muszą polegać na ograniczonych i statycznych danych treningowych. Kolejnym problemem jest jakość danych treningowych – jeśli dane użyte do trenowania modelu zawierają błędy, dezinformacje lub nieprawdziwe informacje, model może powielać te błędy w swoich odpowiedziach. Inną przyczyną jest statystyczny charakter działania LLM, które generują odpowiedzi na podstawie wzorców, a nie rzeczywistego zrozumienia kontekstu. Czasami błędy wynikają również z optymalizacji pod kątem płynności językowej, co prowadzi do generowania odpowiedzi, które brzmią wiarygodnie, ale są całkowicie nieprawdziwe.

4.3 Konsekwencje halucynacji

Halucynacje w modelach językowych mogą mieć poważne konsekwencje w rzeczywistych zastosowaniach. W dziedzinie medycyny, błędne odpowiedzi mogą prowadzić do dezinformacji, która zagraża zdrowiu pacjentów, np. gdy model sugeruje nieistniejące terapie. Podobnie, w sektorze prawnym, halucynacje mogą prowadzić do generowania fałszywych interpretacji przepisów, co może być szkodliwe w procesach decyzyjnych. Długofalowo problem halucynacji wpływa również na zaufanie do technologii AI. Użytkownicy mogą podważać wiarygodność modeli językowych, jeśli często napotykają nieprawdziwe informacje. Ponadto halucynacje mogą wpływać na błędne decyzje biznesowe, gdy modele są wykorzystywane w analizach rynkowych lub finansowych.

4.4 Przykłady halucynacji

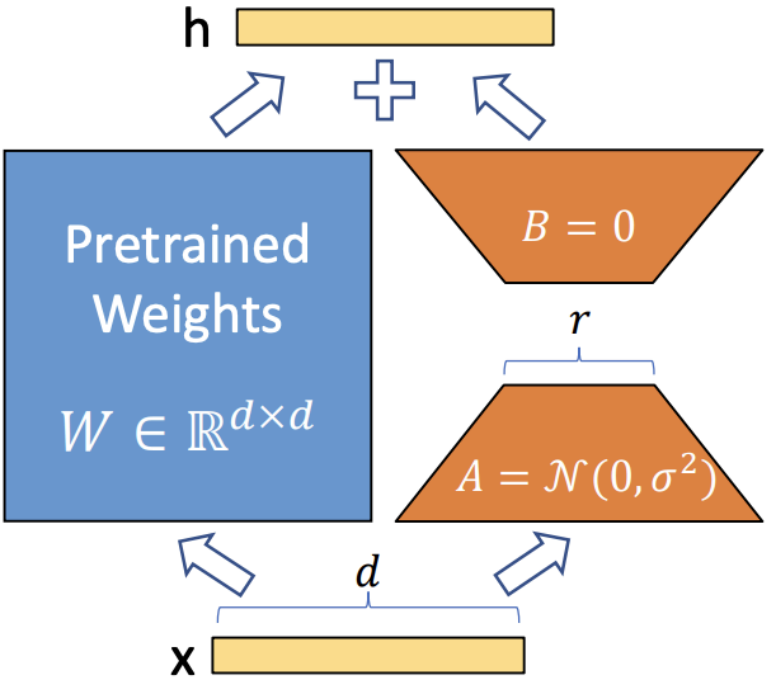
Zjawisko halucynacji można zaobserwować w różnych dziedzinach. Na przykład chatboty medyczne mogą sugerować nieistniejące terapie lub błędnie identyfikować choroby. W tłumaczeniach maszynowych model może wprowadzać treści, które nie były obecne w oryginale, co jest szczególnie szkodliwe w tłumaczeniu dokumentów prawnych lub literackich. W przypadku generowania kodu, modele językowe czasem tworzą syntaktycznie poprawny kod, który jednak jest niezgodny z dokumentacją używanych bibliotek. Wszystkie te przykłady pokazują, jak szeroko zjawisko halucynacji może się rozprzestrzeniać i jak różnorodne są jego formy

5. LOW-RANK ADAPTATION (LORA)

Low-Rank Adaptation (LoRA) to technika stosowana do efektywnego dotrenowywania dużych modeli językowych, która umożliwia wydajną adaptację modeli przy znacznym ograniczeniu zasobów obliczeniowych. Kluczowym problemem w fine-tuningu bardzo dużych modeli jest liczba parametrów wymagających aktualizacji, co wiąże się z wysokimi kosztami pamięci GPU i czasu obliczeń. LoRA, rozwiązuje ten problem poprzez redukcję liczby trenowalnych parametrów.

5.1 Intuicja i koncepcja działania LoRA

Podstawowym założeniem LoRA jest zastąpienie macierzy wag modelu ich niskowymiarowymi reprezentacjami. Standardowo, podczas 'fine-tuningu' (dostrojenia modelu), wagi modelu są aktualizowane zgodnie z regułą $W_0 = W_0 + \Delta W$, gdzie ΔW reprezentuje gradient macierzy wag. W podejściu LoRA macierze wag modelu pozostają bez zmian natomiast gradient ΔW jest przybliżany jako iloczyn dwóch macierzy niskiej rangi. Następnie dodawane są do siebie wartości wyjściowe macierzy niższej rangi z oryginalnymi macierzami modelu. W ten sposób zmniejszana jest liczba operacji koniecznych do obliczenia gradientu a co za tym idzie zużycie pamięci jest znacząco zmniejszone. Taka projekcja pozwala na przeprowadzenie nauki w przestrzeni o niższych wymiarach, a następnie rekonstruowanie aktualizacji w oryginalnej przestrzeni modelu.



Rys. 5.1. Uproszczony schemat działania techniki LoRA (Źródło: [2])

5.2 Praktyczne zastosowanie i efektywność

LoRA pozwala na wykorzystanie małych podprzestrzeni niskowymiarowych, co w praktyce oznacza, że podczas do trenowywania aktualizowane jest zaledwie ułamek parametrów oryginalnego modelu – czasem mniej niż 0,01%. Ta redukcja czyni dostosowywanie modeli językowych, nie tylko technicznie wykonalnym, ale także ekonomicznie opłacalnym. Ponadto, LoRA można stosować nie tylko do modeli językowych, ale także do innych generatywnych modeli wielomodalnych, takich jak Stable Diffusion.

Pierwotne badania Microsoftu, które wprowadziły tę technikę, wykazały, że LoRA może zmniejszyć liczbę trenowalnych parametrów nawet 10 000-krotnie i jednocześnie trzykrotnie zredukować wymagania pamięci GPU [2]. Co istotne, LoRA nie wymaga modyfikacji samego modelu bazowego, co sprawia, że jest kompatybilna z większością istniejących architektur i bibliotek.

5.3 Przykład zmniejszenia parametrów przy użyciu techniki LoRA

Weźmy macierz o wymiarach $(5,\ 768)$ gdzie:

- 5 - liczba próbek,
- 768 - liczba cech.

Chcemy pomnożyć tą macierz przez inną macierz, np. o wymiarach $(768,\ 512)$, aby uzyskać wyjście $(5,\ 512)$. Wzór na wykonanie operacji prezentuje się następująco:

$$W\ =\ W_0\ +\ \Delta W.$$

Jeżeli użyjemy macierzy ΔW o pełnych wymiarach $(768,\ 512)$ to liczba jej parametrów wynosi:

$$P\ =\ 768\ \times\ 512\ =\ 393,216.$$

Przy użyciu LoRA macierz ΔW zastępujemy kombinacją dwóch mniejszych macierzy A i B

$$W\ =\ W_0\ +\ A\ \times\ B,$$

gdzie:

- A - macierz o wymiarach $(768,\ r)$,
- B - macierz o wymiarach $(r, 512)$.

Dla przykładu wybierzmy $r\ =\ 8$. Wtedy liczba parametrów prezentuje się następująco:

$$P\ =\ 768\ \times\ 8\ +\ 8\ \times\ 512\ =\ 10,240.$$

Procentowo mamy ponad 97% mniej parametrów do zaktualizowania.

5.4 Zalety LoRA w praktyce

LoRA szczególnie wyróżnia się w scenariuszach, gdzie dostępne są ograniczone zasoby obliczeniowe, a jednocześnie wymagane jest dostosowanie modelu do specyficznych zadań, np. generowania odpowiedzi w określonym stylu lub kontekście. Dzięki możliwości tworzenia niezależnych „filtrów" dla różnych zadań, ta technika umożliwia ponowne wykorzystanie tego samego modelu bazowego do wielu zastosowań. W rezultacie LoRA oferuje nie tylko wydajność obliczeniową, ale także elastyczność w projektowaniu i wdrażaniu modeli.

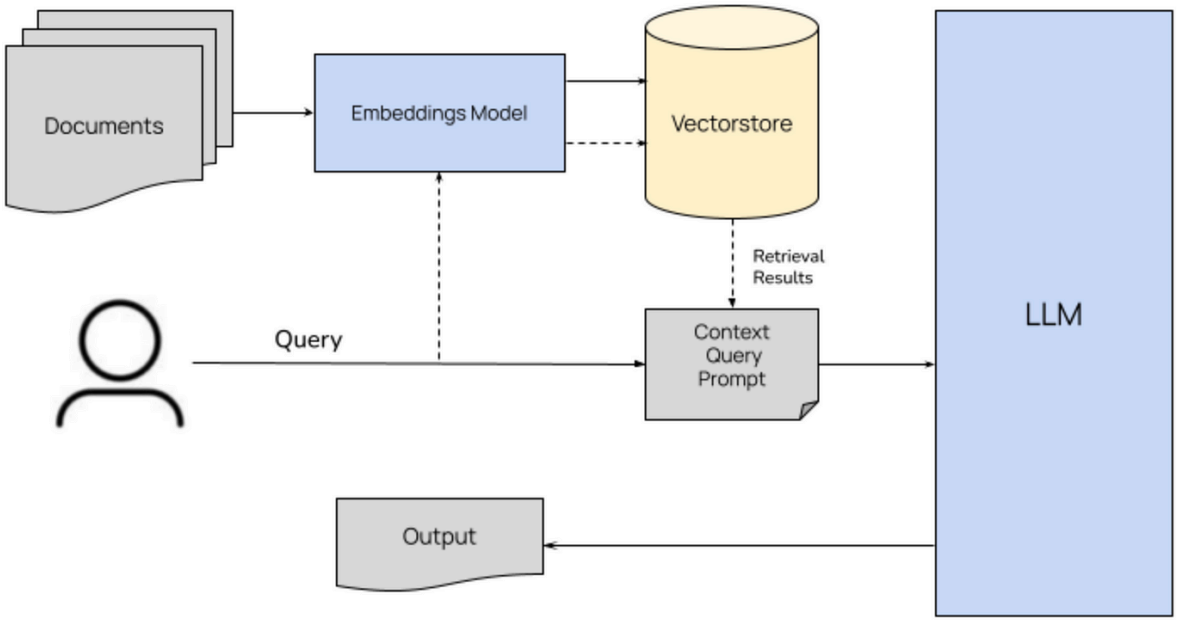
5.5 Inne warianty LoRA

Ze względu na zalety tego podejścia powstało kilka następujących wariantów:

- QLoRA [3] - wersja wprowadzająca kwantyzację wag,
- LongLoRA [4] - pozwala na zastosowanie LoRA dla sekwencjach o bardzo dużej długości,
- S-LoRA [5] - wersja która znacząco optymalizuje zużycie pamięci.

6. RETRIEVAL-AUGMENTED GENERATION (RAG)

Retrieval-Augmented Generation (RAG) [6] to technika łącząca pamięć parametryczną, zawartą w modelach sekwencyjnych, z pamięcią nieparametryczną, opartą na zewnętrznych źródłach wiedzy, takich jak indeks dokumentów. Główną ideą RAG jest rozszerzenie możliwości generowania języka poprzez dodanie mechanizmu wyszukiwania i integracji informacji spoza samego modelu. W tej architekturze pamięć parametryczna pochodzi z pretrenowanego modelu generacyjnego (np. BART [8]), a pamięć nieparametryczna z gęstych wektorowych indeksów, takich jak Wikipedia. Takie podejście umożliwia dynamiczne i zróżnicowane przetwarzanie danych, ponieważ mechanizm wyszukiwania zapewnia dostęp do najbardziej odpowiednich fragmentów informacji, które następnie są używane do generowania wyników. Rys. 6.1 przedstawia uproszczony schemat działania mechanizmu RAG.



Rys. 6.1. Uproszczony schemat działania RAG (Źródło: [20])

6.1 Zalety RAG w Zastosowaniach Praktycznych

Jedną z największych zalet RAG jest zdolność do radzenia sobie z zadaniami takimi jak na przykład otwarte odpowiadanie na pytania. W tych zadaniach klasyczne modele często "halucynują", generując odpowiedzi niezgodne z rzeczywistością, ponieważ polegają wyłącznie na wiedzy zapisanej w ich parametrach. RAG eliminuje ten problem, ponieważ umożliwia pobieranie aktualnych i sprawdzonych informacji z bazy danych. Modele RAG mogą również łatwo dostosowywać swoją wiedzę – aktualizacje w bazie dokumentów natychmiast wpływają na wyniki, bez konieczności ponownego trenowania modelu.

Efektywność RAG wynika także z jego zdolności do generowania odpowiedzi bardziej szczegółowych, różnorodnych i zgodnych z faktami w porównaniu do podejść czysto parametrycznych. Dzięki użyciu komponentu wyszukiwawczego, np. Dense Passage Retriever (DPR), RAG selektywnie pobiera najbardziej trafne fragmenty tekstu, które mogą być następnie przetwarzane przez model generacyjny. To połączenie skutkuje znaczną poprawą wyników w porównaniu do modeli ekstrakcyjnych i czysto generacyjnych, co potwierdzają eksperymenty na popularnych zestawach danych, takich jak Natural Questions [19] czy TriviaQA [9].

6.2 Interpretowalność i Elastyczność RAG

Kolejną korzyścią z użycia RAG jest interpretowalność – dzięki nieparametrycznej pamięci użytkownicy mogą prześledzić, które dokumenty były używane do generowania odpowiedzi. To sprawia, że RAG jest atrakcyjnym rozwiązaniem w aplikacjach wymagających wysokiego poziomu przejrzystości, takich jak weryfikacja faktów czy generowanie raportów. Ponadto jego zdolność do pracy w różnych domenach oraz elastyczność w zakresie zadań czynią go jednym z najbardziej obiecujących podejść do zaawansowanego przetwarzania języka naturalnego.

6.3 Wykorzystanie FAISS w RAG

RAG korzysta z biblioteki FAISS (Facebook AI Similarity Search) [18], która odgrywa istotną rolę w komponentach związanych z pamięcią nieparametryczną. FAISS to wysoce wydajne narzędzie do wyszukiwania podobieństw w dużych zbiorach danych, które pozwala na szybkie odnajdywanie najbardziej odpowiednich dokumentów spośród milionów dostępnych. W kontekście RAG FAISS umożliwia budowę indeksów dokumentów w postaci gęstych reprezentacji wektorowych, które są następnie wykorzystywane przez algorytmy wyszukiwania, takie jak Maximum Inner Product Search (MIPS). Dzięki zastosowaniu FAISS modele RAG mogą z dużą szybkością przeszukiwać rozległe bazy wiedzy, takie jak cała Wikipedia, i efektywnie dobierać dokumenty najbardziej pasujące do zapytania.

Kluczową zaletą FAISS jest jego zdolność do pracy w czasie zbliżonym do rzeczywistego dzięki zastosowaniu technik takich jak hierarchiczne grafy małego świata (Hierarchical Navigable Small World Graphs) [17]. To pozwala na skalowanie RAG nawet w scenariuszach z ogromnymi zbiorami danych. FAISS nie tylko przyspiesza przetwarzanie, ale także zwiększa precyzję wyszukiwania, co przekłada się na wyższą jakość generowanych odpowiedzi. Dzięki integracji FAISS, RAG staje się nie tylko bardziej wydajny, ale również elastyczny i zdolny do dynamicznego aktualizowania wiedzy w miarę dodawania nowych informacji do indeksu.

7. ZBIÓR DANYCH

Do treningu i ewaluacji modeli wykorzystano dataset SQuAD (Stanford Question Answering Dataset) [13], który jest jednym z najpopularniejszych zbiorów danych przeznaczonych do zadań odpowiadania na pytania. Składa się on z zestawu pytań w języku angielskim, wraz z odpowiadającymi im fragmentami tekstu pochodzącymi z artykułów Wikipedii, na podstawie których można znaleźć odpowiedź. Wersja 2.0 dodatkowo wprowadza pytania, na które odpowiedzi mogą nie istnieć w danym kontekście, co zwiększa trudność zadania i lepiej symuluje rzeczywiste sytuacje.

Jedną z największych zalet SQuAD jest jego bogactwo językowe i różnorodność tematów, co czyni go odpowiednim zbiorem do treningu modeli w zadaniach wymagających zrozumienia kontekstu i formułowania precyzyjnych odpowiedzi. Odpowiedzi w datasetcie są krótkimi fragmentami tekstu, co dobrze współgra z zadaniami wymagającymi jednoznacznych, zwięzłych odpowiedzi. Jest on także szeroko stosowany jako benchmark, co ułatwia porównywanie wyników modeli z istniejącymi rozwiązaniami i pozwala na ocenę ich wydajności w standardowych warunkach.

7.1 Wstępne przetwarzanie oraz podział zbioru danych

W celu przygotowania zbioru danych SQuAD do treningu i ewaluacji zastosowano procedurę wstępnego przetwarzania oraz podziału danych przedstawioną w skrypcie znajdującym się w Dodatku A. Zbiór danych został pobrany w wersji przeznaczonej do treningu, a następnie poddany filtrowaniu i oczyszczaniu w celu usunięcia duplikatów oraz zapewnienia zgodności z ograniczeniami modelu. Po przetworzeniu zbiór treningowy składał się z 10 tysięcy pytań i odpowiedzi.

Część testowa również została w sposób losowy zmniejszona do około 5 tysięcy przykładów aby zapewnić losowość pytań oraz przyspieszyć proces ewaluacji który dla niektórych przypadków i tak trwał kilkanaście godzin.

8. APLIKACJA

W celu efektywnego porównania rozważanych metod, została stworzona aplikacja umożliwiająca przeprowadzenie szczegółowych eksperymentów i analiz w sposób przejrzysty i zautomatyzowany. Głównym celem było stworzenie narzędzia, które pozwala na dokładną ocenę wyników i wydajności poszczególnych metod w określonych scenariuszach testowych, minimalizując jednocześnie ryzyko błędów manualnych i zapewniając replikowalność eksperymentów. Dodatkowo dołączony zostaje interfejs użytkownika który w bardzo prosty sposób może na swojej lokalnej maszynie wejść w interakcję z najlepszym modelem.

8.1 Technologia

Aby w jak najprostszy i najlepszy sposób dokonać porównania aplikacja została napisana w języku Python, co zapewnia elastyczność oraz dostęp do szerokiej gamy zaawansowanych narzędzi. Do implementacji rozważanych metod wykorzystano bibliotekę *transformers*, rozwijana przez organizację *huggingface* [14] która dostarcza zaawansowane narzędzia i modele do uczenia maszynowego, a także umożliwia łatwą integrację i adaptację nowoczesnych architektur przetwarzania języka naturalnego.

W przypadku metody RAG dodatkowo zastosowano bibliotekę FAISS (Facebook AI Similarity Search), która pozwala na efektywne wyszukiwanie w dużych zbiorach danych. FAISS umożliwia szybkie i precyzyjne wyszukiwanie podobieństw, co jest kluczowe w procesie odzyskiwania informacji w metodzie RAG.

Technika LoRA została zaimplementowana przy użyciu biblioteki *peft* która jest rozwijana pod skrzydłami tej samej organizacji co wcześniej wspomniana biblioteka *transformers*, dzięki temu zapewnia ona bezproblemową integrację modeli z metodą LoRA.

8.2 Model językowy

Do przeprowadzenia eksperymentu wykorzystano model **Qwen 2.5-1.5B-Instruct** [15], należący do najnowszej serii dużych modeli językowych Qwen. Model ten posiadający 1.5 miliarda parametrów został wybrany ze względu na swoje zaawansowane możliwości w zakresie przetwarzania języka naturalnego oraz dostosowania do instrukcji.

Qwen 2.5-1.5B-Instruct wyróżnia się:

- wiedzą dziedzinową oraz poprawionymi zdolnościami w obszarach kodowania i matematyki, dzięki zastosowaniu specjalistycznych modeli eksperckich,

- poprawioną obsługą danych strukturalnych, takich jak tabele, oraz generowaniem wyników w ustrukturyzowanej formie, w tym JSON,

- większą odpornością na różnorodność promptów systemowych, co ułatwia implementację symulacji ról i ustalanie warunków dla chatbotów,
- wsparciem dla 29 języków, w tym m.in. angielskiego, chińskiego, francuskiego, niemieckiego, rosyjskiego i arabskiego.

8.3 Eksperyment z długością zapytania

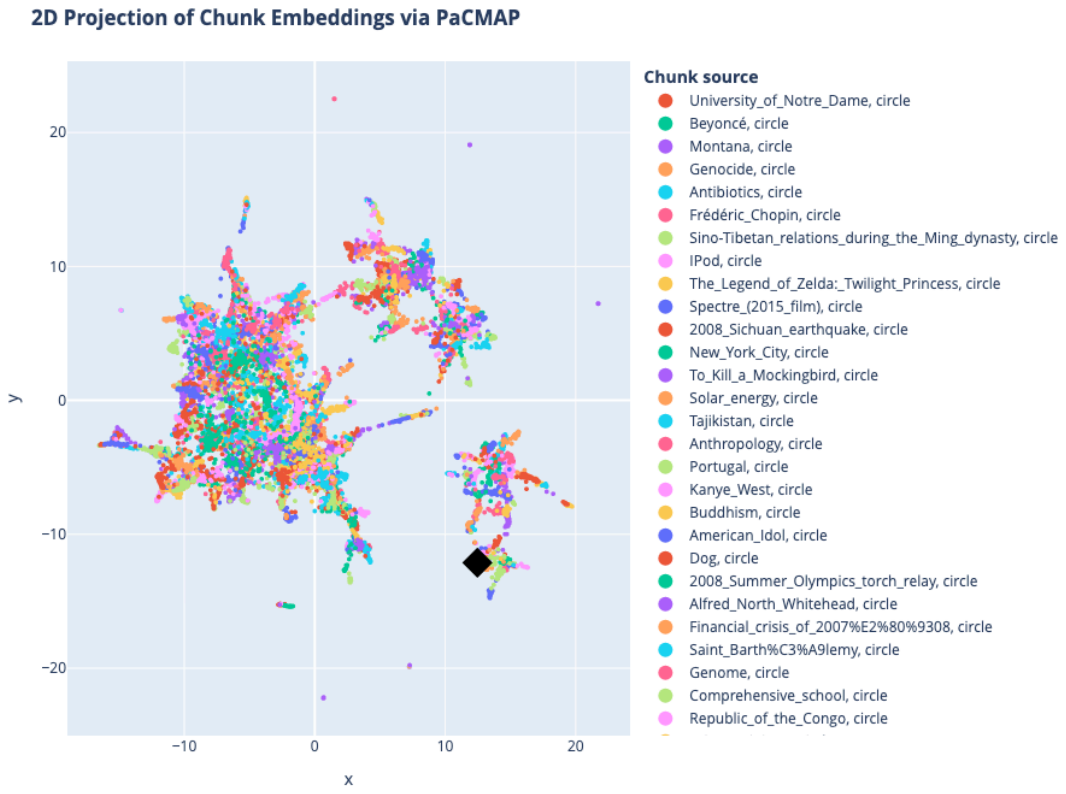
Przed rozpoczęciem procesu treningu i ewaluacji poszczególnych metod należało odpowiednio przygotować zbiór danych. Zbiór SQuAD został pobrany w wersji przeznaczonej do treningu, a następnie poddany filtrowaniu i oczyszczaniu w celu usunięcia duplikatów oraz zapewnienia zgodności z ograniczeniami modelu, tak aby długość pytania i kontekstu nie przekraczała długości tekstu jaki model może przyjąć, ponieważ może to znacznie wpłynąć na otrzymane rezultaty, gdyż model obetnie tekst do wymaganej długości (w tym przypadku 512 tokenów) co może skutkować utratą kontekstu lub przycięciem pytania. Następnie dla techniki RAG ze zbioru treningowego zostały usunięte rekordy które miały taki sam kontekst, miało to na celu znacząco zmniejszyć rozmiar bazy danych i przyspieszyć wyszukiwanie podobnych tekstów podczas inferencji.

8.4 Tworzenie wektorowej bazy danych

Do utworzenia wektorowej bazy danych która będzie służyć jako źródło wiedzy dla jednej z rozważanych metod użyto wcześniej wspomnianej biblioteki FAISS która z pomocą biblioteki *langchain_huggingface* ładuje tekst, konwertuje go na wektory które potem są porównywane ze sobą przy użyciu podobieństwa kosinusowego [22].

Warto zaznaczyć w tym miejscu, że podczas inferencji zapytanie użytkownika jest osadzane w tę samą przestrzeń wektorową i na tej podstawie wyłaniane są dokumenty najbardziej zbliżone do pytania użytkownika.

W procesie tworzenia wektorowej bazy danych przetworzono zbiór danych zawierający ponad 10 tysięcy dokumentów. Wizualizacja na rys. 7.1 ilustruje sposób dystrybucji tych dokumentów po projekcji przy użyciu PaCMAP [21] na dwuwymiarową przestrzeń wektorową, pomaga to zrozumieć i zyskać intuicję na temat organizacji danych w bazie danych.



Rys. 8.1. Wizualizacja 2D przestrzeni wektorowej osadzeń w RAG (Źródło: opracowanie własne)

8.5 Implementacja modelu z RAG

Logika odpowiadająca za implementację metodologii RAG została zaimplementowana w skrypcie języka Python patrz Dodatek B. Proces ten rozpoczyna się od przygotowania wektorowej bazy danych, w której dokumenty są reprezentowane w formie osadzeń wektorowych. W tym celu wykorzystano wcześniej wspominaną bibliotekę FAISS. Do generowania osadzeń zastosowano model osadzania **thenpler/gte-base** dostarczony przez bibliotekę HuggingFace, co pozwala na przekształcenie tekstów w przestrzeń wektorową, uwzględniającą semantyczne podobieństwo.

W pierwszym etapie funkcja **create_vector_db** tworzy bazę danych dokumentów. Zbiór danych, zawierający konteksty tekstowe, jest przetwarzany na obiekty typu **LangchainDocument**, które są następnie konwertowane na wektory i przechowywane w bazie FAISS. W procesie tym stosowany jest algorytm porównania na podstawie kosinusowej miary podobieństwa, co zapewnia skuteczność w wyłanianiu dokumentów najbardziej zbliżonych do zapytań użytkownika.

Funkcja **get_llm** odpowiedzialna jest za inicjalizację modelu oraz jego tokenizatora, zapewniając możliwość wydajnej pracy na urządzeniach lokalnych. W kolejnym kroku, funkcja **apply_prompt** tworzy szablon zapytania użytkownika, łącząc dane wejściowe (kontekst i pytanie) z instrukcjami dla modelu.

Podczas procesu inferencji zapytanie użytkownika jest osadzane w tej samej przestrzeni wektorowej co dokumenty w bazie danych. Funkcja **create_prompt** odpowiada za wyłonienie najbardziej podobnych dokumentów poprzez wyszukiwanie w bazie FAISS oraz ich przetworzenie do formy odpowiedniej do wykorzystania przez model językowy. Następnie generowana jest odpowiedź na podstawie dostarczonego kontekstu.

8.6 Implementacja modelu z LoRA

Kod przedstawiony jako Dodatek C implementuje technikę LoRA. Funkcja **get_lora** oferuje dwie opcje: ładowanie wcześniej wytrenowanego modelu z techniką LoRA lub tworzenie nowego modelu z zerową konfiguracją LoRA. W pierwszym przypadku model jest załadowany z biblioteki Hugging Face, a w drugim tworzony od podstaw przy użyciu określonych parametrów LoRA, takich jak liczba wymiarów osadzenia i współczynnik dropout.

W obu przypadkach wykorzystywany jest model Qwen 2.5-1.5B-Instruct, który jest ładowany w trybie bfloat16, co pozwala na efektywne wykorzystanie pamięci. Po załadowaniu lub utworzeniu modelu, jest on integrowany z techniką LoRA, co umożliwia dostosowanie go do specyficznych zadań przy minimalnym zużyciu pamięci. Dodatkowo kod umożliwia ocenę pamięci zajmowanej przez model oraz liczbę parametrów, które podlegają treningowi.

8.7 Trening modelu

Do przeprowadzenia treningu opracowano skrypt patr, który zapewnia powtarzalny i w pełni zautomatyzowany proces, minimalizując ryzyko błędów i usprawniając zarządzanie eksperymentami. Struktura skryptu została zaprojektowana w sposób modułowy, co umożliwia łatwy wybór i konfigurację jednej z rozważanych technik treningowych, takich jak RAG czy LoRA. Dzięki temu użytkownik może w prosty sposób dostosować proces treningowy do swoich potrzeb, bez konieczności modyfikowania głównego kodu. Elastyczność ta pozwala na szybką iterację i eksperymentowanie z różnymi metodami w celu znalezienia optymalnego rozwiązania.

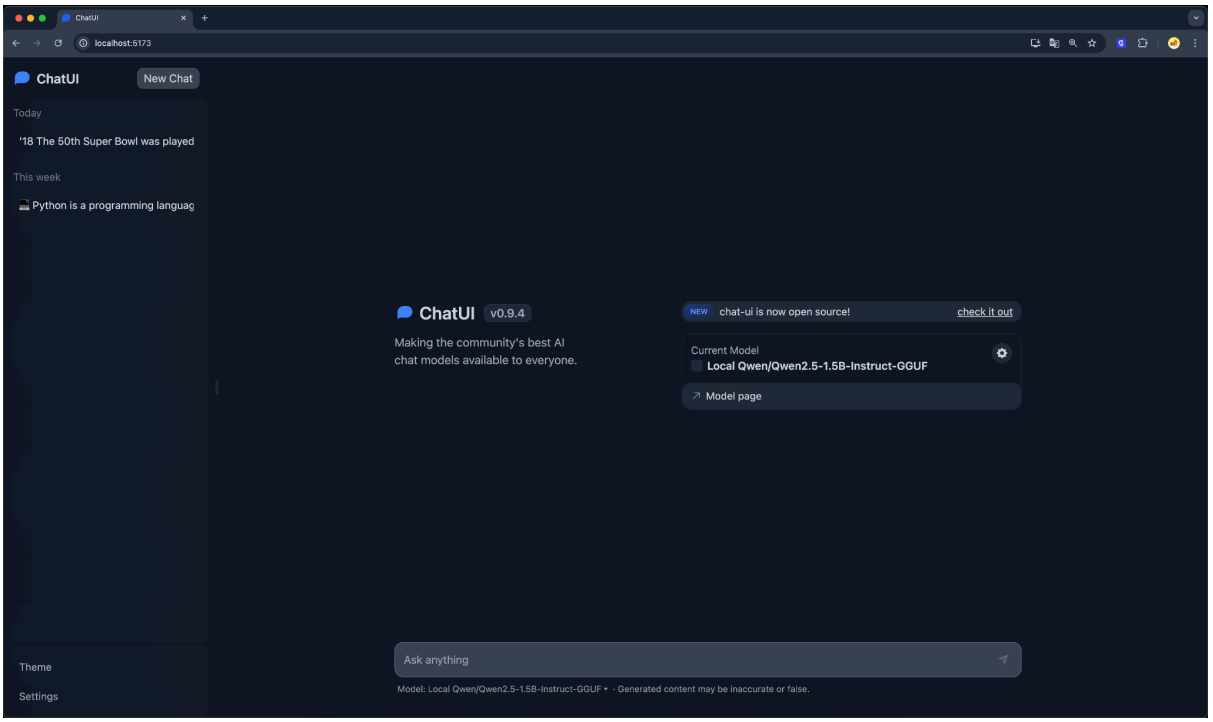
Model wzbogacony o metodę LoRA trenował się przez 1 epokę na urządzeniu MacBookPro (M3 36 GB RAM) zajęło to ponad 6 godzin a zużycie pamięci sięgało 34 GB.

Metoda RAG była znacząco mniej obciążająca dla komputera, zajęła około 30 min nie zużywając ogromnych zasobów CPU i GPU. Warto zaznaczyć, że w tym przypadku należało jedynie utworzyć wektory z tekstu znajdującego się w zbiorze treningowym i wpisać je do wektorowej bazy, więc nie było tutaj dokonywanych aktualizacji wag modelu.

8.8 Interfejs użytkownika

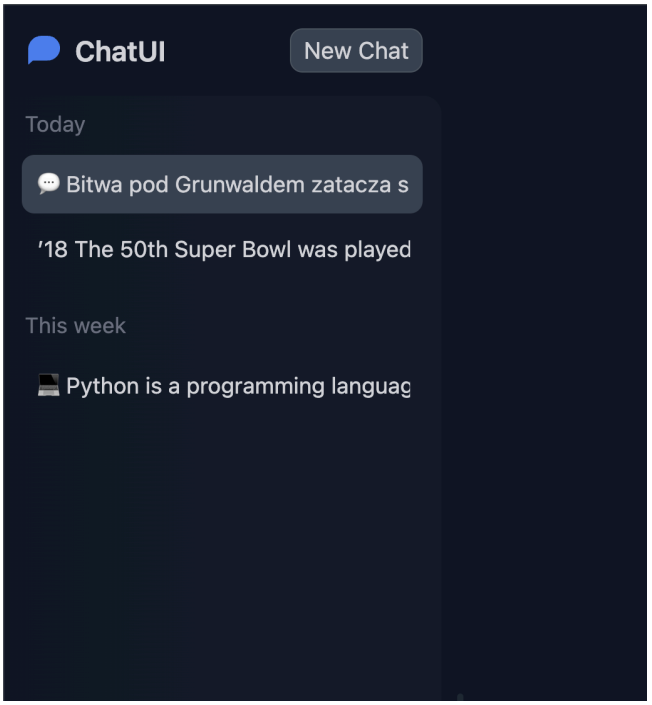
Interfejs użytkownika to aplikacja napisana która pozwala na serwowanie modelu lokalnie przy użyciu *llama-server* rozwiązania które stworzone zostało do wykorzystania z dużymi modelami językowymi na lokalnych stacjach bez konieczności posiadania ogromnych klastrów komputerowych. Polega ono na skompilowaniu modelu do wydajnego formatu *GGUF* [16] zaproponowanego przez autora wspomnianego rozwiązania.

Użytkownik może wchodzić w interakcję z modelem w bardzo prosty sposób. Interfejs przedstawiony na rys. 8.2 jest bardzo podobny do tych znanych z ChatGPT czy innych dostępnych rozwiązań pozwalających na interakcję z LLM.



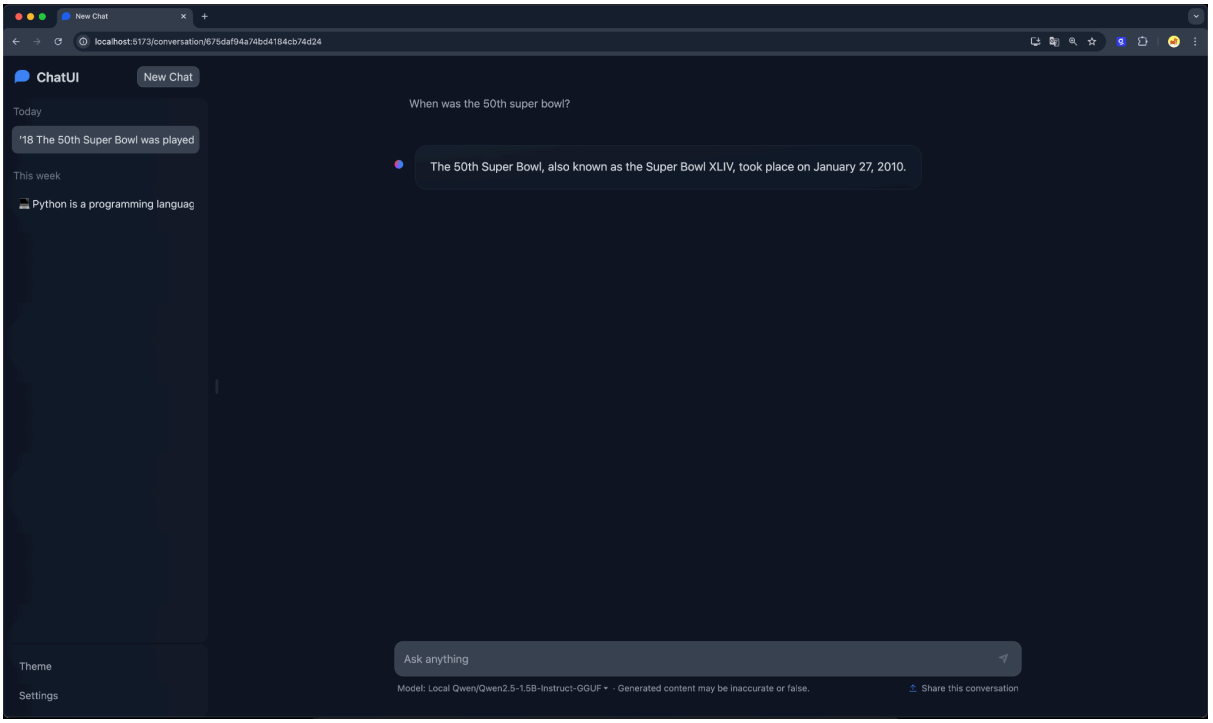
Rys. 8.2. Strona główna interfejsu (Źródło: opracowanie własne)

Dzięki użyciu narzędzia Docker interfejs może być łatwo uruchomiony na większości maszyn. Dodatkowo używa nierelacyjnej bazy danych MongoDB do przechowywania historii czatów użytkownika które są umieszczone po lewej stronie interfejsu widoczne na rys. 8.3.



Rys. 8.3. Historia czatów użytkownika (Źródło: opracowanie własne)

Po wpisaniu przez użytkownika zapytania i wciśnięcia klawisza enter, interfejs przechodzi w tryb odpowiedzi jak na rys. 8.4, pokazując pytanie użytkownika a następnie odpowiedź modelu.



Rys. 8.4. Interfejs po wysłaniu zapytania do modelu (Źródło: opracowanie własne)

9. METRYKI EWALUACYJNE

W celu kompleksowej ewaluacji zdolności modeli do odpowiadania na pytania zastosowano cztery zaawansowane metryki: **ROUGE**, **BLEU**, **oraz accuracy.** Każda z tych metryk mierzy inne aspekty jakości generowanych odpowiedzi, co pozwala na ich wielowymiarową ocenę i identyfikację mocnych oraz słabych stron poszczególnych podejść.

9.1 ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

Jest metryką oceniającą pokrycie między odpowiedzią generowaną przez model a tekstem referencyjnym. Koncentruje się na liczbie wspólnych n-gramów, słów lub fraz, a jej szczególna wartość leży w pomiarze recall (czyli jak duża część kluczowych informacji z odpowiedzi referencyjnej została uchwycona w generowanym tekście). Jest to szczególnie istotne w zadaniach, gdzie istotniejsze jest pokrycie treści niż jej precyzyjne odwzorowanie, takich jak generowanie streszczeń czy odpowiadanie na pytania z otwartym zakresem odpowiedzi.

9.2 BLEU (Bilingual Evaluation Understudy)

To popularna metryka w zadaniach przetwarzania języka naturalnego, szczególnie tłumaczenia maszynowego, ale znajduje zastosowanie również w ocenie odpowiedzi na pytania. BLEU analizuje n-gramy generowanego tekstu, porównując je z referencyjnymi odpowiedziami, i uwzględnia długość odpowiedzi, co pozwala na ocenę płynności i spójności. W przypadku odpowiadania na pytania BLEU jest szczególnie przydatna, gdy odpowiedzi muszą być dokładne i sformułowane w sposób zbliżony do wzorca.

9.3 Accuracy (dokładność)

Jest jedną z najbardziej intuicyjnych metryk i mierzy odsetek odpowiedzi poprawnych względem danych referencyjnych. W przypadku modeli odpowiadających na pytania jest to metryka szczególnie przydatna, gdy odpowiedzi mają formę jednoznaczną, na przykład w zadaniach z zamkniętym zestawem odpowiedzi. Accuracy pozwala ocenić zdolność modelu do udzielania poprawnych odpowiedzi na poziomie binarnym – odpowiedź jest albo poprawna, albo nie.

Połączenie tych trzech metryk pozwala na dokładną i wszechstronną ocenę zdolności modeli do odpowiadania na pytania. ROUGE i BLEU dostarczają informacji o powierzchniowej jakości tekstu i odtwarzaniu kluczowych n-gramów, accuracy ocenia ogólną poprawność odpowiedzi.

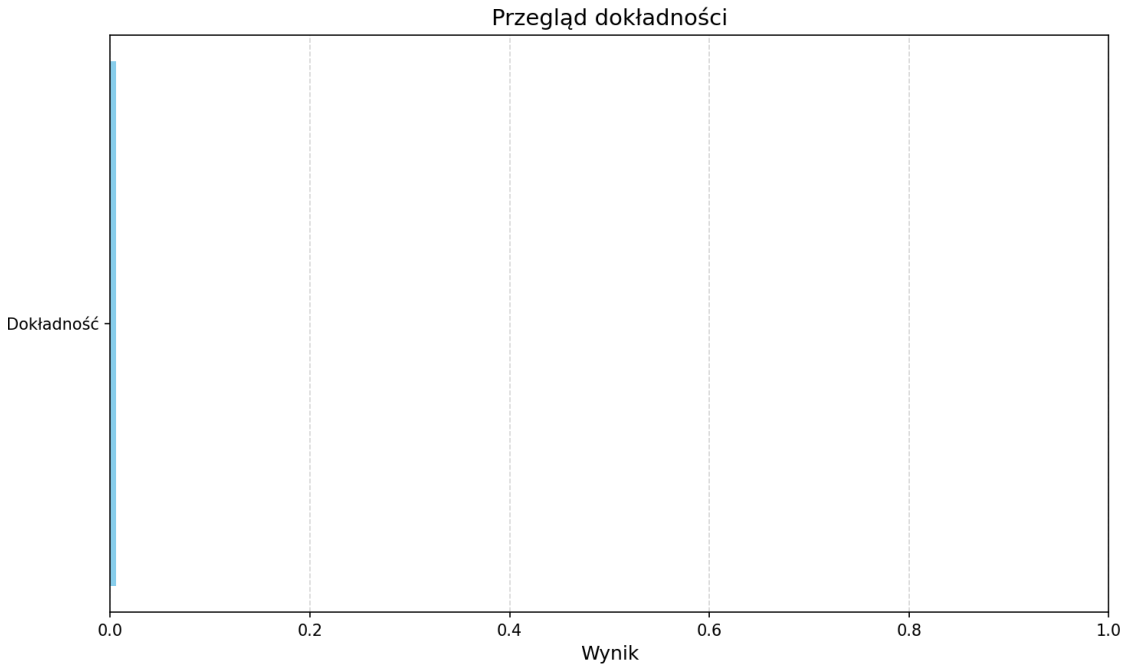
10. ANALIZA WYNIKÓW

Każdy z analizowanych wariantów modelu został poddany ewaluacji na pełnym zbiorze testowym. Następnie, przy zastosowaniu walidacji krzyżowej, zbiór testowy został podzielony na 10 podzbiorów. Na każdym z tych podzbiorów obliczono wspomniane metryki, po czym wyznaczono średnią oraz odchylenie standardowe z uzyskanych wyników. Ewaluacja na podzbiorach nie była możliwa dla modelu z RAG z racji bardzo dużych wymagań czasowych i sprzętowych. Dodatek E zawiera pełen skrypt użyty do ewaluacji rozważanych metod, na podstawie utworzonych podczas ewaluacji plików json zawierających wyniki wygenerowane zostały wszystkie wykresy załączone na rysunkach poniżej.

Walidacja krzyżowa z 10 podzbiorami ma na celu zwiększenie wiarygodności oceny modelu poprzez redukcję wpływu losowego podziału danych testowych na wyniki. Dzięki temu możliwe jest uzyskanie bardziej reprezentatywnych i stabilnych wyników, co pozwala na dokładniejszą ocenę zdolności generalizacyjnych modelu na różnych częściach danych testowych.

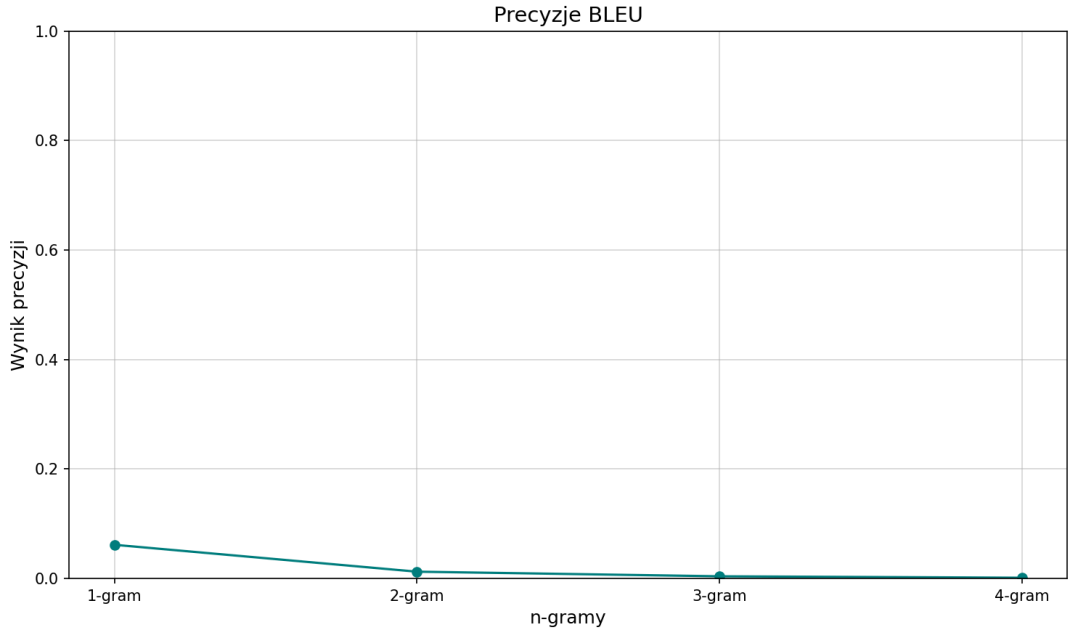
10.1 Model bazowy

Model bazowy wykazuje znaczące trudności w generowaniu krótkich i poprawnych odpowiedzi, co wynika z analizy danych ewaluacyjnych. Accuracy na poziomie 0.0059 (około 0.6%) widoczne na rys. 10.1 pokazuje, że jedynie niewielki odsetek odpowiedzi jest zgodny z oczekiwaniami. Wynik ten wskazuje na słabą zdolność modelu do precyzyjnego rozpoznawania i przedstawiania istotnych informacji w formie wymaganej przez zadanie.



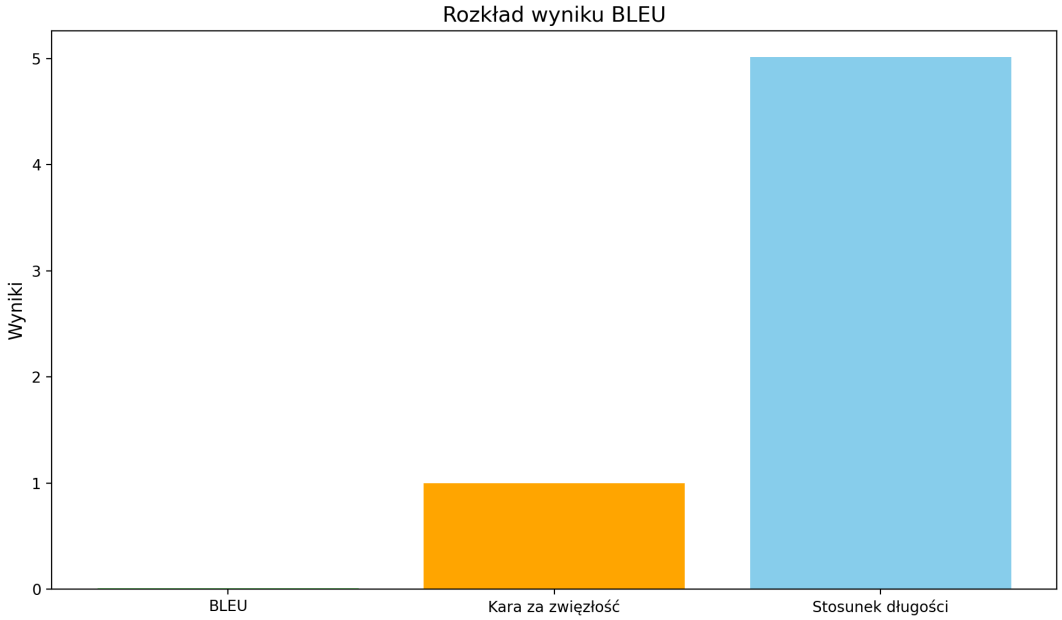
Rys. 10.1. Dokładność modelu bazowego (Źródło: opracowanie własne)

BLEU wynoszące 0.0081 pokazane na rys. 10.2 odzwierciedla słabą zgodność odpowiedzi modelu z referencyjnymi wzorcami. Precyzja dla 1-gramów wynosi 6.15%, co oznacza, że pojedyncze słowa referencyjne pojawiają się w generowanych odpowiedziach, jednak wartości dla dłuższych n-gramów gwałtownie spadają – do 0.14% dla 4-gramów. Wskazuje to, że model ma trudności z generowaniem spójnych fraz lub zdań, które odpowiadałyby strukturze referencyjnej. Dodatkowo, brak kary za długość (brevity penalty = 1.0) pokazuje, że problemem nie jest zbyt krótka forma odpowiedzi, ale wręcz przeciwnie – ich nadmierna rozwlekłość.



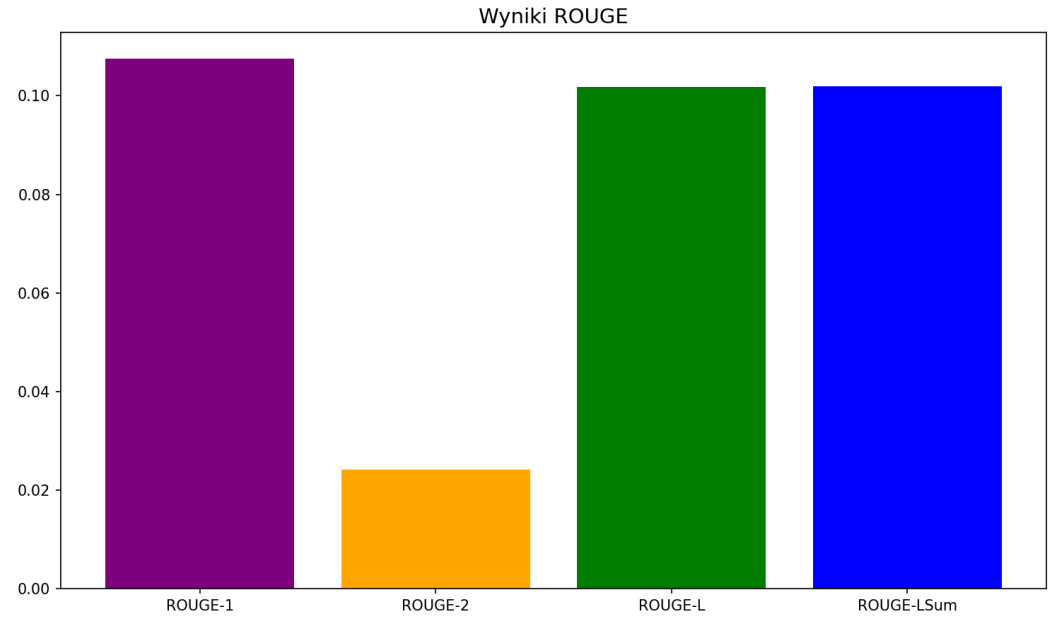
Rys. 10.2. Precyzje n-gramów BLEU dla modelu bazowego (Źródło: opracowanie własne)

Stosunek długości generowanych odpowiedzi do referencyjnych widoczny na rys. 10.3 wynosi aż 5.01, co oznacza, że odpowiedzi modelu są ponad pięciokrotnie dłuższe niż oczekiwane. To znaczące przeszacowanie wskazuje na tendencję modelu do rozwijania odpowiedzi w sposób niepotrzebnie szczegółowy i redundantny. Długość ta jest jednym z głównych czynników wpływających na obniżenie jakości generowanych odpowiedzi, zarówno w zakresie ich zwięzłości, jak i precyzji.



Rys. 10.3. Rozkład wyników BLEU dla modelu bazowego (Źródło: opracowanie własne)

Wyniki metryk ROUGE również podkreślają problemy modelu. Pokazane na rys. 10.4 ROUGE-1 wynosi 10.75%, co oznacza, że około 10% słów referencyjnych występuje w odpowiedziach modelu. ROUGE-2 spada do 2.41%, co wskazuje na bardzo niską zgodność na poziomie bigramów, natomiast ROUGE-L, wynoszący 10.18%, pokazuje trudności modelu w uchwyceniu struktury referencyjnych odpowiedzi. ROUGE-Lsum, na poziomie 10.19%, potwierdza te problemy również w kontekście organizacji całych tekstów.

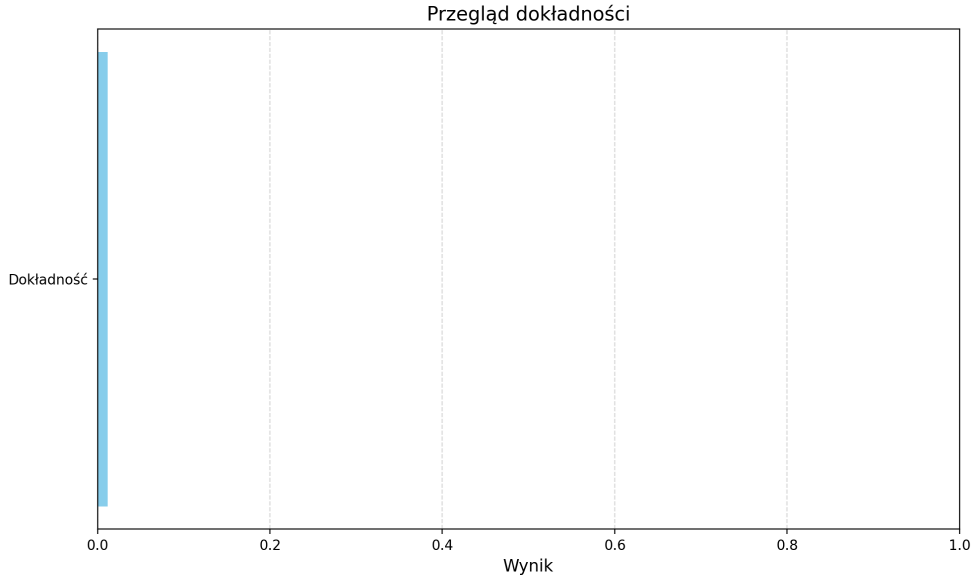


Rys. 10.4. Wyniki ROUGE dla modelu bazowego (Źródło: opracowanie własne)

Model bazowy wydaje się mieć poważne trudności z dostosowaniem generowanych treści do wymogu zwięzłości i precyzji. Jego strategia, polegająca na nadmiernym rozwijaniu odpowiedzi, prowadzi do znaczącej redundancji i bardzo niskiej zgodności z referencjami. Należy również pamiętać, że model bazowy sam w sobie nie jest największy, jego stosunkowo niewielkie rozmiary 1.5 miliarda parametrów to niewiele aby w scenariuszu zero-shot odpowiadać na trudne pytania ze zbioru danych SQuAD.

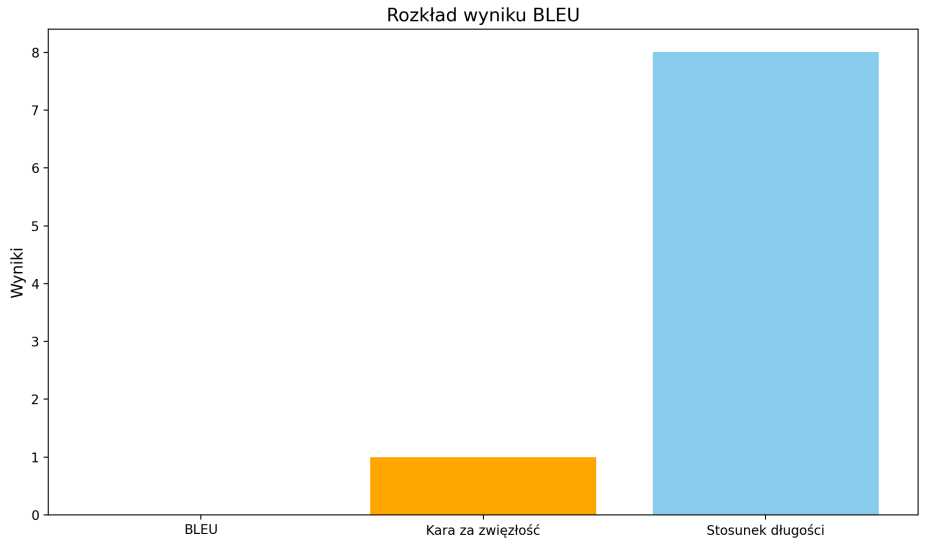
10.2 Model z RAG

Wyniki ewaluacji modelu Qwen2.5 (1.5B Instruct) w zadaniu wymagającym generowania krótkich, poprawnych odpowiedzi wskazują na trudności w realizacji tego celu. Wartość accuracy widoczna na rys 10.5 wynosi tylko 2.3%. Tak niski wynik może być spowodowany niewystarczającą zdolnością modelu do przekształcenia dostarczonego kontekstu w poprawne, precyzyjne odpowiedzi. Istnieje możliwość, że problem leży zarówno w samym modelu, jak i w sposobie, w jaki zintegrowano go z mechanizmem Retrieval-Augmented Generation (RAG), który może zwracać niewystarczająco trafne lub nadmiarowe dane kontekstowe.



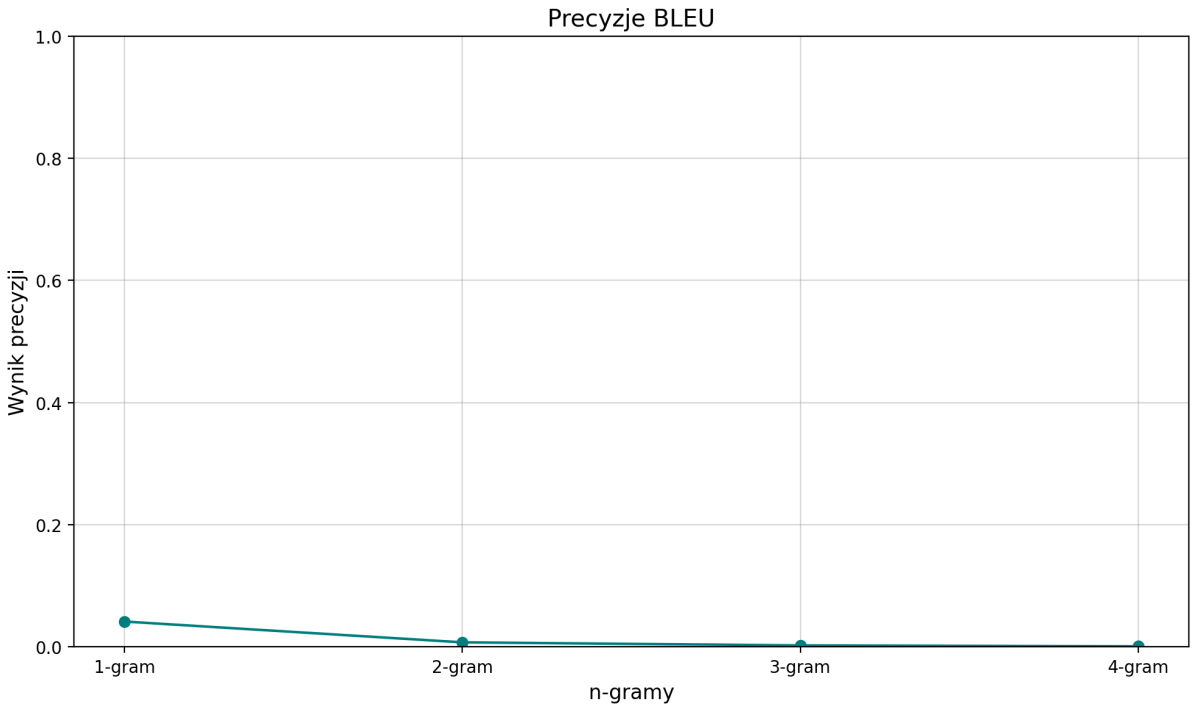
Rys. 10.5. Dokładność modelu z RAG (Źródło: opracowanie własne)

Metryka BLEU, wynosząca 0.0109, dodatkowo podkreśla rozbieżności między odpowiedziami modelu a referencyjnymi odpowiedziami. Brak kary za długość (brevity penalty wynoszące 1.0) widoczne na rys. 10.6 sugeruje, że odpowiedzi modelu nie były krótsze od referencyjnych, ale problemem był ich nadmiarowy charakter – wskazuje na to stosunek długości wygenerowanych odpowiedzi do referencyjnych, wynoszący 1.69. Oznacza to, że odpowiedzi modelu były niemal 70% dłuższe od oczekiwanych. Zbyt rozwlekła forma generowanych odpowiedzi jest sprzeczna z wymaganiem zwięzłości i wpływa negatywnie na wyniki wszystkich innych metryk, takich jak precyzja BLEU i ROUGE.



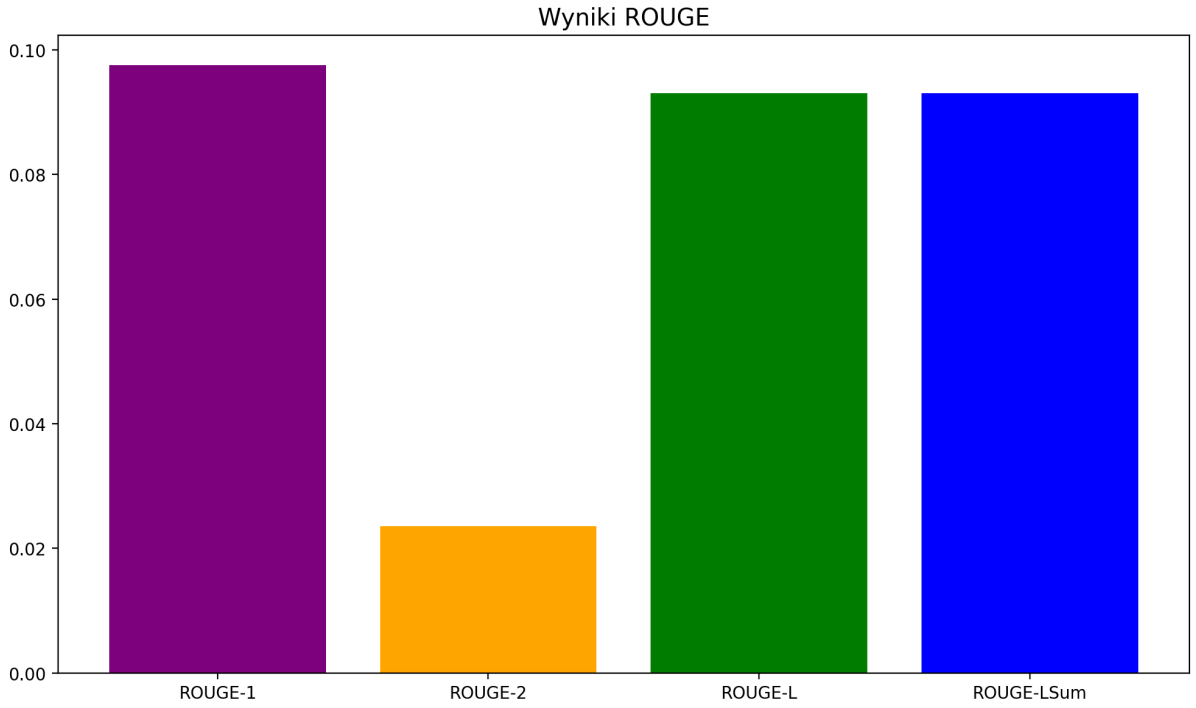
Rys. 10.6. Rozkład wyników BLEU dla modelu z RAG (Źródło: opracowanie własne)

Analiza precyzji n-gramów przedstawiona na rys. 10.7 pokazuje znaczące rozbieżności. Precyzja dla 1-gramów wynosi 7.8%, co wskazuje, że model był w stanie wygenerować pojedyncze słowa zgodne z referencyjnymi odpowiedziami, ale w miarę analizy dłuższych n-gramów (2-, 3- i 4-gramów) precyzja gwałtownie spada, osiągając zaledwie 0.18% dla 4-gramów. Taki wynik sugeruje, że model generuje odpowiedzi, które rzadko odzwierciedlają strukturalne wzorce występujące w referencjach. Może to wynikać z trudności w generowaniu spójnych fraz lub zdolności modelu do selekcji odpowiednich informacji w ograniczonym formacie odpowiedzi.



Rys. 10.7. Precyzje n-gramów BLEU dla modelu z RAG (Źródło: opracowanie własne)

Metryka ROUGE widoczna na rys. 10.8, która ocenia pokrycie odpowiedzi na poziomie n-gramów i podciągów, wykazuje podobne problemy. ROUGE-1 wynosi 10.18%, co oznacza, że około 10% słów referencyjnych pojawia się w odpowiedziach modelu. Jednak ROUGE-2, mierzący bigramy, spada do 2.37%, co podkreśla trudności modelu w generowaniu odpowiedzi, które pokrywają się z referencjami na poziomie fraz. Wartość ROUGE-L wynosząca 9.99% wskazuje, że model ma również problemy z zachowaniem struktury odpowiedzi referencyjnych. Co ciekawe, niski ROUGE-Lsum potwierdza, że te problemy występują zarówno na poziomie zdania, jak i całego tekstu.



Rys. 10.8. Wyniki ROUGE dla modelu z RAG (Źródło: opracowanie własne)

Długość odpowiedzi generowanych przez model i związane z tym problemy wydają się kluczowe dla zrozumienia niskiej jakości jego wyników. Odpowiedzi są rozwlekłe, co może wynikać z preferencji modelu do wyjaśniania lub rozszerzania odpowiedzi, zamiast ich kondensacji. To wskazuje na potencjalne braki w dostrojeniu modelu do zadania wymagającego zwięzłości. Dodatkowo, problemy z integracją RAG mogą odgrywać istotną rolę – jeśli zwracane dane są zbyt obszerne lub niedostosowane, model może nie być w stanie wyselekcjonować istotnych informacji i ograniczyć się do krótkiej odpowiedzi.

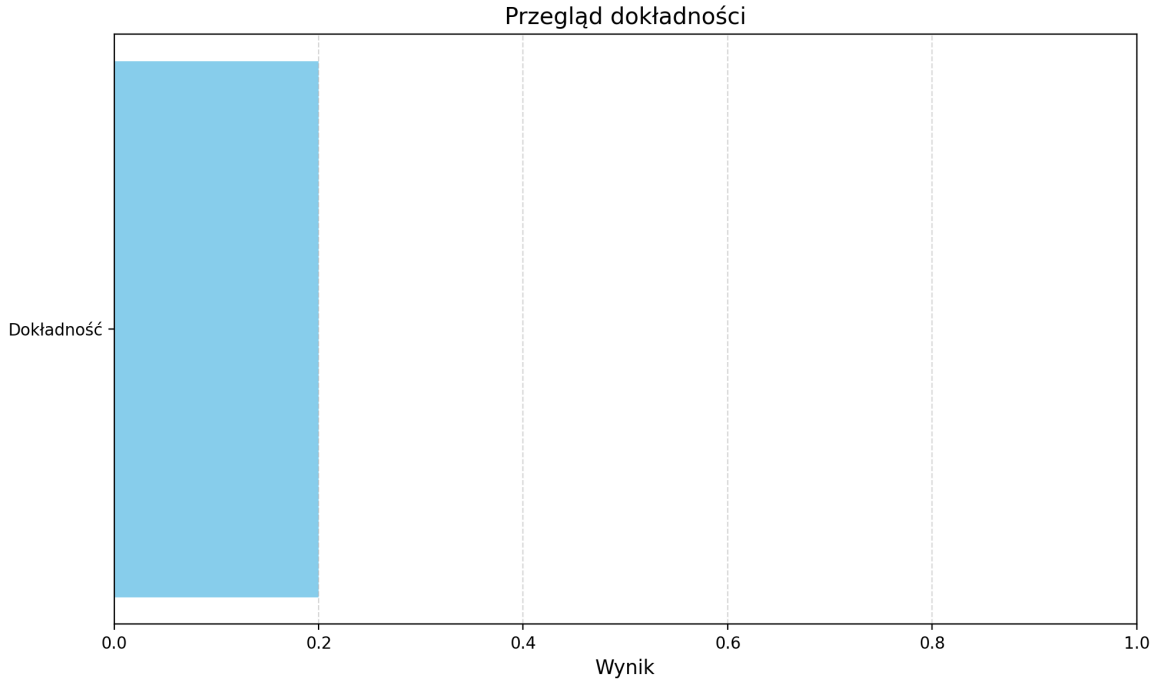
Porównując wyniki różnych metryk, zauważamy, że niski wynik accuracy (2.3%) znajduje odzwierciedlenie w metrykach BLEU i ROUGE, które również wskazują na słabe dopasowanie odpowiedzi do referencji. Wspólnym mianownikiem problemów jest brak precyzji, trudności w zachowaniu struktury referencyjnej oraz nadmierna długość odpowiedzi, co znacząco obniża wartości metryk opartych na zgodności tekstowej.

Podsumowując, wyniki te wskazują na konieczność dalszego usprawnienia modelu Qwen2.5 zarówno w zakresie jego dostosowania do generowania zwięzłych odpowiedzi, jak i poprawy integracji z mechanizmem RAG. Model wymaga precyzyjniejszego dostrojenia do specyfiki zadania oraz lepszego wykorzystania kontekstu zwracanego przez RAG, aby osiągnąć większą zgodność odpowiedzi z oczekiwaniami.

10.3 Model z LoRA

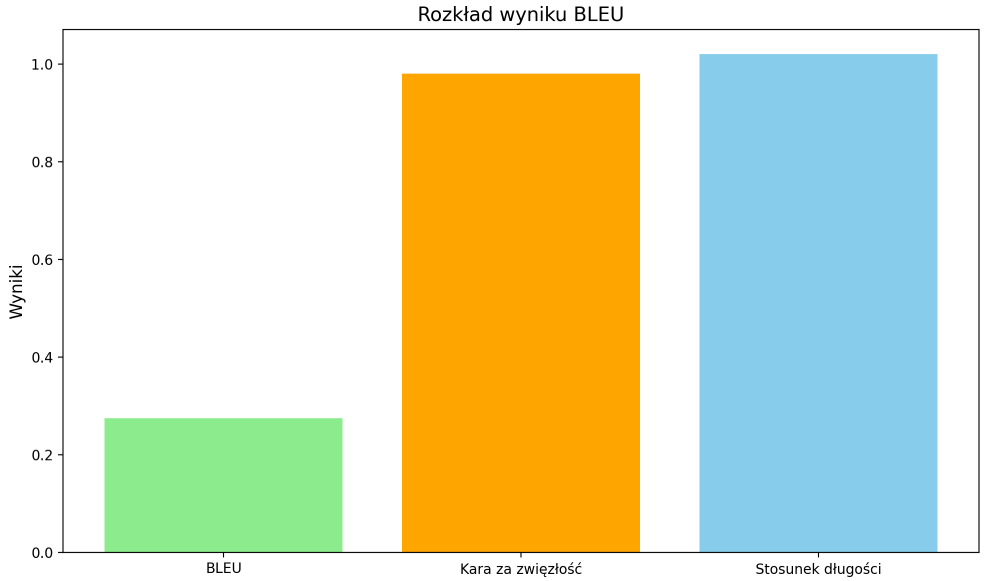
Model bazowy dotrenowany metodą LoRA (Low-Rank Adaptation) wykazuje istotną poprawę w wielu metrykach ewaluacyjnych, co sugeruje, że metoda ta skutecznie zwiększa zdolność modelu do generowania odpowiedzi bliskich referencyjnym pod względem poprawności, zwięzłości i struktury. Wyniki wskazują na znacznie lepszą zgodność z danymi referencyjnymi w porównaniu do poprzednich wariantów.

Accuracy widoczne na rys. 10.9 wynosi 0.2, czyli 20%, co oznacza, że co piąta odpowiedź modelu była w pełni zgodna z referencjami. To znaczący wzrost precyzji generowanych odpowiedzi w porównaniu z wcześniejszymi rezultatami. Wartość ta pokazuje, że model lepiej rozumie wymagania zadania i dostosowuje swoje odpowiedzi do oczekiwanej formy.



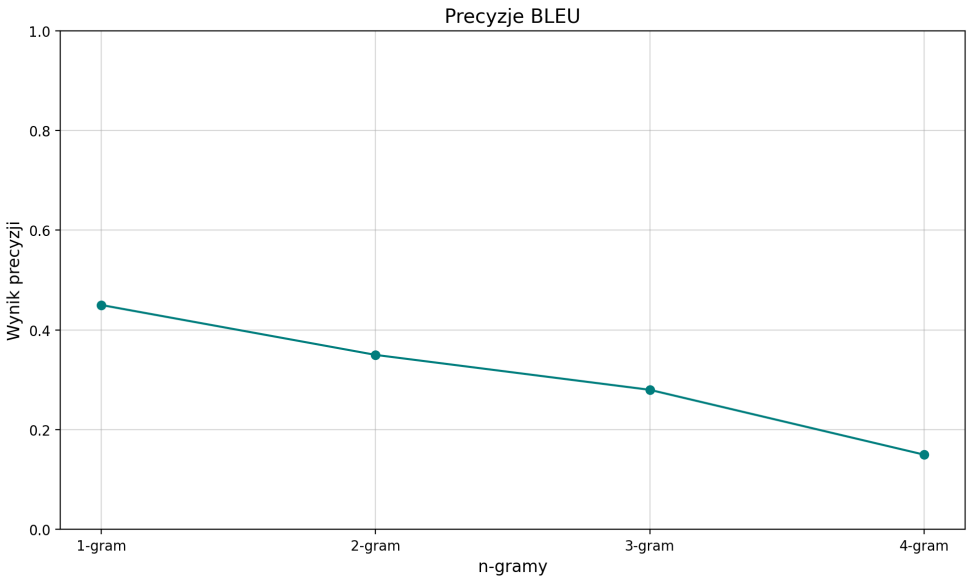
Rys. 10.9. Dokładność modelu z LoRA (Źródło: opracowanie własne)

Wynik BLEU widoczny na rys. 10.10 wynosi 0.275 wskazuje na całkiem dobrą zgodność odpowiedzi modelu z referencyjnymi wzorcami, zarówno na poziomie pojedynczych słów, jak i dłuższych n-gramów.



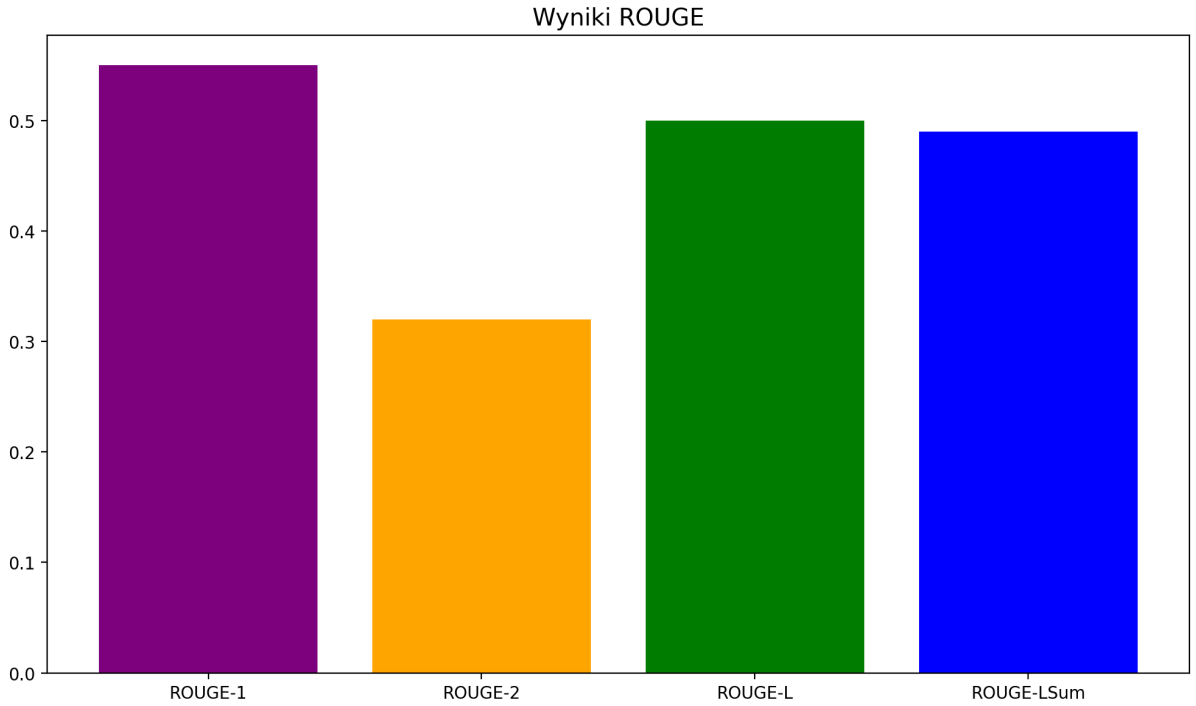
Rys. 10.10. Rozkład wyników BLEU dla modelu z LoRA (Źródło: opracowanie własne)

Precyzja dla 1-gramów widoczne na rys. 10.11 osiąga 45%, co sugeruje, że niemal połowa słów w odpowiedziach modelu jest poprawna. Spada ona dla dłuższych n-gramów (np. 15% dla 4-gramów), co jest typowe dla modeli generatywnych, ale utrzymuje relatywnie wysokie wartości w porównaniu do wyników BLEU w innych konfiguracjach.



Rys. 10.11. Precyzje n-gramów BLEU dla modelu z LoRA (Źródło: opracowanie własne)

Metryki ROUGE widoczna na rys. 10.12 również wskazują na znacznie lepsze wyniki. ROUGE-1 wynosi 0.55 (55%), co oznacza, że ponad połowa tokenów referencyjnych znajduje swoje odpowiedniki w generowanych odpowiedziach. ROUGE-2, na poziomie 0.32 (32%), potwierdza, że model zachowuje znaczną część poprawnych bigramów, a ROUGE-L (50%) oraz ROUGE-Lsum (49%) pokazują, że struktura odpowiedzi jest dobrze dopasowana do referencji zarówno na poziomie zdań, jak i całych tekstów. Te wyniki wskazują, że odpowiedzi są spójne, a model lepiej radzi sobie z organizacją treści.



Rys. 10.12. Wyniki ROUGE dla modelu z LoRA (Źródło: opracowanie własne)

10.4 Porównanie wyników każdej z metod

10.4.1 Dokładność

Tabela 10.1. Wyniki dokładności rozważanych metod dla całego zbioru testowego

Model	**Dokładność**
Model bazowy	0.59%
Model z LoRA	20%
Model z RAG	2.3%

Tabela 10.2. Wyniki walidacji krzyżowej dokładności dla dziesięciu podzbiorów zbioru testowego

Model	**Dokładność (średnia ± odchylenie standardowe)**
Model bazowy	0.0057 ± 0.0004
Model z LoRA	0.19 ± 0.01

Jak widać w tabeli 10.1 i 10.2, model bazowy osiągnął bardzo niski wynik accuracy na poziomie 0.0059 (około 0.6%), co oznacza, że jedynie niewielki odsetek odpowiedzi był w pełni poprawny. Model wspomagany RAG poprawił accuracy do 0.0231 (około 2.3%), ale dopiero zastosowanie metody LoRA przyniosło znaczącą poprawę, osiągając wynik 0.20 (20%). Jest to znacząca poprawa wynikającą najpewniej z tego, że model nauczył się w oczekiwany sposób odpowiadać na pytania - krótką i zwięzłą odpowiedzią.

10.4.2 BLEU

Tabela 10.3. Wyniki BLEU

Metryka	**Model bazowy**	**Model z RAG**	**Model z LoRA**
BLEU	0.0081	0.0109	0.275
Brevity Penalty	1.0	1.0	0.98
Length Ratio	5.01	1.69	1.02
BLEU 1-gram	6.15%	7.81%	45%
BLEU 4-gram	0.14%	0.18%	14%

Tabela 10.4. Wyniki walidacji krzyżowej BLEU dla dziesięciu podzbiorów zbioru testowego

Metryka (średnia ± odchylenie standardowe)	**Model bazowy**	**Model z LoRA**
BLEU	0.0079 ± 0.0004	0.261 ± 0.015
Brevity Penalty	1.0 ± 0	0.97 ± 0.01
Length Ratio	5.00 ± 0.02	1.02 ± 0.01
BLEU 1-gram	0.060 ± 0.002	0.42 ± 0.06
BLEU 4-gram	0.0013 ± 0.0001	0.12 ± 0.3

Wyniki BLEU, przedstawione w tabeli 10.3 i 10.4, podkreślają wyraźną różnicę w jakości generowanych odpowiedzi. Model bazowy osiągnął BLEU na poziomie 0.0081, co wskazuje na bardzo niską zgodność odpowiedzi z referencjami. Model z RAG nieznacznie poprawił ten wynik do 0.0109, ale to model z LoRA osiągnął BLEU 0.275, co jest ogromnym skokiem jakościowym. Widać również, że metoda LoRA zachowuje lepszy balans długości odpowiedzi (length ratio 1.02) i lepsze wyniki w precyzji n-gramów, szczególnie dla 1-gramów (45%) i 4-gramów (15%). Brevity penalty w przypadku LoRA wynosi 0.98, co oznacza, że odpowiedzi są zbliżone długością do referencyjnych, podczas gdy model bazowy ma tendencyjnie zbyt rozwlekłe odpowiedzi (length ratio 5.01).

10.4.3 ROUGE

Tabela 10.5. Wyniki ROUGE

Metryka	**Model bazowy**	**Model z RAG**	**Model z LoRA**
ROUGE-1	0.0974	0.1135	0.55
ROUGE-2	0.0277	0.0235	0.32
ROUGE-L	0.1110	0.1111	0.51
ROUGE-Lsum	0.1111	0.1090	0.50

Tabela 10.6. Wyniki ROUGE

Metryka (średnia ± odchylenie standardowe)	**Model bazowy**	**Model z LoRA**
ROUGE-1	0.0960 ± 0.002	0.52 ± 0.04
ROUGE-2	0.0277 ± 0.004	0.32 ± 0.02
ROUGE-L	0.1109 ± 0.002	0.49 ± 0.01
ROUGE-Lsum	0.1112 ± 0.003	0.50 ± 0.01

Jak pokazano w tabeli 10.5 i 10.6, model bazowy uzyskał niskie wyniki w metrykach ROUGE (ROUGE-1: 0.0974, ROUGE-2: 0.0277), co świadczy o jego ograniczonej zdolności do generowania spójnych odpowiedzi na poziomie słów, fraz i struktury. Model z RAG miał

porównywalne wyniki w tych metrykach, co sugeruje, że dodatkowy kontekst nie wpłynął znacząco na poprawę ich jakości. Natomiast model z LoRA osiągnął najwyższe wyniki: ROUGE-1 na poziomie 0.55 i ROUGE-2 na poziomie 0.32, co wskazuje na wyraźną poprawę w uchwyceniu poprawnych fraz. Wyniki ROUGE-L (0.50) oraz ROUGE-Lsum (0.49) potwierdzają, że model z LoRA generuje odpowiedzi o lepszej organizacji i zgodności strukturalnej.

10.4.4 Podsumowanie porównania

Zestawienie wyników wskazuje, że metoda LoRA znacząco przewyższa zarówno model bazowy, jak i model wspomagany RAG, w niemal wszystkich analizowanych metrykach. Model z LoRA wykazuje największą zgodność z referencyjnymi wzorcami zarówno pod względem dokładności (accuracy), jak i zgodności leksykalnej (BLEU) oraz strukturalnej (ROUGE). Tabele 10.1, 10.2, 10.3, 10.4, 10.5, 10.6 jasno pokazują, że model dotrenowany metodą LoRA lepiej spełnia wymagania zadania, generując bardziej precyzyjne i zwięzłe odpowiedzi, podczas gdy inne podejścia borykają się z problemami rozwlekłości i niskiej trafności. Wynika to najprawdopodobniej z tego, że niewielki model nauczył się odpowiadać w konkretny sposób tzn. krótko i zwięźle oraz całkiem nieźle przyswoił dane zawarte w zbiorze danych.

10.5 Czasy inferencji (wnioskowania)

Aby jednoznacznie określić który z wariantów najszybciej radzi sobie z **inferencją,** utworzony został przypadek testowy podczas którego dokonujemy inferencji na stu tych samych zapytaniach ze zbioru testowego. Wyniki tej analizy znajdują się w tabeli 10.7. Jak widać model bazowy oraz model bazowy + LoRA mają zbliżone czasy inferencji natomiast wariant modelu z RAG zajmuje znacząco więcej czasu.

Tabela 10.7. Percentyle czasu wykonania dla rozważanych metod

Model	**p50**	**p90**	**p95**	**p99**
Model Podstawowy	0.33s	0.59s	0.65s	1.19s
Model + RAG	11.67s	13.12s	14.42s	29.36s
Model + LoRA	0.35s	0.60s	0.67s	1.21s

10.6 Zużycie pamięci

Podczas testów szybkości inferencji brane było również pod uwagę zużycie pamięci dla każdego z wariantów. Podobnie do testów czasu wnioskowania użyto stu tych samych zapytań ze zbioru testowego. Było ono co do zasady stałe z niewielkimi skokami, tabela 10.8 prezentuje zużycie pamięci dla rozważanych metod.

Tabela 10.8. Średnie i maksymalne zużycie pamięci podczas testowania

Model	**Średnie (100 przypadków testowych)**	**Maksymalne**
Model Podstawowy	7.11 GB	7.51 GB
Model + RAG	7.60 GB	9.11 GB
Model + LoRA	7.15 GB	7.60 GB

PODSUMOWANIE

Podjęte w niniejszej pracy działania miały na celu minimalizację halucynacji w dużych modelach językowych poprzez zastosowanie technik RAG (Retrieval-Augmented Generation) oraz LoRA (Low-Rank Adaptation). Analiza wyników pozwoliła na kompleksowe porównanie skuteczności i efektywności poszczególnych metod w kontekście jakości generowanych odpowiedzi, czasu inferencji oraz zużycia pamięci. W odniesieniu do dokładności (accuracy), metoda LoRA okazała się najbardziej efektywna, osiągając wynik 20%, co stanowi znaczący postęp w porównaniu z modelem bazowym (0.6%) i RAG (2.3%). Podobną przewagę odnotowano w metrykach BLEU i ROUGE, gdzie LoRA wyraźnie poprawiła zgodność leksykalną i strukturalną odpowiedzi w porównaniu z innymi wariantami. Co istotne, LoRA zachowała balans między precyzją a zwięzłością odpowiedzi, co było widoczne w długości generowanych tekstów oraz ich podobieństwie do referencji.

Przeprowadzone testy szybkości inferencji wykazały, że LoRA nie wpłynęła negatywnie na czas wykonywania obliczeń, który pozostał zbliżony do modelu bazowego i wynosił około 0.35–1.21 sekundy w różnych percentylach. W przeciwieństwie do tego, wariant z RAG charakteryzował się znacznie dłuższym czasem przetwarzania (p99 = 29.36 sekundy), co ogranicza jego praktyczne zastosowanie w zadaniach wymagających dużej responsywności. Zużycie pamięci również było korzystne dla LoRA, utrzymując się na poziomie bliskim modelu bazowego (średnie zużycie: 7.15 GB, maksymalne: 7.60 GB), co stanowiło wyraźną przewagę nad RAG, który wymagał znacznie większych zasobów (do 9.11 GB). Wyniki te potwierdzają, że metoda LoRA stanowi optymalne rozwiązanie pod względem kompromisu między jakością generowanych odpowiedzi, czasem obliczeń oraz efektywnością zasobów. W związku z tym zastosowanie LoRA w minimalizacji halucynacji wydaje się najbardziej obiecujące, oferując zarówno wyraźną poprawę wyników, jak i praktyczną użyteczność w rzeczywistych scenariuszach zastosowań.

W ramach tego eksperymentu użyto stosunkowo niewielkiego modelu językowego który posiadał 1.5 miliarda parametrów. Test tak małego modelu wynikał z ograniczeń sprzętowych jakie autor tej pracy posiadał. Z racji, że model sam w sobie nie jest duży w porównaniu do innych zdecydowanie większych wariantów jego możliwości ekstrakcji istotnych danych z kontekstu jest ograniczony co widzimy po testach modelu z metodą RAG, nadal są to wyniki lepsze od modelu bazowego natomiast nadal niezadowalające.

Na podstawie przeprowadzonych testów można stwierdzić, że metoda LoRA wykazuje największy potencjał w minimalizacji halucynacji dla stosunkowo niewielkiego LLM, łącząc wysoką jakość generowanych odpowiedzi z efektywnością obliczeniową. Jej zastosowanie pozwoliło na znaczące zwiększenie dokładności odpowiedzi i zgodności strukturalnej przy jednoczesnym utrzymaniu niskiego zużycia zasobów sprzętowych.

Z kolei metoda RAG, mimo poprawy wyników względem modelu bazowego, okazała się mniej praktyczna ze względu na wydłużony czas inferencji i większe zapotrzebowanie na pamięć, co czyni ją trudniejszą do zastosowania w środowiskach o ograniczonych zasobach lub wymagających dużej responsywności, nie został także wykorzystany pełny potencjał tej

metody najpewniej z racji użycia stosunkowo niewielkiego modelu. Wyniki sugerują, że metoda ta może być bardziej odpowiednia w scenariuszach, gdzie kluczowa jest precyzja wyników oparta na dostępie do zewnętrznych danych, ale mniej istotny jest czas przetwarzania.

Ostatecznie wybór odpowiedniej techniki zależy od specyficznych wymagań danego zastosowania, jednak w testowanych warunkach LoRA okazała się rozwiązaniem najlepiej dostosowanym do optymalizacji balansu między jakością, szybkością i efektywnością zasobów.

BIBLIOGRAFIA

[1] Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Łukasz; Polosukhin, Illia (2017). "Attention is All you Need". *Advances in Neural Information Processing Systems*. 30. Curran Associates, Inc, arXiv:1706.03762.

[2] HU, Edward J., et al. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[3] DETTMERS, Tim, et al. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 2024, 36.

[4] CHEN, Yukang, et al. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*, 2023.

[5] SHENG, Ying, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[6] LEWIS, Patrick, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 2020, 33: 9459-9474.

[7] BROWN, Tom, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 2020, 33: 1877-1901.

[8] LEWIS, Mike. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[9] JOSHI, Mandar, et al. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

[10] ChatGPT, https://chatgpt.com/ (data dostepu 05.01.2025 r.).

[11] Gemini, https://gemini.google.com/app (data dostepu 05.01.2025 r.).

[12] ACHIAM, Josh, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[13] RAJPURKAR, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[14] Hugging Face, https://huggingface.co (data dostepu 05.01.2025 r.).

[15] Qwen/Qwen2-1.5B-Instruct · Hugging Face, https://huggingface.co/Qwen/Qwen2-1.5B-Instruct (data dostepu 05.01.2025 r.).

[16] GGUF, https://github.com/ggerganov/ggml/blob/master/docs/gguf.md (data dostepu 05.01.2025 r.).

[17] MALKOV, Yu A.; YASHUNIN, Dmitry A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018, 42.4: 824-836. arXiv:1603.09320.

[18] DOUZE, Matthijs, et al. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[19] KWIATKOWSKI, Tom, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 2019, 7: 453-466, https://research.google/pubs/natural-questions-a-benchmark-for-question-answering-research/.

[20] Mastering Retrieval-Augmented Generation (RAG) Architecture: Unleash the Power of Large Language Models, https://blog.stackademic.com/mastering-retrieval-augmented-generation-rag-architecture-unleash-the-power-of-large-language-a1d2be5f348c (data dostepu 05.01.2025 r.).

[21] WANG, Yingfan, et al. Understanding how dimension reduction tools work: an empirical approach to deciphering t-SNE, UMAP, TriMAP, and PaCMAP for data visualization. *Journal of Machine Learning Research*, 2021, 22.201: 1-73, https://jmlr.org/papers/v22/20-1061.html.

[22] LAHITANI, Alfirna Rizqi; PERMANASARI, Adhistya Erna; SETIAWAN, Noor Akhmad. Cosine similarity to determine similarity measure: Study case in online essay assessment. In: *2016 4th International conference on cyber and IT service management*. IEEE, 2016. p. 1-6. https://ieeexplore.ieee.org/document/7577578.

SPIS RYSUNKÓW

SPIS TABEL

DODATKI

Dodatek A: Skrypt inicjalizujący i procesujący zbiór danych

```
import datasets
import pandas as pd
from transformers import AutoTokenizer
from typing import Tuple

PROMPT_TEMPLATE = [
	{
	"role": "user",
	"content": """Context: {context}\nHere is the question you need to
answer. Question: {question}\nInstruction: Answer with the shortest, most
precise answer possible. Example: When did the first world war started?\nAnswer:
1914"""
	}
]
DATASET = "rajpurkar/squad"
LLM = "Qwen/Qwen2.5-1.5B-Instruct"
MAX_PROMPT_LEN = 460


def prepare_dataset_for_training(tokenize: bool = True) ->
Tuple[datasets.Dataset, datasets.Dataset]:
	dataset = datasets.load_dataset(DATASET, split='train')
	tokenizer = AutoTokenizer.from_pretrained(LLM)
	dataset = datasets.Dataset.from_pandas(
	pd.DataFrame(dataset)
		.drop_duplicates(['context'])
	)
	dataset = dataset.filter(lambda x: len(tokenizer.encode(x['context'])) <
MAX_PROMPT_LEN)
	print(f'Lenght of filterd dataset: {len(dataset)}')
	template = tokenizer.apply_chat_template(PROMPT_TEMPLATE, tokenize=False,
add_generation_prompt=True)

	def map_chat_template_to_question(sample):
	q_c = template.format(context=sample['context'],
question=sample['question'])
	targets = sample.get("answers", {}).get("text", [""])[0]
	return {'q_c': q_c, 'target_text': targets}

	def tokenize_function(examples):
	q_c = template.format(context=examples['context'],
question=examples['question'])
	targets = examples.get("answers", {}).get("text", [""])[0]
	model_inputs = tokenizer(q_c, max_length=512, truncation=True,
padding="max_length", return_tensors='pt')
	labels = tokenizer(targets, max_length=512, truncation=True,
```

```
padding="max_length", return_tensors='pt')['input_ids'].squeeze(0)
        model_inputs['labels'] = labels
        model_inputs['input_ids'] = model_inputs['input_ids'].squeeze(0)
        return model_inputs

        if tokenize:
        dataset = dataset.map(tokenize_function, num_proc=12,
remove_columns=dataset.column_names)
        dataset.set_format(type="torch", columns=["input_ids", "labels"])
        dataset = dataset.train_test_split(0.2, shuffle=True, seed=2137)

        return dataset['train'], dataset['test']

        return dataset['train'], dataset['test']

if __name__ == '__main__':
        train, test = prepare_dataset_for_training()
        print(len(train), len(test))
        print(train[1])
```

Dodatek B: Skrypt tworzący wektorową bazę danych oraz inicjalizujący model z RAG

```
import datasets, os
import torch
from tqdm import tqdm
from langchain_community.vectorstores.utils import DistanceStrategy
from langchain_core.documents import Document as LangchainDocument
from langchain_community.vectorstores import FAISS
from langchain_huggingface import HuggingFaceEmbeddings
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline, Pipeline
from typing import Tuple, List

DATASET = "rajpurkar/squad"
LLM = "Qwen/Qwen2.5-1.5B-Instruct"
EMBEDDING_MODEL = "thenlper/gte-base"
FAISS_PATH = "faiss_store"
DEVICE = 'mps' if torch.backends.mps.is_available() else 'cpu'

def load_dataset(split: str, num_tokens: int) -> datasets.Dataset:
        ds = datasets.load_dataset(DATASET, split=split)
        tokenizer = AutoTokenizer.from_pretrained(LLM)
        ds = ds.filter(lambda sample: len(tokenizer.encode(sample['context']) <
num_tokens))
        return ds

def create_vector_db(dataset: datasets.Dataset=None, path: str = FAISS_PATH) ->
Tuple[FAISS, HuggingFaceEmbeddings]:
        embedding_model = HuggingFaceEmbeddings(
        model_name=EMBEDDING_MODEL,
```

```
        show_progress=True,
        multi_process=True,
        encode_kwargs={"normalize_embeddings": True},
        )
        if os.path.isdir(path):
        return FAISS.load_local(path, embedding_model,
allow_dangerous_deserialization=True), embedding_model

        if dataset is None:
        raise ValueError("Could not find local vector db and the dataset was not
provided!")

        docs = [
        LangchainDocument(
                page_content=doc['context'],
                metadata={'title': doc['title']}
        ) for doc in tqdm(dataset)
        ]
        return FAISS.from_documents(
        docs, embedding_model, distance_strategy=DistanceStrategy.COSINE
        ), embedding_model

def get_llm() -> Tuple[AutoTokenizer, Pipeline]:
        tokenizer = AutoTokenizer.from_pretrained(LLM)
        model = AutoModelForCausalLM.from_pretrained(
        LLM,
        torch_dtype=torch.bfloat16,
        device_map=DEVICE,
        )
        model.eval()
        llm_pipeline = pipeline(
        model=model,
        tokenizer=tokenizer,
        task="text-generation",
        do_sample=True,
        temperature=0.8,
        repetition_penalty=1.1,
        return_full_text=False,
        max_new_tokens=5,
        )
        return tokenizer, llm_pipeline

def apply_prompt(tokenizer: AutoTokenizer):
        template = [
        {
                "role": "user",
                "content": """Context: {context}\nHere is the question you need to
answer. Question: {question}\nInstruction: Answer with the shortest, most
precise answer possible. Example: When did the first world war started?\nAnswer:
1914"""
```

```
        }
        ]
        return tokenizer.apply_chat_template(
        template, tokenize=False, add_generation_prompt=True
        )

def select_documents(documents: List[LangchainDocument], tokenizer:
AutoTokenizer) -> int:
        MAX_TOKENS = 512 - 52
        current, count = 0, 0
        for doc in documents:
        current += int(tokenizer(doc.page_content,
return_tensors="pt")['input_ids'].shape[-1])
        if current < MAX_TOKENS:
                count += 1
        else:
                return count
        return count

def concat_context_documents(documents: list) -> str:
        return "".join([f"Document {i}:::\n{doc.page_content}\n" for i, doc in
enumerate(documents, 1)])

def prepare_docs(documents: List[LangchainDocument], tokenizer: AutoTokenizer)
-> str:
        num_docs = select_documents(documents, tokenizer)
        return concat_context_documents(documents[: num_docs])


def create_prompt(question: str, vector_db: FAISS, template: str, tokenizer:
AutoTokenizer) -> Tuple[str, List[LangchainDocument]]:
        simmilar_topics = vector_db.similarity_search(question, k=5)
        context_documents = prepare_docs(simmilar_topics, tokenizer)
        return template.format(question=question, context=context_documents),
context_documents

def run(pipeline: Pipeline, question: str, vector_db: FAISS, tokenizer:
AutoTokenizer, template: str):
        prompt, relevant_docs = create_prompt(question, vector_db, template,
tokenizer)
        llm_response = pipeline(prompt)[0]['generated_text']
        return llm_response, relevant_docs



if __name__ == "__main__":
        db, embedding_model = create_vector_db()
        tokenizer, llm_pipeline = get_llm()
        prompt_template = apply_prompt(tokenizer)
        while True:
```

```
        question = input("Provide question:")
        print(run(llm_pipeline, question, db, tokenizer, prompt_template))
        print(llm_pipeline(question)[0]['generated_text'])
```

Dodatek C: Skrypt inicjalizujący model z LoRA

```
from peft import LoraConfig, TaskType, PeftModel, get_peft_model
from transformers import AutoModelForCausalLM
import torch

LLM = "Qwen/Qwen2.5-1.5B-Instruct"

def get_lora(predtrained_path: str = None, train: bool=True) -> PeftModel:
        if predtrained_path is not None:
        model = AutoModelForCausalLM.from_pretrained(LLM,
device_map='auto',torch_dtype=torch.bfloat16)
        peft_model = PeftModel.from_pretrained(model, predtrained_path,
is_trainable=train)
        else:
        lora_config = LoraConfig(
                r=16,
                task_type=TaskType.CAUSAL_LM,
                lora_dropout=0.1,
                lora_alpha=32,
                target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
"gate_proj", "up_proj", "down_proj"],
                bias="all",
        )
        model = AutoModelForCausalLM.from_pretrained(LLM,
device_map='auto',torch_dtype=torch.bfloat16)
        peft_model = get_peft_model(model, lora_config)
        print(
        f"\n\nMemory footprint of model {LLM} with LORA:"
        + f" {peft_model.get_memory_footprint()*1e-9:.2f} GB\n\n"
        )
        num_params = peft_model.get_nb_trainable_parameters()
        print(
        f"Number of parameters for {LLM} with LORA\n\t\t-> trainable:
{num_params[0]/1_000_000:,.2f}M\n\t\t-> all parameters:
{num_params[1]/1_000_000_000:,.2f}B"
        )
        return peft_model

if __name__ == '__main__':
        get_lora()
```

Dodatek D: Skrypt dotrenowujący dla modelu bazowego oraz modelu z LoRA

```
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments, AutoTokenizer
```

```
from dataset import prepare_dataset_for_training
from lora.config import get_lora
import uuid, os, json

def train():
	out_dir = f"./lora/model/lora-{uuid.uuid4()}"

	train, val = prepare_dataset_for_training()
	model = get_lora()

	training_args = Seq2SeqTrainingArguments(
	learning_rate=3e-3,
	lr_scheduler_type="cosine",
	num_train_epochs=1,
	per_device_train_batch_size=4,
	per_device_eval_batch_size=4,
	gradient_accumulation_steps=8,
	output_dir="./lora/train",
	optim="adamw_torch",
	logging_steps=5,
	logging_strategy="steps",
	logging_first_step=True,
	save_strategy="epoch",
	load_best_model_at_end=True,
	metric_for_best_model="eval_loss",
	greater_is_better=False,
	eval_strategy="epoch",
	bf16=True
	)
	trainer = Seq2SeqTrainer(
	model=model,
	args=training_args,
	train_dataset=train,
	eval_dataset=val,
	)
	print("Training...")
	try:
	trainer.train()
	except KeyboardInterrupt:
	pass

	print("Saving model...")
	trainer.save_model(out_dir)
	trainer.save_state()

	print("Saving training metrics...")

	metrics = trainer.state.log_history
	metrics_path = os.path.join(training_args.output_dir, "all_metrics.json")
```

```
        with open(metrics_path, "w") as f:
        json.dump(metrics, f, indent=4)
        print(f"Metrics saved to {metrics_path}")

if __name__ == '__main__':
        train()
```

Dodatek E: Skrypt ewaluacyjny

```
import os
import json
import numpy as np
import matplotlib.pyplot as plt
import torch
from tqdm import tqdm
from timeit import default_timer as timer
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline
from datasets import load_dataset
from evaluate import load
from rag import config
from lora import config as lora
from sklearn.model_selection import KFold

DATASET = "rajpurkar/squad"
MAX_PROMPT_LEN = 460
MODEL_PATH = "Qwen/Qwen2.5-1.5B-Instruct"
KFOLDS = 10

def load_model(model_type):
        tokenizer = AutoTokenizer.from_pretrained(MODEL_PATH)
        if model_type == "base":
        model = AutoModelForCausalLM.from_pretrained(
                MODEL_PATH,
                torch_dtype=torch.bfloat16,
                device_map="auto"
        )
        pipeline_model = pipeline(
                model=model,
                tokenizer=tokenizer,
                task="text-generation",
                return_full_text=False,
                max_new_tokens=16
        )
        return pipeline_model, tokenizer, None
        elif model_type == "lora":
        pipeline_model = lora.get_lora(pretrained_path="./lora/model",
train=False)
        return pipeline_model, tokenizer, None
        elif model_type == "rag":
        db, embedding_model =
```

```
config.create_vector_db(path=os.path.join(os.getcwd(), "rag", "faiss_store"))
	tokenizer, pipeline_model = config.get_llm()
	prompt_template = config.apply_prompt(tokenizer)
	return pipeline_model, tokenizer, db
	else:
	raise ValueError(f"Nieobsługiwany typ modelu: {model_type}")

def evaluate_fold(pipeline_model, tokenizer, db, test_subset, model_type):
	rouge = load("rouge")
	bleu = load("bleu")
	bertscore = load("bertscore")

	results, labels, times = [], [], []
	with torch.no_grad():
	for example in tqdm(test_subset):
		t0 = timer()
		if model_type == "rag":
		output, _ = config.run(pipeline_model, example["question"], db,
tokenizer, config.apply_prompt(tokenizer))
		else:
		prompt = f"Question: {example['question']}\nAnswer:"
		output = pipeline_model(prompt)[0]["generated_text"]
		results.append(output)
		labels.append(example["answers"]["text"])
		t1 = timer()
		times.append(t1 - t0)

	rouge_res = rouge.compute(predictions=results, references=labels)
	bleu_res = bleu.compute(predictions=results, references=labels)
	bertscore_res = bertscore.compute(predictions=results, references=labels,
lang="en")

	accuracy = sum([1 for res, label in zip(results, labels) if res in
label]) / len(results)

	metrics = {
	"rouge": rouge_res,
	"bleu": bleu_res,
	"accuracy": accuracy,
	"bertscore": bertscore_res,
	"times": times,
	}
	return metrics

def run_cross_validation(model_type):
	pipeline_model, tokenizer, db = load_model(model_type)
	test = load_dataset(DATASET, split="validation").shuffle(seed=42)
	test = test.select(range(len(test)//2))
	kf = KFold(n_splits=KFOLDS)
```

```
        metrics_list = []
        for train_idx, test_idx in tqdm(kf.split(test)):
        test_subset = test.select(test_idx.tolist())
        metrics = evaluate_fold(pipeline_model, tokenizer, db, test_subset,
model_type)
        metrics_list.append(metrics)
        break

        aggregated_results = aggregate_metrics(metrics_list)
        return aggregated_results

def aggregate_metrics(metrics_list):
        aggregated = {}
        for key in metrics_list[0]:
        if isinstance(metrics_list[0][key], dict):
                aggregated[key] = {subkey: compute_mean_std([m[key][subkey] for m
in metrics_list]) for subkey in metrics_list[0][key]}
        elif isinstance(metrics_list[0][key], list):
                flat_list = [item for m in metrics_list for item in m[key]]
                aggregated[key] = {"mean": np.mean(flat_list), "std":
np.std(flat_list)}
        else:
                aggregated[key] = compute_mean_std([m[key] for m in metrics_list])
        return aggregated

def compute_mean_std(values):
        return {"mean": np.mean(values), "std": np.std(values)}

def save_results(eval_dict, model_type):
        json_path = f"eval_{model_type}_cross_validation.json"
        with open(json_path, "w") as f:
        json.dump(eval_dict, f, indent=4)
        print(f"Wyniki zapisane do {json_path}")

        metrics = eval_dict

        plt.figure(figsize=(10, 6), dpi=200)
        plt.barh(["Dokładność"], [metrics["accuracy"]["mean"]], color="skyblue")
        plt.title("Przegląd dokładności", fontsize=14)
        plt.xlabel("Wynik", fontsize=12)
        plt.xlim(0, 1)
        plt.tight_layout()
        plt.savefig(f"accuracy_{model_type}_cross_validation.png")
        plt.close()

        print(f"Wykresy zapisane dla modelu {model_type}")

if __name__ == "__main__":
        import argparse
        parser = argparse.ArgumentParser()
```

```
        parser.add_argument("--model", choices=["base", "lora", "rag"],
required=True, help="Wybierz model do ewaluacji.")
        args = parser.parse_args()

        evaluation_results = run_cross_validation(args.model)
        save_results(evaluation_results, args.model)
```