Implementacja transformera oraz eksperymenty z wariantami mechanizmu uwagi (Performer, Reformer) w zadaniach klasyfikacji tekstu

Dokumentacja projektu — część I

Bartłomiej Borycki, Michał Iwaniuk

21.10.2025

1 Cel biznesowy

Projekt ma na celu opracowanie modelu językowego opartego na architekturze enkodera transformera, który łączy rozwiązania z BERT-a i RoBERTy z nowoczesnymi mechanizmami uwagi.

Celem biznesowym projektu jest opracowanie efektywnego kosztowo rozwiązania do klasyfikacji tekstu, które umożliwi wykorzystanie nowoczesnych metod przetwarzania języka naturalnego (NLP) również przy ograniczonych zasobach obliczeniowych. Projekt ma na celu potencjalne obniżenie kosztów trenowania modeli językowych przy zachowaniu jakości wymaganej w zastosowaniach produkcyjnych.

Rozwiązanie ma zastosowanie w zadaniach takich jak moderacja treści, analiza opinii klientów, automatyczna kategoryzacja dokumentów oraz innych zadaniach opartych na klasyfikacji tekstu. Projekt ma również znaczenie badawcze i rozwojowe – pozwala testować różne warianty mechanizmów uwagi i architektur transformera, wspierając rozwój modeli NLP.

Podstawową hipotezą biznesową projektu jest założenie, że poprzez przemyślany dobór alternatywnych mechanizmów uwagi oraz właściwą konfigurację architektury można osiągnąć istotną redukcję kosztu rozwiązania przy jednoczesnym zachowaniu jakości predykcji wymaganej w zastosowaniach produkcyjnych. Całkowity koszt obejmuje zarówno bezpośrednie nakłady finansowe na infrastrukturę i energię elektryczną, jak również czas potrzebny na trening i inferencję.

2 Wizja systemu

Architektura rozwiązania

Projekt jest realizowany w języku Python z wykorzystaniem frameworka PyTorch. Implementacje są wykonane bez użycia wysokopoziomowych skrótów takich jak `nn.TransformerEncoder`.

Komponenty systemu

1. **Mechanizmy uwagi** – zaimplementowane od zera mechanizmy uwagi: scaled_dot_product o złożoności obliczeniowej $O(n^2)$, reformer_lsh o złożoności obliczeniowej $O(n \log n)$ oraz performer_favor o złożoności $O(n)$.
2. **Rdzeń modelu** – pełna implementacja bloku enkodera z wymiennym mechanizmem uwagi. Każdy blok obejmuje wielogłowicową uwagę, warstwę MLP, normalizację oraz połączenia rezydualne.
3. **Warstwa klasyfikacyjna** – pooling reprezentacji tokenów (CLS/mean/max) oraz projekcja liniowa do przestrzeni klas z funkcją softmax.

4. **Głowica MLM** – moduł pretreningowy rekonstruujący zamaskowane tokeny w stylu BERT/RoBERTa.
5. **Tryby uczenia** – pretrening (MLM) oraz fine-tuning (klasyfikacja).
6. **Interfejs treningowy** – skrypty `pretrain.py`, `finetune.py`, `eval.py` z pełną integracją z Weights & Biases, checkpointowaniem i wznowieniem treningu.
7. **Konfiguracja YAML** – zarządzanie parametrami modelu i treningu (architektura modelu, rodzaj uwagi, optymalizator, scheduler, hiperparametry treningu)
8. **Tokenizacja WordPiece** – obsługa tokenów specjalnych (`[CLS]`, `[SEP]`, `[MASK]`, `[PAD]`), dynamiczny padding i przycinanie sekwencji.
9. **Reproduktywność** – deterministyczne ziarna, zapisy konfiguracji, wersji bibliotek i raportów z przebiegu eksperymentów.

3 Wymagania funkcjonalne

WF-1 — Pretrening i fine-tuning

Opis System umożliwia pełny cykl uczenia: pretrening z maskowanym modelowaniem języka (MLM), a następnie fine-tuning do klasyfikacji na danych docelowych z wykorzystaniem checkpointu z pretreningu. Proces jest wznawialny, wersjonowany i logowany.

Wejście/Wyjście

- **Wejście:** korpusy tekstowe (zbiory docelowe + zbiory zewnętrzne), plik YAML z konfiguracją, losowe ziarno, tryb `pretrain` lub `finetune`.
- **Wyjście:** kompletne checkpointy `.pt` obejmujące stan modelu, optymalizatora, schedulera i tokenizera; logi i metryki (CSV/JSON); artefakty W&B; finalne wagi głowicy klasyfikacyjnej.

Kryteria akceptacji

1. Uruchomienie `pretrain.py` z poprawną konfiguracją generuje co najmniej jeden checkpoint co N epok lub M kroków (parametry konfigurowalne) i zapisuje historię strat oraz metryk walidacyjnych.
2. Uruchomienie `finetune.py` wczytuje stan z pretreningu i poprawnie inicjalizuje głowicę klasyfikacyjną zgodnie z konfiguracją.
3. Skrypt `eval.py` wylicza `accuracy`, `macro_F1`, `weighted_F1`, a dla zadań binarnych/wieloklasowych (one-vs-rest) także `AUROC`; wyniki są logowane do W&B.

WF-2 — Wymienne mechanizmy uwagi

Opis Każdy blok enkodera może używać jednego z mechanizmów: `scaled_dot_product`, `reformer_lsh`, `performer_favor`. Wybór odbywa się deklaratywnie w YAML dla całego modelu.

Wejście/Wyjście

- **Wejście:** `model.attention_type` $\in$ {`sdpa`, `reformer_lsh`, `performer_favor`}; parametry specyficzne (np. liczba hashy LSH, rozmiar projekcji random features).
- **Wyjście:** tensor o kształcie `[B, N, D]` oraz raport czasu na krok i zużycia pamięci dla wybranego wariantu.

Kryteria akceptacji

1. Zmiana `model.attention_type` umożliwia trening i inferencję bez modyfikacji kodu bloku enkodera.
2. Maski `key_padding_mask` i `attn_mask` są respektowane przez wszystkie trzy warianty uwagi.
3. Dla losowego wejścia `[B, N, D]` każdy wariant zwraca `[B, N, D]`;

WF-3 — Obsługa długich sekwencji

Opis Model przyjmuje wejścia o długości do 16 k tokenów (konfigurowalne) i wspiera schematy pozycjonowania: sinusoidalne, uczone, względne oraz RoPE. Mechanizmy działają zarówno z pełną, jak i aproksymacyjną uwagą.

Wejście/Wyjście

- **Wejście:** `max_seq_len` $\in [1, 16\text{k}]$; `position_embedding` $\in$ {`sin`, `learned`, `relative`, `rope`}
- **Wyjście:** poprawnie zastosowane osadzenia pozycyjne; wyjątek z komunikatem diagnostycznym przy próbie przekroczenia limitu długości.

Kryteria akceptacji

1. Dla długości z zakresu 4k–16k tokenów przetwarzanie partii kończy się sukcesem bez błędów OOM przy parametrach zasobów z konfiguracji.
2. RoPE i pozycje względne działają z `sdpa`, `reformer_lsh` i `performer_favor`, a maski są stosowane prawidłowo.
3. Test długich sekwencji raportuje metryki walidacyjne oraz czas i zużycie pamięci

WF-4 — Pipeline danych

Opis Dane są przetwarzane przez tokenizację WordPiece ze wsparciem tokenów specjalnych i dynamicznego przygotowania partii (przycinanie w `DataLoader`); implementacja MLM stosuje reguły BERT/RoBERTa.

Wejście/Wyjście

- **Wejście:** surowe rekordy tekstowe, słownik WordPiece
- **Wyjście:** tensory `input_ids`, `attention_mask`, `labels`

Kryteria akceptacji

1. Przycinanie paddingu do najdłuższej sekwencji w partii redukuje średnie zużycie pamięci względem statycznego paddingu.
2. Maskowanie MLM jest zgodne z regułą 80% `[MASK]` / 10% losowy token / 10% oryginalny token.

WF-5 — Konfiguracja i CLI

Opis System wykorzystuje generator konfiguracji pretreningu, który z szablonu tworzy katalog `pretrain/<nazwa_eksperymentu>/` z plikiem `config.yaml` gotowym do ewentualnych modyfikacji. Trening wstępny uruchamia się, podając wyłącznie nazwę eksperymentu; wszystkie artefakty (logi, metryki, checkpointy) trafiają do tego katalogu. Podczas fine-tuningu podaje się *dwie* nazwy: nazwę eksperymentu fine-tuningowego oraz nazwę eksperymentu pretreningowego, z którego mają zostać odziedziczone ustawienia architektury (backbone). W fine-tuningu można zmieniać wyłącznie głowicę klasyfikacyjną (np. `num_classes`, `pooling`), pozostawiając resztę architektury zgodną z pretreningiem.

Wejście/Wyjście

- **Wejście (generator pretrain):** nazwa eksperymentu `<EXP>`; opcjonalnie ścieżka do szablonu. Efekt: `pretrain/<EXP>/config.yaml`.
- **Wejście (pretrain):** nazwa eksperymentu `<EXP>` (odwołanie do `pretrain/<EXP>/config.yaml`).

- **Wejście (finetune):** nazwa eksperymentu FT `<FT_EXP>` oraz nazwa eksperymentu pretrain `<PRE_EXP>`; konfiguracja FT automatycznie dziedziczy architekturę z `pretrain/<PRE_EXP>/config.yaml`, dopuszczając zmianę wyłącznie parametrów głowicy i ustawień treningu klasyfikacji.
- **Wyjście (pretrain):** `pretrain/<EXP>/{config.yaml, logs, metrics.csv, events.jsonl, ckpt_*.pt}`.
- **Wyjście (finetune):** `finetune/<FT_EXP>/{resolved_config.yaml, logs, metrics.csv, events.jsonl, ckpt_*.pt}`; w `resolved_config.yaml` architektura jest *spłaszczona* (po dziedziczeniu) i zawiera referencję do `<PRE_EXP>`.

Kryteria akceptacji

1. Uruchomienie generatora z `<EXP>` tworzy `pretrain/<EXP>/config.yaml` na podstawie szablonu oraz podstawia wartości domyślne (ścieżki danych, hiperparametry, nazwy artefaktów).
2. `python pretrain.py -exp <EXP>` wczytuje `pretrain/<EXP>/config.yaml`, zapisuje checkpointy i metryki w `pretrain/<EXP>/`, a `-resume` w tym katalogu wznawia trening bez utraty postępu (różnica kroków ≤ 1).
3. `python finetune.py -exp <FT_EXP> -pretrain <PRE_EXP>`:
   3.a. odczytuje architekturę (wymiary, warstwy, typ uwagi, pozycjonowanie itp.) z `pretrain/<PRE_EXP>/config.yaml`,
   3.b. pozwala zmienić wyłącznie parametry głowicy klasyfikacyjnej (`num_classes`, `pooling`, `dropout`) oraz ustawienia treningu FT,
   3.c. zapisuje `finetune/<FT_EXP>/resolved_config.yaml` z *pełną*, rozstrzygniętą konfiguracją (po dziedziczeniu) i referencją do `<PRE_EXP>`.
4. Załadowanie wag w FT odbywa się z najlepszego (lub wskazanego) checkpointu z `pretrain/<PRE_EXP>/`; w przypadku niezgodności wymiarów głowicy model automatycznie inicjalizuje nową głowicę i emituje stosowne ostrzeżenie w logach.

WF-6 — Raporty porównawcze

Opis System generuje raporty porównujące warianty uwagi i konfiguracje (jakość, czas/epoka, przepustowość, zużycie pamięci) oraz umożliwia replikację wyników.

Wejście/Wyjście

- **Wejście:** co najmniej dwa zakończone przebiegi z różnymi konfiguracjami; ścieżki do katalogów eksperymentów lub identyfikatory W&B; wyniki baseline'u TF-IDF+LogReg.
- **Wyjście:** tabela `results.csv` (model, metryki, parametry modelu/treningu), inne tabele i wykresy zbiorcze (CSV/PNG/HTML/W&B) z rankingiem według wybranej metryki i kosztów oraz kolumną `PASS_WNF`.

Kryteria akceptacji

1. Uruchomienie `eval.py -compare runs/` generuje raport zawierający najlepszą `macro_F1` z odchyleniem standardowym, `accuracy`, `AUROC` (jeśli dotyczy), `time_per_epoch`, `tokens/s` oraz `max_gpu_mem_MiB`.
2. Raport zawiera metadane konfiguracji (parametry architektury oraz treningu) oraz skrypt `reproduce.sh` umożliwiający replikację.
3. Nieudane przebiegi (np. OOM, NaN loss, przerwanie) są oznaczane, opisane przyczyną i wykluczane z rankingów.
4. Plan testów akceptacyjnych: uruchomienia z `sdpa`, `performer_favor`, `reformer_lsh`; baseline TF-IDF+LogReg trenowany na tych samych danych; raport PASS/FAIL względem WNF.
5. Tabela `SDPA vs Performer vs Reformer vs TF-IDF+LR` z oznaczeniem PASS WNF oraz wykres jakość (np. macro-F1) vs koszt (czas/VRAM).

4 Wymagania niefunkcjonalne

WNF-1 — Wydajność i efektywność zasobowa

System powinien umożliwiać uruchomienie i trenowanie modeli w środowisku Google Colab przy zachowaniu akceptowalnych czasów treningu i niższego zużycia pamięci względem pełnej uwagi.

- **Środowisko GPU (Colab):** docelowo *NVIDIA V100 16 GB HBM2* (Tensor Cores, FP16); alternatywnie *A100 40 GB HBM2e* (jeśli dostępna) lub *T4 16 GB*.
- **Pomiar kosztu:** raportowane są `time_per_epoch` [s], `tokens/s` oraz `max_gpu_mem_MiB`, mierzone przy `seq_len=512`, zadanym `batch_size`.
- **Wymóg kosztowy (Performer):** $time/epoch \leq$ **SDPA** -20% lub $\max VRAM \leq$ **SDPA** -30% przy spadku jakości ≤ 1 pp macro-F1.
- **Wymóg kosztowy (Reformer):** $time/epoch \leq$ **SDPA** -15% lub $\max VRAM \leq$ **SDPA** -25% przy spadku jakości ≤ 1 pp macro-F1.
- Wykorzystywana jest mieszana precyzja obliczeń (FP16/BF16) oraz techniki optymalizacji pamięci, takie jak gradient checkpointing i akumulacja gradientów.

WNF-2 — Jakość, niezawodność i testowalność

System powinien być odporny na błędy w danych oraz zapewniać stabilność procesu uczenia.

- **Jakość (SDPA):** macro-F1 (walidacja) $\geq$ **TF-IDF+LogReg** +5 pp, mierzone na AG News i/lub IMDb, `seq_len=512`, `batch`/tokenizer jak w konfiguracji.
- **Jakość (Performer):** macro-F1 $\geq$ **TF-IDF+LogReg** +3 pp oraz w odległości ≤ 1 pp od SDPA przy tej samej konfiguracji.
- **Jakość (Reformer):** macro-F1 $\geq$ **TF-IDF+LogReg** +3 pp oraz w odległości ≤ 1 pp od SDPA przy tej samej konfiguracji.
- **Stabilność:** brak `NaN`/OOM; 3 różne seedy; rozrzut macro-F1 (max–min) ≤ 1 pp. Raport zawiera GPU, batch, `seq_len=512`.
- Kluczowe komponenty (uwaga, maski, pozycjonowanie, tokenizacja) są objęte testami; logowane są błędy i ostrzeżenia diagnostyczne.
- Model nie generuje wartości `NaN`/`Inf`; w razie naruszenia przebieg jest oznaczany jako nieudany i wykluczany z rankingów.

WNF-3 — Użyteczność i utrzymanie

System powinien być łatwy w konfiguracji, rozbudowie i ponownym uruchomieniu eksperymentów.

- Dokumentacja opisuje sposób przygotowania danych, pretreningu i fine-tuningu krok po kroku.
- Struktura katalogów i plików jest spójna i hierarchiczna.
- Kod jest zgodny ze standardem PEP 8, a kluczowe moduły są opatrzone docstringami.
- Wszystkie hiperparametry są definiowane w pliku YAML, bez konieczności modyfikacji kodu źródłowego.
- README zawiera komende do uruchomienia przebiegu reprodukcji (wykorzystując `config.yaml` i `reproduce.sh`).

WNF-4 — Skalowalność i elastyczność architektury

- Architektura systemu umożliwia dodawanie nowych wariantów mechanizmów uwagi bez ingerencji w istniejące komponenty.
- System wspiera równoległe uruchamianie wielu eksperymentów z różnymi konfiguracjami.
- Konfiguracja modeli powinna być przenośna pomiędzy różnymi środowiskami (Colab, serwer GPU, lokalny PC).

WNF-5 — Przenośność i kompatybilność

- Kod działa w środowiskach Python $\geq$ 3.10 oraz z frameworkiem PyTorch $\geq$ 2.2 (CUDA 11.8/12.x); obsługa AMP (FP16/BF16).
- Biblioteki: scikit-learn (baseline TF-IDF+LogReg), NumPy, PyYAML, HuggingFace (datasets/tokenizer narzędziowo), Weights&Biases.
- Konfiguracja zawiera ustawienia `seed`, `dtype` (FP16/BF16/FP32), informacje o GPU/VRAM i wersjach (Python, PyTorch, CUDA).

WNF-6 — Monitorowanie i obserwowalność

- Wszystkie metryki (loss, accuracy, F1, zużycie pamięci GPU, czas/epoka) są logowane do Weights & Biases.
- System udostępnia możliwość wznowienia eksperymentu od ostatniego checkpointu z zachowaniem historii logów.
- Każdy eksperyment generuje raport w formacie `metrics.csv` oraz agregację do `results.csv` (model, metryki, hiperparametry).

WNF-7 — Aspekty formalne i zgodność z dobrymi praktykami

- Dokumentacja projektu jest zgodna z wytycznymi wydziału dotyczących raportowania projektów inżynierskich.
- Kod źródłowy jest wersjonowany w systemie kontroli wersji (Git) oraz wykorzystuje dobre praktyki inżynierii oprogramowania.

5 Część badawcza

W części badawczej analizowane są małe architektury transformera w warunkach ograniczonych zasobów obliczeniowych. Celem eksperymentów jest zrozumienie wpływu różnych wariantów mechanizmu uwagi oraz konfiguracji modelu na koszt obliczeniowy i jakość uzyskiwanych wyników.

Badania koncentrują się w szczególności na następujących zagadnieniach:

- **Wpływ doboru mechanizmu uwagi na koszt i jakość:** porównanie klasycznej uwagi (*scaled dot-product*) z mechanizmami Reformer LSH i Performer FAVOR+ pod względem szybkości uczenia, jakości w zadaniach klasyfikacyjnych oraz kosztów obliczeniowych (czas treningu, zużycie pamięci). W tabelach wyników uwzględniany jest baseline **TF-IDF+LogReg**.
- **Wpływ długości sekwencji wejściowej:** analiza, jak długość sekwencji (w szczególności długie sekwencje rzędu 8–16k tokenów) wpływa na szybkość i jakość uczenia w zależności od zastosowanego mechanizmu uwagi.
- **Modyfikacje mechanizmów uwagi:** Wprowadzamy zmiany w mechanizmach uwagi (LSH, FAVOR+) i porównujemy ich działanie z oryginalnymi implementacjami, testując zarówno oryginalne mechanizmy, jak i nasze modyfikacje, aby ocenić ich wpływ na wyniki.
- **Analiza hiperparametrów:** badanie wpływu parametrów architektury (np. rozmiar modelu, sposób kodowania pozycji) na jakość uczenia.
- **Ocena kosztów obliczeniowych:** pomiar liczby parametrów do wytrenowania, złożoności obliczeniowej i pamięciowej etapów *forward* i *backward*, a także identyfikacja kompromisu między jakością modelu a kosztem treningu i inferencji.

Środowisko treningowe: eksperymenty uruchamiane na Google Colab. GPU docelowe: **V100 16 GB** (preferowane); alternatywnie **A100 40 GB** (jeśli przydzielone) lub **T4 16 GB** z odpowiednim zmniejszeniem batch size.

6 Analiza ryzyka

Celem analizy jest szybkie wykrywanie i ograniczanie ryzyk poprzez jasne wskaźniki (triggery) oraz gotowe działania.

R1. **Ograniczenia zasobów (czas/VRAM).** *Trigger:* OOM przy $N \geq 16$k. *Mitigacje:* FP16/BF16, gradient checkpointing, akumulacja gradientów, dynamiczny padding. *Plan awaryjny:* skrócenie N, redukcja warstw/głów.

R2. **Niestabilność treningu (NaN, exploding gradients).** *Trigger:* NaN/Inf w stracie lub gradientach, >2 restarty na epokę. *Mitigacje:* gradient clipping, dłuższy warmup, niższy learning rate, loss scaling dla FP16. *Plan awaryjny:* zmiana optymalizatora/schedulera, hiperparametrów treningu.

R3. **Błędy implementacyjne (Reformer/Performer).** *Trigger:* niezgodność kształtów/masek, rozjazd wyników na danych syntetycznych. *Mitigacje:* testy jednostkowe i funkcjonalne, asercje. *Plan awaryjny:* Ponowna implementacja/naprawa wadliwego komponentu, po czym re-walidacja testami.

R4. **Niedopasowanie tokenizera do domeny.** *Trigger:* wysoki udział tokenów `[UNK]`. *Mitigacje:* dostrojenie słownika (dołączenie domenowych subwordów). *Plan awaryjny:* pełny retrening tokenizera na mieszance domenowej.

R5. **Zależności zewnętrzne (Colab, W&B) i awarie sesji.** *Trigger:* przerwane sesje, brak artefaktów/logów. *Mitigacje:* snapshoty lokalne, tryb W&B `-offline`, autosave checkpointów co N minut/kroków. *Plan awaryjny:* uruchomienia lokalne bez W&B i późniejsza synchronizacja.

Ryzyko	**Prawd.**	**Wpływ**	**Mitigacja**	**Właściciel**
OOM przy długich sekwencjach	Średnia	Wysoki	FP16/BF16, checkpointing, akumulacja, redukcja N	Bartłomiej Borycki
NaN/niestabilność treningu	Średnia	Wysoki	Clipping, warmup, LR, loss scaling	Bartłomiej Borycki
Błędy w Reformer/Performer	Średnia	Średni	Testy jednostkowe/funkcjonalne, asercje	Michał Iwaniuk
Awaria sesji Colab/W&B	Niska	Średni	Częste checkpointy, tryb offline, snapshoty	Michał Iwaniuk
Niedopasowanie tokenizera	Niska	Średni	Dostrojenie słownika/retrening tokenizera	Michał Iwaniuk

Wymaganie	**Moduł**	**AT/metryka**	**Eksperyment**
WF-2 (wymienne uwagi)	Encoder+Attention	Test funkcjonalny (kształty, maski)	Jednostkowe na losowych wejściach
WNF-2 (jakość SDPA)	Trening+Eval	macro-F1 vs TF-IDF+LR (+5 pp)	AG News/IMDb, 3 seedy
WNF-1 (koszt Performer, Reformer)	Attention	time/epoch, max VRAM	Porównanie do SDPA
WNF-2 (stabilność)	Trening	brak NaN/OOM, rozrzut ≤ 1 pp	3 seedy, stała konfiguracja
WF-6 (raporty)	Eval+Logger	results.csv, PASS_WNF	`eval.py -compare`

7 Harmonogram

Harmonogram wykonywany zgodnie z sekcją *Podział Pracy*.

Październik — Implementacja systemu

- (01.10–7.10): implementacja mechanizmów uwagi od podstaw (SDPA, Reformer LSH, Performer FAVOR+).

- (8.10–14.10): złożenie bloku enkodera i modelu z wymienną architekturą uwagi.
- (15.10–19.10): pipeline danych (WordPiece, dynamiczny padding), implementacja maskowania MLM.
- (20.10–26.10): skrypty pretrain/finetune/eval z integracją W&B, checkpointowaniem i wznowieniem; testy jednostkowe i funkcjonalne.

Listopad — Trening modeli i zbieranie metryk

- (27.10–9.11): pretrening modeli na wybranych korpusach (MLM), w różnych konfiguracjach.
- (10.11–23.11): fine-tuning do klasyfikacji (AG News/IMDb) z checkpointów pretreningu.

Grudzień, Styczeń — Analiza wyników i dokumentacja

- (24.11–7.12): agregacja i analiza wyników, wnioski dot. użyteczności rozwiązań.
- (8.12–21.12): przygotowanie materiałów (tabele/wykresy).
- (22.12–4.01): opracowanie pełnej dokumentacji technicznej (implementacja, metodologia, rezultaty).
- (5.01–31.01): finalizacja dokumentacji zgodnie z wymaganiami pracy inżynierskiej.

8 Podział pracy

Osoba odpowiedzialna	**Zakres**
Bartłomiej Borycki	Implementacja klasy transformera, implementacja skryptów treningowych i ewaluacyjnych, implementacja mechanizmu uwagi Performer FAVOR+ (łącznie z modyfikacjami)
Michał Iwaniuk	Implementacja enkodera, implementacja mechanizmów uwagi scaled dot product i reformer LSH (łącznie z modyfikacjami), implementacja mechanizmu maskowania MLM
Wspólne	Wybranie zbiorów danych, preprocessing danych, trenowanie modeli, analiza i wizualizacja wyników, przygotowanie dokumentacji