



**WYŻSZA SZKOŁA
INFORMATYKI I ZARZĄDZANIA**
z siedzibą w Rzeszowie

Projekt

DataLab - system do przetwarzania dużego zbioru danych NYC Yellow Taxi

Prowadzący:

Dr inż. Leszek Puzio

Student:

Kacper Kulig, w69199

Przedmiot:

Szkolenie Techniczne 4

Kierunek:

informatyka – 6 IIZ

Rzeszów 2025

Spis treści

Wstęp	3
Opis założeń projektu	4
Wymagania funkcjonalne i нефункционалне projektu	5
Opis struktury projektu	6
Instrukcja uruchomienia aplikacji	8
Analiza kodu	9
Testy	24
Prezentacja warstwy użytkowej	29
Podsumowanie	35

Wstęp

DataLab jest projektem służącym do analizy dużych zbiorów danych pochodzących z przejazdów taksówek w Nowym Jorku (NYC Yellow Taxi). Dane wejściowe, udostępnione w formacie Parquet, obejmują niemal 3 miliony rekordów dotyczących kursów taxi. Skalę problemu podkreśla fakt, że ręczne lub sekwencyjne przetwarzanie tak obszernego zbioru informacji jest czasochłonne i podatne na błędy. W związku z tym powstała potrzeba opracowania rozwiązania umożliwiającego szybkie, równoległe przetwarzanie danych oraz interaktywną prezentację wyników analiz. Projekt DataLab odpowiada na to zapotrzebowanie poprzez połączenie wydajnego zaplecza obliczeniowego (wykorzystującego wieloprocusowość) z intuicyjnym interfejsem webowym. W ramach projektu zrealizowano aplikację webową (opartą na bibliotece Streamlit), która pozwala użytkownikowi na uruchomienie automatycznej analizy danych, podgląd wybranych fragmentów surowego zbioru oraz wizualizację wyników, wszystko to w przeglądarce internetowej. Dokumentacja poniżej przedstawia założenia przyjęte w projekcie, opisuje główny problem i sposób jego rozwiązania, a także wyszczególnia wymagania funkcjonalne, нефункционалне oraz użyte technologie.

Opis założeń projektu

Cel projektu

Celem projektu DataLab jest stworzenie systemu umożliwiającego efektywną analizę ogromnego zbioru danych dotyczących przejazdów taksówek miejskich. Aplikacja ma automatyzować proces ładowania i przetwarzania danych (w tym czyszczenie i walidację), obliczać kluczowe statystyki opisowe oraz identyfikować potencjalne nieprawidłowości w danych. Istotnym elementem celu jest również udostępnienie interaktywnego interfejsu webowego, dzięki któremu zarówno analitycy danych, jak i mniej zaawansowani użytkownicy, mogą łatwo interpretować wyniki poprzez czytelne raporty tekstowe i wykresy. Realizacja tak postawionego celu ma przyczynić się do skrócenia czasu potrzebnego na uzyskanie wniosków z danych oraz zwiększenia dostępności analizy dużych zbiorów danych dla szerszego grona odbiorców.

Główny problem wymagający rozwiązania

Głównym problemem, na który odpowiada DataLab, jest wydajne przetwarzanie i analiza masowych danych transakcyjnych z ograniczonymi zasobami sprzętowymi. Tradycyjne metody analizy (np. w arkuszach kalkulacyjnych lub skryptach działających jednowątkowo) zawodzą w obliczu zbioru liczącego kilka milionów wierszy, przetwarzanie takiej ilości informacji może trwać zbyt długo oraz przekraczać dostępne zasoby pamięci. Ponadto, przy ręcznym podejściu trudno jest wykryć ukryte wzorce lub anomalie w tak dużym wolumenie danych. Dodatkowym wyzwaniem jest udostępnienie wyników analizy w przystępnej formie. Samo przeprowadzenie obliczeń to nie wszystko, rezultaty muszą zostać zaprezentowane w sposób umożliwiający ich interpretację przez użytkownika końcowego. Problemem do rozwiązania stała się więc nie tylko szybka obróbka danych, ale i stworzenie interfejsu prezentującego istotne informacje (wykresy, wskaźniki, raporty) w intuicyjny i interaktywny sposób.

Istota przedstawionego problemu

Istotą problemu jest zapewnienie skalowalności i interaktywności analizy danych przy jednoczesnym zachowaniu poprawności wyników. Duży zbiór danych taksówkowych zawiera informacje o kursach (m.in. liczba pasażerów, dystans trasy, opłata, napiwek, itp.), które mogą posłużyć do wyciągnięcia cennych wniosków biznesowych i operacyjnych. Jednak bez odpowiednich narzędzi analitycznych potencjał tych danych pozostaje niewykorzystany. Występują również typowe dla rzeczywistych danych problemy jakościowe – na przykład błędne lub odstające wartości, takie jak ujemne kwoty opłat czy ekstremalnie wysokie dystanse przejazdów (w zbiorze zdarzały się rzędy o dystansie rzędu 312 722 mil, co jasno wskazuje na błąd w danych). Brak systemu wykrywającego i obsługującego takie przypadki może prowadzić do zaburzenia wyników analizy.

Sposób rozwiązania problemu i rezultaty

Rozwiązaniem zastosowanym w projekcie DataLab jest wieloetapowy pipeline przetwarzania danych z wykorzystaniem równoległości. Aplikacja łączy dane z pliku .parquet w porcjach (chunkach), co pozwala ograniczyć zużycie pamięci, jednorazowo przetwarzany jest tylko fragment zbioru. Każdy chunk trafia następnie do oddzielnego procesu działającego równolegle (z użyciem

multiprocessing.Pool), co umożliwia pełne wykorzystanie dostępnych rdzeni CPU i skrócenie czasu analizy.

W każdym procesie wykonywana jest walidacja i czyszczenie danych, usuwane są rekordy niepoprawne (np. z ujemnymi opłatami, pustymi polami, nielogicznymi datami). Następnie obliczane są statystyki cząstkowe i wykrywane anomalie, np. przypadki, w których napiwek przekracza sumę całkowitą kursu. Wyniki z poszczególnych procesów są łączone, co daje pełne podsumowanie danych – m.in. łączna liczba kursów, średnie dystanse, opłaty i napiwki. Dla analizowanego zbioru (styczeń 2024) zarejestrowano 2 964 624 przejazdy, ze średnim dystansem 3.65 mili i napiwkiem ~\$3.34. Wykryto także ponad 35 000 anomalii.

Dodatkowo tworzony jest raport zbiorczy z podziałem na dostawców usług oraz szczegółowe raporty tekstowe (statystyki, anomalie). Równolegle generowane są wykresy: histogramy, wykresy rozrzutu, mapy cieplne, ilustrujące kluczowe zależności (np. liczba pasażerów a długość trasy, napiwek a kwota końcowa). Wszystkie grafy i raporty zapisywane są automatycznie w katalogu wyjściowym.

Wyniki analizy prezentowane są w interfejsie webowym zbudowanym w Streamlit. Użytkownik może uruchomić pipeline jednym kliknięciem, a następnie przeglądać wyniki w zakładkach: wykresy, raporty tekstowe, logi systemowe oraz podgląd kilku pierwszych rekordów zbioru. Interfejs jest prosty w obsłudze.

Wymagania funkcjonalne i niefunkcjonalne projektu

Wymagania funkcjonalne:

- **Ładowanie danych z pliku Parquet:** System musi wczytać pełny zbiór danych wejściowych (.parquet) zawierający miliony rekordów, dzieląc go na porcje do przetwarzania.
- **Czyszczenie i walidacja danych:** Aplikacja powinna automatycznie filtrować lub oznaczać nieprawidłowe dane (np. ujemne wartości opłat, błędne liczby pasażerów) w każdym przetwarzanym fragmencie.
- **Równoległe przetwarzanie wsadowe:** System wykorzystuje wieloprocusowość do równoczesnej analizy wielu fragmentów danych, co znacząco skraca czas obliczeń w porównaniu z wykonaniem sekwencyjnym.
- **Obliczanie statystyk zbiorczych:** Dla pełnego zbioru danych generowane są kluczowe statystyki (liczba kursów, średnie i sumy finansowe, średni dystans, liczba pasażerów itp.) dostępne dla użytkownika w formie raportu podsumowującego.
- **Wykrywanie anomalii:** Aplikacja automatycznie identyfikuje nietypowe przypadki w danych, takie jak opisywane wcześniej relacje (np. napiwek przewyższający sumę opłat) lub inne odstające wartości, i prezentuje ich zestawienie.
- **Generowanie wizualizacji:** Na podstawie wyników analizy tworzony jest zestaw wykresów (histogramy, wykresy punktowe, mapy cieplne), które użytkownik może oglądać w interfejsie webowym. Wykresy mają za zadanie zilustrować rozkłady cech i korelacje między nimi.
- **Interfejs webowy (Streamlit):** Wyniki pracy systemu udostępniane są przez przeglądarkę – interfejs zawiera zakładki lub sekcje pozwalające na: uruchomienie analizy, podgląd

przykładowych surowych danych, przeglądanie wygenerowanych wykresów oraz raportów tekstowych, a także wgląd w logi działania aplikacji.

- **Logowanie operacji:** System prowadzi dziennik zdarzeń (log), w którym rejestrowane są informacje o przebiegu analizy (np. czasy wykonania etapów, liczba odfiltrowanych rekordów). Użytkownik ma możliwość podejrzenia logu, co ułatwia diagnozowanie ewentualnych problemów.

Wymagania niefunkcjonalne:

- **Wydajność:** System powinien umożliwiać analizę dużych zbiorów. W tym celu wykorzystano przetwarzanie równoległe i chunkowe.
- **Skalowalność:** Aplikacja musi poprawnie obsługiwać większe zbiory danych po dostosowaniu parametrów (chunk, RAM). Architektura umożliwia łatwą rozbudowę o nowe metryki i reguły.
- **Niezawodność:** Niepoprawne dane są pomijane lub raportowane. System kontynuuje analizę bez przerywania pracy. Poprawność obliczeń weryfikowana jest przez testy jednostkowe.
- **Użyteczność:** Interfejs Streamlit jest prosty, z czytelnym podziałem na sekcje (wykresy, raporty, dane).
- **Przenośność:** Projekt działa w Pythonie i przeglądarce, bez zależności od konkretnego systemu operacyjnego. Może być uruchomiony na Windows, Linux i macOS.
- **Bezpieczeństwo:** Dane przetwarzane są lokalnie, a logi nie zawierają informacji wrażliwych.

Opis struktury projektu

Wykorzystane technologie i narzędzia:

- Język programowania: Python 3.11
- Środowisko programistyczne: PyCharm
- Typ aplikacji: Aplikacja webowa (lokalna)
- Interfejs użytkownika: Streamlit
- Obsługa plików danych: PyArrow (.parquet)
- Wzorec architektury: Modularny podział na warstwy funkcjonalne (Loader, Cleaner, Analizer, Visualizer)
- Repozytorium: GitHub | <https://github.com/Kacper20001/DataLab>

Wykorzystane biblioteki:

- andas – obsługa danych tabelarycznych
- pyarrow – szybki odczyt plików .parquet
- matplotlib – tworzenie wykresów
- seaborn – estetyczne wizualizacje
- streamlit – UI aplikacji webowej
- memory_profiler – analiza zużycia pamięci
- pytest – testy jednostkowe
- multiprocessing – przetwarzanie równoległe
- logging – logowanie działania aplikacji

Minimalne wymagania sprzętowe:

- System operacyjny: Windows 10 / Linux / macOS
- Procesor: 4-rdzeniowy (zalecane)
- Pamięć RAM: min. 4 GB (zalecane 8 GB)
- Miejsce na dysku twardym: ok. 200 MB + pliki danych
- Python: Wersja 3.11 lub wyższa
- Przeglądarka: Dowolna współczesna (do obsługi Streamlit przez localhost)

Struktura działania:

- Dane wczytywane chunkami (partiami)
- Każdy chunk przetwarzany w osobnym procesie (Pool)
- Etapy: walidacja → statystyki → agregacja → raporty i wykresy
- Wyniki prezentowane w aplikacji Streamlit (zakładki: wykresy, raporty, dane, logi)

Struktura projektu:

- core/ – główna logika aplikacji:
 - loader.py – wczytywanie danych z pliku Parquet w chunkach,
 - cleaner.py – czyszczenie i weryfikacja danych,
 - analyzer.py – analiza statystyczna,
 - pool_processor.py – równoległe przetwarzanie danych (multiprocessing),
 - visualizer.py – generowanie wykresów,
 - logger.py – logowanie zdarzeń,
 - profiling/ – moduł do profilowania CPU i RAM (profiler.py).
- decorators/ – dekoratory:
 - timer.py – pomiar czasu,
 - counter.py – zliczanie wywołań.
- validation/ – walidacja danych:
 - base.py – klasa bazowa walidatora,
 - validators.py – konkretne reguły walidacji,
 - validation_runner.py – silnik uruchamiający walidację.
- pipeline/ – obsługa przepływu danych:
 - taxi_pipeline.py – pełny pipeline analityczny,
 - meta.py – logika łączenia analiz cząstkowych.
- tests/ – testy jednostkowe dla modułów (pytest).
- streamlit_app.py – interfejs użytkownika (web UI).
- main.py – punkt wejścia aplikacji: uruchamia pipeline oraz UI Streamlit.
- data/ – dane wejściowe i wyjściowe:
 - raw/ – plik źródłowy .parquet,
 - output/ – wykresy i raporty .txt,
 - profiling/ – pliki profilowania .prof i .memlog,
 - logs/ – logi aplikacji.

Instrukcja uruchomienia aplikacji

W celu uruchomienia aplikacji DataLab należy wykonać następujące kroki:

1. Pobranie projektu

- Sklonuj repozytorium z GitHub:
- `git clone https://github.com/Kacper20001/DataLab.git`
- `cd DataLab`

2. Przygotowanie środowiska

Utwórz i aktywuj środowisko wirtualne:

- Windows:
 - `python -m venv venv`
 - `.\venv\Scripts\activate`
- Linux/macOS:
 - `python3 -m venv venv`
 - `source venv/bin/activate`

3. Dodanie pliku danych

- Umieść plik `yellow_tripdata_2024-01.parquet` w katalogu:
- `data/raw/`

4. Uruchomienie aplikacji

W terminalu wpisz:

- `python main.py`
- Aplikacja wykona analizę danych i uruchomi interfejs Streamlit dostępny lokalnie pod adresem: `http://localhost:8501`

5. Korzystanie z aplikacji

Interfejs użytkownika umożliwia:

- uruchomienie analizy,
- przegląd wykresów i raportów,
- podgląd danych źródłowych oraz logów.

Aplikacja działa całkowicie lokalnie – nie wymaga połączenia z Internetem poza pierwszym pobraniem danych.

Analiza kodu

DataLab/core/profiling/profiler.py

Moduł odpowiedzialny za profilowanie wydajności aplikacji. Zawiera dwie główne funkcje: `profile_cpu()` oraz `profile_memory()`, służące odpowiednio do pomiaru zużycia procesora i pamięci operacyjnej przez wskazane funkcje. Dane z profilowania są automatycznie zapisywane w katalogu `data/profiling` w postaci plików `.prof` oraz `.memlog`, co umożliwia późniejszą analizę wydajności poszczególnych etapów przetwarzania danych. W przypadku błędów, system loguje wyjątki, nie przerywając działania aplikacji. Moduł wykorzystuje biblioteki `cProfile`, `memory_profiler` oraz `logging`.

```
"""
profiler.py

Moduł odpowiedzialny za profilowanie wydajności kodu.

Zawiera dwie funkcje:
- profile_cpu: tworzy profil CPU działania przekazanej funkcji (z użyciem cProfile)
- profile_memory: mierzy użycie pamięci funkcji (z użyciem memory_profiler)

Wyniki profilowania są zapisywane do katalogu 'data/profiling'.
"""

import os
import cProfile
import logging
from memory_profiler import memory_usage

logger = logging.getLogger(__name__)

def profile_cpu(func, filename="cpu_profile.prof"):
    """
    Profiluje zużycie CPU przez przekazaną funkcję i zapisuje wynik do pliku .prof.

    Args:
        func (Callable): Funkcja do profilowania.
        filename (str): Nazwa pliku wynikowego (.prof) zapisywanego w 'data/profiling'.

    Raises:
        Exception: Błąd podczas profilowania – logowany i przekazywany dalej.
    """
    os.makedirs("data/profiling", exist_ok=True)
    output_path = os.path.join("data", "profiling", filename)

    logger.info("Rozpaczynam profilowanie CPU.")
    profiler = cProfile.Profile()

    try:
        profiler.enable()
        func()
        profiler.disable()
        profiler.dump_stats(output_path)
        logger.info(f"Profil CPU zapisany do: {output_path}")
    except Exception as e:
        logger.exception("Błąd podczas profilowania CPU:")
        raise e

def profile_memory(func, filename="memory_profile.memlog"):
    """
    Profiluje zużycie pamięci przez przekazaną funkcję i zapisuje dane do pliku tekstowego.

    Args:
        func (Callable): Funkcja do profilowania.
        filename (str): Nazwa pliku wynikowego (.memlog) zapisywanego w 'data/profiling'.

    Raises:
        Exception: Błąd podczas profilowania – logowany i przekazywany dalej.
    """
    os.makedirs("data/profiling", exist_ok=True)
    output_path = os.path.join("data", "profiling", filename)

    logger.info("Rozpaczynam profilowanie pamięci.")
    try:
        mem_usage = memory_usage(func, interval=0.1, timeout=None)
        with open(output_path, "w") as f:
            for usage in mem_usage:
                f.write(f"{usage}\n")
        logger.info(f"Profil pamięci zapisany do: {output_path}")
    except Exception as e:
        logger.exception("Błąd podczas profilowania pamięci:")
        raise e
```

Rysunek 1. Kod DataLab/core/profiling/profiler.py

DataLab/core/cleaner.py

Plik cleaner.py odpowiada za czyszczenie i walidację danych w podziale na fragmenty (chunki). Kluczowa funkcja `clean_data` filtruje dane przy użyciu zewnętrznego walidatora, loguje liczbę rekordów przed i po przetworzeniu oraz mierzy zużycie pamięci RAM. W razie błędu funkcja zwraca pusty DataFrame, by nie przerwać całego procesu analizy. Obsługuje wyjątki i loguje zdarzenia, wspierając niezawodne przetwarzanie dużych zbiorów danych.

```
"""
cleaner.py

Moduł odpowiedzialny za czyszczenie i walidację danych w formie chunków.

Zawiera funkcję clean_data, która:
- filtruje dane za pomocą walidatora
- loguje liczbę rekordów przed i po przetworzeniu
- mierzy zużycie pamięci RAM

W przypadku błędu zwraca pusty DataFrame, aby nie przerywać całego procesu.
"""

import pandas as pd
import psutil
from decorators.timer import measure_time
from decorators.counter import count_calls
from validation.validation_runner import validate_chunk
from core.logger import logger

NEEDED_COLUMNS = [
    "passenger_count", "trip_distance", "tip_amount", "total_amount",
    "fare_amount", "tpep_pickup_datetime", "tpep_dropoff_datetime"
]

@measure_time
@count_calls
def clean_data(df: pd.DataFrame) -> pd.DataFrame:
    """
    Czyści i waliduje pojedynczy chunk danych.

    Args:
        df (pd.DataFrame): Chunk danych do przetworzenia.

    Returns:
        pd.DataFrame: Przefiltrowany i zweryfikowany chunk. W razie błędu – pusty DataFrame.
    """
    try:
        initial_len = len(df)
        cleaned_df = validate_chunk(df)
        cleaned_len = len(cleaned_df)
        mem = psutil.Process().memory_info().rss / 1024 ** 2
        logger.info(
            f"[Cleaner] Chunk: {initial_len} -> {cleaned_len} rekordów po walidacji | RAM: {mem:.2f} MB"
        )
        return cleaned_df
    except Exception as e:
        logger.error(f"[Cleaner] Błąd podczas czyszczenia chunku: {e}")
        return pd.DataFrame() # Nie przerywamy całego procesu przy multiprocessing
```

Rysunek 2. DataLab/core/cleaner.py

DataLab/core/analyzer.py

Plik analyzer.py odpowiada za analizę danych z pliku .parquet. Zawiera funkcje do przetwarzania danych w trybie równoległym (`parallel_analysis`) oraz sekwencyjnym (`streaming_global_analysis`). Funkcja `analyze_chunk` analizuje pojedyncze fragmenty danych i wylicza metryki (suma opłat, napiwki, długość trasy itd.), natomiast `_save_summary` zapisuje wyniki do pliku. Moduł korzysta z dekoratorów do pomiaru czasu i zliczania wywołań, obsługuje wyjątki i loguje zdarzenia. Całość wspiera przetwarzanie dużych zbiorów danych z zachowaniem kontroli nad jakością i wydajnością przetwarzania.

```

'''
analyzer.py

Moduł odpowiedzialny za analizę danych z pliku .parquet w dwóch trybach:
- parallel_analysis: analiza z użyciem multiprocessing.Pool
- streaming_global_analysis: analiza sekwencyjna chunków + walidacja

Funkcje obliczają summaryczne metryki (dystans, napiwki, pasażerowie itp.)
i zapisują podsumowanie do pliku tekstowego. Obsługuje błędy, loguje zdarzenia
i wykorzystuje dekoratory pomiaru czasu i zliczania wywołań.
'''

import os
import logging
import pandas as pd
from multiprocessing import Pool, cpu_count
from decorators.timer import measure_time
from decorators.counter import count_calls
from core.loader import load_parquet_in_chunks
from validation.validation_runner import run_all_validations

logger = logging.getLogger(__name__)

def analyze_chunk(df: pd.DataFrame) -> dict:
    '''
    Analizuje pojedynczy fragment danych (chunk) i zwraca metryki.

    Args:
        df (pd.DataFrame): Fragment danych do analizy.

    Returns:
        dict: Słownik z wynikami (liczba wierszy, sumy wartości, liczba długich kursów).
    '''
    try:
        return {
            "rows": len(df),
            "distance": df["trip_distance"].sum(),
            "tip": df["tip_amount"].sum(),
            "amount": df["total_amount"].sum(),
            "passengers": df["passenger_count"].sum(),
            "long_trips": (df["trip_distance"] > 10).sum()
        }
    except Exception as e:
        logger.exception("Błąd podczas analizy chunku: %s", e)
        return {
            "rows": 0, "distance": 0.0, "tip": 0.0, "amount": 0.0,
            "passengers": 0, "long_trips": 0
        }

@measure_time
@count_calls
def parallel_analysis(path: str, chunksize: int = 100_000) -> dict:
    '''
    Wykonuje równoległą analizę danych z pliku .parquet z użyciem multiprocessing.Pool.

    Args:
        path (str): Ścieżka do pliku .parquet.
        chunksize (int): Liczba wierszy na chunk.

    Returns:
        dict: Podsumowanie analizowanych danych (zapisane też do pliku).
    '''
    os.makedirs("data/output", exist_ok=True)
    total = {
        "rows": 0, "distance": 0.0, "tip": 0.0,
        "amount": 0.0, "passengers": 0, "long_trips": 0
    }

    try:
        with Pool(cpu_count()) as pool:
            results = pool.map(analyze_chunk, load_parquet_in_chunks(path, chunksize))

            for result in results:
                for key in total:
                    total[key] += result[key]

            average_fare = total["amount"] / total["rows"] if total["rows"] > 0 else 0

            summary = {
                "Liczba rekordów": total["rows"],
                "Średnia długość trasy (mile)": round(total["distance"] / total["rows"], 2),
                "Średni napiwek ($)": round(total["tip"] / total["rows"], 2),
                "Łączna kwota opłat ($)": round(total["amount"], 2),
                "Średnia opłata za kurs ($)": round(average_fare, 2),
                "Liczba pasażerów (łącznie)": int(total["passengers"]),
                "Średnia liczba pasażerów na kurs": round(total["passengers"] / total["rows"], 2),
                "Liczba długich kursów (>10 mil)": total["long_trips"]
            }

            save_summary(summary, "data/output/parallel_summary.txt")
            logger.info("Analiza równoległa zakończona. Wynik zapisany.")
            return summary
    except Exception as e:
        logger.exception("Błąd podczas analizy równoległej: %s", e)
        return {}

@measure_time
@count_calls
def streaming_global_analysis(path: str, chunksize: int = 100_000) -> dict:
    '''
    Wykonuje analizę danych chunk po chunku z walidacją, bez multiprocessing.

    Args:
        path (str): Ścieżka do pliku .parquet.
        chunksize (int): Liczba wierszy na chunk.

    Returns:
        dict: Podsumowanie analizowanych danych (zapisane też do pliku).
    '''
    os.makedirs("data/output", exist_ok=True)
    total = {
        "rows": 0, "distance": 0.0, "tip": 0.0,
        "amount": 0.0, "passengers": 0, "long_trips": 0
    }

    try:
        for chunk in load_parquet_in_chunks(path, chunksize):
            chunk = run_all_validations(chunk)

            result = analyze_chunk(chunk)
            for key in total:
                total[key] += result[key]

            average_fare = total["amount"] / total["rows"] if total["rows"] > 0 else 0

            summary = {
                "Liczba rekordów": total["rows"],
                "Średnia długość trasy (mile)": round(total["distance"] / total["rows"], 2),
                "Średni napiwek ($)": round(total["tip"] / total["rows"], 2),
                "Łączna kwota opłat ($)": round(total["amount"], 2),
                "Średnia opłata za kurs ($)": round(average_fare, 2),
                "Liczba pasażerów (łącznie)": int(total["passengers"]),
                "Średnia liczba pasażerów na kurs": round(total["passengers"] / total["rows"], 2),
                "Liczba długich kursów (>10 mil)": total["long_trips"]
            }

            save_summary(summary, "data/output/streaming_summary.txt")
            logger.info("Analiza streamingowa zakończona. Wynik zapisany.")
            return summary
    except Exception as e:
        logger.exception("Błąd podczas analizy streamingowej: %s", e)
        return {}

def _save_summary(summary: dict, path: str) -> None:
    '''
    Zapisuje słownik wyników analizy do pliku tekstowego.

    Args:
        summary (dict): Dane do zapisania.
        path (str): Ścieżka zapisu.
    '''
    try:
        with open(path, "w", encoding="utf-8") as f:
            for k, v in summary.items():
                f.write(f"{k}: {v}\n")
        logger.debug(f"Wynik analizy zapisany do {path}")
    except Exception as e:
        logger.exception("Nie udało się zapisać pliku z wynikami: %s", e)

```

Rysunek 3. Kod Datalab/core/analyzer.py

DataLab/core/loader.py

Plik loader.py zawiera funkcję load_parquet_in_chunks, która odpowiada za etapowe wczytywanie danych z pliku .parquet w postaci chunków przy użyciu biblioteki Pandas i silnika PyArrow. Funkcja działa jako generator, zwracając kolejne fragmenty danych typu DataFrame, co umożliwia efektywne przetwarzanie dużych zbiorów bez przeciążania pamięci RAM.

```
import pandas as pd
from decorators.timer import measure_time
from decorators.counter import count_calls
from core.logger import logger

@measure_time
@count_calls
def load_parquet_in_chunks(path: str, chunksize: int = 100_000, columns: list[str] | None = None):
    """
    Generator wczytujący dane z pliku Parquet w chunkach przy użyciu pandas.

    Args:
        path (str): Ścieżka do pliku Parquet.
        chunksize (int): Liczba wierszy na chunk.
        columns (list[str] | None): Lista kolumn do załadowania (opcjonalnie).

    Yields:
        pd.DataFrame: Kolejny fragment danych jako DataFrame.
    """
    try:
        df = pd.read_parquet(path, columns=columns, engine="pyarrow")
        total_rows = len(df)
        logger.info(f"[Loader] Wczytano {total_rows} wierszy z pliku: {path}")

        if total_rows == 0:
            logger.warning(f"[Loader] Brak danych do przetworzenia w pliku {path}")
            return

        for i in range(0, total_rows, chunksize):
            chunk = df.iloc[i:i + chunksize]
            logger.info(f"[Loader] Chunk {i // chunksize + 1} załadowany ({len(chunk)} wierszy)")
            yield chunk
    except Exception as e:
        logger.error(f"[Loader] Błąd podczas wczytywania pliku {path}: {e}")
        return
```

Rysunek 4. Kod DataLab/core/loader.py

DataLab/core/logger.py

Plik logger.py zawiera funkcję setup_logger, która konfiguruje logowanie aplikacji. Umożliwia zapis komunikatów do pliku (domyślnie data/logs/app.log) oraz ich wyświetlanie w konsoli. Funkcja tworzy logger z określonym poziomem szczegółowości, dodaje odpowiednie handlersy i formatory. Może być importowana globalnie w całym projekcie.

```

"""
logger.py

Konfiguruje logger aplikacyjny zapisywany do pliku i wyświetlany na konsoli.
Domyślnie logi trafiają do 'data/logs/app.log'.
Logger tworzony funkcją setup_logger może być importowany w całym projekcie.

import logging
import os

def setup_logger(name: str, log_file: str = "data/logs/app.log", level=logging.INFO) -> logging.Logger:
    """
    Tworzy i konfiguruje logger z obsługą zapisu do pliku oraz wyjścia na konsolę.

    Args:
        name (str): Nazwa loggera (np. 'DataLab').
        log_file (str): Ścieżka do pliku logów.
        level (int): Poziom logowania (np. logging.INFO, logging.DEBUG).

    Returns:
        logging.Logger: Skonfigurowany logger.
    """
    os.makedirs(os.path.dirname(log_file), exist_ok=True)

    log = logging.getLogger(name)
    log.setLevel(level)
    log.propagate = False # Zapobiega podwójnemu logowaniu

    if not log.handlers:
        ch = logging.StreamHandler()
        ch.setLevel(level)

        fh = logging.FileHandler(log_file, encoding='utf-8')
        fh.setLevel(level)

        formatter = logging.Formatter('%(asctime)s[%(levelname)s] %(message)s', "%Y-%m-%d %H:%M:%S")
        ch.setFormatter(formatter)
        fh.setFormatter(formatter)

        log.addHandler(ch)
        log.addHandler(fh)

    return log

# Główny logger aplikacji
logger = setup_logger("DataLab")

```

Rysunek 5. DataLab/core/logger.py

DataLab/core/pool_processor.py

Plik pool_processor.py odpowiada za główną logikę równoległej analizy danych z pliku Parquet. Zawiera funkcję analyze_chunk, która oblicza statystyki z jednego fragmentu danych, oraz aggregate_results, która sumuje wyniki z wielu chunków. Dodatkowo save_summary_by_vendor tworzy zestawienie statystyk wg firm taksówkowych, a save_anomalies_report zapisuje nietypowe rekordy (np. napiwek większy niż suma opłat). Funkcja parallel_analysis koordynuje całość procesu przy użyciu multiprocessing. Wyniki zapisywane są do folderu data/output.

```

'''
pool_processor.py
Moduł wykonujący równoległą analizę danych z pliku .parquet z wykorzystaniem multiprocessing.Pool.

Zawiera funkcje:
- analyze_chunk: analizuje pojedynczy fragment danych (sumy, długie trasy itp.)
- aggregate_results: sumuje wyniki z chunków
- save_summary_by_vendor: tworzy raport per VendorID
- save_anomalies_report: wykrywa podejrzane rekordy (tip > total)
- parallel_analysis: główna funkcja analizy równoległej

Zapisuje wszystkie raporty do katalogu 'data/output'.
'''

import os
import pandas as pd
from multiprocessing import Pool, cpu_count
from decorators.timer import measure_time
from decorators.counter import count_calls
from core.loader import load_parquet_in_chunks
from core.logger import logger

REQUIRED_COLUMNS = [
    "passenger_count", "trip_distance", "tip_amount", "total_amount", "VendorID"
]

def analyze_chunk(df: pd.DataFrame) -> dict:
    """
    Analizuje chunk danych, obliczając sumaryczne metryki.

    Args:
        df (pd.DataFrame): Fragment danych.

    Returns:
        dict: Wyniki analizy (lub zera w razie błędu).

    try:
        if not all(col in df.columns for col in REQUIRED_COLUMNS):
            missing = [col for col in REQUIRED_COLUMNS if col not in df.columns]
            raise ValueError(f"Brakuje kolumn: {missing}")

        return {
            "rows": len(df),
            "distance": df["trip_distance"].sum(),
            "tip": df["tip_amount"].sum(),
            "amount": df["total_amount"].sum(),
            "passengers": df["passenger_count"].sum(),
            "long_trips": (df["trip_distance"] > 10).sum()
        }

    except Exception as e:
        logger.warning(f"Błąd w analizie_chunk: {e}")
        return {
            "rows": 0, "distance": 0.0, "tip": 0.0,
            "amount": 0.0, "passengers": 0, "long_trips": 0
        }

def aggregate_results(results: list[dict]) -> dict:
    """
    Sumuje dane ze wszystkich chunków.

    Args:
        results (list[dict]): Lista wyników z analyze_chunk.

    Returns:
        dict: Podsumowanie statystyk globalnych.

    """
    total = {
        "rows": 0, "distance": 0.0, "tip": 0.0,
        "amount": 0.0, "passengers": 0, "long_trips": 0
    }

    for result in results:
        for key in total:
            total[key] += result[key]

    rows = total["rows"]
    return {
        "Liczba rekordów": rows,
        "Średnia długość trasy (mile)": round(total["distance"] / rows, 2) if rows else 0,
        "Średni napisek ($)": round(total["tip"] / rows, 2) if rows else 0,
        "Średnia kwota opłat ($)": round(total["amount"] / rows, 2) if rows else 0,
        "Średnia opłata za kurs ($)": round(total["amount"] / rows, 2) if rows else 0,
        "Liczba pasażerów (średnio)": int(total["passengers"] / rows),
        "Średnia liczba pasażerów na kurs": round(total["passengers"] / rows, 2) if rows else 0,
        "Liczba długich kursów (>10 mil)": total["long_trips"]
    }

def save_summary_by_vendor(df: pd.DataFrame, output_path="data/output/summary_by_vendor.txt"):
    """
    Tworzy raport średnich i maksymalnych wartości dla każdego VendorID.

    Args:
        df (pd.DataFrame): Dane wejściowe.
        output_path (str): Ścieżka zapisu pliku.

    """
    if "VendorID" not in df.columns:
        logger.warning("Brak kolumny 'VendorID', pominięto raport per VendorID.")
        return

    grouped = df.groupby("VendorID").agg({
        "fare_amount": ["mean", "max"],
        "tip_amount": ["mean", "max"],
        "trip_distance": ["mean", "max"]
    })

    os.makedirs(os.path.dirname(output_path), exist_ok=True)
    with open(output_path, "w", encoding="utf-8") as f:
        f.write("Podsumowanie per VendorID:\n")
        f.write(grouped.to_string())
        f.write("\n")

def save_anomalies_report(df: pd.DataFrame, output_path="data/output/anomalies_report.txt"):
    """
    Wyszukuje i zapisuje rekordy, w których tip_amount > total_amount.

    Args:
        df (pd.DataFrame): Dane wejściowe.
        output_path (str): Ścieżka zapisu raportu.

    """
    if not all(col in df.columns for col in ["tip_amount", "total_amount"]):
        logger.warning("Brakuje kolumn tip_amount lub total_amount, pominięto raport anomalii.")
        return

    anomalies = df[df["tip_amount"] > df["total_amount"]]
    os.makedirs(os.path.dirname(output_path), exist_ok=True)

    with open(output_path, "w", encoding="utf-8") as f:
        f.write("Anomalie: tip_amount > total_amount\n")
        f.write(f"Liczba podejrzanych rekordów: {len(anomalies)}\n\n")
        if not anomalies.empty:
            f.write(anomalies[["VendorID", "fare_amount", "tip_amount", "total_amount"]].head(10).to_string())

@measure_time
@count_calls
def parallel_analysis(path: str, chunksize: int = 100_000) -> dict:
    """
    Główna funkcja analizy danych z wykorzystaniem multiprocessing.

    Args:
        path (str): Ścieżka do pliku .parquet.
        chunksize (int): Liczba wierszy na chunk.

    Returns:
        dict: Podsumowanie wyników analizy (lub pusty słownik przy błędzie).

    """
    os.makedirs("data/output", exist_ok=True)
    logger.info(f"Start analizy równoległej ({cpu_count()} CPU)...")

    try:
        chunks = list(load_parquet_in_chunks(path, chunksize))
        logger.info(f"Założono {len(chunks)} chunków.")

        with Pool(cpu_count()) as pool:
            results = pool.map(analyze_chunk, chunks)

        summary = aggregate_results(results)

        with open("data/output/parallel_summary.txt", "w", encoding="utf-8") as f:
            for k, v in summary.items():
                f.write(f"{k}: {v}\n")

        # Łączenie wszystkich chunków do raportów szczegółowych
        full_df = pd.concat(chunks, ignore_index=True)

        save_summary_by_vendor(full_df)
        save_anomalies_report(full_df)

        logger.info("Analiza zakończona sukcesem. Raporty zapisane.")
        return summary

    except Exception as e:
        logger.error(f"Błąd podczas analizy multiprocessing: {e}")
        return {}

```

Rysunek 6 Kod DataLab/core/pool_processor.py

DataLab/core/sample_loader.py

Plik sample_loader.py odpowiada za wczytanie niewielkiej próbki danych z pliku .parquet, wykorzystywanej np. do testów lub wizualizacji. Funkcja load_sample_for_visualization łączy i czyści pierwsze N chunków danych (domyślnie 2), łącząc je w jeden DataFrame gotowy do prezentacji. Wykorzystuje funkcję clean_data z modułu cleaner i generator chunków z loader.py.

```
"""
sample_loader.py

Moduł odpowiedzialny za załadowanie niewielkiej próbki danych z pliku .parquet
(np. do testów, eksploracji, wykresów). Próbka pochodzi z pierwszych kilku chunków
i jest czyszczona za pomocą clean_data.
"""

import pandas as pd
from decorators.timer import measure_time
from decorators.counter import count_calls
from core.cleaner import clean_data
from core.loader import load_parquet_in_chunks

@measure_time
@count_calls
def load_sample_for_visualization(path: str, chunksize: int = 100_000, max_chunks: int = 2) -> pd.DataFrame:
    Ładuje i oczyszcza dane z pierwszych N chunków jako próbkę do wizualizacji.

    Args:
        path (str): Ścieżka do pliku .parquet.
        chunksize (int): Liczba wierszy na chunk.
        max_chunks (int): Maksymalna liczba chunków do załadowania.

    Returns:
        pd.DataFrame: Połączona i oczyszczona próbka danych.
    """
    chunks = []
    for i, chunk in enumerate(load_parquet_in_chunks(path, chunksize=chunksize)):
        cleaned = clean_data(chunk)
        chunks.append(cleaned)
        if i + 1 >= max_chunks:
            break

    if not chunks:
        return pd.DataFrame() # fallback na pusty DataFrame

    return pd.concat(chunks, ignore_index=True)
```

Rysunek 7. Kod DataLab/core/sample_loader.py.

DataLab/core/visualizer.py

Plik visualizer.py odpowiada za generowanie wizualizacji danych NYC Taxi na podstawie przekazanego DataFrame i zapis ich w formacie PNG do folderu data/output/. Zawiera funkcję visualize_data(df), która tworzy pięć typów wykresów: histogram długości trasy (do 30 mil), wykres słupkowy średnich napiwków w zależności od liczby pasażerów (1–6), boxplot wartości napiwków (do 15\$), scatterplot korelacji napiwku z całkowitą kwotą przejazdu (do 100\$) oraz heatmapę przedstawiającą średnią długość trasy względem liczby pasażerów. Dodatkowo funkcja plot_memory_usage(memlog_path) rysuje wykres liniowy zużycia pamięci RAM na podstawie pliku .memlog wygenerowanego podczas analizy. Wszystkie funkcje wykorzystują dekoratory @measure_time i @count_calls do logowania czasu wykonania i liczby wywołań. Wizualizacje są automatycznie zapisywane jako pliki PNG, gotowe do przeglądania w interfejsie Streamlit.

```

"""
visualizer.py

Moduł odpowiedzialny za tworzenie wizualizacji danych z NYC Taxi.

Zawiera funkcje:
- visualize_data: generuje i zapisuje 5 typów wykresów analitycznych na podstawie danych wejściowych
- plot_memory_usage: rysuje wykres zużycia pamięci RAM na podstawie pliku .memlog

Wszystkie wykresy zapisywane są do folderu 'data/output'.
"""

import matplotlib.pyplot as plt
import seaborn as sns
import os

from decorators.counter import count_calls
from decorators.timer import measure_time

@count_calls
@measure_time
def visualize_data(df):
    """
    Tworzy zestaw wykresów wizualizujących dane taxi NYC.

    Generowane są:
    1. Histogram długości trasy (do 30 mil)
    2. Średni napiwek vs liczba pasażerów (1-6)
    3. Boxplot napiwków (0-15 $)
    4. Scatter plot: całkowita kwota vs napiwek (0-100 $)
    5. Heatmapa: średnia długość trasy vs liczba pasażerów (1-6)

    Wszystkie wykresy zapisywane są jako PNG do folderu 'data/output'.

    Args:
    ... df (pandas.DataFrame): Oczyszczony DataFrame z danymi NYC Taxi.
    """
    os.makedirs("data/output", exist_ok=True)

    # Histogram długości trasy (do 30 mil)
    filtered_df = df[df["trip_distance"] <= 30]
    plt.figure(figsize=(10, 6))
    filtered_df["trip_distance"].hist(bins=50, edgecolor="black")
    plt.title("Rozkład długości trasy (mile) – tylko do 30 mil")
    plt.xlabel("Długość trasy (mile)")
    plt.ylabel("Liczba kursów")
    plt.tight_layout()
    plt.savefig("data/output/trip_distance_hist_filtered.png")
    plt.close()

    # Średni napiwek vs liczba pasażerów (1-6)
    df_tip = df[df["passenger_count"].between(1, 6)]
    tip_by_passengers = df_tip.groupby("passenger_count")["tip_amount"].mean()
    plt.figure(figsize=(10, 6))
    tip_by_passengers.plot(kind="bar", color="skyblue", edgecolor="black")
    plt.title("Średni napiwek ($) względem liczby pasażerów (1-6)")
    plt.xlabel("Liczba pasażerów")
    plt.ylabel("Średni napiwek ($)")
    plt.tight_layout()
    plt.savefig("data/output/tip_by_passenger_count_filtered.png")
    plt.close()

    # Boxplot napiwków (do 15$)
    df_tip30 = df[(df["tip_amount"] >= 0) & (df["tip_amount"] <= 15)]
    plt.figure(figsize=(10, 6))
    sns.boxplot(
        data=df_tip30[["tip_amount"]],
        orient="h",
        color="white",
        linewidth=1.5,
        filtersize=1,
        boxprops=dict(facecolor='lightblue')
    )
    plt.title("Rozrzut wartości napiwków (do 15$)")
    plt.xlabel("Wartość napiwku ($)")
    plt.tight_layout()
    plt.savefig("data/output/tip_amount_boxplot.png")
    plt.close()

    # Scatter: całkowita kwota vs napiwek (sensowne zakresy)
    df_scatter = df[
        (df["total_amount"] > 0) & (df["total_amount"] <= 100) &
        (df["tip_amount"] >= 0) & (df["tip_amount"] <= 30)
    ]
    plt.figure(figsize=(10, 6))
    plt.scatter(df_scatter["total_amount"], df_scatter["tip_amount"], alpha=0.05, s=5, color="green")
    plt.title("Korelacja: Całkowita kwota vs Napiwek")
    plt.xlabel("Całkowita kwota ($)")
    plt.ylabel("Napiwek ($)")
    plt.tight_layout()
    plt.savefig("data/output/tip_vs_total_scatter.png")
    plt.close()

    # Heatmapa: średnia długość trasy vs liczba pasażerów (1-6)
    df_heat = df[df["passenger_count"].between(1, 6)]
    pivot = df_heat.pivot_table(index="passenger_count", values="trip_distance", aggfunc="mean")
    plt.figure(figsize=(8, 5))
    sns.heatmap(pivot, annot=True, cmap="YlGnBu", fmt=".1f")
    plt.title("Średnia długość trasy vs liczba pasażerów")
    plt.xlabel("Liczba pasażerów")
    plt.tight_layout()
    plt.savefig("data/output/passenger_vs_distance_heatmap.png")
    plt.close()

    print("[Visualizer] Wykresy zapisane do folderu: data/output/")

@count_calls
@measure_time
def plot_memory_usage(memlog_path: str):
    """
    Tworzy wykres liniowy przedstawiający zużycie pamięci RAM na podstawie pliku .memlog.

    Plik .memlog powinien zawierać liczby (MB) – jedna wartość w każdej linii.

    Args:
    ... memlog_path (str): Ścieżka do pliku .memlog.
    """
    with open(memlog_path) as f:
        data = [float(line.strip()) for line in f]

    plt.figure(figsize=(10, 6))
    plt.plot(data, color="red", linewidth=1)
    plt.title("Zużycie pamięci RAM podczas analizy strumieniowej")
    plt.xlabel("Czas (x0.1s)")
    plt.ylabel("Pamięć (MB)")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig("data/output/memory_usage_plot.png")
    plt.show()

```

Rysunek 8. Kod DataLab/core/visualizer.py

DataLab/decorators/counter.py

Plik counter.py zawiera prosty, lecz przydatny dekorator @count_calls, który zlicza liczbę wywołań oznaczonej nim funkcji i wypisuje tę informację do konsoli. Dzięki niemu możliwe jest śledzenie, jak często dana funkcja była uruchamiana podczas działania programu, co bywa szczególnie pomocne podczas debugowania lub profilowania kodu. Dekorator dodaje do funkcji atrybut calls, który inkrementuje się przy każdym wywołaniu, a następnie loguje komunikat z nazwą funkcji i jej licznikiem.

```
"""
counter.py
Dekorator 'count_calls' zlicza i wypisuje do konsoli liczbę wywołań danej funkcji.
Przydatny do monitorowania działania funkcji w czasie rzeczywistego działania programu.
"""

from functools import wraps

def count_calls(func):
    """
    Dekorator zliczający liczbę wywołań funkcji.
    Przy każdym wywołaniu wypisuje liczbę dotychczasowych uruchomień.

    Args:
        func (Callable): Funkcja, którą dekorujemy.

    Returns:
        Callable: Owiniecia funkcja z licznikiem wywołań.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        print(f"[Counter] Funkcja '{func.__name__}' wywołana {wrapper.calls} raz(y)")
        return func(*args, **kwargs)

    wrapper.calls = 0
    return wrapper
```

Rysunek 9. Kod DataLab/decorators/counter.py.

DataLab/decorators/timer.py

Plik timer.py zawiera dekorator @measure_time, który mierzy czas wykonania funkcji i wypisuje wynik na konsolę w milisekundach. Jest przydatny do profilowania wydajności kodu i identyfikacji wolnych fragmentów. Mierzy czas przed i po wykonaniu funkcji, a następnie loguje wynik.

```
"""
timer.py
Dekorator 'measure_time' służy do mierzenia czasu wykonania funkcji.
Czas wykonywania jest wypisywany na konsolę w milisekundach (ms).
"""

import time
from functools import wraps

def measure_time(func):
    """
    Dekorator mierzący czas wykonania funkcji i wypisujący go w ms.

    Args:
        func (Callable): Funkcja, którą chcemy profilować czasowo.

    Returns:
        Callable: Funkcja opakowana pomiarem czasu.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        duration = (time.time() - start) * 1000
        print(f"[Timer] Funkcja '{func.__name__}' wykonała się w {duration:.2f} ms")
        return result

    return wrapper
```

Rysunek 10. Kod DataLab/decorators/timer.py

DataLab/pipeline/base.py

Plik base.py definiuje abstrakcyjną klasę BasePipeline, będącą podstawą dla wszystkich pipeline'ów przetwarzania danych. Dzięki zastosowaniu metaklasz PipelineMeta, kolejne kroki pipeline'u (metody oznaczone w klasach dziedziczących) są automatycznie rejestrowane i wykonywane w ustalonej kolejności poprzez metodę run(). Rozwiązanie to upraszcza zarządzanie złożonymi procesami analitycznymi.

```
"""
base.py
Moduł definiuje klasę bazową dla pipeline'ów przetwarzania danych.
Klasa BasePipeline wykorzystuje metaklasę PipelineMeta do automatycznej rejestracji metod
pipeline'u (tzw. kroków przetwarzania), które następnie są wykonywane w ustalonej kolejności
przy użyciu metody run().
"""
from pipeline.meta import PipelineMeta

class BasePipeline(metaclass=PipelineMeta):
    """
    Klasa bazowa dla każdego pipeline'u przetwarzającego dane.
    Wykorzystuje metaklasę PipelineMeta do dynamicznej rejestracji kroków.
    Zapewnia metodę run(), która wykonuje wszystkie zarejestrowane kroki
    pipeline'u w kolejności ich definicji w klasie dziedziczącej.
    """
    def run(self) -> None:
        """
        Wykonuje wszystkie zarejestrowane kroki pipeline'u.
        Każdy krok to metoda oznaczona przez metaklasę PipelineMeta.
        """
        for step_name in self.steps:
            step = getattr(self, step_name)
            print(f"\n[Pipeline] Wykonuję krok: {step_name}")
            step()
```

Rysunek 11. Kod DataLab/pipeline/base.py

DataLab/pipeline/meta.py

Plik 'meta.py' zawiera definicję metaklasz 'PipelineMeta', która odpowiada za automatyczne wykrywanie i rejestrowanie metod reprezentujących kroki przetwarzania danych w pipeline'ach. Metody oznaczone atrybutem '_is_step = True' są automatycznie dodawane do listy '_steps', co umożliwia ich sekwencyjne wykonanie w klasie bazowej 'BasePipeline'. Dzięki temu rozwiązaniu kod pipeline'u staje się przejrzysty i łatwy w rozbudowie.

```
"""
meta.py
Definicja metaklasz PipelineMeta, służącej do automatycznego rejestrowania kroków
pipeline'u na podstawie metod oznaczonych atrybutem '_is_step = True'.
Metaklasa tworzy listę '_steps', która zawiera nazwę każdej oznaczonej metody,
a następnie przekazuje ją do klasy bazowej pipeline'u.
"""
class PipelineMeta(type):
    """
    Metaklasa do automatycznego rejestrowania kroków pipeline'u.
    Każda metoda w klasie potomnej oznaczona atrybutem '_is_step = True'
    zostanie automatycznie dodana do listy '_steps', która definiuje
    kolejność wykonywania kroków w pipeline.
    """
    def __new__(cls, name: str, bases: tuple, dct: dict):
        steps = [key for key, value in dct.items() if hasattr(value, "_is_step")]
        dct["_steps"] = steps
        return super().__new__(cls, name, bases, dct)
```

Rysunek 12. Kod DataLab/pipeline/meta.py

DataLab/pipeline/taxi_pipeline.py

Plik taxi_pipeline.py definiuje klasę TaxiPipeline, będącą kompletną implementacją przetwarzania danych NYC Yellow Taxi z wykorzystaniem multiprocessing. Klasa dziedziczy po BasePipeline i korzysta z metaklasy PipelineMeta, która automatycznie wykrywa kroki przetwarzania oznaczone dekoratorem @step. Pipeline obejmuje załadowanie próbki danych i jej podgląd, profilowanie CPU i pamięci, generowanie wykresów oraz tworzenie raportów tekstowych. Każdy etap oznaczony metodą @step wykonywany jest automatycznie po wywołaniu run().

```
'''
taxi_pipeline.py

Definicja klasy TaxiPipeline – konkretnej implementacji pipeline'u przetwarzającego
dane NYC Yellow Taxi przy użyciu multiprocessing. Pipeline obejmuje:
- podgląd danych wejściowych,
- analizę z profilowaniem CPU/pamięci,
- generowanie wykresów,
- tworzenie raportów.

Pipeline korzysta z klasy bazowej BasePipeline i metaklasy PipelineMeta.
'''

import logging
import os
import pandas as pd

from core.visualizer import visualize_data, plot_memory_usage
from core.sample_loader import load_sample_for_visualization
from core.profiling_profiler import profile_memory, profile_cpu
from core.pool_processor import parallel_analysis, save_summary_by_vendor, save_anomalies_report
from core.loader import load_parquet_in_chunks
from decorators.counter import count_calls
from decorators.timer import measure_time
from pipeline.base import BasePipeline

def step(func):
    '''
    Dekorator oznaczający metodę jako krok pipeline'u.
    Dodaje atrybut _is_step, dzięki czemu metaklasa PipelineMeta automatycznie
    rozpoznaje metodę jako krok do wykonania.
    '''
    func._is_step = True
    return func

class TaxiPipeline(BasePipeline):
    '''
    Pipeline do przetwarzania danych NYC Yellow Taxi z użyciem multiprocessing.
    Składa się z kroków: podgląd danych, analiza równoległa, wizualizacja, raporty.

    def __init__(self, file_path: str):
    '''
    Inicjalizuje pipeline z podaną ścieżką do pliku .parquet.

    Args:
    file_path (str): Ścieżka do danych wejściowych.
    self.file_path = file_path

    @step
    @measure_time
    @count_calls
    def preview_parallel_data(self):
    '''
    Wczytuje pierwszy chunk danych i wypisuje jego podgląd w konsoli.
    Służy jako szybka kontrola zawartości danych przed analizą.
    '''
    for i, chunk in enumerate(load_parquet_in_chunks(self.file_path, chunksize=100_000)):
        print(f"[Chunk {i + 1}] Preview:")
        print(chunk.head())
        break

    @step
    @measure_time
    @count_calls
    def analyze_parallel(self):
    '''
    Wykonuje analizę danych z równoległym przetwarzaniem.
    Profiluje CPU i pamięć, a następnie generuje wykres zużycia pamięci.
    '''
    logger = logging.getLogger(__name__)
    logger.info("Profilowanie CPU i pamięci...")

    def analysis_task():
        parallel_analysis(self.file_path)

    # Profilowanie CPU i pamięci w jednej sesji
    profile_cpu(lambdab: profile_memory(analysis_task))

    # Wizualizacja zużycia pamięci
    memlog_path = "data/profiling/memory_profile.memlog"
    if os.path.exists(memlog_path):
        plot_memory_usage(memlog_path)
    else:
        logger.warning("Nie znaleziono memory_profile.memlog – pominięto wykres pamięci.")

    @step
    @measure_time
    @count_calls
    def visualize(self):
    '''
    Wczytuje próbkę danych (2 chunki), oczyszcza ją i generuje wykresy.
    Służy jako szybka wizualna kontrola jakości i rozkładów danych.
    '''
    df_sample = load_sample_for_visualization(
        self.file_path,
        chunksize=100_000,
        max_chunks=2
    )
    visualize_data(df_sample)

    @step
    @measure_time
    @count_calls
    def generate_reports(self):
    '''
    Generuje raporty tekstowe:
    - summary_by_vendor.txt – statystyki według VendorID,
    - anomalies_report.txt – podejrzone napiwki większe niż całkowita kwota.
    '''
    df = pd.read_parquet(self.file_path)
    save_summary_by_vendor(df)
    save_anomalies_report(df)

    def run(self):
    '''
    Ręczne uruchomienie wszystkich kroków pipeline'u.
    Metoda służy do testów lub wywołania spoza mechanizmu metaklasy.
    '''
    self.preview_parallel_data()
    self.analyze_parallel()
    self.visualize()
    self.generate_reports()
```

Rysunek 13. Kod DataLab/pipeline/taxi_pipeline.py

DataLab/validation/base.py

Plik base.py definiuje abstrakcyjną klasę BaseValidator, która stanowi wspólną bazę dla wszystkich walidatorów danych w systemie. Klasa ta wykorzystuje bibliotekę abc i wymaga od klas dziedziczących implementacji metody validate(df: DataFrame), której zadaniem jest przyjąć dane typu DataFrame i zwrócić ich przefiltrowaną wersję zgodnie z określoną regułą walidacyjną.

```
from abc import ABC, abstractmethod
import pandas as pd

class BaseValidator(ABC):
    """
    Abstrakcyjna klasa bazowa dla wszystkich walidatorów danych.

    Każda klasa dziedzicząca musi zaimplementować metodę 'validate', która
    przyjmuje DataFrame i zwraca przefiltrowany DataFrame zgodnie z daną regułą.
    """

    @abstractmethod
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Waliduje dane i zwraca DataFrame po filtracji.

        :param df: DataFrame do przefiltrowania
        :return: Przefiltrowany DataFrame
        """
        pass
```

Rysunek 14. Kod DataLab/validation/base.py

DataLab/validation/validation_runner.py

Plik validation_runner.py zarządza walidacją danych wejściowych. Udostępnia funkcje validate_chunk i run_all_validations, które przekazują dane typu DataFrame przez sekwencję walidatorów dziedziczących po BaseValidator. Każdy walidator sprawdza konkretną regułę, np. brak kolumn, NaN, ujemne wartości, czas trwania. Wyniki są przekazywane dalej w łańcuchu. Moduł wykorzystywany jest w pipeline'ie, czyszczeniu i testach.

```
"""
Moduł 'validation_runner' odpowiada za centralne zarządzanie walidacją danych wejściowych.

Zawiera funkcje, które uruchamiają sekwencję walidatorów (implementujących klasę 'BaseValidator')
na danych typu DataFrame, filtrując niepoprawne rekordy i przygotowując dane do dalszego przetwarzania.

Używany m.in. w pipeline, czyszczeniu i testach.
"""

import pandas as pd
from validation.validators import (
    ColumnExistenceValidator,
    NoMissingValuesValidator,
    TripDurationValidator,
    PositivePassengerCountValidator,
    PositiveDistanceValidator,
    PositiveFareValidator,
    PositiveTipValidator,
    ValidDateRangeValidator,
    DropDuplicatesValidator
)

REQUIRED_COLUMNS = [
    "passenger_count", "trip_distance", "tip_amount", "total_amount",
    "fare_amount", "tpep_pickup_datetime", "tpep_dropoff_datetime"
]

def validate_chunk(df: pd.DataFrame) -> pd.DataFrame:
    """
    Przepuszcza dany DataFrame przez zestaw walidatorów.

    Każdy walidator sprawdza określone zasady poprawności:
    - obecność wymaganych kolumn
    - brak wartości NaN
    - dodatnie wartości liczbowe
    - poprawność dat i czasu trwania przejazdu
    - usunięcie duplikatów

    :param df: Surowy DataFrame do walidacji
    :return: Oczyszczony i zweryfikowany DataFrame
    """
    validators = [
        ColumnExistenceValidator(REQUIRED_COLUMNS),
        NoMissingValuesValidator(),
        PositivePassengerCountValidator(),
        PositiveDistanceValidator(),
        PositiveFareValidator(),
        PositiveTipValidator(),
        ValidDateRangeValidator(),
        TripDurationValidator(),
        DropDuplicatesValidator()
    ]

    for validator in validators:
        df = validator.validate(df)

    return df.reset_index(drop=True)

def run_all_validations(df: pd.DataFrame) -> pd.DataFrame:
    """
    Alias dla validate_chunk - stosowany w pipeline i testach.

    :param df: DataFrame do walidacji
    :return: Zweryfikowany i przefiltrowany DataFrame
    """
    return validate_chunk(df)
```

Rysunek 15. Kod DataLab/validation/validation_runner.py

DataLab/validation/validators.py

Plik validators.py zawiera implementacje konkretnych walidatorów dziedziczących po BaseValidator. Każdy odpowiada za jedną regułę filtrowania danych: np. brak duplikatów, dodatnie wartości, poprawność czasu trwania kursu, brak NaN lub wymaganych kolumn. Każda klasa posiada metodę validate, która przyjmuje DataFrame i zwraca przefiltrowany wynik. Walidatory mogą działać niezależnie lub jako zestaw w validation_runner.py.

```
"""
Moduł `validators.py` zawiera implementacje konkretnych walidatorów dziedziczących po `BaseValidator`.
Każdy walidator realizuje jedną konkretną zasadę filtracji danych, np.:
- dodatnie wartości liczbowe,
- brak duplikatów,
- poprawność czasów rozpoczęcia i zakończenia kursu,
- brak brakujących danych.
Walidatory mogą być stosowane niezależnie lub jako sekwencja w `validation_runner.py`.
"""

import pandas as pd
from validation.base import BaseValidator

class PositivePassengerCountValidator(BaseValidator):
    """Przepuszcza tylko rekordy z liczbą pasażerów > 0."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df[df["passenger_count"] > 0]

class PositiveDistanceValidator(BaseValidator):
    """Filtruje rekordy z dodatnią długością trasy (> 0 mil)."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df[df["trip_distance"] > 0]

class PositiveFareValidator(BaseValidator):
    """Akceptuje tylko rekordy, gdzie fare_amount i total_amount są dodatnie (> 0)."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df[(df["fare_amount"] > 0) & (df["total_amount"] > 0)]

class ValidDateRangeValidator(BaseValidator):
    """Usuwa rekordy, gdzie data zakończenia kursu jest wcześniejsza niż data rozpoczęcia."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df[df["tpep_dropoff_datetime"] > df["tpep_pickup_datetime"]]

class DropDuplicatesValidator(BaseValidator):
    """Usuwa zduplikowane wiersze w DataFrame."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df.drop_duplicates()

class PositiveTipValidator(BaseValidator):
    """Usuwa rekordy z ujemną wartością napiwku (tip_amount >= 0)."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df[df["tip_amount"] >= 0]

class ColumnExistenceValidator(BaseValidator):
    """Sprawdza, czy wszystkie wymagane kolumny istnieją w DataFrame.
    :param required_columns: Lista wymaganych nazw kolumn
    :raises ValueError: Gdy brakuje którejkolwiek kolumny
    """
    def __init__(self, required_columns: list[str]):
        self.required_columns = required_columns

    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        missing = [col for col in self.required_columns if col not in df.columns]
        if missing:
            raise ValueError(f"Brakuje wymaganych kolumn: {missing}")
        return df

class NoMissingValuesValidator(BaseValidator):
    """Usuwa rekordy zawierające jakiekolwiek wartości NaN."""
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        return df.dropna()

class TripDurationValidator(BaseValidator):
    """Filtruje rekordy, gdzie czas trwania przejazdu jest ≤ 0 lub > 24h.
    Zakładamy, że kurs nie powinien trwać dłużej niż 86400 sekund (24 godziny).
    """
    def validate(self, df: pd.DataFrame) -> pd.DataFrame:
        duration = (df["tpep_dropoff_datetime"] - df["tpep_pickup_datetime"]).dt.total_seconds()
        return df[(duration > 0) & (duration < 86400)]
```

Rysunek 16. Kod DataLab/validation/validators.py.

DataLab/streamlit_app.py

Plik streamlit_app.py uruchamia frontendowy interfejs do wizualizacji wyników analizy danych NYC Yellow Taxi. Wykorzystuje bibliotekę Streamlit do obsługi widoku, umożliwia uruchomienie pipeline'u, podgląd danych, generowanych wykresów, raportów tekstowych oraz logów aplikacji. Obsługuje różne zakładki (m.in. „Wykresy”, „Raporty tekstowe”, „Podgląd danych”, „Logi”), a uruchomienie analizy za pomocą przycisku powoduje przetworzenie danych przez klasę TaxiPipeline. Wykresy i raporty ładowane są z katalogu data/output, a logi z data/logs/app.log.

```
"""
Moduł 'streamlit_app.py' uruchamia interfejs webowy dla projektu analizy danych NYC Yellow Taxi.
Wykorzystuje bibliotekę Streamlit do wizualizacji wyników przetwarzania danych:
- uruchomienie pipeline'u (TaxiPipeline)
- podgląd danych surowych
- prezentacja wykresów i raportów
- przegląd logów
Plik może być uruchamiany samodzielnie jako aplikacja frontendowa.
"""

import streamlit as st
import os
import pandas as pd
from core.logger import logger as app_logger
from pipeline.taxi_pipeline import TaxiPipeline

# Zdefiniuj ścieżki i pliki
OUTPUT_DIR = "data/output"
RAW_DATA_PATH = "data/raw/yellow_tripdata_2024-01.parquet"
LOG_FILE_PATH = "data/logs/app.log"

# Wykresy do wyświetlenia
PLOTS = {
    "Liczba pasażerów a długość przejazdu": "passenger_vs_distance_heatmap.png",
    "Mapki - rozrzut wartości": "tip_amount_boxplot.png",
    "Mapki vs całkowita cena": "tip_vs_total_scatter.png",
    "Długość przejazdu - histogram": "trip_distance_hist_filtered.png",
    "Mapki a liczba pasażerów": "tip_by_passenger_count_filtered.png",
    "Zużycie pamięci podczas analizy": "memory_usage_plot.png"
}

# Raporty tekstowe
TEXT_REPORTS = {
    "Podsumowanie analizy równoległej": "parallel_summary.txt",
    "Raport anomalii": "anomalies_report.txt",
    "Raport podsumujący przebiegów": "summary_by_vendor.txt"
}

def show_image(file_name: str) -> None:
    Wyświetla wykres PNG z katalogu 'data/output'.
    :param file_name: Nazwa pliku wykresu (np. "trip_distance_hist.png")
    full_path = os.path.join(OUTPUT_DIR, file_name)
    if os.path.exists(full_path):
        st.image(full_path, use_column_width=True)
    else:
        st.warning(f"Brak wykresu: {file_name}")
        app_logger.warning(f"[Streamlit] Brak pliku graficznego: {full_path}")

def show_text(file_name: str) -> None:
    Wyświetla zawartość pliku tekstowego (np. raportu).
    :param file_name: Nazwa pliku z raportem (np. "summary.txt")
    full_path = os.path.join(OUTPUT_DIR, file_name)
    if os.path.exists(full_path):
        with open(full_path, "r", encoding="utf-8") as f:
            st.text(f.read())
    else:
        st.warning(f"Brak raportu: {file_name}")
        app_logger.warning(f"[Streamlit] Brak pliku tekstowego: {full_path}")

def show_logs() -> None:
    Wyświetla ostatnie 100 linii z logów aplikacji ('app.log').
    if os.path.exists(LOG_FILE_PATH):
        with open(LOG_FILE_PATH, "r", encoding="utf-8") as log_file:
            log = log_file.readlines()[-100:]
            st.text("\n".join(logs))
    else:
        st.error("Brak pliku logów.")
        app_logger.warning(f"[Streamlit] Brak pliku logów.")

def preview_raw_data() -> None:
    Wczytuje pierwsze 10 rekordów z pliku Parquet i wyświetla jako DataFrame.
    if os.path.exists(RAW_DATA_PATH):
        try:
            df = pd.read_parquet(RAW_DATA_PATH, engine="pyarrow")
            st.dataframe(df.head(10))
        except Exception as e:
            st.error(f"Nie udało się załadować danych: {e}")
            app_logger.exception(f"[Streamlit] Błąd przy czytaniu Parquet")
    else:
        st.error("Brak pliku danych wejściowych.")
        app_logger.warning(f"[Streamlit] Brak pliku wejściowego: Parquet")

def main() -> None:
    Główna funkcja uruchamiająca interfejs Streamlit.
    Obsługuje zakładki, uruchomienie pipeline'u, ładowanie danych, wykresy i raporty.
    st.set_page_config(layout="wide", page_title="DataLab - NYC Taxi Analysis")
    st.title("DataLab - Interaktywna analiza przejazdów taksówkami w NYC")
    tab = st.sidebar.radio("Co chcesz zobaczyć?", [
        "Wykresy",
        "Raporty tekstowe",
        "Wszystkie wykresy",
        "Podgląd danych",
        "Debug / Logi aplikacji"
    ])
    if st.sidebar.button("Odśwież / Uruchom analizę"):
        st.info("Uruchamiam pipeline analizy danych...")
        try:
            pipeline = TaxiPipeline(file_path=RAW_DATA_PATH)
            pipeline.run()
            st.success("Pipeline zakończony powodzeniem.")
        except Exception as e:
            st.error(f"Wystąpił błąd: {e}")
            app_logger.exception(f"[Streamlit] Błąd podczas uruchamiania pipeline")
    if tab == "Wykresy":
        st.header("Dostępne wizualizacje")
        available_plots = [k for k, v in PLOTS.items() if os.path.exists(os.path.join(OUTPUT_DIR, v))]
        if available_plots:
            selected_plot = st.selectbox("Wybierz wykres:", available_plots)
            st.subheader(selected_plot)
            show_image(PLOTS[selected_plot])
        else:
            st.error("Brak dostępnych wykresów. Uruchom analizę danych.")
            st.divider()
    elif tab == "Raporty tekstowe":
        st.header("Raporty z analizy")
        available_reports = [k for k, v in TEXT_REPORTS.items() if os.path.exists(os.path.join(OUTPUT_DIR, v))]
        if available_reports:
            selected_report = st.selectbox("Wybierz raport:", available_reports)
            st.subheader(selected_report)
            show_text(TEXT_REPORTS[selected_report])
        else:
            st.error("Brak dostępnych raportów. Uruchom analizę.")
            st.divider()
    elif tab == "Wszystkie wykresy":
        st.header("Dashboard - Wszystkie dostępne wykresy")
        for name, filename in PLOTS.items():
            full_path = os.path.join(OUTPUT_DIR, filename)
            if os.path.exists(full_path):
                st.subheader(name)
                show_image(filename)
            else:
                st.info(f"Pominięto: {name} (brak pliku)")
            st.divider()
    elif tab == "Podgląd danych":
        st.header("Podgląd danych źródłowych")
        preview_raw_data()
        st.divider()
    elif tab == "Debug / Logi aplikacji":
        st.header("Logi aplikacji")
        show_logs()
        st.divider()
    st.caption("Dane: NYC Yellow Taxi | Projekt: DataLab")
if __name__ == "__main__":
    main()
```

Rysunek 17. Kod DataLab/streamlit_app.py.

DataLab/main.py

Plik main.py jest punktem startowym aplikacji Datalab. Sprawdza, czy wszystkie wymagane pliki wyjściowe (wykresy, raporty, plik profilujący) już istnieją. Jeśli nie, uruchamia przetwarzanie danych przez klasę TaxiPipeline, generując brakujące pliki. Następnie automatycznie uruchamia aplikację Streamlit (streamlit_app.py) jako graficzny interfejs użytkownika. To główna funkcja integrująca cały projekt w spójną aplikację.

```
"""
Moduł 'main.py' to punkt startowy aplikacji Datalab.

Sprawdza obecność wymaganych plików wyjściowych (wykresy, raporty, pliki profilujące).
Jeśli ich brakuje, uruchamia TaxiPipeline do przetwarzania danych.
Następnie odpala interfejs Streamlit ('streamlit_app.py').
"""

import os
import subprocess
from pipeline.taxi_pipeline import TaxiPipeline
from core.logger import logger

# Ścieżki do plików i folderów
RAW_DATA_PATH = "data/raw/yellow_tripdata_2024-01.parquet"
OUTPUT_DIR = "data/output"
PROFILING_DIR = "data/profiling"

# Lista plików, które muszą istnieć po przetworzeniu
NEEDED_FILES = [
    # Wykresy
    "passenger_vs_distance_heatmap.png",
    "tip_amount_boxplot.png",
    "tip_vs_total_scatter.png",
    "trip_distance_hist_filtered.png",
    "tip_by_passenger_count_filtered.png",
    "memory_usage_plot.png",

    # Raporty tekstowe
    "parallel_summary.txt",
    "vendor_analysis.txt",
    "anomalies_report.txt",

    # Profilowanie CPU i RAM
    "cpu_profile.prof",
    "memory_profile.memlog"
]

def is_output_complete() -> bool:
    """
    Sprawdza, czy wszystkie wymagane pliki wyjściowe już istnieją.
    :return: True jeśli wszystkie pliki obecne, False w przeciwnym razie.
    """
    return all(
        os.path.exists(
            os.path.join(OUTPUT_DIR if f.endswith(("png", ".txt")) else PROFILING_DIR, f)
        ) for f in NEEDED_FILES
    )

def generate_outputs() -> None:
    """
    Uruchamia pipeline 'TaxiPipeline', jeśli plik źródłowy istnieje.
    Generuje wykresy, raporty oraz pliki profilowania.
    """
    if not os.path.exists(RAW_DATA_PATH):
        logger.error(f"Nie znaleziono pliku danych: {RAW_DATA_PATH}")
        return

    logger.info("Uruchamiam pipeline...")
    pipeline = TaxiPipeline(RAW_DATA_PATH)
    pipeline.run()
    logger.info("Pipeline zakończony pomyślnie.")

def main() -> None:
    """
    Główna funkcja:
    - sprawdza, czy dane wyjściowe już istnieją,
    - w razie potrzeby uruchamia pipeline,
    - odpala aplikację Streamlit.
    """
    logger.info("Uruchamianie aplikacji Datalab")

    if not is_output_complete():
        logger.warning("Brakuje plików wyjściowych – wykonuję pipeline.")
        generate_outputs()
    else:
        logger.info("Pliki już istnieją – pomijam pipeline.")

    logger.info("Odpalam Streamlit...")
    subprocess.run(["streamlit", "run", "streamlit_app.py"], check=True)

if __name__ == "__main__":
    main()
```

Rysunek 18. Kod DataLab/main.py.

Testy

Poniżej przedstawiono testy systemu.

DataLab/tests/test_cleaner.py

Plik test_cleaner.py zawiera zestaw testów jednostkowych dla funkcji clean_data z modułu core.cleaner. Sprawdza, czy prawidłowe rekordy przechodzą walidację bez zmian, czy błędne dane są usuwane (ujemne wartości, 0, błędne daty, wartości NaN), oraz czy pusty DataFrame nie powoduje błędów. Dzięki temu moduł zapewnia wysoką jakość danych wejściowych przed dalszym przetwarzaniem.

```
'''
test_cleaner.py

Zestaw testów jednostkowych dla funkcji 'clean_data' z modułu core.cleaner.

Testy sprawdzają:
- czy poprawne dane przechodzą walidację,
- czy błędne dane są usuwane (ujemne wartości, zera, błędne daty),
- czy obsługiwane są wartości NaN,
- czy pusty DataFrame nie powoduje błędów.
'''

import pandas as pd
from core.cleaner import clean_data

def test_clean_data_valid_row_passes():
    '''
    Poprawny rekord powinien przejść walidację bez zmian.
    '''
    df = pd.DataFrame([
        {
            "trip_distance": 2.5,
            "fare_amount": 12.0,
            "passenger_count": 1,
            "tip_amount": 3.0,
            "total_amount": 15.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-01 10:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-01 10:30")
        }
    ])

    cleaned = clean_data(df)

    assert len(cleaned) == 1
    row = cleaned.iloc[0]
    assert row["trip_distance"] >= 0
    assert row["fare_amount"] > 0
    assert row["passenger_count"] > 0
    assert row["total_amount"] > 0
    assert row["tpep_dropoff_datetime"] > row["tpep_pickup_datetime"]

def test_clean_data_removes_invalid_rows():
    '''
    Rekordy z błędnymi wartościami powinny zostać usunięte.
    Zostanie tylko jeden poprawny rekord.
    '''
    df = pd.DataFrame([
        {
            "trip_distance": -1.0,
            "fare_amount": 5.0,
            "passenger_count": 1,
            "tip_amount": 1.0,
            "total_amount": 6.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-01 10:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-01 10:30")
        },
        {
            "trip_distance": 2.0,
            "fare_amount": 0.0,
            "passenger_count": 1,
            "tip_amount": 1.0,
            "total_amount": 3.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-01 10:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-01 10:30")
        },
        {
            "trip_distance": 1.5,
            "fare_amount": 10.0,
            "passenger_count": 0,
            "tip_amount": 2.0,
            "total_amount": 12.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-01 10:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-01 10:30")
        },
        {
            "trip_distance": 3.0,
            "fare_amount": 8.0,
            "passenger_count": 1,
            "tip_amount": 2.0,
            "total_amount": 10.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-01 12:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-01 11:00")
        },
        {
            "trip_distance": 1.0,
            "fare_amount": 5.0,
            "passenger_count": 1,
            "tip_amount": 1.0,
            "total_amount": 6.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-02 09:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-02 09:30")
        }
    ])

    cleaned = clean_data(df)

    assert len(cleaned) == 1, f"Powinno zostać tylko 1 poprawny rekord, a dostałem {len(cleaned)}"
    pickup = cleaned.iloc[0]["tpep_pickup_datetime"]
    assert pickup == pd.to_datetime("2024-01-02 09:00")

def test_clean_data_handles_nan_values():
    '''
    Rekord z wartością NaN powinien zostać usunięty.
    '''
    df = pd.DataFrame([
        {
            "trip_distance": 2.0,
            "fare_amount": None,
            "passenger_count": 1,
            "tip_amount": 1.0,
            "total_amount": 10.0,
            "tpep_pickup_datetime": pd.to_datetime("2024-01-01 10:00"),
            "tpep_dropoff_datetime": pd.to_datetime("2024-01-01 10:30")
        }
    ])

    cleaned = clean_data(df)
    assert cleaned.empty, "Rekord z NaN powinien zostać wyrzucony"

def test_clean_data_empty_df_returns_empty():
    '''
    Pusty DataFrame powinien zwrócić pusty wynik (bez błędów).
    '''
    df = pd.DataFrame(columns=[
        "trip_distance", "fare_amount", "passenger_count",
        "tip_amount", "total_amount", "tpep_pickup_datetime", "tpep_dropoff_datetime"
    ])

    cleaned = clean_data(df)
    assert cleaned.empty, "Pusty DataFrame powinien zwrócić pusty wynik"
```

Rysunek 19. Kod DataLab/tests/test_cleaner.py.

DataLab/tests/test_decorators.py

Plik `test_decorators.py` zawiera testy jednostkowe dla dekoratorów `@count_calls` i `@measure_time`. Sprawdza, czy dekorowana funkcja zwraca poprawny wynik, czy licznik wywołań zwiększa się prawidłowo oraz czy na wyjściu pojawiają się odpowiednie komunikaty (czas wykonania i liczba wywołań). Testy wykorzystują funkcję `dummy_function`, która jest celowo opóźniona, aby umożliwić pomiar czasu, a także przechwytywać komunikaty z konsoli i weryfikują ich treść.

```
"""
test_decorators.py

Test jednostkowy dla dekoratorów '@count_calls' i '@measure_time'.

Sprawdza:
- czy dekorowana funkcja zwraca poprawny wynik,
- czy licznik wywołań działa poprawnie,
- czy na wyjściu pojawiają się komunikaty z dekoratorów (czas i liczba wywołań).
"""

import time
from decorators.counter import count_calls
from decorators.timer import measure_time

# Globalna zmienna do sprawdzenia działania dekoratora count_calls
call_counter = {"count": 0}

@count_calls
@measure_time
def dummy_function():
    """
    Przykładowa funkcja testowa dekorowana przez count_calls i measure_time.
    """
    call_counter["count"] += 1
    time.sleep(0.01) # sztuczne opóźnienie do pomiaru czasu
    return sum(range(1000))

def test_count_calls_and_timer(capsys):
    """
    Testuje dekoratory count_calls i measure_time na przykładzie dummy function.
    Sprawdza wynik działania funkcji oraz obecność odpowiednich komunikatów.
    """
    result = dummy_function()
    captured = capsys.readouterr()

    # Czy funkcja coś zwraca?
    assert isinstance(result, int)

    # Czy licznik działa?
    assert call_counter["count"] == 1

    # Czy dekoratory wypisały odpowiednie logi
    assert "[Timer]" in captured.out
    assert "wykonała się w" in captured.out or "ms" in captured.out.lower()

    assert "[Counter]" in captured.out
    assert "wywołana 1 raz" in captured.out
```

Rysunek 20. Kod `DataLab/tests/test_decorators.py`.

DataLab/tests/test_validators.py

Plik `test_validators.py` zawiera testy jednostkowe dla funkcji `run_all_validations` z modułu `validation_runner`. Sprawdza, czy poprawne dane przechodzą walidację bez zmian, a błędne są skutecznie odrzucane. Testy obejmują przypadki braku wymaganych kolumn, obecności wartości NaN, niepoprawnych zakresów czasowych, wartości ujemnych lub zerowych oraz duplikatów. Dodatkowo, testowane jest poprawne podnoszenie błędów (`ValueError`) przy niezgodności schematu.

```

"""
test_validators.py

Testy jednostkowe dla funkcji `run_all_validations` z modułu `validation_runner`.

Sprawdzane przypadki:
- poprawna walidacja czystych danych,
- odrzucanie niepoprawnych lub niekompletnych wierszy,
- sprawdzanie poprawności kolumn i wartości null,
- wykrywanie złych zakresów czasu i duplikatów.
"""

import pandas as pd
import pytest
from validation.validation_runner import run_all_validations

def test_validators_pass_on_clean_data():
    """
    Dane spełniające wszystkie kryteria powinny przejść walidację bez zmian.
    """
    df = pd.DataFrame({
        "trip_distance": [1.0, 2.0],
        "fare_amount": [10.0, 20.0],
        "total_amount": [15.0, 25.0],
        "passenger_count": [1, 2],
        "tip_amount": [1.5, 2.0],
        "tpep_pickup_datetime": pd.to_datetime(["2024-01-01 00:00", "2024-01-02 00:00"]),
        "tpep_dropoff_datetime": pd.to_datetime(["2024-01-01 01:00", "2024-01-02 01:00"])
    })

    validated_df = run_all_validations(df)
    assert len(validated_df) == 2

def test_validators_remove_invalid_rows():
    """
    Rzędy z błędnymi wartościami (ujemne, zero, złe daty) powinny zostać usunięte.
    """
    df = pd.DataFrame({
        "trip_distance": [1.0, -5.0],
        "fare_amount": [10.0, 0.0],
        "total_amount": [15.0, -1.0],
        "passenger_count": [1, 0],
        "tip_amount": [1.5, -3.0],
        "tpep_pickup_datetime": pd.to_datetime(["2024-01-01 00:00", "2024-01-01 02:00"]),
        "tpep_dropoff_datetime": pd.to_datetime(["2024-01-01 01:00", "2024-01-01 01:30"])
    })

    validated_df = run_all_validations(df)
    assert len(validated_df) == 1

def test_validator_raises_on_missing_columns():
    """
    Brak wymaganych kolumn powinien skutkować wyjątkiem ValueError.
    """
    df = pd.DataFrame({
        "trip_distance": [1.0],
        "fare_amount": [10.0]
    })

    with pytest.raises(ValueError, match="Brakuje wymaganych kolumn"):
        run_all_validations(df)

def test_validator_removes_rows_with_missing_values():
    """
    Rekordy zawierające NaN powinny zostać odrzucone.
    """
    df = pd.DataFrame({
        "trip_distance": [1.0, None],
        "fare_amount": [10.0, 5.0],
        "total_amount": [15.0, 10.0],
        "passenger_count": [1, 1],
        "tip_amount": [1.5, 2.0],
        "tpep_pickup_datetime": pd.to_datetime(["2024-01-01 00:00", "2024-01-02 00:00"]),
        "tpep_dropoff_datetime": pd.to_datetime(["2024-01-01 01:00", "2024-01-02 01:00"])
    })

    validated_df = run_all_validations(df)
    assert len(validated_df) == 1

def test_trip_duration_filtering():
    """
    Kursy z czasem trwania <= 0 lub > 24h powinny zostać odrzucone.
    """
    df = pd.DataFrame({
        "trip_distance": [1.0, 1.0],
        "fare_amount": [10.0, 10.0],
        "total_amount": [15.0, 15.0],
        "passenger_count": [1, 1],
        "tip_amount": [1.0, 1.0],
        "tpep_pickup_datetime": pd.to_datetime(["2024-01-01 00:00", "2024-01-01 00:00"]),
        "tpep_dropoff_datetime": pd.to_datetime(["2024-01-01 00:00", "2024-01-02 01:00"]) # 0s i 25h
    })

    validated_df = run_all_validations(df)
    assert validated_df.empty

def test_duplicates_are_removed():
    """
    Duplikaty (identyczne wiersze) powinny zostać usunięte.
    """
    df = pd.DataFrame({
        "trip_distance": [1.0, 1.0],
        "fare_amount": [10.0, 10.0],
        "total_amount": [15.0, 15.0],
        "passenger_count": [1, 1],
        "tip_amount": [1.0, 1.0],
        "tpep_pickup_datetime": pd.to_datetime(["2024-01-01 00:00", "2024-01-01 00:00"]),
        "tpep_dropoff_datetime": pd.to_datetime(["2024-01-01 01:00", "2024-01-01 01:00"])
    })

    validated_df = run_all_validations(df)
    assert len(validated_df) == 1

```

Rysunek 21. Kod DataLab/tests/test_validators.py.

DataLab/tests/loader.py

Plik `test_loader.py` zawiera testy jednostkowe dla funkcji `load_parquet_in_chunks` z modułu `core.loader`. Testy sprawdzają poprawność ładowania danych z pliku Parquet w chunkach, poprawne zachowanie przy nieistniejącej ścieżce (funkcja powinna zwracać pustą listę, a nie rzucać wyjątek) oraz reakcję na symulowany pusty plik (poprzez monkeypatching `read_parquet` tak, by zwracał pusty `DataFrame`). Wszystkie testy oceniają, czy wynik jest zgodny z oczekiwanym, niepusta lista chunków przy poprawnym pliku, pusta przy błędnym lub pustym.

```
"""
test_loader.py

Testy jednostkowe dla funkcji 'load_parquet_in_chunks' z modułu 'core.loader'.

Sprawdzane przypadki:
- poprawne wczytywanie danych z dużego pliku Parquet w chunkach,
- obsługa nieistniejącej ścieżki (zwraca pustą listę zamiast wyjątku),
- poprawne zachowanie przy pustym pliku/parquet mockowanym do pustego DataFrame.
"""

import os
import pandas as pd
from core.loader import load_parquet_in_chunks

def test_load_parquet_in_chunks_reads_data():
    """
    Funkcja powinna poprawnie wczytać dane w chunkach z istniejącego pliku Parquet.
    """
    path = "data/raw/yellow_tripdata_2024-01.parquet"
    assert os.path.exists(path), f"Plik nie istnieje: {path}"

    chunks = list(load_parquet_in_chunks(path, chunksize=100_000))
    assert len(chunks) > 0, "Nie wczytano żadnych chunków"

    for chunk in chunks:
        assert isinstance(chunk, pd.DataFrame)
        assert not chunk.empty
        assert len(chunk) > 0

def test_load_parquet_in_chunks_invalid_path():
    """
    Dla nieistniejącej ścieżki powinien zostać zwrócony pusty wynik (lista).
    """
    invalid_path = "data/raw/fake_file.parquet"
    chunks = list(load_parquet_in_chunks(invalid_path, chunksize=100_000))
    assert chunks == [], "Dla nieistniejącego pliku powinien być pusty wynik"

def test_load_parquet_in_chunks_empty_file(monkeypatch):
    """
    Gdy 'read_parquet' zwraca pusty DataFrame (symulacja pustego pliku),
    funkcja powinna zwrócić pustą listę.
    """
    def fake_parquet(*args, **kwargs):
        return pd.DataFrame()

    monkeypatch.setattr(pd, "read_parquet", fake_parquet)

    chunks = list(load_parquet_in_chunks("fake_path.parquet", chunksize=10_000))
    assert chunks == [], "Pusty plik powinien dawać pusty wynik"
```

Rysunek 22. Kod `DataLab/tests/loader.py`.

DataLab/tests/test_pool_processor.py

Plik `test_pool_processor.py` zawiera testy jednostkowe dla funkcji `parallel_analysis` z modułu `core.pool_processor`. Pierwszy test sprawdza, czy analiza równoległa na poprawnym pliku `.parquet` zwraca słownik z oczekiwanymi kluczami (takimi jak liczba rekordów, średnia długość trasy, itp.) oraz czy generowane są pliki wyjściowe (`summary.txt`, `summary_by_vendor.txt`, `anomalies_report.txt`). Drugi test weryfikuje, że dla nieistniejącej ścieżki funkcja nie rzuca wyjątku, ale zwraca pusty słownik.

```

"""
test_pool_processor.py

Testy jednostkowe dla funkcji `parallel_analysis` z modułu `core.pool_processor`.

Sprawdzone przypadki:
- poprawna analiza pliku `.parquet` z danymi (czy generuje wynik i pliki wyjściowe),
- obsługa błędnej/niewłaściwej ścieżki (czy zwraca pusty słownik).
"""

import os
from core.pool_processor import parallel_analysis

def test_parallel_analysis_runs():
    """
    Testuje pełną analizę równoległą na istniejącym pliku danych.
    Sprawdza strukturę słownika wynikowego oraz istnienie plików wynikowych.
    """
    path = "data/raw/yellow_tripdata_2024-01.parquet"
    assert os.path.exists(path), f"Plik nie istnieje: {path}"

    result = parallel_analysis(path, chunksize=100_000)

    assert isinstance(result, dict), "Wynik powinien być słownikiem"

    expected_keys = [
        "Liczba rekordów",
        "Średnia długość trasy (mile)",
        "Średni napiwek ($)",
        "Łączna kwota opłat ($)",
        "Średnia opłata za kurs ($)",
        "Liczba pasażerów (łącznie)",
        "Średnia liczba pasażerów na kurs",
        "Liczba długich kursów (>10 mil)"
    ]

    for key in expected_keys:
        assert key in result, f"Brakuje klucza: {key} w wyniku analizy"

    assert result["Liczba rekordów"] > 0
    assert result["Liczba pasażerów (łącznie)"] > 0
    assert result["Łączna kwota opłat ($)"] > 0.0

    # Sprawdź czy pliki wynikowe się utworzyły
    assert os.path.exists("data/output/parallel_summary.txt"), "Brak pliku podsumowania"
    assert os.path.exists("data/output/summary_by_vendor.txt"), "Brak pliku per VendorID"
    assert os.path.exists("data/output/anomalies_report.txt"), "Brak pliku z anomaliami"

def test_parallel_analysis_invalid_path():
    """
    Dla nieistniejącej ścieżki funkcja powinna zwrócić pusty słownik.
    """
    path = "data/raw/nonexistent_file.parquet"
    result = parallel_analysis(path, chunksize=100_000)

    assert result == {}, "Dla nieistniejącego pliku wynik powinien być pustym słownikiem"

```

Rysunek 23. Kod DataLab/tests/test_pool_processor.py

Wynik działania testów

```

Terminal Local x
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

(.venv) PS C:\Users\kuliq\PycharmProjects\DataLab> pytest tests/
===== test session starts =====
platform win32 -- Python 3.11.5, pytest-8.4.1, pluggy-1.6.0
rootdir: C:\Users\kuliq\PycharmProjects\DataLab
configfile: pytest.ini
collected 16 items

tests\test_cleaner.py .... [ 25%]
tests\test_decorators.py . [ 31%]
tests\test_loader.py ... [ 50%]
tests\test_pool_processor.py .. [ 62%]
tests\test_validators.py ..... [100%]

===== 16 passed in 5.99s =====
(.venv) PS C:\Users\kuliq\PycharmProjects\DataLab>

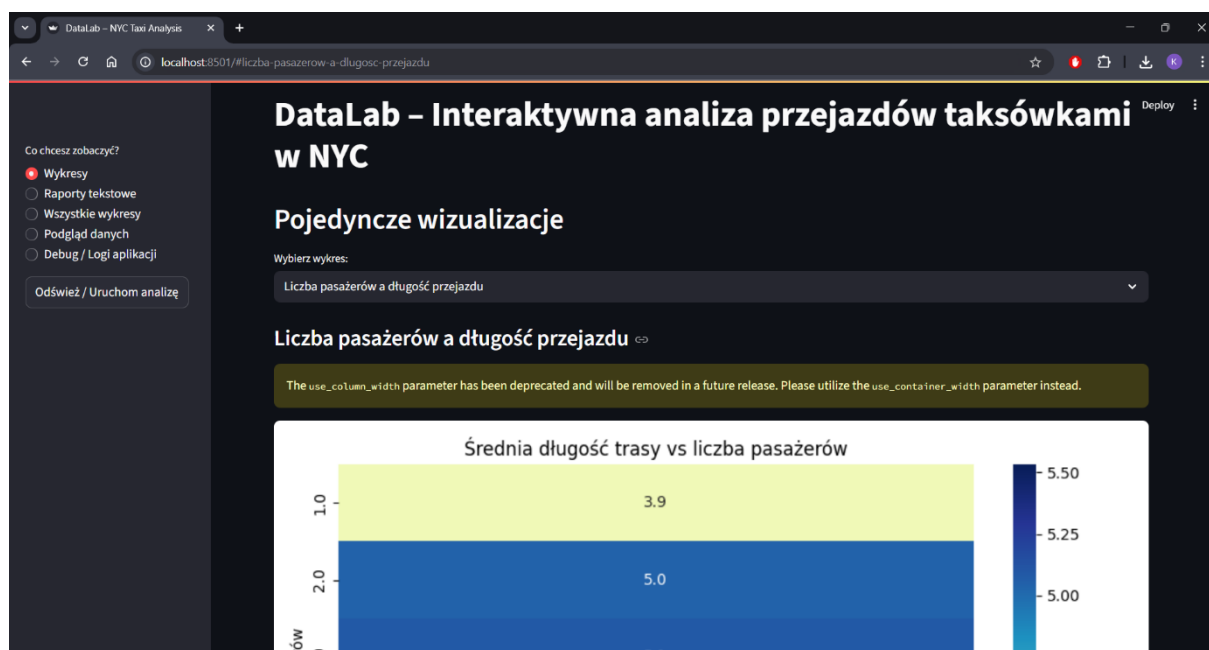
```

Rysunek 24. Uruchomienie oraz wynik testów jednostkowych

Prezentacja warstwy użytkowej

Ekran startowy

Ten ekran przedstawia sekcję Pojedyncze wizualizacje w aplikacji DataLab. Użytkownik może wybrać konkretny wykres do analizy. Po lewej stronie dostępne są inne zakładki nawigacyjne.

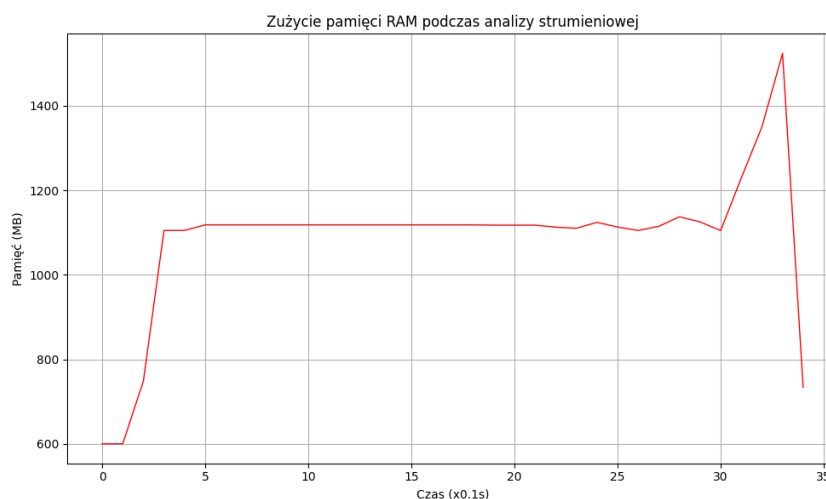


Rysunek 25. Ekran startowy aplikacji.

Wykresy do wyboru:

1. Zużycie pamięci RAM podczas analizy strumieniowej

Wykres liniowy pokazujący zmienność zużycia pamięci RAM (w MB) w czasie działania analizy strumieniowej. Wskazuje na stabilne zużycie pamięci z krótkotrwałymi skokami w późniejszej fazie przetwarzania danych.



Rysunek 26. Wykres -Zużycie pamięci RAM podczas analizy strumieniowej

2. Liczba pasażerów a długość przejazdu

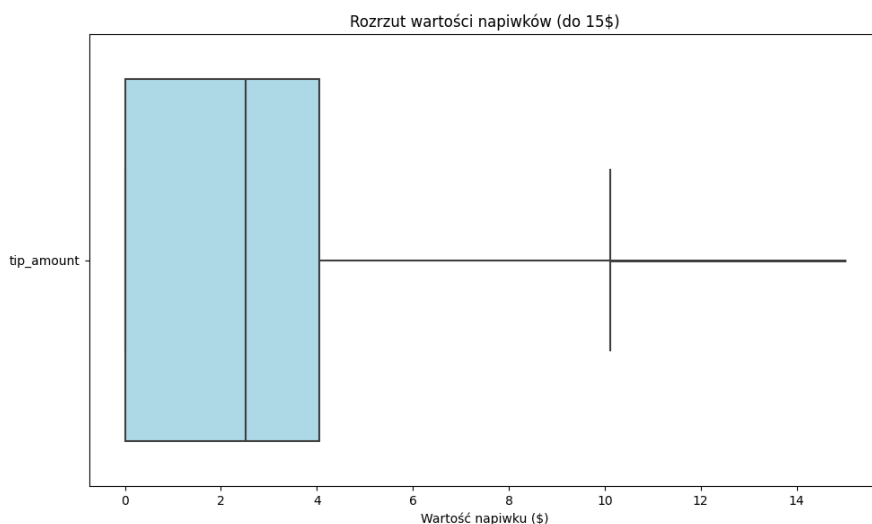
Mapa cieplna prezentująca średnią długość przejazdu (w milach) w zależności od liczby pasażerów. Najdłuższe trasy występowały przy czterech pasażerach – średnio 5.5 mili.



Rysunek 27. Wykres - Liczba pasażerów a długość przejazdu.

3. Rozrzut wartości napiwków (do 15\$)

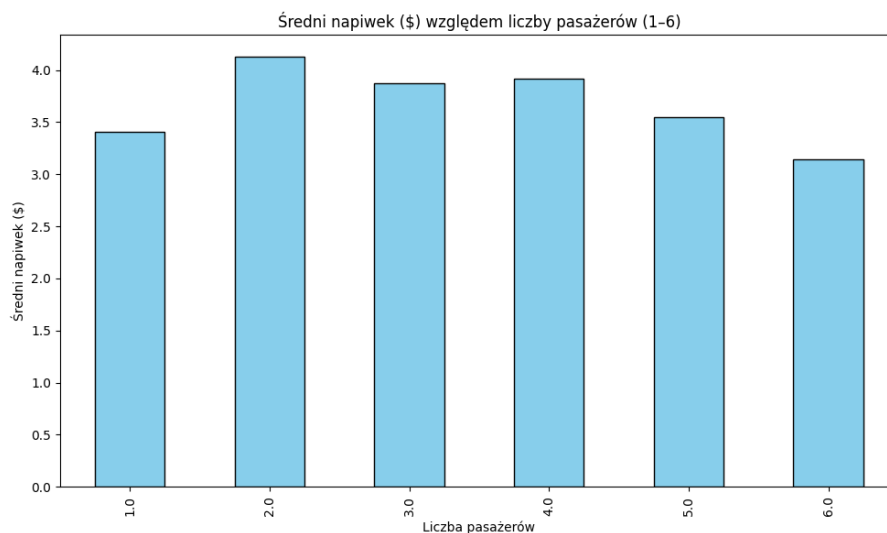
Wykres pudełkowy ilustrujący rozkład wartości napiwków. Pokazuje medianę, rozrzut danych oraz obserwacje odstające, co pozwala zidentyfikować typowe oraz nietypowe wartości napiwków.



Rysunek 28. Wykres - Rozrzut wartości napiwków (do 15\$).

4. Średni napiwek względem liczby pasażerów (1–6)

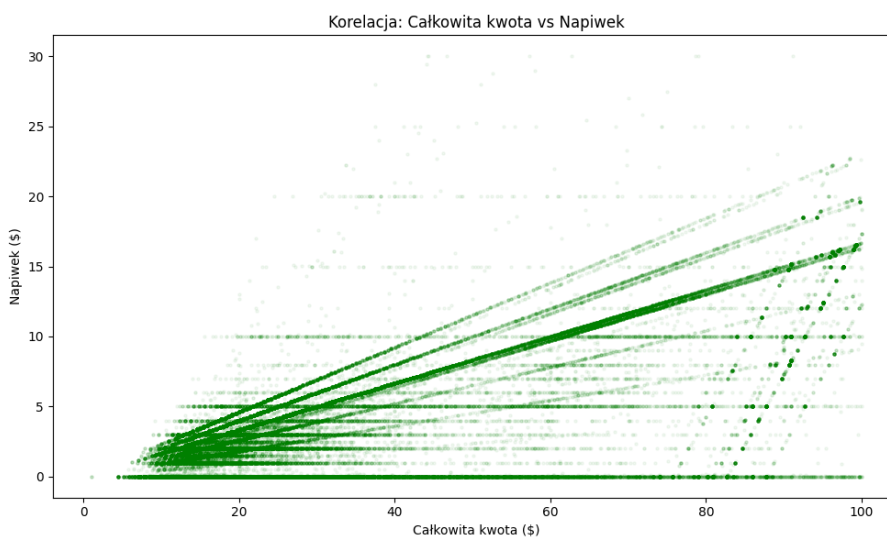
Wykres słupkowy prezentujący średni napiwek dla różnych liczby pasażerów. Najwyższe średnie napiwki występowały przy dwóch pasażerach, natomiast najniższe przy sześciu.



Rysunek 29. Wykres - Średni napiwek względem liczby pasażerów (1-6).

5. Korelacja: Całkowita kwota vs napiwek

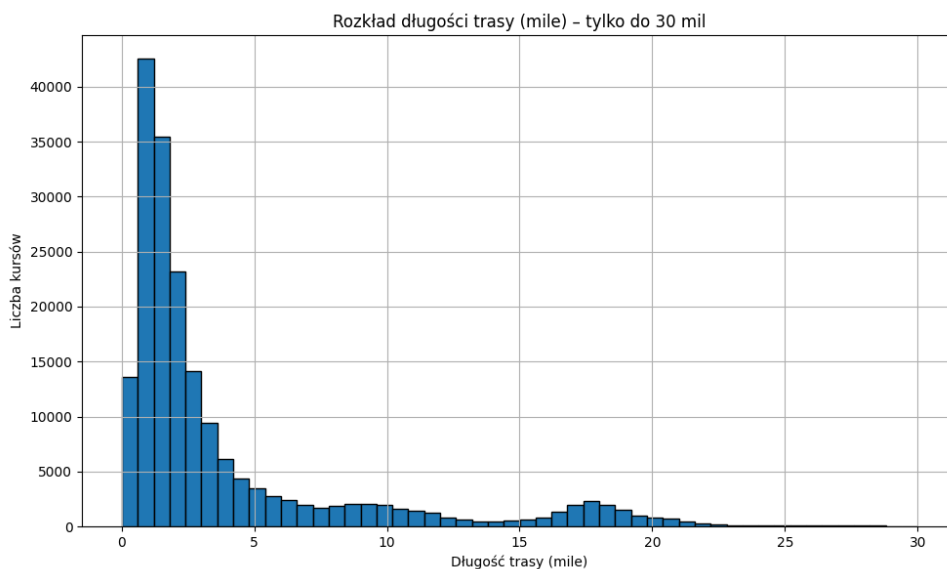
Wykres rozrzutu pokazujący zależność między całkowitą opłatą a wysokością napiwku. Widoczna jest dodatnia korelacja – im wyższa opłata, tym wyższy napiwek.



Rysunek 30. Wykres - Korelacja: Całkowita kwota vs napiwek.

6. Rozkład długości trasy (mile) – tylko do 30 mil

Histogram przedstawiający liczbę kursów dla różnych długości trasy. Zdecydowana większość przejazdów mieści się w przedziale do 5 mil, co świadczy o krótkim dystansie typowych kursów w NYC.

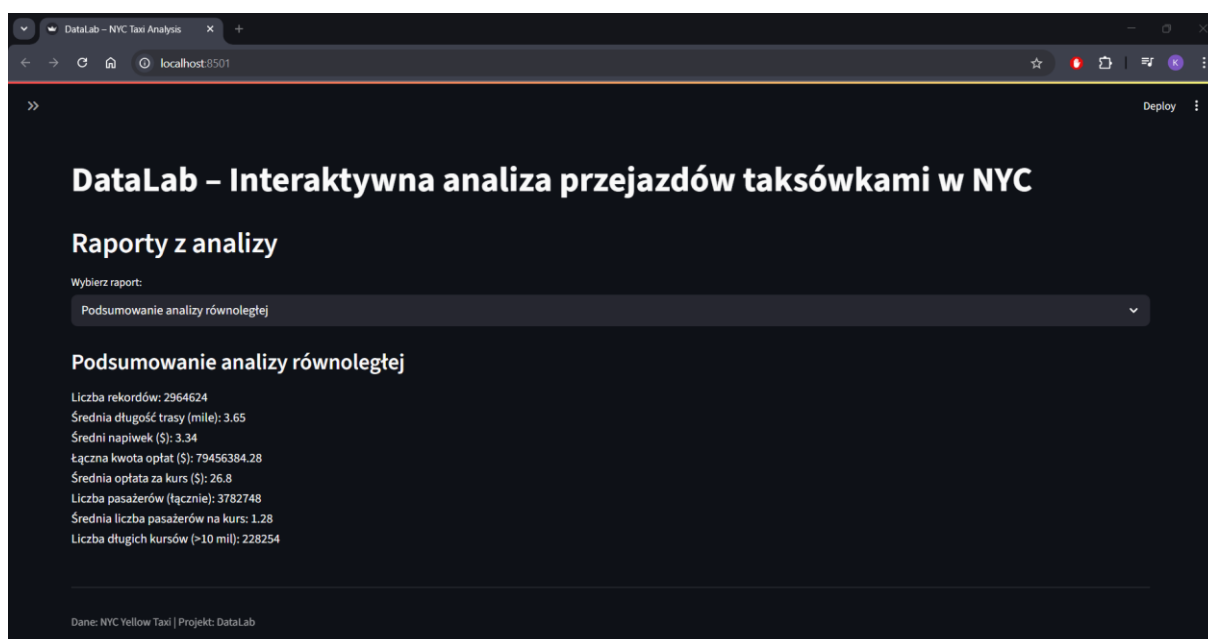


Rysunek 31. Wykres - Rozkład długości trasy (mile) – tylko do 30 mil.

Widok z podsumowaniem analizy równoległej

Ten ekran prezentuje ogólne statystyki zbiorcze przetworzonych danych o przejazdach taksówkami w NYC. Wyświetlane są m.in.:

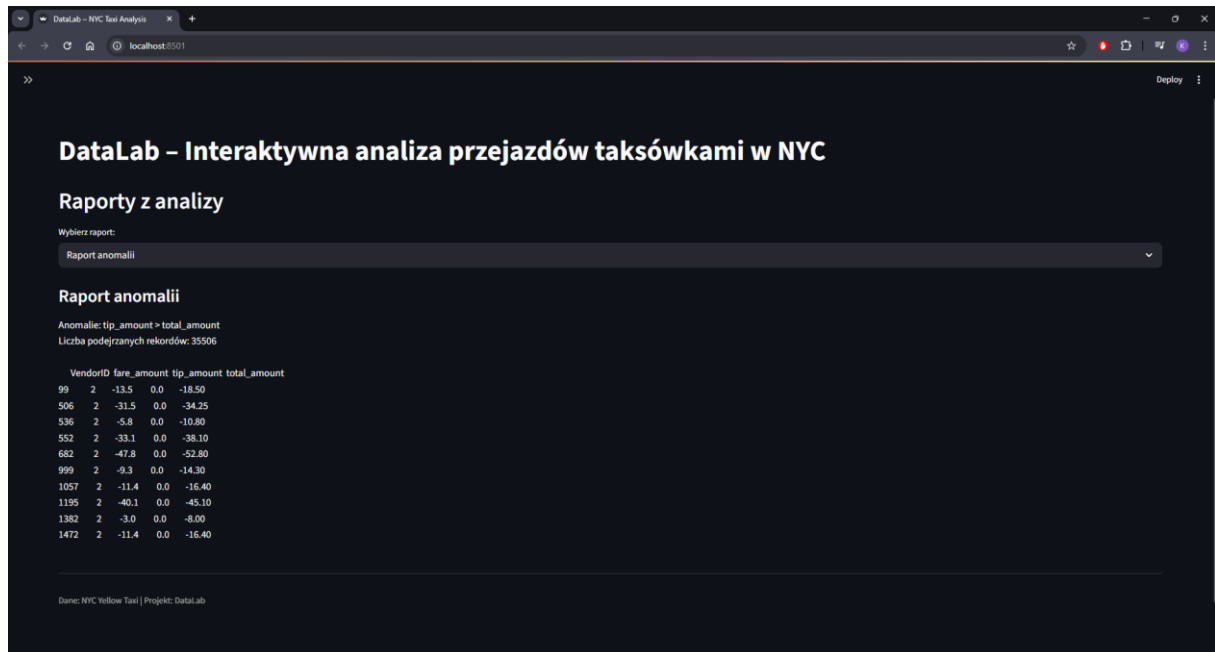
- całkowita liczba rekordów,
- średnia długość przejazdu i średni napiwek,
- łączna suma opłat oraz średnia opłata za kurs,
- całkowita liczba pasażerów,
- średnia liczba pasażerów na kurs,
- liczba długich przejazdów (>10 mil).



Rysunek 32. Widok z podsumowaniem analizy równoległej

Widok z raportami anomalii

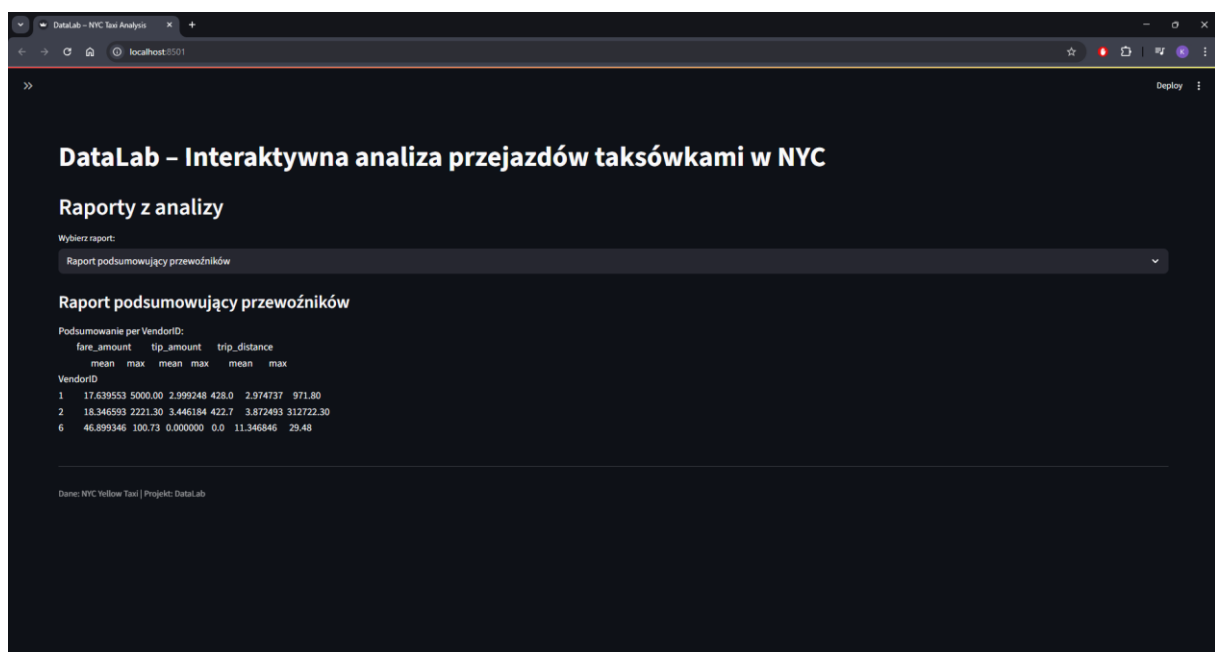
Raport identyfikuje podejrzane rekordy, w których napiwek (tip_amount) przewyższa całkowitą kwotę opłaty (total_amount), co wskazuje na potencjalny błąd w danych. Wyświetlana jest liczba takich przypadków oraz przykładowe wiersze zawierające anomalie.



Rysunek 33. Widok z raportem anomalii.

Widok z raportem podsumowującymi przewoźników

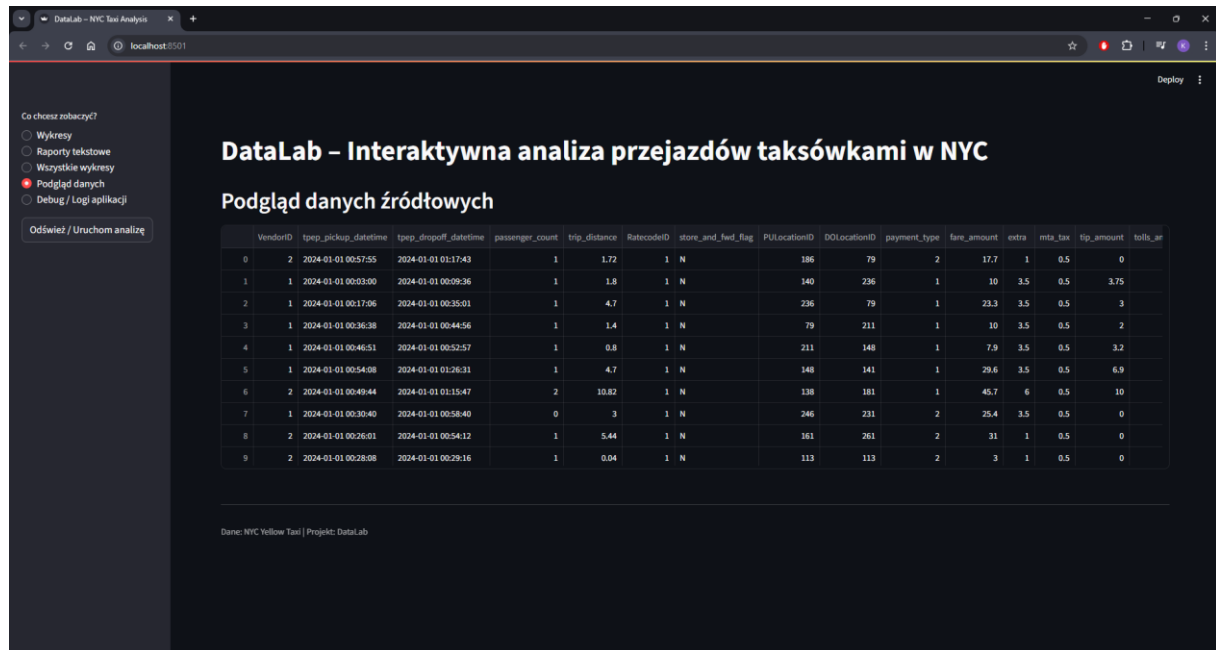
To zestawienie prezentuje statystyki zbiorcze pogrupowane według VendorID, czyli identyfikatorów firm taksówkarskich. Dla każdej z firm podano średnie i maksymalne wartości za: fare_amount, tip_amount i trip_distance.



Rysunek 34. Widok z raportem podsumowującymi przewoźników.

Widok „Podgląd danych źródłowych”

Prezentuje przykładowe rekordy z załadowanego zbioru danych o przejazdach taksówkami w NYC. Zawiera on szczegółowe informacje dotyczące m.in. daty i godziny kursu, długości trasy, liczby pasażerów, lokalizacji początkowej i końcowej, rodzaju płatności oraz opłat takich jak napiwek czy podatek.



VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocationID	DOLocationID	payment_type	fare_amount	extra	mta_tax	tip_amount	tolls_ar
0	2024-01-01 00:57:55	2024-01-01 01:17:43	1	1.72	1	N	186	79	2	17.7	1	0.5	0	
1	2024-01-01 00:03:00	2024-01-01 00:09:36	1	1.8	1	N	140	236	1	10	3.5	0.5	3.75	
2	2024-01-01 00:17:06	2024-01-01 00:35:01	1	4.7	1	N	236	79	1	23.3	3.5	0.5	3	
3	2024-01-01 00:36:38	2024-01-01 00:44:56	1	1.4	1	N	79	211	1	10	3.5	0.5	2	
4	2024-01-01 00:46:51	2024-01-01 00:52:57	1	0.8	1	N	211	148	1	7.9	3.5	0.5	3.2	
5	2024-01-01 00:54:08	2024-01-01 01:26:31	1	4.7	1	N	148	141	1	29.6	3.5	0.5	6.9	
6	2024-01-01 00:49:44	2024-01-01 01:15:47	2	10.82	1	N	138	181	1	45.7	6	0.5	10	
7	2024-01-01 00:30:40	2024-01-01 00:58:40	0	3	1	N	246	231	2	25.4	3.5	0.5	0	
8	2024-01-01 00:26:01	2024-01-01 00:54:12	1	5.44	1	N	161	261	2	31	1	0.5	0	
9	2024-01-01 00:28:08	2024-01-01 00:29:16	1	0.04	1	N	113	113	2	3	1	0.5	0	

Rysunek 35. Widok „Podgląd danych źródłowych”.

Widok „Logi aplikacji”

Przedstawia pełen zapis operacji wykonywanych przez system — w tym przebieg ładowania danych, uruchamianie analizy oraz ewentualne błędy i ostrzeżenia. Umożliwia to łatwe debugowanie i śledzenie historii działania aplikacji DataLab.

Rysunek 36. Widok „Logi aplikacji”.

34

Podsumowanie

Projekt DataLab to nowoczesna aplikacja do analizy dużych zbiorów danych o przejazdach taksówek w Nowym Jorku. System wykorzystuje przetwarzanie równoległe (multiprocessing), dzięki czemu analiza prawie 3 milionów rekordów odbywa się szybko i efektywnie, bez przeciążania pamięci. Aplikacja została stworzona w Pythonie, z użyciem bibliotek takich jak pandas, streamlit czy matplotlib, i posiada modułową strukturę ułatwiającą rozwój. Efektem działania systemu są raporty tekstowe i wizualizacje, prezentowane w intuicyjnym interfejsie webowym. DataLab spełnia założone wymagania funkcjonalne i нефункционалне, poprawnie czyści dane, wykrywa anomalie, generuje statystyki i umożliwia ich przegląd w aplikacji. Projekt jest w pełni przetestowany i gotowy do dalszej rozbudowy.