Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: _____ Informatyka techniczna (ITE)
Specjalność: _____ Systemy informatyki w medycynie (IMT)

PRACA DYPLOMOWA Inżynierska

Implementacja systemu optycznego rozpoznawania znaków oraz opracowanie środowiska eksperymentalnego opartego o samodzielnie skonstruowany zbiór danych

Krzysztof Zalewa

Opiekun pracy
Dr inż., Paweł Zyblewski

Słowa kluczowe: 3-6 słów kluczowych

WROCŁAW (2025)

Streszczenie

Dodaj streszczenie pracy w języku polskim. Staraj się uwzględnić wymienione na stronie tytułowej słowa kluczowe. Uwaga przedstawiony rekomendowany szablon dotyczy pracy dyplomowej pisanej w języku angielskim. W przeciwnym wypadku, student powinien samodzielnie zmienić nazwy „Chapter" na „Rodział" itp stosując odpowiednie pakiety systemu LaTeXoraz ustawienia w pliku *latex-settings.tex*.

Abstract

Streszczenie w języku angielskim.

Spis treści

1. Wstęp

W wyniku skanowania dokumentów powstają obrazy zawierające tekst. Jednakże większość narzędzi do przetwarzania tekstu nie jest w stanie operować na plikach graficznych. Aby rozwiązać ten problem, można zastosować algorytmy optycznego rozpoznawania znaków (Optical Character Recognition, OCR). Są to narzędzia, które umożliwiają rozpoznanie tekstu na obrazie oraz zapisanie go w formie bardziej dogodnej do przetwarzania.

Jeden z pierwszych takich algorytmów powstał w roku 1974, a jego twórcą był Ray Kurzweil. Początkowo algorytm ten miał na celu ułatwienie funkcjonowania osobom z niepełnosprawnością wzrokową. Tekst był najpierw skanowany, a następnie przetwarzany przez OCR, dzięki czemu mógł on zostać odczytany na głos. Pierwsze algorytmy rozpoznawania znaków opierały się na skomplikowanym zbiorze zasad i reguł. Wraz ze wzrostem popularności sztucznej inteligencji oraz metod uczenia maszynowego algorytmy OCR stopniowo zaczęły coraz częściej korzystać z tych technologii. Dzięki temu nie tylko poprawiła się dokładność tych algorytmów, ale także zwiększyła się liczba obsługiwanych krojów pisma oraz języków. Obecnie zdecydowana większość takich algorytmów wykorzystuje zaawansowane metody sztucznej inteligencji.

Współcześnie narzędzia te nadal służą osobom z różnymi niepełnosprawnościami. jednak zakres zastosowań algorytmów OCR znacznie się poszerzył i obejmuje inne dziedziny. Są one niezbędnym elementem w procesie digitalizacji tekstów historycznych, automatyzacji procesów biznesowych, przetwarzania dokumentów urzędowych oraz w aplikacjach mobilnych.

1.1. Cel pracy

Celem pracy było utworzenie autorskiego zbioru danych służącego do testowania wydajności algorytmów OCR. Zbiór zawiera dokumenty o zróżnicowanej charakterystyce, uwzględniając m.in. różne kroje pisma, a także modyfikacje utrudniające poprawne odczytanie treści. Zbiór ten został opracowany przy pomocy interfejsu programistycznego (API) serwisu WolneLektury.pl. Serwis ten zawiera bogaty zbiór książek, opowiadań oraz wierszy wszystkie dostępne tam pozycje znajdują się w domenie publicznej.

2. Przegląd literatury

2.1. Narzędzia OCR

Do badań wybrano następujące cztery algorytmy optycznego rozpoznawania znaków. Badania były wykonywane na najnowszych dostępnych wersjach danego algorytmu.

2.1.1. Tesseract

Tesseract OCR to najstarszy z wybranych algorytmów optycznego rozpoznawania znaków. Został on stworzony przez firmę HP w latach 1984 - 1994. Obecnie Tesseract jest jednym z najpopularniejszych algorytmów OCR w kręgach akademickich [2, 6, 10] Do badań użyto wersji 5.3.4; jest to znaczące, gdyż wcześniejsze wersje algorytmu(do wersji 3 włącznie) nie wykorzystywały sieci neuronowych. Algorytm ten działa w następujących krokach:

1. Analiza komponentów, gdzie zarys tych komponentów jest przechowywany. Takie podejście, mimo że nakłada dodatkowe koszty obliczeniowe, pozwala na łatwiejsze rozpoznawanie tekstu w odwróconych kolorach (biały tekst na czarnym tle) [11].

2. Wyszukiwanie linii w komponentach. Celem tego kroku była eliminacja potrzeby korekty przekrzywienia.

3. Podział linii na słowa.

4. Pierwsza iteracja rozpoznawania. Zaczynając od góry strony, algorytm próbuje rozpoznać każde kolejne słowo. Tokeny rozpoznane z wysokim stopniem pewności są wykorzystywane do douczenia klasyfikatora. W ten sposób z każdym kolejnym słowem celność klasyfikatora powinna rosnąć.

5. Druga iteracja rozpoznawania. Po wykonaniu pierwszej iteracji istnieje szansa, że klasyfikator uzyskałby lepsze wyniki. Więc po raz drugi algorytm próbuje rozpoznać tekst na stronie i aktualizuje słowa, które były mniej celnie rozpoznane.

2.1.2. Paddle OCR

Pomimo tego, że algorytm ten jest stosunkowo nowy (pierwsze wersje zostały wypuszczone w 2020 roku [3]). Paddle OCR jest drugim pod względem popularności algorytmem(zaraz po Tesserakcie). Początkowo pierwsze wersje algorytmu skupiały się na balansie między jakością wyniku a czasem jego otrzymania. Wraz z czasem w kolejnych wersjach udoskonalano wydajność algorytmu oraz rozszerzano jego umiejętności (Np. obsługą wielu języków, rozpoznawanie pisma ręcznego). Najnowsze wersje tego algorytmu(W pracy użyto wersji 3.2.0) składają się trzech głównych modułów. **PP-OCR** Rdzeń całego algorytmu służący do rozpoznawania

znaków na obrazie. **PP-Structure** Moduł służący do rozpoznawania ustrukturyzowanych obrazów (np. Zawierających tabele). **PP-ChatOCR** Moduł służący do ekstrakcji kluczowych informacji z obrazów przy pomocy dużego modelu językowego(z ang. Larg Language Model lub LLM). W tej pracy zastosowany został jedynie moduł PP-OCR. Działa on w następujących krokach:

1. **Preprocesowanie** - W celu uzyskania jak najlepszej jakości obrazu algorytm może usunąć niektóre zniekształcenia oraz problemy z orientacją obrazu.

2. **Wykrycie tekstu** - Algorytm tworzy mapę prawdopodobieństwa, w której każdy piksel ma przydzieloną wartość określającą jakie jest prawdopodobieństwo, że piksel ten jest częścią obszaru tekstowego. Następnie przy pomocy binaryzacji różniczkowalnej (ang. Differentiable Binarization) dynamicznie określany jest próg pomiędzy tekstem a tłem. Ostatecznie na podstawie tej mapy tworzone są wielokąty będące zarysem obszaru tekstowego.

3. **Wykrycie orientacji linii** - Wykryty tekst dzielony jest na linie. Następnie algorytm upewnia się, że wykryte linie tekstu są w prawidłowej orientacji

4. **Rozpoznanie tekstu** - Poprzez zastosowanie konwolucyjnej sieci neuronowej (z ang. Convolutional Neural Network lub CNN) wykrywane są charakterystyczne elementy tekstu jak pociągnięcia, krzywe i pętle. Elementy te podawane są do rekurencyjnej sieci neuronowej (z ang. Recurrent Neural Network lub RNN), która dzięki możliwości "zapamiętania" poprzednich elementów jest w stanie odróżnić poszczególne znaki. Na koniec koneksjonistyczna klasyfikacja czasowa (z ang. Connectionist Temporal Classification lub CTC) działa jako mechanizm wyrównywania i zwijania i znajduje najbardziej prawdopodobny napis (Np. "ccccczzzzzaaass" zostaje zmienione na "czas")

2.1.3. DocTR OCR

Algorytm ten jest skupiony na rozpoznawaniu dokumentów takich jak skany faktur, paragonów, formularzy czy listów. Stąd też nazwa Document Text Recognition, czy docTR w skrócie. Główną filozofią tego projektu jest "bezproblemowe optyczne rozpoznawanie znaków dostępne dla każdego" [8]. Algorytm ten stosuje dwuetapowe podejście do rozpoznawania tekstu:

1. **Wkrywanie tekstu** - DocTr pozwala na wykorzystanie w tym celu wielu różnych modeli. Jednakże większość z nich,podobnie jak w Tesseract i Paddle OCR, oparta jest na konwolucyjnych sieciach neuronowych. Jednakże w przeciwieństwie do tych dwóch algorytmów docTr używa piramidy cech (z ang. Feature Pyramid Network) co pozwala na odczyt tekstu w wielu różnych rozmiarach (Przydatne na przykład do odróżnienia nagłówka od adnotacji itp.). Następnie algorytm dla każdego piksela w obrazie przewiduje czy jest on w obszarze tekstowym i tworzy wielokąty wokół wykrytego tekstu.

2. **Rozpoznanie tekstu** - Podobnie jak Paddle OCR docTr najpierw rozpoznaje cechy charakterystyczne tekstu przy pomocy CNN. Jednakże do odróżnienia znaków używana

jest dwukierunkowa rekurencyjna sieć neuronowa. Sieć ta różni się od zwykłej sieci RNN tym że czyta sekwencje znaków od lewej do prawej a następnie od prawej do lewej. W niektórych przypadkach ciąg liter "cl" może być bardzo zbliżony do "d". Dlatego też zabieg ten redukuje możliwość pomylenia znaków.

2.1.4. Easy OCR

EasyOCR jest jednym z nowszych algorytmów użytych w tej pracy (Pierwsza wersja pochodzi z 2019 roku [5]). Rozwojem tego projektu zajmuje się zespół Jaided AI, specjalizujący się w wizji komputerowej i uczeniu maszynowym. Mimo stosunkowo krótkiej historii, algorytm ten stał się popularny wśród deweloperów oraz w kręgach akademickich. Projekt ten priorytetyzuje prostotę, szybkość oraz wygodę w użytkowaniu. Podobnie jak DocTR OCR algorytm ten działa w dwóch krokach:

1. **Wkrywanie tekstu** - W przeciwieństwie do pozostałych algorytmów EasyOCR, w tym kroku korzysta z modelu CRAFT (z ang. Character-Region Awareness For Text detection). Model ten tworzy dwie mapy prawdopodobieństwa. W pierwszej mapie zapisane jest prawdopodobieństwo, że dany piksel znajduje się na środku znaku. Druga mapa zawiera prawdopodobieństwo, że dany piksel jest na środku przerwy między znakami. Nakładając na siebie te dwie mapy, model jest w stanie precyzyjnie wyliczyć zarys każdego znaku, a następnie wyrysować wielokąt zawierający w sobie dane słowo.

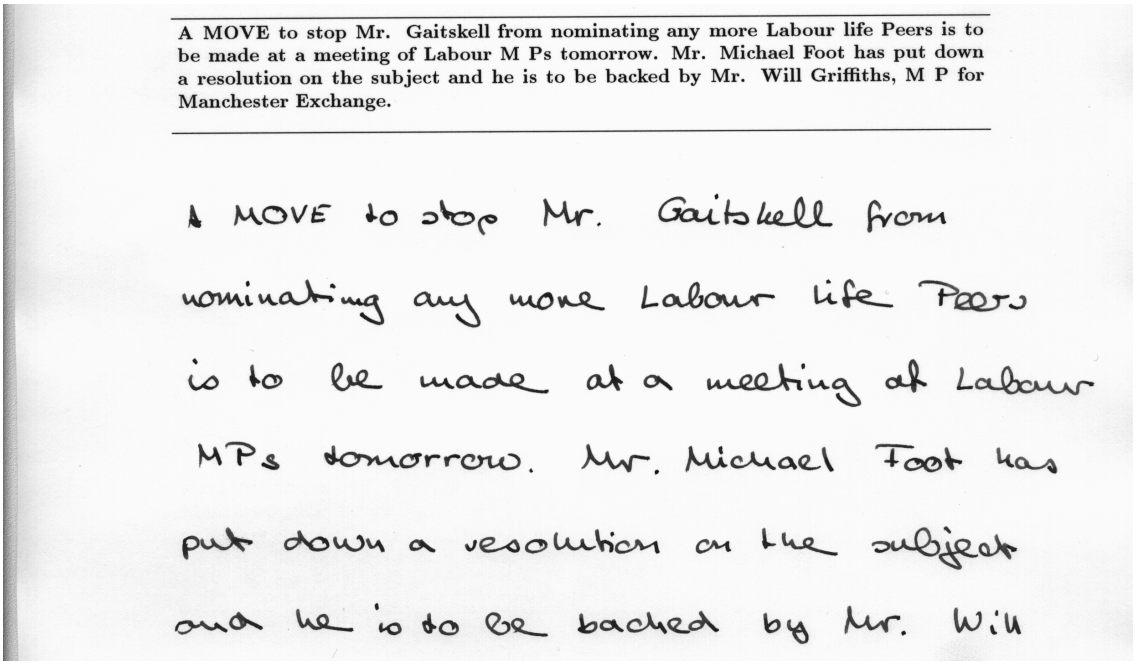
2. **Rozpoznanie tekstu** - Ten krok jest już znacznie bardziej standardowy. Przebiega podobnie jak analogiczny krok w algorytmie DocTR OCR.

2.2. Zbiory danych

Aby uzasadnić przydatność wykonanego zbioru danych wybrano zbiory IAM oraz old-books-dataset. Dla zbiorów tych zostały przeprowadzone badania [4, 6] w których wykonano testy dla algorytmu Tesseract.

2.2.1. IAM

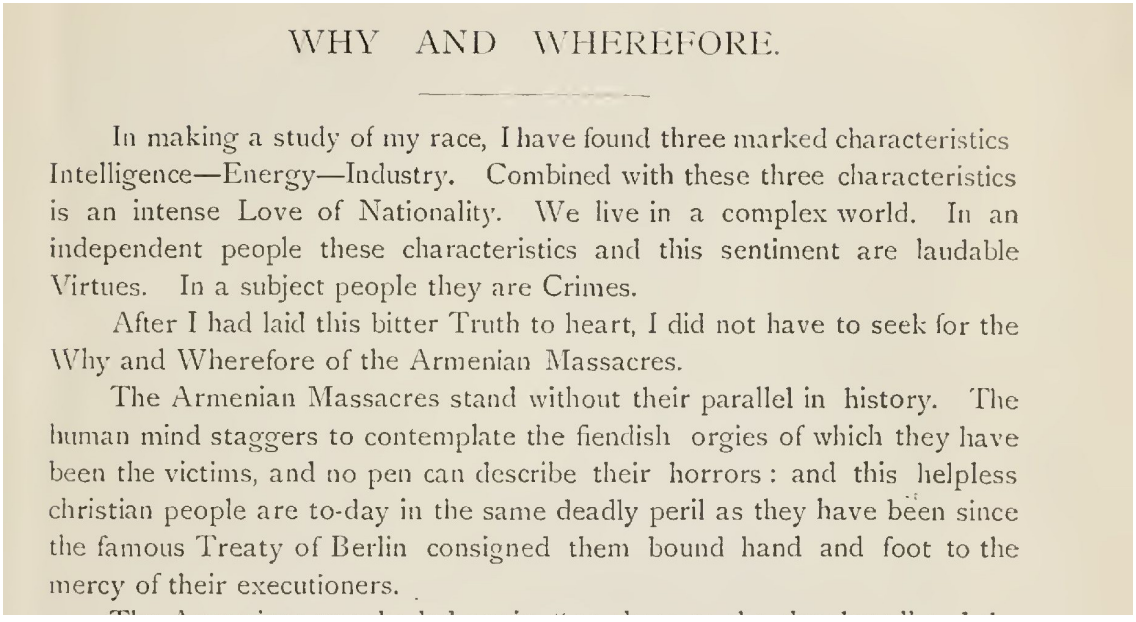
IAM to zbiór ręcznie zapisanych tekstów w języku angielskim. Wykonany przez Instytut matematyki i informatyki na Uniwersytecie Breńskim. [7] Zbiór zawiera obrazy w rozdzielczości 300 dpi zapisane w formacie PNG w 256 odcieniach szarości. Każdy podkatalog zawiera teksty zapisane przez jedną osobę.



Rysunek 2.1: Przykładowy obraz ze zbioru danych IAM

2.2.2. oldbooksdataset

Zbiór udostępniony na platformie GitHub zawierający skany książek w języku angielskim. Książki zapisane są w formacie .tiff w rozdzielczości 300 dpi oraz 500 dpi [1].



Rysunek 2.2: Przykładowy obraz ze zbioru danych old-books-dataset

2.3. Metryka CER

CER (Character Error Rate, z ang. częstotliwość błędnych znaków) to metryka, dzięki której możliwa jest ocena różnic między tekstem wytworzonym przez model OCR a tekstem rzeczywistym. W tym przypadku CER obliczane jest poprzez zsumowanie operacji(wstawień, usunięć oraz zamian znaków) potrzebnych do uzyskania tekstu rzeczywistego.

$$CER = \frac{S + D + I}{N_c}$$

Gdzie:

- S - Liczba zamian znaków (ang. Substitutions)
- D - Liczba usunięć znaków (ang. Deletions)
- I - Liczba wstawień znaków (ang. Inserts)
- N_c - Liczba znaków w tekście (ang. Number of characters)

Na przykład:
Tekst oryginalny: Życiem wschód, śmiercia południe;
Tekst wygenerowany przez model: Zyciem wschod, siercia poudniex;

Aby przekształcić tekst wygenerowany do tekstu oryginalnego, należy wykonać 4 zamiany (Brakujące znaki polskie), 1 wstawienie (Brakujące 'm' w tekście wygenerowanym) oraz 1 usunięcie ('x' nie występuje w tekście oryginalnym). Więc $CER = 6/28 = 0.2141 \approx 21.4\%$

2.4. Szum Perlina

Ten typ generacji pseudo-losowego szumu został zaproponowany przez Kena Perlina w 1985 [9]. W przeciwieństwie do czysto losowych danych, które charakteryzują się ostrymi, nieciągłymi zmianami, szum Perlina generuje wartości zmieniające się w sposób płynny i gradualny, co lepiej odwzorowuje procesy obserwowane w środowisku naturalnym.

Algorytm, choć przedstawiony na Rysunku () na przykładzie jednowymiarowym, może być rozszerzony do dwóch lub trzech wymiarów. Wielowymiarowe warianty szumu Perlina znajdują zastosowanie w wielu środowiskach dla przykładu w grafice komputerowej w szczególności podczas proceduralnej generacji tekstur. Z szumu Perlina kożysta się też w produkcji filmów kożystających z grafiki generowanej komputerowo (CGI) oraz tworzeniu środowisk w grach komputerowych(np. w grze Minecraft proces generacja terenu kożysta z tego algorytmu).

Implementacja szumu Perlina przebiega w trzech głównych etapach.

Definicja przestrzeni

W pierwszym etapie generowana jest regularna siatka w przestrzeni n-wymiarowej, gdzie każdemu węzłowi siatki przypisywany jest losowy wektor jednostkowy, określający gradient w danym punkcie przestrzeni.

Obliczenie iloczynu skalarnego

Dla dowolnego punktu P w przestrzeni identyfikowana jest komórka siatki, w której się on znajduje. Następnie dla każdego wierzchołka V_i tej komórki obliczany jest wektor przesunięcia

$\vec{d_i} = P - V_i$ oraz iloczyn skalarny między tym wektorem a wektorem gradientu przypisanym do wierzchołka:

$$s_i = \vec{d_i} \cdot \vec{g_i}$$

Interpolacja

Ostateczna wartość szumu w punkcie P obliczana jest poprzez interpolację wartości s_i uzyskanych dla wszystkich wierzchołków komórki. Interpolacja przeprowadzana jest przy użyciu funkcji wygładzającej, która zapewnia ciągłość pochodnych na granicach komórek.

W wykorzystanej implementacji, która pochodzi z biblioteki `noise` , podczas generacji szumu Perlina można podać następujące parametry.

1. **x,y** - Kordynaty generowanego punktu.
2. **Liczba oktaw (Octaves)** - Określa ilość nakładanych warstw szumu. Większa liczba oktaw powoduje dodanie detali o wyższych częstotliwościach, co skutkuje bardziej złożonym i szczegółowym wzorem wyjściowym.
3. **Trwałość (Persistence)** - Współczynnik określający amplitudę kolejnych oktaw. Przyjmuje wartości z zakresu [0, 1]. Wysokie wartości powodują zachowanie większej ilości szczegółów, generując ostrzejsze wzory. Natomiast niskie wartości skutkują bardziej wygładzonymi rezultatami.
4. **Lacunarity** - Parametr określający współczynnik zmiany częstotliwości między kolejnymi oktawami. Wartość musi być większa od 1. Wyższe wartości powodują szybszy wzrost częstotliwości w kolejnych oktawach, co skutkuje większym zróżnicowaniem skali detali.
5. **Repeatx** - Parametr określający powtarzalność szumu w osi X. Jeżeli parametr ten równa się szerokości obrazu, wzór nie będzie się powtarzał.
6. **Repeaty** - Parametr określający powtarzalność szumu w osi Y. Jeżeli parametr ten równa się wysokości obrazu, wzór nie będzie się powtarzał.

2.5. Testy statystyczne

2.5.1. Test Wilcoxona

Test Wilcoxona jest dobrym zamiennikiem testu T-Studenta w przypadku, w którym rozkład różnic między parami obserwacji nie jest normalny. W tym teście zakładana jest słabsza hipoteza. Mianowicie rozkład różnic jest symetryczny wokół jednej wartości. Celem tego testu jest udowodnienie, że ta wartość znacznie różni się od zera. **Założenia testu:**

1. Rozkład różnic jest symetryczny wokół mediany
2. Skala pomiarowa jest przynajmniej porządkowa
3. Różnice między parami są niezależne

Procedura obliczeniowa:

1. Obliczenie różnic d_i dla każdej pary obserwacji
2. Usunięcie różnic równych zero
3. Uporządkowanie wartości bezwzględnych różnic od najmniejszej do największej i przypisanie im rang
4. Sumowanie rang dla różnic dodatnich (W^+) i ujemnych (W^-)
5. Statystyka testowa W to mniejsza z sum: $W = \min(W^+, W^-)$

3. Modyfikatore tekstu

3.1. Kroje pisma

Jednym z problemów, na które często napotykają się algorytmy OCR, jest pismo odręczne (oraz kroje pisma je imitujące). Ten typ pisma jest bardzo zróżnicowany. Każdy człowiek ma swój własny charakter pisma. Zapisane znaki różnią się kształtem, rozmiarem czy nachyleniem. Ponadto zdarza się, że w tym samym zdaniu te same znaki mogą się znacznie różnić. Wszystkie te czynniki sprawiają, że wiele algorytmów OCR ma mniejszą skuteczność w rozpoznawaniu pisma odręcznego niż pisma maszynowego.

W tej pracy wybrano cztery kroje pisma. Pierwsze dwa mają przypominać pismo odręczne, a pozostałe dwa są popularnymi przykładami standardowych krojów.

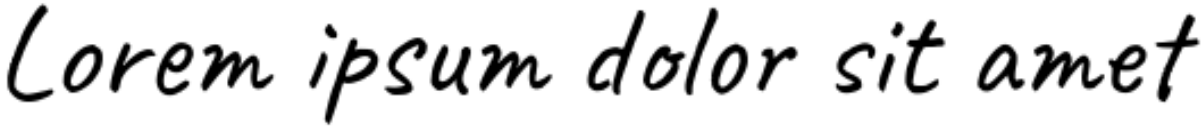
Allura Regular

Ten krój ma symulować pismo odręczne. Jednakże, jak widać na poniższej próbce (Rys 3.1), znaki w tym kroju są ze sobą połączone. Może to utrudniać rozpoznanie poszczególnych znaków, co wpłynie na wyniki algorytmów.

Rysunek 3.1: Próbka tekstu z użyciem kroju Allura ("Lorem ipsum dolor sit amet")

Caveat

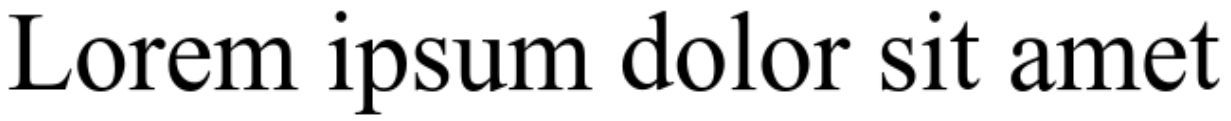
Podobnie jak poprzedni krój, Caveat ma przypominać pismo odręczne. Jednakże w przeciwieństwie do poprzednika, w tym kroju znaki są wyraźnie rozdzielone (Rys 3.2).



Rysunek 3.2: Próbka tekstu z użyciem kroju Caveat ("Lorem ipsum dolor sit amet")

Times New Roman

W przeciwieństwie do poprzednich krojów, Times New Roman nie jest stylizowany na pismo odręczne (Rys 3.3). Jest to jeden z bardziej popularnych krojów szeryfowych. Szeryfy to poprzeczne lub ukośne zakończenia głównych pociągnięć znaków.



Rysunek 3.3: Próbka tekstu z użyciem kroju Times New Roman ("Lorem ipsum dolor sit amet")

Ubuntu

Kroje szeryfowe są popularne w publikacjach papierowych, jednak podczas czytania tekstu elektronicznego mogą być męczące dla oczu. Dlatego obecnie coraz częściej stosowane są kroje bez-szeryfowe (ang. sans-serif). Ubuntu jest przykładem takiego kroju (Rys 3.4).

Rysunek 3.4
Próbka tekstu z użyciem kroju Ubuntu ("Lorem ipsum dolor sit amet")

3.2. Zniekształcenia obrazu

Kolejnym problemem jest jakość dostarczonego obrazu. W rzeczywistości rzadko spotykane są przypadki, gdy skanowany tekst jest równomiernie oświetlony, nie ma na nim cieni lub papier, na którym znajduje się tekst, jest niepogięty. Takie zniekształcenia mogą utrudnić rozdzielenie tekstu od tła, przez co skuteczność algorytmu zmaleje.

W tej pracy wybrano trzy rodzaje takich zniekształceń obrazu. Są to:

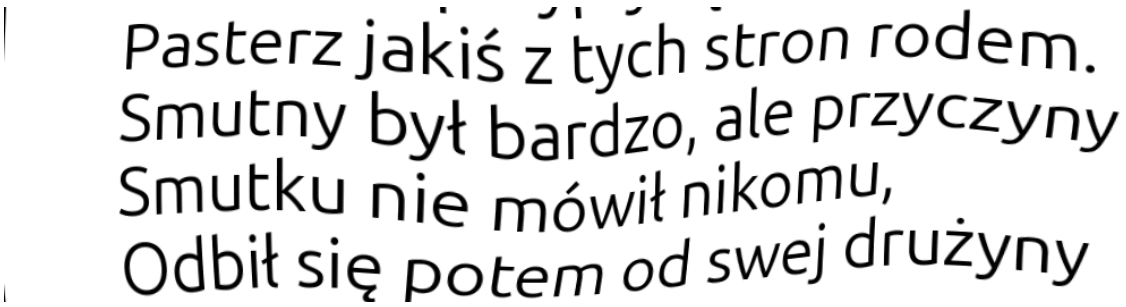
Przesunięcie perspektywy

Obrazy poddawane są losowej deformacji przestrzennej poprzez przekształcenie projektowe. Dla każdego obrazu generowane są cztery współczynniki deformacji z zakresu [0.01, 0.2], które określają przemieszczenie narożników. Na podstawie punktów źródłowych i docelowych obliczana jest macierz transformacji, a następnie aplikowana na obraz wejściowy przy użyciu funkcji `cv2.warpPerspective`.

Kod źródłowy 3.1: Fragment autorskiej klasy ImageGenerator odpowiedzialnej za generację przekrzywionych obrazów

```
1  import os
2  import random
3
4  import cv2
5  import numpy as np
6
7
8  class ImageGenerator:
9      def tilt_img(self, input_dir: str, file_name: str):
10         os.makedirs(self.TILT_DIR, exist_ok=True)
11         input_path = os.path.join(input_dir, file_name)
```

```
12          output_path = os.path.join(self.TILT_DIR, file_name)
13
14          original_image = cv2.imread(input_path)
15          height, width = original_image.shape[:2]
16          src_points = np.float32([[0, 0], [width, 0], [0, height], [
                width, height]])
17
18          random.seed(self.seed)
19          tilt_tl: float = random.uniform(0.01, 0.2)
20          tilt_tr: float = random.uniform(0.01, 0.2)
21          tilt_bl: float = random.uniform(0.01, 0.2)
22          tilt_br: float = random.uniform(0.01, 0.2)
23          dst_points = np.float32(
24              [
25                  [width * tilt_tl, 0],
26                  [width * (1 - tilt_tr), 0],
27                  [0, height * (1 - tilt_bl)],
28                  [width * (1 - tilt_br), height * (1 - tilt_br)],
29              ]
30          )
31
32          matrix = cv2.getPerspectiveTransform(src_points, dst_points)
33
34          tilted_image = cv2.warpPerspective(original_image, matrix, (
                width, height))
35          cv2.imwrite(output_path, tilted_image)
```



Rysunek 3.5: Fragment tekstu po nałożeniu efektu zmiętej kartki

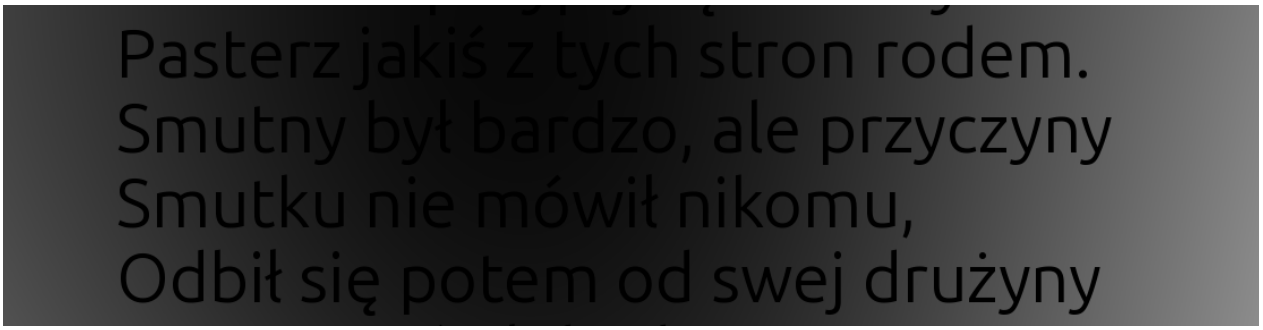
Zacienienie

Na oryginalny obraz nałożono losowo wygenerowane cienie, które zostały stworzone przy pomocy metody `pnoise2` z biblioteki `noise`, służącej do generowania losowego szumu Perlina. Dobór wartości parametrów tej funkcji przeprowadzono metodą eksperymentalną, poprzez wielokrotne iterację z różnymi konfiguracjami, aż do osiągnięcia pożądanego efektu w postaci delikatnego, lokalnego zacienienia.

Kod źródłowy 3.2: Fragment autorskiej klasy ImageGenerator odpowiedzialnej za generację zacienionych obrazów

```
import os

import cv2
import numpy as np
from noise import pnoise2


class ImageGenerator:
    def shadow_img(self, input_dir: str, file_name: str):
        scale = 5000.0
        octaves = 2
        persistence = 0.5
        lacunarity = 2.0

        os.makedirs(self.SHADOW_DIR, exist_ok=True)

        input_path = os.path.join(input_dir, file_name)
        output_path = os.path.join(self.SHADOW_DIR, file_name)

        shadow = np.zeros((self.A4_HEIGHT, self.A4_WIDTH))

        for i in range(self.A4_HEIGHT):
            for j in range(self.A4_WIDTH):
                shadow[i][j] = pnoise2(
                    i / scale,
                    j / scale,
                    octaves=octaves,
                    persistence=persistence,
                    lacunarity=lacunarity,
                    repeatx=self.A4_WIDTH,
                    repeaty=self.A4_HEIGHT,
                    base=self.seed,
                )

        normalized_shadow = (shadow - shadow.min()) / (shadow.max()
            - shadow.min())

        shadow_dark = normalized_shadow + 0.05
        original_image = cv2.imread(input_path)

        shadow_rgb = cv2.cvtColor(shadow_dark.astype(np.float32),
            cv2.COLOR_GRAY2BGR)
        image_with_shadow = original_image.astype(np.float32) *
            shadow_rgb
        image_with_shadow = np.clip(image_with_shadow, 0, 255).
            astype(np.uint8)

        cv2.imwrite(output_path, image_with_shadow)
```

Wygenerowany w ten sposób szum został poddany normalizacji. Następnie otrzymaną mapę cieni nałożono na oryginalny obraz.



Rysunek 3.6: Fragment tekstu po zacienieniu

Efekt zmiętej kartki

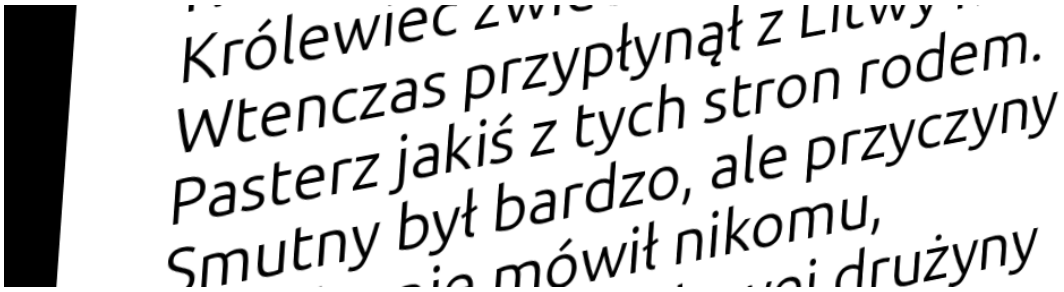
Oryginalny obraz został poddany deformacji, obliczonej na podstawie losowo wygenerowanego szumu Perlina. W celu otrzymania tego efektu, podobnie jak w przypadku obrazów zacienionych, skorzystano z metody `pnoise2` biblioteki `noise`. Także procedura wyboru parametrów była zbliżona do poprzednika. Jednakże w tym przypadku pożądanym efektem były podłużne, nieregularne formy imitujące naturalne zgięcia i deformacje materiału papieru, takie jak występują w fizycznych, użytkowanych dokumentach.

Kod źródłowy 3.3: Fragment autorskiej klasy ImageGenerator odpowiedzialnej za generację zmiętych obrazów

```
import os

import cv2
import numpy as np
from noise import pnoise2


class ImageGenerator:
    def wrinkle_img(self, input_dir: str, file_name: str):
        scale = 500.0
        octaves = 3
        persistence = 0.5
        lacunarity = 2.0

        os.makedirs(self.WRINKLE_DIR, exist_ok=True)

        input_path = os.path.join(input_dir, file_name)
        output_path = os.path.join(self.WRINKLE_DIR, file_name)

        wrinkles = np.zeros((self.A4_HEIGHT, self.A4_WIDTH))

        for i in range(self.A4_HEIGHT):
            for j in range(self.A4_WIDTH):
                wrinkles[i][j] = pnoise2(
                    i / scale,
                    j / scale,
                    octaves=octaves,
                    persistence=persistence,
                    lacunarity=lacunarity,
```

```
30                      repeatx=self.A4_WIDTH,
31                      repeaty=self.A4_HEIGHT,
32                      base=self.seed,
33                  )
34
35          normalized_wrinkles = (wrinkles - wrinkles.min()) / (
36              wrinkles.max() - wrinkles.min()
37          )
38
39          original_image = cv2.imread(input_path)
40
41          scale_intensity = 50
42          disp_x = (normalized_wrinkles - 0.5) * scale_intensity
43          disp_y = (normalized_wrinkles - 0.5) * scale_intensity
44
45          x, y = np.meshgrid(np.arange(self.A4_WIDTH), np.arange(self.
                A4_HEIGHT))
46
47          map_x = (x + disp_x).astype(np.float32)
48          map_y = (y + disp_y).astype(np.float32)
49
50          wrinkled_image = cv2.remap(original_image, map_x, map_y, cv2
                .INTER_LINEAR)
51          cv2.imwrite(output_path, wrinkled_image)
```

Po wygenerowaniu otrzymany szum został znormalizowany w celu uzyskania mapy przesunięć. Następnie otrzymaną w ten sposób mapę nakładano na oryginalny obraz przy pomocy wbudowanych funkcji biblioteki `cv2`.



Rysunek 3.7: Fragment tekstu po deformacji przestrzennej

4. Akwizycja danych

4.1. Interfejs programistyczny WolneLektury.pl

Dane do zbioru zostały pozyskane z serwisu WolneLektury.pl. Dostęp do tych danych możliwy jest poprzez publiczny interfejs programistyczny (Application Programming Interface – API), dostępny pod adresem bazowym: https://wolnelektury.pl/api/.

Architektura API opiera się na zasadach REST (Representational State Transfer), oferując strukturalny dostęp do zasobów poprzez logiczne endpointy. Serwis WolneLektury.pl udostępnia wiele punktów dostępu. Jednak te najważniejsze dla tej pracy to:

- /api/books/ – Wszystkie utwory
- /api/authors/ – Lista autorów

Domyślnie dane otrzymywane w wyniku zapytania są serializowane w formacie JSON, możliwa jest też zmiana na format XML (poprzez dodanie parametru ?format=xml do zapytania). Jednakże w tej pracy wykorzystano domyślny format.

Mechanizm filtrowania i precyzyjnego wyszukiwania rekordów realizowany jest poprzez rozbudowę ścieżki URL o odpowiednie parametry. Na przykład, zapytanie pod adres /api/authors/adam-mickiewicz/ zwraca informacje o autorze "adam-mickiewicz".


```
{
    "name": "Adam Mickiewicz",
    "url": "https://wolnelektury.pl/katalog/autor/adam-mickiewicz/",
    "sort_key": "mickiewicz adam",
    "description": "<dl><dt>Ur.</dt><dd> 24 grudnia 1798 r. w Zaosiu koł
    "description_pl": "<dl><dt>Ur.</dt><dd> 24 grudnia 1798 r. w Zaosiu
    "plural": "",
    "genre_epoch_specific": false,
    "adjective_feminine_singular": "",
    "adjective_nonmasculine_plural": "",
    "genitive": "Adama Mickiewicza",
    "collective_noun": ""
}
```


Rysunek 4.1: Wynik zapytania /api/authors/adam-mickiewicz/

Najważniejszą funkcjonalnością tego API z punktu widzenia tej pracy jest możliwość łączenia endpointów w łańcuchy zapytań. Dzięki temu możliwe jest dokładne definiowanie podzbiorów danych. Dla ilustracji, zapytanie /api/authors/adam-mickiewicz/books/ zwraca metadane dla książek, których autorem jest Adam Mickiewicz.

```
[
    {
        "kind": "Liryka",
        "full_sort_key": "mickiewicz adam$ballady i romanse$291",
        "title": "Ballady i romanse",
        "url": "https://wolnelektury.pl/katalog/lektura/ballady-i-romanse/",
        "cover_color": "#db4b16",
        "author": "Adam Mickiewicz",
        "cover": "book/cover/ballady-i-romanse.jpg",
        "epoch": "Romantyzm",
        "href": "https://wolnelektury.pl/api/books/ballady-i-romanse/",
        "has_audio": true,
        "genre": "Ballada",
        "simple_thumb": "https://wolnelektury.pl/media/book/cover_api_thumb/ballady-i-romanse_BZcruYT.jpg",
        "slug": "ballady-i-romanse",
        "cover_thumb": "book/cover_thumb/ballady-i-romanse_dr22alD.jpg",
        "liked": null
    },
```


Rysunek 4.2: Fragment wyniku zapytania /api/authors/adam-mickiewicz/books/

Wynik takiego zapytania zawiera listę skróconych opisów pozycji. Aby otrzymać dokładniejsze dane na temat danej książki, należy wykorzystać atrybut "href", który zawiera link do szczegółowego opisu pozycji.


```
    "parent": null,
    "preview": false,
    "epub": "https://wolnelektury.pl/media/book/epub/ballady-i-romanse.epub",
    "mobi": "https://wolnelektury.pl/media/book/mobi/ballady-i-romanse.mobi",
    "pdf": "https://wolnelektury.pl/media/book/pdf/ballady-i-romanse.pdf",
    "html": "https://wolnelektury.pl/media/book/html/ballady-i-romanse.html",
    "txt": "https://wolnelektury.pl/media/book/txt/ballady-i-romanse.txt",
    "fb2": "https://wolnelektury.pl/media/book/fb2/ballady-i-romanse.fb2",
    "xml": "https://wolnelektury.pl/media/book/xml/ballady-i-romanse.xml",
    "media": [],
    "audio_length": "",
    "cover_color": "#db4b16",
    "simple_cover": "https://wolnelektury.pl/media/book/cover_simple/ballady-i-romanse_P6innIl.jpg",
    "cover_thumb": "https://wolnelektury.pl/media/cache/65/dd/65dd19f39a51dd3c10dc6c3170322e21.jpg",
    "cover": "https://wolnelektury.pl/media/book/cover/ballady-i-romanse.jpg",
    "simple_thumb": "https://wolnelektury.pl/media/book/cover_api_thumb/ballady-i-romanse_BZcruYT.jpg",
    "isbn_pdf": "ISBN 978-83-288-0533-0",
    "isbn_epub": "ISBN 978-83-288-3529-0",
    "isbn_mobi": "ISBN 978-83-288-4615-9"
}
```


Rysunek 4.3: Widok szczegółowy dla pozycji ballady i romanse

W szczegółowych danych na temat książki najważniejsze, z punktu widzenia tej pracy, były pola "epub", "html","txt" oraz "xml". Zawierają one linki do pobrania zawartości książki w odpowiednim formacie.

Kod źródłowy 4.1: Fragment kodu odpowiedzialny za pobieranie metadanych pozycji.

```
1 import json
2 import os
3
4 import requests
5
6
7 def get_book_list() -> None | list[Book]:
8     os.makedirs(CACHE_DIR, exist_ok=True)
9     path = os.path.join(CACHE_DIR, f"{AUTHOR}.json")
```

```
10
11      try:
12          print(f"Getting: {url}")
13          response = requests.get(url, timeout=TIMEOUT_MAX)
14
15          if response.status_code != 200:
16              print("Error: ", response.status_code)
17              return None
18
19          books_data = response.json()
20          data = response.json()
21          with open(path, "w", encoding="utf-8") as file:
22              json.dump(data, file, indent=2, ensure_ascii=False)
23
24          return [Book(**book) for book in books_data]
25
26      except requests.exceptions.RequestException as e:
27          print("Error: ", e)
28          return None
```

Podczas tworzenia zbioru, stała AUTHOR zawierała napis "adam-mickiewicz". Ten fragment kodu jest odpowiedzialny za pobranie listy książek dla wybranego autora. Book to autorska klasa służąca do deserializacji danych w formacie JSON.

Kod źródłowy 4.2: Fragment kodu odpowiedzialny za pobranie podanej pozycji

```
1  import os
2
3
4  def scrape(epub_url: str, name: str):
5      print("Downloading url: ", epub_url)
6      try:
7          response = requests.get(epub_url, timeout=TIMEOUT_MAX)
8
9          if response.status_code != 200:
10             print("Error: ", response.status_code)
11
12         path = os.path.join(RAW_DIR, f"{name}.epub")
13
14         total_size = int(response.headers.get("content-length", 0))
15
16         with open(path, "wb") as file:
17             if total_size == 0:
18                 file.write(response.content)
19             else:
20                 downloaded = 0
21                 for chunk in response.iter_content(chunk_size=8192):
22                     if chunk:
23                         file.write(chunk)
24                         downloaded += len(chunk)
25
26     except requests.exceptions.RequestException as e:
27         print("Error: ", e)
```

4.2. Ekstrakcja tekstu

Początkowo rozważono wykorzystanie formatów tekstowych, takich jak TXT, HTML oraz XML, które ze względu na swoją prostą, czytelną dla maszyn strukturę wydawały się obiecującym rozwiązaniem problemu ekstrakcji treści. Proces pozyskania danych z tych formatów sprowadzałby się wówczas do bezpośredniego odczytu i parsowania plików.

Okazało się jednak, że zasoby udostępniane w tych formatach przez serwis WolneLektury.pl są często niekompletne w kontekście całych zbiorów dzieł. Dla zilustrowania tego problemu można posłużyć się przykładem cyklu „Ballady i romanse" Adama Mickiewicza, który składa się z czternastu osobnych utworów. Tymczasem pliki w formatach TXT, HTML i XML dostępne do pobrania dla tego cyklu zawierają wyłącznie tekst pojedynczego wiersza „To lubię", pomijając pozostałe części dzieła.

Wobec powyższych ograniczeń, ostatecznym wyborem formatu źródłowego do budowy korpusu wybrany został format EPUB. Format ten można traktować jako format binarny, jednakże w rzeczywistości jest to archiwum zawierające ustrukturyzowane pliki (X)HTML, metadane oraz zasoby multimedialne. Taka struktura znacząco komplikuje proces ekstrakcji tekstu w porównaniu z prostymi formatami tekstowymi. Aby uprościć proces wydobywania tekstu z plików EPUB, wykorzystano biblioteki: `EbookLib` do niskopoziomowego parsowania struktury archiwum EPUB oraz `Beautiful Soup` do przetwarzania i ekstrakcji czystego tekstu z sekcji HTML zawartych w książce.

Kod źródłowy 4.3: Fragment autorskiej klasy TextExtractor odpowiedzialnej za ekstrakcję tekstu

```
import ebooklib
from bs4 import BeautifulSoup
from ebooklib import epub


class TextExtractor:
    def load_file(self, input_path: str):
        book = epub.read_epub(input_path)
        self.book_name = input_path.split("/")[-1]
        items = list(book.get_items_of_type(ebooklib.ITEM_DOCUMENT))

        chapter_items = []
        for item in items:
            if "part" in item.get_name().lower():
                chapter_items.append(item)

        return chapter_items

    def parse_file(self, chapter):
        soup = BeautifulSoup(chapter.get_body_content(), "html.
            parser")
        target_elements = soup.find_all(
            ["h2", "div"], class_=lambda x: x != "stanza" and x != "
                stanza-spacer"
        )

        filtered_elements = []
```

```
26          for element in target_elements:
27              if element.name == "h2":
28                  filtered_elements.append(element)
29              elif element.name == "div" and "verse" in element.get("
                    class", []):
30                  filtered_elements.append(element)
31
32          text_elements = []
33          for element in filtered_elements:
34              element_text = element.get_text().strip()
35              if element_text:
36                  text_elements.append(element_text)
37
38          text = "\n".join(text_elements)
39          return text
```

Plik źródłowy zostaje otwarty przy pomocy funkcji z biblioteki `EbookLib`. Otrzymane w ten sposób to pliki składowe archiwum jakim jest format EPUB. Pliki te składają się z wielu elementów, jednakże na potrzeby tej pracy wykorzystano dane z elementów typu "h2" i "div". Zawierają one odpowiednio nagłówki oraz główny tekst z publikacji.

4.3. Przygotowanie obrazów

Obrazy wyjściowe generowane były w rozdzielczości odpowiadającej standardowemu formatowi A4 (2480 × 3508 pikseli), przy założeniu rozdzielczości 300 DPI (dots per inch). W przyjętej konfiguracji tekst na obrazach miał rozmiar 60 pikseli, a marginesy 100 pikseli. W celu zapewnienia pełnej widoczności tekstu oraz uniknięcia jego przycięcia, podzielono go na fragmenty zawierające x linii tekstu. Wartość x wyliczono za pomocą poniższego wzoru:

$$x = \left\lfloor \frac{H - 2m}{h} \right\rfloor$$

Gdzie:

- H - Wysokość obrazu (3508 px)
- m - Margines (100 px)
- r - Rozmiar tekstu (60 px + odstęp między wierszowy)

Do generacji obrazów wykorzystano bibliotekę `Pillow`, jest to standardowa biblioteka do generacji obrazów w języku Python. Zaletą wykorzystania tej biblioteki jest prostota zmiany kroju pisma użytego do generacji. `Pillow` udostępnia klasę FreeTypeFont, która pozwala na załadowanie pliku w formacie ttf. Następnie załadowany krój pisma można przekazać jako argument podczas rysowania tekstu.

Kod źródłowy 4.4: Fragment autorskiej klasy ImageGenerator odpowiedzialnej za generację obrazów

```
from PIL import Image, ImageDraw


class ImageGenerator:
    def text_to_img(self, output_path, text: str):
        output_path = output_path.replace(".txt", ".png")
        lines = text.split("\n")

        img = Image.new("RGB", (self.A4_WIDTH, self.A4_HEIGHT),
            color="white")
        draw = ImageDraw.Draw(img)

        y_position = self.margin

        for line in lines:
            if y_position + self.line_height < self.A4_HEIGHT - self
                .margin:
                draw.text((self.margin, y_position), line, fill="
                    black", font=self.font)
                y_position += self.line_height

        img.save(output_path)
```

5. Aspekt badawczy

Pytania badawcze

1. W jakim stopniu opracowany autorski zbiór danych wykazuje porównywalność z uznanymi benchmarkami OCR pod względem trudności i zdolności do weryfikacji podstawowej skuteczności algorytmów?

2. Który z badanych algorytmów OCR osiąga statystycznie istotnie lepszą efektywność dla poszczególnych krojów pisma?

3. Który z badanych algorytmów OCR osiąga statystycznie istotnie lepszą efektywność dla poszczególnych kategorii zniekształceń wizualnych?

Protokuł eksperymentalny

W celu zapewnienia rzetelności procedury badawczej, otrzymane wyniki poddano analizie statystycznej z wykorzystaniem testu parnego T-Studenta oraz nieparametrycznego testu Wilcoxona. Dla obu testów przyjęto $\alpha = 0.05$. Wszystkie testy powtórzono 5-krotnie.

5.1. Scenariusze eksperymentów

Eksperyment 1 Porównanie z innymi zbiorami

Celem tego eksperymentu było uzasadnienie, że autorski zbiór danych jest porównywalny z innymi benchmarkami algorytmów OCR(zbiory IAM oraz old-books-dataset). Eksperyment został przeprowadzony na czystej wersji (bez zniekształceń oraz kroju pisma Ubuntu) autorskiego zbioru danych. Eksperyment przeprowadzono w celu odpowiedzi na pytanie badawcze nr. 1. Wyniki zostały porównane na podstawie współczynnika błędów znakowych (CER).

Eksperyment 2 Wpływ zniekształceń obrazu

Trzeci eksperyment ma na celu zbadanie wpływu zastosowanych zniekształceń obrazu (opisane w rozdziale 3) na uzyskane wyniki. Dla każdego rodzaju zniekształcenia został wygenerowany nowy fragment autorskiego zbioru danych. Eksperyment zostanie przeprowadzony na tych fragmentach. Eksperyment przeprowadzono w celu odpowiedzi na pytanie badawcze nr. 2. Wyniki zostały porównane na podstawie metryki współczynnika błędów znakowych (CER).

Eksperyment 3 Wpływ krojów pisma

Natomiast celem czwartego eksperymentu było zbadanie wpływu wybranych krojów pisma (opisane w rozdziale 3) na otrzymane wyniki. Dla każdego kroju pisma został wygenerowany nowy fragment autorskiego zbioru danych. Eksperyment zostanie przeprowadzony na tych fragmentach. Eksperyment przeprowadzono w celu odpowiedzi na pytanie badawcze nr. 3. Wyniki zostały porównane na podstawie metryki współczynnika błędów znakowych (CER).

6. Wyniki

6.1. Eksperyment 1 Porównanie z innymi zbiorami

Tabela 6.1: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Ubuntu

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.0972	0.0000	2.9361	0.1956
Easyocr	0.1795	0.0000	1.8550	0.0313
Paddle	0.0692	0.0000	1.2890	0.0102
Tesseract	0.0761	0.0000	1.3744	0.0139

Tabela 6.2: Szczegółowe wyniki dla każdego algorytmu

	DocTr		Easyocr		Paddle		Tesseract	
Powtórzenie	CER	Czas[s]	CER	Czas[s]	CER	Czas[s]	CER	Czas[s]
1	0.0972	2.7302	0.1795	1.8428	0.0692	1.2990	0.0761	1.3611
2	0.0972	2.7207	0.1795	1.8059	0.0692	1.3013	0.0761	1.3582
3	0.0972	3.0820	0.1795	1.8829	0.0692	1.2821	0.0761	1.3843
4	0.0972	3.1245	0.1795	1.8673	0.0692	1.2816	0.0761	1.3794
5	0.0972	3.0231	0.1795	1.8758	0.0692	1.2809	0.0761	1.3888

Tabela 6.3: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Ubuntu

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.0972	0.0153	2.7302	0.3807
Easyocr	0.1795	0.0461	1.8428	0.2465
Paddle	0.0692	0.0118	1.2990	0.0903
Tesseract	0.0761	0.0117	1.3611	0.1767

Jako pierwszy, wykonano test dla wersji zbioru autorskiego, korzystającej z kroju Ubuntu. Tabela 6.1 zawiera uśrednione wyniki pomiędzy pięcioma powtórzeniami. Jednakże jak widać odchylenie standardowe dla CER, kluczowej metryki której użyto do oceny wydajności algorytmów, jest w każdym przypadku równe 0. Po dokładniejszej analizie (Tabela 6.2)

ustalono że taki wynik jest poprawny. Dlatego też kolejne testy były analizowane na podstawie pierwszego powtórzenia.

Tabela 6.4: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Ubuntu

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.0972	0.0153	2.7302	0.3807
Easyocr	0.1795	0.0461	1.8428	0.2465
Paddle	0.0692	0.0118	1.2990	0.0903
Tesseract	0.0761	0.0117	1.3611	0.1767

Tabela 6.5: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Ubuntu

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.0972	0.0153	2.7302	0.3807
Easyocr	0.1795	0.0461	1.8428	0.2465
Paddle	0.0692	0.0118	1.2990	0.0903
Tesseract	0.0761	0.0117	1.3611	0.1767

6.2. Eksperyment 2 Wpływ zniekształceń obrazu

Wyniki działania algorytmów dostępne są w tabeli 6.5. Z tego powodu nie zostały ponownie umieszczone w tej sekcji.

Tabela 6.6: Test Wilcoxona dla wyników testu na zbiorze korzystającym z kroju Ubuntu

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	1856.5000	0.0000	-0.3397
Paddle vs DocTR	0.0000	0.0000	-0.8606
Paddle vs EasyOCR	0.0000	0.0000	-0.9961
Tesseract vs DocTR	15.5000	0.0000	-0.7423
Tesseract vs EasyOCR	0.0000	0.0000	-0.9933
DocTR vs EasyOCR	0.0000	0.0000	-0.9584

Tabela 6.7: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Allura

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.2108	0.0245	2.5894	0.3481
Easyocr	0.4348	0.0568	1.9503	0.2775
Paddle	0.0775	0.0113	1.2232	0.0756
Tesseract	0.3199	0.0584	1.2542	0.1543

Tabela 6.8: Test Wilcoxona dla wyników testu na zbiorze korzystającym z kroju Allura

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	0.0000	0.0000	-1.0000
Paddle vs DocTR	0.0000	0.0000	-1.0000
Paddle vs EasyOCR	0.0000	0.0000	-1.0000
Tesseract vs DocTR	45.0000	0.0000	0.9660
Tesseract vs EasyOCR	1147.0000	0.0000	-0.8428
DocTR vs EasyOCR	1.0000	0.0000	-0.9999

Tabela 6.9: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Caveat

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.1613	0.0274	3.2743	0.7884
Easyocr	0.3286	0.0588	2.1997	0.3805
Paddle	0.1496	0.0246	1.1880	0.0703
Tesseract	0.1707	0.0252	1.2508	0.1403

Tabela 6.10: Test Wilcoxona dla wyników testu na zbiorze korzystającym z kroju Caveat

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	942.5000	0.0000	-0.5044
Paddle vs DocTR	1436.5000	0.0000	-0.2782
Paddle vs EasyOCR	0.0000	0.0000	-0.9982
Tesseract vs DocTR	4138.5000	0.0000	0.2480
Tesseract vs EasyOCR	0.0000	0.0000	-0.9912
DocTR vs EasyOCR	0.0000	0.0000	-0.9933

Warto zaznaczyć, że CER jest miarą błędu więc malejące wartości są lepsze. Z tego też powodu ujemna wartość Delta np. w pierwszym wierszu tabeli 6.6 oznacza, że algorytm Paddle usyskał wynik lepszy od algorytmu Tesseract w stopniu statystycznie znaczącym.

W przeprowadzonym eksperymencie bardzo wyróżnia się algorytm Paddle. We wszystkich

Tabela 6.11: Wyniki algorytmów OCR dla zbioru korzystającego z kroju Times New Roman

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.0897	0.0140	2.7539	0.3784
Easyocr	0.1886	0.0460	1.8497	0.3642
Paddle	0.0672	0.0128	1.2525	0.1413
Tesseract	0.0787	0.0118	1.3371	0.1711

Tabela 6.12: Test Wilcoxona dla wyników testu na zbiorze korzystającym z kroju Times New Roman

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	1461.0000	0.0000	-0.5260
Paddle vs DocTR	70.0000	0.0000	-0.7737
Paddle vs EasyOCR	0.0000	0.0000	-0.9996
Tesseract vs DocTR	712.5000	0.0000	-0.4702
Tesseract vs EasyOCR	0.0000	0.0000	-0.9992
DocTR vs EasyOCR	0.0000	0.0000	-0.9915

testch dominuje nad konkurencją. Szczególnie widoczne jest to dla badania przerowadzonego na zbiorze wykonanym przy użyciu kroju Allura. Gdzie algorytm ten uzyskał średnie CER na poziomie 0.0775 następny w kolejności algorytm (DocTr) uzyskał wynik na poziomie 0.2108.

6.3. Eksperyment 3 Wpływ krojów pisma

Tabela 6.13: Wyniki algorytmów OCR dla zacienionych obrazów

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.1011	0.0172	2.6866	0.3602
Easyocr	0.1831	0.0477	1.8281	0.2999
Paddle	0.7344	0.3241	1.2343	0.2159
Tesseract	0.7219	0.3406	1.1655	0.2651

W eksperymencie 2 algorytm Paddle miały znaczącą przewagę. Jednakże w tym eksperymencie ta przewaga nie była tak znacząca. W przypadku testu dla obrazów zacienionych (tabela 6.13) wyniki tego algorytmu były najgorsze. Co ciekawe algorytm DocTr w tym samym teście otrzymał najlepszy wynik.

Ciekawym wynikiem jest też test dla tekstów przekrzywionych. W tym teście wszystkie algorytme otrzymały wynik CER bilski 0.25 (lub wyższy).

Tabela 6.14: Test Wilcoxona dla wyników testu na zbiorze zacienionych obrazów

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	6597.0000	0.6836	-0.0086
Paddle vs DocTR	51.0000	0.0000	0.9483
Paddle vs EasyOCR	531.0000	0.0000	0.8037
Tesseract vs DocTR	58.0000	0.0000	0.9374
Tesseract vs EasyOCR	562.0000	0.0000	0.7977
DocTR vs EasyOCR	1.0000	0.0000	-0.9477

Tabela 6.15: Wyniki algorytmów OCR dla przechylonych obrazów

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.2420	0.1578	2.7522	0.6201
Easyocr	0.3016	0.1566	1.8904	0.2970
Paddle	0.2501	0.3644	1.1734	0.1518
Tesseract	0.3364	0.3262	1.4357	0.1920

Tabela 6.16: Test Wilcoxona dla wyników testu na zbiorze przekrzywionych obrazów

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	7632.5000	0.0000	-0.4583
Paddle vs DocTR	9571.5000	0.0002	-0.5525
Paddle vs EasyOCR	8966.0000	0.0000	-0.6000
Tesseract vs DocTR	9745.0000	0.0005	-0.0348
Tesseract vs EasyOCR	13230.0000	0.8688	-0.2114
DocTR vs EasyOCR	4957.0000	0.0000	-0.2874

Tabela 6.17: Wyniki algorytmów OCR dla zmiętych obrazów

	Średnie CER	Odchylenie standardowe dla CER	Średni czas[s]	Odchylenie standardowe dla czasu
DocTr	0.0996	0.0187	2.7331	0.3663
Easyocr	0.2812	0.0584	1.8696	0.2535
Paddle	0.2948	0.2081	1.1853	0.0603
Tesseract	0.1261	0.0427	1.5180	0.1989

Tabela 6.18: Test Wilcoxona dla wyników testu na zbiorze zmiętych obrazów

	W	Wartość p	Delta Cliffa
Paddle vs Tesseract	2059.0000	0.0000	0.5303
Paddle vs DocTR	1144.5000	0.0000	0.6876
Paddle vs EasyOCR	12453.5000	0.3529	-0.1165
Tesseract vs DocTR	3218.5000	0.0000	0.5247
Tesseract vs EasyOCR	120.0000	0.0000	-0.9642
DocTR vs EasyOCR	0.0000	0.0000	-0.9984

7. Podsumowanie

Ponawiając pytania badawcze:

1. W jakim stopniu opracowany autorski zbiór danych wykazuje porównywalność z uznanymi benchmarkami OCR pod względem trudności i zdolności do weryfikacji podstawowej skuteczności algorytmów?

2. Który z badanych algorytmów OCR osiąga statystycznie istotnie lepszą efektywność dla poszczególnych krojów pisma?

3. Który z badanych algorytmów OCR osiąga statystycznie istotnie lepszą efektywność dla poszczególnych kategorii zniekształceń wizualnych?

Pytanie 1
Pytanie 2

Dla dokumentów korzystających z wybranych krojów pisma (Allura, Caveat, Ubuntu oraz Times New Roman) najlepszą efektywność osiąga algorytm Paddle. Szczególnie dobrze widać to w przypadku w którym skorzystano z kroju Allura (tabela 6.7), gdzie algorytm ten otrzymał średnie CER na poziomie 0.0775. Jednoznacznie dominując nad konkurencją, drugi w kolejności był algorytm DocTR z wynikiem 0.2108.

Obserwowana znaczna różnica między otrzymanymi wynikami była wynikiem oczekiwanym. Algorytm Paddle OCR jako jednyny z wybranych umożliwiał wykożystanie akceleracji przy pomocy procesora graficznego(GPU). Ta funkcjonalność zapewniła mu dostęp do znacznie większej mocy obliczeniowej, co bezpośrednio przełożyło się na krótszy czas przetwarzania przy zachowaniu wysokiej dokładności rozpoznawania.

Wyniki algorytmu EasyOCR były niepokojące w niemal wszystkich przypadkach były one istotnie wyższe od pozostałych badanych rozwiązań. Może to wskazywać na błąd w implementacji.

Pytanie 3

Dla dokumentów poddanych zniekształceniom wizualnym (zacienienie, przechylenie perspektywy oraz efekt zmiętej kartki) najlepszą efektywność osiąga algorytm DocTR. W przeciwieństwie do poprzedniego eksperymentu ta dominacja nie była tak jednoznaczna. W przypadku obrazów zacienionch DocTR otrzymał wyniki istotnie niższe od pozostałych(tabela 6.13). W pozostałych przypadkach różnica między algorytmem DocTR a pozostałymi nie były aż tak widoczne (np. w tabeli 6.15 różnica między DocTR a Paddle to jedynie 0.01).

Obserwowana wyższość algorytmu DocTR nad pozostałymi była, w tym przypadku, wynikiem oczekiwanym. DocTR jest algotymem trenowanym do rozpoznawania obrazów bliższych rzeczywistym. Dla przykładu zbiór treningowy zawiera obrazy przedstawiające zmięte paragony lub inne dokumenty.

Wyniki dla algorytmów Paddle oraz Tesseract otrzymane w wyniku badania na zacienionych obrazach były istotnie niższe od wszystkich pozostałych. Dane otrzymane w wyniku pierwszego

powtórzenia porównano z pozostałymi czteroma powtórzeniami. W wyniku tego działania potwierdzono, że dane otrzymane w kolejnych powtórzeniach się nie różnią. Przez co można wykluczyć możliwość tego, że otrzymane wyniki są anomalią. Może to jednak oznaczać, że efektywność tych algorytmów jest znacznie bardziej zależna od kontrastu między tekstem a tłem. Jednakże potwierdzenie tej teorii wymagać będzie dalszych badań.

W tym eksperymencie algorytm EasyOCR otrzymał wyniki znacznie bliższe pozostałym. Co zmniejsza ryzyko błędnej implementacji.

Warte odontowania są także średnie czasy analizy obrazu. Algorytm Paddle OCR uzyskiwał istotnie niższe wyniki w wielu kategoriach podczas gdy czas wykonania pozostawał najniższy. Jednakże wynik ten jest zależny od zastosowania przyspieszenia przy pomocy procesora graficznego (wykonano jeden test w którym użyto Paddle bez przyspieszenia ,jednakże średni czas wykonania przekraczał 70 sekund). Drugim pod względem szybkości wykonywania był algorytm Tesseract, otrzymywał on wyniki bliższe algorytmowi Paddle podczas gdy czas pozostawał zbliżony do drugiego algortmu. Jednakże w przeciwieństwie do algorytmu Paddle Tesseract nie potrzebuje przyspieszenia GPU, więc może być bardziej użyteczny w środowiskach w których procesor graficzny jest nie dostępny.

Bibliografia

[1] P. Barcha. Old books dataset. https://github.com/PedroBarcha/old-books-dataset, 2024. Accessed: 2024.

[2] A. Chowdhury, A. A. Sami, S. M. P. Mamun, S. Absar, F. Biswas, and M. Kohinoor. Performance analysis of tesseract and easyocr for bangla optical character recognition on the novel bangla crosshair dataset. In *Proceedings of the International Conference on Computer and Communication Systems*, 01 2025.

[3] C. Cui, T. Sun, M. Lin, T. Gao, Y. Zhang, J. Liu, X. Wang, Z. Zhang, C. Zhou, H. Liu, Y. Zhang, W. Lv, K. Huang, Y. Zhang, J. Zhang, J. Zhang, Y. Liu, D. Yu, and Y. Ma. Paddleocr 3.0 technical report. *arXiv preprint*, 2025.

[4] T. Hegghammer. Ocr with tesseract, amazon textract, and google document ai: a benchmarking experiment. *Journal of Computational Social Science*, 5:861–882, 2022.

[5] JaidedAI. Ready-to-use ocr. https://github.com/JaidedAI/EasyOCR, 2024.

[6] Y. Li. Synergizing optical character recognition: A comparative analysis and integration of tesseract, keras, paddle, and azure ocr. *University of Sydney Technical Reports*, 45:45–60, 2023.

[7] U. Marti and H. Bunke. The iam-database: An english sentence database for off-line handwriting recognition. *International Journal on Document Analysis and Recognition*, 5:39–46, 2002.

[8] Mindee. doctr: Document text recognition. https://github.com/mindee/doctr, 2021.

[9] K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, 1985.

[10] V. S and S. A. Performance comparison of ocr tools. *International Journal of UbiComp*, 6(3):19–30, July 2015.

[11] R. Smith. An overview of the tesseract ocr engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR)*, volume 2, pages 629–633, 2007.