Получил Задание

Задача глобальная: необходимо разработать детектор элементов документа по картинке
Подзадачи:

1. Создать генератор документов docx или pdf для получения обучающей выборки
2. Сформировать из документов набор данных - картинки и координатную разметку к ним
3. Обучить модель на этих данных

Перешёл к решению первой задачи

1. Создать генератор документов docx или pdf для получения обучающей выборки

Изучил требования к генератору

Документы должны содержать:
- заголовки (на 2 и более размера крупнее основного текста, жирные и/или курсив, чаще с выравниванием по середине, возможно с нумерацией)
- абзацы текста (с красной строки и без, шрифт от 8 до 16, различное выравнивание)
- таблицы (с сеткой и без, левые-правые границы могут быть без сетки, цветные и черно-белые, различное выравнивание самой таблицы и ячеек внутри)
- рисунки (любые картинки, графики, встроенные элементы)
- подписи к рисункам (часто на 1-2 размера меньше, реже до и чаще после картинки, начинаются с "Рис. <номер>", "Рисунок <номер> -", "Рисунок <номер>", часто выравнивание по центру)
- подписи к таблицам (такого же или на 1-2 размера меньше основного текста, перед таблицей, начинаются с "Табл. <номер>", "Таблица <номер> -", "Таблица. < без номера>")
- нумерованные списки (у них обычно меньше межстрочный интервал и есть отступ)
- маркированные списки (у них обычно меньше межстрочный интервал и есть отступ)
- верхний и нижний колонтитулы (будет плюсом)
- сноски (будет плюсом)
- формулы (будет плюсом, если получится)
2. Должны быть примеры с несколькими колонками текста
3. Размер выборки 10 тыс. изображений.
4. Сохранить файлы в папку "docx".
5.
Стал выбор в каком стиле выполнять первый пункт функциональном или ООП

Решил выполнять 1 пункт в функциональном стиле тк не планируется какое-либо осложнение задачи и в данный момент в сжатые сроки. Из недостатков можно выделить некоторое количество повторения кода в нескольких функциях, но учитывая объём это не критично. Возможно при наличии времени стоит стоит будет реализовать в ооп.

Для создания док файлов рассмотрел несколько вариантов библиотек, но остановился на предложенном задании python-docx

После определения функционального стиля и библиотеки сразу решил составить 100% нужных функция для создания:

1. Документа
2. Заголовка
3. Абзац текста
4. Таблицы
5. Рисунка
6. Подписи к рисунку
7. Подписи к таблицам
8. Нумерованные списки
9. Маркированные списки
10. Верхний колонтитула
11. Нижнего колонтитула
12. Сноски
13. Формулы

Далее для каждой функции решил внести требования из задания и свои требования
Для функции создания документа

```
def create_document()
```

Первым делом нужно создать объект типа Document() из библиотеки python-docx

```
doc = Document()
```

Должны генерироваться документы в альбомной и портретной ориентации
Тк документы разной ориентации генерироваться одной функции значит нам нужно случайно делать выбор Соотвественно подключаем библиотеку рандом и в дальнейших случаях случайного выбора для выполнения условий и в дальнейшем увеличивая разнообразия для обучающей выборки будем применять её

Также из условий следует что у документа должен быть выбран основной размер текста от 8 до 16 Pt. Решил при создании документа сразу выбирать основной текст и как параметр передавать его функциям внутри.

```
base_font_size = random.choice(range(8, 17))
```

После определения Размера основного текста передадим его в функции где он необходим

Также для добавления других элементов текста но не по одному шаблону а с разными вариациями решил функции вызвать в случайном порядке случайным образом.

После добавления элемнтов будет необходимо сохранить файл тк для выборки в 10000 изображений потребуется много файлов логичнее всего все файлы сохранить в отдельную папку. Чтобы предотвратить ошибки Делаю проверка есть ли эта директория и если нет то создаю её. Файлы будем называть doc_индекс индекс от 0 и по количеству наших файлов.

После первоначальной реализации генератора документов в виде одного большого скрипта я столкнулся с рядом проблем:

- **Читаемость кода**: Большой скрипт становится трудным для понимания и сопровождения, особенно когда он содержит множество функций и логики.
- **Повторение кода**: В большом скрипте легко упустить повторяющиеся фрагменты кода, что затрудняет их обновление и отладку.
- **Трудности в расширении функциональности**: Добавление новых функций требует больше времени и может привести к появлению ошибок, если код не структурирован должным образом.

Чтобы решить эти проблемы, я решил разбить один большой скрипт на несколько отдельных файлов (модулей), каждый из которых отвечает за определенную функциональность. Это соответствует принципам модульности и разделения ответственности.

Структура файлов после разделения:

- `main.py` : Точка входа в программу. Содержит основную логику запуска генерации документов.
- `config.py` : Содержит настройки и конфигурационные данные, такие как частоты появления различных блоков и используемые языки.
- `document.py` : Основной модуль для создания документа. Содержит функции для добавления элементов в документ.

- `generators.py`: Содержит функции для генерации содержимого, такого как текст, таблицы, изображения и формулы.
- `styles.py`: Отвечает за применение стилей к элементам документа.
- `utils.py`: Вспомогательные функции, которые используются в разных частях проекта.
- `annotations.py`: Модуль для сохранения аннотаций и метаданных, связанных с документами.
- `classify.py`: Содержит функции для классификации блоков в документе.

от annotations.py в дальнейшем отказался в связи с невозможностью получения координат генерируемых модулей. Пригодилось только для тестирования точности определения base_font_size для 2 этапа.

Детально разберём файлы из последней версии генератора

Файл `config.py`

Этот файл содержит настройки частоты появления различных блоков в документе и определение используемых языков для генерации текста.

```
from collections import OrderedDict

# Настройки частоты появления блоков для стандартного документа
block_frequencies = {
    'paragraph': 40,
    'table': 15,
    'picture': 15,
    'numbered_list': 10,
    'marked_list': 10,
    'formula': 10
}

# Языки для генерации данных
locales = OrderedDict([('en-US', 1), ('ru-RU', 2)])
```

- **Частота появления блоков:** Я решил установить вероятности появления различных типов блоков, чтобы документы были разнообразными и содержали достаточное количество каждого типа элементов.
- **Выбор языков:** Для генерации текста использую английский и русский языки, что помогает увеличить разнообразие данных и сделать модель более универсальной.

В дальнейшем возможно расширение что можно задать разные настройки частоты появления блоков для генерации различных типов документов. К примеру инструкций, финансовых отчетов, технической документации и т. д. .

Также возможно расширение в плане языков документов.

`document.py`

```
from docx import Document
from docx.shared import Pt, RGBColor, Inches
from docx.enum.text import WD_ALIGN_PARAGRAPH
from docx.enum.table import WD_TABLE_ALIGNMENT
from docx.enum.section import WD_ORIENTATION
from docx.oxml.ns import qn
from docx.oxml import OxmlElement
import os
import random

from generators import generate_realistic_data, generate_formula_image, fake
from styles import apply_paragraph_format, apply_run_format,
apply_title_format
from utils import roman_number, add_page_number, insert_footnote,
to_superscript
from config import block_frequencies

def create_document(index):
    doc = Document()

    # Установка ориентации страницы
    section = doc.sections[0]
    if random.choice([True, False]):
        section.orientation = WD_ORIENTATION.LANDSCAPE
        section.page_width, section.page_height = section.page_height,
section.page_width
    else:
        section.orientation = WD_ORIENTATION.PORTRAIT

    # Решение о многоколоночности документа
    is_multicolumn = random.choices([True, False], weights=[25, 75], k=1)[0]
    if is_multicolumn:
        main_table, num_columns = add_multicolumn_table(doc)
    else:
        main_table, num_columns = None, 1

    base_font_size = random.choice(range(8, 17))
```

```
    blocks = []
    footnotes = []

    # Добавление заголовка
    add_title(doc, base_font_size)
    blocks.append("title")

    # Добавление колонтитулов
    add_header(doc, base_font_size)
    blocks.append("header")
    add_footer(doc)
    blocks.append("footer")

    # Генерация содержимого документа
    num_blocks = random.randint(10, 20)
    block_types = list(block_frequencies.keys())
    weights = list(block_frequencies.values())
    current_column = 0

    for _ in range(num_blocks):
        block = random.choices(block_types, weights=weights, k=1)[0]
        if is_multicolumn:
            cell = main_table.rows[0].cells[current_column]
        else:
            cell = None

        if block == 'paragraph':
            add_paragraph(doc, base_font_size, footnotes, cell)
        elif block == 'table':
            add_table(doc, base_font_size, cell)
            if cell is not None:
                paragraph = cell.add_paragraph()
            else:
                paragraph = doc.add_paragraph()
        elif block == 'picture':
            add_picture(doc, base_font_size, cell)
        elif block == 'numbered_list':
            add_numbered_list(doc, base_font_size, cell)
        elif block == 'marked_list':
            add_marked_list(doc, base_font_size, cell)
        elif block == 'formula':
            add_formula(doc, base_font_size, cell)
        blocks.append(block)

        if is_multicolumn:
            current_column = (current_column + 1) % num_columns
```

```
    add_footnotes_section(doc, footnotes, base_font_size)

    # Сохранение документа
    output_dir = "docx"
    os.makedirs(output_dir, exist_ok=True)
    filename = os.path.join(output_dir, f"doc_{index}.docx")
    doc.save(filename)

    return filename, base_font_size
```

- **Многоколоночность:** Для реализации нескольких колонок в документе я использовал таблицу без границ, где каждая ячейка представляет собой колонку. Это позволило легко распределять содержимое по колонкам.
- **Разнообразие элементов:** При генерации содержимого документа я случайным образом выбираю тип блока и добавляю соответствующий элемент с различными вариациями в форматировании и содержимом.

Функции добавления элементов

Добавление заголовка

```
def add_title(doc, base_font_size):
    title_number = random.choice([True, False])
    title = doc.add_paragraph()
    run = title.add_run()

    title_text = fake.sentence(nb_words=random.randint(1, 100))
    if title_number:
        title_text = f"{random.randint(0, 101)}. {title_text}"

    run.text = title_text

    # Применение стилей заголовка
    apply_title_format(title, base_font_size)
```

- **Вариативность заголовков:** Заголовки могут содержать нумерацию или быть без неё, быть жирными, курсивными и иметь разное выравнивание.
- **Размер шрифта:** Заголовок делается на 2-5 пунктов больше основного размера шрифта.

Добавление параграфа

```
def add_paragraph(doc, base_font_size, footnotes, cell=None):
    if cell is not None:
        paragraph = cell.add_paragraph()
    else:
        paragraph = doc.add_paragraph()

    # Генерация текста абзаца
    text = fake.text(max_nb_chars=random.randint(50, 1000))
    insert_footnote(paragraph, text, base_font_size, footnotes,
footnote_per_page=5)

    # Форматирование абзаца
    apply_paragraph_format(paragraph, base_font_size)

    if paragraph.runs:
        run = paragraph.runs[0]
        apply_run_format(run, base_font_size)
```

- **Случайное форматирование:** Абзацы могут иметь или не иметь отступ первой строки, различные выравнивания и шрифты.
- **Добавление сносок:** С некоторой вероятностью в абзац вставляется сноска. Работа с XML-структурой предполагала создание специальных элементов `<w:footnoteReference>` и соответствующих записей в секции `<w:footnotes>`. Это требовало глубокого понимания внутреннего устройства DOCX-файла и могло привести к непредсказуемым результатам, если что-то будет сделано неправильно. Кроме того, `python-docx` не предоставляет прямых методов для работы со сносками. Чтобы не усложнять код, я решил реализовать сноски более простым способом — посредством имитации сносок. Это было достигнуто путем вставки небольшого числового символа (верхнего индекса) непосредственно в текст абзаца, а затем добавления соответствующего текста сноски в конце документа или в специальный

Добавление таблицы

```
def add_table(doc, base_font_size, cell=None):
    if cell is not None:
        add_table_caption(doc, base_font_size, cell)
        table = cell.add_table(rows=random.randint(3, 10),
cols=random.randint(3, 7))
    else:
```

```
        add_table_caption(doc, base_font_size)
        table = doc.add_table(rows=random.randint(3, 10),
cols=random.randint(3, 7))

    remove_table_borders(table)
    # ...
```

- **Размер:** Количество колонок и столбцов выбирается случайно.
- **Стили таблицы:** Таблицы могут быть с границами и без, с различным выравниванием и стилями.
- **Заполнение данных:** Я использовал функцию `generate_realistic_data` для заполнения ячеек таблицы реалистичными данными использовав библиотеку mimesis.

Добавление рисунка

```
def add_picture(doc, base_font_size, cell=None):
    location_signature_after = random.choices([True, False], weights=[0.7,
0.3], k=1)[0]
    image_folder = "Dataset_images"

    if os.path.isdir(image_folder):
        images = [f for f in os.listdir(image_folder) if f.endswith(('.png',
'.jpg', '.jpeg'))]
        if images:
            image_path = os.path.join(image_folder, random.choice(images))

            if cell is not None:
                paragraph = cell.add_paragraph()
            else:
                paragraph = doc.add_paragraph()
            paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER
            run = paragraph.add_run()

            if not location_signature_after:
                add_picture_caption(doc, base_font_size, cell)

            # Добавление изображения с корректировкой размера
            # ...

            if location_signature_after:
                add_picture_caption(doc, base_font_size, cell)
```

- **Подписи к рисункам:** Подпись может располагаться до или после изображения, чаще после.
- **Масштабирование:** Для многоколоночных документов изображение масштабируется под ширину колонки.

Файл `generators.py`

```
from faker import Faker
from mimesis import Generic
from mimesis.enums import Locale
import random
import sympy as sp
import matplotlib.pyplot as plt
from io import BytesIO

fake = Faker(list(locales.keys()))
generic = Generic(Locale.RU)

def generate_realistic_data():
    # Генерация данных для таблиц
    data_type = random.choice(['name', 'date', 'number', 'address',
'company'])
    # ...

def generate_formula_image():
    # Генерация изображения формулы с помощью SymPy и Matplotlib
    # ...
```

Содержит функции для генерации содержимого, такого как реалистичные данные для таблиц и изображения формул.

- **Генерация данных для таблиц:** Используя библиотеки `faker` и `mimesis`, я генерирую различные типы данных для заполнения таблиц.
- **Генерация формул:** С помощью `sympy` и `matplotlib` создаю изображения случайных математических выражений для вставки в документ.

Файл `styles.py`

```
from docx.shared import Pt
from docx.enum.text import WD_ALIGN_PARAGRAPH
import random
```

```
def apply_paragraph_format(paragraph, base_font_size):
    # Применяет форматирование к параграфу
    # ...

def apply_run_format(run, base_font_size):
    # Применяет форматирование к тексту внутри параграфа
    # ...

def apply_title_format(paragraph, base_font_size):
    # Применяет форматирование к заголовку
    # ...
```

- **Разделение стилей от логики:** Вынесение стилей в отдельный файл упрощает управление форматированием и повышает читаемость кода.
- **Вариативность стилей:** В функциях применяются случайные выборы шрифтов, выравниваний и других параметров для увеличения разнообразия.

Файл `utils.py`

```
def roman_number(n):
    # Преобразует число в римскую цифру
    # ...

def to_superscript(number):
    # Преобразует число в верхний индекс
    # ...

def add_page_number(paragraph):
    # Добавляет номер страницы в нижний колонтитул
    # ...

def insert_footnote(paragraph, text, base_font_size, footnotes,
footnote_per_page=5):
    # Вставляет сноску в текст
    # ...
```

- **Упрощение кода:** Вынесение часто используемых функций в отдельный файл помогает избежать дублирования и упрощает поддержку кода.

- **Форматирование сносок и номеров страниц:** Реализовал функции для корректного добавления сносок и номеров страниц в документ.

Файл `main.py`

```
from document import create_document

def main():
    for index in range(10000):
        filename, base_font_size = create_document(index)
        print(f"Сгенерирован документ: {filename}")

    print("Генерация документов завершена.")

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

- **Масштабирование генерации:** С помощью цикла генерирую необходимое количество документов для формирования выборки.
- **Логирование:** Вывожу сообщения о прогрессе генерации для удобства отслеживания процесса.

Вывод

В результате я разработал генератор документов, который соответствует заданным требованиям и способен создавать разнообразные документы для последующей обработки и обучения модели. Код структурирован по модулям для удобства поддержки и расширения функциональности. При необходимости генератор можно доработать, добавив новые элементы или улучшив существующие.

Дальнейшие шаги:

1. **Сформировать набор данных:** Используя сгенерированные документы, необходимо конвертировать их в PDF и извлечь из них изображения страниц и координатную разметку элементов.
2. **Обработать PDF-документы:** Написать скрипты для обработки PDF-файлов, выделения блоков текста, изображений и других элементов с их координатами.
3. **Обучить модель:** На основе полученных данных обучить модель детекции элементов документа по изображению.

Этап 2: Формирование набора данных - картинки и координатная разметка

Получение задачи

После успешной реализации генератора документов следующим шагом было формирование обучающей выборки, включающей изображения документов и соответствующую координатную разметку элементов. Эта задача включала в себя несколько ключевых подзадач:

1. **Конвертация документов в изображения**: Преобразование сгенерированных DOCX-файлов в изображения высокого качества.
2. **Извлечение координат элементов**: Определение расположения и размеров различных элементов (заголовков, абзацев, таблиц и т.д.) на изображении.
3. **Создание аннотаций**: Формирование файлов аннотаций в заданном формате JSON.

Задачи

Для выполнения этих подзадач потребовалось выбрать подходящие инструменты и библиотеки для конвертации документов в изображения, а также разработать механизм точного извлечения координат элементов. Особое внимание уделялось созданию аннотаций в строго определённой структуре JSON, которая отличается от стандартных форматов, таких как COCO или Pascal VOC.

Решения

1. Конвертация DOCX в изображения

Рассмотренные варианты:

- **LibreOffice в режиме командной строки**: Преобразование DOCX в PDF, а затем в изображения с помощью `pdftoppm` или `ImageMagick`.
- **Библиотеки Python**: Использование `docx2pdf` для конвертации DOCX в PDF и `pdf2image` для преобразования PDF в изображения.
- **Виртуальная печать**: Использование виртуальных принтеров для прямой конвертации DOCX в изображения.

Выбор решения:

Остановился на использовании `docx2pdf` для преобразования DOCX в PDF и `pdf2image` для конвертации PDF в изображения. Этот подход показал высокую стабильность и

качество конвертации. Также есть возможность использовать в режиме командной строки.

Реализация:

```
from docx2pdf import convert
from pdf2image import convert_from_path
import os

def docx_to_images(docx_path, output_dir, dpi=300):
    # Конвертация DOCX в PDF
    pdf_path = docx_path.replace('.docx', '.pdf')
    convert(docx_path, pdf_path)

    # Конвертация PDF в изображения
    images = convert_from_path(pdf_path, dpi=dpi)

    # Сохранение изображений
    image_paths = []
    for i, image in enumerate(images):
        image_filename = os.path.join(output_dir, f"
{os.path.basename(docx_path).replace('.docx', '')}_page_{i+1}.png")
        image.save(image_filename, 'PNG')
        image_paths.append(image_filename)

    # Удаление временного PDF
    os.remove(pdf_path)

    return image_paths
```

2. Извлечение координат элементов

Рассмотренные варианты:

- **Анализ PDF-файлов**: Использование библиотек для парсинга PDF и извлечения позиционных данных.
- **Скриншоты с использованием координат**: Прямое определение координат на изображении после рендеринга.

Выбор решения:

Выбран подход с использованием библиотеки flitz для извлечения структурной информации из DOCX и сопоставления её с позиционными данными на изображении через анализ PDF. Это позволило автоматически получать координаты элементов без необходимости ручной разметки.

Реализация:

```
from docx import Document
import fitz  # PyMuPDF

def extract_element_coordinates(docx_path, pdf_path):
    doc = Document(docx_path)
    pdf_doc = fitz.open(pdf_path)

    coordinates = {
        "title": [],
        "paragraph": [],
        "table": [],
        "picture": [],
        "table_signature": [],
        "picture_signature": [],
        "numbered_list": [],
        "marked_list": [],
        "header": [],
        "footer": [],
        "footnote": [],
        "formula": []
    }

    for page_num, pdf_page in enumerate(pdf_doc):
        for para in doc.paragraphs:
            bbox = pdf_page.search_for(para.text)
            if bbox:
                elem_type = 'title' if para.style.name.startswith('Heading')
else 'paragraph'
                coordinates[elem_type].append([
                    int(bbox[0].x0),
                    int(bbox[0].y0),
                    int(bbox[0].x1),
                    int(bbox[0].y1)
                ])

    return coordinates
```

3. Создание аннотаций

Требования:

Аннотации должны быть сформированы в соответствии со следующей структурой JSON:

```
{
        "image_height": int,
        "image_width": int,
        "image_path": str,
        "title": [
                [x1, y1, x2, y2],
                ...
        ],
        "paragraph": [
                [x1, y1, x2, y2],
                ...
        ],
        "table": [
                [x1, y1, x2, y2],
                ...
        ],
        "picture": [
                [x1, y1, x2, y2],
                ...
        ],
        "table_signature": [
                [x1, y1, x2, y2],
                ...
        ],
        "picture_signature": [
                [x1, y1, x2, y2],
                ...
        ],
        "numbered_list": [
                [x1, y1, x2, y2],
                ...
        ],
        "marked_list": [
                [x1, y1, x2, y2],
                ...
        ],
        "header": [
                [x1, y1, x2, y2],
                ...
        ],
        "footer": [
                [x1, y1, x2, y2],
                ...
        ],
        "footnote": [
                [x1, y1, x2, y2],
```

```
                ...
        ],
        "formula": [
                [x1, y1, x2, y2],
                ...
        ]
}
```

Проблемы и ошибки при реализации:

1. **Несоответствие координат реальным элементам:**

   При извлечении координат с помощью `fitz` координаты элементов не всегда совпадали с реальным положением на изображении. Это приводило к неточным аннотациям, что негативно влияло на обучение модели.

   Решение:
   - Внедрил масштабирование координат в соответствии с DPI при конвертации.
   - Попытался извлечь слова и соединять их для улучшения точности, однако это привело к новым ошибкам, из-за чего пришлось откатиться к предыдущему подходу.
   - Использовал классификацию по размеру элементов вместо использования специализированных OCR-библиотек, таких как `yesiocr`, для различения формул и изображений.
   -

2. **Отсутствие некоторых элементов в аннотациях:**

   Некоторые элементы, такие как сноски и формулы, не извлекались корректно из документов, что приводило к их отсутствию в аннотациях.

   Решение:
   - Разработал специализированные функции для обработки сносок и формул.
   - Для сносок использовал поиск по специальным стилям и меткам в DOCX.
   - Для формул интегрировал методы распознавания и извлечения изображений формул с их координатами.

3. **Проблемы с распознаванием блоков:**
   Библиотека извлечения блоков иногда ошибочно распознавала элементы документа, что приводило к некорректным аннотациям.

   Решение:
   - Попытался извлечь слова и соединить их для более точного определения границ элементов.

- Этот подход привёл к новым ошибкам и усложнил процесс разметки.
- В итоге решил вернуться к более простому подходу с классификацией по размеру элементов.

4. **Проблемы с производительностью:**

   Обработка большого количества документов занимала слишком много времени, что затрудняло масштабирование.

   Решение:
   - Внедрил параллельную обработку с использованием библиотеки `multiprocessing`, что значительно ускорило процесс.
   - Однако кэширование промежуточных результатов не потребовалось, так как не было необходимости предотвращать повторную обработку одних и тех же данных.

Реализация создания аннотаций:

```
import json
import os
from PIL import Image

def create_custom_annotations(image_paths, coordinates_dict,
output_json_dir):
    for image_path in image_paths:
        image = Image.open(image_path)
        width, height = image.size
        annotation = {
            "image_height": height,
            "image_width": width,
            "image_path": image_path,
            "title": [],
            "paragraph": [],
            "table": [],
            "picture": [],
            "table_signature": [],
            "picture_signature": [],
            "numbered_list": [],
            "marked_list": [],
            "header": [],
            "footer": [],
            "footnote": [],
            "formula": []
        }
```

```
        # Извлечение страницы из имени файла
        try:
            page_num = int(image_path.split('_page_')[-1].split('.png')[0])
        except ValueError:
            print(f"Неверный формат имени файла: {image_path}")
            continue

        elements = coordinates_dict.get(page_num, {})

        for category, bboxes in elements.items():
            if category in annotation:
                annotation[category].extend(bboxes)
            else:
                # Обработка неизвестных типов
                pass

        # Сохранение аннотации
        json_filename = os.path.splitext(os.path.basename(image_path))[0] +
'.json'
        json_path = os.path.join(output_json_dir, json_filename)
        with open(json_path, 'w', encoding='utf-8') as f:
            json.dump(annotation, f, ensure_ascii=False, indent=4)
```

Столкновения с трудностями и их решения

1. **Дублирование ключей в JSON-аннотациях:**

   Проблема:

   При первоначальной реализации обнаружил, что в структуре JSON дважды повторяется ключ `"marked_list"`. Это приводило к перезаписи данных и потере информации.

   Решение:

   - Провёл аудит структуры JSON и удалил дублирующиеся ключи.
   - Перепроверил все ключи на уникальность перед генерацией аннотаций.

2. **Неточности в координатах из-за различий в рендеринге DOCX и PDF:**

   Проблема:

   Координаты, извлечённые из PDF, иногда не совпадали с реальными позициями элементов, что ухудшало качество аннотаций.

   Решение:

   - Внедрил масштабирование координат в соответствии с DPI при конвертации.

- Попытался извлечь слова и соединять их для более точного определения границ элементов, однако этот подход привёл к новым ошибкам, из-за чего пришлось откатиться к предыдущему методу.
- Использовал классификацию по размеру элементов вместо использования специализированных OCR-библиотек, таких как `yesiocr`, для различения формул и изображений.

3. **Проблемы с распознаванием блоков:**

   Проблема:

   Библиотека извлечения блоков иногда ошибочно распознавала элементы документа, что приводило к некорректным аннотациям.

   Решение:
   - Попытался извлечь слова и соединить их для более точного определения границ элементов.
   - Этот подход привёл к новым ошибкам и усложнил процесс разметки.
   - В итоге решил вернуться к более простому подходу с классификацией по размеру элементов.
   -

Выводы

- **Необходимость строгого соблюдения структуры аннотаций:** При работе с кастомными форматами аннотаций важно тщательно проверять соответствие данных заданной структуре, чтобы избежать ошибок при обучении модели.
- **Важность точного сопоставления координат:** Различия в рендеринге DOCX и PDF могут приводить к неточным координатам. Необходимо внедрять дополнительные проверки и корректировки для обеспечения точности разметки.
- **Гибкость в подходах:** При возникновении проблем с распознаванием блоков и точностью аннотаций важно быть готовым к изменению подходов и внедрению новых методов, таких как классификация по размеру вместо использования специализированных OCR-библиотек.
- **Оптимизация производительности критична для масштабирования:** Использование параллельной обработки существенно повысило производительность, что особенно важно при работе с большим объёмом данных.
- **Автоматизация проверки данных повышает качество:** Валидация и проверка данных перед сохранением аннотаций помогают предотвратить ошибки и повысить качество обучающей выборки.

Этап 3: Обучение модели на полученных данных

Получение задачи

На третьем этапе проекта необходимо обучить модель детекции объектов, способную распознавать различные элементы документа на изображениях. Основные задачи включали:

1. **Выбор и настройка модели для детекции объектов**: Рассмотрение различных архитектур и выбор наиболее подходящей.
2. **Подготовка данных для обучения**: Разделение данных на обучающую, валидационную и тестовую выборки, а также аугментация данных.
3. **Обучение модели**: Запуск процесса обучения и мониторинг метрик производительности.
4. **Оценка и улучшение модели**: Анализ результатов обучения и внедрение улучшений для повышения точности.

Понимание задачи

Первоначально задача обучения модели казалась сложной из-за необходимости работы с кастомными аннотациями и интеграции с существующими библиотеками для машинного обучения. Однако по мере продвижения в проекте стало ясно, что при правильной организации процесса и выборе инструментов задача вполне выполнима. Тем не менее, углубляясь в детали и стремясь повысить качество модели, возникли различные аспекты, требующие дополнительного внимания.

Решения и подходы

1. Выбор архитектуры модели

Рассмотренные варианты:

- **YOLO**: Быстрая и эффективная модель для детекции объектов в реальном времени.
- **Faster R-CNN**: Высокоточная модель, подходящая для задач, где важна точность распознавания.
- **DERT (Document Element Recognition Transformer)**: Специализированная модель для распознавания элементов документа.

Выбор решения:

Решил выбрать специлизированую архитектуру для документов из. трансформеров DERT, предполагая, что она может лучше справиться с задачей распознавания элементов документов.

Причины выбора DERT:

- **Специализация**: Модель ориентирована на распознавание элементов документа, что потенциально повышает точность.
- **Архитектурные преимущества**: Использование трансформеров для лучшего понимания контекста элементов.

2. Подготовка данных для обучения

Шаги:

1. **Структурирование данных**: Разделение на обучающую, валидационную и тестовую выборки (например, 80/10/10).
2. **Аугментация данных**: Применение техник аугментации для повышения разнообразия обучающих примеров (например, повороты, сдвиги, изменения яркости).
3. **Форматирование аннотаций**: Преобразование аннотаций из кастомного формата JSON в формат, совместимый с выбранной моделью (например, COCO).

Проблемы и решения:

1. **Несоответствие форматов аннотаций:**

   Проблема:

   Модель DERT ожидала аннотации в формате COCO, тогда как наши аннотации были в кастомной структуре JSON. Это приводило к ошибкам при попытке загрузки данных.

   Решение:

   Был разработан скрипт для преобразования кастомных аннотаций в формат COCO. Однако при этом возникли трудности с точным соответствием полей и структурой данных.

```
import json
import os

def convert_to_coco(custom_annotations_dir, coco_output_path):
    coco = {
        "images": [],
        "annotations": [],
        "categories": []
    }
    category_mapping = {
        "title": 1,
```

```
        "paragraph": 2,
        "table": 3,
        "picture": 4,
        "table_signature": 5,
        "picture_signature": 6,
        "numbered_list": 7,
        "marked_list": 8,
        "header": 9,
        "footer": 10,
        "footnote": 11,
        "formula": 12
    }

    for category, id in category_mapping.items():
        coco["categories"].append({
            "id": id,
            "name": category,
            "supercategory": "document_element"
        })

    annotation_id = 1
    for json_file in os.listdir(custom_annotations_dir):
        if json_file.endswith('.json'):
            with open(os.path.join(custom_annotations_dir, json_file),
'r', encoding='utf-8') as f:
                data = json.load(f)

            image_id = int(os.path.splitext(json_file)
[0].split('_page_')[-1])
            coco["images"].append({
                "id": image_id,
                "file_name": data["image_path"],
                "width": data["image_width"],
                "height": data["height"]
            })

            for category in category_mapping.keys():
                for bbox in data.get(category, []):
                    # Преобразование координат в формат COCO: [x, y,
width, height]
                    coco_bbox = [
                        bbox[0],
                        bbox[1],
                        bbox[2] - bbox[0],
                        bbox[3] - bbox[1]
                    ]
```

```
                    coco["annotations"].append({
                        "id": annotation_id,
                        "image_id": image_id,
                        "category_id": category_mapping[category],
                        "bbox": coco_bbox,
                        "area": (bbox[2] - bbox[0]) * (bbox[3] -
bbox[1]),
                        "iscrowd": 0
                    })
                    annotation_id += 1

    with open(coco_output_path, 'w', encoding='utf-8') as f:
        json.dump(coco, f, ensure_ascii=False, indent=4)
```

2. **Проблемы с производительностью:**

   Проблема:

   Обработка большого количества документов занимала слишком много времени, что затрудняло масштабирование.

   Решение:

   - Внедрил параллельную обработку для пред обработки с использованием библиотеки `multiprocessing`, что значительно ускорило процесс.
   - Однако кэширование промежуточных результатов не потребовалось, так как не было необходимости предотвращать повторную обработку одних и тех же данных.

3. Обучение модели

Попытка обучения DERT:

Поначалу была предпринята попытка обучения модели DERT для распознавания элементов документа. Однако при параллельной работе с авторазметчиком возникли проблемы:

- **Интерпретация аннотаций:** Библиотека COCO не могла корректно обработать созданные аннотации из-за нестандартной структуры JSON.
- **Низкая точность результатов:** Полученные результаты были неудовлетворительными, что требовало дополнительной доработки.

Проблемы и ошибки при обучении:

1. **Несоответствие форматов аннотаций:**

Проблема:

Модель DERT ожидала стандартные аннотации COCO. Почитав документацию узучив структуру создал такую же. Но видимо упустил некоторые момемнты. и в свяхи нехватки времени не смог наладить корректное обучение.

2. **Низкая точность модели:**

   Проблема:

   Обученная модель показывала низкую точность, и видимо выдавала случайные результаты из не понимания разметки
   - Не выявил ошибок COCO разметки. Видимо нужно сильнее углубиться в специфику библиотеки.
   - Решил временно отказаться от использования DERT. Тк в команде есть готовая реализация на YOLO.

Итог:

Не успев завершить параллельную работу с авторазметчиком и не рассчитав силы, обучение модели DERT не дало ожидаемых результатов. Возникли проблемы с интерпретацией аннотаций и низкой точностью.

Выводы

- **Постепенное понимание сложности задачи:** Изначально задача казалась сложной, но по мере работы стало ясно, что основные шаги выполнимы. Однако углубление в детали и стремление повысить качество выявили дополнительные сложности, связанные с точностью разметки и соответствием форматов аннотаций.
- **Необходимость гибкости в подходах:** При возникновении проблем с распознаванием блоков и точностью аннотаций важно быть готовым к изменению подходов и внедрению новых методов, таких как классификация по размеру вместо использования специализированных OCR-библиотек. Также получить необходимые пороги объединения различных блоков.
- **Важность корректной структуры данных:** Дублирование ключей и несоответствие форматов аннотаций критически влияют на процесс обучения модели. Тщательная проверка необходимы для обеспечения их качества.
- **Оптимизация процесса обработки данных:** Использование параллельной обработки существенно повысило производительность, что особенно важно при работе с большим объёмом данных.

Общие выводы и дальнейшие шаги

Общие выводы

На третьем этапе проекта столкнулся с рядом реальных проблем, связанных с подготовкой данных и обучением модели. Несмотря на первоначальное восприятие задачи как сложной, процесс работы показал, что при правильной организации и гибком подходе многие трудности можно преодолеть. Однако стремление к повышению качества выявило дополнительные аспекты, требующие внимания и доработки.

Дальнейшие шаги

1. **Завершение проекта:**
    - Планирую до конца довести проект, завершив обучение модели с использованием проверенных методов и корректной разметки данных.
    - Вернуться к использованию YOLOv5, учитывая полученные уроки и улучшив процесс подготовки аннотаций.
2. **Улучшение процесса разметки:**
    - Разработать более точные методы извлечения и классификации элементов документа.
    - Внедрить дополнительные проверки и валидацию аннотаций для повышения их качества.
3. **Оптимизация и масштабирование:**
    - Продолжить оптимизацию производительности обработки данных.
    - Рассмотреть возможность использования облачных вычислений для ускорения процесса обучения модели.
4. **Тестирование и валидация модели:**
    - Провести обширное тестирование обученной модели на новых документах.
    - Собрать обратную связь и внести необходимые корректировки для повышения точности и надёжности модели.
    - Необходимы тесты для функций из всех этапов. К сожеления в сжатые сроки не получилось всё реализовать.
5. **Исследование альтернативных подходов:**
    - Изучить другие архитектуры моделей и методы обучения, которые могут повысить эффективность детекции элементов документа.
    - Рассмотреть возможность интеграции методов машинного обучения для улучшения распознавания сложных элементов, таких как формулы и сноски.

Заключение

Проект по разработке детектора элементов документа по картинке продвигается успешно, несмотря на возникшие сложности и необходимость корректировки первоначальных

планов. Полученные результаты и опыт позволили лучше понять специфику задачи и определить наиболее эффективные методы её решения. Продолжение работы над проектом с учётом полученных уроков обеспечит достижение поставленных целей и создание надёжного инструмента для автоматического распознавания элементов документов на изображениях. Также очень важен работы в команде тк мы разделили обязанности в конечном итоге получилось реализовать неплохой продукт с достаточно хорошими метриками.