UM6P AL KHWARIZMI Department LSD (License en Science des Données)

End-of-Studies Project Report

Field of Study : Data Science

Development of an OCR System for Supply Chain Management

Enhancing Efficiency and Accuracy

Presented By : …EL KHALDI Zakariyae.....On …..06/07/2023……

Committee :

M. Abdelouahad AITHAMAD, OBENS AUTHENTICS SARL

M. Imad KISSAMI, UM6P MSDA

Academic Year 2022-2023

1 Acknowledgments

I would like to take this opportunity to extend my sincerest gratitude to my supervisors, who have been instrumental throughout my internship journey.

Firstly, to Mr. AITHAMAD Abdelouahad, my supervisor at the startup, who served as a pillar of guidance and support. His availability and commitment to my learning experience played a pivotal role in my project's success. His relentless encouragement spurred my interest and deepened my understanding of Optical Character Recognition technology. It was an honor to learn under your expert tutelage.

To Mr. Imad KISSAMI, my academic supervisor, I express my heartfelt thanks for your meticulous guidance in writing this report. You have steered me throughout this academic endeavor with precision, making this journey both enriching and rewarding. Your expertise and understanding were key to shaping my report and my outlook towards the vast field of AI and data science.

I am also extremely grateful to Mr. Badr OUBENYAHYA, the CEO of the startup, for entrusting me with this project. Your faith in my capabilities motivated me to deliver my best and the opportunity you provided has been truly transformative for my career.

Lastly, I extend my appreciation to the entire 'OBENS' team. The supportive and fun work environment that you created fostered productivity and innovation. Each of you has contributed to my learning process and overall experience, making this internship a memorable journey. It has been a pleasure working with such a dynamic and enthusiastic team.

The knowledge and experience gained during this internship have been invaluable and will certainly guide my future endeavors. My journey would not have been the same without the collective effort and belief of each one of you. Thank you.

Résumé

Ce rapport de projet de fin d'études décrit en détail mon expérience au sein d'une startup spécialisée dans la gestion de la chaîne d'approvisionnement, où mon objectif était de développer un système de reconnaissance optique de caractères (OCR). L'objectif était d'automatiser le processus d'extraction de données à partir de divers types de documents, réduisant ainsi considérablement les erreurs de saisie manuelle et augmentant l'efficacité.

Le rapport commence par une introduction générale, mettant en scène mon projet. Il explique la motivation derrière le projet, notamment la nécessité de la technologie OCR pour améliorer l'efficacité opérationnelle et la traçabilité numérique dans le domaine de la gestion de la chaîne d'approvisionnement. Parallèlement, j'expose mes objectifs d'apprentissage, visant à appliquer mes connaissances académiques en programmation, extraction de données et intelligence artificielle (IA) aux défis du monde réel.

Le Chapitre 1, "Comprendre la technologie OCR", donne un aperçu complet de la technologie OCR. Le chapitre commence par la définition et le concept général de l'OCR, expliquant comment elle peut transformer des documents statiques en données dynamiques, alimentant ainsi les processus d'automatisation numérique. Le chapitre approfondit ensuite les différents types de solutions OCR, y compris les solutions OCR basées sur le cloud, sur site, open-source, commerciales et basées sur l'apprentissage profond. Chaque type est présenté avec ses avantages uniques, offrant un paysage complet de la technologie OCR et de ses diverses applications.

Dans le Chapitre 2, "Choisir la meilleure solution OCR", j'ai entrepris une analyse approfondie pour sélectionner la solution OCR la plus appropriée pour les opérations de la startup. Le chapitre commence par identifier les types de documents et leurs caractéristiques qui sont utilisés dans la startup, tels que divers formats (PDF, PNG, JPG, JPEG) et langues (français, arabe, anglais), des formulaires structurés, des tables et du texte brut. Avec ces exigences définies, j'ai pris en compte six facteurs essentiels -

précision, évolutivité, personnalisation, intégration, coût et support - pour choisir une solution OCR appropriée. Diverses solutions OCR ont été évaluées avec l'aide d'un expert en OCR, aboutissant finalement à AWS Textract et Tesseract OCR comme les options les plus prometteuses.

Le Chapitre 3, "Apprendre et appliquer les technologies sélectionnées", se concentre sur une exploration détaillée des deux technologies OCR sélectionnées - AWS Textract et Tesseract OCR intégré avec la bibliothèque img2table. Ce chapitre documente une comparaison de caractéristique à caractéristique des technologies, ainsi que l'évaluation de leur facilité d'utilisation, performance, coût et flexibilité. L'analyse conclut AWS Textract comme le choix supérieur, principalement en raison de son ensemble de fonctionnalités complet et prêt à l'emploi, de ses performances robustes avec des documents complexes, et de son intégration transparente avec l'écosystème AWS existant.

Dans le Chapitre 4, "Développer le système OCR", je détaille le processus de construction du système OCR en utilisant AWS Textract. Le chapitre couvre l'étape de prétraitement des données, qui comprend la conversion des fichiers PDF en format PNG en utilisant la bibliothèque Python 'pdf2image'. Il continue ensuite à approfondir la compréhension et la gestion des buckets AWS S3 pour gérer efficacement le stockage des données pour le système OCR. J'apprends ensuite et applique le SDK AWS pour Python (boto3) pour intégrer les services AWS avec Python, automatiser l'étape de prétraitement des données et configurer le système OCR en utilisant la programmation orientée objet. Le système comprend deux classes, Document et DocumentProcessor, qui gèrent le traitement des documents avec plusieurs méthodes. Le chapitre se conclut par l'application réussie du système OCR développé sur de vrais documents, en extrayant des lignes de texte, des formulaires, des signatures et des tables, et en déployant les données extraites sur la plateforme.

Dans la conclusion générale, j'aborde "Les suggestions pour les améliorations futures et le travail potentiel à venir", présentant la feuille de route pour affiner le système OCR. Des améliorations significatives, telles que l'ajustement du système à l'aide d'un ensemble plus

large de documents et l'application du traitement du langage naturel (NLP), sont discutées. Malgré l'indisponibilité de ces améliorations lors du stage, en raison de contraintes de temps et de manque de données d'entraînement adéquates, ces opportunités identifiées préparent le terrain pour des développements futurs excitants. Cette vision d'une amélioration continue du système OCR contribue à son potentiel et promet un processus d'extraction de données plus précis et contextuel à l'avenir.

Ce rapport complet résume mon parcours d'apprentissage dans le domaine de la technologie OCR. J'ai couvert chaque étape, de la compréhension des principes de base à l'évaluation et la sélection d'une solution OCR appropriée, et finalement au développement et à la mise en œuvre d'un système OCR complexe. Le projet m'a fourni une expérience pratique inestimable, améliorant considérablement mes capacités techniques tout en contribuant à l'avancement technologique de la startup. Il m'a permis de faire l'expérience de la puissance de l'IA de première main et a renforcé mon engagement envers mes futurs efforts dans ce domaine passionnant.

Mots clés :

'Reconnaissance optique de caractères (OCR)', 'Technologies OCR', 'AWS Textract', 'Tesseract OCR', 'Bibliothèque Img2table', 'Gestion de la chaîne d'approvisionnement', 'Extraction de données', 'Prétraitement des données', 'Intelligence artificielle', 'OCR basée sur le cloud', 'OCR sur site', 'OCR open-source', 'OCR commerciale', 'OCR basée sur l'apprentissage profond', 'Traitement de documents', 'Programmation Python', 'AWS SDK (boto3)', 'Programmation orientée objet (OOP)', 'Buckets AWS S3', 'Développement du système OCR', 'Caractéristiques des documents', 'Évaluation et comparaison de l'OCR', 'Application de l'OCR dans des scénarios réels', 'Conversion PDF en PNG', 'Automatisation', 'Traçabilité numérique', 'Évolutivité des solutions OCR', 'Support et intégration OCR', 'Personnalisation OCR', 'Bibliothèque Python - pdf2image', 'Transition de la programmation fonctionnelle vers OOP', 'Mise en œuvre du système OCR', 'Science des données', 'Apprentissage automatique', 'Reconnaissance d'images', 'Précision du système OCR', 'Évolutivité du système OCR', 'Personnalisation du système OCR', 'Intégration du système OCR', 'Analyse des coûts OCR', 'OCR multilingue (anglais, français, arabe)', 'Analyse de documents structurés', 'Analyse de documents non

structurés', 'Évaluation de la solution OCR', 'OCR dans les chaînes d'approvisionnement', 'Gestion des données', 'Pandas DataFrames', 'Extraction de données tabulaires', 'Reconnaissance de texte', 'Reconnaissance de l'écriture manuscrite', 'Reconnaissance de signatures', 'Extraction de données de formulaires', 'Plan de développement du système OCR', 'Processus d'automatisation numérique'.

Abstract

This end-of-study internship project report provides an in-depth account of my journey with a supply chain management startup, where my objective was to develop an Optical Character Recognition (OCR) system. The goal was to automate the data extraction process from various types of documents, significantly reducing manual data entry errors and increasing efficiency.

The report begins with a General Introduction, setting the stage for my project. It explains the motivation behind the project, particularly the necessity for OCR technology to improve operational efficiency and digital traceability in the realm of supply chain management. In parallel, I state my learning objectives, aiming to apply my academic knowledge in programming, data extraction, and artificial intelligence (AI) to real-world challenges.

Chapter 1, "Understanding OCR Technology," provides a comprehensive overview of OCR technology. The chapter initiates with the definition and general concept of OCR, explaining how it can transform static documents into dynamic data, thereby fueling digital automation processes. The chapter further dives into the different types of OCR solutions, including cloud-based, on-premises, open-source, commercial, and deep learning-based OCR solutions. Each type is presented with its unique advantages, providing a complete landscape of OCR technology and its diverse applications.

In Chapter 2, "Choosing the Best OCR Solution," I embarked on an extensive analysis to select the most suitable OCR solution for the startup's operations. The chapter starts by identifying the types of documents and their characteristics that are used in the startup, such as various formats (PDF, PNG, JPG, JPEG) and languages (French, Arabic, and English), structured forms, tables, and raw text. With these requirements defined, I considered six essential factors - accuracy, scalability, customization, integration, cost,

and support - to choose an appropriate OCR solution. Various OCR solutions were evaluated with the assistance of an OCR expert, eventually narrowing down to AWS Textract and Tesseract OCR as the most promising options.

Chapter 3, "Learning and Applying Selected Technologies," focuses on a detailed exploration of the two selected OCR technologies - AWS Textract and Tesseract OCR integrated with the img2table library. This chapter documents a feature-to-feature comparison of the technologies, along with the evaluation of their ease of use, performance, cost, and flexibility. The analysis concludes AWS Textract as the superior choice, mainly due to its comprehensive and ready-to-use feature set, robust performance with complex documents, and seamless integration with the existing AWS ecosystem.

In Chapter 4, "Developing the OCR System," I detail the journey of building the OCR system using AWS Textract. The chapter covers the data preprocessing step, which includes converting PDF files into PNG format using the 'pdf2image' Python library. It then proceeds to delve into understanding and managing AWS S3 buckets to efficiently handle data storage for the OCR system. I then move on to learn and apply AWS SDK for Python (boto3) for integrating AWS services with Python, automating the data preprocessing step, and setting up the OCR system using Object-Oriented Programming. The system comprises two classes, Document and DocumentProcessor, which manage document processing with several methods. The chapter concludes with the successful application of the developed OCR system on real documents, extracting lines of text, forms, signatures, and tables, and deploying the extracted data on the platform.

In the general conclusion, I touch upon "Suggestions for Future Improvements and Potential Future Work," presenting the roadmap for refining the OCR system. Significant enhancements, such as fine-tuning the system using a larger set of documents and the application of Natural Language Processing (NLP), are discussed. Despite the unavailability of these improvements during the internship, due to time limitations and lack of adequate training data, these identified opportunities set the stage for exciting future developments. This vision for continual refinement of the OCR system contributes

to its potential and promises a more accurate and contextual data extraction process in the future.

This comprehensive report encapsulates my learning journey in the field of OCR technology. I've covered every step, from understanding the core principles to evaluating and selecting an appropriate OCR solution, and ultimately developing and implementing a complex OCR system. The project provided me with invaluable hands-on experience, significantly enhancing my technical capabilities while contributing to the startup's technological advancement. It allowed me to experience the power of AI first-hand and reinforced my commitment to my future endeavors in this exciting field.

Keywords :

'Optical Character Recognition (OCR)', 'OCR Technologies', 'AWS Textract', 'Tesseract OCR', 'Img2table Library', 'Supply Chain Management', 'Data Extraction', 'Data Preprocessing', 'Artificial Intelligence', 'Cloud-based OCR', 'On-premises OCR', 'Open-source OCR', 'Commercial OCR', 'Deep Learning-based OCR', 'Document Processing', 'Python Programming', 'AWS SDK (boto3)', 'Object-Oriented Programming (OOP)', 'AWS S3 Buckets', 'OCR System Development', 'Document Characteristics', 'OCR Evaluation and Comparison', 'OCR Application in Real-world Scenarios', 'PDF to PNG Conversion', 'Automation', 'Digital Traceability', 'Scalability of OCR Solutions', 'OCR Support and Integration', 'OCR Customization', 'Python Library - pdf2image', 'Functional Programming to OOP transition', 'OCR System Implementation', 'Data Science', 'Machine Learning', 'Image Recognition', 'OCR System Accuracy', 'OCR System Scalability', 'OCR System Customization', 'OCR System Integration', 'OCR Cost Analysis', 'Multilingual OCR (English, French, Arabic)', 'Structured Document Analysis', 'Unstructured Document Analysis', 'OCR Solution Assessment', 'OCR in Supply Chains', 'Data Management', 'Pandas DataFrames', 'Tabular Data Extraction', 'Text Recognition', 'Handwriting Recognition', 'Signature Recognition', 'Form Data Extraction', 'OCR System Development Plan', 'Digital Automation Processes'.

List of Abbreviations :

1. **AI** - Artificial Intelligence
2. **AWS** - Amazon Web Services
3. **Boto3** - AWS SDK for Python
4. **OCR** - Optical Character Recognition
5. **OOP** - Object-Oriented Programming
6. **PDF** - Portable Document Format
7. **PNG** - Portable Network Graphics
8. **S3** - Simple Storage Service (part of Amazon Web Services)
9. **SDK** - Software Development Kit
10. **JPG** - Joint Photographic Experts Group (a type of image file)
11. **JPEG** - Joint Photographic Experts Group (a type of image file)
12. **NLP** - Natural Language Processing

List of Tables :

Table of Contents

General Introduction

General Motivation of the Subject :

Managing supply chains involves a lot of details and requires careful attention to record-keeping and accurate data analysis. The startup I interned at is primarily focused on managing supply chains, ensuring traceability, and using data analytics. They identified a need for a more efficient and accurate way to process data from various document types, including PDFs, JPEGs, and PNGs. Previously, data was entered manually, a process that was slow and could lead to mistakes. These issues were a barrier to the startup's aim of using AI tools for digital traceability in supply chains. The desire to streamline this process, reduce errors, and speed up data extraction led to the development of an Optical Character Recognition (OCR) system.

Working in a startup environment offered me an invaluable experience as a data science student. The dynamic nature of startups, with their fast-paced decision-making and innovative approach, provided a fertile ground for learning and growing. It also allowed me to see first-hand the transformative power of data science in redefining industries. The OCR project was an opportunity to put my theoretical knowledge into practice, and bridge the gap between academic learning and real-world problem-solving. It also let me explore and familiarize myself with the rapidly evolving AI technologies and their role in improving business processes.

Project Objectives:

The primary aim of my internship was to develop an OCR system that could automatically extract data from various document types, automating the startup's data entry process. This system was envisioned to increase efficiency, save time, and cut costs, contributing to the startup's goal of using AI tools for digital traceability of supply chains.

As a data science student, my goal was to gain practical experience in applying data science principles and machine learning techniques to real-world problems. I aimed to

understand how the theoretical knowledge I gained from my studies can be transformed into practical solutions, improving my skills in programming, data extraction, and AI.

Document Structure:

This document is organized into chapters that chronologically detail my journey and experiences throughout the internship. Chapter 1 provides an introduction to OCR technology and its various types. Chapter 2 discusses the process of selecting the most suitable OCR solution based on various criteria. Chapter 3 delves into the application of selected technologies, highlighting the challenges encountered and the problem-solving approaches employed. Chapter 4 details the development of the OCR system, from the initial stages of planning to the implementation of the final product. The final chapter, Chapter 5, offers a reflection on the project's outcomes and future prospects. Each chapter concludes with a brief summary of the main points covered. Appendices containing the complete code and a user manual are provided at the end of the report for reference.

1 Chapter 1: Understanding OCR Technology

In this initial chapter, we set the foundation for our exploration into the Optical Character Recognition (OCR) technology, a crucial element of my internship project. The significance of OCR is profound as it forms the bedrock of our objective - to transform how the startup processes documents and to improve efficiency in the data entry process.

The chapter begins with an introduction to OCR technology, outlining its basic functions and explaining how it operates. This will shed light on why it was considered an ideal solution for the startup's data processing challenges. This section also gives an account of my journey in learning and understanding this new technology.

Following the introduction, we will delve into the diverse landscape of OCR solutions available. Each OCR solution brings a unique combination of capabilities and constraints, reflecting the rapid evolution and vastness of the technological field. We will categorize these solutions, offering a brief description of each, encapsulating my exploration of the different OCR technologies during the initial phase of my internship.

By the end of this chapter, you will gain a comprehensive understanding of OCR technology and the range of solutions available. This serves as a foundation for the following chapters, where I share insights on our selection process, the challenges faced, and the development of our customized OCR system. This chapter provides context to my learning experience and the growth I experienced as a data science intern.

1.1 Definition and General Concept of OCR

The starting point of my internship journey revolved around a crucial technology I was encountering for the first time – Optical Character Recognition, or OCR. As a novice in this area, I embarked on a learning process, acquainting myself with this essential tool that bridges the gap between physical and digital information realms.

OCR, as I discovered, is a transformative technology in our increasingly digitized world. It enables the conversion of diverse document types, such as scanned paper, PDF files, or

digital images, into an editable and searchable format. The act of transforming static documents into dynamic data underpins many digital automation processes, making OCR a pivotal tool in our project.

OCR operates by recognizing patterns and shapes in characters present within a scanned image or document. Once identified, these recognized characters are then transcribed into a computer-readable format. In recent years, the incorporation of machine learning and artificial intelligence techniques has significantly elevated the precision and efficiency of OCR systems.

During my internship, I was not just studying OCR from a theoretical standpoint. Instead, I was on the frontline of its practical application, applying the technology to real-world data extraction and automation tasks. It was an invaluable, hands-on opportunity to witness the transformative potential of OCR first-hand.

Throughout this section, I will detail my exploration of OCR technologies during my internship. I'll discuss their distinct functionalities, their role in the broader landscape of digital transformation and automation, and how my understanding of these technologies evolved over time.

1.2 Different Types of OCR Solutions

Upon realizing the vast scope of OCR technology, my initial steps in the internship involved understanding the landscape of OCR solutions available. I dived into a pool of resources, ranging from expert YouTube videos to online documents, to compile a comprehensive understanding of various OCR solutions. Each solution type I discovered appeared to serve specific needs, offering its own set of advantages, making the process of choosing an OCR type an essential task. Below are the various OCR solutions I familiarized myself with during my internship:

1. **Cloud-based OCR:** These solutions, hosted on remote servers and accessible through APIs, stood out for their scalability and convenience. They offered high accuracy rates and were capable of handling large data volumes, making them suitable for applications requiring scalability.
2. **On-premises OCR:** I found that on-premises OCR solutions provided heightened control over data privacy and security. These solutions seemed ideal for dealing with

sensitive or confidential data, even though they required more setup and maintenance compared to their cloud-based counterparts.

3. **Open-source OCR:** During my research, I found that open-source OCR solutions offered the advantage of customization. Despite requiring more technical expertise for setup, these cost-effective solutions could serve smaller projects with limited budgets quite well.
4. **Commercial OCR:** Commercial OCR solutions came with dedicated support and boasted higher accuracy rates than open-source alternatives. These proprietary solutions offered a range of features, making them a potentially attractive choice for businesses.
5. **Deep learning-based OCR:** What intrigued me the most was deep learning-based OCR solutions. Their ability to use neural networks and machine learning algorithms for text recognition tasks presented an opportunity for high precision document processing.

The process of exploring these various OCR types was as enlightening as it was challenging, marking a significant step in my journey to understand and apply OCR technology.

1.3 Conclusion:

In conclusion, the first phase of my internship was dedicated to gaining a comprehensive understanding of Optical Character Recognition technology. Given its potential to transform the manual data entry process into an automated one, learning about this technology was a vital step.

I began by studying the basic concept of OCR, how it works, and its applications in different fields. I came to understand that OCR technology not only streamlines data processing tasks but also unlocks the potential for increased efficiency, better accuracy, and reduced errors.

Moreover, I explored various types of OCR solutions available, ranging from cloud-based and on-premises solutions to open-source, commercial, and deep learning-based OCR. Each type presents its own set of benefits and challenges, and it became clear that

choosing the right OCR solution would depend on several factors including accuracy, scalability, cost, and data privacy.

Thus, the following stage of my internship, which is covered in the next chapter, focused on comparing different OCR solutions and selecting the most appropriate one for the startup's specific needs. This process was both challenging and insightful, providing me with a deeper understanding of how to apply theoretical knowledge to practical scenarios.

2 Chapter 2 : Choosing the Best OCR Solution

Chapter 2, provides an in-depth view of the decision-making process employed during my internship to identify and select the most fitting OCR solution for the startup's needs. The chapter begins by exploring the various OCR solutions and their respective advantages and disadvantages, in light of the startup's specific requirements and constraints. It continues to discuss the features sought in the OCR system, and how these factors guided our choices.

An essential part of this chapter involves understanding how the unique, fast-paced nature of the startup environment influenced our decisions and the necessary trade-offs made. This exploration provides a comprehensive view of the intricate decision-making process, serving as an exemplar of translating academic knowledge to real-world applications, a critical aspect of my internship.

Moreover, this chapter emphasizes the role of critical thinking and problem-solving skills necessary to navigate complex technological decisions in a dynamic startup setting. By the end, the discussion illuminates how all these aspects harmoniously combined to lead us to the OCR solution that best aligned with our objectives.

2.1 Identification of Document Characteristics

My initial task during the internship was to get to grips with the startup's platform, particularly focusing on the characteristics of the various documents used in their system. The documents, rather than following a single structure, were diverse, spanning formats

such as PDF, PNG, JPG, and JPEG. They were predominantly in French, with some in Arabic and English, and a subset that was bilingual.

Each document represented a piece of the supply chain traceability puzzle, encapsulating data from different stages of the supply chain process, such as transactions, transport, quality control, invoices, and contracts. The extraction of this key data was a critical step towards the startup's goal of utilizing AI tools for digital traceability in supply chains.

A significant trait of these documents was their structured formats - tables and forms, which contained critical information. Conversely, there were also documents with valuable raw text and handwritten sections. Official documents typically contained these handwritten elements, and some even included signatures, the presence of which often denoted a document's official status.

Understanding these varied document characteristics was a crucial step in selecting the appropriate OCR solution. The solution had to be versatile enough to handle different document types and languages, be capable of processing both printed and handwritten text, accurately extract data from tables, forms, and raw text, and detect whether a document was signed or not.

By identifying these requirements early in my internship, I was able to streamline my search for an OCR solution, focusing on those that could address the multifaceted and diverse nature of the documents within the startup's system.

Characteristics:

- **Document Types:** The documents were found in several different formats including PDF, PNG, JPG, and JPEG.
- **Languages:** The documents were mostly in French, with a significant number also in Arabic and English. Some documents were bilingual.
- **Document Structure:** The structure of documents varied, with no unique pattern. Information was often found in tables, forms, raw text, handwritten sections and signatures.

2.2 Factors to Consider when Choosing an OCR Solution

In the pursuit of finding an OCR solution that would effectively cater to the startup's diverse needs, I identified several key factors that guided my selection process. These factors are critical to consider for anyone looking to implement an OCR solution, especially in a similar context of dealing with diverse document types and formats. They are as follows:

1. **Accuracy:** The accuracy of an OCR solution is paramount. It dictates the reliability of the extracted text, and high accuracy rates are desirable, especially when considering specific use cases and language requirements. For the startup, an OCR solution had to efficiently handle French, Arabic, and English documents, making accuracy a high-priority factor.
2. **Scalability:** The startup's platform involved dealing with a large volume of documents, often necessitating real-time data processing. Therefore, the scalability of the OCR solution was a significant consideration. Cloud-based solutions are typically more scalable, while on-premises solutions might need additional resources to scale effectively.
3. **Customization:** The OCR solution's customization level was a critical aspect to consider. The startup required an OCR solution that could be tailored to meet its specific needs, whether by fine-tuning the recognition model or customizing the output format.
4. **Integration:** The ease of integration into the existing workflow or system was an important aspect. Solutions that provide APIs or libraries for seamless integration into the existing software stack were prioritized.
5. **Cost:** The cost of the OCR solution was a crucial consideration. This included the license fees, subscription costs, or any additional charges. It was essential to strike a balance between the cost and the features offered by the OCR solution to ensure that the startup received the best value for its investment.
6. **Support:** The level of support offered by the OCR solution was also of significance. The availability of comprehensive documentation, tutorials, and reliable customer

support can make a crucial difference when resolving any issues or challenges encountered during the implementation and usage of the OCR solution.

These factors played a vital role in my decision-making process during the internship, guiding me to an OCR solution that was aligned with the startup's specific needs and constraints.

2.3 Collaboration with an OCR expert to assess potential solutions

During my internship, I was privileged to collaborate with an OCR expert brought on board by the startup. This collaboration was instrumental in the evaluation and selection of the appropriate OCR solution.

We started by exploring a variety of OCR technologies across various categories. Here are the best-rated technologies we considered based on the factors previously identified:

Best-rated Technologies:

Cloud-based OCR:

1. **Google Cloud Vision API:** A cloud-based OCR solution by Google that offers a wide range of OCR capabilities, including text recognition, document layout analysis, and language detection. It is highly scalable, provides high accuracy, and offers various pricing tiers to suit different needs.
2. **Microsoft Azure OCR:** A cloud-based OCR solution by Microsoft that provides text recognition and document analysis capabilities. It offers OCR functionality as part of the Azure Cognitive Services suite, which includes other AI-powered services like computer vision and natural language processing.
3. **AWS Textract:** A cloud-based OCR solution by Amazon that offers text recognition and data extraction capabilities. It's scalable, offering a flexible usage and payment model based on volume.

On-premises OCR:

1. **ABBYY FineReader:** A popular on-premises OCR solution that offers advanced text recognition capabilities, including support for multiple languages, document layout

analysis, and data extraction. It provides high accuracy and allows for customization, making it suitable for various industries and use cases.

2. **OmniPage by Kofax:** An on-premises OCR solution that provides comprehensive text recognition and document conversion capabilities. Known for its high accuracy and support for multiple languages, OmniPage allows for a high degree of customization and is widely used in various sectors, from education to legal services.

Open-source OCR:

1. **Tesseract OCR:** A widely used open-source OCR engine developed by Google that offers text recognition for multiple languages. It has an active community of users and developers, providing continuous updates and improvements.
2. **GOCR:** Another open-source OCR engine that offers basic text recognition capabilities for multiple languages. It is known for its simplicity and ease of use, making it suitable for small-scale OCR projects with limited customization needs.

Commercial OCR:

1. **Adobe Acrobat Pro:** A commercial OCR solution that offers advanced text recognition capabilities, document analysis, and data extraction features. It is widely used in businesses for document management and automation tasks and provides high accuracy and customization options.
2. **Readiris:** A commercial OCR software by IRIS that offers text recognition, document conversion, and data extraction capabilities. It supports multiple languages and offers advanced features such as PDF editing and document compression.

Deep learning-based OCR:

1. **OCRopus:** An OCR solution based on the OCRopus OCR engine that utilizes deep learning techniques for text recognition. It offers high accuracy for complex text recognition tasks and supports multiple languages.
2. **CuneiForm:** A deep learning-based OCR solution that provides text recognition for multiple languages. It is known for its accuracy and flexibility in handling various types of documents, including handwritten text.

OCR Technology	Accuracy	Scalability	Customization	Integration	Cost	Support	Overall
AWS Textract	5	5	4	5	4	5	4.8
Google Vision API	5	4	3	3	4	4	3.8
Microsoft Azure	4	4	4	3	3	4	3.7
ABBYY FineReader	4	3	5	2	4	4	3.7
OmniPage	4	3	5	2	4	4	3.7
Tesseract OCR	3	3	4	3	5	3	3.5
GOCR	3	2	3	3	5	3	3.2
Adobe Acrobat Pro	4	3	4	2	3	4	3.3
Readiris	4	3	4	2	3	4	3.3
OCRopus	4	3	4	2	4	3	3.3
CuneiForm	4	3	4	2	4	3	3.3

Table 1: Rating of OCR Solutions Based on the Selected Factors

We first broke down our analysis into two categories: cloud-based and open-source solutions. This was primarily because these two categories were most relevant to the startup's cost considerations and source control preferences.

In the cloud-based category, AWS Textract emerged as the top solution, while Tesseract OCR (integrated with the img2table library) shone as the most viable open-source solution. We rated these two solutions, as well as others, across several factors: accuracy, scalability, customization, integration, cost, and support. The rating scale was from 1 to 5, with 1 being poor and 5 being excellent.

Here are the results of our ratings:

As evident, AWS Textract scored the highest overall, largely due to its high ratings in accuracy, scalability, and ease of integration with the startup's existing AWS infrastructure. This made it a compelling choice for a cloud-based solution.

On the open-source side, Tesseract OCR integrated with img2table library also scored respectably. It offers good accuracy, scalability, and cost-effectiveness, making it an ideal choice for those looking for a robust, free solution.

In conclusion, after a thorough evaluation process, our team was able to narrow down our choices to two solutions: AWS Textract for a high-performing, cloud-based OCR, and Tesseract OCR as a robust, open-source solution. This set the stage for the final selection, which will be discussed in the next section.

2.4 Conclusion:

In this chapter, we meticulously evaluated various OCR technologies against multiple factors such as accuracy, scalability, customization, integration, cost, and support. Given the startup's current data infrastructure and cost considerations, we primarily focused on cloud-based and open-source solutions.

As a result of our thorough assessment, AWS Textract and Tesseract OCR (integrated with the img2table library) emerged as the most viable solutions in their respective categories. The seamless integration of AWS Textract with our existing AWS services made it a strong contender. At the same time, Tesseract OCR provided a robust and cost-effective open-source alternative.

This evaluation process, conducted in collaboration with an OCR expert, was an enlightening experience. It not only expanded my knowledge of OCR technologies but also enhanced my ability to apply technical criteria for decision-making in real-world projects. As we conclude this chapter, we look forward to a deeper comparative analysis of our top picks, AWS Textract and Tesseract OCR, in the next chapter.

3 Chapter 3 : Learning and Applying Selected Technologies

In the previous chapter, we performed a comprehensive evaluation of various Optical Character Recognition (OCR) technologies and narrowed down our top choices to AWS Textract and Tesseract OCR integrated with the img2table library. As a result of our analysis, these solutions stood out for their robustness, scalability, and potential for integration with our current setup.

In this chapter, we delve deeper into these technologies, exploring their features, capabilities, and nuances. This hands-on learning experience will involve studying documentation, experimenting with the technologies, and assessing how well they can handle our specific OCR needs.

The knowledge and experience gained from this process will not only provide a firm understanding of these technologies but also inform the final decision-making process, ensuring the chosen solution aligns with the startup's requirements and long-term objectives. Let's start this journey of learning and application.

AWS Textract :

Amazon Textract is a service that automatically extracts text, handwriting, and data from scanned documents. Amazon Textract goes beyond simple OCR to identify the contents of fields in forms and information stored in tables.

Features of AWS Textract :

- **Text and Data Extraction:** AWS Textract can extract text and data without manual intervention, even from complex documents such as forms or tables.
- **Handwriting Recognition:** AWS Textract can extract handwritten text from documents - a feature that most OCR solutions do not provide.
- **Integration with AWS Services:** AWS Textract can easily be integrated with other AWS services, making it a seamless fit for our startup that already uses AWS.
- **Security and Compliance:** As a part of AWS, Textract is highly secure and compliant with privacy standards.

Tesseract OCR :

Tesseract OCR, developed by Google, is an open-source Optical Character Recognition engine. It is considered one of the most accurate open-source OCR engines currently available and supports a wide range of languages.

Features of Tesseract OCR

- **Text Recognition:** Tesseract OCR supports a wide variety of languages and can recognize multiple fonts, making it versatile in its application.
- **Open Source:** Being open-source, Tesseract OCR allows for flexibility and customization that's beyond many commercial OCR engines.
- **Active Community:** Tesseract OCR has an active community of users and developers, providing continuous updates and improvements.
- **Integration with img2table:** Our chosen combination integrates Tesseract with the img2table library, enhancing Tesseract's ability to extract tabular data.

In the following sections, we will learn these technologies in-depth, apply them to our data, evaluate their performance, and draw comparisons. The journey starts here.

3.1 Familiarization with Tesseract and img2table

After performing a thorough evaluation of OCR technologies in the previous chapter, our focus narrowed to two potential candidates for our startup's OCR system – Tesseract OCR integrated with the img2table library, and AWS Textract. The former, an open-source solution, stood out due to its accuracy, multi-language support, and the active community behind its continuous development. Its integration with img2table promises enhanced extraction of tabular data, a crucial requirement for our OCR needs. This section aims to share our hands-on experience with Tesseract and img2table, encompassing our understanding derived from documentation and practical implementation.

3.1.1 Understanding the Technologies

We began by familiarizing ourselves with the theoretical aspects of these technologies. Tesseract OCR is renowned for its text detection capabilities from images, thanks to extensive training on multiple languages. Its open-source nature invites contributions from developers worldwide, fostering continuous improvements to the engine.

Img2table complements Tesseract's abilities by identifying and extracting tables from various document formats. It leverages advanced image processing techniques to detect tables, understand their structure, and extract the content. The ability to handle complex table structures, coupled with its feature to convert the extracted tables into Pandas DataFrame representations and then export to Excel files, makes img2table a valuable addition to our OCR toolset.

3.1.2 Practical Implementation and Challenges

Armed with a foundational understanding of Tesseract and img2table, we embarked on practical testing. Our primary experiment involved using these technologies to identify and extract tables from a PDF document. However, the real-world application proved more challenging than anticipated.

Our final testing code was as follows, also you can find the code in following github repository:
https://github.com/zakariyaeELKHALDI/PFE/blob/master/tesseract_trial.ipynb

```python
from img2table.document import PDF
from img2table.ocr import TesseractOCR
from langdetect import detect
import re
import pandas as pd
from pdf2image import convert_from_path
import pytesseract

# Path to the PDF
pdf_path = "PDFdocs/Certificat d?origine - Eur.1.pdf"

# Convert the PDF into images
pages = convert_from_path(pdf_path)

# Instantiation of the pdf
pdf = PDF(src=pdf_path)

# Instantiation of the OCR, Tesseract, which requires prior installation
ocr = TesseractOCR(lang="eng+fra+ara")

# Table identification and extraction
pdf_tables = pdf.extract_tables(ocr=ocr)

# Initialize an empty dictionary to hold dataframes
dfs = {}
```

```
    # Process each table
    for page_num, tables in pdf_tables.items():
        # Get the image corresponding to the page
        image = pages[page_num]

        for i, table in enumerate(tables):
            # Get the bounding box of the table
            bbox = table.bbox

            try:
                bbox_tuple = (bbox.x1, bbox.y1, bbox.x2, bbox.y2)
                # Crop the image to the bounding box
                cropped_image = image.crop(bbox_tuple)

                # Extract text from the cropped image
                table_text = pytesseract.image_to_string(cropped_image, lang="eng+fra+ara")

                # Convert the table text into a dataframe
                lines = table_text.split('\n')
                rows = [line.split() for line in lines if line.strip()]
                df = pd.DataFrame(rows)

                # Add the dataframe to the dictionary
                dfs[f'Page {page_num + 1} Table {i + 1}'] = df

                # Now, process each line in the table
                for line in rows:
                    # Separate the line into different languages and numerical values
                    line_parts = line
                    english_parts = []
                    french_parts = []
                    arabic_parts = []
                    numeric_parts = []

                    for part in line_parts:
                        if part.isdigit():
                            numeric_parts.append(part)
                        else:
                            try:
                                lang = detect(part)
                                if lang == 'en':
                                    english_parts.append(part)
                                elif lang == 'fr':
                                    french_parts.append(part)
                                elif lang == 'ar':
                                    arabic_parts.append(part)
                            except:
                                pass

                    print(f'English: {" ".join(english_parts)}, French: {" ".join(french_parts)},
                          Arabic: {" ".join(arabic_parts)}, Numeric: {" ".join(numeric_parts)}')
            except AttributeError:
                print("Unable to get bounding box coordinates.")

    # Write the dataframes to an Excel file
    with pd.ExcelWriter('tables.xlsx') as writer:
        for name, df in dfs.items():
            df.to_excel(writer, sheet_name=name)
```

PS : the error shown is because i got back in line so that the line could fit in the screenshot frame, check the repository for the code and its execution.

Despite careful preparation, we were met with several issues. We encountered AttributeErrors related to class naming conflicts, improper function utilization, and difficulties in creating bounding boxes for table extraction. Each challenge compelled us to delve deeper into the technologies, dissect the problems, and work towards effective solutions.

The journey with Tesseract and img2table, though challenging, has deepened our understanding of these technologies. Our goal is to establish an efficient OCR solution that can seamlessly extract and process data in different formats and languages.

These experiences set the stage for our next phase, testing AWS Textract, a cloud-based OCR solution. Our final decision will be influenced by a comparative evaluation of the experiences and performance of the tested technologies.

3.2 Practical Application of AWS Textract :

After a comprehensive theoretical understanding of AWS Textract, we decided to put this technology to the test. Given that our startup already uses AWS for its database services, getting access to Textract was a straightforward process. AWS generously provides a demonstration part on their website with a robust graphical interface, which allowed us to explore and evaluate all its features thoroughly.

3.2.1 Test Drive with AWS Textract Demo

We used the AWS Textract Demo to test raw text detection, table detection, forms detection, and even signature detection. We chose several documents with varying complexity levels, including documents with complicated table structures and forms with various data types.

The demo proved to be quite informative and user-friendly, providing clear, well-structured outputs for all the tested features. The raw text was accurately detected and presented. Even when it came to complex tables, Textract preserved the table structure, accurately identifying rows, columns, and cell contents.

What impressed us further was the forms detection. Textract was able to identify the keys and values in the forms and presented them in a well-formatted 2D table, which made data extraction and interpretation significantly easier. Moreover, Textract demonstrated a high accuracy in signature detection, which is an important feature for various business applications.

3.2.2 The Need for Understanding Textract's API and Pricing

Although the AWS Textract demo gave us a good sense of the service's capabilities, using Textract effectively in a production environment requires a deeper understanding of its API and pricing structure.

AWS Textract's documentation is a comprehensive resource that provides detailed information about how to make requests to the Textract API, what features are available, and how to interpret the results. This knowledge is essential for maximizing the benefits of Textract while keeping costs under control.

One critical consideration when using Textract is its pricing structure. The cost of using Textract is based on the number of requests made and the features used. Therefore, to keep expenses manageable, it's essential to optimize the number of requests and strategically select the features that provide the most value for our OCR needs.

Navigating Textract's API and pricing structure may seem daunting at first, but the potential benefits in terms of text extraction, table and form data recognition, and signature detection capabilities are significant. We believe these capabilities are worth the investment of time and effort to fully understand and utilize this service.

Up next, we will consolidate our experiences with Textract and Tesseract OCR integrated with the img2table library, compare their strengths and weaknesses, and analyze how well each aligns with our startup's needs. This comparative analysis will be crucial for making an informed decision about which OCR solution is best for our startup. Stay tuned for our in-depth comparison in the next section.

3.3 Making the Final Choice : AWS Textract vs Tesseract OCR with img2table

After having closely examined both AWS Textract and Tesseract OCR in combination with the img2table library, we are now in a position to make an informed decision on the OCR solution that would be best suited to our startup's needs. Both technologies have demonstrated their capabilities and potential to effectively address our OCR needs. However, their unique characteristics and capabilities have pointed towards different directions in the OCR process. In this section, we will weigh both technologies in terms of their features, ease of use, performance, cost, and flexibility, to arrive at our final decision.

3.3.1 Feature Comparison

AWS Textract provides features such as text and data extraction, handwriting recognition, and seamless integration with AWS services. It has successfully showcased its proficiency in recognizing not just text but also forms and tables, including those with complex structures. The ability to recognize signatures adds a significant edge.

Conversely, Tesseract OCR with img2table also showed encouraging results. It delivers powerful text recognition and is supported by an active community that continuously updates and improves it. Its integration with img2table broadens its capabilities to efficiently recognize and extract tables, making it a strong contender to AWS Textract.

3.3.2 Ease of Use and Performance

In terms of ease of use, AWS Textract takes the lead, owing to its intuitive API and extensive documentation. Tesseract OCR, although not as user-friendly, still has robust documentation and a supportive community for troubleshooting.

On the performance front, AWS Textract delivers rapid and precise results even with complex documents. In contrast, Tesseract OCR integrated with img2table, while capable of accurate text recognition, faced some hurdles in table extraction, requiring additional effort to refine the results.

3.3.3 Cost and Flexibility

From a cost perspective, Tesseract OCR with img2table has an advantage being an open-source solution, whereas AWS Textract has usage costs depending on the number of requests and features used.

In terms of flexibility, Tesseract OCR's open-source nature gives it an advantage as it allows for customization. However, AWS Textract's ability to integrate with other AWS services can streamline the OCR process, particularly for systems already using the AWS ecosystem.

3.3.4 Final Verdict

After detailed analysis and consideration of all factors, the decisive factor that led us to choose AWS Textract was its comprehensive and ready-to-use feature set. Although Tesseract OCR with img2table provides powerful text recognition and has extensive language support, building a robust system to process our specific document types would require separate development and extensive training for each type of detection, which would consume significant time and resources.

In contrast, AWS Textract, with its advanced features such as form, table, and signature recognition, offers a ready-to-use solution that can reduce development and training time significantly. Its seamless integration with AWS, its ease of use, and its strong performance with complex documents further solidify its suitability. Although there is a cost associated with using AWS Textract, we believe the investment is justified given the time, effort, and resources it saves, and the quality of the features it provides.

Thus, with a focus on efficiency and operational feasibility for our startup, AWS Textract emerges as the superior choice. As we move forward, we will delve deeper into AWS Textract, exploring ways to optimize our use of the service to extract maximum value while managing costs effectively.

3.4 Conclusion:

Chapter 3 has been an immersive learning journey as I navigated two promising OCR technologies - AWS Textract and Tesseract OCR integrated with the img2table library.

This exploration marked a pivotal phase in my internship as a data science student, deepening my understanding of OCR technologies and their practical applications.

The thorough evaluation of both technologies honed my analytical skills, allowing me to critically assess their features, capabilities, and potential for integration. Despite the robust competition offered by Tesseract OCR with its open-source nature and powerful text recognition, AWS Textract emerged as the technology of choice for my project, demonstrating superior user-friendliness, efficiency, and advanced features.

The decision to choose AWS Textract was influenced not only by its inherent capabilities but also by its seamless integration with the existing AWS ecosystem, which aligns perfectly with the project's needs. This decision was a significant milestone in my data science journey, reflecting my ability to apply theoretical knowledge to real-world situations.

Having laid the groundwork in Chapter 3, I am eager to apply these learnings to the next phase of my internship project: the development of the OCR system using AWS Textract. In Chapter 4, I will detail my development journey, marking my progression from conceptual understanding to practical application, further augmenting my growth as a data scientist.

4 Chapter 4: Developing the OCR System

Chapter 4 signals a pivotal point in the internship, as it transitions from evaluation and selection of Optical Character Recognition (OCR) technologies to the actual development of the OCR system. This shift mirrors the broader learning process, moving from theoretical understanding to hands-on application.

The focal point of this chapter is the practical implementation of AWS Textract, the OCR technology selected based on the comprehensive evaluation conducted in Chapter 3. This process will begin with an in-depth exploration of AWS documentation, followed by a detailed understanding of S3 buckets and the AWS SDK for Python. These components are vital for integrating AWS Textract effectively into the system.

Further, a detailed project plan will outline the strategic steps for system development, ensuring optimal use of resources and time. Coding and problem-solving stages will bring

into play various programming paradigms, demonstrating the versatility required in practical applications.

Success in this phase involves the successful extraction of raw text, table-formatted data, form-formatted data, and signatures. Each achievement signifies a critical milestone, indicative of the progress made in translating theoretical knowledge into a practical solution.

Let's continue this exciting journey into the world of practical data science application !

4.1 Data Preprocessing - Converting PDFs to Images

The first stage of my journey towards building an OCR system with AWS Textract was data preprocessing. Given that AWS Textract performs excellently with image formats like PNG, JPG, and JPEG, but occasionally fails with PDF files, I decided to convert all documents into images before processing.

The challenge with PDFs was that AWS Textract sometimes failed to process PDF files which were essentially images in a PDF format. However, it performed well with PDFs in their numeric format. This discovery came from iterative testing with various document types, highlighting the necessity of hands-on experience to complement theoretical knowledge. To overcome this, I chose to convert all PDF files into PNG format.

To achieve this, I leveraged the 'pdf2image' Python library, which allows the conversion of PDFs into images. Here is the Python code I used:

```python
import os
from pdf2image import convert_from_path

def convert_pdf_to_images(pdf_path, output_folder, dpi=400):
    # Convert PDF to list of images
    images = convert_from_path(pdf_path, dpi)

    # Check if the output folder exists, create it if it doesn't
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Get the base name of the PDF file
    pdf_name = os.path.splitext(os.path.basename(pdf_path))[0]

    # Save images to the output folder
    for i, image in enumerate(images):
        # Generate the output file name
        if len(images) > 1:
            output_filename = f'{pdf_name}_{i}.png'
        else:
            output_filename = f'{pdf_name}.png'

        # Save the image
        output_path = os.path.join(output_folder, output_filename)
        image.save(output_path, 'PNG')

# Specify the folder path
input_folder_path = 'PDFdocs'
output_folder_path = 'PNG&JPGdocs'

# Iterate over the files in the folder
for filename in os.listdir(input_folder_path):
    file_path = os.path.join(input_folder_path, filename)

    # Check if the file is a regular file and has a PDF extension
    if os.path.isfile(file_path) and filename.endswith('.pdf'):
        # Process the PDF file here
        convert_pdf_to_images(file_path, output_folder_path, dpi=400)
    else:
        # Print the file path if it's not a regular PDF file
        print(f"Skipping: {file_path}")
```

The process of converting PDF files to images was a crucial initial step in the data preprocessing pipeline. While it presented an unexpected challenge, it reinforced the importance of iterative testing and flexibility in my approach. The experience underlined the role of hands-on experience in identifying and solving potential issues in the pipeline.

With the conversion complete, the path was now clear for the next phase of the project, which would involve understanding and implementing AWS S3 buckets for data storage.

4.2 Understanding AWS S3 Buckets

In the pursuit of creating an efficient OCR system with AWS Textract, it became clear that local data storage would not be sufficient. AWS S3 buckets, a scalable and versatile cloud storage service, emerged as the most suitable solution. However, understanding and implementing this service required an in-depth understanding of its functions and features.

S3, or Simple Storage Service, is a service offered by Amazon Web Services (AWS) that provides object storage through a web service interface. It uses the concept of 'buckets' to organize data, much like directories on a computer. Buckets can store any amount of data and any type of file, with individual file sizes ranging from 1 byte to 5 terabytes.

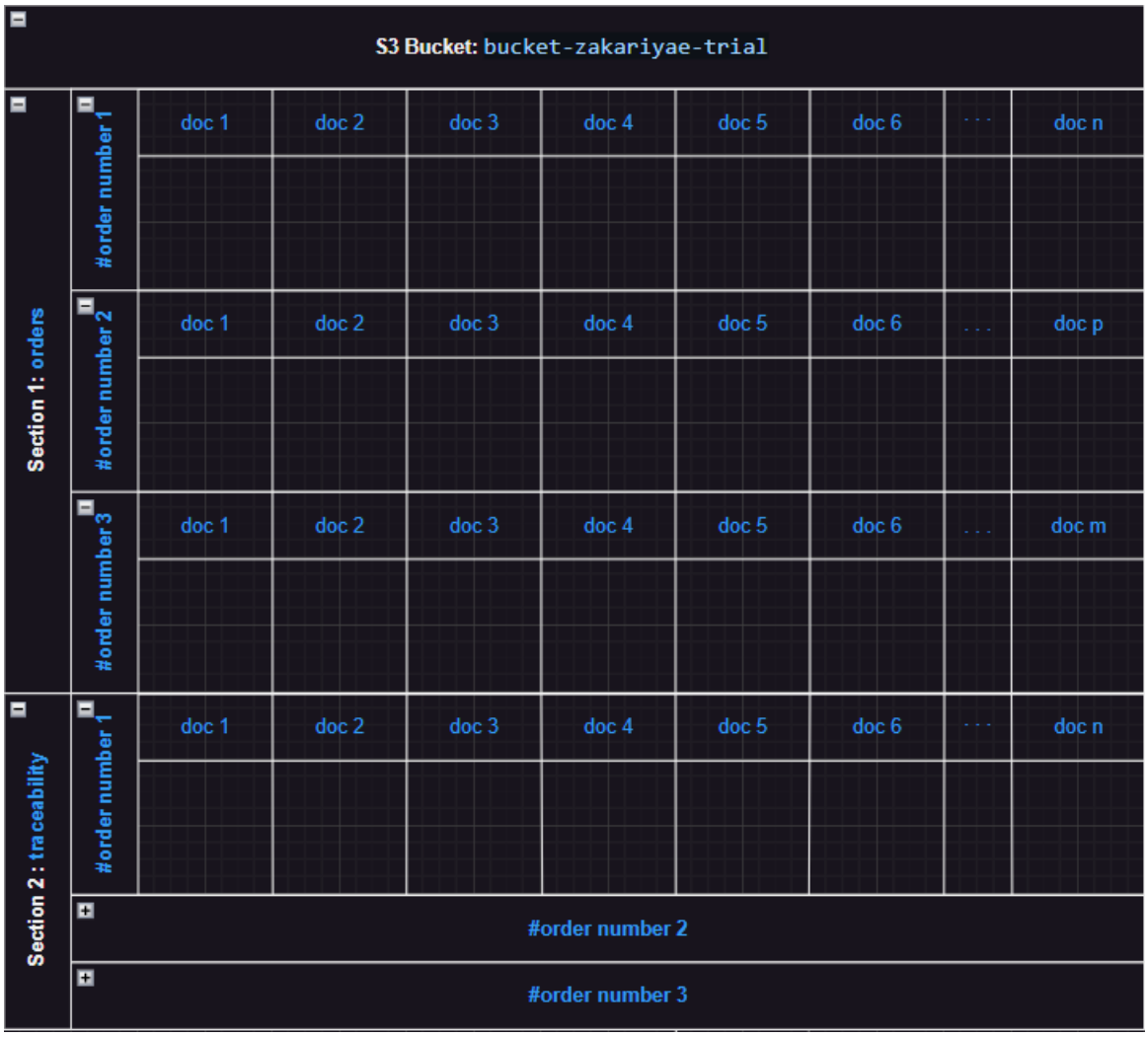
One can upload, download, and delete files (objects) in a bucket and configure a bucket to log access requests, enabling monitoring. Buckets have various features such as versioning to keep multiple variants of an object in the same bucket, event notifications, lifecycle policies, and transfer acceleration.

Understanding S3 buckets was a learning curve, but it was essential for effectively managing the vast amounts of document data I would be handling in this project.

4.3 Mapping the Organisation of Documents in S3 Buckets

With a solid understanding of AWS S3 buckets, the next task was to figure out an efficient way to store the documents in the S3 buckets. The ultimate aim was to have an organizational system that would facilitate easy access and manipulation during the programming phase.

I decided to use a structured approach, creating a separate bucket with various folders, each of which would represent a different section on the website. For instance, in the 'orders' section, each order number had its own folder within the 'orders' folder. Each of these order numbers’ folders housed the documents related to that particular order. Here’s an overall mapping of how the bucket would look like:



This approach ensured clarity and manageability of data, streamlining the process for the upcoming phase of extracting data from these documents. The skills and knowledge I gained during this process were fundamental for the progression of the internship project, once again emphasizing the practical aspects of data science.

4.4 Getting Familiar with AWS SDK for Python - boto3

4.4.1 Connecting to AWS Services :

The first step in integrating the OCR system with AWS services was establishing a connection using the SDK for Python - boto3. This task required the use of the credentials provided by my supervisor to create a session with the desired region. This session was

then used to create an S3 resource object, thereby establishing the necessary foundation to interact with AWS services programmatically. Below is the Python code used for this process:

```
import boto3

# Specify the desired region
region_name = 'eu-central-1'  # Replace with your desired region

# Create a session with the specified region
session = boto3.Session(region_name=region_name)

# Create an S3 resource object using the default session
# (which will use credentials from the ~/.aws/credentials file)
s3 = session.resource('s3')
```

4.4.2 Manipulating S3 Buckets with boto3 :

Once the connection was established, the next step was the manipulation of the S3 buckets - a critical aspect of data management in AWS. This manipulation encompassed various tasks such as creating a bucket, uploading files to it, deleting files from it, and finally, deleting the bucket itself. All these operations were performed using the boto3 Python library.

One of the crucial learning outcomes from this phase was the understanding of the difference between an S3 resource and an S3 client, two concepts that form the core of boto3 functionality.

An S3 resource in boto3 represents a higher-level, object-oriented interface that comes with automated handling of certain tasks. This abstracted layer enables the user to interact with resources (like S3 buckets) as if they were Python objects, making it a user-friendly interface to AWS services.

On the other hand, an S3 client represents a lower-level, procedural interface to AWS services. It provides methods that correspond one-to-one with the underlying AWS REST operations. The client gives more flexibility and control to the user and is particularly useful for complex tasks that may require explicit and detailed configuration.

In this project, I primarily used the S3 resource for more straightforward tasks like creating a bucket, uploading a file, or deleting an object from a bucket. It simplified operations by allowing the use of method calls on the S3 bucket object directly.

Conversely, I used the S3 client when more control and configuration was needed, for instance, when listing all the objects in a bucket, getting an object from the bucket, or performing operations that require interaction with the AWS RESTful API.

Understanding the difference between S3 client and S3 resource and choosing the right one for each task is crucial to efficiently interacting with AWS services and optimizing code.

Here is the code used to perform these tasks:

```
#Creating a bucket
bucket_name = 'bucket-zakariyae-trial'
s3.create_bucket(Bucket = bucket_name)

#Uploading a file to S3:
data = open('PNG&JPGdocs/Facture.png', 'rb')
s3.Bucket(bucket_name).put_object(Key='Facture.png', Body=data)

#Deleting a file from S3
s3.Object(bucket_name, 'Facture.png').delete()

#Deleting a bucket (all the objects inside the bucket should be deleted firs
s3.Bucket(bucket_name).delete()

# Print out all bucket names
for bucket in s3.buckets.all():
    print(bucket.name)
```

4.4.3 Integrating Data Preprocessing with boto3:

The final part involved putting the lessons learned to practical use. The goal was to automate the data preprocessing phase with the newfound knowledge of boto3. This involved downloading PDF files from the S3 bucket, converting them into images using the 'pdf2image' library, and then re-uploading them back into the S3 bucket. The 'os' library was used to manipulate the file paths and names.

This step marks the completion of the data preprocessing phase, setting the foundation for the development of the OCR system using AWS Textract.

Here is the Python code for this process:

```python
import os
import boto3
from pdf2image import convert_from_bytes

# Create an S3 client
s3_client = boto3.client('s3')

# Specify the S3 bucket name and folder path
bucket_name = 'your-bucket-name'
folder_path = 'your-folder-path'

# List objects in the S3 bucket folder
response = s3_client.list_objects_v2(Bucket=bucket_name, Prefix=folder_path)

# Iterate over the objects in the S3 bucket folder
for obj in response['Contents']:
    # Get the object key
    object_key = obj['Key']

    # Download the PDF file from S3
    response = s3_client.get_object(Bucket=bucket_name, Key=object_key)
    pdf_bytes = response['Body'].read()

    # Convert the PDF to images
    images = convert_from_bytes(pdf_bytes)

    # Get the PDF filename without the extension
    pdf_name = os.path.splitext(os.path.basename(object_key))[0]

    # Iterate over the images and save them as PNG
    for i, image in enumerate(images):
        if len(images) > 1:
            output_filename = f'{pdf_name}_{i}.png'
        else:
            output_filename = f'{pdf_name}.png'

        # Convert the image to bytes
        image_bytes = image.tobytes()

        # Upload the PNG image to S3 in the same folder
        s3_client.put_object(Body=image_bytes, Bucket=bucket_name,
                             Key=os.path.join(folder_path, output_filename))
        # Delete the PDF file from S3
        s3_client.delete_object(Bucket=bucket_name, Key=object_key)

    print(f"Converted and uploaded PDF '{object_key}' to PNG images.")
```

The successful application of these concepts marked a major milestone in the development of the OCR system. It opened doors for more advanced manipulation of the S3 buckets in the subsequent stages, thereby bringing me closer to the ultimate goal of automating document management.

4.5 Elaborating a Development Plan and building the OCR System

In this section, we delve deeper into the planning and construction of the OCR (Optical Character Recognition) system, built to extract data from documents. Given the complexity of the tasks at hand and the diverse nature of the documents to be analyzed, a meticulous development plan was essential to guide the process and ensure success.

Over the course of the internship, the realization that procedural programming alone could not cater to the complexity of the system led us to embrace Object-Oriented Programming (OOP). This major shift will be discussed in detail, alongside the rationale behind designing the system's classes and methods. The mapping of the classes, their attributes, and the methods are intricately explained to provide a complete understanding of their functions and their role in the OCR system.

This section serves as a comprehensive guide to the development of the OCR system. It illustrates the transformation from an initial concept to a functional, scalable, and maintainable system, focusing on every major decision, the obstacles encountered, and how they were overcome. By the end, it should give a clear understanding of the system architecture and functionality, all while elucidating the technical skills acquired during the internship.

The first part of this section will begin with the transition from functional programming to OOP, explaining the reasons behind this crucial decision, and how it was implemented in designing the OCR system. Let's dive in!

4.5.1 Switching from Functional Programming to OOP and building the OCR System

As the complexity of the OCR system grew, it became increasingly clear that sticking to the procedural style of programming would not be sufficient to handle the various components and processes that the system would need. This realization led to the decision to switch to an Object-Oriented Programming (OOP) approach.

OOP is a programming paradigm that uses "objects" – instances of classes – which are capable of storing data and have their own attributes and methods. The primary benefits of

using OOP in this project were its ability to provide structure, improve code readability and reusability, and make the system more scalable and maintainable.

Mapping the Classes for the OCR System :

To kickstart the transition to OOP, I began by identifying the entities or components of the OCR system that could be represented as classes. These classes, their attributes, and methods were defined based on the functionality they would provide in the system.

Document Class:

The Document class represents the files that need to be processed by our OCR system. An instance of this class represents a unique document in the system.

The attributes of this class are:

- **path:** This attribute stores the location of the document in your local file system.
- **bucket_name:** This attribute is a placeholder for the name of the S3 bucket where the document will be uploaded.
- **order_num:** This attribute stores the order number of the document, which is yet to be specified.
- **name:** This attribute stores the name of the document, which is yet to be specified.
- **type:** This attribute stores the file extension of the document. This is determined using the **get_file_extension()** method.
- **is_analysable:** This attribute checks if the document is of a file type that is compatible with OCR analysis. This is determined using the **is_ocr_analysable() method**.

```python
class Document:
    def __init__(self, path):
        """
        The constructor of the Document class, which initializes instance variables.
        """
        self.path = path  # The path of the document
        self.bucket_name = "Name of bucket containing the document is not specified yet!"  # The bucket name
        self.order_num = "Order number of document is not specified yet!"  # The order number
        self.name = "Name of document is not specified yet!"  # The document's name
        self.type = self.get_file_extension()  # The type of the document
        self.is_analysable = self.is_ocr_analysable()  # Checks whether the document can be analyzed with OCR

    def get_file_extension(self):
        """
        Method to get the file extension from the path.
        """
        _, extension = self.path.rsplit('.', 1)  # Split the path by '.' and get the last part
        return '.' + extension.lower()  # Return the extension with a '.' prefix and in lower case

    def is_ocr_analysable(self):
        """
        Method to check whether the document is analyzable with OCR.
        """
        if self.type == ".pdf":  # If the document is a PDF
            return False  # Return False
        elif self.type in [".jpg", ".jpeg", ".png"]:  # If the document is a JPG, JPEG, or PNG
            return True  # Return True
        else:
            # If the document is not any of the supported types, raise a ValueError
            raise ValueError("The imported document has an unsupported file format.")
```

DocumentProcessor Class:

The DocumentProcessor class handles the task of processing the documents. An instance of this class encapsulates the functionality to process documents.

Constructor:

The constructor of the DocumentProcessor class initializes important AWS resources and clients, namely, S3 and Textract. This happens when an instance of the class is created.

It takes an optional parameter, **region_name**, which defaults to ***'eu-central-1'***. This value sets the region for the AWS session. It then establishes an AWS session and creates an S3 resource and client, along with a Textract client. These are critical for the class's methods, as they allow interaction with the S3 storage and Textract services of AWS.

This constructor enables the DocumentProcessor class to interact with AWS services right after an instance is created, making the class self-sufficient and ready to handle document processing tasks immediately upon instantiation.

```python
import boto3
from pdf2image import convert_from_bytes
import io
import pandas as pd

class DocumentProcessor:
    def __init__(self, region_name = 'eu-central-1'):
        """
        The constructor of the DocumentProcessor class.
        """
        self.session = boto3.Session(region_name=region_name)  # Create a session with the specified region
        self.s3_resource = self.session.resource('s3')  # Create an S3 resource from the session
        self.s3_client = self.session.client('s3')  # Create an S3 client from the session
        self.textract = self.session.client('textract')  # Create a Textract client from the session
```

Some of the main methods in this class include:

- **get_bucket_folder_content():** This method lists the content of a specified S3 bucket folder.
- **file_converter():** This method converts a document (like PDF) to a format that is compatible with the OCR (like PNG).
- **analyze_document():** This method uses the AWS Textract service to analyze a document.
- **extract_data():** This method extracts relevant data from the Textract response.
- **tables_to_excel():** This method saves extracted tables to an Excel file.
- **forms_to_df():** This method converts extracted form data to a pandas DataFrame.
- **process_document():** This method initiates the document processing workflow, which includes analyzing the document using Textract and extracting useful information from the response.
- **process_documents():** This method can process multiple documents at once.
- **group_and_order_text():** This method groups and orders lines or words by their position in the document.

Now let’s get to each method, how it works and what it does exacly, then introduce its code :

- **get_bucket_folder_content():** this method acts as the initial gatekeeper in the document processing workflow. It essentially gets the content of a specified S3 bucket folder and transforms it into an array of ‘**Document’** instances.

This method takes a **'folder_path'** and a **'bucket_name'** as input, lists all the objects (documents) in the specified folder of the S3 bucket, and for each of these documents, it creates an instance of the **'Document'** class. The details of each document such as the bucket name, order number, and name are extracted and stored within the **'Document'** instance.

All these **'Document'** instances are then appended to a list named **'documents'** documents which is returned by the method. If no content is found in the bucket, it informs the user with a print statement.

So, in essence, this method is preparing and providing the necessary data (documents from the S3 bucket) in a structured way for the upcoming steps in the OCR system.

```
def get_bucket_folder_content(self, folder_path, bucket_name = 'bucket-zakariyae-trial'):
    """
    Method to get the content of a folder in an S3 bucket.
    """
    folder_name = folder_path.rstrip('/').split('/')[-1]  # Extract the folder name from the path

    # List the objects in the specified S3 bucket and folder
    response = self.s3_client.list_objects_v2(Bucket = bucket_name, Prefix = folder_path)

    documents = []  # List to store the documents
    if 'Contents' in response:  # If the response contains 'Contents'
        for object in response['Contents']:  # For each object in the response
            object_key = object['Key']  # Get the object key

            if object_key.endswith('/'):  # If the object key is a directory
                continue  # Skip to the next iteration

            document = Document(object_key)  # Create a Document instance for the object
            document.order_num = folder_name  # Assign the folder name to the order number of the document

            # Extract the document name from the object key and assign it to the name of the document
            document_name = object_key.rsplit('/', 1)[-1].rsplit('.', 1)[0]
            document.name = document_name

            document.bucket_name = bucket_name  # Assign the bucket name to the bucket name of the document

            documents.append(document)  # Append the document to the list
    else:
        print('No objects found in the specified bucket and folder.')
    return documents  # Return the list of documents
```

- **file_converter() :** this method's role is to convert a PDF document into a set of PNG images. This is crucial for the OCR process as most OCR engines, including AWS Textract, cannot directly process PDF files.

Here's how it works:

  - The method accepts a **'Document'** instance as input. This **'Document'** instance contains the information about the document such as the path and the bucket name.

- It then retrieves the document from the specified S3 bucket using the bucket name and path information from the **'Document'** instance.
- The method uses the **'convert_from_bytes'** function from the **'pdf2image'** library to convert the PDF file (obtained from the S3 bucket) into a series of images.
- It returns a list of these images, with each item in the list representing a page of the PDF document.

```
def file_converter(self, document):
    """
    Method to convert a PDF document to PNG images.
    """
    # Fetch the document from the S3 bucket
    response = self.s3_client.get_object(Bucket = document.bucket_name, Key = document.path)

    # Convert the document to images
    images = convert_from_bytes(response['Body'].read())

    return images  # Return the images
```

Now, let's consider how it would work with the **'get_bucket_folder_content'** method:

The **'get_bucket_folder_content'** method fetches the contents of a specified S3 bucket folder and returns a list of **'Document'** instances, each representing a document in the S3 bucket. Each **'Document'** instance contains the bucket name and the path, which can then be passed as input to the **'file_converter'** method. The **'file_converter'** method can then fetch the document from the S3 bucket and convert it into a set of images suitable for the OCR process.

By this means, the **'get_bucket_folder_content'** method prepares the documents for processing and the **'file_converter'** method ensures they are in the right format for OCR analysis.

- **analyze_document() :** this method is the core method for performing Optical Character Recognition (OCR) using AWS Textract service. The goal of this method is to analyze a document and extract information from it. It supports both image formats like JPG, JPEG, and PNG, and PDF documents.

Here's how it works:

- The method accepts a Document instance as an argument, which includes all the necessary details about the document (type, path, bucket name, etc.).

- If the document is an image file (JPG, JPEG, or PNG), it directly uses the Textract service to analyze the document. It specifies the S3 bucket name and document path in the Document parameter, and includes "TABLES", "FORMS", and "SIGNATURES" in the FeatureTypes parameter to indicate that it wants to detect these types of data in the document. The method then returns the response from Textract.
- If the document is a PDF, the method first converts the PDF to a set of PNG images using the file_converter method. It then loops over each image, saves it to a byte stream, and analyzes it using Textract, similar to how it analyzes image files. The method appends the response for each image to a list and returns the list at the end.
- If the document is neither an image file nor a PDF, the method raises a ValueError indicating that the document has an unsupported file format.

```python
def analyze_document(self, document):
    """
    Method to analyze a document using AWS Textract.
    """
    if document.type in [".jpg", ".jpeg", ".png"]:  # If the document is a JPG, JPEG, or PNG
        # Analyze the document using AWS Textract
        response = self.textract.analyze_document(
            Document={'S3Object': {'Bucket': document.bucket_name, 'Name': document.path}},
            FeatureTypes=["TABLES", "FORMS", "SIGNATURES"])
        return response  # Return the response

    elif document.type == ".pdf":  # If the document is a PDF
        png_files = self.file_converter(document)  # Convert the PDF to PNG images

        responses = []  # List to store the responses
        for png_file in png_files:  # For each PNG image
            byte_stream = io.BytesIO()  # Create a byte stream
            png_file.save(byte_stream, format='PNG')  # Save the PNG image to the byte stream
            byte_stream = byte_stream.getvalue()  # Get the byte data from the byte stream

            # Analyze the PNG image using AWS Textract
            response = self.textract.analyze_document(Document={'Bytes': byte_stream},
                                                      FeatureTypes=["TABLES", "FORMS", "SIGNATURES"])
            responses.append(response)  # Append the response to the list

        return responses  # Return the list of responses

    else:
        # If the document is not a supported type, raise a ValueError
        raise ValueError(f'The document "{document.name}" has an unsupported file format.')
```

This method works hand-in-hand with the **'get_bucket_folder_content'** and **'file_converter'** methods. The **'get_bucket_folder_content'** method retrieves the document information from the S3 bucket, which the **'analyze_document'** method uses

to analyze the document using Textract. If the document is a PDF, the **‘file_converter’** method is used to convert the PDF to PNG images, which can then be analyzed by Textract. This way, the methods work together to provide a comprehensive OCR solution.

- **extract_data() :** this method processes the responses obtained from the AWS Textract service, extracting valuable data, such as tables, forms (key-value pairs), signatures, lines, and words. It organizes the extracted data into respective lists, allowing for more efficient and specific use of the data later in the workflow.

Here's how the process unfolds:

- The method starts by ensuring that the responses are in a list format. This is necessary since the method needs to loop over each response, and there might be only a single response (dict format) in some cases.
- The method initializes lists to store tables, forms, signatures, lines, and words extracted from the responses.
- For each response, the method loops over all blocks (each representing an item detected by Textract) and identifies the type of each block. Depending on the block type, it performs specific extraction operations:
    - For 'TABLE' blocks, the method creates a dictionary for each table and a list to hold cell data. It iterates over 'CHILD' relationships within the block (these correspond to cells in the table), extracting cell data using the **‘_extract_cell_data’** helper method, which also processes 'CHILD' relationships to fetch associated words in each cell.
    - For 'KEY_VALUE_SET' blocks, the method identifies 'KEY' and 'VALUE' entities and extracts their respective texts. Again, 'CHILD' relationships are used to link associated word blocks. The key-value pairs are stored in a dictionary and added to the forms list.
    - For 'SELECTION_ELEMENT' blocks with 'SELECTED' status, the method assumes a signature presence and extracts its bounding box information.
    - For 'LINE' and 'WORD' blocks, the method simply extracts the text along with the top coordinate of the bounding box.

The helper function _extract_cell_data is used to consolidate the cell information for each table. This function creates a dictionary that holds the row index, column index, and text for each cell. If the cell block has 'CHILD' relationships, the function loops over each relationship to extract the associated word blocks, combining the text into the cell's text.

```
    def extract_data(self, responses):
        """
        Extract data from the response of the AWS Textract API.
        """
        if isinstance(responses, dict):
            responses = [responses]  # If there is only one response, convert it to a list for the loop below

        tables = []  # List to hold extracted tables
        forms = []   # List to hold extracted form key-value pairs
        signatures = []  # List to hold extracted signatures
        lines = []  # List to hold extracted lines
        words = []  # List to hold extracted words

        # Loop over each response
        for response in responses:
            # Iterate through blocks in the response
            for block in response['Blocks']:
                block_type = block['BlockType']

                # Extract tables
                if block_type == 'TABLE':
                    table = {}  # Dictionary to hold table data
                    cells = []  # List to hold cell data

                    # Check if block has relationships
                    if 'Relationships' in block:
                        # Loop over each relationship
                        for relationship in block['Relationships']:
                            # Check if the relationship is of type 'CHILD'
                            if relationship['Type'] == 'CHILD':
                                # Loop over each ID in the relationship
                                for cell_id in relationship['Ids']:
                                    # Find the block with the matching ID
                                    cell_block = [b for b in response['Blocks'] if b['Id'] == cell_id][0]
```

```
                            # Extract cell data
                            cell = self._extract_cell_data(cell_block, response)
                            # Append cell data to cells list
                            cells.append(cell)
                    # Add cells to the table dictionary
                    table['Cells'] = cells
                    # Append the table to the tables list
                    tables.append(table)

                # Extract forms (key-value pairs)
                elif block_type == 'KEY_VALUE_SET':
                    key_value = {}  # Dictionary to hold key-value pair
                    # Check if block has entity types
                    if 'EntityTypes' in block:
                        # Extract keys
                        if 'KEY' in block['EntityTypes']:
                            key = ''  # String to hold the key
                            # Check if block has relationships
                            if 'Relationships' in block:
                                # Loop over each relationship
                                for relationship in block['Relationships']:
                                    # Check if the relationship is of type 'CHILD'
                                    if relationship['Type'] == 'CHILD':
                                        # Loop over each ID in the relationship
                                        for word_id in relationship['Ids']:
                                            # Find the block with the matching ID
                                            word = [b for b in response['Blocks'] if b['Id'] == word_id][0]
                                            # Append the text of the word to the key
                                            key += word.get('Text', '') + ' '
                            # Remove trailing whitespace and add the key to the key_value dictionary
                            key_value['Key'] = key.strip()
                        # Extract values
                        elif 'VALUE' in block['EntityTypes']:
                            value = ''  # String to hold the value
                            # Check if block has relationships
                            if 'Relationships' in block:
                                # Loop over each relationship
                                for relationship in block['Relationships']:
                                    # Check if the relationship is of type 'CHILD'
                                    if relationship['Type'] == 'CHILD':
                                        # Loop over each ID in the relationship
                                        for word_id in relationship['Ids']:
                                            # Find the block with the matching ID
                                            word = [b for b in response['Blocks'] if b['Id'] == word_id][0]
                                            # Append the text of the word to the value
                                            value += word.get('Text', '') + ' '
                            # Remove trailing whitespace and add the value to the key_value dictionary
                            key_value['Value'] = value.strip()

                    # Add the key-value pair to the forms list if it contains data
                    if key_value:
                        forms.append(key_value)

                # Extract signatures
                elif block_type == 'SELECTION_ELEMENT':
                    if 'SelectionStatus' in block:
                        if block['SelectionStatus'] == 'SELECTED':
                            # Add the bounding box of the signature to the signatures list
                            signatures.append(block['Geometry']['BoundingBox'])
```

```
            # Extract lines
            elif block_type == 'LINE':
                # Append a tuple of the line text and its bounding box top position
                lines.append((block['Text'], block['Geometry']['BoundingBox']['Top']))

            # Extract words
            elif block_type == 'WORD':
                # Append a tuple of the word text and its bounding box top position
                words.append((block['Text'], block['Geometry']['BoundingBox']['Top']))

    return tables, forms, signatures, lines, words

def _extract_cell_data(self, cell_block, response):
    """
    Helper method to extract cell data from a cell block.
    """
    # Initialize a dictionary to hold the cell data
    cell = {"RowIndex": cell_block['RowIndex'], "ColumnIndex": cell_block['ColumnIndex'], "Text": ""}
    # Check if the cell block has relationships
    if 'Relationships' in cell_block:
        # Loop over each relationship in the cell block
        for relationship in cell_block['Relationships']:
            # Check if the relationship is of type 'CHILD'
            if relationship['Type'] == 'CHILD':
                # Loop over each ID in the relationship
                for word_id in relationship['Ids']:
                    # Find the block with the matching ID
                    word_block = [b for b in response['Blocks'] if b['Id'] == word_id][0]
                    # Append the text of the word to the cell's text
                    cell['Text'] += word_block.get('Text', '') + ' '
    # Remove trailing whitespace from the cell's text
    cell['Text'] = cell['Text'].strip()
    return cell
```

The ‘extract_data’ method returns five lists: ‘tables’, ‘forms’, ‘signatures’, ‘lines’, and ‘words’.

- The **‘tables’** list contains dictionaries, each representing a table found in the document. Each dictionary has a 'Cells' key that maps to a list of cells. Each cell is represented by a dictionary containing the 'RowIndex', 'ColumnIndex', and 'Text' of the cell.
- The **‘forms’** list contains dictionaries where each dictionary represents a key-value pair detected in the document. The dictionary has 'Key' and 'Value' fields that store the respective texts.
- The **‘signatures’** list contains bounding box information for each detected signature in the document. Each bounding box is represented as a dictionary with 'Width', 'Height', 'Left', and 'Top' keys.

- The **‘lines’** list contains tuples, each representing a line in the document. Each tuple contains the text of the line and the top coordinate of the bounding box of the line. This coordinate can help maintain the order of lines.
- The **‘words’** list is similar to the **‘lines’** list but for individual words. Each tuple contains the text of the word and the top coordinate of the bounding box of the word.

In the end, the method returns the lists of tables, forms, signatures, lines, and words, which provides a structured and convenient way to handle the extracted information. This extraction and organization of data are crucial for subsequent data manipulation or export processes.

- **tables_to_excel():** this method takes the list of tables and a file name as parameters. It aims to save the extracted tables into an Excel file where each table is written to a separate sheet.

This method starts by opening an ExcelWriter object. For each table in the tables list, it first determines the size of the DataFrame that will be used to represent the table. It does this by finding the maximum row index and the maximum column index among the cells of the table, which are then used as the dimensions of the DataFrame. An empty DataFrame of the appropriate size is then created, with cell values initialized to an empty string.

Afterward, the method iterates through each cell in the table. It assigns the text of each cell to the corresponding location in the DataFrame, identified by the row index and the column index of the cell.

```python
def tables_to_excel(self, tables, file_name):
    """
    Save extracted tables to an Excel file.
    """
    # Open an ExcelWriter object
    with pd.ExcelWriter(file_name) as writer:
        # Loop over each table in the tables list
        for i, table_data in enumerate(tables):
            # Find the maximum row and column indices to determine the size of the DataFrame
            max_row_index = max(cell['RowIndex'] for cell in table_data['Cells'])
            max_col_index = max(cell['ColumnIndex'] for cell in table_data['Cells'])

            # Create an empty DataFrame with the appropriate size
            df = pd.DataFrame('', index=range(1, max_row_index + 1), columns=range(1, max_col_index + 1))

            # Fill the DataFrame with the cell data
            for cell in table_data['Cells']:
                df.at[cell['RowIndex'], cell['ColumnIndex']] = cell['Text']

            # Write the DataFrame to the Excel file
            df.to_excel(writer, sheet_name=f'Table {i}')
```

Finally, the DataFrame, which now represents the table, is written to a sheet in the Excel file. The sheet name is based on the order of the table in the tables list. Once all tables have been processed, the Excel file is saved, providing a structured and human-readable representation of the tables extracted from the document.

- **forms_to_df():** this method is designed to convert the extracted forms, which are stored in a list of dictionaries, into a pandas DataFrame for better visualization and manipulation.

This method works by iterating over each dictionary (representing a form) in the given list. It checks if the dictionary contains a 'Key' or a 'Value'. If a 'Key' is found, it is stored in the **‘current_key’** variable. If a 'Value' is found, it is added to the forms_dict dictionary under the **‘current_key’**.

The process continues until all dictionaries in the list have been processed. This way, the **‘forms_dict’** ends up containing all keys and their corresponding values extracted from the form.

Once all the forms have been processed, the method converts **‘forms_dict’** into a DataFrame. It then transposes the DataFrame to swap rows and columns, which can make it easier to view the data if there are a large number of keys.

The **‘reset_index’** function is then called on the DataFrame to convert the index (which contains the keys) into a regular column. Finally, the column names are renamed to 'Key' and 'Value', and the DataFrame is returned.

```python
def forms_to_df(self, forms):
    """
    Convert extracted forms to a DataFrame.
    """
    # Dictionary to hold the keys and values
    forms_dict = {}

    # Variable to hold the current key
    current_key = None

    # Iterate over each dictionary in the forms list
    for form in forms:
        # Check if 'Key' is in the dictionary
        if 'Key' in form:
            # Store the key in current_key variable
            current_key = form['Key']
        # Check if 'Value' is in the dictionary
        elif 'Value' in form:
            # Add the value to the forms_dict under the current key
            forms_dict[current_key] = form['Value']

    # Convert the forms_dict into a dataframe and transpose it for better view
    df = pd.DataFrame(forms_dict, index=[0]).T.reset_index()

    # Rename the columns to 'Key' and 'Value'
    df.columns = ['Key', 'Value']

    # Return the dataframe
    return df
```

This DataFrame can be particularly useful for further data analysis or manipulation, as pandas offers a wide range of functionalities for working with DataFrames.

- **process_document():** this method is the main orchestrator that ties together the functionalities of the other methods defined in the **'DocumentProcessor'** class to process a given document.

Here's a step-by-step breakdown of what this method does:

- It starts by calling the **'analyze_document'** method on the input document. This method fetches the document from the S3 bucket and uses the AWS Textract service to analyze the document, extracting text and other features (like tables, forms, and signatures). The output of this step is a response from the Textract service, which is stored in the **'ocr_responses'** attribute of the document.
- The method then calls **'extract_data'** on the **'ocr_responses'**. This method loops over the blocks in the Textract response and categorizes them based on their block type (tables, forms, signatures, lines, and words), extracting the necessary information from each.

- The output of the **'extract_data'** method, **'extracted_data'**, is a tuple containing five elements, each corresponding to a different type of data (tables, forms, signatures, lines, words).
- Each of these elements is then processed accordingly and assigned to the respective attribute of the document. Specifically:
    - The tables are directly assigned to the **'tables'** attribute of the document.
    - The forms are first converted into a pandas DataFrame using the **'forms_to_df'** method and then assigned to the **'forms'** attribute of the document.
    - The signatures are directly assigned to the **'signatures'** attribute of the document.
    - The lines and words are both processed by the **'group_and_order_text'** method (which we'll cover next), which groups and orders them before assigning to the **'lines'** and **'words'** attributes of the document respectively.
    - After all the above steps, the processed document (now enriched with the extracted data) is returned.

```python
def process_document(self, document):
    """
    Method to process a document.
    Process here means analyzing the document using Textract,
    and then extracting the useful information (tables, forms, signatures)
    from the response.
    """
    document.ocr_responses = self.analyze_document(document)  # Analyze the document
    extracted_data = self.extract_data(document.ocr_responses)  # Extract the data from the analysis response
    document.tables = extracted_data[0]  # Assign the extracted tables to the document instance
    document.forms = self.forms_to_df(extracted_data[1])  # Assign the extracted forms to the document instance
    document.signatures = extracted_data[2]  # Assign the extracted signatures to the document instance
    document.lines = self.group_and_order_text(extracted_data[3])  # Group and order the lines
    document.words = self.group_and_order_text(extracted_data[4])  # Group and order the words
    return document  # Return the document
```

In a nutshell, the **'process_document'** method coordinates the entire document processing pipeline, from analyzing the document using AWS Textract to extracting and organizing the relevant information from the Textract response.

- **group_and_order_text() :** this method is designed to organize the lines or words extracted from the document by their top position on the page. This is done to maintain the original order and structure of the text in the document.

Here's a step-by-step explanation of the method:

- The method takes a list of tuples (**'text_data'**) as input. Each tuple contains a string of text (a line or a word) and its top position on the page.
- The method starts by sorting the **'text_data'** list by the top position of the text. This is done using Python's built-in **'sorted'** function and providing a lambda function as the **'key'** argument to sort by the second element of the tuples.
- Next, the method initializes the list of groups with the first element of **'sorted_text_data'**. Each group is a list of tuples that are close together vertically on the page.
- The method then loops over the rest of the **'sorted_text_data'**, grouping together tuples that are close to each other in terms of their top position. This is determined by comparing the top position of the current tuple with the top position of the last tuple in the last group. If the difference is less than a specified threshold (0.01 in this case), the current tuple is added to the last group. If the difference is greater, a new group is started with the current tuple. The threshold can be adjusted as needed depending on the specific requirements of your application.
- After grouping the tuples, the method converts each group into a string by joining the words in each group with a space character. The result is a list of strings, with each string representing a line or a group of words in their original order in the document.
- The method finally returns the **'grouped_text'**.

```python
def group_and_order_text(self, text_data):
    """
    Group and order lines or words by their top position.
    """
    # Sort the text data by the top position
    sorted_text_data = sorted(text_data, key=lambda x: x[1])

    # Initialize the first group with the first element
    groups = [[sorted_text_data[0]]]

    # Group the elements
    for i in range(1, len(sorted_text_data)):...

    # Convert the groups to strings of text
    grouped_text = [' '.join([word for word, _ in group]) for group in groups]

    return grouped_text
```

In summary, **'group_and_order_text'** is a helper method that helps maintain the structure of the text in the document by grouping and ordering lines or words based on their top position on the page.

- **process_documents() :** this method is used for processing multiple documents.

Here's an explanation of how it works:

  - It starts by initializing an empty list **'processed_documents'** which will be used to store the processed documents.
  - It then loops through each **'document'** in the **'documents'** list.
  - For each **'document'**, it tries to process the document using the **'process_document'** method which performs the analysis using Textract, and then extracts useful information such as tables, forms, and signatures from the response. The processed document is then appended to the **'processed_documents'** list.
  - The **'try/except'** block is used to handle any **'ValueError'** exceptions that might be raised during the document processing. This is important because if a document has an unsupported file format, a **'ValueError'** is raised in the **'analyze_document'** method. By catching this error, the code can continue processing the remaining documents even if one document fails to process. The error message is printed to the console to provide information about why the document failed to process.
  - Finally, after all documents in the **'documents'** list have been processed, the method returns the **'processed_documents'** list.

```
def process_documents(self, documents):
    """
    Method to process multiple documents.
    """
    processed_documents = []  # List to store the processed documents
    for document in documents:  # For each document in the list
        try:
            processed_document = self.process_document(document)  # Try to process the document
            processed_documents.append(processed_document)  # If successful, append the processed document to
                                                            # the list
        except ValueError as e:  # If a ValueError is raised in process_document
            print(e)  # Print the error message
    return processed_documents  # Return the list of processed documents
```

This method ensures that a batch of documents can be processed all at once, handling any errors for individual documents without stopping the entire operation. This makes it very

useful when working with large sets of documents, allowing for efficient and streamlined processing.

4.6 Application of the OCR System

In this final part of my internship, I was able to successfully apply the OCR system that I had developed to process real documents. Below is the Python code I used to accomplish this:

```
processor = DocumentProcessor()

# Process a batch of documents
documents = processor.get_bucket_folder_content('orders/4969182449/')
processor.process_documents(documents)
```

This code initializes an instance of the DocumentProcessor class, and calls on it to process a batch of documents found in a specific folder within an S3 bucket. Each document is then processed through the pipeline, resulting in the extraction of useful data from each document.

Once the documents have been processed, the extracted data can be accessed and used as shown:

```
documents[0].lines
documents[0].forms
documents[0].signatures
documents[0].forms
documents[0].tables
```

These lines of code access the various data attributes of the first document in the processed list. These include the lines of text, forms, signatures, and tables extracted from the document. The .tables attribute in particular is noteworthy, as it represents the data extracted from tables in the document.

In addition to being accessible directly, the data extracted from the tables is also saved in Excel format. This allows for easy viewing, manipulation, and sharing of the data, as well as facilitating integration with other systems or processes.

The data extracted from the documents, whether it be lines of text, forms, signatures, or tables, can then be deployed and used to fill in the values on the platform. This highlights the practical value and applicability of the OCR system developed during my internship, showing how it can be used to automate data extraction and entry tasks, saving significant time and effort.

4.7 Conclusion :

Over the course of developing the OCR system during my internship, I have been deeply immersed in the practicalities of data science and cloud technology. My journey began with a focus on data preprocessing, where I found the importance of adapting to unforeseen challenges, such as the occasional failure of AWS Textract to process certain types of PDF files. Overcoming this obstacle through the conversion of PDF files to PNG format was a vital first step in the project.

Next, my exploration of AWS S3 buckets opened up the world of cloud storage to me. I learned that handling the magnitude of document data required in this project was feasible through a scalable and flexible cloud storage solution, like S3. The intricacies of using S3 buckets, such as uploading, downloading, and deleting files, as well as managing data access and usage, were a significant learning curve. Still, the practical knowledge gained was vital in handling the large data volumes that I was to manage in this project.

The AWS SDK for Python, boto3, introduced me to the possibility of interacting with AWS services programmatically. I was able to automate data preprocessing by integrating AWS services with the Python boto3 library. I wrote a Python script that could download PDF files from the S3 bucket, convert them into images, and re-upload them back into the bucket. Successfully manipulating AWS services with boto3 was a significant milestone in the project and a valuable addition to my data management skillset.

The actual development of the OCR system demanded a comprehensive understanding of Python's object-oriented programming. Building this system introduced me to the critical attributes of the Document and DocumentProcessor classes, which work in tandem to convert, process, and extract valuable data from documents. The application of methods like get_bucket_folder_content(), file_converter(), analyze_document(), extract_data(),

tables_to_excel() in unison, showed me how individual components of a system can be orchestrated to perform complex tasks.

Finally, the successful application of the OCR system to process real documents and extract data marked the culmination of my internship project. I could automate data extraction and entry tasks, saving significant time and effort, highlighting the practical value and applicability of the system developed during my internship.

The development and application of the OCR system was an enriching learning experience, where I could apply the theories of data science and cloud technology to solve a real-world problem. The knowledge and skills I acquired will serve as a robust foundation for my future work.

As I look forward, I can see numerous possibilities for refining and expanding upon the OCR system. Fine-tuning the model, improving the accuracy of data extraction, and leveraging Natural Language Processing (NLP) to extract specific data types are potential avenues for future work. Finally the next step in this project would be delving deeper into these suggestions for future improvements, ensuring that the project continues to evolve and contribute value beyond my internship.

General Conclusion :

The primary goal of my end-of-study internship was to leverage my data science knowledge in a real-world context by developing an Optical Character Recognition (OCR) system for a supply chain management startup. This system was aimed at automating the data extraction process from various documents, increasing efficiency, and reducing human errors. The journey from understanding the concept and diversity of OCR technology to building a complete, functional OCR system has been both challenging and rewarding.

Throughout this project, I employed a systematic approach. Beginning with a foundational understanding of OCR technology, I explored different types of OCR solutions, including cloud-based, on-premises, open-source, commercial, and deep learning-based OCR. These explorations were centered on a number of factors like the types of documents used within the startup, the capabilities of different OCR solutions, and their feasibility in the startup's context.

This evaluation process led to the selection of two potential OCR solutions - AWS Textract and Tesseract OCR with img2table - each with its own strengths. A detailed comparison of these technologies resulted in AWS Textract emerging as the more suitable choice due to its advanced features, ease of use, strong performance with complex documents, and seamless integration with the AWS ecosystem.

With AWS Textract, I embarked on the journey of building the OCR system. I began by converting PDF documents into PNG format, making them suitable for processing. I then navigated the functionalities of AWS S3 Buckets to address data storage needs. Next, I organized the documents in S3 Buckets to streamline data access and manipulation.

Following this, I familiarized myself with AWS SDK for Python - boto3, which allowed the Python application to interact with AWS services.

To handle the growing complexity of the OCR system, I transitioned to Object-Oriented Programming (OOP). Creating the Document class to represent each unique file and the DocumentProcessor class to manage document processing was pivotal to the successful conversion, processing, and extraction of data from documents.

This systematic approach culminated in the successful application of the developed OCR system to process real documents and extract essential data. This application demonstrated the effectiveness of the OCR system, with significant time savings and improved accuracy in data extraction and entry tasks.

The accomplishment of the objectives set out at the beginning of the internship attests to the value of the methodologies used and my growth as a data science student.

In conclusion, this internship has provided a remarkable opportunity to translate theoretical knowledge into a practical solution that addresses real business needs. It served as a robust platform for learning, innovation, and growth. I look forward to applying the skills and insights gained during this project to future data science endeavors.

Suggestions for future improvements and potential future work :

Looking ahead, the most significant potential enhancements for the OCR system are rooted in finetuning the system through training the model on a larger set of documents to improve OCR recognition. This could be achieved through the use of Natural Language Processing (NLP), allowing the system not only to extract data but also to understand the context of the documents and adjust the recognition accordingly. By doing so, the system could correct any misinterpreted data and improve its overall accuracy. Furthermore, NLP could be utilized to enhance the OCR system by identifying and extracting specific required values from the documents. This focus on refining the OCR system's performance through the combination of extensive training and contextual understanding opens up exciting prospects for future development.

However, this in-depth finetuning and the utilization of NLP were not achievable during the internship due to a couple of limiting factors. The internship was limited to only three

months, which didn't provide sufficient time for the preparation and implementation of these complex enhancements. Furthermore, the lack of adequate training data was also a major constraint. The preparation of training data is a time-intensive process requiring careful selection and annotation of documents. The limited timeframe of the internship didn't allow for this preparation. However, with these identified potential improvements and an understanding of the requirements for their implementation, the stage is set for future enhancements to the system.

Appendix A : Full Code & Recommendations

This document contains the complete final code of the Optical Character Recognition (OCR) system developed during my internship.

Important Note: AWS Textract is a paid service. I was able to use this service because the startup where I interned provided me with the credentials to use under their subscription to AWS services. Anyone wishing to use the same code will need to pay for the service. Additionally, the names of the buckets, folders, and region names should be changed according to the ones they work with.

```python
class Document:
    def __init__(self, path):
        """
        The constructor of the Document class, which initializes instance variables.
        """
        self.path = path  # The path of the document
        self.bucket_name = "Name of bucket containing the document is not specified yet!"  # The bucket name
        self.order_num = "Order number of document is not specified yet!"  # The order number
        self.name = "Name of document is not specified yet!"  # The document's name
        self.type = self.get_file_extension()  # The type of the document
        self.is_analysable = self.is_ocr_analysable()  # Checks whether the document can be analyzed with OCR

    def get_file_extension(self):
        """
        Method to get the file extension from the path.
        """
        _, extension = self.path.rsplit('.', 1)  # Split the path by '.' and get the last part
        return '.' + extension.lower()  # Return the extension with a '.' prefix and in lower case

    def is_ocr_analysable(self):
        """
        Method to check whether the document is analyzable with OCR.
        """
        if self.type == ".pdf":  # If the document is a PDF
            return False  # Return False
        elif self.type in [".jpg", ".jpeg", ".png"]:  # If the document is a JPG, JPEG, or PNG
            return True  # Return True
        else:
            # If the document is not any of the supported types, raise a ValueError
            raise ValueError("The imported document has an unsupported file format.")
```

```python
import boto3
from pdf2image import convert_from_bytes
import io
import pandas as pd

class DocumentProcessor:
    def __init__(self, region_name = 'eu-central-1'):
        """
        The constructor of the DocumentProcessor class.
        """
        self.session = boto3.Session(region_name=region_name)  # Create a session with the specified region
        self.s3_resource = self.session.resource('s3')  # Create an S3 resource from the session
        self.s3_client = self.session.client('s3')  # Create an S3 client from the session
        self.textract = self.session.client('textract')  # Create a Textract client from the session

    def get_bucket_folder_content(self, folder_path, bucket_name = 'bucket-zakariyae-trial'):
        """
        Method to get the content of a folder in an S3 bucket.
        """
        folder_name = folder_path.rstrip('/').split('/')[-1]  # Extract the folder name from the path

        # List the objects in the specified S3 bucket and folder
        response = self.s3_client.list_objects_v2(Bucket = bucket_name, Prefix = folder_path)

        documents = []  # List to store the documents
        if 'Contents' in response:  # If the response contains 'Contents'
            for object in response['Contents']:  # For each object in the response
                object_key = object['Key']  # Get the object key

                if object_key.endswith('/'):  # If the object key is a directory
                    continue  # Skip to the next iteration

                document = Document(object_key)  # Create a Document instance for the object
                document.order_num = folder_name  # Assign the folder name to the order number of the document


                # Extract the document name from the object key and assign it to the name of the document
                document_name = object_key.rsplit('/', 1)[-1].rsplit('.', 1)[0]
                document.name = document_name

                document.bucket_name = bucket_name  # Assign the bucket name to the bucket name of the document

                documents.append(document)  # Append the document to the list
        else:
            print('No objects found in the specified bucket and folder.')
        return documents  # Return the list of documents

    def file_converter(self, document):
        """
        Method to convert a PDF document to PNG images.
        """
        # Fetch the document from the S3 bucket
        response = self.s3_client.get_object(Bucket = document.bucket_name, Key = document.path)

        # Convert the document to images
        images = convert_from_bytes(response['Body'].read())

        return images  # Return the images

    def analyze_document(self, document):
        """
        Method to analyze a document using AWS Textract.
        """
        if document.type in [".jpg", ".jpeg", ".png"]:  # If the document is a JPG, JPEG, or PNG
            # Analyze the document using AWS Textract
            response = self.textract.analyze_document(
                Document={'S3Object': {'Bucket': document.bucket_name, 'Name': document.path}},
                FeatureTypes=["TABLES", "FORMS", "SIGNATURES"])
            return response  # Return the response

        elif document.type == ".pdf":  # If the document is a PDF
            png_files = self.file_converter(document)  # Convert the PDF to PNG images

            responses = []  # List to store the responses
            for png_file in png_files:  # For each PNG image
                byte_stream = io.BytesIO()  # Create a byte stream
                png_file.save(byte_stream, format='PNG')  # Save the PNG image to the byte stream
                byte_stream = byte_stream.getvalue()  # Get the byte data from the byte stream

                # Analyze the PNG image using AWS Textract
                response = self.textract.analyze_document(Document={'Bytes': byte_stream},
                                                          FeatureTypes=["TABLES", "FORMS", "SIGNATURES"])
                responses.append(response)  # Append the response to the list

            return responses  # Return the list of responses

        else:
            # If the document is not a supported type, raise a ValueError
            raise ValueError(f'The document "{document.name}" has an unsupported file format.')
```

```
    def extract_data(self, responses):
        """
        Extract data from the response of the AWS Textract API.
        """
        if isinstance(responses, dict):
            responses = [responses]  # If there is only one response, convert it to a list for the loop below

        tables = []  # List to hold extracted tables
        forms = []   # List to hold extracted form key-value pairs
        signatures = []  # List to hold extracted signatures
        lines = []  # List to hold extracted lines
        words = []  # List to hold extracted words

        # Loop over each response
        for response in responses:
            # Iterate through blocks in the response
            for block in response['Blocks']:
                block_type = block['BlockType']

                # Extract tables
                if block_type == 'TABLE':
                    table = {}  # Dictionary to hold table data
                    cells = []  # List to hold cell data

                    # Check if block has relationships
                    if 'Relationships' in block:
                        # Loop over each relationship
                        for relationship in block['Relationships']:
                            # Check if the relationship is of type 'CHILD'
                            if relationship['Type'] == 'CHILD':
                                # Loop over each ID in the relationship
                                for cell_id in relationship['Ids']:
                                    # Find the block with the matching ID
                                    cell_block = [b for b in response['Blocks'] if b['Id'] == cell_id][0]
                                    # Extract cell data
                                    cell = self._extract_cell_data(cell_block, response)
                                    # Append cell data to cells list
                                    cells.append(cell)
                        # Add cells to the table dictionary
                        table['Cells'] = cells
                        # Append the table to the tables list
                        tables.append(table)

                # Extract forms (key-value pairs)
                elif block_type == 'KEY_VALUE_SET':
                    key_value = {}  # Dictionary to hold key-value pair
                    # Check if block has entity types
                    if 'EntityTypes' in block:
                        # Extract keys
                        if 'KEY' in block['EntityTypes']:
                            key = ''  # String to hold the key
                            # Check if block has relationships
                            if 'Relationships' in block:
                                # Loop over each relationship
                                for relationship in block['Relationships']:
                                    # Check if the relationship is of type 'CHILD'
                                    if relationship['Type'] == 'CHILD':
                                        # Loop over each ID in the relationship
                                        for word_id in relationship['Ids']:
                                            # Find the block with the matching ID
                                            word = [b for b in response['Blocks'] if b['Id'] == word_id][0]
                                            # Append the text of the word to the key
                                            key += word.get('Text', '') + ' '
                            # Remove trailing whitespace and add the key to the key_value dictionary
                            key_value['Key'] = key.strip()

                        # Extract values
                        elif 'VALUE' in block['EntityTypes']:
                            value = ''  # String to hold the value
```

```
                        # Check if block has relationships
                        if 'Relationships' in block:
                            # Loop over each relationship
                            for relationship in block['Relationships']:
                                # Check if the relationship is of type 'CHILD'
                                if relationship['Type'] == 'CHILD':
                                    # Loop over each ID in the relationship
                                    for word_id in relationship['Ids']:
                                        # Find the block with the matching ID
                                        word = [b for b in response['Blocks'] if b['Id'] == word_id][0]
                                        # Append the text of the word to the value
                                        value += word.get('Text', '') + ' '
                        # Remove trailing whitespace and add the value to the key_value dictionary
                        key_value['Value'] = value.strip()

                    # Add the key-value pair to the forms list if it contains data
                    if key_value:
                        forms.append(key_value)
                # Extract signatures
                elif block_type == 'SELECTION_ELEMENT':
                    if 'SelectionStatus' in block:
                        if block['SelectionStatus'] == 'SELECTED':
                            # Add the bounding box of the signature to the signatures list
                            signatures.append(block['Geometry']['BoundingBox'])
                # Extract lines
                elif block_type == 'LINE':
                    # Append a tuple of the line text and its bounding box top position
                    lines.append((block['Text'], block['Geometry']['BoundingBox']['Top']))

                # Extract words
                elif block_type == 'WORD':
                    # Append a tuple of the word text and its bounding box top position
                    words.append((block['Text'], block['Geometry']['BoundingBox']['Top']))

        return tables, forms, signatures, lines, words
```

```
    def _extract_cell_data(self, cell_block, response):
        """
        Helper method to extract cell data from a cell block.
        """
        # Initialize a dictionary to hold the cell data
        cell = {"RowIndex": cell_block['RowIndex'], "ColumnIndex": cell_block['ColumnIndex'], "Text": ""}
        # Check if the cell block has relationships
        if 'Relationships' in cell_block:
            # Loop over each relationship in the cell block
            for relationship in cell_block['Relationships']:
                # Check if the relationship is of type 'CHILD'
                if relationship['Type'] == 'CHILD':
                    # Loop over each ID in the relationship
                    for word_id in relationship['Ids']:
                        # Find the block with the matching ID
                        word_block = [b for b in response['Blocks'] if b['Id'] == word_id][0]
                        # Append the text of the word to the cell's text
                        cell['Text'] += word_block.get('Text', '') + ' '
        # Remove trailing whitespace from the cell's text
        cell['Text'] = cell['Text'].strip()
        return cell

    def tables_to_excel(self, tables, file_name):
        """
        Save extracted tables to an Excel file.
        """
        # Open an ExcelWriter object
        with pd.ExcelWriter(file_name) as writer:
            # Loop over each table in the tables list
            for i, table_data in enumerate(tables):
                # Find the maximum row and column indices to determine the size of the DataFrame
                max_row_index = max(cell['RowIndex'] for cell in table_data['Cells'])
                max_col_index = max(cell['ColumnIndex'] for cell in table_data['Cells'])

                # Create an empty DataFrame with the appropriate size
                df = pd.DataFrame('', index=range(1, max_row_index + 1), columns=range(1, max_col_index + 1))

                # Fill the DataFrame with the cell data
                for cell in table_data['Cells']:
                    df.at[cell['RowIndex'], cell['ColumnIndex']] = cell['Text']

                # Write the DataFrame to the Excel file
                df.to_excel(writer, sheet_name=f'Table {i}')
```

```
    def forms_to_df(self, forms):
        """
        Convert extracted forms to a DataFrame.
        """
        # Dictionary to hold the keys and values
        forms_dict = {}

        # Variable to hold the current key
        current_key = None

        # Iterate over each dictionary in the forms list
        for form in forms:
            # Check if 'Key' is in the dictionary
            if 'Key' in form:
                # Store the key in current_key variable
                current_key = form['Key']
            # Check if 'Value' is in the dictionary
            elif 'Value' in form:
                # Add the value to the forms_dict under the current key
                forms_dict[current_key] = form['Value']

        # Convert the forms_dict into a dataframe and transpose it for better view
        df = pd.DataFrame(forms_dict, index=[0]).T.reset_index()

        # Rename the columns to 'Key' and 'Value'
        df.columns = ['Key', 'Value']

        # Return the dataframe
        return df

    def process_document(self, document):
        """
        Method to process a document.
        Process here means analyzing the document using Textract,
        and then extracting the useful information (tables, forms, signatures)
        from the response.
        """
        document.ocr_responses = self.analyze_document(document)  # Analyze the document
        extracted_data = self.extract_data(document.ocr_responses)  # Extract the data from the analysis response
        document.tables = extracted_data[0]  # Assign the extracted tables to the document instance
        document.forms = self.forms_to_df(extracted_data[1])  # Assign the extracted forms to the document instance
        document.signatures = extracted_data[2]  # Assign the extracted signatures to the document instance
        document.lines = self.group_and_order_text(extracted_data[3])  # Group and order the lines
        document.words = self.group_and_order_text(extracted_data[4])  # Group and order the words
        return document  # Return the document

    def process_documents(self, documents):
        """
        Method to process multiple documents.
        """
        processed_documents = []  # List to store the processed documents
        for document in documents:  # For each document in the list
            try:
                processed_document = self.process_document(document)  # Try to process the document
                processed_documents.append(processed_document)  # If successful, append the processed document to
                                                                # the list
            except ValueError as e:  # If a ValueError is raised in process_document
                print(e)  # Print the error message
        return processed_documents  # Return the list of processed documents

    def group_and_order_text(self, text_data):
        """
        Group and order lines or words by their top position.
        """
        # Sort the text data by the top position
        sorted_text_data = sorted(text_data, key=lambda x: x[1])

        # Initialize the first group with the first element
        groups = [[sorted_text_data[0]]]

        # Group the elements
        for i in range(1, len(sorted_text_data)):
            # If the top position of the current element is close to the last element of the last group, add it to the group
            if abs(sorted_text_data[i][1] - groups[-1][-1][1]) < 0.01:  # Adjust the threshold as needed
                groups[-1].append(sorted_text_data[i])
            # Otherwise, start a new group with the current element
            else:
                groups.append([sorted_text_data[i]])

        # Convert the groups to strings of text
        grouped_text = [' '.join([word for word, _ in group]) for group in groups]

        return grouped_text
```

To execute the code above, use the OCR System and access the extracted data for further processing :

```
processor = DocumentProcessor()

# Process a batch of documents
documents = processor.get_bucket_folder_content('orders/4969182449/')
processor.process_documents(documents)
```

```
#documents[0] is the first document in the folder of the order:4969182449

#accessing raw text extracted from the doc
documents[0].lines

#accessing forms extracted from the doc as a dataframe of two columns (Keys and Values)
documents[0].forms

#accessing signatures extracted from the doc
documents[0].signatures

#accessing tables extracted from the doc
documents[0].tables

#converting the tables extractaed to an excel format and saving them
processor.tables_to_excel(documents[0].tables, "DUM_table.xlsx")
```

You can find the entire code plus the result of its execution on some documents in my github repo under the name **'final_OCRsystem_code.ipynb'** in the following link : https://github.com/zakariyaeELKHALDI/PFE/blob/master/final_OCRsystem_code.ipynb

Also here's the link to the code for the trial of tesseract with img2table library : https://github.com/zakariyaeELKHALDI/PFE/blob/master/tesseract_trial.ipynb

PS : I might change the codes for future improvements to the OCR system or due to the confidentiality of the documents I used in the demo execution of the codes.

5 Appendix B : Bibliography

The following sources were used during the development of the OCR system for reference, information, and understanding:

Software

[1] Tesseract OCR [Software]. Available at: https://github.com/tesseract-ocr/tesseract [Accessed: May 2023].

Online Resources

2. [2] Tesseract OCR, Tesseract Documentation [Online]. Available at: https://tesseract-ocr.github.io/tessdoc/ [Accessed: May 2023].

[3] Anderson, P., Extracting Tables From Images in Python Made Easy(ier) [Online]. Available at: https://betterprogramming.pub/extracting-tables-from-images-in-python-made-easy-ier-3be959555f6f [Accessed: May 2023].

[4] Amazon Web Services, Inc., AWS SDK for Python (Boto3) [Online]. Available at: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html [Accessed: May & June 2023].

[5] Amazon Web Services, Inc., Amazon S3 [Online]. Available at: https://docs.aws.amazon.com/s3/index.html [Accessed: May & June 2023].

[6] Amazon Web Services, Inc., Amazon Textract Developer Guide [Online]. Available at: https://docs.aws.amazon.com/textract/latest/dg/what-is.html [Accessed: May & June 2023].

[7] Amazon Web Services, Inc., Amazon Textract Pricing [Online]. Available at: https://aws.amazon.com/textract/pricing/ [Accessed: May & June 2023].

[8] Amazon Web Services, Inc., Create a Bucket [Online]. Available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html [Accessed: May & June 2023].

[9] Amazon Web Services, Inc., Extract text and data with Amazon Textract [Online]. Available at: https://aws.amazon.com/getting-started/hands-on/extract-text-with-amazon-textract/?ref=gsrchandson [Accessed: May & June 2023].

[10] Amazon Web Services, Inc., How Amazon Textract Works [Online]. Available at: https://docs.aws.amazon.com/textract/latest/dg/how-it-works.html [Accessed: May & June 2023].

[11] Amazon Web Services, Inc., Textract Analyze Document [Online]. Available at: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/textract/client/analyze_document.html [Accessed: May & June 2023].