Amazon Textract is a service that enables text detection and analysis in various types of documents, including typed and handwritten text. It can extract text, forms, tables, and structured data from documents using simple API operations. With features like document analysis, expense processing, ID document analysis, and custom queries, Textract can handle a wide range of document processing tasks.

Common use cases include creating intelligent search indexes, natural language processing, data capture from different sources, automating form data capture, and document classification. Textract offers benefits such as easy integration into applications, scalable document analysis, and cost-effectiveness with pay-as-you-go pricing.

Textract supports both synchronous and asynchronous processing for single-page and multi-page documents, and its API operations have quotas that can be adjusted based on usage requirements. Overall, Textract simplifies document text detection and analysis tasks, making it accessible to developers without deep learning expertise.

Synchronous and Asynchronous:

The main difference between synchronous and asynchronous processing in the context of Amazon Textract lies in how the document analysis operations are performed and how the results are returned:

1. **Synchronous Processing:**
    - In synchronous processing, the document analysis operation is initiated by sending a request to the Textract API, and the API waits for the analysis to complete before returning the results.
    - This means that the client application waits for the analysis to finish and receives the results in real-time, within the same API call.
    - Synchronous processing is typically used for scenarios where low latency is critical, and the application can afford to wait for the analysis to complete before proceeding.
2. **Asynchronous Processing:**
    - In asynchronous processing, the document analysis operation is initiated by sending a request to the Textract API, but the API immediately returns a response indicating that the analysis has been started.
    - The analysis itself is performed asynchronously in the background by Textract, and the results are made available later.
    - The client application can periodically check for the status of the analysis or be notified via a callback mechanism when the analysis is complete.

- Asynchronous processing is commonly used for scenarios where the analysis of large documents or batches of documents may take a significant amount of time, and the application wants to offload the processing to Textract without blocking its own execution.

In summary, synchronous processing provides real-time results within the same API call, while asynchronous processing allows for offloading long-running document analysis tasks and provides results at a later time.

Amazon Textract provides several API types to perform various document processing tasks.

1. Detect Document Text:
    - This API detects text in a document, including both typed and handwritten text.
    - It returns the location (bounding box) and content of the detected text elements.
    - Useful for basic text detection tasks where extracting text content is the primary requirement.
2. Analyze Document:
    - The Analyze Document API extracts structured data, such as forms, tables, and key-value pairs, from documents.
    - It can identify different types of elements within a document, such as lines, words, tables, and forms.
    - This API is suitable for extracting structured data from various types of documents, such as invoices, receipts, and forms.
3. Analyze Expense:
    - This API is specifically designed for processing invoices and receipts.
    - It extracts key information from invoices, such as vendor name, invoice number, total amount, and line items.
    - Useful for automating invoice processing workflows and extracting financial data from documents.
4. Analyze ID:
    - The Analyze ID API is used for analyzing identification documents, such as driver's licenses and passports issued by the U.S. government.
    - It extracts information from ID documents, such as name, date of birth, address, and ID number.
    - Useful for identity verification and document authentication processes.

Demos:

1. Extract Texts from an image: (Console) (Synchronous)

- ☐ Create an s3 Bucket (TextractImage)
- ☐ Create a Lambda function (TextractImageFunction) (Select create a role with basic lambda permissions)
- ☐ Go to Configuration > Permissions > Click on the Execution roles and attach the following roles - AWSLambdaExecte & AmazonTextractFullAccess (use custom for security)
- ☐ Triggers > Add Trigger > S3 > Select the TextractImage bucket
- ☐ Event type > All object create
- ☐ Check Recursive invocation
- ☐ Go to Code and add the following and save and deploy:

```
import sys
import traceback
import logging
import json
import uuid
import boto3
from urllib.parse import unquote_plus

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def process_error() -> dict:
   ex_type, ex_value, ex_traceback = sys.exc_info()
   traceback_string = traceback.format_exception(ex_type, ex_value, ex_traceback)
   error_msg = json.dumps(
       {
           "errorType": ex_type.__name__,
           "errorMessage": str(ex_value),
           "stackTrace": traceback_string,
       }
   )
   return error_msg


def extract_text(response: dict, extract_by="LINE") -> list:
   text = []
```

```python
    for block in response["Blocks"]:
        if block["BlockType"] == extract_by:
            text.append(block["Text"])
    return text


def lambda_handler(event, context):
    textract = boto3.client("textract")
    s3 = boto3.client("s3")

    try:
        if "Records" in event:
            file_obj = event["Records"][0]
            bucketname = str(file_obj["s3"]["bucket"]["name"])
            filename = unquote_plus(str(file_obj["s3"]["object"]["key"]))

            logging.info(f"Bucket: {bucketname} ::: Key: {filename}")

            response = textract.detect_document_text(
                Document={
                    "S3Object": {
                        "Bucket": bucketname,
                        "Name": filename,
                    }
                }
            )
            logging.info(json.dumps(response))

            # change LINE by WORD if we want word level extraction
            raw_text = extract_text(response, extract_by="LINE")
            logging.info(raw_text)

            s3.put_object(
                Bucket=bucketname,
                Key=f"output/{filename.split('/')[-1]}_{uuid.uuid4().hex}.txt",
                Body=str("\n".join(raw_text)),
            )

            return {
                "statusCode": 200,
                "body": json.dumps("Document processed successfully!"),
            }
    except:
        error_msg = process_error()
        logger.error(error_msg)

    return {"statusCode": 500, "body": json.dumps("Error processing the document!")}
```

Upload a file and navigate to the Lambda monitoring and check the logs (cloudwatch) for the responses.

Using detect document text API, we get only Page, line and word blocks

2. **Key-Value Pairs and Tables extraction using Lambda function: (Synchronous)**

Update the lambda function with the following:

lambdafunction.py

```
import json
import boto3
from pprint import pprint
from parser import (
   extract_text,
   map_word_id,
   extract_table_info,
   get_key_map,
   get_value_map,
   get_kv_map,
)


def lambda_handler(event, context):
   textract = boto3.client("textract")
   if event:
       file_obj = event["Records"][0]
       bucketname = str(file_obj["s3"]["bucket"]["name"])
       filename = str(file_obj["s3"]["object"]["key"])

       print(f"Bucket: {bucketname} ::: Key: {filename}")
       response = textract.analyze_document(
           Document={
               "S3Object": {
                   "Bucket": bucketname,
                   "Name": filename,
               }
           },
           FeatureTypes=["FORMS", "TABLES"],
       )

       print(json.dumps(response))

       raw_text = extract_text(response, extract_by="LINE")
       word_map = map_word_id(response)
       table = extract_table_info(response, word_map)
       key_map = get_key_map(response, word_map)
       value_map = get_value_map(response, word_map)
       final_map = get_kv_map(key_map, value_map)

       print(json.dumps(table))
       print(json.dumps(final_map))
       print(raw_text)

   return {"statusCode": 200, "body": json.dumps("Thanks from Anupam)}
```

parser.py

```
"""
-*- coding: utf-8 -*-
========================
AWS Lambda
========================
Contributor: Chirag Rathod (Srce Cde)
========================
"""


import json
import uuid
```

```python
def extract_text(response, extract_by="WORD"):
   line_text = []
   for block in response["Blocks"]:
       if block["BlockType"] == extract_by:
           line_text.append(block["Text"])
   return line_text


def map_word_id(response):
   word_map = {}
   for block in response["Blocks"]:
       if block["BlockType"] == "WORD":
           word_map[block["Id"]] = block["Text"]
       if block["BlockType"] == "SELECTION_ELEMENT":
           word_map[block["Id"]] = block["SelectionStatus"]
   return word_map


def extract_table_info(response, word_map):
   row = []
   table = {}
   ri = 0
   flag = False

   for block in response["Blocks"]:
       if block["BlockType"] == "TABLE":
           key = f"table_{uuid.uuid4().hex}"
           table_n = +1
           temp_table = []

       if block["BlockType"] == "CELL":
           if block["RowIndex"] != ri:
               flag = True
               row = []
               ri = block["RowIndex"]

           if "Relationships" in block:
               for relation in block["Relationships"]:
                   if relation["Type"] == "CHILD":
                       row.append(" ".join([word_map[i] for i in relation["Ids"]]))
           else:
               row.append(" ")
```

```python
            if flag:
                temp_table.append(row)
                table[key] = temp_table
                flag = False
    return table


def get_key_map(response, word_map):
    key_map = {}
    for block in response["Blocks"]:
        if block["BlockType"] == "KEY_VALUE_SET" and "KEY" in block["EntityTypes"]:
            for relation in block["Relationships"]:
                if relation["Type"] == "VALUE":
                    value_id = relation["Ids"]
                if relation["Type"] == "CHILD":
                    v = " ".join([word_map[i] for i in relation["Ids"]])
                    key_map[v] = value_id
    return key_map


def get_value_map(response, word_map):
    value_map = {}
    for block in response["Blocks"]:
        if block["BlockType"] == "KEY_VALUE_SET" and "VALUE" in block["EntityTypes"]:
            if "Relationships" in block:
                for relation in block["Relationships"]:
                    if relation["Type"] == "CHILD":
                        v = " ".join([word_map[i] for i in relation["Ids"]])
                        value_map[block["Id"]] = v
            else:
                value_map[block["Id"]] = "VALUE_NOT_FOUND"

    return value_map


def get_kv_map(key_map, value_map):
    final_map = {}
    for i, j in key_map.items():
        final_map[i] = "".join(["".join(value_map[k]) for k in j])
    return final_map
```

3. Extract text from multi-page PDF & save it as CSV (Asynchronous)

- [ ] Create an S3 Bucket (Textract-async-process)
- [ ] Create 3 folders/directories (async-doc-text, teaxtract-output, csv)
- [ ] Create SNS Topic - Standard > textract-async-notification
- [ ] IAM > Roles > Create Roles (lambda_textract_async) > Select Services and select Lambda > Permissions - AWSLambdaExecute, AmazonTextractFullAccess
- [ ] Create another role (textract_sns-async) and select service as textract > Permission > AmazonTexractServiceRole & SNSFullAccess
- [ ] Go to Lambda > Additional Resources > Layers > create pandas_3_9_layers
- [ ] Upload the zip files in an s3 bucket and paste the link
- [ ] Create a Lambda Function: Permissions > Use existing role (lambda_textract_async)
- [ ] Create the following code file and deploy them (update the bucket names in the environment variables)

OUTPUT_BUCKET_NAME=textract_async_process
OUTPUT_S3_PREFIX=textract-output
SNS_TOPIC_ARN = arn of topic
SNS_ROLE_ARN = arn of role for sns

Lambda_function_async_text.py

```
import os
import json
import boto3
from urllib.parse import unquote_plus

OUTPUT_BUCKET_NAME = os.environ["OUTPUT_BUCKET_NAME"]
OUTPUT_S3_PREFIX = os.environ["OUTPUT_S3_PREFIX"]
SNS_TOPIC_ARN = os.environ["SNS_TOPIC_ARN"]
SNS_ROLE_ARN = os.environ["SNS_ROLE_ARN"]


def lambda_handler(event, context):

   textract = boto3.client("textract")
   if event:
       file_obj = event["Records"][0]
       bucketname = str(file_obj["s3"]["bucket"]["name"])
       filename = unquote_plus(str(file_obj["s3"]["object"]["key"]))

       print(f"Bucket: {bucketname} ::: Key: {filename}")
       response = textract.start_document_text_detection(
           DocumentLocation={"S3Object": {"Bucket": bucketname, "Name": filename}},
           OutputConfig={"S3Bucket": OUTPUT_BUCKET_NAME, "S3Prefix":
OUTPUT_S3_PREFIX},
           NotificationChannel={"SNSTopicArn": SNS_TOPIC_ARN, "RoleArn":
SNS_ROLE_ARN},
       )
       if response["ResponseMetadata"]["HTTPStatusCode"] == 200:
           return {"statusCode": 200, "body": json.dumps("Job created successfully!")}
       else:
           return {"statusCode": 500, "body": json.dumps("Job creation failed!")}
```

- ☐ Create trigger for s3 and select the bucket name and add the prefix - async-doc-text (this is where we upload the files)
- ☐ Create another Lambda function (extract-response-process)
- ☐ Select the existing policy under Permissions (lambda-textract-async), save it and deploy
- ☐ Scroll down to layers and select the panda layer

Lambda_async_resp_process.py

```
import os
import json
import boto3
import pandas as pd


def lambda_handler(event, context):

   BUCKET_NAME = os.environ["BUCKET_NAME"]
   PREFIX = os.environ["PREFIX"]

   job_id = json.loads(event["Records"][0]["Sns"]["Message"])["JobId"]

   page_lines = process_response(job_id)

   csv_key_name = f"{job_id}.csv"
   df = pd.DataFrame(page_lines.items())
   df.columns = ["PageNo", "Text"]
   df.to_csv(f"/tmp/{csv_key_name}", index=False)

   upload_to_s3(f"/tmp/{csv_key_name}", BUCKET_NAME, f"{PREFIX}/{csv_key_name}")
```

```python
    print(df)

    return {"statusCode": 200, "body": json.dumps("File uploaded successfully!")}


def upload_to_s3(filename, bucket, key):
    s3 = boto3.client("s3")
    s3.upload_file(Filename=filename, Bucket=bucket, Key=key)


def process_response(job_id):
    textract = boto3.client("textract")

    response = {}
    pages = []

    response = textract.get_document_text_detection(JobId=job_id)

    pages.append(response)

    nextToken = None
    if "NextToken" in response:
        nextToken = response["NextToken"]

    while nextToken:
        response = textract.get_document_text_detection(
            JobId=job_id, NextToken=nextToken
        )
        pages.append(response)
        nextToken = None
        if "NextToken" in response:
            nextToken = response["NextToken"]

    page_lines = {}
    for page in pages:
        for item in page["Blocks"]:
            if item["BlockType"] == "LINE":
                if item["Page"] in page_lines.keys():
                    page_lines[item["Page"]].append(item["Text"])
                else:
                    page_lines[item["Page"]] = []
    return page_lines
```

PREFIX=CSV
BUCKET_NAME=textract_async_process

- [ ] Create an SNS subscription > Protocol - lambda > endpoint - copy the arn of the textract-async-response
- [ ] To test - go to topic and publish a sample messsage. Go to the lambda monitoring (cloudwatch) and check
- [ ] Now, upload a multipage PDF file to test the setup (async-doc-text)

Increase the execution time for this function