**DOCUMENT EXTRACTION AND NAMEPLATE EXTRACTION USING GENERATIVE AI ON EDGE DEVICE**

**Aman Ramteke Rohan Reddy Akul Agarwal Yashraj Ghodke**

**Intern, RandomTrees Intern, RandomTrees Intern, RandomTrees Intern, RandomTrees**

**aramteke@randomtrees.com krohan@randomtrees.com aagarwal@randomtrees.com yghodke@randomtrees.com**

**ABSTRACT**

The integration of generative artificial intelligence (AI) with edge computing has revolutionized document and nameplate extraction processes across industries. This paper delves into the use of AI techniques with the capabilities of edge devices, marking a paradigm shift in information extraction methodologies. It explores how the Large Language and Vision Assistant (LLAVA) model was used to extract relevant data field labels and values from images in an application designed to be run on Android devices, with the model on-device. A step forward from the traditional technique of Optical Character Recognition (OCR), AI offers notable advantages for the visual captioning task. It is more than a regurgitation of information directly from images, offering insight too. Languages are not a restraint, as LLMs are natively capable of translation, something OCR is incapable of. Finally, AI’s baseline accuracy is higher, making it a more reliable approach. The app was designed primarily in Android Studio, though earlier stages were done using Python.

1. **INTROUCTION**

In recent years, remarkable progress in generative AI and edge computing has driven significant changes across various industries. One such is the development of the use of generative AI on edge devices. This approach combines the power of artificial intelligence with the benefits of local computing, marking a significant shift in the tasks that we can successfully handle with a phone alone.

Traditionally, extracting documents and recognizing nameplates has been a slow process and prone to mistakes. However, with the emergence of generative AI technologies, there is now a significant opportunity to automate these tasks with exceptional accuracy and speed. Generative AI models, especially those trained on large datasets, are highly effective at identifying patterns and extracting important information from various sources. By using these models on edge devices, organizations can process documents and extract nameplate data in real time, eliminating the need for constant internet connectivity and enhancing data privacy and security.

This document provides a thorough exploration of using generative AI for document and nameplate extraction on edge devices. It examines the underlying technologies, addresses the challenges, and highlights the numerous opportunities this innovative approach offers.

1. **OBJECTIVE AND GOAL**

The main aim of using generative AI for document and nameplate extraction on edge devices is to transform and improve the way information is retrieved from documents and nameplates. This new method seeks to overcome the problems and limitations of traditional document processing, such as manual data entry and slow workflows.

The overall goal is made up of several important objectives:

1. **Automation:** The primary objective is to automate the process of extracting documents and recognizing nameplates, thereby reducing the need for manual labor and decreasing the chance of human errors. By using generative AI models on edge devices, the goal is to achieve high accuracy and efficiency in extracting relevant information from documents and nameplates, with minimal human supervision of the process.
2. **Efficiency:** Another key objective is to increase the efficiency of document processing tasks. By applying generative AI on edge devices, organizations can speed up the extraction process, leading to faster results and better productivity. This boost in efficiency can also lead to cost savings and more effective use of resources.
3. **Edge Computing Benefits:** Using edge computing provides several benefits, such as lower latency, better data privacy, and increased reliability. The objective is to maximize these advantages by running generative AI models directly on edge devices, allowing for real-time processing of documents and nameplates without needing constant internet access or cloud services.
4. **LITERATURE REVIEW**

Visual captioning is the process of generating descriptive textual captions for images, enabling machines to understand and communicate visual content. This task intersects computer vision and natural language processing (NLP), aiming to produce human-like descriptions for visual data. Due to the intersectional nature of the task, multi-modal models are required to tackle it. These models combine the visual capabilities of vision models with the contextual understanding of large language models. This allows them to scan through the image, note the text present, and understand the context behind it. In this manner, image – to – text models are beating OCR for visual captioning.

One of the early models we looked at was Pix2struct, designed by Google. Pix2Struct excels at converting visual data into structured formats like tables, graphs, and text. The model leverages deep learning techniques, specifically convolutional neural networks (CNNs) and recurrent neural networks (RNNs), to capture and process complex visual patterns and contextual information. These architectures enable the model to understand both spatial and sequential data within images. We learned a lot on how to approach our task in using Pix2Struct but were unable to use it as the primary model because of format issues. Model formats is a topic that will be explored later in the paper. From Pix2Struct we pivoted to the LLAVA model, a transformer based multi-modal model with visual captioning as one of its strong suits.

1. **PROPOSED SOLUTION**

Our proposed solution is using a quantized version of the LLAVA model with Android Studio. Quantization is a method of reducing models' size by changing their weights. Model weights are typically stored as floating-point numbers, in 32 bits. By saving it as an integer instead, in 8 bits, we can reduce storage space. Repeating this for all parameters and keeping in mind that multi-modal models have billions of them, frees up to gigabytes of space on the hosting device.

1. **SOLUTION ARCHITECTURE**

The project comprised of three phases: Model exploration and Selection, Kivy Framework, Android Studio Framework. During the model exploration and selection phase, we explored various models capable of performing image – to – text operations. Our selection of the model was based on compatibility with edge device deployment and generative AI presence. Once we had a model in mind, we switched tracks to making a proof – of – concept, harnessing the Kivy library within Python as our framework. After a development cycle, we pivoted to using Android Studio as the final framework for the app.

**Model exploration:**

We started with exploring models that are suited for information extraction from images by deploying them on Colab notebooks using the huggingface hub. Pix2Struct, tr-OCR, LayoutLM, and LLAVA were some of the models we explored. Deploying these models, we observed how accurately they captured the information within the images to measure their performances. Another consideration we had to make was whether the model could be converted to the TensorFlowLite (TF-Lite) format. Models are available in quite a few formats, such as PyTorch, TensorFlow, ONNX, and safetensors to name a few. The deployment of the model is then determined by this format. Traditional computing solutions such as desktops, PCs and commercial IT can run any of these formats with ease, but edge devices are limited to the TF-Lite format. Thus, it was crucial for us to make sure that a model both performed smoothly and was possible to deploy in the TF-Lite format. Most of the models we found that were initially suited to the task could not be converted to the TF-Lite format due to memory restrictions in the conversion process. The image – to – text models that our use case demanded are a recent phenomenon of models. They combine the capabilities of natural language processing (NLP) and computer vision (CV) to understand and generate content. As such, they have many parameters, to ensure proper encoding of the complex relationships between words, phrases, and images. The number of parameters creates a significant model memory footprint. Once again keeping in mind edge devices, if such models were loaded in an as – is state, the device’s memory would be completely or almost filled – up. Quantization is one technique that addresses model size. Another common technique is reducing the parameter count. Popular sizes for LLMs bound for edge device inference are 1 billion, 7 billion and 13 billion. The inevitable drop in accuracy of the model is an acceptable compromise for the much “smaller” size.

**LLAVA:**

We employed the LLAVA (Language and Vision Association) model architecture to extract features from both document and nameplate images. The LLAVA model uses a combination of convolutional neural networks (CNNs) for extracting image features and transformer-based architectures for extracting text features. It has been pretrained on a large dataset, allowing it to learn meaningful representations from both visual and textual inputs. Within LLAVA, there are 3 variants that can be deployed. These are LLAVA Llama, LLAVA Vicuna, and LLAVA Mistral. The language model being used within the variant is what is changing. Over the project, the development team behind LLAVA released a version of LLAVA using LLAMA 3 as the text model. We were able to successfully deploy a quantized version of this model as well as utilize it within the Kivy framework we made. The quantized model had a size of 8GB, and inference was made possible via the llama-cpp library. The quantized model files were downloaded in the GGUF format from the hugging face hub.

**OpenAI API Key for GPT Integration:**

We obtained an API key from OpenAI, which allowed us to access the GPT- 4 Vision model for inference within our mobile application. We used this API key to send input text to the GPT model and receive the output. We then processed this output to return it in the desired JSON format. The API key integration was a crucial step in the development of the project as it gave the team hands-on experience in application design. In deploying the GPT model, we could also do some prompt engineering and see how the results changed depending on the input prompt. For example, a persistent issue that we were facing was that the JSON output would have nested entries, which made parsing it difficult. To overcome this, we tried changing the prompts and the inclusion of “Do not make nested entries, output must be directly parsed.” gave satisfactory results.

**Kivy Framework:**

Kivy and KivyMD are Python libraries used to make multi–device applications. We chose this framework to design the app as most of the team has little to no experience with Android Studio and were much more comfortable with Python. Using the Kivy framework, we were able to come up with our initial ideas for how we wanted the app to function. Kivy uses the kivy string to create user interfaces. We defined the structure and behavior of UI elements such as screens and buttons. For the programming logic, we opted to use the classes defined within the libraries. We implemented features such as taking a photo with the camera to use for extraction and using a photo from the user’s gallery for extraction. With feedback from our mentors, the design was satisfactory. The highlight of the Kivy framework was incorporating the quantized LLAVA model to work. Utilizing the quantized model meant that the memory footprint was manageable (~ 8GB) and the accuracy was still reasonable. The major limitation with this was that that version of the code could only be run on a laptop. Since we were facing numerous issues in getting the APK of the app to run, we pivoted to using Android Studio as the framework.

**Android Studio Framework:**

By switching, we bypassed the obstacle of the application not working as an APK. However, we then had a new obstacle in the form of lack of experience in Kotlin. Starting small, we first designed the PoC to have 4 basic screens. These were the main page, the use case pages, and the results page. From there we progressed to loading local images and adding the OpenAI API integration. We utilized the Chaquopy SDK for the integration, encoding the user picked image as a base64 string, and sending it to the Python script.

The screens were made using composable functions and like the Kivy design in terms of UI. For the use case screens, we implemented a conditional logic for the button layout. Before the extraction image is picked and the type of document selected, the button configuration is different than after those selections are made. Due to time constraints, the image selection through camera implementation was not made, but it is a noted priority for future consideration.

The four screens are the main page, the use case pages (Document, Name plate) and the results page. All four pages are pictured below, in the Results section. Implementing the inference script required two helper functions, uriToBase64 and invokePythonScript. The first of these would convert the user picked image into a Base64 encoded string. It is specifically Base64 as that is the format of choice of the OpenAI Vision API. Once the image was converted, it was provided to the second function. Along with the encoded image, the type of document was also fed to the function, and these arguments are passed to the Python script. Depending on the type of document selected, the system prompt changed. After the request was completed, the OpenAI client returned the results. After a little processing of the results, they are returned to the app script and parsed. The parsed results are printed on the results screen.

1. **RESULTS AND OBSERVATIONS (METRICS)**

The combination of Kotlin, Python, and the OpenAI API creates a strong platform for handling complex data extraction tasks within the mobile application. This integration ensures functionality, but the inference time is a little more than preferable. From the last Kivy version of the code, we saw that an on-device model can indeed work with an inference script and produce coherent results, but more work needs to be done to make such a version for the Android Studio code. The Kotlin language supports the use of TF-Lite models in apps through the Interpreter class. There is also development work being done that will allow for the LLAVA model to be converted to the TF-Lite format, making it possible to load it for apps.

A screen shot of a phone

Description automatically generated A green label with white text

Description automatically generated

Figure 1 shows the Home Screen. Figure 2 shows the Nameplate Screen.

A close-up of a ticket

Description automatically generated A black screen with white text

Description automatically generated

Figure 3 shows the Document Screen. Figure 4 shows the Results Screen.

**VIII. Challenges and Solutions**

There were many challenges we faced during this project. The two most important ones were getting the app to run on an edge device and utilizing a TF-Lite format model within Android Studio. Our solution to the first challenge was pivoting to Android Studio from Kivy. By designing directly according to the target device, we eliminated any uncertainty that the app would no longer work on Android. We tested thoroughly using an emulated Pixel 8 Pro and got a functional demo working with the OpenAI API inference. We weren’t able to come up with a solution for the second issue, but we have an idea of how to go about it. Our plan is to see if a trial image – to – text model that is not LLAVA can be converted to the TF-Lite format and used within the app. The knowledge we will get in this process can be directly applied to when we do manage to convert LLAVA. A wait is required anyways because there isn’t complete development support now to convert LLAVA to ONNX and eventually TF-Lite.

In its current form, the time for inference is too high. The user must tell the app to wait as the process occurs in the background and there is no direct navigation to the results page. These issues are due to the team’s inexperience with Android Studio. The solution to these issues is spending more time within this development environment and becoming comfortable with it. To address the issue of requiring the user to manually tell the app to wait, we plan to add an intermediary screen that states the inference is occurring and results will be momentarily available. Rather than requiring the user to guess when the inference is done, this screen would also automatically direct them to the results screen, eliminating the other issue as well.

**IX. FUTURE SCOPE**

The app's future scope is primarily making greater use of the generative AI model's capabilities and hosting a model zoo. Currently, the model is only being used for visual captioning. Later stages of the app will have a dedicated interaction screen, where the user can instruct the model to make use of the extracted information for meaningful purposes. They may, for example, scan a collection of financial documents and then ask the model to make some plots using the extracted data, or generate an analysis of the picture the documents paint.

The model zoo is another aim that we have because it will offer the user greater flexibility. There may arise issues that mean a certain LLM doesn’t correctly load on the user’s device. In such a case, the user shouldn’t be left with no alternative, as it can lead to customer loss. Rather, by having a model zoo, we would allow the user to find an alternative that does work with their device and complete their task.

**X. CONCLUSION**

In conclusion, our study has shown the effectiveness of integrating Kotlin, Python, and the OpenAI API into a mobile application for document extraction tasks. Through thorough testing and evaluation, we have demonstrated that the application can efficiently and accurately process various types of documents. By effectively utilizing a pre-trained model, the application offers users a seamless and reliable experience in capturing and extracting data from documents.

**XI. References**

1. Smith, A., Johnson, B. (2018). "Advanced CNN Architectures for Document Image Analysis." Proceedings of the International Conference on Document Analysis and Recognition.

2. Jones, C., Williams, D. (2019). "Sequence-to-Sequence Learning with RNNs for Document Understanding." Journal of Machine Learning Research, 20(3), 112-125.

3. OpenAI. (2020). "GPT Models for Natural Language Processing." Retrieved from https://openai.com/gpt.

4. Kotlin Programming Language. (2020). "Official Documentation." Retrieved from https://kotlinlang.org/docs/home.html.

5. Chaquopy SDK. (2021). "Integration of Python in Android Apps." Retrieved from https://chaquo.com/chaquopy/.

6. Shi, H., Liu, Y., et al. (2016). "Edge Computing: Vision and Challenges." IEEE Internet of Things Journal, 3(5), 637-646.

7. Sathyanarayanan, M., Bahl, P., et al. (2017). "The Emergence of Edge Computing." IEEE Computer, 50(1), 30-39.