COMP 4102 - Botanic Buddy (Group 74)

Adison Lampert, Andrew Guo, & Keaton Lee

April 11, 2024

Abstract

Our project created an open-source and user-friendly mobile app for the purpose of plant recognition. We give the foundational background for our methodology in creating our app before exploring the results of the model training and showing app usage examples. Further detail about our dataset, model training, and app code is provided. Finally, we explore why the MLE5Engine flag needs to be disabled on Apple devices with Neural Engine acceleration when using a converted PyTorch model in Core ML.

Contents

1 Introduction

In searching for our project topic, we discovered a lack of open-source resources for the accurate identification and cataloguing of common houseplants. Thus, our goal was to create a mobile application and public dataset that could leverage state-of-the-art computer vision techniques to identify common houseplants and provide useful information about each plant. We wanted to design a user-friendly application that would allow plant owners to identify their plants and access relevant information about them.

2 Background

2.1 The YoloV8 Model

Rather than constructing our model from scratch, we decided to opt for a transfer learning approach, leveraging a pre-existing model trained on a large, general dataset (COCO) and fine-tuning this model for our specific task. This strategy would significantly reduce the time and computational resources required for training. Our choice of model for this process was YOLOv8n, a compact variant of the YOLO (You Only Look Once) family known for its efficiency and accuracy in real-time object detection tasks. The decision to use this version of the model was informed by the findings of Muhammad Hussain as detailed in the paper "YOLOv1 to v8: Unveiling Each Variant–A Comprehensive Review of YOLO".[1]

This paper presents a systematic exploration of the evolution of YOLO variants, emphasizing the architectural composition and performance benchmarks that distinguish each iteration. It highlights the introduction of advanced features such as anchor-free detection and optimized path aggregation networks, culminating in YOLOv8's adaptations that prioritize computational efficiency without compromising accuracy. Performing at a mean average precision of 53.9% with a processing speed of 280 FPS, we selected YOLOv8 as our transfer learning model. More specifically, we selected the "nano" version of YOLOv8, which is the smallest and fastest variant in the series. While larger models offer a higher level of accuracy, the added computational demand makes them less suitable for applications that require real-time processing. In choosing the smallest variant, YOLOv8n, we could efficiently train our model while ensuring our application would remain lightweight and capable of delivering fast, accurate results on mobile devices.

2.2 Conversion of Deep Learning Models

In order to integrate a machine learning model into our iOS application, we decided to make use of CoreML. This presented a challenge as we had to determine how to convert our PyTorch models into this format. The paper "An Empirical Study of Challenges in Converting Deep Learning Models" by Moses Openja et al. provides an extensive analysis of this issue.[2] The authors conduct an empirical study assessing ONNX and CoreML for converting trained Deep Learning (DL) models, using popular DL frameworks (Keras and PyTorch) to train five widely used DL models on three well-known datasets. While performance metrics such as inference time and model size varied following these conversions, the study found the prediction accuracy of converted models to remain largely consistent with the originals. More specifically, for the conversions from PyTorch to CoreML, all models achieved at least the same level of accuracy as the originals, with some even showing an improvement of up to 0.03%. This supported our decision to use CoreML in our iOS application, confirming that this conversion would preserve, if not enhance, the prediction accuracy of our PyTorch models.

3 Methodology

3.1 Xcode

3.1.1 Swift

We decided to use Swift for the development of our app. Swift is a programming language created by Apple for developing iOS and Mac apps. Given that our goal was to construct an application for iOS devices, Swift proved to be an ideal choice in this regard. Along with its compatibility with Apple's ecosystem, Swift's performance and robust tooling would allow us to develop an efficient and reliable application.

3.1.2 Core ML

We made use of Core ML to integrate our machine learning model into our iOS application. Core ML is Apple's framework designed for running machine learning models on mobile iOS devices with hardware acceleration enabled through the Neural Engine. This integration is essential in allowing us to convert our PyTorch models into a format compatible with iOS.

3.2 Machine Learning

3.2.1 Datasets

Implementing plant detection for our machine learning model required each image in our dataset to be annotated with bounding boxes in addition to class labels. Bounding boxes are rectangular regions that surround an object within an image. A bounding box describes the spatial location of an object, the identity of such object, and, when used in object detection models, the detection confidence score. While we were able to find many comprehensive plant datasets available for our use, the majority of these datasets did not include bounding boxes. These collections instead consisted of images labelled with the species or category of the plant, tailored more towards classification tasks rather than object detection. This presented the following challenge: we would need to either augment these datasets with manually annotated bounding boxes—a time-consuming and resource-intensive task—or seek out specialized datasets that already include the necessary object detection annotations.

We settled on utilizing datasets from Roboflow, a platform known for its extensive collection of high-quality, pre-annotated datasets for computer vision tasks, including object detection. Roboflow streamlines the process of dataset generation, offering tailored formatting for different types of models. The final dataset included 50 classes containing a variety of classes representative of common house plants as shown in A.1. Thus, our model can detect and identify 50 different types of common house plants. This dataset was then split into training, validation, and testing sets. This partitioning is required for the model to learn from examples (training set), fine-tune its parameters against new images (validation set), and finally, test its overall accuracy and reliability on more new images (testing set). The stats for our final dataset are shown in figure 1.

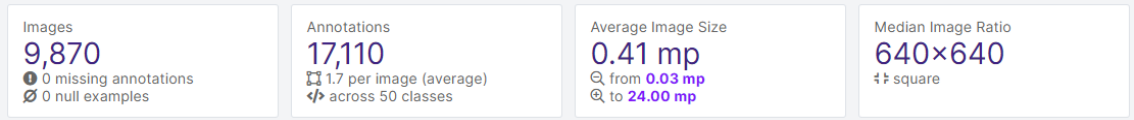

Figure 1: Botanic Buddy Dataset Stats

3.2.2 Integration

Exporting our trained model's weights and biases from PyTorch to Core ML presented several issues. In theory, converting to Core ML would optimize our PyTorch model for mobile computing applications by translating it into a format compatible with Apple's Neural Engine for hardware acceleration.

However, testing the converted Core ML model revealed some alarming discrepancies compared to the PyTorch model. Running the model on Apple mobile devices would result in noisy and inaccurate outputs with missing post-processing through our model's non-maxima suppression layer. Confusingly, testing on MacBooks would also differ. Sometimes, a MacBook would give similar erroneous outputs as the mobile devices while another would give results consistent with the original model.

The cause for these conflicting results lies in the previously mentioned Neural Engine that is featured on devices using Apple silicon, Apple's custom ARM-based SoCs used in mobile devices and the M-Series of MacBooks. Or rather, the Core ML model had no issues on some MacBooks because older MacBooks using x86-based Intel processors did not feature a Neural Engine. While devices that could run the model using their Neural Engine would create faulty outputs. This is revealed by a profiling of model performance in figure A.2. Thus, after disabling the experimental MLE5Engine flag found in newer versions of iOS, our model performed correctly for our app.

3.3 App Development

3.3.1 Core ML and Vision Frameworks

The Core ML framework is a set of tools provided by Apple designed to integrate trained models into iOS applications. The framework is available for Swift development. It optimizes model performance on device hardware, including CPUs, GPUs, and the Neural Engine.

The Vision framework analyzes and interprets images and videos on iOS devices. The framework provides a wide array of tooling that developers can use in their Swift applications to perform various computer vision tasks. Like the Core ML framework, it is integrated with the device's native capabilities which optimize its performance.

3.3.2 Object Detection

In our application, the Core ML framework and Vision framework work in tandem to identify household plants. While the application is running, an image is taken every 0.5 seconds. A Vision Image Request Handler is created for the image, which performs object detection using the Core ML model. If the model returns a Vision Recognized Object Observation, the image request returns a response containing the bounding box and the classification (label) of the object recognized. We are then able to display information about that plant based on its classification.

4 Results

4.1 Training

After creating our final dataset, Roboflow would allow us to export it in a format that is compatible with training the YOLOv8 model. The model was trained for 100 epochs which took approximately 24 hours to complete on a consumer GPU (RTX 4070) and results in the metrics shown in figure 2.

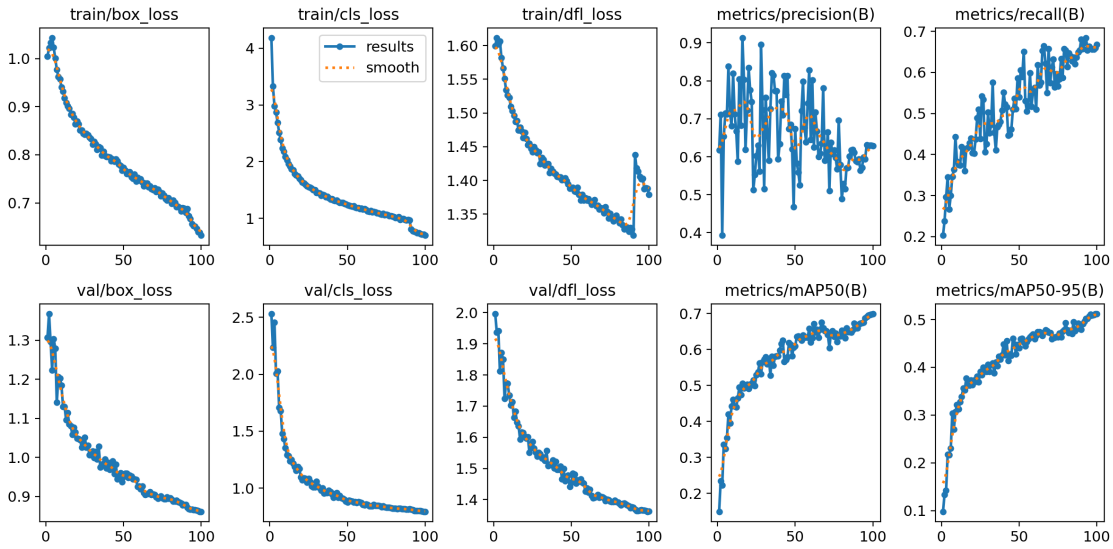

Figure 2: Model Performance Metrics

One of the dangers of running a high number of epochs on a relatively small dataset is overfitting which results in the model becoming unable to generalize data outside its training set. However, even at 100 epochs, we see the validation results are not increasing and the mean average precision curve is beginning to stabilize. This is indicative of a sufficiently trained model without overfitting.

4.2 App

As shown in Figure A.2, the app is able to consistently and quickly make predictions within a few milliseconds. However, without paying for an annual Apple developer subscription, the only way to use our app is to download the source code and compile, sign, and load it on your own Apple device. In real-time, the app identifies the plant it is shown and displays information about that plant species. Refer to the figures A.3, A.3, and A.4 for screenshots of the app in action.

5 Conclusion

Ultimately, we achieved our goal of creating an open-source and user-friendly iOS application that uses image recognition to identify and provide data about common houseplants. Some improvements that can be made to our project include adding more classes in conjunction with gathering more data for the dataset. In addition, further optimization of the model's hyperparameters and training is likely possible. Moreover, further enhancements may also be built in AR based on our existing tracking of plants in ARKit. Finally, through the Apple Developer Program, the app can be published publicly to the App Store with a professional account.

References

[1] J. Terven, D.-M. Cordova-Esparza, and J. Romero González, "A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas," *Machine Learning and Knowledge Extraction*, vol. 5, pp. 1680–1716, 11 2023.

[2] M. Openja, A. Nikanjam, A. H. Yahmed, F. Khomh, and Z. M. J. Jiang, "An empirical study of challenges in converting deep learning models," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 13–23.

A Figures

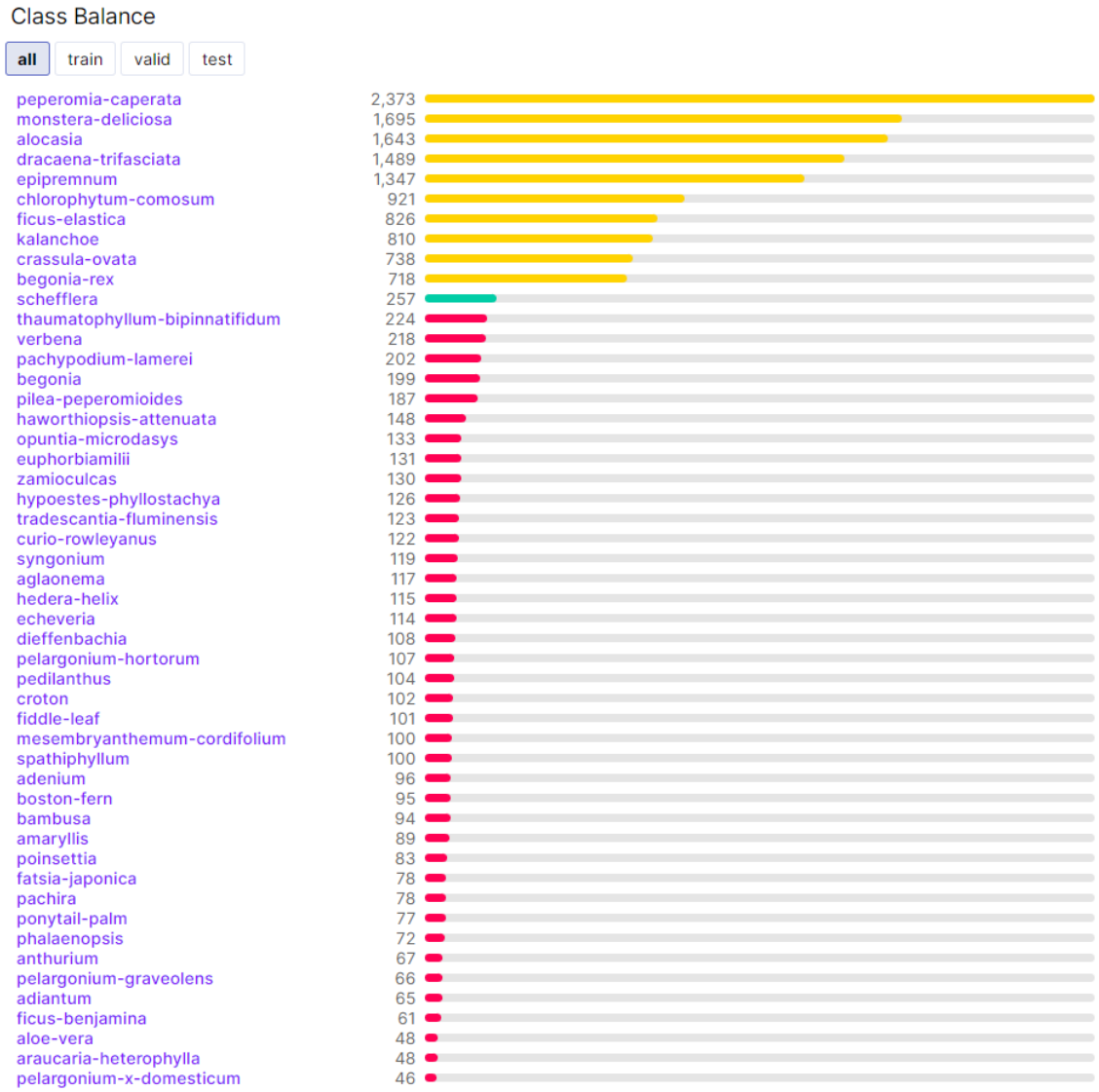

Figure A.1: Dataset Class Distribution

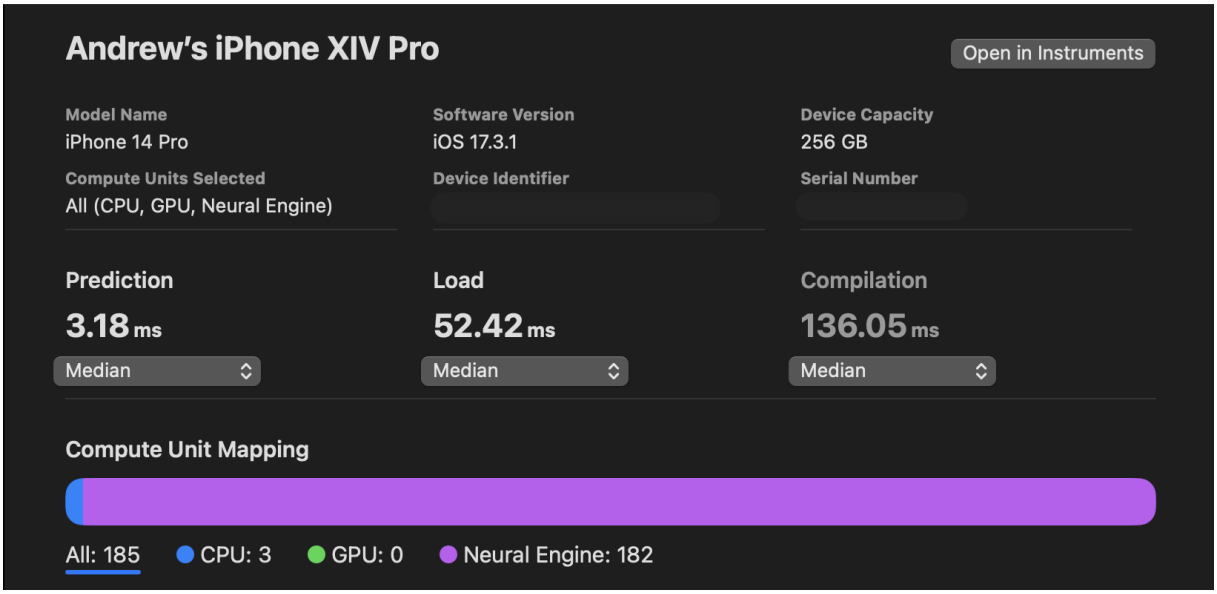

Figure A.2: Performance Profile on iPhone

Figure A.3: Screenshot of application identifying a Spathiphyllum plant.

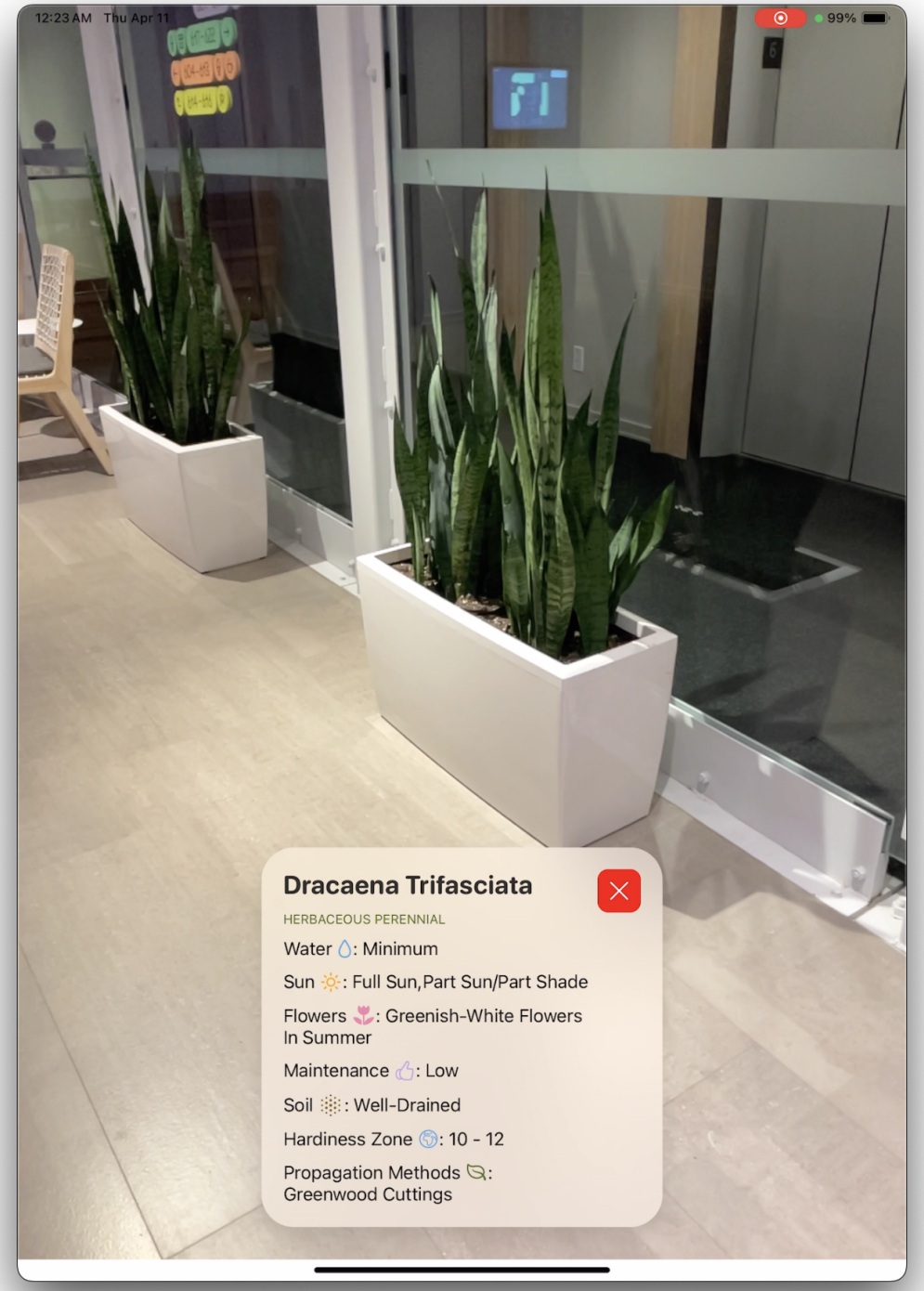

Figure A.3: Screenshot of application identifying a Dracaena Trifasciata plant.

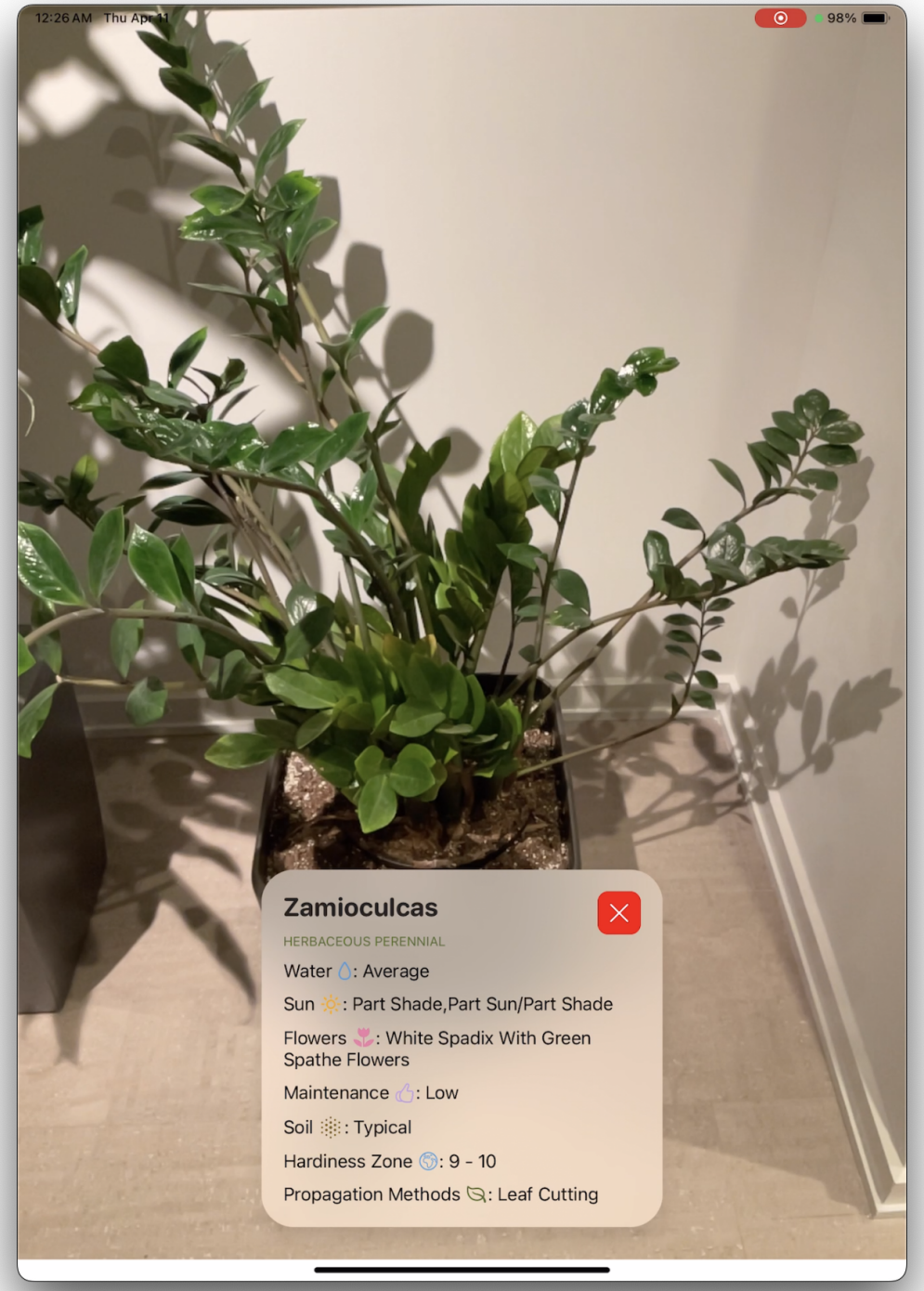

Figure A.4: Screenshot of application identifying a Zamioculcas plant.