**GYMNASIUM JANA KEPLERA**

Parléřova 2/118, 169 00 Prague 6


# Mobile Application for Plant Disease Recognition

Matura Project Assignment

Author: Matyáš Boháček

Class: 4.A

School Year: 2022/2023

Subject: Computer Science and IT

Supervisor: Ing. Šimon Schierreich

Prague, 2023


**GYMNASIUM JANA KEPLERA**
*Computer Science and IT*

# MATURA PROJECT ASSIGNMENT

*Student:* **Matyáš Boháček**

*Class:* **4. A**

*School year:* **2022/2023**

*Supervisor:* **Šimon Schierreich**

*Title:* **Mobile application for plant disease recognition**

*Instructions:*

**The main goal of this thesis is to develop a mobile application for Apple iOS devices that allows users to differentiate between healthy and diseased crops, primarily vegetables. The users of the application should be able to easily take a new photo or upload one from the camera roll. Supposing the image shows just a leaf with a solid background, the application will analyze it and intuitively present predictions (including the recognized leaf species and health status). If the leaf is found to be diseased, the application will briefly describe the disease, its severity, and additional references.**

*References:*

**[1] Hughes, David, and Marcel Salathé. "An open access repository of images on plant health to enable the development of mobile disease diagnostics."arXiv preprint arXiv:1511.08060 (2015).**

**[2] Liu, Charles Z., and S. Ramakrishnan. "Image Recognition: Progress, Trends and Challenges."New York: Nova Science Publishers (2020). ISBN: 978-1-53617-258-4**

**[3] Apple Inc. "ML Compute: Apple Developer Documentation." Available from: https://developer.apple.com/documentation/mlcompute**

*Repository URL:*

**https://github.com/matyasbohacek/gjk-maturitni-prace-inf-2022/**

*student* *supervisor*

*Prague September 26, 2022*

## Statement of Authorship

I hereby declare that I have completed this thesis independently and have used only the sources and literature listed in the bibliographic entries. I have no objection to the disclosure of this thesis in accordance with Act No. 121/2000 Coll. on copyright, copyright-related rights and amending certain acts (Copyright Act implemented in the Czech law, originally "autorský zákon"), as amended.

In Prague on March 24, 2023 Matyáš Boháček

## Acknowledgement

I would hereby like to thank everyone who helped shape the project and the attached paper to their current form, especially my advisor, Ing. Šimon Schierreich. Moreover, I would like to express gratitude to the representatives of all agriculture- and biology-focused internet portals, who kindly allowed their resources to be linked from the app. Namely, these include Royal Horticultural Society, The Old Farmer's Almanac, Bayer Vegetables Solutions, MSU Extension Integrated Pest Management, NC State Extension, Center of Invasive Species Research, PlantVillage. Last but not least, I would like to thank my rubber duck for hearing me out whenever I could not fix a bug.

# Abstrakt

Prudké nákazy rostlinnými chorobami mohou na malé a střední farmáře vytvářet nárazové, často kritické finanční škody, které ročně sahají až desetitisícům dolarů. Identifikace takových chorob přitom vyžaduje patřičnou expertizu – existuje totiž více než 10,000 odlišných druhů, což omezuje škálovatelnost jakýchkoliv manuálních řešení prevence. Rozhodující roli přitom paradoxně hraje právě čas – čím déle ujde nákaza bez povšimnutí, tím větší bude výsledná škoda pro farmáře. Právě proto se tato práce zabývá tvorbou automatizovaného systému na bázi umělé inteligence, který umí takové choroby rozpoznat a poskytnout návod pro jejich následnou léčbu. Práce nejprve popisuje tvorbu a natrénování modelu hlubokého učení, jenž umožňuje rozpoznávání na základě fotky listu, a dále sleduje jeho integraci do mobilní aplikace pro platformu iOS. Aplikace je otestována skupinou beta testerů a bezplatně dostupná ke stažení na App Store. Její zdrojový kód, včetně skriptů pro trénování klasifikátoru, je volně dostupný.



### Klíčová slova

rozpoznávání obrazu, nemoci rostlin, počítačové vidění, strojové učení, mobilní aplikace



# Abstract

For small- and mid-sized farmers, plant disease outbreaks can be a significant financial burden, costing tens of thousands of dollars in annual damage fees. With over 10,000 species, identifying and treating diseases requires expert personnel and can be challenging to achieve on a large scale. Time is of the essence — the longer an outbreak goes unnoticed, the more damage is done. Therefore, this work explores an automated, AI-powered system for crop disease recognition and treatment guidance. We trained a deep-learning model to identify diseases from photos of leaves and integrated it into an iOS app that works without an internet connection. We tested the solution with a group of beta testers and published it for free on the App Store. Its source code, including the classifier training scripts, are publicly available.



### Keywords

image recognition, plant diseases, computer vision, machine learning, mobile application

# Contents

# 1. Theory and Related Work

In the third century BC, farmers in the Chinese province of Henan revolutionized agriculture by employing the first ever tool [19]. It was likely a plow, whose share and moldboard, made of iron and shaped into a V, were mounted on wooden handles. Albeit tiny, this tool enabled farmers to produce more crops and, most notably, paved the way for the transformation of agriculture from manual, human-only labor to an industry at the forefront of scientific and technological progress.

As technology advanced, so did agriculture. State-of-the-art tools, such as steam- and gas-powered vehicles, replaced animals, and each step of the production chain eventually got automated, from seed planting to sorting and packaging. Nowadays, clanking machines made of microchips, sensors, and robot hands have replaced human labor. Perhaps dismal, this shift has ensured that the vast populations of today can be fed reliably, with the convenience of a grocery store down every other street.

The automation of these tasks was made possible because they were purely repetitive and did not change from crop to crop. The few parts of the pipeline, which are highly variable and require additional judgement, have nonetheless remained reserved for humans only, despite being relatively costly and unreliable on a large scale. One of these processes is plant health assessment.

With over 10,000 diseases [1], their identification and deterrence are critical for farmers to ensure consumer safety and brand damage prevention. Even when infected crops are stopped from being sent for sale, the internal damages may get astronomic. As the owner of a small family farm in the Czech Republic told us, the losses go to tens of thousands of dollars. Last year alone, they had to throw out over $16,000 worth of crops, which got infected. For small local farmers, such costs are devastating. "You really want to spot it and stop it at the very first infected crop so that it doesn't spread through the population," the farmer commented on their current strategy, which is completely manual. "That's almost never the case, though," he adds.

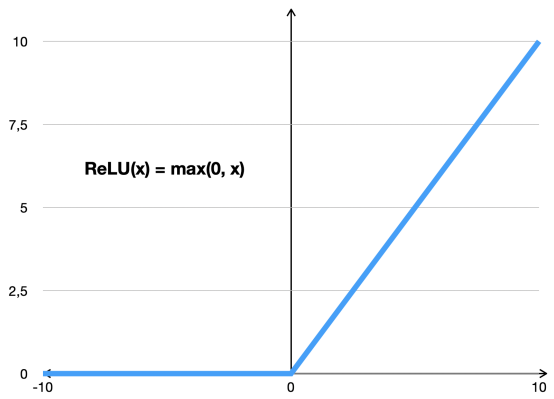



Figure 1.1: **ReLU.** A piece-wise linear activation function, which is generally considered the off-the-shelf activation function for CNNs, as it works well in most applications and outperforms Sigmoid or tanh. It is used after the convolution blocks in the herein-presented classifier architecture.

The issue lies close to home for amateur farmers, too. These inexperienced–yet all the more enthusiastic–individuals usually lack the necessary knowledge to identify potential diseases in their plants, and as a result, often fail to address the problem. "This happened to me two years ago during COVID when I started farming – my first batch of peppers died out completely," a young student from Prague, who grows some vegetables in her free time, shared with us. She agrees with the farmer that an easy guide to plant diseases would be useful.

Thanks to advancements in artificial intelligence and machine learning (ML), allowing for tackling highly-variable problems, the task of health assessment has, too, been automated in recent years. Both the farmer and the student could hence use one of the existing solutions that promise to detect plant diseases, such as *Plant Disease Identifier*[1] or *Plantix*[2], but would have to pay yearly fees of $20 and $1,260 to get the full set of capabilities, respectively. Given that our responders both mentioned that they would look for transparency and simplicity in such a solution, these services might not fit their needs: neither of the services includes a transparent methodology, information about training, or evaluation statistics. Lastly, both of these solutions perform the diagnostics on the server, requiring an internet connection. As connectivity may be poor in the fields, these internet-dependant solutions might not keep up with the day-to-day needs of the users.

Seeing the need for a solution that would help both of these demographics (professional and amateur farmers) grow their crops healthy, this final graduation project in the Computer Science class at Johannes Kepler Grammar School of Prague strives to address these issues. In this work, we present a novel, end-to-end solution for plant disease identification *Plant Doctor AI*. We pay special attention to make sure that the service (a) is free, (b) transparently lists the architecture and resources it uses, (c) shares its performance (benchmark evaluation), and (d) works completely offline.

## 1.1 Problem Statement and Project Goals

After conducting prospective user interviews with two professional and two amateur farmers, we formulated four principles to guide the project's direction: (a) free and accessible use, (b) architecture transparency, (c) public benchmarking results, and (d) on-device processing. With these in mind, we concluded that the best solution would be a mobile app that works in the field just as it does at home, with all processing taking place on the device.

[1] Available at `https://apps.apple.com/vc/app/plant-disease-identifier/id1550499292`, accessed on March 5th, 2023.

[2] Available at `https://play.google.com/store/apps/details?id=com.peat.GartenBank&hl=en_US&gl=US`, accessed on March 5th, 2023.

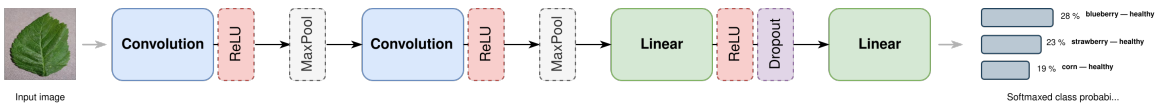



Figure 1.2: **Classifier architecture.** The RGB image at the input is pre-processed: resized to $150 \times 150$ pixels, normalized, and converted into a Tensor. Blocks with horizontal captions denote learnable modules of the neural network, whereas vertical blocks represent matrix operations. At the output, sotmax yields a likelihood percentage for each of the known classes. Generated using *Diagrams*, `https://app.diagrams.net/`.

For the time being, we focus on developing an iOS app, as the platform currently offers a better toolkit for on-device ML inference [3]. However, we recognize that this process could also be replicated for Android, and we touch on this possibility in Subsection 2.2.5.

The primary function of the app is disease recognition, which we approach as an image classification problem. Each class corresponds to a specific species-disease pairing. While the pipeline could be separated in two steps (i.e., first classifying the plant's species and then its health status), we believe it would significantly affect the final performance, since identical diseases take different visual forms on various plants. As the illnesses evince most clearly and distinctively on the leaves [15], the recognition is performed on leaf images only. This also reduces the dimensionality of different photo settings and contexts, which the model would otherwise have to learn.

The user should be able to take the image directly in the app or upload it from the gallery. If the model deems the crop infected, the app should present basic information about the diagnosis and include links to external expert databases.

## 1.2 Image Recognition Methods

The fields of ML and computer vision, which use computational methods to analyze digital imagery, emerged in the second half of the 20th century. Initially, image recognition was tackled with simplistic statistical methods (e.g., Piecewise Linear Discriminant Function or Min-Max Decision Rule [16, 8, 11]) and architectures such as Hidden Markov Models [4, 10, 18], which could only learn a few discriminating features and struggled to perform reliably outside of the experimental domain or in tasks with a larger number of classes.

However, the landscape changed around 2010 with the advent of deep learning, widespread access to accelerated computing (Graphical Processing Units, GPUs), and large datasets. Neural networks – inspired by the human brain – were constructed as layers of interconnected neurons through which

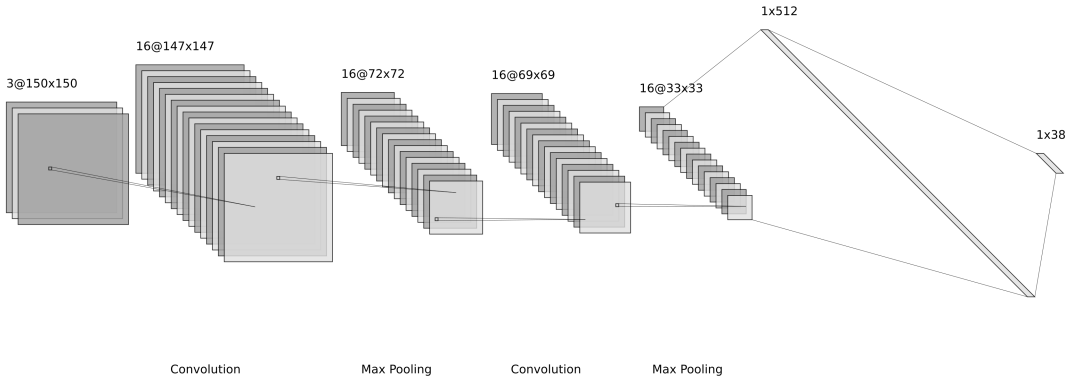



Figure 1.3: **Classifier data flow.** Shape of the processed data changes as it progresses through different modules of the neural network. A three-channel (RGB) image of $150 \times 150$ is first expanded using convolution, only to be later reduced into a vector. In the output vector (of shape $1 \times 38$), the value at each dimension directly corresponds to a class. Generated using *NN-SVG*, `http://alexlenail.me/NN-SVG/LeNet.html`.

input data was propagated. Each neuron held optimized parameters, and gradient descent was used to adjust these parameters through repetitive training with example data and annotations until the model converged. With representative data and proper training, these models could finally analyze complex problems and generalize to new examples. Their scope grew beyond image recognition to tasks such as image segmentation, captioning, and text-driven generation.

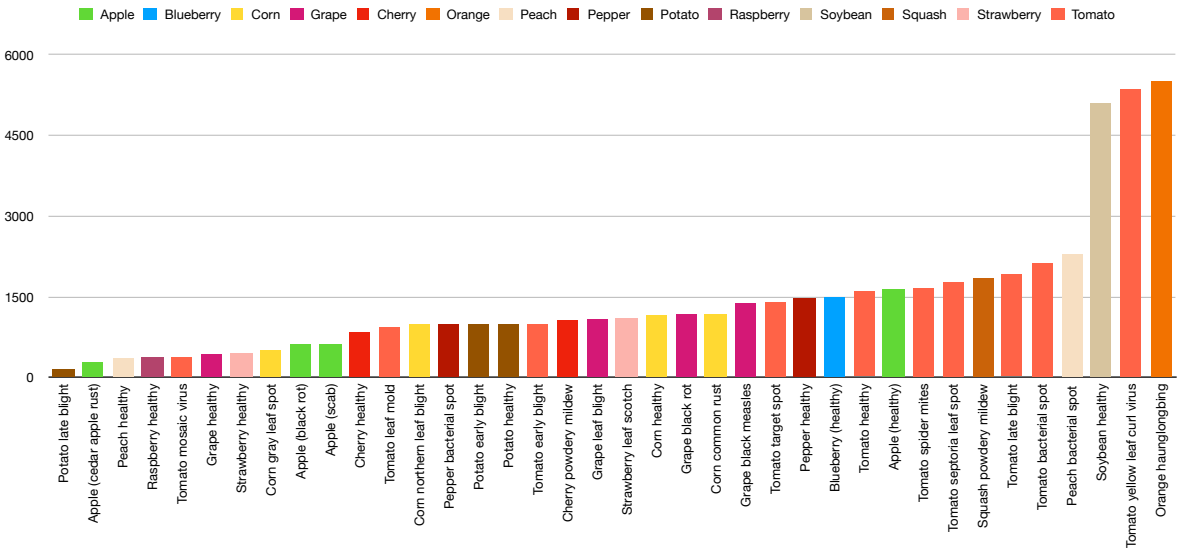



Figure 1.4: **PlantVillage granular class distribution.** All classes, as annotated in the dataset, are included. Since each image holds one class only, the total number of items herein equals to the number of images in the dataset.

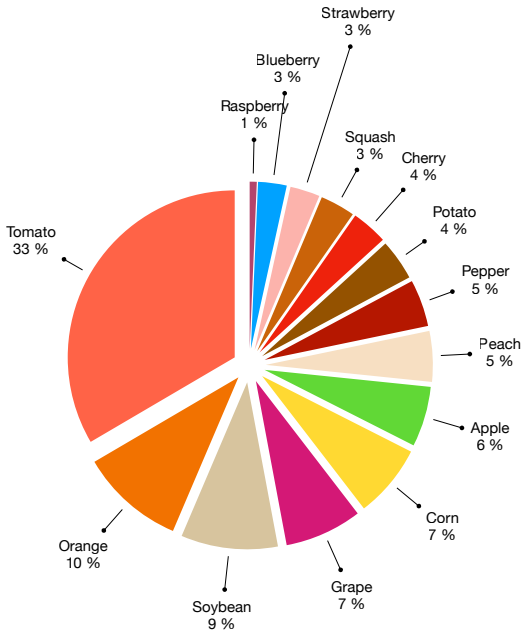



Figure 1.5: **PlantVillage joined class distribution.** Statistics of global plant species are reported, regardless of their diagnosis. These were obtained by joining the subclasses (e.g., 'Tomato' includes classes 'Tomato healthy', 'Tomato yellow leaf curl virus', 'Tomato spider mites', etc.).

Over time, many architectures of neural networks emerged, including Convolutional Neural Networks (CNNs) [9], Recurrent Neural Networks (RNNs) [13], and Long Short-Term Memory networks (LSTMs) [12]. ImageNet [7], with over 14 million images spanning 21 thousand classes, became the standard benchmark for image recognition. The latest state-of-the-art models have achieved an accuracy of 91.1% on this benchmark.

However, achieving high accuracy often requires significant computational power and pre-training on multiple databases. This may be desirable for fine-grained, high-stake applications (e.g., medical analysis or security systems), but for applications where speed is critical, including ours, simpler models are necessary. Additionally, as libraries for model training and on-device inference differ (requiring format conversion), complex architectures with large backbones might not be compatible.

Thus, we disregard the state-of-the-art architectures and instead use a plain CNN, whose module succession is depicted in Figure 1.2. It contains 2 convolution layers, followed by max-pooling, and a multi-layer perceptron (MLP) with 2 linear layers on top. The final linear layers guide each class's latent representation into a single digit and soft-max converts it into a likelihood (in percent). After each convolution layer and the first linear layer, the ReLU activation function is used. It is generally considered the most flexible activation for CNNs, proving effective in most use cases. Shown in Figure 1.1 is the ReLU activation function. The data shapes (as changed after each layer) are depicted in Figure 1.3.

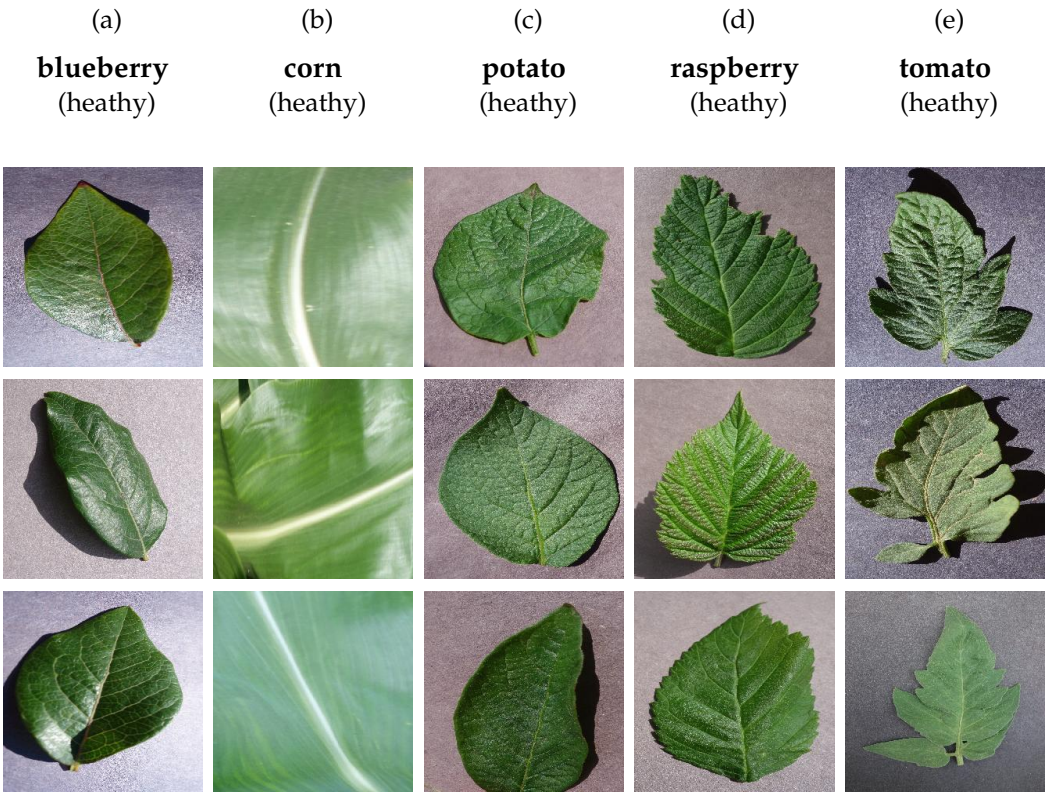



Figure 1.6: **Healthy classes of PlantVillage.** In each column — corresponding to a different class of healthy leaves — three randomly selected example images are shown.

## 1.3 Datasets

To train our model, we rely on the PlantVillage dataset [14]. This dataset was collected by research stations associated with the Land Grant Universities Program in the United States, including Penn State and Cornell.

The dataset comprises 54,309 RGB images. Each is $256 \times 256$ pixels large and comes with annotations indicating the crop species and health status of the depicted leaf. The photos show leaves across 14 different crop species: Apple, Blueberry, Cherry, Corn, Grape, Orange, Peach, (Bell) Pepper, Potato, Raspberry, Soybean, Squash, Strawberry, and Tomato. Apart from perfectly healthy ones, the leaves may exhibit one of the many fungal, bacterial, mold, oomycete, viral, and mite-caused diseases recognized by the dataset. All in all, the dataset has 38 distinct classes. Furthermore, each leaf was shot $4-7$ times from different angles and with diverse backgrounds.

The distribution of instances among the different classes is highly skewed, as shown in Figure 1.4. Some categories, such as *Potato Late Blight* or *Cedar Apple Rust*, contain as few as 152 instances, while others, such as *Tomato Yellow Leaf Curl Virus* or *Orange Haunglongbing*, hold over 5,000 images each. Figure 1.5 provides a visualization of the distribution with classes joined by plant specimen. This figure illustrates the extreme imbalance present in the data, with tomato images making up

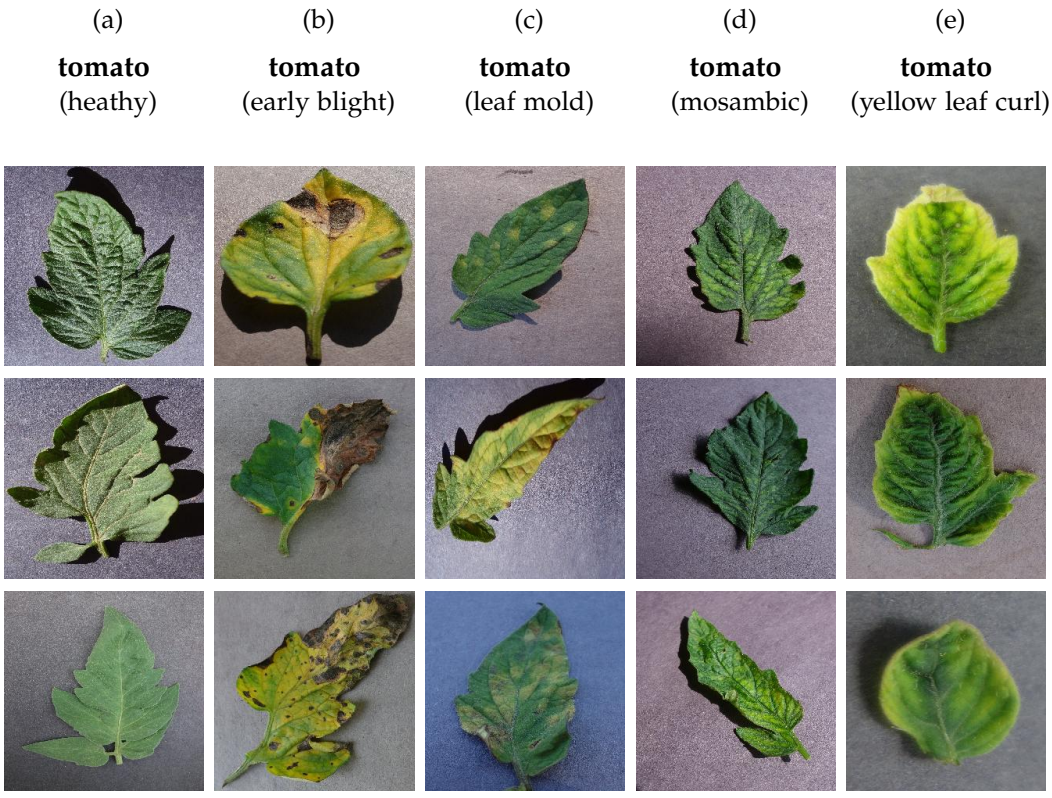



Figure 1.7: **Sub-classes of tomato within PlantVillage.** In each column — corresponding to a sub-class of tomato leaves — three randomly selected example images are shown. Column (a) includes healthy leaves; columns (b-e) comprise leaves of different diseases. Classes displayed in columns (d) and (e) correspond to virus names (i.e., *mosambic* virus and *yellow leaf curl* virus).

| Architecture | Accuracy |
|---|---|
| **GoogleNet** (Mohanty et al., 2016) [17] | 99.35 |
| **DenseNet-121** (Too et al., 2019) [21] | 99.75 |
| **MobileNet-Beta** (Chen et al., 2020) [6] | 99.85 |
| **EfficientNet** (Atila et al., 2021) [2] | 99.91 |
| **EfficientNet Ensemble** (Bruno et al., 2022) [5] | 100.00 |

Table 1.1: **PlantVillage Benchmarks.** Performance of contemporary architectures in the literature. Note that each work selects the testing set differently (usually as 10% or 20% of the dataset, but with different seeds and distribution policy), which scrutinizes the possibility of comparing these definitively. The top-1 macro accuracy is reported. Most of these works do not include an average repetition of this experiment.

one-third of the dataset and strawberry images representing only 1%.

The skewed distribution of instances impacts the prevalence of different classes in the final model's predictions. This imbalance may be beneficial in some cases, such as when a disease is rare or widespread. However, it is unclear whether the dataset's authors intentionally created this imbalance, as they did not discuss it in the original publication.

Shown in Figure 1.6 is a representative example of 5 healthy leaf classes. The shots are close-up and usually display the leaf center-aligned, with its stem down. The background is solid and typically gray. Albeit very similar at first sight, the leaves of one class evince distinctive patterns in shape and color hue.

Shown in Figure 1.7 are representative images from 5 different tomato disease classes. This time, leaves within the same class no longer hold generally homogeneous, unique features. Consider columns (b) and (e), showing images of *Tomato Early Blight* and *Tomato Yellow Leaf Curl Virus*, respectively. While it seems fair to assume that a critical feature in recognizing the latter is spotting yellow-colored edges of the leaf, triggering the class solely based on such observation is incorrect, as yellow edges occur with different classes, such as the one depicted in column (b), too. Overall, various diseases of the same crop look visually similar. Therefore, modeling a combination of different local features is required for a successful prediction.

Table 1.1 summarizes the state-of-the-art performance on this dataset, with 5 best-performing approaches. All of these methods reach a near-perfect performance of over 99% top-1 accuracy; the best model, EfficientNet Ensemble [5], even recognized all testing instances correctly. However, such a performance is likely a result of over-fitting, which may adversely worsen the in-the-wild performance.

# 2. Implementation

Herein, we justify the design decisions and rationale behind employed technologies, describe the course of the development, and present the constituent results. The development of our project was divided into two primary efforts: Image Classification Pipeline and Mobile Application. These comprised the following development substages, which we observed successively:

1. Image Classification Pipeline – engineering and training,
2. Image Classification Pipeline – conversion to a mobile-supported format,
3. Mobile Application – design,
4. Mobile Application – development,
5. Mobile Application – distribution.

## 2.1 Image Classification Pipeline

The first step was to train the ML model for plant disease image classification (hereafter referred to as the Model), which we did in a separate environment from the mobile app.

The Model was implemented in *Python* 3.7[1], current language of choice in the ML community, with the largest support for highly specific modules and functionalities. Other options like *Julia*[2] or *GoLang*[3] lack well-established libraries for building CNNs and do not support converting them to widely used formats, which is necessary for integrating the Model into a mobile application later on.

### 2.1.1 Implementation Details

We constructed our Model using *PyTorch*[4] and *NumPy*[5]. We preferred it over *TensorFlow*, which has lost most of its user base recently, possibly leading to less support across the ecosystem in the future[20]. However, we acknowledge that the simple architecture of our Model could be implemented in *TensorFlow* as well. We also used *Weights and Biases*[6], a web-based service, to internally track our ML experiments. It is free of charge, unlike alternatives such as *neptune.ai*[7] or *Comet*[8]. While *TensorBoard*[9] is also free, it is primarily suited for *TensorFlow* pipelines.

To train the Model, we randomly split the PlantVillage dataset into training (80%) and testing (20%) subsets, while fixing the seed to compare different features' impact on performance across runs. We

[1] Open-source software available from `https://www.python.org/downloads/release/python-37/`, accessed on December 16th, 2022

[2] Open-source software available from `https://julialang.org/`, accessed on December 16th, 2022

[3] Open-source software available from `https://go.dev/`, accessed on December 16th, 2022

[4] Available at `https://pypi.org/project/torch/`, accessed on February 17th, 2023.

[5] Available at `https://pypi.org/project/numpy/`, accessed on February 17th, 2023.

[6] Available at `https://www.wandb.com/`, accessed on February 17th, 2023.

[7] Available at `https://neptune.ai/`, accessed on March 5th, 2023.

[8] Available at `https://www.comet.com/`, accessed on March 5th, 2023.

[9] Available at `https://www.tensorflow.org/tensorboard/`, accessed on March 5th, 2023.

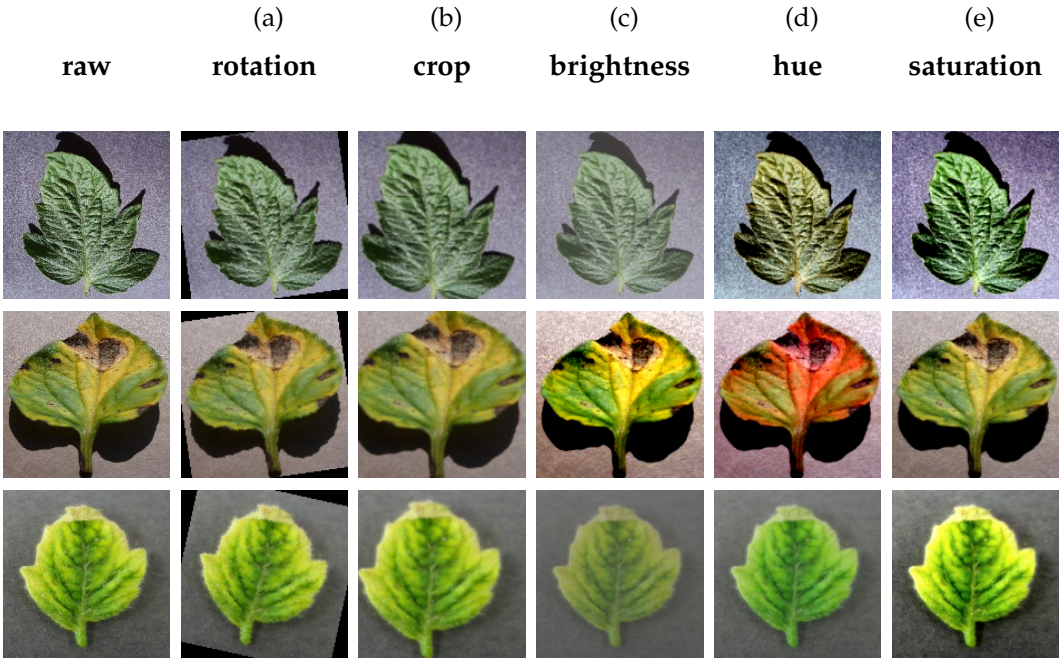



Figure 2.1: **Augmentations.** In each column — corresponding to an augmentation technique — three randomly selected example images, with the respective augmentation applied, are shown. Column (a) includes raw images, just as they appear in the original dataset, columns (b-e) comprise altered images. Note that we often employ more than just a single augmentation — please refer to Section 2.1.3 for details.

utilized the SGD optimizer[10] without a scheduler and Cross-entropy loss[11], as recommended for non-complex image classification tasks by *PyTorch*'s documentation. The Model was trained for 60 epochs, each time from random-distribution starting weights.

### 2.1.2 Pre-processing and Augmentations

In our Image Classification Pipeline, each input photo is first resized to $150 \times 150$ pixels and normalized to the range of $[-1; 1]$. To prevent over-fitting, we employ the following augmentations, whose representative examples are shown in Figure 2.1:

(a) In-plane rotation (whose cap hyperparameter is in degrees, $[0°; 360°]$),
(b) Crop (whose cap hyperparameter is in pixels),
(c) Brightness (whose cap hyperparameter is a digit on the scale of $[0; 1]$),
(d) Hue (whose cap hyperparameter is a digit on the scale of $[0; 1]$),
(e) Saturation (whose cap hyperparameter is a digit on the scale of $[0; 1]$).

These are applied on the images in the training loop at a given proportion of the training set, which is randomly selected at every epoch. Each augmentation carries respective hyperparameters, guid-

[10] Available at `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html#sgd`, accessed on March 5th, 2023.

[11] Available at `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html#crossentropyloss`, accessed on March 5th, 2023.

ing the strength of the alternation. We experimented with different values for both the occurrence and augmentation hyperparameters, as described in Subsection 2.1.3.

While augmentations (a-c) generate new instances, which may be plausible encounters from the wild, augmentations (d-e) significantly alter the color, a crucial discriminator for many plant diseases. We started with this set of augmentations – considered standard – and designed experiments to validate their merits, as we wondered whether augmentations (d-e) would not depreciate the annotations. Later, we selected only a subset of these augmentations for the Model to be used in the mobile application.

### 2.1.3 Experiments and Results

As reported in Table 2.1 and Figure 2.2(a), we first trained models of different learning rates and studied the impact of this hyperparameter on the overall validation accuracy. Note that we report the best attained validation accuracy across the experiment run. With a learning rate of $1E-01$, the accuracy lingered at 65.72 %. At $1E-02$, it soared to 95.61 %, and later only worsened: for learning rates of $1E-03$ and $1E-04$, the models attained an accuracy of 92.42 % and 72.18 %, respectively. From now on, we set the learning rate to $1E-02$.

Next, we conducted a semi-manual hyperparameter search. We identified 4 different values for each augmentation hyperparameter and measured their impact on the overall validation accuracy. We employed only a single augmentation technique during each experiment. While a better result

| **Learning rate** | **Accuracy** |
|---|---|
| $1E-01$ | 65.72 |
| $1E-02$ | 95.61 |
| $1E-03$ | 92.42 |
| $1E-04$ | 72.18 |

Table 2.1: **Learning rate search.** Performance of base models trained without augmentations, depending on the different learning rates. Validation accuracy is reported as the top-1 macro accuracy on the dataset's fixed 20% subsplit.

| **Rotation** | **Crop** | **Color jitter** | **Accuracy** |
|---|---|---|---|
| ✓ | | | 96.93 |
| | ✓ | | 96.21 |
| | | ✓ | 66.62 |
| ✓ | ✓ | | 96.33 |
| | ✓ | ✓ | 67.64 |
| ✓ | | ✓ | 68.01 |
| ✓ | ✓ | ✓ | 65.32 |

Table 2.2: **Ablation study.** Performance of the models trained with different combinations of augmentation techniques. Validation accuracy is reported as the top-1 macro accuracy on the dataset's fixed 20% subsplit.

could be achieved using a fully automatic hyperparameter search (e.g., grid search), we unwrap these features individually to gain more intuition about the system's robustness.

We report the results in Figure 2.2(b-f). The random rotation and crop augmentations, shown in plots (b) and (c), slightly improve the final accuracy to 96.93 % and 96.21 %. However, there seems to be no statistically significant difference among the examined values. The brightness, hue, and saturation augmentations also slightly improve the final testing accuracy, as depicted in plots (d), (e), and (f). Similarly, the examined hyperparameter values' difference seems statistically insignificant. For the comprehensiveness of this study, we joined these three augmentations under the umbrella of color jitter augmentation for the remaining experiments.

Fixing the best hyperparameter values, we conducted an ablation study, as reported in Table 2.2. Perhaps surprisingly, the combination of crop and rotation augmentations slightly lowered the model's accuracy to 96.93 %, ranking below the scores of the single-augmentation model, where random rotation was employed. All remaining configurations where color jitter is used rank significantly below, nonetheless: when joined with crop, the model gains an accuracy of 67.64 %, and when joined with rotation, the model gains an accuracy of 68.01 %. All augmentations combined get to 65.32 % only.

While the single-augmentation model with crop attains the best result, we used the model where it is combined with rotation, regardless of the slightly worse ($-0.6$ %) test performance. Through preventing over-fitting, this set of augmentations helps the model generalize to in-the-wild instances better at virtually no testing-accuracy cost.

When compared with the state of the art (EfficientNet Ensemble at 100.00 % testing accuracy, reported in Table 1.1), our Model – with 96.33 % testing accuracy – ranks worse, but is much simpler in architecture, which makes it portable. The trade-off between performance and simplicity seems reasonable.

### 2.1.4 Model Conversion

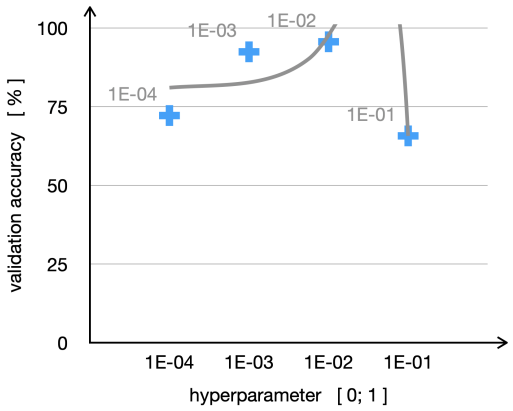
Once we identified which model to use in production, we converted it into the `.mlmodel` format using the open-source *coremltools*[12] library. In this format, the Model can be inferred natively on iOS devices, while utilizing their GPU-accelerated compute. We tested that the conversion was completed correctly by examining predictions of 10 randomly selected images and ensuring that the predictions of both models would match.

## 2.2 Mobile Application

Once the Model was trained and converted, we moved to the Mobile Application development. Using Xcode Integrated Development Environment (IDE), we implemented the app in Swift 5[13], native programming language from Apple. The user interface (UI) was implemented in *SwiftUI*,

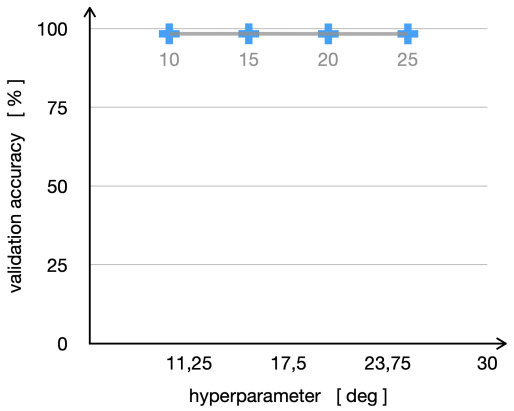
[12]Open-source library available from `https://github.com/apple/coremltools`, accessed on December 22nd, 2022.

[13]Open-source software available from `https://www.swift.org/`, accessed on March 6th, 2022



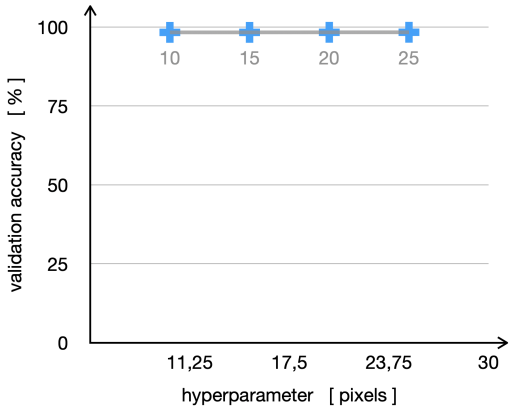

(a) **learning rate**

logarithmic scale, polynomial fit, max: 0.01 → 95.69%



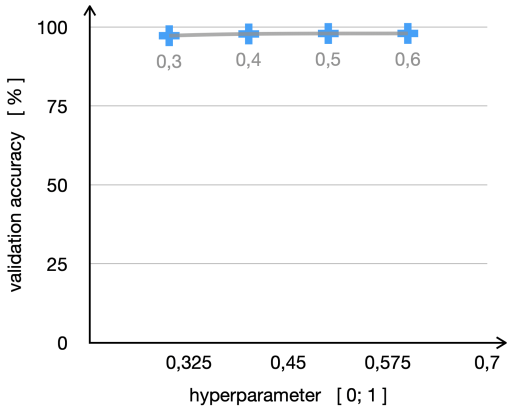

(b) **rotation**

linear scale, polynomial fit, max: 15 → 96.29%



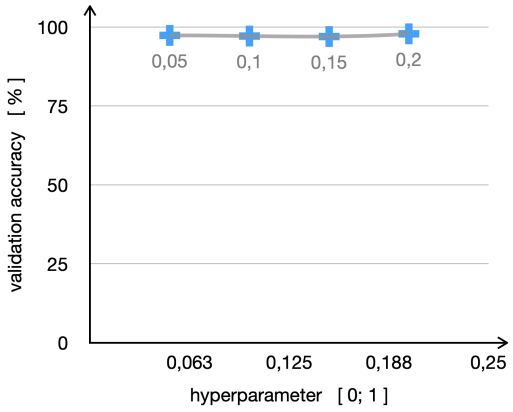

(c) **crop**

linear scale, polynomial fit, max: 15 → 95.79%



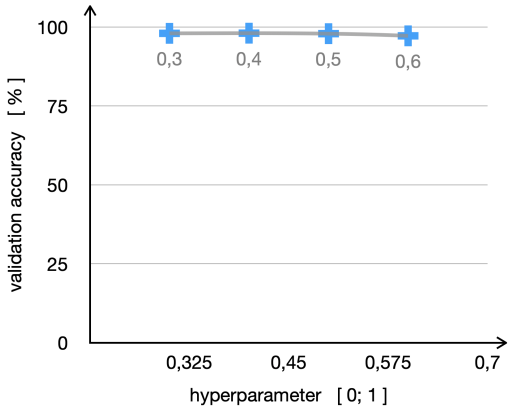

(d) **brightness**

linear scale, polynomial fit, max: 0.5 → 82.95%




(e) **hue**

linear scale, polynomial fit, max: 0.3 → 82.01%




(f) **saturation**

linear scale, polynomial fit, max: 0.05 → 80.69%

Figure 2.2: **Hyperparameter search.** Each subplot shows the performance with different values of a single observed hyperparameter. The x-axis captures examined values, and the y-axis shows the top-1 macro validation accuracy on the testing subsplit.


Figure 2.3: **App's visual identity.** The standalone logo (shown on the left) is used as the app's icon.

Swift's latest UI framework. While the other UI framework, *UIKit*, is still available for use, Apple deemed it for deprecation in the upcoming versions of iOS.

### 2.2.1 Designing and Prototyping

The set of features for the mobile application was first validated and iterated with 2 local farm representatives and 2 amateur farmers. We then created mockups in Figma[14] and ensured that they were understandable for these prospective users. When designing the screens, we followed Apple's Human Interface Guidelines[15], to ensure that the product would naturally fit into the ecosystem.

At this point, we also determined our service's brand identity. To convey its purpose straightforwardly, we chose to name it *Plant Doctor AI*. We made sure that this name had not been used elsewhere and also designed a minimalistic logo (serving as app icon), shown in Figure 2.3. In the app, we used two sets of icons: Apple's built-in SF Symbols[16] and Windows Color Set by Icon8[17], shown in Figure 2.4. Combined with demonstrative photos from the PlantVillage dataset, these were the only external resources used for the application's design.

Basic navigation in the app comprises three views: main menu, 'About' view, and 'Welcome' view, as shown in Figure 2.5. The inference process flow requires the user to take a photo or select one from their gallery; within seconds, the result is presented, with an option to view more detail about the diagnosis in a 'Detail' view, shown in Figure 2.6. The 'Detail' view includes the app's symptoms, spread, and prevention.

### 2.2.2 Codebase and Architecture

The `PlantDiseaseIdentifierApp` structure serves as the entry point to the entire application, since the system initializes it upon opening the application. In addition to wrapping the entire UI, it also handles camera and photo gallery permissions. Adhering to the principles of Object-oriented Programming, we represent each UI view in our application as a separate Swift `Structure`.

[14] Available at `https://www.figma.com/`, accessed on March 6th, 2023.

[15] Available at `https://developer.apple.com/design/human-interface-guidelines/`, accessed on February 17th, 2023.

[16] Available at `https://developer.apple.com/sf-symbols/`, free use for iOS app purposes and their propagation allowed free of charge.

[17] Available at `https://icons8.com/icons/fluency`, license purchased.

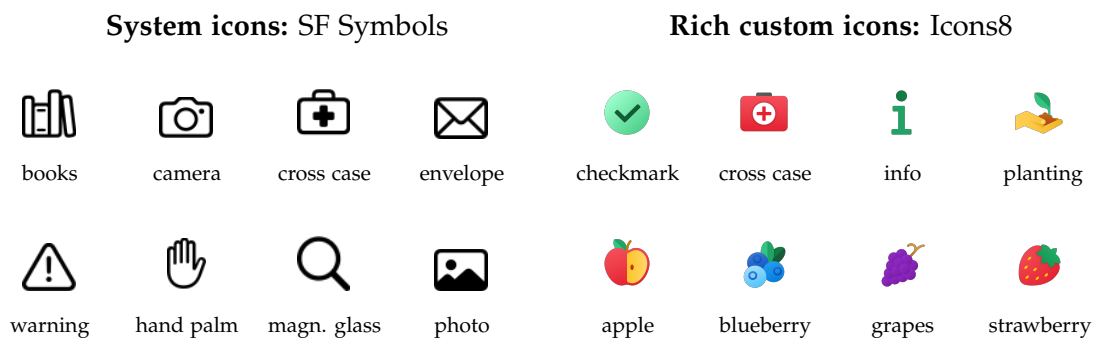



Figure 2.4: **Iconography.** The icons from SF Symbols (presented on the left) are used for granular identifiers within text, buttons, and navigation. The icons from Icons8 (presented on the right) are used as plant identifiers, section headers, or notable announcement bearers.

The `PlantDisease` class is a critical asset of the application, representing each possible diagnosis recognizable by the Model. In addition to its name and identifier, it contains textual details on the disease's spread and treatment, an icon image, an example photo, and links to expert databases.

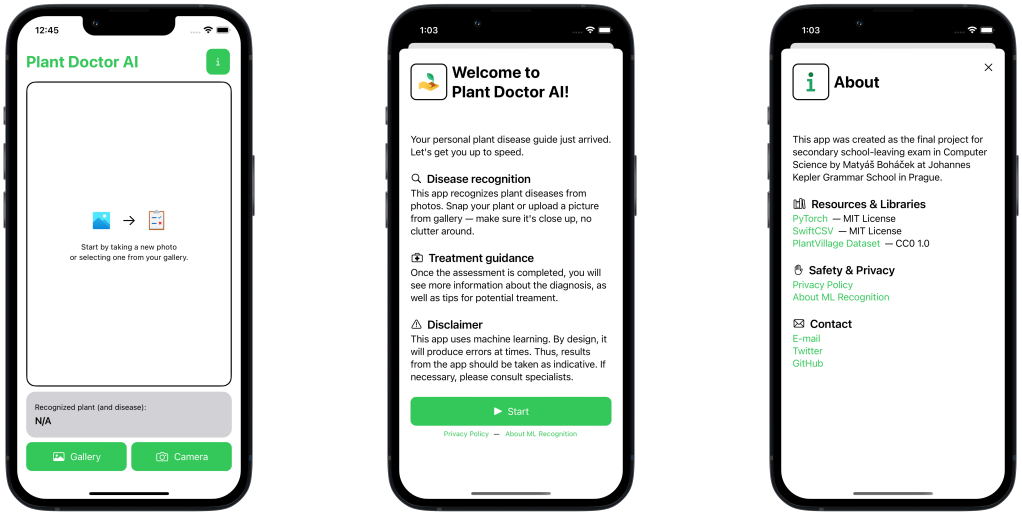
The `PlantDiseaseClassificationManager` and `DatabaseLoadingManager` classes handle the application's heavy lifting. The former wraps the image classification mechanism, performing inference on the Model and structuring its predictions. The latter loads and organizes the database with disease information to present after diagnosis.

### 2.2.3 Database

To present diagnosis details, we had to construct a database with information about each disease's name, symptoms, spread, and prevention. We scraped this information from PlantVillage's public directory[18], but found it inconsistent in writing style and length. Thus, we employed ChatGPT[19] to unify these texts. We presented the entire body of the disease profile and showed an example of a manually written disease description in the desired length and style. We could quickly restructure information about all diseases using the prompt *"Rewrite this plant description into bullet-point detail card, as shown in the example"*. We then verified the factualness of these new captions to ensure that the model did not fabricate them.

In addition to a hard-wired description in our database, we found expert articles about each entry on the following agricultural and gardening portals: Royal Horticultural Society[20], The Old Farmer's Almanac[21], Bayer Vegetables Solutions[22], MSU Extension Integrated Pest Management[23], NC State Extension[24], UC Riverside Center of Invasive Species Research[25]. We contacted each provider or

[18] Available at `https://plantvillage.psu.edu/plants`, accessed on December 20th, 2023.
[19] Available at `https://openai.com/blog/chatgpt/`, accessed on December 20th, 2023.
[20] Available at `https://www.rhs.org.uk/`, accessed on March 8th, 2023.
[21] Available at `https://www.almanac.com/`, accessed on March 8th, 2023.
[22] Available at `https://www.vegetables.bayer.com/us/en-us.html`, accessed on March 8th, 2023.
[23] Available at `https://www.canr.msu.edu/ipm/`, accessed on March 8th, 2023.
[24] Available at `https://www.ces.ncsu.edu/`, accessed on March 8th, 2023.
[25] Available at `https://cisr.ucr.edu/`, accessed on March 8th, 2023.




(a) main menu (b) welcome view (c) about view

Figure 2.5: **Design of the app's skeleton.** Three main navigation screens (outside the recognition flow) are shown. These mock-ups hold actual in-app screenshots. Generated using MockUPhone, `https://mockuphone.com/`.

author to ensure they agreed to include links to their services in our application first.

When selecting the database technology, we considered the following options:

(a) creating and hosting a custom *MySQL*[26] database,
(b) creating a *CloudKit*[27] database (Apple's custom database service),
(c) packaging the data – structured as a CSV file – with the application.

Using options (a) or (b) would make sense if the database was large-scale and we would need to modify its information throughout the app's lifecycle, whereas option (c) best suits small-scale databases with no necessity to be updated immediately. Moreover, option (c) is the only one where no internet connection is necessary. Given that our database had only 16 kB in size and we could not identify a case where an immediate data update–contrary to an app version update–would be necessary, we opted for option (c).

Therefore, we structured the data into a CSV and added it to the app's repository. As core Swift cannot read CSVs, we implemented the data loading using the *SwiftCSV*[28] library. Upon every application launch, the entire database is loaded and converted into `PlantDisease` objects for easy access.

[26] Available at `https://www.mysql.com/`, accessed on March 6th, 2023.
[27] Available at `https://developer.apple.com/icloud/cloudkit/`, accessed on March 6th, 2023.
[28] Open-source library available from `https://github.com/swiftcsv/SwiftCSV`, accessed on December 20th, 2022.

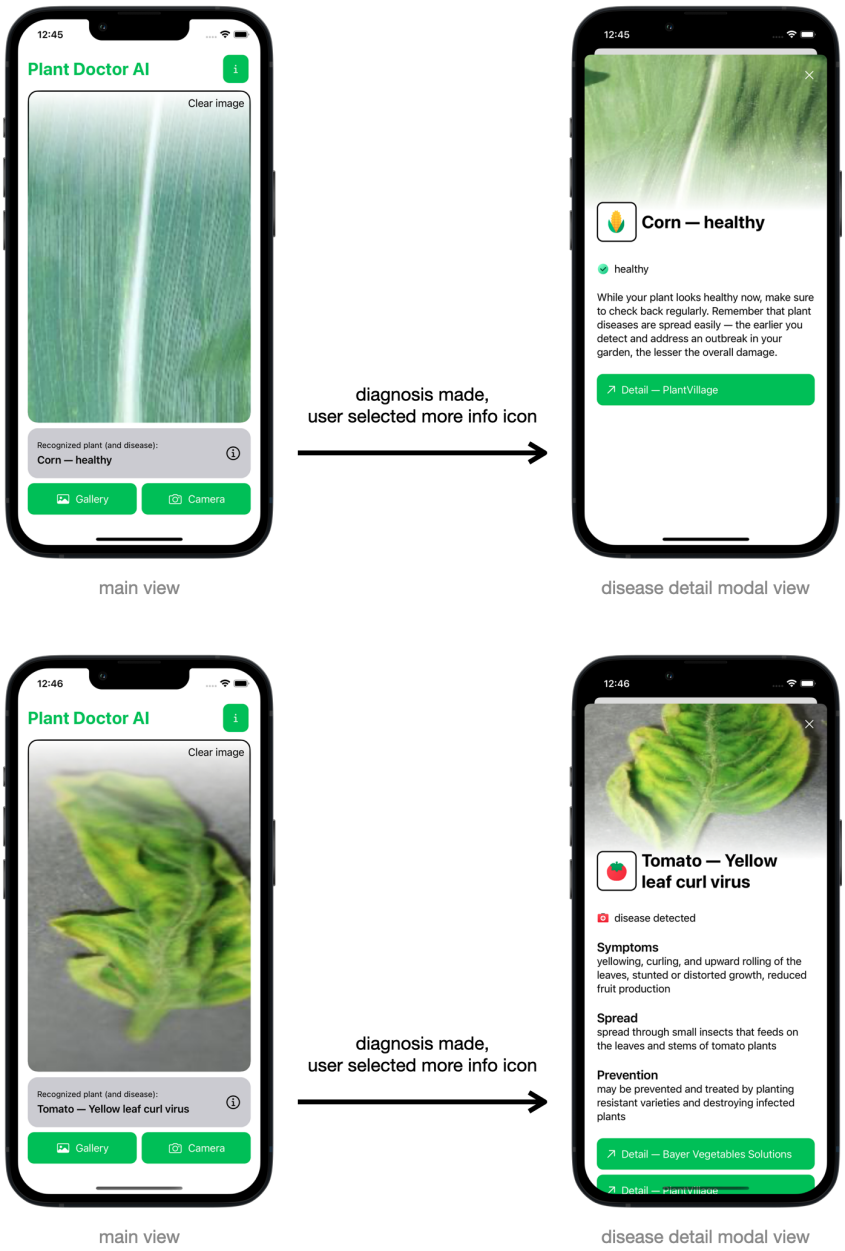



Figure 2.6: **Design of the in-app recognition flow.** These screens show how results are presented, once the user inserts the input photo for inference and its prediction is completed. The mock-ups hold actual in-app screenshots.

### 2.2.4 Machine Learning Inference on Mobile

Swift includes a built-in *CoreML* library for on-device inference on the GPU and Neural Engine. We use it in combination with the *Vision* library, which provides an interface for asynchronous image-based ML inference in the `VNCoreMLRequest` and `VNClassificationObservation` classes.

### 2.2.5 Considerations for Moving to Android

Given the platform's extensive on-device ML inference resources, we've implemented the *Plant Doctor AI* exclusively for iOS. However, we're optimistic about the possibility of creating the application for Android as well. To that end, the following frameworks could be used throughout the code-base:

- **Core app:** For native app development, Kotlin[29] could be used instead of Swift;
- **On-device ML inference:** The newly released *KotlinDL*[30] library could replace *CoreML* and *Vision* for on-device Model inference;
- **Database:** While the data would remain in a CSV, it could be loaded using the *kotlin-csv*[31] library.

## 2.3 Testing

We set up both automated and manual testing procedures to ensure that the application works as expected. The automated tests were implemented on par with the features, ensuring that new builds committed to the repository or sent to the App Store work as expected. The manual testing with external testers was a one-time event, but internal testing still occurs after any minor change.

### 2.3.1 Automated

Located in `mobile_app/PlantDiseaseIdentifierTests/` are the automated unit tests, focusing on two pillars of the application: the database and the machine learning inference. First, `ObservableDatabaseLoadingWrapperTests` class ensures that the database-loading mechanism reads data as expected and responds accordingly to adversaries in the form of erroneous data or unexpected CSV formatting. As for the machine learning inference, `PlantDiseaseClassificationManagerTests` class tests that the pre-processing manipulates input imagery as expected and that several input images — whose predictions were saved in the *Python* environment before converting the model — evaluate to the same classes.

The automated UI tests are located in `mobile_app/PlantDiseaseIdentifierUITests/`, ensuring that the screens compile and show on the screen. These tests also calculate benchmarks, raising flags when the app has an unexpected computational bottleneck that slows down the UI.

### 2.3.2 Manual

We conducted multiple rounds of manual testing. First, we ensured that the application and all its functionalities work as expected on virtual simulators of different devices: iPhone 8 Plus, iPhone 11 Pro Max, iPhone 14, and iPhone 14 Pro Max. We paid particular attention to the design, ensuring that smaller screens do not get any parts of the screen cut off. Next, we installed the application on 2 physical devices and repeated the exact testing process. Finally, we resolved any bugs encountered.

[29]Open-source software available from `https://kotlinlang.org/`, accessed on March 6th, 2022

[30]Open-source library available from `https://github.com/Kotlin/kotlindl`, accessed on March 6th, 2022.

[31]Open-source library available from `https://github.com/doyaaaaaken/kotlin-csv`, accessed on March 6th, 2022.

After a week of internal testing, we moved to a closed beta phase with 5 directly contacted users. We distributed the application using Apple's *TestFlight*[32], through which the testers could send screenshots and feedback, along with diagnostics data. We received 2 suggestions; 1 was implemented. Overall, the feedback was encouraging.

## 2.4 Deployment

To fulfill the goal *(a) free and accessible use*, as outlined at the beginning of the project, we deployed the application on the App Store[33] using Apple's App Store Connect[34] toolkit (the author possesses an Apple Developer License). The application successfully underwent a review process and got listed for free download[35].

[32] Open-source library available from `https://developer.apple.com/testflight/`, accessed on March 6th, 2022.

[33] Available at `https://www.apple.com/app-store/`, accessed on March 5th, 2023.

[34] Available at `https://appstoreconnect.apple.com/`, accessed on March 5th, 2023.

[35] Available at `https://apps.apple.com/us/app/plant-doctor-ai/id6446052415`, accessed on March 12th, 2023.

# 3. Technical Documentation

We open-source the implementation of our project on GitHub[1]. The repository comprises 2 folders: `machine_learning_pipeline`, holding the scripts for the Model's training and conversion, and `mobile_app`, with the mobile application package and resources.

Throughout this chapter, we describe how to install necessary requirements and build this project. We include Terminal commands intended for macOS and Linux, if not specified otherwise. Linux commands were tested on Ubuntu 18.04.6[2]; macOS commands were tested on macOS 13.1 Ventura[3].

To start, clone the repository:

```
git clone https://github.com/matyasbohacek/gjk-maturitni-prace-inf-2022
```

## 3.1 Image Classification Pipeline

For this part of the project, a Linux machine with Anaconda 4.12.0[4] software installed is required. Altered versions of this pipeline may work for macOS and Windows as well, but will require changes with regard to the GPU management.

For this part of the documentation, you should find yourself in the `machine_learning_pipeline/` subdirectory.

### 3.1.1 Installation

1. If you already have the desired version of Anaconda, skip to step 5. Otherwise, start by moving to the `/tmp` directory:

```
cd /tmp
```

2. Download the necessary installation script:

```
curl https://repo.anaconda.com/archive/Anaconda3-2022.05-Linux-x86_64.sh --output
    anaconda.sh
```

3. Make the script executable:

[1] `https://github.com/matyasbohacek/gjk-maturitni-prace-inf-2022`

[2] Open-source software available from `https://releases.ubuntu.com/18.04/`, accessed on December 16th, 2022

[3] Free Mac operating system, `https://developer.apple.com/documentation/macos-release-notes/macos-13_1-release-notes`, accessed on December 17th, 2022

[4] Open-source software available from `https://docs.anaconda.com/anaconda/reference/release-notes/#anaconda-2022-05-may-10-2022`, accessed on December 16th, 2022

```
chmod +x anaconda.sh
```

4. Run the installation script and follow the instructions on the screen.

```
./anaconda.sh
```

5. Next, locate the directory with this project (`gjk-maturitni-prace-inf-2022/`) and move there using the `cd` command.

6. Open the folder with training scripts:

```
cd machine_learning_pipeline
```

7. Create a Conda environment with Python 3.7.15[5] and PyPI[6]:

```
conda create -n plant-doctor-ai python=3.7.15
```

8. Activate the environment:

```
conda activate plant-doctor-ai
```

9. Finally, install the dependencies:

```
pip install -r requirements.txt
```

### 3.1.2 Model Training

1. To train the Model, download the dataset[7] and unwrap it. Make note of its location.

2. Activate the Conda environment:

```
conda activate plant-doctor-ai
```

3. Train the Model using:

```
python3 -m train --dataset-path "[/path/to/PlantVillage]"
```

Note that the hyperparameters may be specified in the `train.py` (as defaults) or provided as flags

[5]Open-source software available from `https://www.python.org/downloads/release/python-3715/`, accessed on December 16th, 2022

[6]Available at `https://pypi.org`, accessed on December 16th, 2022

[7]At the time of writing, the original distribution of the dataset through PlantVillage has been temporarily suspended. You may download it from an official Mendeley repository at `https://data.mendeley.com/datasets/tywbtsjrjv` or an unofficial clone at `https://github.com/spMohanty/PlantVillage-Dataset`. Both of these sources were verified as of January 8th, 2023.

with the last command. The recognized hyperparameters include:

- `--dataset-path`: (string) path to the PlantVillage dataset, unzipped and structured in the default class-corresponding folder scheme,
- `--epochs`: (integer, $n > 0$) number of epochs to train for,
- `--batch-size`: (integer, $n > 0$) number of training items to batch during the training loop, constrainted by your GPU's memory,
- `--learning-rate`: (float, $n > 0$) learning rate parameter passed onto the selected optimizing algorithm (Adam),

- `--transforms-color-jitter`: (boolean flag) indicator of whether to use the color jitter augmentations,
- `--transforms-random-crop`: (boolean flag) indicator of whether to use the random crop augmentations,
- `--transforms-rotation`: (boolean flag) indicator of whether to use the rotation augmentations,

- `--rotation-deg`: (integer, $n > 0$) maximum degrees to rotate using the rotate augmentation,
- `--crop-min-pix`: (integer, $n > 0$) number of pixels to subtract in both height and width for the random crop augmentation window,
- `--brightness-range`: (float, $0 < n < 1$) maximum relative change in brightness to use with the brightness augmentation,
- `--saturation-range`: (float, $0 < n < 1$) maximum relative change in saturation to use with the saturation augmentation,
- `--hue-range`: (float, $0 < n < 1$) maximum relative change in hue to use with the hue augmentation.

Upon finishing the last epoch, the resulting Model's weights will be saved in your current directory as `model.pt`.

If you wish to reproduce the results reported in this work, you can run:

```
cd shell-scripts
chmod +x ablations.sh
./ablations.sh
chmod +x augmentation-combinations.sh
./augmentation-combinations.sh
```

### 3.1.3 Model Conversion

To convert the Model into the `.mlpackage` format, locate your *PyTorch*-produced weights (file ending with `.pt`). In the same directory, call:

```
python3 -m convert_model --pt-model "[path/to/model.pt]" --output-model
    "[output-model.mlpackage]"
```

Note that the `--pt-model` parameter expects the path to the existing model, while the `--output-model` parameter specifies the name of newly converted file. The latter must end with `.mlpackage`.

### 3.1.4 Troubleshooting

Having tested the reproducibility of our project on different machines, we would like to draw the reader's attention to some considerations, which should be made prior to training. These are based on reoccurring problems we ran into on different systems.

Note that the dependencies of our repository include a specific version of *PyTorch*, which uses CUDA[8] to accelerate training. Depending on your GPU, your system might support only some CUDA versions, limiting the range of supported *PyTorch* versions. If the version required by the repository lies out of your supported range, you should consider experimenting with different *PyTorch* versions; nonetheless, note that we cannot guarantee that the code will work for these.

Additionally, we often ran into errors that concerned the CUDA toolkit and were not related to the training scripts as such. To avoid these, ensure that CUDA is set up correctly.

## 3.2 Mobile Application

For this part of the documentation, you should find yourself in the `mobile_app/` subdirectory.

To build and run the application, a macOS 13.1 Ventura[9] machine with Xcode 14.2[10] and Simulator 14.2[11] applications is required. Both of these applications are freely available on the macOS App Store.

First, complete the set-up of Xcode: provide your Apple ID with the Developer Account setting enabled[12]. Once you're logged in, open the app's Xcode project file, located at `mobile_app/PlantDiseaseIdentifier.xcodeproj`.

### 3.2.1 Installation

Xcode should automatically install all requirements on the first run. As we tested the reproducibility of our repository, however, we found that Xcode sometimes does not manage to install the *SwiftCSV* library. If that is the case, clone it into `mobile_app/` directory:

```
git clone https://github.com/swiftcsv/SwiftCSV
```

[8] Available at `https://developer.nvidia.com/cuda-toolkit`, accessed on March 19th, 2023

[9] Free Mac operating system, `https://developer.apple.com/documentation/macos-release-notes/macos-13_1-release-notes`, accessed on December 17th, 2022

[10] Free macOS-only software available from `https://apps.apple.com/us/app/xcode/id497799835`, accessed on December 17th, 2022

[11] Free macOS-only software packaged with the Xcode application, accessed on December 17th, 2022

[12] Details about set-up and enrollment can be found at `https://developer.apple.com/support/app-account/`, accessed on December 17th, 2022

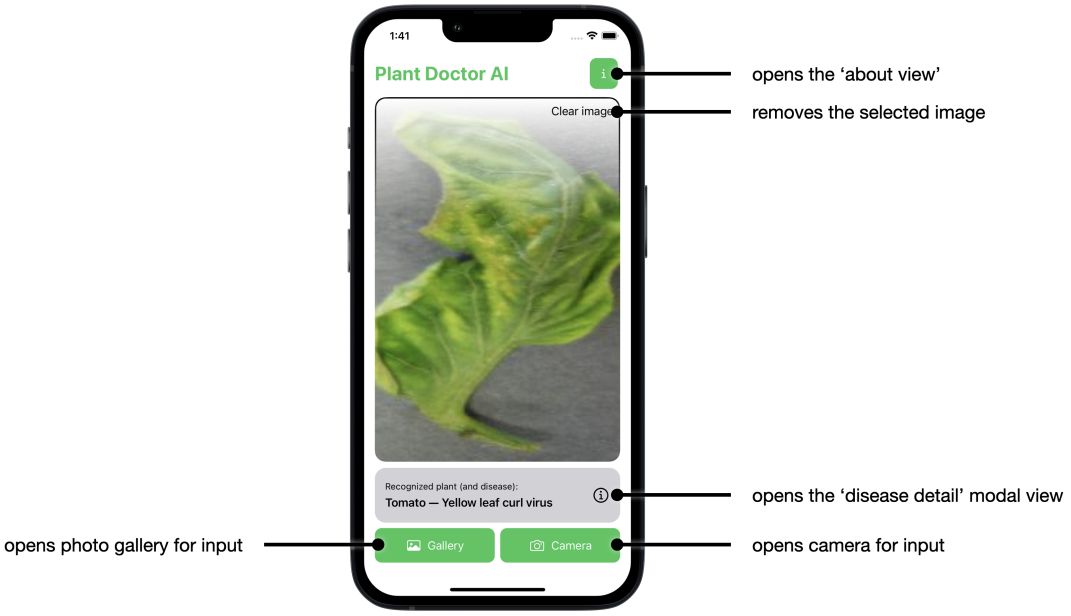



Figure 3.1: **Main menu actions.** Each button is highlighted and its action described.

You should now see the *SwiftCSV* package linked in the Xcode's Packages sidebar. Ensure that it is also linked in the Project overview pane under **General** > **Frameworks, Libraries, and Embedded Content**. If that is not the case, add it using the plus button.

### 3.2.2 Building

To build the application in Xcode, change the Target to *any iOS Device (arm64)*. From the top bar, select **Product** > **Build** (or hit **`Command + B`**).

### 3.2.3 Running (Local Testing)

To run the app, you may either employ the virtual simulator or your own physical device. In the top bar's *Target menu*, select the preferred form of testing (and device, if applicable). If you are to test on a physical device, make sure that it is connected to your computer with a cord and that you confirm any agreements presented on your phone first. When you are ready, select **Product** > **Run** (or hit **`Command + R`**).

### 3.2.4 Distribution

To archive the app and distribute it through the means of App Store or TestFlight, change the Target to *any iOS Device (arm64)*. From the top bar, select **Product** > **Archive**, and you will be guided through the archival process; simply follow on-screen instructions.

Once the upload finishes, visit App Store Connect, where marketing and distribution details are handled. From there, you can select to publish the app.

## 3.3 User Guide

When you open the app for the first time, you are prompted with the *Welcome view* (Figure 2.5 (b)), which introduces the app's features and provides tips for a smooth recognition experience.

Having closed it, you will see the *Main menu* (Figure 2.5 (a)). Shown in Figure 3.1 is the overview of its buttons and respective actions. At the very top right, the button titled "i" opens the *About view* (Figure 2.5 (c)) with the list of resources used for development and contact information. Additionally, this view also contains links to three notable documents, which you should review before you start using the core features: *Terms & Conditions*[13], *Privacy Policy*[14], and *About ML Recognition*[15]. These are also prompted in the *Welcome view* (i.e., on your first run of the app).

To start predicting diseases, use either the "Gallery" or "Camera" button at the bottom of the screen. Each will prompt you to select a photo as input. Once selected, the photo will be analyzed using the Model, and the results will be displayed in the center of the screen. You can clear these results and start over by tapping the "Clear image" button. If you want to learn more about the diagnosis, you can tap the "i" button in the gray results area. This will open the *Disease detail model view* where you can learn more about the disease or find additional expert resources.

[13] Available at `https://www.matyasbohacek.com/plant-doctor-ai/terms-and-conditions.pdf`, accessed on March 12th, 2023.

[14] Available at `https://www.matyasbohacek.com/plant-doctor-ai/privacy-policy.pdf`, accessed on March 12th, 2023.

[15] Available at `https://www.matyasbohacek.com/plant-doctor-ai/about-ml-recognition.pdf`, accessed on March 12th, 2023.

# Conclusion

In summary, this work presents *Plant Doctor AI*: a new iOS application for plant disease recognition and treatment guidance powered by a deep-learning Model, which works without an internet connection. Whenever a diagnosis is made, information about the symptoms, spread, and prevention is presented, as well as links to expert resources. The mobile development was preceded by a survey of current literature and existing solutions, as well as a rigorous experimental phase when the deep-learning Model was trained. Our code is open-sourced, and the app is available for free from the App Store[16].

To our best belief, we completed the assignment and requirements outlined by the advisor in full. We believe that we have met the constituent goals and attributes, which we set up in Section 1.1, too:

(a) **Free and accessible use:** met as the source code is open-sourced, and the mobile application is listed on the App Store;
(b) **Architecture transparency:** met as the architecture description is presented in this report and the application's App Store page;
(c) **Public benchmarking results:** met as the benchmarking scores are presented in this report and the application's App Store page;
(d) **On-device processing:** met as the deep-learning Model was converted to a *Swift*-supported format, which runs completely offline thanks to a built-in CSV database.

In terms of work ethic and performance, we believe to have met the consultations requirements, as well as other formalites. We delivered features on time and stuck to the initially proposed schedule.

The mobile application's potential is far from complete. We acknowledge that many valuable features would be ideal to have but simply did not fit into the so-far development. Looking forward, we enumerate multiple leads for the following features and build-up work:

- Improve the Model's performance on unseen examples;
- Expand Model's knowledge base (set of recognized diseases);
- Detect when a non-leaf photo is presented;
- Segment the leaf;
- Expand mobile application's language support (i.e., translate it to more languages);
- Instead of presenting just the single most likely disease, list top-5 classes with representative images to help mitigate misclassifications and bring the user to the loop;
- Show recognition confidence;
- Add an option to report incorrect predictions.

Even though we met the goals, we encountered countless errors throughout the development; indeed, not everything went according to plan. Most notably, the conversion of the Model brought much confusion. It required countless hours of debugging, as we noticed that the Model would initially provide different results for identical input in the Python and Swift environments. Simultaneously, no errors were triggered, so it took time to understand where to even begin investigating.

[16] Available at `https://apps.apple.com/us/app/plant-doctor-ai/id6446052415`, accessed on March 12th, 2023.

Eventually, we discovered that the cause was relatively trivial and absurd – at first, we implemented the normalization of the data differently in Python and Swift. While the Model was expecting normalized RGB data with each color component normalized to $[-1; 1]$, we initially passed raw-integer RGB data ($[0; 255]$). On a different note, we struggled to contact expert database providers to obtain their permissions. It took many follow-up emails and phone calls to eventually secure them.

Throughout the development, we learned how to work with many new tools and libraries; perhaps the most treasured new skill is the just-mentioned conversion of machine learning models from their original representations into mobile-supported formats that can run offline using *coremltools*. Moreover, we learned the importance of data pre-processing (especially its uniformity across contexts). We understood just how complicated it is to run a beta testing program with non-technical people. Lastly, connected administrative responsibilities – such as obtaining permissions to use third-party data or creating legal documents like Terms of Service – were explored.

We believe this work can help improve access to crop disease detection technology and challenge existing solutions, which are either paid or non-transparent in their inner architecture. Furthermore, our pipeline and architecture may serve the machine learning community as a template for other image classification use cases, which ought to be brought to mobile devices.

# References


[1] George N Agrios. *Plant pathology*. Elsevier, 2005.

[2] Ümit Atila et al. "Plant leaf disease classification using EfficientNet deep learning model". In: *Ecological Informatics* 61 (2021), p. 101182.

[3] Kodra Austin. "Machine learning on iOS and Android". In: *Heartbeat* (2019). URL: https://heartbeat.comet.ml/machine-learning-on-ios-and-android-bd77f6e92c7b.

[4] Leonard E Baum and Ted Petrie. "Statistical inference for probabilistic functions of finite state Markov chains". In: *The annals of mathematical statistics* 37.6 (1966), pp. 1554–1563.

[5] Antonio Bruno et al. "Improving plant disease classification by adaptive minimal ensembling". In: *Frontiers in Artificial Intelligence* 5 (2022), p. 868926.

[6] Junde Chen, Defu Zhang, and Yaser Ahangari Nanehkaran. "Identifying plant diseases using deep transfer learning and enhanced lightweight network". In: *Multimedia tools and applications* 79 (2020), pp. 31497–31515.

[7] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. IEEE. 2009, pp. 248–255.

[8] Keinosuke Fukunaga. *Introduction to statistical pattern recognition*. Elsevier, 1973.

[9] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[10] Xiao Gong and N-K Huang. "Textured image recognition using hidden Markov model". In: *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*. IEEE Computer Society. 1988, pp. 1128–1129.

[11] Robert M. Haralick, K. Shanmugam, and Its'Hak Dinstein. "Textural Features for Image Classification". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-3.6 (1973), pp. 610–621. DOI: 10.1109/TSMC.1973.4309314.

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[13] John J Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558.

[14] David Hughes, Marcel Salathé, et al. "An open access repository of images on plant health to enable the development of mobile disease diagnostics". In: *arXiv preprint arXiv:1511.08060* (2015).

[15] Margaret Tuttle McGrath. "General tips on identifying plant diseases". In: *Cornell Vegetables* (). URL: https://www.vegetables.cornell.edu/pest-management/disease-factsheets/general-tips-on-identifying-plant-diseases/.

[16] Jerry M Mendel. "Adaptive learning and pattern recognition systems". In: *Theory and appricationss* (1970).

[17] Sharada P Mohanty, David P Hughes, and Marcel Salathé. "Using deep learning for image-based plant disease detection". In: *Frontiers in plant science* 7 (2016), p. 1419.

[18] R Nag, K Wong, and Frank Fallside. "Script recognition using hidden Markov models". In: *ICASSP'86. IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 11. IEEE. 1986, pp. 2071–2074.

[19] Wayne D. Rasmussen et al. "Origins of Agriculture". In: *Encyclopedia Britannica* (2022). URL: https://www.britannica.com/topic/agriculture.

[20] O'Connor Ryan. "PyTorch vs TensorFlow in 2023". In: *AssemblyAI Blog* (2023). https://www.assemblyai
vs-tensorflow-in-2023/.

[21] Edna Chebet Too et al. "A comparative study of fine-tuning deep learning models for plant disease identification". In: *Computers and Electronics in Agriculture* 161 (2019), pp. 272–279.

# List of Figures

# List of Tables