IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 11, November 2026

Croplytics: A Python-Based Smart Agriculture Assistant for Leaf Image Handling, Crop-Yield Estimation and Carbon-Footprint Estimation

Miss. Hemani Pravinsing Girase[1], Prof. Manoj V. Nikum[2]

Research Scholar MCA, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[1]

HOD MCA & Assistant Professor, Shri Jaykumar Rawal Institute of Technology, Dondaicha, India[2]

Abstract: Croplytics is a Python-based smart agriculture web application that gathers a few farming-support utilities on a single platform: plant-leaf image handling, crop-yield estimation, carbon-footprint estimation and agricultural product information. The backend is written in Python with the Flask framework, which receives browser requests, passes them to the relevant application logic and returns the result. Pillow and NumPy handle image operations, Pandas organizes agricultural datasets, and scikit-learn supports the yield-estimation workflow whenever a trained and compatible model is available. SQLite stores user and history records, and the interface is built with HTML, CSS and JavaScript. The application also provides user registration and login, with validation on both the browser and the server, salted password hashing and parameterized database queries. Croplytics is an educational prototype. Its image-analysis and prediction outputs should be read as demonstrations or estimates until the models and calculations behind them have been validated independently. This paper presents the motivation, background, analysis and design, module-wise implementation, validation approach, observed interface and possible future work.

Keywords: Python, Flask, smart agriculture, crop yield, image processing, carbon footprint, SQLite.

I. INTRODUCTION

Farming decisions are usually made with scattered information. A farmer may check a leaf problem with one person, look up expected production in a notebook or a government leaflet, and find product details from a local shop. Digital tools can reduce this running around, but many of them solve only one narrow problem, and some need a smartphone app, a paid subscription or an internet connection of good quality. Smart-agriculture research has pointed out for several years that data collection and analysis are becoming part of everyday farm management [1], and that the practical gap is often not the lack of algorithms but the lack of simple tools that bring several services together [2].

Croplytics was planned as a small answer to this gap. It is a web application written in Python that places four utilities under a single login: handling of plant-leaf images, estimation of crop yield, estimation of the carbon footprint of farm activities, and a catalogue of agricultural product information. The aim of the project is not to compete with commercial precision-farming systems. It is to show, in a form that a student team can build and a reader can follow, how such utilities can be organized inside one Flask application with a lightweight database behind it.

The scope is deliberately limited. Croplytics is an educational prototype. Whatever the leaf-image module or the yield module returns should be read as a demonstration or an estimate, because the models and the calculations used in the prototype have not been validated against field data by an independent party. We state this limitation at the start because a farming tool that sounds more certain than it is can mislead the person using it.

The objectives of the work are as follows:

- to study the needs of a small-scale farming user and identify a few utilities that can be offered together;
- to design a modular Flask application in which each utility is separate from the others;
- to implement secure registration and login with validation at both the browser and the server;
- to build the image, yield, carbon-footprint and product-information modules and connect them to a common interface; and
- to describe how the prototype can be checked and how it can be improved later.

The rest of the paper is arranged as follows. Section II reviews related work. Section III explains the analysis, design, implementation and validation approach. Section IV reports the observed interface and behaviour of the prototype, and Section V concludes with the limitations and the planned enhancements.

 This work is licensed under a Creative Commons Attribution 4.0 International License

II. LITERATURE REVIEW

Wolfert et al. surveyed the use of big data in smart farming and described how sensors, machinery and software are turning farms into data-driven operations [1]. Liakos et al. reviewed machine learning in agriculture and grouped the applications into crop management, livestock management, water management and soil management [2]. Their review also noted that most of the reported systems are built for one task, which is one reason a combined platform is worth experimenting with.

Image-based plant health work has received the most attention in recent years. Kamilaris and Prenafeta-Boldú surveyed deep-learning studies in agriculture and found that image classification of plant diseases and weeds was among the most common uses [3]. Mohanty et al. trained convolutional networks on a public set of leaf images and reported high accuracy on laboratory-style photographs, but they also cautioned that performance drops when the images are taken under different field conditions [4]. The open repository of plant-health images described by Hughes and Salathé was created to support exactly this kind of work [5]. For a prototype like Croplytics these studies are useful mainly as a warning: a leaf-analysis output is only as reliable as the data and the model behind it.

Crop-yield prediction has also been studied widely. Van Klompenburg et al. went through a large number of machine-learning studies on yield prediction and found that the most common input features were temperature, rainfall and soil properties, while the most common algorithms were neural networks and tree-based or regression methods [6]. This helps decide what a simple yield form should ask the user for, and why a classical scikit-learn model is a sensible choice for a small web application.

On the implementation side, scikit-learn provides a consistent interface for training and applying classical machine-learning models [7], NumPy supplies the array operations on which image handling depends [8], and Pandas is a convenient way of reading and organizing tabular datasets [9]. Flask is a small web framework that leaves the choice of database and form handling to the developer [10], which suits a student project in which each module is built and tested separately.

For the carbon-footprint utility, the usual approach in national inventories is to multiply an activity quantity, such as litres of diesel or kilograms of fertilizer, by an emission factor [11]. This method is simple enough to be implemented in a web form, but its accuracy depends entirely on the emission factors that are used. Although each of these areas is well covered in the literature, we did not find many small, open and teachable examples that join them in one application with a proper login system. Croplytics is an attempt to fill that narrow gap.

III. METHODOLOGY

A. System Analysis

The analysis began with a simple question: what would a small farmer or an agriculture student want to do on a single website? Four activities came out of the discussion, along with the need for each person to have an account so that history can be kept separately. The functional requirements were written as follows.

- **Account handling:** a visitor can register, log in and log out, and unauthorized visitors cannot open the private pages.
- **Leaf image handling:** a user can upload a leaf photograph, and the application checks and prepares it for analysis.
- **Yield estimation:** a user enters crop and field details and receives an estimated yield.
- **Carbon-footprint estimation:** a user enters farm-activity quantities and receives an estimated emission value.
- **Product information:** a user can browse details of agricultural products.

The non-functional requirements were equally important. The application should run on an ordinary computer without special hardware, the database should need no separate server, and a failure in one module (for example a missing model file) should not stop the others. Passwords must never be stored in readable form, and every value that comes from the browser must be treated as untrusted.

B. System Design

Croplytics follows a three-layer structure, shown in Fig. 1. The browser forms the presentation layer and is written in HTML, CSS and JavaScript. Flask forms the application layer: it receives each request, validates it, calls the logic of the relevant module and returns a response. SQLite, together with the model and dataset files, forms the data layer. Keeping every utility as its own module behind the same router makes it possible to change one of them without touching the rest.

 This work is licensed under a Creative Commons Attribution 4.0 International License

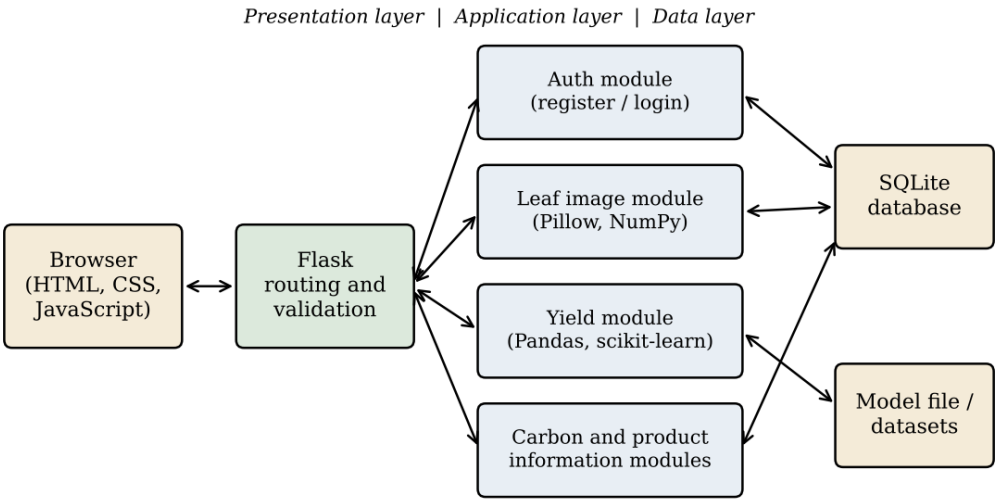


Fig. 1 Layered architecture of the Croplytics application

The technologies used and the role of each are summarized in Table I. All of them are open-source and are commonly used in Python teaching, which keeps the prototype easy to install and maintain.

TABLE I TECHNOLOGIES USED IN CROPLYTICS

Technology	Role in the application
Python	Main programming language for all server-side logic
Flask	Receives browser requests, routes them to modules and returns results
Pillow	Opening, checking, converting and resizing uploaded leaf images
NumPy	Numerical operations on image arrays
Pandas	Reading and organizing agricultural datasets
scikit-learn	Crop-yield machine-learning workflow when a compatible trained model exists
SQLite	Lightweight relational storage for user and history records
HTML, CSS, JavaScript	User interface and browser-side validation

C. Implementation of Modules

1) User Registration and Login:

The registration form asks for the basic details of the user and checks them in the browser with JavaScript, for instance for empty fields, a valid e-mail format and a minimum password length. Browser checks only improve the user experience and can be bypassed, so the same rules are applied again on the server before anything is saved. The password is never stored as typed. It is converted into a salted hash and only the hash is written to the database, following the general guidance for password storage [12]. During login the entered password is hashed again and compared with the stored value. All database statements use parameter placeholders and not string joining, which closes the common path for SQL injection. Once the user is logged in, a session identifies them and the private pages check this session before responding.

2) Plant-Leaf Image Handling:

The image module deals with the photograph before any analysis can take place. The server first checks that a file was actually sent, that its extension is among the allowed ones and that its size is within a fixed limit. Pillow then opens the file, which also confirms that the content is a real image and not some other file renamed to look like one. The image is converted to a single colour mode, resized to a fixed dimension and turned into a NumPy array whose values are scaled to a common range. These steps are the standard preparation for a classifier, and they also support simple numerical operations on the image. Because the module's output is a demonstration, the result is presented to the user as indicative only.

3) Crop-Yield Estimation:

The yield module collects crop and field details through a form and validates each value for type and sensible range. Pandas is used to organize the agricultural dataset from which the model is built, and scikit-learn is used to train and apply the model. The workflow is shown in Fig. 2. The important design decision is that the prediction is attempted only when a trained model file is present and compatible with the installed library version. If it is not, the application displays a clear message instead of failing or returning an invented number. The estimated total production can then be derived from the predicted yield per unit area as

$$P = Y \times A \qquad (1)$$

where P is the total production, Y is the predicted yield per hectare and A is the cultivated area in hectares.

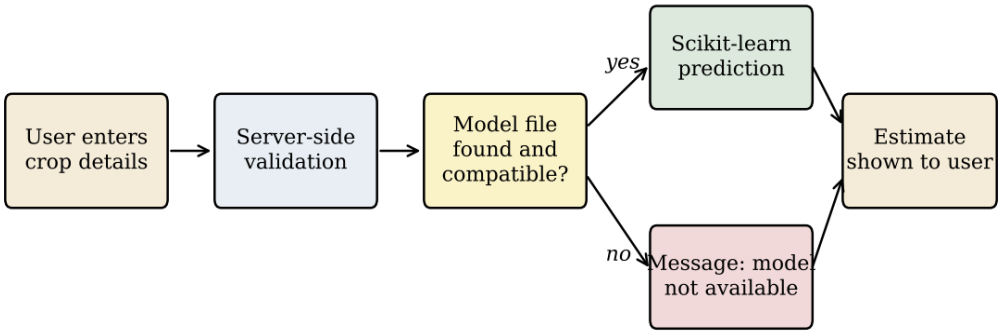


Fig. 2 Workflow of the crop-yield estimation module

4) Carbon-Footprint Estimation:

This module follows the activity-data-multiplied-by-emission-factor approach used in greenhouse-gas inventories [11]. The user enters the quantities of the inputs used on the farm, for example fuel, electricity and fertilizer, and the application adds up the emissions from each source. The total is expressed in kilograms of carbon dioxide equivalent:

$$E = \Sigma \left(Q_i \times EF_i \right) (2)$$

where Q is the quantity of input i and EF is the emission factor assigned to it. The factors are kept in one place in the code so that they can be updated or replaced with region-specific values without changing the calculation itself. Since the result depends directly on these factors, it is shown as an estimate.

5) Agricultural Product Information:

The product-information module is the simplest of the four. It reads stored product details, such as name, category and usage notes, and shows them in a browsable list with a search option. It is useful mainly as a reference page, and it also demonstrates how the same Flask routing and template structure can serve static information as well as computed results.

6) Data Storage:

SQLite was selected because it needs no separate database server and keeps the whole database in a single file, which is convenient for a prototype that will be run on student laptops and demonstration machines. It holds the user records and the history of supported operations. For larger deployments the same schema could later be moved to a server database.

D. Validation Approach

Because the prototype produces estimates, the validation plan separates two questions. The first is whether the software behaves correctly, that is, whether it accepts valid input, rejects invalid input and fails safely. The second is whether the numbers it returns are scientifically reliable, which is a much larger task and can only be answered with independently collected data. The present work addresses the first question directly and leaves the second for future work. The checks planned for each module are listed in Table II.

TABLE II VALIDATION CHECKS AND EXPECTED BEHAVIOUR

Module	Check	Expected behaviour
Registration	Empty fields, invalid e-mail, short password	Rejected in the browser and again on the server with a clear message
Login	Wrong password; private URL opened without login	Access denied; redirect to the login page
Database	Input containing quotes or SQL keywords	Treated as plain text; no change in query behaviour
Leaf image	Non-image file renamed with an image extension; oversized file	File rejected with a message; application keeps running
Yield	Missing or out-of-range values; model file absent	Validation message, or a notice that the model is unavailable
Carbon footprint	Negative or non-numeric quantities; known test values	Rejected input; total matches a manual calculation
Product information	Search with no matching product	Empty-result message instead of an error

IV. RESULT

A. Observed Interface

The interface is organized around a simple flow. A visitor first reaches the registration and login pages. After a successful login the user sees a common dashboard from which each of the four utilities can be opened. Each utility has its own page with a form or an upload control, and the result is displayed on the same page or on a short result page. Because the interface uses plain HTML and CSS with a limited amount of JavaScript, the pages load quickly and remain usable on a small screen.

B. Observed Behaviour of the Modules

The modular structure behaved as intended during development: the product-information and carbon-footprint pages, which do not depend on a trained model, work independently of the yield module. The yield page depends on a trained and compatible model, and when that is not present the application reports this plainly. This graceful fall-back was one of the design goals, since a missing model file is a common situation when a project is moved from one computer to another or when the library version changes. The registration and login pages reject invalid input at the browser and again at the server, which confirms that the two validation levels are working as separate layers.

The outputs of the leaf-image and yield modules were treated throughout as demonstrations. We did not compare them with laboratory or field measurements, and we therefore make no claim about their accuracy in this paper.

C. Limitations

Several limitations should be kept in mind. First, the image-analysis and prediction outputs are not independently validated. Second, the yield estimate depends on the quality and the regional relevance of the dataset used for training. Third, the carbon-footprint result depends on the emission factors adopted, which differ between regions and crops. Fourth, SQLite is suitable for a prototype but is not meant for many simultaneous users. Finally, the application has not been tested with real farmers, so its usability for the intended audience is still unproven.

V. CONCLUSION

This paper described Croplytics, a Python and Flask web application that places leaf-image handling, crop-yield estimation, carbon-footprint estimation and agricultural product information in one platform, with a registration and login system built on server-side validation, hashed passwords and safe database queries. The work shows that a small team can build a modular agriculture assistant from common open-source tools, and that a clear separation between modules lets the application continue to work when one of its parts, such as a trained model, is unavailable. The project remains an educational prototype, and its estimates must not be used as the sole basis for farming decisions.

A. Future Enhancements

The next steps follow from the limitations above. The most important is to train and validate the leaf-image and yield models on properly labelled, region-specific data and to report measured accuracy. The carbon-footprint module can be improved with locally published emission factors and a wider list of farm activities. Other possible improvements are

support for regional languages, a mobile-friendly layout, weather data integration, an administrator page for managing product records, and a move to a server database when the number of users grows. A field trial with a few farmers would also give useful feedback about whether the interface and the wording of the results are understandable.

ACKNOWLEDGMENT

The authors thank the Department of MCA, Shri Jaykumar Rawal Institute of Technology, Dondaicha, for providing the laboratory facilities and guidance during this project work.

REFERENCES

[1] S. Wolfert, L. Ge, C. Verdouw and M.-J. Bogaardt, "Big data in smart farming – A review," *Agricultural Systems*, vol. 153, pp. 69–80, 2017.

[2] K. G. Liakos, P. Busato, D. Moshou, S. Pearson and D. Bochtis, "Machine learning in agriculture: A review," *Sensors*, vol. 18, no. 8, p. 2674, 2018.

[3] A. Kamilaris and F. X. Prenafeta-Boldú, "Deep learning in agriculture: A survey," *Computers and Electronics in Agriculture*, vol. 147, pp. 70–90, 2018.

[4] S. P. Mohanty, D. P. Hughes and M. Salathé, "Using deep learning for image-based plant disease detection," *Frontiers in Plant Science*, vol. 7, p. 1419, 2016.

[5] D. P. Hughes and M. Salathé, "An open access repository of images on plant health to enable the development of mobile disease diagnostics," arXiv:1511.08060, 2015.

[6] T. van Klompenburg, A. Kassahun and C. Catal, "Crop yield prediction using machine learning: A systematic literature review," *Computers and Electronics in Agriculture*, vol. 177, p. 105709, 2020.

[7] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[9] W. McKinney, "Data structures for statistical computing in Python," in *Proc. 9th Python in Science Conf.*, 2010, pp. 56–61.

[10] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[11] IPCC, *2006 IPCC Guidelines for National Greenhouse Gas Inventories, Volume 4: Agriculture, Forestry and Other Land Use*. Hayama, Japan: IGES, 2006.

[12] OWASP Foundation, "Password Storage Cheat Sheet," OWASP Cheat Sheet Series. [Online]. Available: https://cheatsheetseries.owasp.org

 This work is licensed under a Creative Commons Attribution 4.0 International License