



deti

universidade de aveiro
departamento de electrónica,
telecomunicações e informática

Online Camellia Cultivar Information System

Changes and Improvements



Nuno Santos Fahla

97631

Index

Context	3
Reputation System	3
Email and Twitter Notifications	5
Camellia Recognition System	7
Reporting an Identified Camellia	9
Deployment	11
Other Small Changes and Improvements	11
Sources	12

Context

This document serves as a log of all changes and improvements made to the Camellia Cultivar system. The system was developed by a group of students for the Projeto em Informática course, the changes reported here were made by one of the original members of the group.

Some features weren't implemented as planned and others needed to be fine tuned or tested more to find bugs and errors. This is the reasoning behind some changes such as the user's reputations, the notifications sent to users referring to certain changes in the system, the camellia photograph recognition system, the ability to report a misidentified camellia, the deployment and small fixes and quality of life changes.

Reputation System

Previous Work

When quizzes were submitted, the API received an array in which each answer was the combination of the user's answer and the correct answer. There were 9 pairs in each array. If a user wrote the name of a camellia that is unknown to the system or left it blank, the array would have less those entries.

After a submission, for each unidentified specimen, the system would calculate the probability of the suggested user answer being correct, with the percentage of the sum of all users who considered that specimen to be that cultivar, divided by the number of users that voted on this specimen to be any cultivar. The answer would be considered correct if the percentage was above 80 and there would be no reputation updates. As for the answers pertaining to reference specimens, André's model was used, to attribute reputation to users:

$$R_x = A_w \times \frac{A_{Cx}}{A_{Tx}} + V_w \times \frac{V_x}{V_T},$$

R_x : is the reputation of user x

A_w : is the weight given to the user answers

A_{Cx} : is the number of correct answers given by user x

A_{Tx} : is the number of total answers given by user x

V_w : is the weight given to the user votes

V_x : is the number of votes from user x

V_T : is the number of all user votes in the system

New Model

The first change was in the JSON object sent to the API. The field *specimen_id* was changed to *specimenId*, according to the *QuizAnswer* model, otherwise, the answers wouldn't be processed. Unidentified or empty answers, now are given the *id* '0', so it can be processed as a wrong answer.

The reputation attributed to users based on correct answers of reference specimen quizzes was changed. If wrong answers are given, no penalty to the reputation occurs and the more answers are correct in a quizz, the more reputation a user gains, based on the following formula $x^{1.49}/1.5$, where x equals the number of correct answers in a quizz. This was chosen with the intent to provide a good incentive to consecutive correct answers, to prevent rewarding users that answer randomly and while not giving too much reputation too soon to users who are frequently correct, since the maximum reputation per quizz does not pass 10 points.



As for the reputation changes from quizzes with unidentified specimens, when a confirmation occurs in the system, it fetches a dictionary containing the cultivars answered and their respective correctness probabilities, as key-value pairs, to search for the one that triggered the system to considered it "correct". Next fetches from the repository, all users who answered this specimen's quizz with that exact cultivar, to update their reputation. This time, the formula is a bit more complex, consisting of two parts, one based on each user's current reputation and the other based on how certain the system is of this answer being the correct one. The formula of the first part $15 \cdot e^{-(reputation/12)} + 3$ is intended to boost the reputation of users with a low score, and give a little bit less to expert users, although they all answered the same thing, this is made to even the playing field for new users and make sure the users in higher reputation are really trustworthy and didn't just "google it". The second part of the formula is $\frac{x}{40}$, where x is the certainty the system has of this being the correct answer. This is designed to be a small penalty in cases where a camellia is identified with ease just because it is popular, since it is more popular, the reputation given should decrease accordingly. The same formula is applied to wrong answers, except it subtracts a fifth of the result, instead of adding the result. Here we can see the full formula:

$$\frac{15.e^{-(reputation/12)} + 3}{\frac{x}{40}}$$

The probability of a specimen's quiz answer being considered correct also changed, now it involves the average reputation of all users who answered with a certain camellia, the number of answers the quiz has for that camellia and the number of total answers that quiz has (any camellia). A vote is a valid answer and a specific vote is an answer for a certain camellia, the formula for this

probability is $avg^{\text{specificVotes/totalVotes}} \cdot \frac{avg}{10}$. This is designed to significantly boost the result the higher the average reputation is while exponentially increasing the more the answers converge to the same camellia.

Another change was the introduction of user levels of reputation, simply put, there are 5 levels: Newbie, Recruit, Camellia Lover, Botany Expert and Garden Master. They are achieved at 0, 10, 20, 35 and 50, respectively and are represented in the user profile page, as such:

 nuno fahla	 nuno fahla
@ nunofahla@ua.pt	@ nunofahla@ua.pt
☆ 9.62 (Newbie)	☆ 47.14 (Botany Expert)

Email and Twitter Notifications

The emailing system has changed a bit. The previous service, Spring Boot Starter Mail wasn't working due to connectivity issues, therefore Javax Mail was used, another Gmail account created and an API access password generated. For this implementation, an Email java class was created, with simple *sender*, *receiver*, *subject* and *body* attributes, an Email Constants file was also created to store the email and access password for the API and an Email Service Java class, annotated with `@Service`, to handle the processing. On instancing this class loads all necessary variables(email, host, port, etc) and

instantiates a new Session with the credentials provided. The method *send()*, simply sends the Email class object sent in the method's parameter.

The system sends an email to verify all user's emails, upon registering, they must first access the platform via the link in their inbox, to complete registration.

An email is also sent when the system considers a specimen to be successfully identified, to all users who voted correctly and, of course, to the user who photographed and submitted that specimen. Here is a code example:

```
private void notifyUser(User usr, Specimen spcmn, Cultivar cltvr){
    String content = "Dear [[name]],<br>"
        + "Your identification a specimen (ID:"+ spcmn.getSpecimenId() +") with the cultivar "
        + cltvr +" has been deemed correct by the community, congratulations!<br>"
        + "Your Reputation will change accordingly, keep up the good work.<br>"
        + "Thank you,<br>"
        + "Cammelia Cultivar";
    content = content.replace("[[name]]", usr.getFirstName().concat(" ").concat(usr.getLastName()));

    Email e = new Email(EmailConsts.OUR_EMAIL , usr.getEmail());
    e.setSubject(sub: "A specimen you voted on was identified!");
    e.setText(content);

    if(emailService.send(e)){
        System.out.printf("\nMail Sent with success!\n");
    }else{
        System.out.println("ERROR sending mail!");
    }
}
```

A Twitter bot was developed, to communicate with the Twitter API, through the library Twitter4J. Tweets are sent every time the system confirms a specimen to be a camellia, giving information about the specimen that was identified and the correct camellia. To give more use to this powerful feature, tweets are also sent when the system reaches certain milestones. Right now, every 100 specimen identification requests are celebrated with a tweet and also for every 100 registered users, until 1000 total users, and for every 1000 users after that, tweets are sent out with current user count.

An important note about the Twitter account history is that it was used for a previous unrelated project, therefore justifying the music album related tweets up until february 2021 and the unrelated Twitter handle. This had to be done because a new account couldn't be created due to Twitter's phone verification system being down, thus blocking an important step in account registration. Here is a tweet's code example:

```
private void checkIfSpecimenMilestone() throws TwitterException{
    long specimenCount = specimenRepository.count();

    // sends a tweet every 100 new identification requests
    if (specimenCount%100==0){
        TwitterFactory tf = new TwitterFactory();
        Twitter twitter = tf.getInstance();

        twitter.updateStatus("Thanks to the community, we have reached "
            + specimenCount + " specimen identification requests!");
    }
}
```

Camellia Recognition System

Users, when submitting a new identification request, should add a photograph of the spotted camellia. But what if the photograph is something other than a camellia? A small system was developed to mitigate this problem and reduce the amount of requests admin/moderator users have to manually analyze, the solution is to use a model for automatic classification of flower types in photographs, should the system verify the existence of a camellia then the user can proceed, should it not then the user is notified.

The base model solves a Classification problem, comes from PyTorch, and is based on a public dataset of flowers(found on *Kaggle*), which has 102 different flower types. With this data a Neural Network was developed to predict what type of flower is in an image. Using *Jupyter Notebook* and the Python module *PyTorch*, the file *camelias.ipynb* was developed to load the basic model, process the dataset, train the model and export the result to the file *densenet_model.pth*, as a persistent object to be later loaded.

There is a Flask API responsible for handling the Camellia Recognition System. Through an endpoint that receives an URL of an image as a parameter, */predict?url=*[], an image is loaded and, to be as accurate and unbiased as possible, transforms the image to something similar to the samples used for the model training. With *PyTorch*, the model is loaded from the file and based on the patterns found when training it, it analyzes the incoming image's pixels and outputs a dictionary with the 102 flower types and their respective probability.

```
def predict(model, image):
    # number 96 is the camellia code
    image = parseImage(image)
    results = {}
    topResults = []
    image = image.to(device)
    outputs = model(image)
    _, preds = torch.max(outputs.data, 1)
    for o in range(len(outputs[0])):
        results[o] = float(outputs[0][o])
```

If the *Camellia* is in the top 5 of the resulting flower types, the image is approved. If *Camellia* is in the top 15 the system suggests the picture isn't a very good one and to try again. If *Camellia* is below that, the image is then rejected. In the dictionary *results*, camellia has the key with value 96.

```

for x in range(5):
    maxV = max(results, key=results.get)
    del results[maxV]
    topResults.append(maxV)

if 96 in topResults:
    print("best case scenario")
    return Response("", status=201, mimetype='application/json')

for y in range(10):
    maxV = max(results, key=results.get)
    del results[maxV]
    topResults.append(maxV)

if 96 in topResults:
    print("acceptable case scenario")
    return Response("", status=200, mimetype='application/json')

return Response("", status=406, mimetype='application/json')

```

However, this model isn't the most appropriate to the problem, because it is a Classification, not a Recognition one. It always outputs a probability of an image being a camellia, even if it is of someone's cat, so there is a small chance for a random image to pass verification, should it be closer to an image of a camellia than it is to an image of other flower types.

In order to have an Object Recognition Model as desired, it would not only be necessary to have way more computational power but also a publicly available and pertinent dataset of camellias, or as an alternative, time and expertise to develop one. The work developed serves as proof of concept to solving the problem, for a more precise implementation it would just be required to train an object recognition model and a pertinent dataset.

Reporting an Identified Camellia

After enough users have voted for a specimen to be a certain camellia, the system might deem it correct and promote that specimen to a reference camellia. That being said, what if enough users decide to vote incorrectly? Or just share a common source that happens to be wrong? In that case, a certain user who spots the mistake, can use the *Report* button on any Camellia page. This button displays a previously hidden text box so the user can justify the Camellia Report Request and send it to the database:

Scientific Name C. japonica hybrid 'Ack-Scent'

More Photos

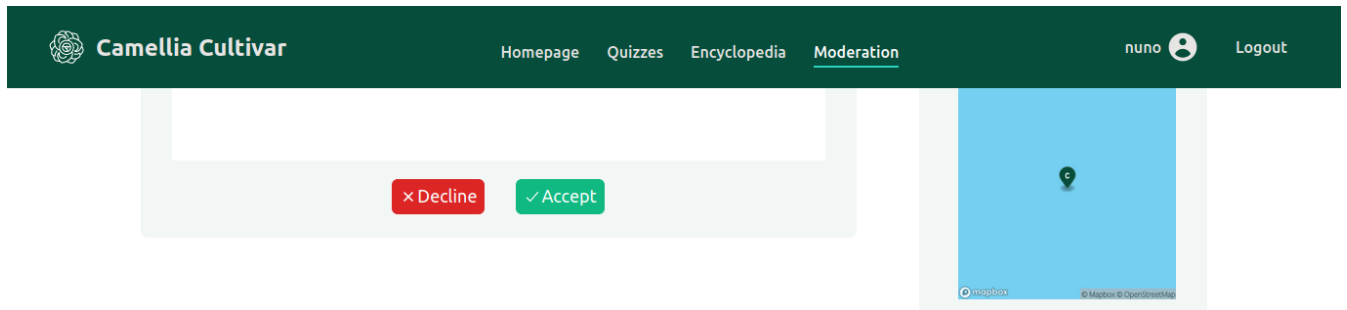
Report

Judging by the petiole, can't be an Ack-Scent, must be a Gloria

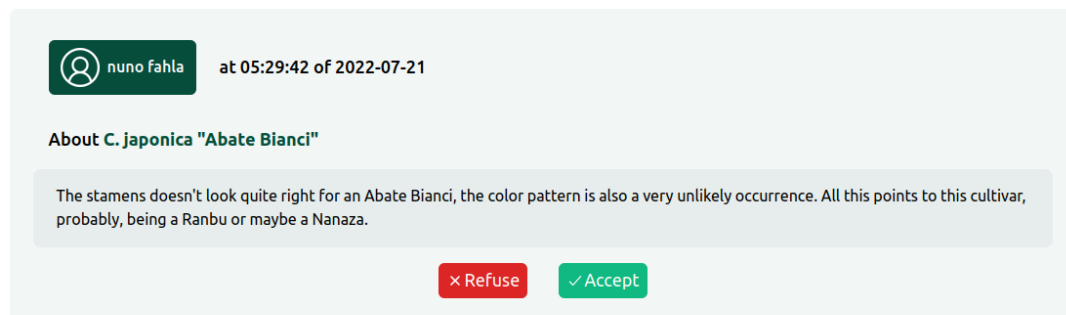
A HTTP post is sent to `/api/requests/report` with the reported cultivar ID and the user's justification as a JSON object. The data comes in *ReportRequestDTO* format and, if the API can confirm the user who submitted it, then the data is transformed into a new *ReportRequest* object and saved in the corresponding repository. Below, we can see the report submission javascript method in the React app:

```
const submitReport = () => {
  let report = {
    cultivarId: camellia.id,
    reportText: answerText
  }
  let user = JSON.parse(localStorage.getItem('userToken'));
  axios.post(`${proxy}/api/requests/report`, report,
    { headers: { Authorization: `Bearer ${user.loginToken}` } })
    .then((_response) => {
      if (_response.status === 201){
        loadMore2();
        setAnswerText("");
        console.log("reported successfully");
      }
    })
    .catch((_error) => {
      return
    })
}
```

The request is now in the database and is now represented on the *Moderation* page, in the reports section, through HTTP GET from the */api/moderator/report* endpoint. Information is shown about the user who submitted the request, the justification, the time/date and, of course, the cultivar the request is about. A screenshot from the Admin/Moderator's view:



Reports



In case there are no Report Requests:

Reports

No reports pending

The Admin/Moderator can now refuse or Accept the Report Request. If it is refused, the web app sends an HTTP DELETE to */api/moderator/report/refuse/{id}*, which deletes the Report Request with the given ID from the database. If the the Report is accepted, the web app also sends an HTTP DELETE, but this time a camellia ID as well, to the endpoint */api/moderator/report/accept/{id}/{cId}*, which instructs the API to delete the Report Request with the provided ID and also delete the cultivar with the provided cultivar ID, from the database. Code example:

```
@DeleteMapping("/report/accept/{id}/{cultivarId}")
public ResponseEntity<String> acceptReportRequest(@PathVariable(value = "id") long requestId,
                                                  @PathVariable(value = "cultivarId") long cultivarId) {
    System.out.println("accepted report request with id:" + requestId + ", deleting cultivar with id: " + cultivarId);
    if (checkRole()){
        reportRequestService.deleteReportRequest(requestId);
        return ResponseEntity.status(HttpStatus.ACCEPTED).body(cultivarService.deleteCultivarById(cultivarId));
    }
    return ResponseEntity.status(HttpStatus.FORBIDDEN).body(body: null);
}
```

Deployment

The Associação Portuguesa de Camélias, our “product owner”, did not have the infrastructure to support our system, therefore, the deployment was made through Microsoft Azure, which made our system publicly accessible for a short period of time. In order to make the system available again, it was deployed to a virtual machine in our department. First the URL <https://192.168.160.226:8085> must be accessed to accept the self signed certificate, then the system is easily accessible on <http://192.168.160.226:3000>. More details in **Sources**, below.

Other Small Changes and Improvements

- Both backend and frontend are in docker containers
- Set the docker-compose HTTP request timeout to a higher value, to prevent image fetching failure on slow connection internets
- Reference quiz answers are processed before To Identify quiz answers, in order to calculate reputation that doesn't depend on other user's answers first
- Unidentified specimen quizzes now appear in quiz groups
- Quiz groups can now have less than 3 unidentified specimens
- Quiz groups now need to have 6 reference specimens
- Made it so all cultivars and specimens that initialize the database have photos
- Reference cultivares in quizzes refer to actual different cultivars and not all the first one
- Removed useless imports and variables from backend
- Removed Moderator reference in Requests Class, since it served no purpose
- Request's getters provide more varied and pertinent information
- Report Request and Report Request DTO classes created
- Created an endpoint to delete cultivars
- API container now always restarts if it fails to connect to DB(in case it boots before it)
- SpecimenDTO class now has an image url attribute

Sources

- GitHub Repository: <https://github.com/broth2/Camellia-Cultivar>
- Original GitHub Repository: <https://github.com/Camellia-Cultivar/Camellia-Cultivar>
- Web App: <http://192.168.160.226:3000>
- SpringBoot API: <https://192.168.160.226:8085>
- Flask API: <http://192.168.160.226:5000/predict?url=>
- PostgreSQL database: <https://192.168.160.226:5432>
- Twitter Account: <https://twitter.com/onthisdayalbum>
- In order to run the application locally simply execute the `run_docker.sh` bash script. Relevant URLs above
- Email service executed through the account cultivar.camellia@gmail.com
- Database is initialized with admin nunoadmin@ua.pt and normal user nunofahla@ua.pt, both with password `batman123`