This document describes a technical challenge that forms part of the Hectre recruitment process for Spectre back-end engineers.

It should not take more than one working day. Don’t worry if the project isn’t complete by then

- just email back what you have with some notes on where you got to and what’s left to do.

# Background

Spectre AI tool is one of Hectre’s products that utilises machine learning, computer vision, and cloud computing to provide size & colour distributions for fruit bins within seconds from uploaded images/videos.

Here is one example of uploaded images for Kiwifruit with expected diameter lines.



Your task is to design and deploy a backend machine learning solution that could allow the frontend to upload image(s) and save the information to a structured database. The tasks are:

1. Design the solution/service architecture (ideally combining with cloud platform)
2. Deploy one Python backend API endpoint for uploading one or many images with at least the following information:
   * Image ID (Primary key)
   * Created date (UTC format)
   * Fruit type
   * User email
   * Tenant ID (Assume this is the ID of the user)

And save this information into a relational database.

My solution to this objective is given in the files 1) “*initialize\_backendAPI.py*”, 2) “*db\_add\_initial.py*” and 3) “*db\_query.py*”. Basically, the code is run from 1) to 3) as labelled and deployed locally to start. “*initialize\_backendAPI.py*” utilized the Flask SQL Alchemy and RESTful API to create a database and provided a way to input data into the database through parsed arguments. I also added an additional input argument for the file path of the image. My assumption for this problem was that the input arguments were already available to me through the front-end user interface, and the user interface passed this information to my code as arguments. To test the code I used “*db\_add\_initial.py*” which inserted 3 samples of manually defined rows into the database. Basically the 3 samples were manually parsed, while in reality they would be automatically fed into the script. “*db\_query.py*” then did a few get/put/delete methods, which I defined, on the database to check that I created the database correctly.

In terms of uploading the images, I specified the image path as an input argument and read the image in with open-cv, then saved it into a folder as I thought filesystem is more efficient that SQL databases for storing images. I just used the image provided in the first page as input/output such that the input path was simply my current repository (re-used it three times for simplicity), and the output path was just to a folder I just created inside the repository. An assumption is that the user interface would take a picture of a fruit bin and the path that it is stored on the local device of the user would be feed into my code so that it can load the images. I also tried some other code at around line 193 of “*initialize\_backendAPI.py*” which attempts to route directly to an interface which receives the data, then stores it into the SQL database. I could not get it to work so I commented it out. Also I think this is an inefficient way to store data.

For the “relational database” part, it is more efficient to store user emails in another table, and link it to the main table with Tenant ID. I assumed tenant ID meant the ID of a user. The second table would just have email and tenant ID, while the main table would have everything apart from email. This way the same email for the same user ID is not stored repeatedly. This was done in my code using a one-to-many relation between the user table (email + ID) and the image table (main table containing all other information). We can then access the table containing all users, linked to the table containing all images with the tenant ID. I did not implement the patch/delete methods for the user table but it would basically be a copy paste of the current methods except there the query would be for user table instead of the image table.

In my test script “*db\_add\_initial.py*”, I purposely put the same user ID twice to make sure that I could handle the same user inputting a new image, such that only the image table needs to be updated and not the user table.

The API would also have another method to execute a pre-trained machine learning model every time a new data sample is uploaded, and return the prediction/summary of the fruit sizes/colors etc. This information would be returned back to the user through the front end and also stored in the filesystem image database with an identifier that can match the raw input image and output image. Another table could potentially be created to match the IDs of the input and output images (two columns), and have a one-to-one relation with the main table containing the image information.

1. Demo test with simple Unit test(s)

*“test\_backend\_API.py”* script provided to check objects defined in “*initialize\_backendAPI.py*”.

1. Document the detailed steps on how to deploy and run the code in Mac/Linux Environment

I’m not sure exactly where I should outline this info so I am doing it here. Its also included in the README. Most of the work involves installing the dependencies. You can have all the commands below inside one single bash/shell script so you don’t have to do each step individually. But for now I with just list the commands and describe them:

1. SSH into the desired server (could be cloud, hpc cluster, or just a local server)
2. Set up virtual environment (or could be container):
   1. mkdir Virtual\_ENV
   2. cd Virtual\_ENV
   3. virtualenv -p /usr/bin/python3.6 Virtual\_ENV

This will then install python/pip and other basic tools

1. Activate virtual environment:
   1. source Virtual\_ENV/bin/activate
2. Install the python dependencies which are in the requirements.txt file that I have provided in the repository:
   1. pip install –r requirements.txt
3. The python code can then be run through commands in the terminal etc:
   1. python example.py

But make sure to activate the virtual environment with #4 every time you reconnect to the server.

1. Run “*initialize\_backendAPI.py*” which sets up the API so data can be fed in to the database in real time.
2. Deploy the one of core blocks of solution on cloud with Infrastructure as code (IaC) (In case of designing solution with cloud)

The overall diagram of the proposed solution is given on page 10 of this document under the “deliverables” of this project, with a brief description of the parts which will be on the cloud. Here I have listed a summary of the steps to take to deploy the solution given in task 2. I only provide the rough steps as I did not have enough time to set up the cloud accounts and plans. I have chosen Heroku as it seemed the simplest, and I have not deployed any models like such before. The steps are as follows:

* Install Heroku command line interface and create a new app, named ImageApp, using the following command in the Heroku terminal

heroku create ImageApp

* Install gunicorn using pip which provides the python web server to run the app on Heroku

pip install gunicorn

* Create a procfile which is used to declare commands run by the App on the Heroku server.

echo web: gunicorn app:app > Procfile

* Create a remote database (sqlite was used) on the heroku server with the free plan (hobby-dev)

heroku addons:create heroku-sqlite:hobby-dev -- app ImageApp

* Update the SQLALCHEMY\_DATABASE\_URI variable with the new database URI. The new URI is retrieved with

heroku config --app ImageApp

* Commit and push files to the Heroku master branch

git add .

git commit --m “heroku commit”

git push heroku main

* Migrate the local database to the new instance of the remote Heroku database through the Heroku python terminal

heroku run python

from app import db

db.create\_all()

1. One of Spectre AI features is grading fruit size. Assume that we have to develop a machine learning solution to find the pixel diameter of Kiwifruits. From the visualization image above, can you describe your practical solution:
   * Which kind of ML model should you use? Why?

The primary group of neural networks to be considered is ones designed for object detection/instance segmentation. For example Mask R-CNN, SSD, YOLO etc. By performing segmentation of all objects of interest in an image, the subsequent measurements of individual objects can be easily calculated from the mask. The presence of the segmentation masks also allows for easier extraction of fruit measurements in oddly shaped fruits.

Mask R-CNN has several advantages over single-shot methods such as SSD or YOLO. Mask R-CNN utilizes two parts, the convolutional feature extractor, and the region proposal network, both being trainable end-to-end. The convolutional feature extractor produces an abstract representation of the initial input image by mapping it onto a larger number of feature maps as opposed to the 3 feature maps of the initial image (represented by R/G/B channels). The abstract representation then provides more information to the region proposal stage. The regional proposal network then learns potential regions of interest with bounding boxes (defined by 4 coordinates and 1 value for the confidence) which are then segmented by a subsequent network. In this case, the convolutional feature extractor can be re-used for segmentation to avoid additional parameters. The major advantage of Mask R-CNN is the fact that the detection and segmentation are unified into a single framework, making the approach extremely robust given sufficient training data. Prior research has also demonstrated its efficacy on segmenting small objects, which is important for fruit segmentation. The fully trainability of the region proposal network also means computational costs with estimating potential regions of interest are minimized as the network learns this part on its own. However, the major disadvantage is the difficulty in training the network, as several objective costs has to be balanced to obtain a desirable outcome. These optimization objectives include the accuracy of the region proposals (classification loss and bounding-box regression loss), and the accuracy of the final detected objections (also with classification loss and bounding-box regression loss). The network then has to be optimized for accuracy of segmentation in each bounding box, introducing an overlap loss for segmentation. The current state of literature suggests Mask R-CNN is the state-of-the-art in instance segmentation.

Single-shot methods such as YOLO (you only look once) and SSD (single-shot detection) are much easier to train than mask R-CNN. However, less flexibility is offered as the regional proposals are predefined manually instead of learnt by the network. The method breaks an input image into sub-regions, then optimizes by adjusting the set of estimated bounding box coordinates in each sub-region to the ground truth location of the object of interest. While this allows for the entire processing to be done in one step, the loss function is significantly larger as it is now N x N x (5 x B + C). where N = the number of sub-divisions in each of the x/y axis, 5 is for 4 values defining the bounding box and 1 value defining the confidence, and C is the score for the class prediction. In some ways, this loss function is also hard to optimize. However, the execution speed of such methods is significantly slower, as much inefficiencies are introduced by the repeated computation of the manually defined set of bounding boxes. This may be problematic if the solution needs to be deployed in real-time at a high frame rate.

If the task to be address can be achieved through the use of a bounding box outline on an object of interest, then YOLO/SSD should be considered due to the simplicity in the method. However, if the objective is to eventually obtain accurate measurements of all objects of interest in an image, Mask R-CNN can potentially be more accurate and computationally efficient. A Mask R-CNN with a shallow Res-Net convolutional feature extractor is also fairly lightweight in terms of memory consumption in comparison. This, in addition with using deep neural network compression, may possibly make mask R-CNNs embeddable into apps.

* + Any idea about data, training ML?

The most important aspect of any machine learning pipeline is the datasets. Images should ideally be high resolution, sharp, and contain high contrast. The images would also be fully labelled. Ideally, each fruit (in a bin for example) will have their own independent manually segmented mask e.g. an image with 50 apples has 50 different annotations of each individual apple. The higher the quality of the data label, the more flexibility and potential for future development can be made available, as labels with higher complexities can be manipulated in more ways. Worst comes to worst, a complex label can always be simplified (merge all 50 masks into one mask for example), but a simple mask cannot be made more complex easily. It is therefore imperative to optimize the manual labelling workflow from the very start so that time is not wasted, as human labor is expensive and time-consuming.

The machine learning algorithm would be trained on the largest dataset possible, with the dataset being split into train/validation/test (usually 70%/15%/15% in the case of very large datasets, 80/10/10 also works). The validation accuracy is constantly evaluated during every epoch to ensure the algorithm is not over fitting to the training set. Once the accuracy stops increasing on the validation set, the algorithm is then tested on the test set to check the accuracy. The difference between the training accuracy and the validation accuracy is a result of variance in the dataset, and the difference between the validation and testing accuracy is a result of overfitting of the validation set.

Once the algorithm is trained, the network parameters can then be saved and used for initialization in the next training session so backpropagation and stochastic gradient descent does not need to start from scratch. This can be applied to every additional training session after the first, if we want to fine-tune the network parameters. Transfer learning can also be leveraged to extend the algorithm onto different fruits i.e. an initial large apple database can be used to pre-train a network which is then fine-tuned on a much smaller orange dataset resulting in accuracy orange detection despite the small dataset.

Other practical considerations would be the training time and network complexity trade-off, speed of data access from memory during training, systematic methods of hyper-parameter tuning, which are all important in minimizing the computational resources and time taken to develop a satisfactory model. Sometimes it may be better to tune a smaller model many times in the same time period it would take to train a larger model once. All the connections from the database to the network should be optimized so the training process is not bottlenecked by the speed of loading/saving data.

* + Any idea about organizing/deploying ML model on production?

After obtaining a fully trained model such as one in Tensorflow, the weights should be frozen to fix all the trainable parameters for deployment. The model would then be saved into a protobuf (Tensorflow offers this functionality by default) or some other executable format. During deployment, a simple Tensorflow wrapper would then be written to load the model given defined input size, run the data through the model, and create the prediction. In total this would be less than 10 lines of code. This would thus require Tensorflow (gpu version) to be installed on the server running the predictions. The model file can then be updated on the server when a new model is trained, and the remaining code wouldn’t change if the same model is used.

It may also be possible to convert the Tensorflow model into a self-contained executable such that you would not need to install all the dependencies. This would be ideal for embedding the model into an app. This would also require the use of deep compression as CNN models are usually at least a couple of hundred megabytes. The networks should ideally be compressed to only take up 10-50 MBs on a smart device. Deep compression has been shown to produce 30-50x compression on popular ImageNet models such a VGG-Net from 500MB to 15MB or less.

Deep compression can traditionally be performed in three stages: pruning, quantization, and Huffman coding. Pruning would first mask out certain weights by fine-tuning the network on the data. This would then cut out many weights below a certain threshold, and the resultant weight matrix can be stored in a sparse matrix. Quantization involves converting the tf.float32 data types (32-bit) used to store weights into much less bits such as 3 or 4 which significantly decreases the size of the network in terms of memory. This is achieved through the use of K-means clustering. If we wanted to convert the network into 4-bit from 32-bit, a K-means clustering algorithm of K = 16 (16 values to store each weight) would be run so that all weights are optimized towards only 16 values distributed across the entire range of values in the original set of weights. Overall, this is basically to convert the weights into a less precise representation. Studies have however shown that getting rid of some of the decimal points actually does not impact the accuracy significantly, but offers multi-fold compression. Huffman coding is the most complex and basically takes advantage of the biased distribution of values to encode the most frequently occurring values with shorter bits and less frequently occurring values with longer bits. This does not result in as much compression as the other two methods but may help depending on the specification for the size of the model.

* + Post-process algorithm idea to get the expected diameter size

If the output is a mask of each individual fruit and the fruit is assumed to be spherical, then the diameter of the fruit can easily be extracted by measuring the diameter of the masks. The real diameter can then be calculated by multiplying the number of pixels with the image resolution to obtain a measurement in cm or mm. Multiple diameters can be taken to increase the confidence of the estimation, as this can be vectorized and does not contributed to much more increased computational cost. I will discuss measuring oddly shaped fruits in the section below.

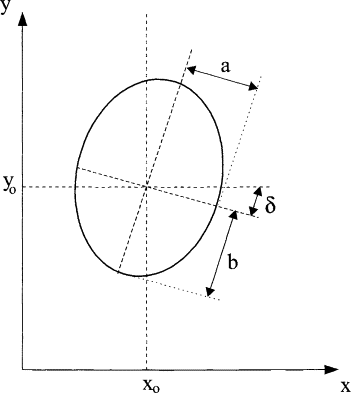
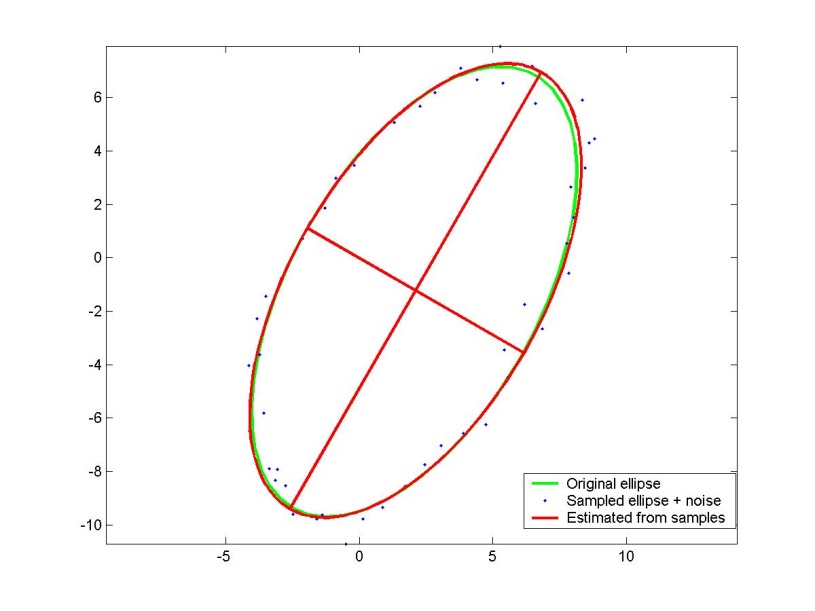
If the output is a bounding box of each individual fruit (less difficult task than segmentation to train but less accurate), then edge detection algorithms may potentially be used to get the outline of each fruit. This would be performed after the image has under gone contrast normalization. Thresholding may be another step to extract the fruit from the background, but this would not work too well if the fruit is among other fruit, so it might just be an extra step in the process. For spherical fruits like apples, the bounding box itself may be used to directly measure the diameter. But for elliptical shapes such as kiwifruits, a potential method would be to calculate the orientation of the fruit inside the bounding box. If the shape is elliptical and the fruit is not exactly aligned with the box (not vertically or horizontally oriented), then you may be able to use edge detection and where the edges touch the bounding box to figure out the location. i.e kiwifruits at 45 degree offset from the bounding box will only touch two opposite corners, and not the other two corners. After figuring out the orientation, you would then be able to draw a line for the diameter such as displayed in the image on page 1.

Once the diameters of all fruits in a bin are obtained, a further step for checking can be performed by performing a similarity test between the current distribution obtained versus a collated distribution of all previous measurements for the same fruit. If the distribution is shifted to the right or left, there may be an offset error due to, for example, incorrect pixel resolution considered. If the shape of the distribution is odd (e.g. bimodal when it’s supposed to be unimodal), then there may be errors in the masks or method of extracting the pixel diameter.

A more “fancy” approach would be to embed the diameters of the fruit as an additional label in the training datasets. This would be done either fully automatically on known distributions or semi-automatically with human quality control. The Mask R-CNN network can then be trained with an additional branch in the output to predict both the mask and the diameter of each fruit. This loss function is the case would be a regression loss trained with a mean squared error objective. Similarly, if we are only interested in the diameter (and not for example, the volume which estimates the weight), then we can train a single-shot network to simply predict the bounding box of each fruit along with its diameter as a single additional output. However, this would be less scalable to other fruits.

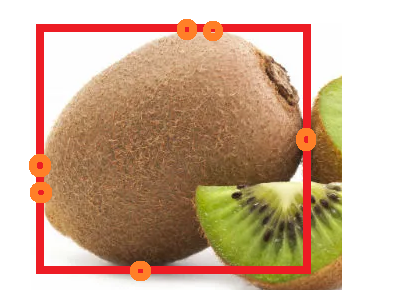
* + How can we best deal with the flat shape of a Kiwifruit?

If a Mask-R-CNN is used, then this makes it significantly easier as the oval-mask of the kiwifruit is available. After obtaining the mask, then an ellipse would be fitted to the shape. A representative ellipse is shown below for illustration, along with another sample for fitting the ellipse to a shape:

The algorithm would use a least-squared criteria to optimize the major and minor axis (a and b) as well as the orientation (delta). Conic equation to represent the ellipse would be ax2+bxy+cy2+dx+ey+f=0, where a-f are parameters to be optimized. Points can initially be sampled from the boundary of the mask of the kiwifruit. An outline can be simply obtained by subtracting a 1-pixel erosion of the mask from the original mask. These points can then be used to fit the ellipse. After obtaining the ellipse, we can then define the diameter of the kiwifruit with major and minor axis. The only issue would then be to find out the orientation of the kiwifruit with respect to the camera angle, as some times it can be orientated so the tip is directly facing the camera so we only see a circle instead of an oval. We would then need to pre-define exactly what measurements we would want to obtain for the diameter (major axis or minor axis). If we have sufficient samples of kiwifruits, we can potentially find the ratio between the major and minor axis, and use that ratio to compute the major axis if only the minor axis are shown in the image due to the orientation.

In the case of the output that is a bounding box. We can use edge detection to find the intersection between the bounding box and the edges of the fruit as an initial estimate for the points to sample. An example is shown below with the orange points indicating the initial sample set of points used to fit the ellipse:



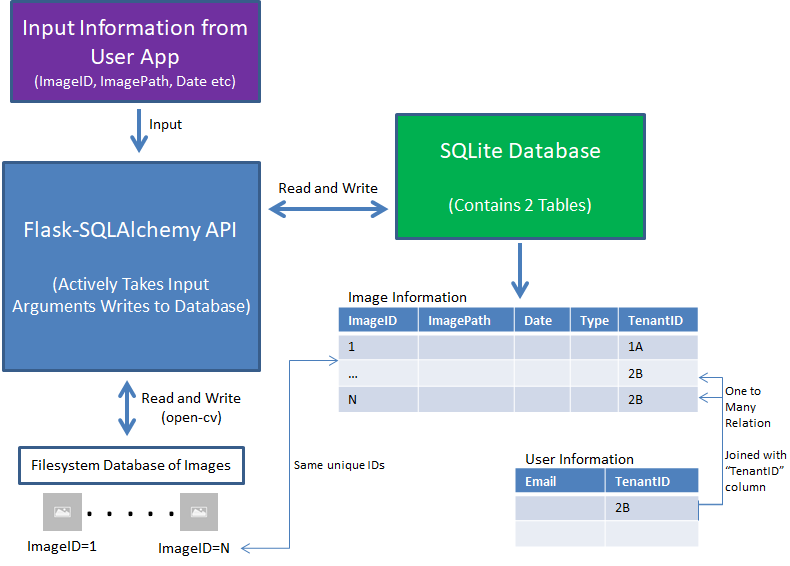
We can then improve the point sampling by extending it with edge detection to sample more points which are near the intersection with the bounding box. We would also want to ensure the ellipse is contained within the bounding box. This would pretty much guarantee the ellipse is a relatively accurate representation of the outline of the kiwifruit. If somehow there is no intersection with the bounding box and the kiwifruit outline, we may want to shrink the bounding box in incremental steps until we do get an intersection.

Lastly, if the kiwifruit is more of a rectangular shape than oval, that shouldn’t be an issue as we can still fit the ellipse if we are only interested in measuring the diameter.

The deliverable product should contain:

* A design solution with diagram(s) and explanations

The flow diagram below shows what has already been implemented in my code. Other details about additions or improvements are detailed above under the individual tasks (1-5)



The Flask-SQLAlchemy API (blue box) and Database (green box) would be deployed to a cloud server. The database will also be migrated to the cloud. All the setup would thus be down on the cloud in either a virtual env or container (with docker) instead of setting up locally as I have done in this exercise.

* A Python application

Python files are all in the repository.

* A .git folder with the commit history for the project

Folder provided in repository.

* A README that explains how to install and use the project

README.md provided in repository.

The project should work on macOS or Linux.

Feel free to document any assumptions made, or ideas for improving the project. Don’t upload the project to a public repo since it might be unfair for other candidates.

# Application notes

* The Python scripts, packages are compatible with Python 3.6, 3.7, 3.8.
* Use Flask API or other blueprint for backend APIs
* Use a PostgreSQL or other structured database, including DB script/code for creation.
* It is better to use ORM (Object Relational Mapper) and a suitable backend design pattern.
* Ideal unit test is Pytest but not limited to this option