Capstone 2: Milestone Report 1

Image Recognition – Fruit Classifier with > 90% accuracy on unseen data.

Objective:

The objective is to train a CNN (Convoluted Neural Network) to recognize everyday fruits in at least 5 categories. This would be an end to end activity which will involve data curation, data transformation, augmentation, training, validation and testing to achieve at least 90% accuracy on test set (unseen data). They will be trained using 300 (100 x 100 size) color images of each category that are collected from free online resources. For testing, 10 images will be taken for each class. There will be five categories of **fruits - Red Apples, Strawberries, Green Grapes, Kiwi and Blackberries**. The objective is to build a foundational level classifier that can be scaled up to applications in numerous technologies like facial recognition, computer vision, self-driving cars etc. The CNN will need to ensure that it gets over all the below challenges in achieving its target accuracy.

To build the classifier, the project will be broken down in to smaller tasks, each of which will be a mini milestone, whereby a conclusion will reached. At a high level, the tasks would be:

1. Data collection and curation
2. Data cleaning and transforming
3. Data wrangling – conversion and preprocessing
4. Training model

Step 1: Data Collection and curation: Using Flickr API and Google Image Search URLs

Target is to have a "good" (post cleaning) collection of 300 images for each fruit at least for training. In order to do this, 1000+ images of fruits are downloaded for selection and cleaning. Data is collected mainly using 2 methods: *Flickr* and *Google*. Collection methods are very distinct and both are employed to overcome the limitation on number of free-to-download images. Although you can specify a high number of images to be downloaded in Flickr API, for example 1000, the actual number of relevant images that get downloaded is less. Some of the images do not get downloaded due to various reasons such as connectivity, download restriction and sometimes also have very poor quality. If the number of images from one source is not enough, the set will be supplemented by the other source. For example, to supplement the Flickr data set, Google images are downloaded via fetching the URLs using Java Script in to a txt file and downloading them in bulk by using *urllib*. Note: Flickr API downloaded more than 900 relevant images of Strawberries and so it did not need to be supplemented by Google dataset.

Using Flickr API:

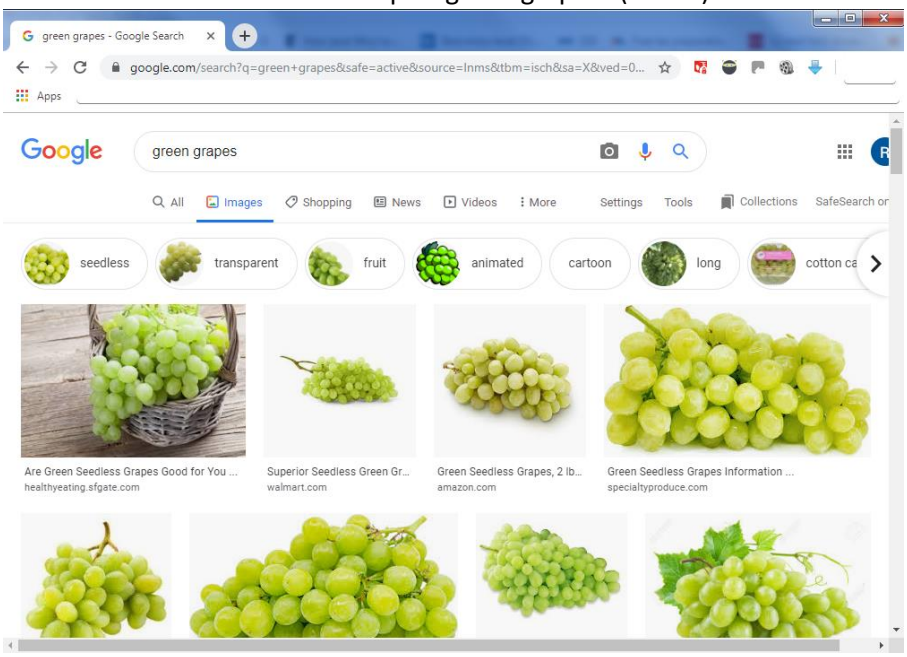
To use Flickr API, first the API service was registered after which they emailed a KEY and SECRET that need to be used every time an API call is made. A python function was written to download via the API by passing parameters of 'keyword', 'size' (optional) and 'number of images' (optional).

```
def download_flickr_photos(keywords, size='original', max_nb_img=-1):
```

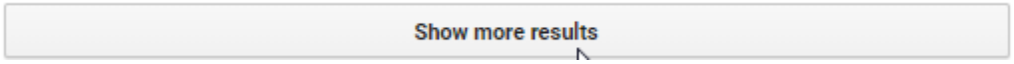
Using this function, the keywords of 'green grapes', 'kiwi', 'blackberry', 'lemons' and 'red apples' was passed to download 1000 images of ‘square’ size (150 x 150), slightly larger than target size (100 x 100).

Using Google Image Search:

First the keywords to be searched were put in Image Search option of Google for the images to be loaded in the browser. For example “green grapes” (below).



Next to load maximum number of images in the browser window, the scroll bar was clicked through bottom until a button of “Show more results” appeared.



After clicking on that, the browser loads more images and needs to be scrolled until no more images can be loaded. This is the full set of images that are available by Google to be downloaded. To automate the download, we need to get link of each image. To achieve this, the following javascript needs to be executed in the “Developer tools” of Chrome browser.

```
// Step1: pull down jquery into the JavaScript console
var script = document.createElement('script');
script.src = "https://ajax.googleapis.com/ajax/libs/jquery/2.2.0/jquery.min.js";
document.getElementsByTagName('head')[0].appendChild(script);

// Step2: grab the URLs
var urls = $('.rg_di .rg_meta').map(function() { return JSON.parse($(this).text()).ou; });


// Step3:write the URls to file (one per line)
var textToSave = urls.toArray().join('\n');
var hiddenElement = document.createElement('a');
hiddenElement.href = 'data:attachment/text,' + encodeURI(textToSave);
hiddenElement.target = '_blank';
hiddenElement.download = 'urls.txt';
```

```
    hiddenElement.click();
```

This script will download '*urls.txt*' file will have the link of each of the images in browser. This file is renamed for each fruit class and used in a python function 'download_imgs'

```
def dowload_imgs(arg_str):
```

The argument 'arg_str' is used to pass the arguments to the function. For example:

```
# Download Blackberry images
arg_str = "--urls H:\\DataScience\\Cap2\\urls_blackberries.txt --output data/Full_Set_Raw/Blackberry/"
download_imgs(arg_str)
```

This downloads the entire set of images that act as 'raw' image source for this project. Next, we will look at data cleaning.

Step 2: Data Cleaning

This step consisted of manual and auto deletion of images.

Method 1: Auto-deletion via OpenCV

In this, the second part of function 'download_imgs' is employed to automatically delete the images that are bad. Bad images are classified as the ones that cannot be loaded by OpenCV's CV2. This is factored in to the function itself and the 'culling' part is run after all images are downloaded. From the execution log, you can clearly see which images were deleted.

```
In [47]: arg_str = "--urls H:\\DataScience\\Cap2\\urls_kiwi.txt --output data/Full_Set_Raw/Kiwi/"
         dowload_imgs(arg_str)
```

```
[INFO] downloaded: H:\DataScience\Cap2\Kiwi\\00000587.jpg
[INFO] downloaded: H:\DataScience\Cap2\Kiwi\\00000588.jpg
[INFO] deleting H:\DataScience\Cap2\Kiwi\00000025.jpg
[INFO] deleting H:\DataScience\Cap2\Kiwi\00000040.jpg
[INFO] deleting H:\DataScience\Cap2\Kiwi\00000047.jpg
```

Method 2: Manual

For purposes of data curation, each image was manually looked up and kept only if it fit the following rules:

1. Image is clear and is relevant representation of the object. Any non-relevant images were deleted.
2. Image is of whole fruits and not slices. This is because fruits can be carved/sliced in different ways and we would need a bigger set than 300 images to train different representations of the fruit. Exception to this is kiwi as there are more images of sliced Kiwi than whole Kiwi.
3. Image does not have other fruits in the picture, especially the ones we are training the CNN on. This would lead to confusion and misclassification.

4. Images have similar distance from which it is taken. This is to avoid extreme zoom (Macro) or big distances to mislead the CNN. To be able to get to this level of sophistication, you would need a bigger set and even then, it would be hard to classify extremes/outliers.

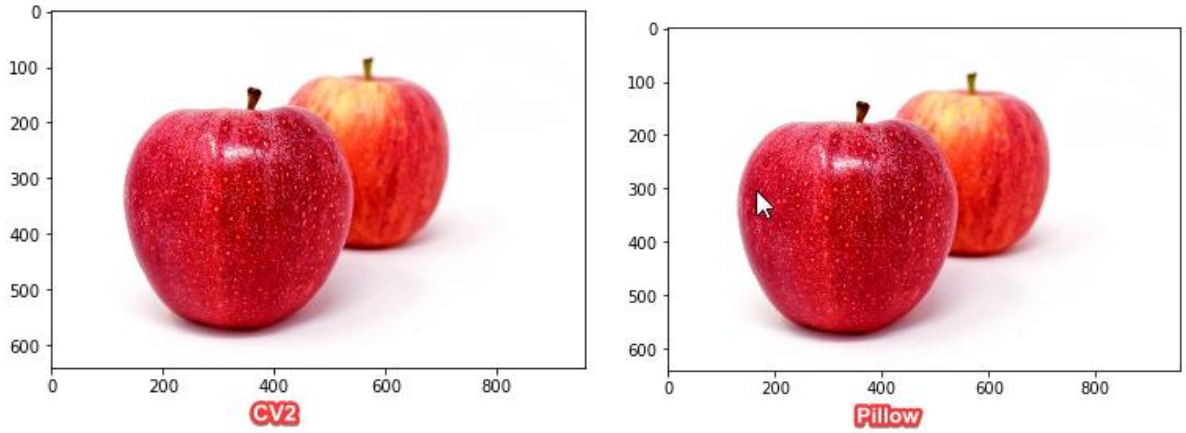
Step 3: Data Wrangling - Image conversion and pre-processing

The downloaded images are in various shapes and need to be standardized in to thumbnail size, (100x100) to be trained. The original image will be preserved whilst downsizing, but this gives rise to the issue of padding. Some images are rectangular and need to be transformed to thumbnail squares preserving the aspect ratio, and in the process will have some areas that need filling. The approach is to fill these areas with white pixels as majority of the images have a white background. Also, there are various libraries available in Python that accomplish the same thing. For purposes of the project, two of the options are compared, CV2 and Pillow.

Compare: CV2 vs Pillow

A. **Image display:** First we see if there is any noticeable difference displaying the images.

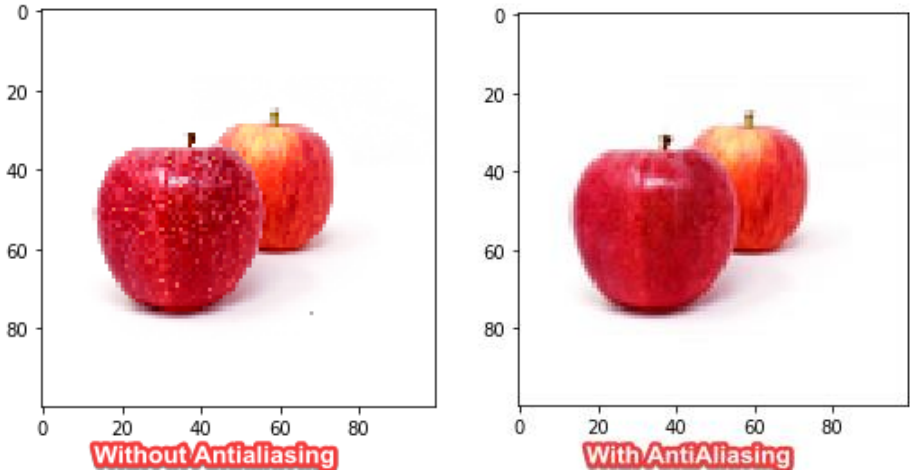
Pillow uses *Image* to load the images and Matplotlib's *imshow* to diplay them. CV2 uses numpy array to display the images. But there is no noticeable difference to the naked eye.



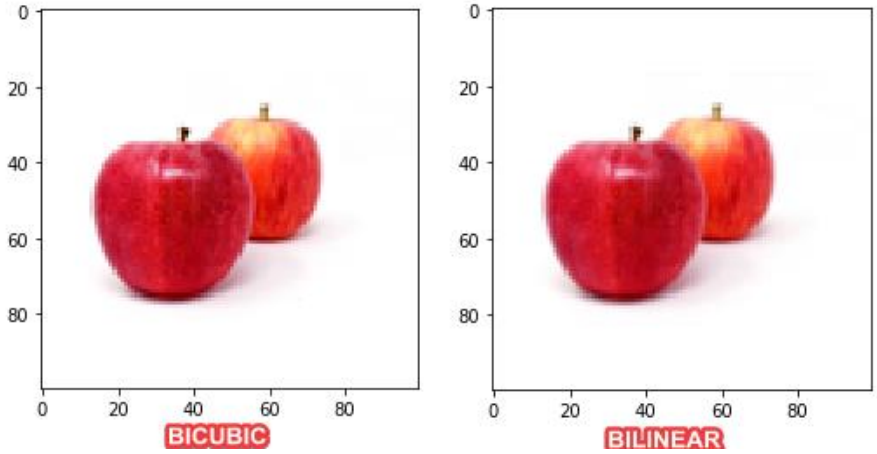
B. **Resizing:** We see what alternations need to be done when changing size to thumbnails.

Going by common notions, whenever we alter the size of image, the quality tends to suffer. To compensate for this Pillow and CV2 use different methods. Pillow uses *AntiAlias* where as CV2 uses *InterArea Interpolation*. Again, to the naked eye, both are indistinguishable.

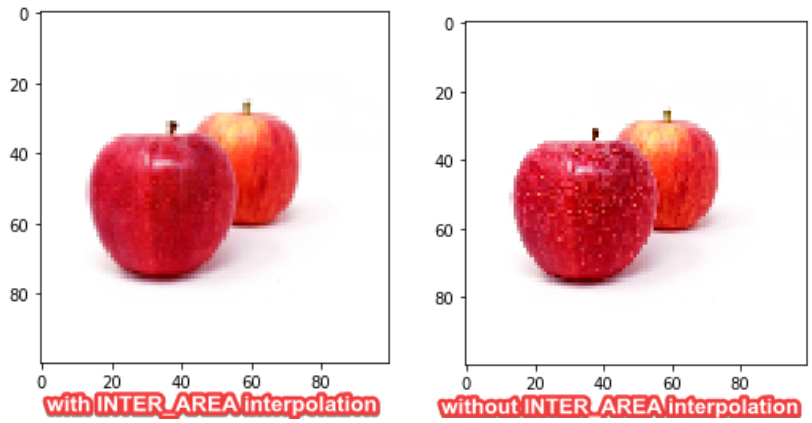
Pillow with ANTI_ALIAS



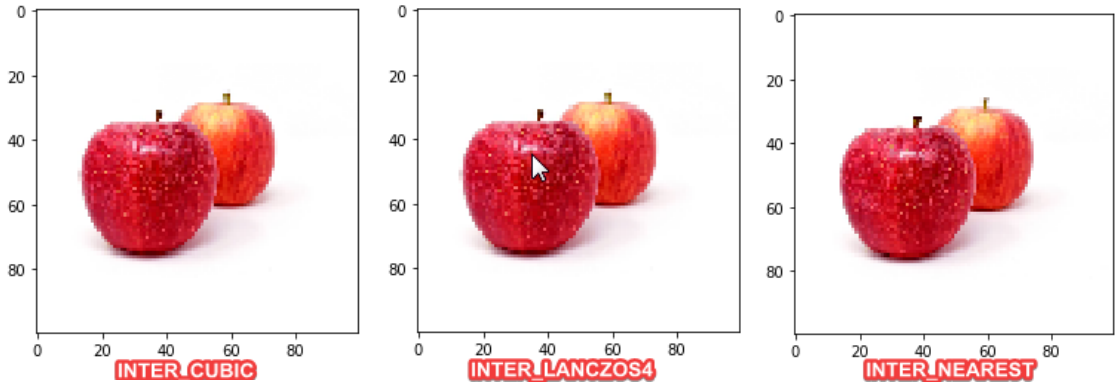
Other options there were explored but dropped in favor our antialiasing being favorite in online community.



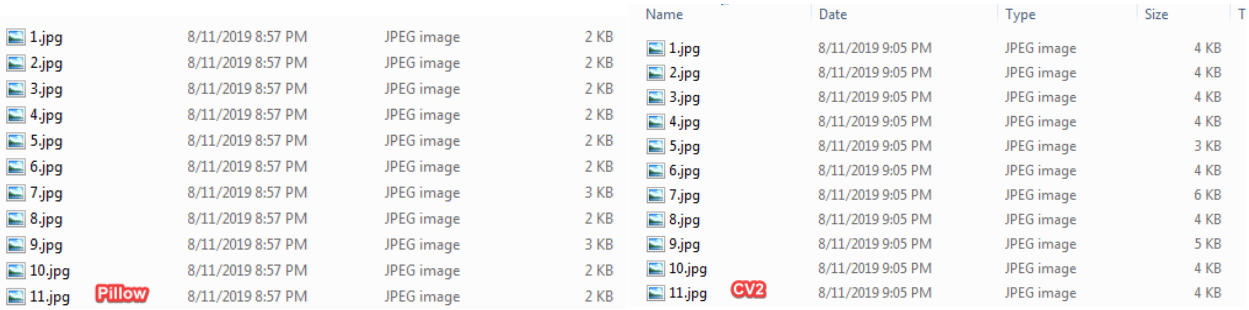
CV2 with INTER_AREA interpolation



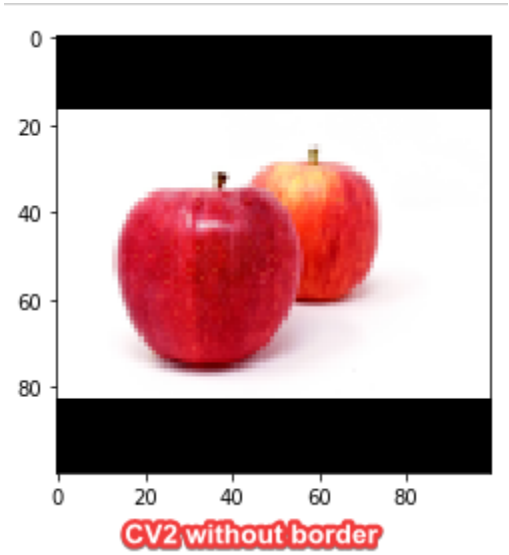
Other options that were explored but dropped in favor of INTER_AREA



C. **Image weight:** The size of the images when saved was smaller in Pillow than that of CV2. Below is the comparison of grayscale sizes between the two



D. **Padding method**: In Pillow, a blank image with padding color is taken on which the resized image is pasted. This insures that the left out areas automatically take over the padding color. With CV2, the *copyMakeBorder* method is used to perform padding of the desired color.



E. **Performance on grayscale:** Grayscale will be used to train an initial model for final compare between the two image converters to conclude which one is better suited for the project. Grayscale training was chosen simply because there are less channels (one color) and therefore it is easier to train than color images that have 3 channels (RGB). A simple 4-layer CNN was used training pre-processed images from Pillow and CV2 for 10 epochs. Validation loss and validation accuracy were chosen to measure performance.

Pillow:
- 39s 80ms/step - loss: 0.1875 - acc: 0.9125 - val_loss: 0.1700 - val_acc: 0.9333
CV2:
- 36s 76ms/step - loss: 0.1413 - acc: 0.9458 - val_loss: 0.0857 - val_acc: 0.9667

As CV2 gave better performance and was chosen to be the preprocessing tool of choice.

Preprocessing: Using CV2, all raw images are preprocessed in to two folders 'Full_Set_Processed' which has subfolders for each class with a train.csv and also to a 'Full_Set_Proc_Comb', where all images are present without subdirectories. There are intentionally two different processed folders to try out different options of Keras library, which will be explored in Milestone 2. For now, let's train a basic classifier to see the accuracy we can reach.

Step 4: Training basic model

A basic 4 layer CNN was setup to measure the time and performance of the classifier with colored thumbnails on fully processed set with this configuration:

```
model3 = Sequential()
model3.add(Conv2D(32, kernel_size=(3, 3),activation='relu',input_shape=(100,100,3), paddin
g='same'))
model3.add(Conv2D(32, (3, 3), activation='relu'))
model3.add(MaxPooling2D(pool_size=(2, 2)))
model3.add(Dropout(0.25))
model3.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model3.add(Conv2D(64, (3, 3), activation='relu'))
model3.add(MaxPooling2D(pool_size=(2, 2)))
model3.add(Dropout(0.25))
model3.add(Flatten())
model3.add(Dense(512, activation='relu'))
model3.add(Dropout(0.5))
model3.add(Dense(5, activation='softmax'))
model3.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model3.fit(X_train, y_train, epochs=60, validation_data=(X_test, y_test), verbose=2)
```

Different configurations were tried by adjusting the number of filters, kernel_size, dropout, pool_size etc., but the above configuration was proved to be optimum. In the interest of time, only 100 epochs were run to measure the trend or direction the model was going in. The model ran for 1 hr 7 mins and gave the below result. The training loss dropped to 0.015 and training validation was 99.67%, which is very good, but the validation loss was still very high at 1.0457 and accuracy was only 80%. This meant that whilst the model was very efficient on the training set, it wasn't so good with the test set. It points to a performance mismatch problem, where the model is **overfitting** on the training set. The test set

could have images that are under-represented in the training and/or the optimizer used wouldn't be the best and different one needs to be chosen. But increasing the training set size isn't an option as more images are hard to get. So, we'll have a look at resolving this issue and taking the model to the next level in Milestone 2 report.

```
model3.fit(X_train, y_train, epochs=60, validation_data=(X_test, y_test), verbose=2)
```

```
 - 67s - loss: 0.0129 - acc: 0.9950 - val_loss: 1.0358 - val_acc: 0.8000
Epoch 53/60
 - 67s - loss: 0.0104 - acc: 0.9958 - val_loss: 1.0623 - val_acc: 0.8233
Epoch 54/60
 - 67s - loss: 0.0097 - acc: 0.9958 - val_loss: 1.2012 - val_acc: 0.8100
Epoch 55/60
 - 67s - loss: 0.0096 - acc: 0.9958 - val_loss: 1.0871 - val_acc: 0.8267
Epoch 56/60
 - 67s - loss: 0.0254 - acc: 0.9967 - val_loss: 1.5769 - val_acc: 0.7700
Epoch 57/60
 - 67s - loss: 0.0594 - acc: 0.9792 - val_loss: 1.1930 - val_acc: 0.8233
Epoch 58/60
 - 67s - loss: 0.0282 - acc: 0.9917 - val_loss: 1.0721 - val_acc: 0.8000
Epoch 59/60
 - 67s - loss: 0.0206 - acc: 0.9942 - val_loss: 1.0258 - val_acc: 0.8133
Epoch 60/60
 - 67s - loss: 0.0150 - acc: 0.9967 - val_loss: 1.0457 - val_acc: 0.8000
Wall time: 1h 7min 7s
```

```
Out[360]: <keras.callbacks.History at 0x48318d08>
```