CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

The computing language that is used in this research is Python and with the help of google colab to run the program, the author choose google drive to store the zipped dataset and input images. All the dataset and input images consist of raw data therefore data preprocessing is required so it can be used for the mass estimation model.

5.1.1. Remove Input Image Background to Transparent

```
img = cv.imread('/tmp/inputtopview.jpg', cv.IMREAD_UNCHANGED)
imagecopy = img.copy()

kernele = np.ones((8, 8), 'uint8')
kerneld = np.ones((4, 4), 'uint8')

filter = cv.cvtColor(img, cv.COLOR_BGR2HSV)
filter = cv.cvtColor(filter, cv.COLOR_HSV2RGB)
filter = cv.cvtColor(filter, cv.COLOR_RGB2GRAY)
filter = cv.GaussianBlur(filter, (17, 17), 17)
filter = cv.erode(filter, kernele, iterations=3)
filter = cv.Canny(filter, 23, 23)
filter = cv.dilate(filter, kerneld, iterations=3)

_, thresh = cv.threshold(filter, 0, 255, cv.THRESH_BINARY  + cv.THRESH_OTSU)
kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE, (5, 5))
mask = cv.morphologyEx(thresh, cv.MORPH_CLOSE, kernel, iterations=4)

data = mask.tolist()
sys.setrecursionlimit(10**8)
for i in  range(len(data)):
    for j in  range(len(data[i])):
        if data[i][j] !=  255:
            data[i][j] =  -1
        else:
            break
    for j in  range(len(data[i])-1, -1, -1):
        if data[i][j] !=  255:
            data[i][j] =  -1
        else:
            break
image = np.array(data)
image[image !=  -1] =  255
image[image ==  -1] =  0imag

mask = np.array(image, np.uint8)
main = cv.bitwise_and(imagecopy, imagecopy, mask=mask)
main[mask ==  0] =  255
cv.imwrite('/tmp/topviewextracted.jpg', main)
img = Image.open('/tmp/topviewextracted.jpg')
img.convert("RGBA")
datas = img.getdata()

```

```
44  datas2 = []
45  for item in datas:
46      if item[0]==255 and item[1]==255 and item[2]==255:
47          datas2.append((255, 255, 255, 0))
48      else:
49          datas2.append(item)
50
51  img.putdata(datas2)
52  img.save("/tmp/topviewextractedtr.png", "PNG")
53  plt.imshow(img)
```

The first line is to read the input image we are going to use for volume estimation. Same code works for input image top view and side view. As to discuss the code line, on the above code the author uses the top view image only. Second line is to copy the image so that it can be used later on line 37 to mask the image. The fourth and fifth is to initialize the kernel that is going to be used for erosion in line 11 and dilation in line 13.

Lines 7 to 13 is to convert color and filter the image to have a clean edge cut. As cv2 reads the image in BGR so the image needs to be converted to HSV to have a big color difference between the object and the background like in line 7. Next is to turn the image to RGB like in line 8 and to convert it again to grayscale. With grayscale, it increases the color difference and is easier for canny detection. There's a slight difference by using HSV or grayscale on but the edge can be cut more precisely with HSV. The next step is to add Gaussian Blur like in line 10 reasons behind this are to eliminate unwanted noise whether on the object or the background since the author calculates the area by object's image for the more accurate result is not counting the noise and eliminating them.

The next line is to use erosion with kernel (8,8) that have been mentioned before with 3 iterations by erode it helps the resulting image to not have a black border around the object as the result of Gaussian blur. Line 12 is canny edge detection to detect the object edge accurately. On line 13 is dilation with the kernel (4,4) and 3 iterations, dilation helps the object to have a better edge cut unlike without dilation where the edge cut is too small and untidy. After finishing to identify the precise edge, the result needs to be thresholded. The author uses Otsu's Thresholding like in line 15 and the next line, 16 is to determine the kernel. Here the structuring element is used to provide an elliptical/circular kernel instead of numpy which provides a rectangular shaped kernel since most of our object is centered and round shaped.

Next is to use closing from cv2 morphologyEx in line 17 which consists of dilation followed by erosion to remove noises and create a better edge result for the mask with 4 iterations. Next line which is 19 is to store the mask in list form and for the next line 20 set recursion limit is needed to have deeper recursion limit in order to avoid any crashes. Line 21 is for looping each line and line 22 with 27 is to loop every pixel color. The difference between those two lines is line 22 is started from the front pixel of the list while 27 is started from behind, the last pixel on the row. Inside of the second loop there's a decision part where if the loop meets 255 in the row member then the loop will break like in line 26 and 31 if not then it will be replaced by -1 like in line 24 and 29. So if in the row there’s 255 the loop will break the same with the second row looping which is from behind so only the center pixel is left considering the result before is an edge detection, in other words only the lined edge of the object (255) in the picture. To have a white color (255) blocked inside the border to represent the object. On line 32 the result is saved into a new variable so that on line 33 if the image pixel is not -1 then it will be replaced by 255 which inside of the border all consist of 0 pixel and 255 represent the object. Line 34 is to replace all the -1 to 0 so only the background will have black color.

Line 36 is to insert the result before to mask variables in list form. Next in line 37 is to use bitwise AND from OpenCV to select the ‘interesting part’ which is object image only of the picture by using the mask we have made earlier. Next line 38 is to select all black colour which is the background to white colour. The result is saved to the tmp folder like in line 40 and for line 41 is to open the image and insert to img variable. Next line 41 is to convert the image to RGBA and Line 42 is to insert again img pixel in datas variable. Here on line 44 initializing the datas2 variable saves the list of new images later in list form. On line 45 until 49 is to change the picture background from white to transparent. Line 45 is looping to access all list that has been saved in datas variable. Inside of the loop is the decision which if the pixel has white colour 255,255,255 then it will be replaced with transparent which is 255,255,255,0 and 0 stands for alpha like in line 46 to 47. If not white colour then the pixel colour is stayed that way like in line 48 to 49. Next in line 51 is to have it in image form by using putdata() function and line 52 to save the image final result to the tmp folder. Line 53 is to show the final image result by using matplotlib. Same steps of the code works for the side view image too so here as the result the author has two transparent background images which are top view of the object and side view of the object.

5.1.2. Extract The Zipped Dataset

```
54  Files = namedtuple('File', 'name path')
55  dataset = []
56  p = Path('/tmp/Training/')
```

```
57  for item in p.glob('**/*'):
58    name = item.name
59    path = Path.resolve(item).parent
60    dataset.append(Files(name, path))
61
62  dataset
```

Line 54 is to declare a variable to store the extracted result in tuple form with named fields. The result later will be stored in the Files variable. Next is to declare a variable dataset in line 55 to store each of the tuples in list form. Line 56 is to store the selected path where the dataset is going to be extracted which is '/tmp/Training/'. As for accessing zipped folders and subfolders, looping and glob is utilized in line 57. Inside of the looping, line 58 is to store each of the image names in the 'name' variable next line 59 is to store the image path in the 'path' variable. Last in line 60, after the name and path of the images are obtained, both of them are stored in a dataset variable which has been declared in line 55 consisting of file tuples which are image name and path. Line 62 is to print the dataset variable to show the extracted result.

5.1.3. Image Hash

```
63  def hashes_calculation(files, hashfunc=imagehash.whash):
64      hashes, names = [], []
65      for i, name in enumerate(files):
66          try:
67              img = Image.open(name)
68              hash = hashfunc(img)
69              hashes.append(hash)
70              names.append(name)
71          except:
72              pass
73
74      return hashes, names
```

As to calculate the images hash value the author built a function which called hashes_calculation. Line 63 is to initialize the function with two params which are files to get the image path and hashfunc to store the built in function from imagehash which is whash (wavelet hashing). Inside of the function first is to initialize variables that are going to be used to store the result which are hashed and names like in line 64. Looping is required to access each of the images and enumerate them in line 65. Line 66 and 71 is for try and except where try is to test a block of code while except for handles the error. Inside of the try, line 67 is to open the image with it's particular path and stored in img variable. Next is to hash the image with the help of hashfunc which consists of imagehash from line 63 and stored it in a hash variable. Followed by line 69 keeping the hash result in hashes variable and image’s name in names variable in line 70. Line 72 is pass to avoid any error and the final step is to return the result which are hashes and names in line 74.

5.1.4. Distance Matrix

```
75  def distances_calculation(hashes):
76      matrix = np.zeros((len(hashes), len(hashes)))
77      for i, j in combinations(range(len(hashes)), 2):
78          dist = hashes[i] - hashes[j]
79          matrix[i, j] = matrix[j,i] = dist
80      return matrix
```

After finishing with hash, now is to calculate the distance between each of the images which the result will be in matrix form. First is to initialize the function like in line 75 which is distances_calculation and one parameter which is hashes where the list of hash values that have been calculated is stored. In line 76, 2 dimensional array both with the length of hashes list is initialized and stored in matrix variable. Followed by looping in line 77 to access every two images with combinations to help obtain every possible image couple. Inside of the loop is to calculate the distance of each image by subtracting hash value of both the images and storing it in dist variable like in line 78. Afterwards, line 79 is to insert the result in the matrix that has been made earlier. Last is to return the result of this function which is the matrix consisting of all distances between two images where can be seen in line 80.

5.1.5. Coordinates Form

```
81  pca = PCA(n_components=2)
82  dist2d = pca.fit_transform(matrix)
```

Now we have each image's distance in matrix form, to enter DBSCAN coordinates dataset is required. Besides it is easier for DBSCAN to process, it’s easier to visualize either. To turn the distance matrix to coordinates a help from PCA is required. In line 81 is to set up the PCA with total components of 2 because of 2 dimensional vectors which are x and y and stored in the pca variable. Next, line 82 is to call the PCA built in function which is fit_transform to transform the distance matrix to coordinates and save in dist2d variable.

5.1.6. DBSCAN Algorithm

Built DBSCAN Object

```
83  class DBSCAN():
84      def __init__(self):
85          self.core = -1
```

```
86        self.border = -2
```

Now the dataset has been transformed into coordinates. This section will discuss how to build the DBSCAN model. The first step is to create a DBSCAN object like in line 83 by initializing the DBSCAN class. Next followed by line 84 which is initializing a DBSCAN constructor. In line 85 is for initializing core and line 86 border points both with arbitrary value.

Find Neighbour Function

```
87      def find_neighbour(self, data, point_id, eps):
88          points = []
89          for i in range(len(data)):
90              # Euclidian distance
91              if np.linalg.norm([a_i - b_i for a_i, b_i in zip(data[i], data[point_id])]) <= eps:
92                  points.append(i)
93          return points
```

After initializing the constructor, it's time to build a function here in line 87 the author built a find_neighbour function to find neighbours of each data point with 4 params which are self for represent the instance of the DBSCAN class, data for each of the dataset coordinates and point_id to represent each of the dataset id, and eps where to determine how close the distance of a point to be considered as neighbour. In this function Euclidean Distance is featured in order to find distances between two points. Inside of the function, first is to initialize the points variable where the result is going to be stored in list form that can be seen in line 88. In line 89, to access every data point, iteration through every coordinate is required. Inside of the loop in line 91 is the implementation of Euclidean Distance formula which if the result is less than equal to eps it's considered as it's point neighbour. The point will be stored in points variable like in line 92 and Last line 93 is the function return which is a points variable that consists of the final result.

Fit Function

```
94      def fit(self, data, Eps, MinPts):
95          point_label = [0] * len(data)
96          point_count = []
97
98          core = []
99          border = []
100
101         for i in range(len(data)):
102             point_count.append(self.find_neighbour(data, i, Eps))
103
104         for i in range(len(point_count)):
105             if (len(point_count[i]) >= MinPts):
106                 point_label[i] = self.core
107                 core.append(i)
108             else:
```

```
109                 border.append(i)
110
111         for i in border:
112             for j in point_count[i]:
113                 if j in core:
114                     point_label[i] = self.border
115                     break
116
117         cluster = 1
118
119         for i in range(len(point_label)):
120             q = queue.Queue()
121             if (point_label[i] == self.core):
122                 point_label[i] = cluster
123                 for x in point_count[i]:
124                     if(point_label[x] == self.core):
125                         q.put(x)
126                         point_label[x] = cluster
127                     elif(point_label[x] == self.border):
128                         point_label[x] = cluster
129                 while not q.empty():
130                     neighbors = point_count[q.get()]
131                     for y in neighbors:
132                         if (point_label[y] == self.core):
133                             point_label[y] = cluster
134                             q.put(y)
135                         if (point_label[y] == self.border):
136                             point_label[y] = cluster
137                 cluster += 1
138
139         return point_label, cluster
```

Calculating distance function has been done now it's turn to build a clustering function which is fit function. First is to initialize a Fit function like in line 94 which consists of 4 params which are self, data, eps, and minPts. Self is to represent the instance of the DBSCAN class, data for the coordinates dataset, eps will be determined how close the distance to be considered as a neighbour, and minPts is for minimum data points to be considered as a cluster. Next line 95 is initializing the point_label variable of the data points cluster labels which are in list form and [0] here stands for cluster 0 or outliers in other words all of the data points considered as outliers first. In line 96 point_count variable is initialized to store the find neighbours function result earlier. After, line 98 is to initialize the core variable to store all the core points in list form and line 99 is to initialize border points to store all the border points in list form either.

All the required variables have been initialized. Next line 101 is to iterate through all the dataset and line 102 inside of the loop is the function we have made earlier find_neighbour to find all the neighbours of particular data points and the result is stored in the point_count variable.

Each point has its own neighbours but it needs to be defined which points are core points, border points, and outliers. It starts with iteration to access every data point like in line 104. Inside of the loop is decision therefore in line 105 if the point has more than equal to minPts which minimum points than it's point is considered as core point like in line 106 and line 107 is to store core point in core variable. Then if the point doesn't fulfill the requirements then the point goes into the else in line 108 and in line 109 the point is stored in the border variable. Followed by another iteration in line 111 for every point in the border list which is to confirm whether the point is border point or core point. Line 112 is another iteration to access it's particular point in point_count variable. Inside or two iteration in line 113 is a decision if the particular point in the core list then it's point_label is replaced with a border like in line 114 and line 115 is to break the loop.

After determining which point is the core point and which point is the border point now it's turn to form the clusters. In line 117 cluster variable is initialized to count the number of clusters formed. The count starts from one. Followed by iteration in line 119 to access every point label. In line 120 is to initialize q which consists of queue built in function Queue() which is a constructor for a FIFO queue. Next is line 121, it's a decision if the particular point is in the core list then the point is in the new cluster like in line 122 by replacing the particular point label. In line 123 there is iteration for every point group in point_count. Followed by decision in line 124 if a particular point label is considered as a core point then in line 125 there is put queue function to put an item into the queue until a free slot is available before the item is added. Before, all point labels are 0 here in line 126 the point label of a particular point is replaced with the cluster number. Next is line 127 for the else if decision, if a particular point is in the border point list then in line 128 a particular point label is replaced with cluster number.

Next line 129 is a while loop as long as the queue is not empty then in line 130 there's get queue function to wait until an item in queue is available. Point count of a particular point is stored in the neighbors variable. In this loop is to check every point of the particular point neighbour. In line 131, for every point in neighbours variable, if the point is a core point like in line 132 then the point label of it's point is replaced with the cluster number as in line 133. Next line 134 is to be put in queue again. Followed by line 135 if the particular point is border point then the point label of the particular point is replaced with cluster number either as in line 136. Line 137 is to initialize the cluster number by adding plus one for every iteration. Last line in this function is line 139 to return the point label and cluster this function has formed. With this function Breadth First Search is formed by every node in n level then n+1 level and so on.

Visualization Function

```
140  def visualize(self, data, cluster, clusters_count):
141      N = len(data)
142
143      colors = np.array(list(islice(cycle(['#FE4A49', '#2AB7CA']), 3)))
144
145      for i in range(clusters_count):
146          if (i == 0):
147              color = '#000000'
148          else:
149              color = colors[i % len(colors)]
150
151          x, y = [], []
152          for j in range(N):
153              if cluster[j] == i:
154                  x.append(data[j, 0])
155                  y.append(data[j, 1])
156          plt.scatter(x, y, c=color, alpha=1, marker='.')
157      plt.show()
```

To ease reading the clustering result, visualization is needed. Here the visualization function will be discussed. First is to initialize the visualization function like in line 140 which consists of 4 params. Self, to represent the instance of the DBSCAN class, data is for the dataset, cluster for the point_labels by fit function, and clusters_count is to get the cluster number that’s been formed by fit function. Next in line 141 there is N variable to store the length of the dataset. By line 143 is to store various colors that are going to be used to represent the clusters by adding them into an array and stored in *colors* variable. Next in line 145 an iteration is used to access every cluster. Inside of the loop is a decision part in line 146 if the cluster is 0 then in line 147 is to represent the cluster color which is black and where cluster 0 is an outlier. Else in line 148, if the cluster is not 0 then in line 149 the cluster color is represented based on colors that have been initialized earlier in line 143.

In line 151, x and y are initialized in order to store the dataset coordinates later. After, line 152 is an iteration which is needed to access every dataset stored in N. Inside of the loop there’s a decision that can be seen in line 153 for a particular cluster then x coordinates of the data points is stored in x variable like in line 154 and y coordinates of the data points is stored in y variable like in line 155. Line 156 is to form the scatter plot of the clustering result by matplotlib. Last, line 157 is to show the visualization result.

DBSCAN Implementation

```
158 df = pd.read_csv("/content/ProjectDataset/MyDrive/coordinates.csv")
159 dataset = df.astype(float).values.tolist()
160
161 X = StandardScaler().fit_transform(dataset)
162 DBSCAN = DBSCAN()
163 point_labels, clusters = DBSCAN.fit(X, 0.1, 4)
164 print(point_labels, clusters)
165 DBSCAN.visualize(X, point_labels, clusters)
```

In this section is where all of the DBSCAN functions we have made are called. First line 158 is to read our coordinates dataset in csv form for easier callable and stored in df variable. In line 159, the dataset from df is transformed into list form and stored in the dataset variable. In line 161 the dataset needs to be normalized in order to have new values but still maintain general distribution and ratios among the data source while still keeping values within the scale of all the numeric columns in the model to avoid problems of great difference values to scale in the modelling. After the dataset has been normalized, now begin the DBSCAN clustering. In line 162 the DBSCAN object is created and stored in the DBSCAN variable. Next line 163 is to call the fit function with 3 parameters to be sent which are the normalized dataset, determined eps and minPts to be saved into point_labels and clusters variable. Line 164 is to print the point_labels and clusters result. Last is line 165 to call the visualization function which consists of 3 parameters X stands for the normalized dataset, point_labels and clusters from the fit function result.

5.1.7. Input Image Cluster

```
166 pntlbl = int(input_img['Point Labels'])
167 filter_data = all_files[all_files['Point Labels'] == pntlbl]
168 filter_data
```

This section is to filter only images that belong to the same cluster as the input image is taken to become k-NN training data, the other cluster is dropped. Point Labels represent the cluster number. First is to find what the input image cluster number like that have been done in line 166 and the result is stored in pntlbl variable. Next is to filter the data which is to find all the dataset images that have the same number of point labels as the input image like in line 167, the result is stored in filter_data variable and line 168 is to present them.

5.1.8. k-NN Algorithm

Euclidian Distance Function

```
169 def euclidean_distance(vec1, vec2):
170   distance = 0.0
171   for i in range(len(vec1)-1):
172     distance += (vec1[i] - vec2[i])**2
173   return sqrt(distance)
```

After the data has already been filtered, only images with the same cluster with the input image are left. Here the data becomes k-NN training data and the goal is to find the closest point with the input image. To build the k-NN algorithm started by building the calculation. To calculate the distance between two vectors the author uses Euclidean Distance. Started by initializing the euclidean distance function which has 2 params vec1 and vec2 where vec1 stands for the input image coordinate that becomes the reference and vec2 stands for all the training dataset coordinates that can be seen in line 169. Next is to initialize the distance variable like in line 170 that is going to be used. In line 171 an iteration is used in order to iterate every item in the test dataset but the test image that is used in this model is 1 so only 1 iteration. Inside of the loop there is the Euclidean Distance formula in line 172 which is to subtract vector 1 (test image) with vector 2 (training image) and square them then the result is stored in distance variable. This function returns the square root of the distance variable which consists of the square result like in line 163.

Get Neighbours Function

```
174 def get_neighbors(train, test_row, num_neighbors):
175   distances = list()
176   for train_row in train:
177     dist = euclidean_distance(test_row, train_row)
178     distances.append((train_row, dist))
179   distances.sort(key=lambda tup: tup[1])
180   neighbors = list()
181   for i in range(num_neighbors):
182     neighbors.append(distances[i][0])
183   return neighbors
```

The distance calculation function has been done now it's turn to make a function to find the closest neighbours of the input image. In line 174, the get_neighbours function is initialized with 3 params, those are train for the dataset, test_row for the reference which is input image, and num_neighbors is for the k value. In line 175 distances variable is declared to save data in list form. Followed by iteration for every item in the train dataset like in lime 176. Inside of the loop can be found a euclidean_distance function. Here in line 177 the euclidean function is called to count distance between reference coordinate and particular train dataset coordinate, the result is stored in dist variable. Next is to store the train coordinates and it's distance in distances variable like in line 178. Here in line 179 is to sort the result stored in distances variable based on the closest one with the input image. Afterwards another variable is initialized in line 180 which is neighbours to store data in list form. Another iteration is used in line 181 based on the k value that has been determined. Inside of the loop is line 182 where the distance variable is being called. This iteration is used in order to find the closest distance based on k value. In this model the author uses 2 k values so only 2 iterations which is the input image and one point that have the closest distance with the input image. Last line in this function is 183 where this function returns the neighbours variable consisting of the final result.

K-NN Implementation

```
184 n =  []
185 dataset = dist2d2
186 neighbors = list(get_neighbors(dataset, dataset[0], 2))
187 n = n + neighbors
188 print (n)
```

After finishing building all the required functions, the final step is to call the get_neighbours function that has been built earlier. First in line 184 is to declare n variable to store data in list form. Next, line 185 is to store the normalized dataset in dataset variable. After, in line 186 the get_neighbors function is called and the result is stored in neighbors variable in list form. In line 187 the neighbors variable is being added into n variable that has been made earlier. Line 188 is to display the final result which consists of the reference coordinate which is the input image and a coordinate of the closest point with the input image. Now we have the final result about what the input image object is based on what object has the closest distance with the input image.

5.1.9. Get Object's Density

```
189 final_dense = dens[dens['Fruit'] == fr]
190 density = float(final_dense['Density (g/cm3)'])
```

After the object has been successfully identified, the object's density is required. Start by finding the object's density from the stored density which has the same object's name. Line 189 is to select the row which has the same object's name and store it in the final_dense variable. Line 190 is to select the density of the selected object and store it in the density variable. Now the object's density has been acquired.

5.1.10. Volume Calculation

Black Background Transformation

```
191 image = skimage.io.imread(fname='/tmp/topviewextractedtr.png')
192 blur = skimage.color.rgb2gray(image)
193 mask = blur < 0.98
194 blackbg = np.zeros_like(image)
195 blackbg[mask] = image[mask]
196 skimage.io.imshow(blackbg)
```

In order to estimate the object's volume, area calculation and object's height is required. Since all the input images have a transparent background, both of the input images, top view and side view need to be transformed to have a black background for easier calculation. Start by reading the selected image, for the first section the input image is the object's top view and stored in the image variable. Here in line 191 skimage is used in the reason skimage reads images in RGB. Next is to transform the image to grayscale and store the result in a blur variable like in line 192. After, in line 193 blurring effect is being added and the result stored in the mask variable. In line 193, blackbg is being declared consisting of black color image with the same feature as the image. Next is both of the black background and image is being masked based on the transformed blurred image resulting in combined result with only the background being changed into black. With black background threshold for the next section can be done easier and more precisely. In this section only the top view image is being presented, same steps happen for the side view image either.

Area Calculation

```
197 img = blackbg
198 height = img.shape[0]
199 width = img.shape[1]
200 gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
201 ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY)
202 plt.imshow(thresh)
203 count = cv2.countNonZero(thresh)
204 area = count*0.14*0.14/(width*height)
205 print(area)
```

Object's area calculation is performed by calculating the non zero pixel. First step is by reading the object's top view image that has been transformed to black background like in line 197. Next, in line 198 image height is being calculated and in line 199 image width is being calculated. Afterwards the image is transformed into grayscale like in line 200. After it is finished turning into grayscale and stored into a gray variable, the image needs to be thresholded like in line 201. Turning a black and white image and stored in the thresh variable. Next in line 202 is to show the result of threshold. To calculate the area is to count the non zero pixel considering only the image background is black or zero so the object or non zero pixel is being calculated like in line 203. The calculated result is being stored in the count variable. In line 204, the area is calculated by multiplying count with 0.14 which represents the ground area or dimensions in meters and divided by multiplied images height and width, the result is stored in the area variable. Line 205 is to show the result.

Height Calculation

```
206 img = blackbg2
207 gray2 = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
208 ret, thresh2 = cv2.threshold(gray2,0,255,cv2.THRESH_BINARY)
209 plt.imshow(thresh2)
210
211 image = skimage.io.imread(fname='/tmp/sideviewextractedtr.png')
212 x,y,w,h = cv2.boundingRect(thresh2)
213 cv2.rectangle(image, (x, y), (x + w, y + h), (36,255,12), 2)
214 cv2.putText(image, "w={},h={}".format(w,h), (x,y -
     10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (36,255,12), 2)
215 plt.imshow(image)
216 print(h)
```

Next step is to calculate the object’s height. Same steps as been mentioned before where the side view image needs to be transformed to have a black background either. After the side view image has a black background, it is stored in an img variable like in line 206. Next in line 207 the image is transformed into grayscale, the result stored in gray2 variable and in line 208 the image is thresholded, the result stored in thresh2 variable. Line 209 is to show the threshold result. Same steps with the top view image, the top view image uses the non zero pixels to calculate the object’s area but here the side view image uses non zero pixels to draw a bounding box and estimate the object height.

In line 211 the input image is being read and stored to the image variable. Side view input image is used as a background where the bounding box is drawn. Next line 212 is to create a bounding box according to non zero pixels from the threshold result where x,y is the top left coordinate of the rectangle and w,h is it’s width and height. In line 213 a rectangle bounding box with green color is being drawn. Line 214 is to put text on the top left of the bounding box. Next, line 215 is to show the image with the bounding box result consisting of the object's height and width. Last is line 216 is to print the object’s height. Object’s height has been found.

Objects Volume

```
217 V = round((area*10000))*round((h/100)/2)
```

Object’s density and height has been obtained and by multiplying them, volume can be acquired. In order to have a real size estimation, the object's area needs to be multiplied by 10000 and the object's height needs to be divided by 100 and again by 2 because the object’s real size height is half of the result. In line 217, by multiplying area and height the result is the object's volume and object’s volume is being stored in the V variable.

5.1.11. Object’s Mass Estimation

```
218 M = density * V
219 round(M)
```

Last but not least is to calculate the goal of this model which is the object's mass. Object’s density and object’s volume has been acquired, by multiplying them the object’s mass can be found. In line 218 the object’s mass is being calculated and the result is stored in the M variable. Followed by line 219 where the result is being rounded. Object’s mass has been found.

5.2. Results

5.2.1. DBSCAN Algorithm

Table 5.1: DBSCAN Time and Cluster Test Result 1

No.	Eps, minPts		Training Data 10		Training Data 20	
			Time	Cluster	Time	Cluster
1.	0,1	2	5m 4s	128	22m 35s	116
2.	0,1	3	5m 2s	99	22m 38s	81
3.	0,1	4	5m 3s	90	22m 39s	61
4.	0,1	5	5m 3s	80	22m 37s	50
5.	0,1	6	5m 0s	69	22m 23s	47

Table 5.2: DBSCAN Time and Cluster Test Result 2

No.	Eps, minPts		Training Data 30		Training Data 40	
			Time	Cluster	Time	Cluster
1.	0,1	2	47m 59s	105	1h 39m 5s	94
2.	0,1	3	46m 15s	82	1h 38m 41s	71
3.	0,1	4	46m 6s	70	1h 28m 14s	65
4.	0,1	5	46m 39s	58	1h 26m 40s	58
5.	0,1	6	46m 34s	59	1h 26m 30s	54

5.2.2. Object Identification

10 Kinds in Training Data (4.802 images)

Table 5.3: Object Identification Result of 10 Fruit Kinds

No,	Objects Name	Identified (Eps ; Minpts)				
		0,1 ; 2	0,1 ; 3	0,1 ; 4	0,1 ; 5	0,1 ; 6
1.	Apple Red Delicious	Yes	Yes	Yes	Yes	Yes
2.	Blueberry	Yes	Yes	Yes	Yes	Yes
3.	Cantaloupe 2	Yes	Yes	Yes	Yes	Yes
4.	Cocos	Yes	Yes	Yes	Yes	No
5.	Dragon Fruit	Yes	Yes	Yes	Yes	Yes
6.	Guava	Yes	Yes	Yes	Yes	Yes
7.	Lemon	Yes	Yes	Yes	Yes	Yes
8.	Onion White	Yes	Yes	Yes	Yes	Yes
9.	Orange	Yes	Yes	Yes	Yes	Yes
10.	Tomato	Yes	Yes	Yes	Yes	Yes

20 Kinds in Training Data (10.207 images)

Table 5.4: Object Identification Result of 20 Fruit Kinds

No,	Objects Name	Identified (Eps ; Minpts)				
		0,1 ; 2	0,1 ; 3	0,1 ; 4	0,1 ; 5	0,1 ; 6
1.	Apple Granny Smith	Yes	Yes	Yes	Yes	Yes
2.	Apple Red	No	No	No	No	No
3.	Blueberry	Yes	Yes	Yes	Yes	Yes
4.	Cantaloupe 2	Yes	Yes	Yes	Yes	Yes
5.	Cocos	Yes	Yes	Yes	No	No

6.	Corn	Yes	Yes	Yes	Yes	Yes
7.	Dragon Fruit	Yes	Yes	Yes	Yes	Yes
8.	Lemon	Yes	Yes	Yes	Yes	Yes
9.	Orange	Yes	Yes	Yes	Yes	Yes
10.	Pepper Red	No	No	No	No	No

5.2.3. Object's Area

Apple

Table 5.5: Mass Estimation Result of 6 Apples

No.	Fruits	Distances	Estimated Area (cm)	Estimated Height (cm)	Estimated Mass (g)	Real Mass (g)
1.	Apple 1	15 cm	23	9	199	139
		17 cm	21	8	161	139
		19 cm	17	8	131	139
		21 cm	16	6	92	139
		23 cm	12	6	69	139
		25 cm	11	5	53	139
2.	Apple 2	15 cm	45	11	475	161
		17 cm	36	9	311	161
		19 cm	28	8	215	161
		21 cm	23	7	155	161
		23 cm	18	6	104	161
		25 cm	16	6	92	161
3.	Apple 3	15 cm	41	11	433	160
		17 cm	34	9	294	160
		19 cm	25	8	192	160
		21 cm	21	7	141	160
		23 cm	16	6	92	160
		25 cm	11	6	63	160
4.	Apple 4	15 cm	24	8	184	164
		17 cm	24	9	207	164
		19 cm	17	8	131	164
		21 cm	16	7	108	164
		23 cm	12	6	69	164
		25 cm	12	6	69	164

5.	Apple 5	15 cm	30	6	173	156
		17 cm	21	7	141	156
		19 cm	16	7	108	156
		21 cm	14	6	81	156
		23 cm	13	6	75	156
		25 cm	11	6	63	156
6.	Apple 6	15 cm	26	9	225	159
		17 cm	17	9	147	159
		19 cm	16	8	123	159
		21 cm	14	7	94	159
		23 cm	13	13	162	159
		25 cm	11	6	63	159

Lemon

Table 5.6: Mass Esimation Result of 6 Lemons

No.	Fruits	Distances	Estimated Area (cm)	Estimated Height (cm)	Estimated Mass (g)	Real Mass (g)
1.	Lemon 1	15 cm	27	12	295	107
		17 cm	21	11	211	107
		19 cm	19	9	156	107
		21 cm	14	8	102	107
		23 cm	13	7	83	107
		25 cm	11	6	60	107
2.	Lemon 2	15 cm	30	12	328	113
		17 cm	24	10	219	113
		19 cm	17	9	140	113
		21 cm	13	8	95	113
		23 cm	13	7	83	113
		25 cm	9	7	57	113
3.	Lemon 3	15 cm	35	12	383	107
		17 cm	25	10	228	107
		19 cm	18	8	131	107
		21 cm	13	7	83	107
		23 cm	10	7	64	107
		25 cm	8	6	44	107

4.	Lemon 4	15 cm	23	11	231	110
		17 cm	19	9	156	110
		19 cm	14	8	102	110
		21 cm	12	8	88	110
		23 cm	10	7	64	110
		25 cm	8	6	44	110
5.	Lemon 5	15 cm	25	10	228	108
		17 cm	18	9	148	108
		19 cm	14	7	89	108
		21 cm	12	7	77	108
		23 cm	10	6	55	108
		25 cm	8	6	44	108
6.	Lemon 6	15 cm	25	10	228	108
		17 cm	18	9	148	108
		19 cm	14	7	89	108
		21 cm	12	7	77	108
		23 cm	10	6	55	108
		25 cm	8	6	44	108

Orange

Table 5.7: Mass Estimation Result of 6 Oranges

No.	Fruits	Distances	Estimated Area (cm)	Estimated Height (cm)	Estimated Mass (g)	Real Mass (g)
1.	Orange 1	15 cm	53	10	546	237
		17 cm	37	8	305	237
		19 cm	32	7	231	237
		21 cm	28	6	173	237
		23 cm	22	5	113	237
		25 cm	20	5	103	237
2.	Orange 2	15 cm	68	10	700	250
		17 cm	47	8	387	250
		19 cm	35	7	252	250
		21 cm	30	6	185	250
		23 cm	26	6	161	250
		25 cm	22	5	113	250

3.	Orange 3	15 cm	64	10	659	237
		17 cm	49	8	404	237
		19 cm	39	7	281	237
		21 cm	32	6	198	237
		23 cm	25	6	154	237
		25 cm	23	5	118	237
4.	Orange 4	15 cm	58	9	538	209
		17 cm	39	7	281	209
		19 cm	33	6	204	209
		21 cm	27	5	139	209
		23 cm	20	5	103	209
		25 cm	20	5	103	209
5.	Orange 5	15 cm	38	9	352	205
		17 cm	28	8	231	205
		19 cm	27	7	195	205
		21 cm	26	6	160	205
		23 cm	18	5	93	205
		25 cm	19	5	98	205
6.	Orange 6	15 cm	53	10	546	236
		17 cm	36	8	297	236
		19 cm	31	7	224	236
		21 cm	28	6	173	236
		23 cm	23	5	118	236
		25 cm	19	5	98	236

Average Accuracy Result

Table 5.8: Average Accuracy of Apples

No.	Object's Name	Accuracy	No.	Objects's Name	Accuracy
1.	Apple 1	94,2%	4.	Apple 4	79,9%
2.	Apple 2	96,3%	5.	Apple 5	90,4%
3.	Apple 3	88,1%	6.	Apple 6	92,4%
				Average	90,2%

Table 5.9: Average Accuracy of Lemons

No.	Object's Name	Accuracy	No.	Objects's Name	Accuracy
1.	Lemon 1	95,3%	4.	Lemon 4	92,7%
2.	Lemon 2	84%	5.	Lemon 5	82,4%
3.	Lemon 3	77,6%	6.	Lemon 6	82,4%
				Average	85,7%

Table 5.10: Average Accuracy of Oranges

No.	Object's Name	Accuracy	No.	Objects's Name	Accuracy
1.	Orange 1	97,5%	4.	Orange 4	97,6%
2.	Orange 2	99,2%	5.	Orange 5	95,1%
3.	Orange 3	83,5%	6.	Orange 6	94,9%
				Average	94,6%

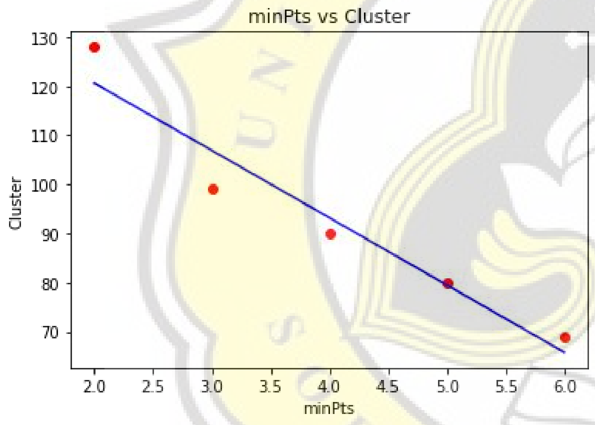

Figure 5.1: Cluster and minPts Regression Linear 1

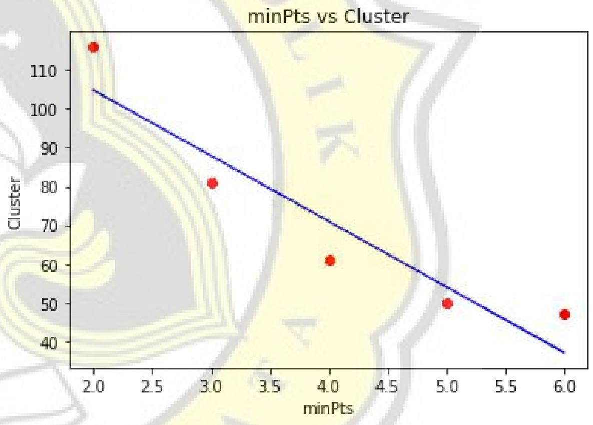

Figure 5.2: Cluster and minPts Regression Linear 2

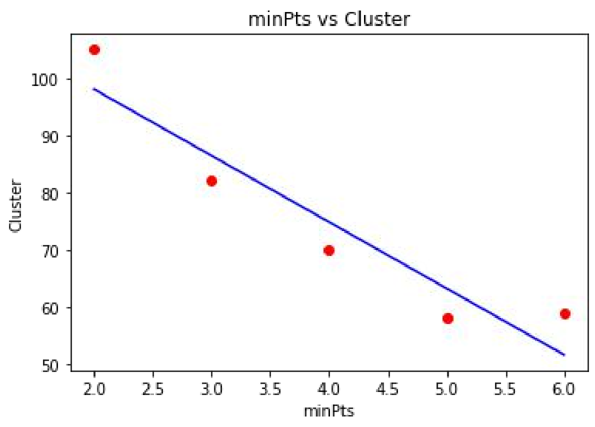

Figure 5.3: Cluster and minPts Regression Linear 3

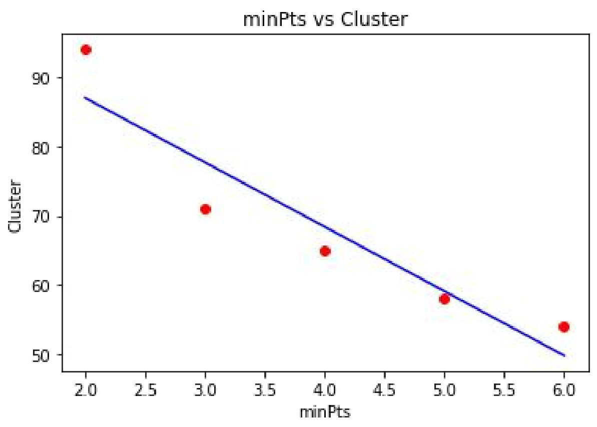

Figure 5.4: Cluster and minPts Regression Linear 4

The first test is the author uses 10, 20, 30, and 40 training data to measure time and cluster that has been formed by DBSCAN. eps stands for neighbourhood radius and minPts stands for minimum point to form a cluster. Here the author uses 0,1 for the eps to have a detailed result as every close point is counted. As for the result from Table 5.1 and 5.2, The higher the value of the minPts the less time it is required and less cluster is formed. Can be seen that minPts has a nonlinear relationship with time and cluster amount.

Next is to test the success of this algorithm by identifying objects. First test is held for 10 kinds of fruit each which consist of total of 4.802 images. Can be seen in Table 5.3 where all the object has successfully identified out of all the tests except Cocos with minPts 0,6 that misidentified as apple red delicious.

Another test is held with 20 kinds of fruit that consist of 10.207 images. And the result is not as good as the previous one, because although more data is loaded, each fruit has the same portion of train data. Out of 20 kinds of fruits there are 2 kinds that are not identified which are apple red and pepper red, besides the same as previous one this model fails in identifying cocos in high value of minPts which are 5 and 6.

Next is the mass estimation is tested within 3 kinds of fruit which are apples, lemons, and oranges each with 6 amounts of fruits. Out of 6 apples being tested, the closest one is the 2nd apple with 6 grams different and 96,3% accuracy. Next is the lemons, out of 6 lemons being tested the closest one with their real mass is the first lemon which has 5 grams difference and 95,3% accuracy with it's real mass. Next is the orange, here the 2nd orange being tested and the result is 2 grams different and 99,2% accuracy with the orange real mass.

Out of 18 fruits with distances of 15, 17, 19, 21, 23, and 25, most of the results achieve the highest accuracy in distance of 19cm between the camera and the object. Among all the highest results obtained, the author calculated the average accuracy with 90,2% for the apples, 85,7% for the lemons, and 94,6% for the oranges.

Figure 5.1 to 5.4 represent the relationship between minPts and the amount of cluster formed. Here minPts and clusters from 10, 20, 30, and 40 training data in table 5.1 and 5.2 are represented with Regression Linear graph and all the results are proved that minPts and the amount of clusters formed have a nonlinear relationship which the higher value of minPts the smaller amount of cluster formed.