Project Report

On

Iron Ore Pellet Size Prediction using Machine Learning

Submitted by

Rahul Das

B.E. Electronics and Instrumentation Engineering (2021-25)

BITS ID- 2021A8PS2060H

Birla Institute of Technology and Science, Pilani

Hyderabad Campus

Secunderabad, Telangana-500078

(30th May 2023 to 25th July 2023)

Under the guidance of

Dr. Santosh Kumar Behera, Principal Scientist

Process Engineering & Instrumentation Department

Council of Scientific & Industrial Research - Institute of Minerals and Materials Technology, Bhubaneswar

CSIR-Institute of Minerals & Materials Technology Bhubaneswar-751013, Odisha, India

सीएसआइआर - खनिज एवं पदार्थ प्रौद्योगिकी संस्थान
(वैज्ञानिक तथा औद्योगिक अनुसंधान परिषद)
भुवनेश्वर 751013, ओडिशा, भारत
CSIR-INSTITUTE OF MINERALS & MATERIALS TECHNOLOGY
(Council of Scientific & Industrial Research)
Bhubaneswar-751013, Odisha, INDIA

This is to certify that the work incorporated in the report entitled **Iron Ore Pellet Size Prediction using Machine Learning** is a record of research work carried out by the **Rahul Das** under my guidance and supervision of **Dr. Santosh Kumar Behera, Principal Scientist** from **30/05/2023** to **25/07/2023** in **Process Engineering & Instrumentation Department, CSIR-IMMT, Bhubaneswar**.

Signature of the Student
Date: ………………..

Rahul Das
2nd Year
B.E. Electronics and Instrumentation (2021-25)
Birla Institute of Technology and Science, Pilani
Hyderabad Campus
Secunderabad, Telangana, 500078

Signature of the Guide
Date: …………………

Dr. Santosh Kumar Behera
Principal Scientist
Process Engineering &
Instrumentation
Department

ACKNOWLEDGEMENT

I am fortunate to have the opportunity to avail the resources, facilities and most importantly the scientific and innovative atmosphere provided by the Council of Scientific & Industrial Research - Institute of Minerals and Materials Technology, Bhubaneswar in the process of working for our project.

It is hereby to extend my heartfelt gratitude to all who have been involved in the successful completion of our project.

I thank **Dr. Ramanuj Narayan**, **Director**, CSIR-IMMT, Bhubaneswar and **Dr. Debi Prasad Das**, **Head PE&I Dept**, Senior Principal Scientist. for giving us the opportunity to pursue our project ambitions in this prestigious institute under Govt. of India.

I am immensely grateful to our guide **Dr. Santosh Kumar Behera**, Principal Scientist for diligently guiding us throughout the project duration. The project would not have been successful without his invaluable perseverance, encouragement and inspiration.

I extend my heartfelt thanks to my Faculty In-Charge (FIC), **Dr. Subhasis Pradhan**, Assistant Professor, BITS Pilani, Pilani Campus. His guidance throughout my PS-1 has been invaluable in ensuring the smooth conduct of my research internship and in evaluating and providing feedback on my progress. I am truly grateful for his continuous support and guidance throughout this journey.

I am immensely grateful to the **Practice School Division** of Birla Institute of Technology and Science, Pilani for providing me with the opportunity to participate in my PS-1 at a prestigious institute and research and development laboratory like CSIR-IMMT.

Rahul Das
2nd Year
BITS Pilani, Hyderabad Campus

DECLARATION

This is to certify that the work incorporated in the report entitled **Iron Ore Pellet Size Prediction using Machine Learning** is a record of research work carried out by me under the guidance and supervision of ***Dr. Santosh Kumar Behera, Principal Scientist*** from ***30th May 2023*** *to* ***25th July 2023*** in ***Process Engineering and Instrumentation Department, CSIR-IMMT, Bhubaneswar*** and the same cannot be either published or applied for an IP (Intellectual Property) without the consent of him.

Signature of the Student
Date: ………………..

Rahul Das
2nd Year
B.E. Electronics and Instrumentation (2021-25)
Birla Institute of Technology and Science, Pilani
Hyderabad Campus
Secunderabad, Telangana, 500078

CONTENTS

INTRODUCTION

In the steel manufacturing sector, pelletizing is a typical procedure. The iron ore fines are fed through a spinning disc and sprayed with water. The iron ore fines are combined with water, bentonite, limestone, coke breeze and various organic binders etc. Fine particles are gathered as the disc rotates continuously, shaped into larger pellets, and then released off the disc. After being conveyed by a conveyer belt into a furnace, the on-size green pellets (diameter: 9–16 mm) are subsequently calcined to create hardened black pellets.

A crucial factor is the green pellets' size dispersion. It needs to be closely watched during the pelletizing process quality control, as excessively coarse (above 16mm) or fine particles (below 9mm) considerably reduce the product's quality and the furnace's thermal efficiency.

In order to obtain the proper size distribution, the production process of iron green pellets should be closely monitored. Manual sieve size operation is the currently used traditional method of determining the variation of green pellets.

This method is intrusive, time-consuming, and unable to function instantly. Due to these restrictions, it is practically impossible to measure the size variation of iron green pellets in real-time using the manual methods now in use, which causes the associated control of the pelletizing process to lag.

Challenges faced in Image Processing:

Imaging system design: The disc pelletizer runs at a high rate of speed in a dusty environment. Iron green pellets are difficult to photograph in a stable, high-quality manner due to the fast-spinning disc pelletizer (10-60 rpm), and the photographs that are captured have a low contrast due to the heavy dusts, background and pellets being indistinguishable/unsegmentable incases where pelletizer disc has been rotating for long hours, background being filled with Iron ore dust. The interference of ambient light can also cause inhomogeneous image brightness, which complicates image processing.

Overlapped pellets: The image-based method for measuring PSD often entails segmenting images of green pellets, figuring out how big each one is, and figuring out how big they are all together. However, due to the green pellets' wetness, relative stickiness, and irregular shape, they can occasionally overlap and form overlapping pellets.

Control of Quality: The final product's quality is greatly influenced by the size distribution of the pellets. The strength, porosity, and reducibility of the pellets, as well as their performance and other characteristics, can all be affected by variations in size distribution. Pellet industries can proactively monitor and control the quality of their products to make sure that they satisfy the specified criteria and standards by anticipating the size distribution over time.

Process optimization: The efficiency and effectiveness of the pelletizing process are directly impacted by size distribution. To ensure the best pellet formation and subsequent calcination, operating parameters like disc speed, feed composition, or moisture content may need to be changed depending on the size distribution. Pellet industries may predict future size distribution patterns and proactively change their processes thanks to time series forecasting.

Resource Planning: For the pellet industry, precise size distribution estimates offer useful information for resource planning. Companies can allocate resources more effectively by altering the raw material mix, managing moisture levels, or making the best use of binders by being aware of the projected size distribution trends. This preemptive planning guarantees resource availability and effective use, reducing manufacturing costs and waste.

Forecasts of the size distribution can assist the pellet industry in matching output to market demand, which is important for inventory management and market demand. Companies can foresee changes in consumer requirements and modify their pellet production in response by looking back at historical trends and patterns. By doing this, they can meet client expectations on schedule and prevent instances where there is an accumulation of excess inventory or a shortage.

Maintenance and equipment optimization: Size distribution time series forecasting can help with preventative maintenance and equipment planning. Pellet companies can promptly schedule maintenance tasks like disc refurbishment or the replacement of worn-out components by seeing potential deviations or trends in the size distribution. This reduces equipment downtime and guarantees efficient operations.

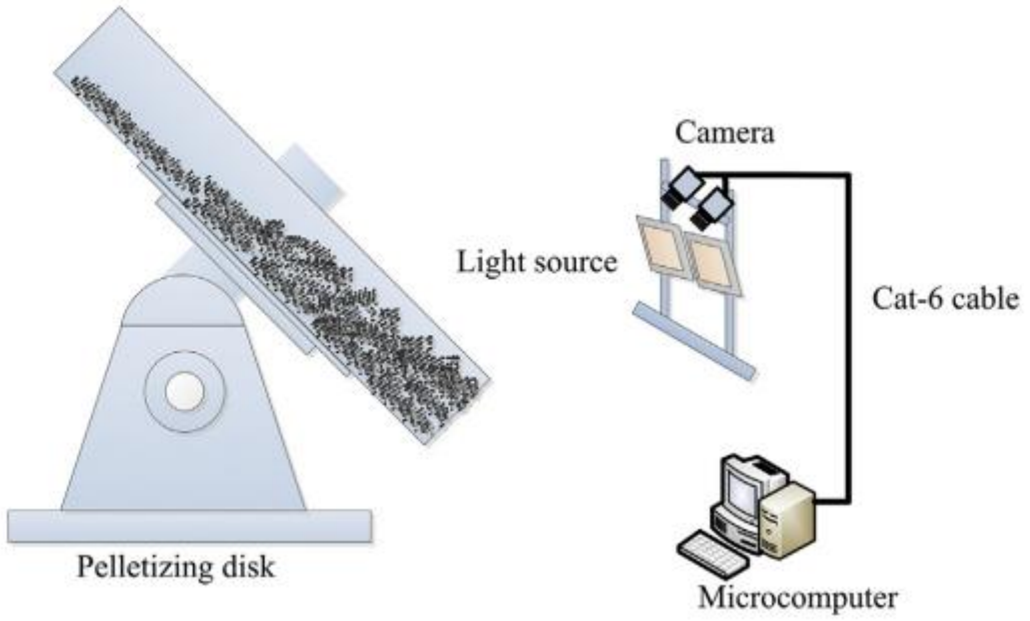


Fig. 2. Systematic setup of Camera, Light source and Processing device for Pellet Image processing and processing Machine Learning Models

Image Processing of Pellet Images

The whole source code is written on Python using the following open-source libraries and tools:

OpenCV (Open Source Computer Vision) Library

OpenCV is a well-known computer vision library that offers a number of tools and techniques for processing images and videos. It does image processing duties in this code.
Purpose: Image and video processing, computer vision tasks.
Features: Image manipulation, video processing, object detection, machine learning integration.
Functions: Image filtering, feature detection, face/object detection, camera calibration.
Platform: Cross-platform support (C++, Python, Java) on Windows, macOS, Linux, iOS, Android.
Applications: Robotics, surveillance, augmented reality, medical imaging, and more.

NumPy: NumPy (np) is an essential Python package for scientific computing. Large, multidimensional arrays and matrices are supported, in addition to a number of mathematical operations. It is used in this code for numerical calculations and array manipulation.

Tkinter (Tk): A common Python library for graphical user interface (GUI) development. It offers a selection of widgets and building blocks for GUI applications. To pick an image file, a file dialog is opened in this code using Tkinter.

Importing Dependencies Python code snippet:

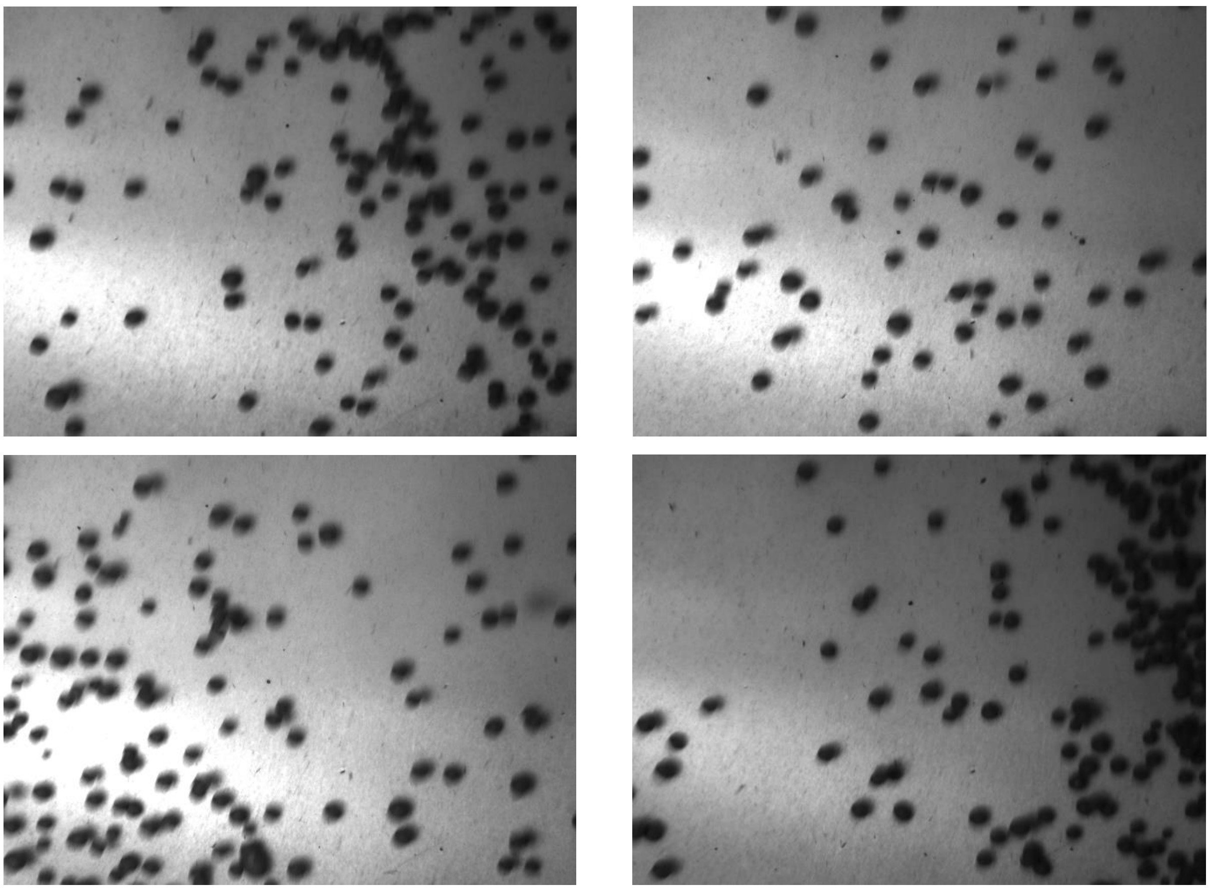
```python
import cv2 as cv
import numpy as np
from tkinter import Tk
from tkinter.filedialog import askopenfilename
```

Create a Tkinter root window and prompt the user to select an image file:

```
# Create a Tkinter root window
root = Tk()
root.withdraw()

# Prompt the user to select an image file
print("Select an image file:")
image_path = askopenfilename()
```

Sample Pellet Images before reading images



Function to rescale the image:

```
def rescaleframe(frame, scale=1):
    width = int(frame.shape[1] * scale)
    height = int(frame.shape[0] * scale)
    dimensions = (width, height)
    return cv.resize(frame, dimensions,
interpolation=cv.INTER_AREA)
```

The rescale_frame function applies a scaling factor to resize an input image. Based on the original measurements and the supplied scale, it determines the new width and height of the image. Following that, the resizing is carried out using the computed dimensions and the cv.INTER_AREA interpolation method for downsampling. As output, the scaled image is given back.

Gaussian Smoothing/Blurring

Gaussian Smoothing, also known as Gaussian Blurring, is a widely used image processing technique for reducing image noise and enhancing image features. It involves applying a Gaussian filter to the image, which performs a weighted average of nearby pixels, giving more weight to the central pixels and less weight to the pixels farther away. This weighted averaging helps to blur the image and suppress small fluctuations or noise while preserving important image structures.

The Gaussian filter is defined by a Gaussian kernel, which is a two-dimensional matrix with a bell-shaped distribution. The size and standard deviation (sigma) of the kernel determine the extent of blurring. Larger kernel sizes and higher sigma values result in stronger blurring.

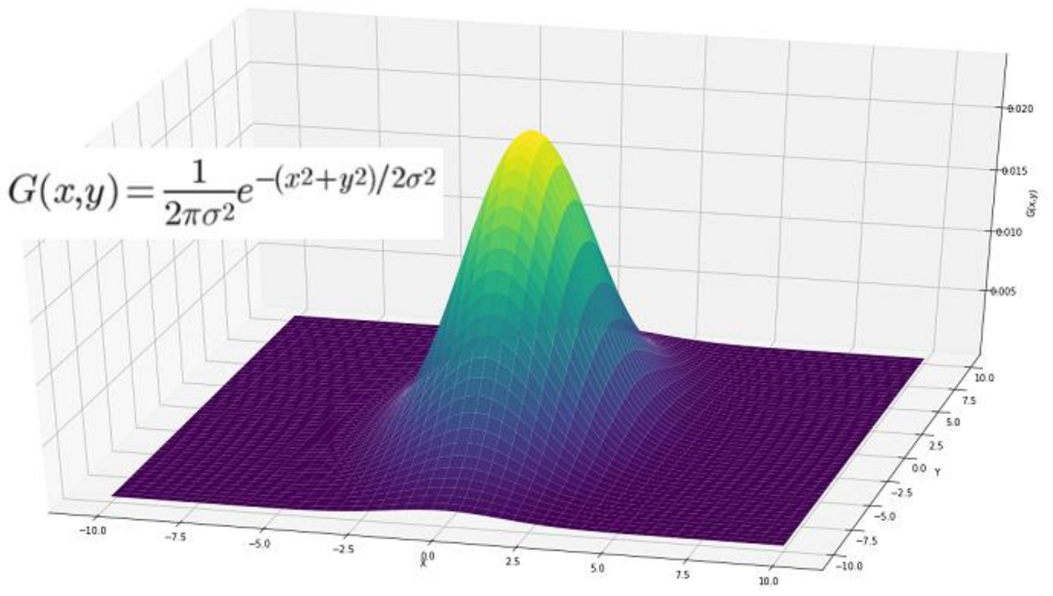
The Gaussian smoothing process is mathematically represented as the convolution of the image with the Gaussian kernel. The convolution operation involves sliding the kernel over the image and calculating the weighted average at each pixel location.

Python-OpenCV(CV2) format of function:

```
cv.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[,
borderType]]])
```

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma 2}$$

Above is the Gaussian Normal Distribution (Bell Curve) function. Area under the f(x) vs x plot is 1.



$$G(x,y) = \frac{1}{2\pi\sigma 2} e^{-(x2+y2)/2\sigma 2}$$

Above is the plot of 2 dimensional Gaussian Normal Distribution function

F(x,y) = f(x)*f(y)

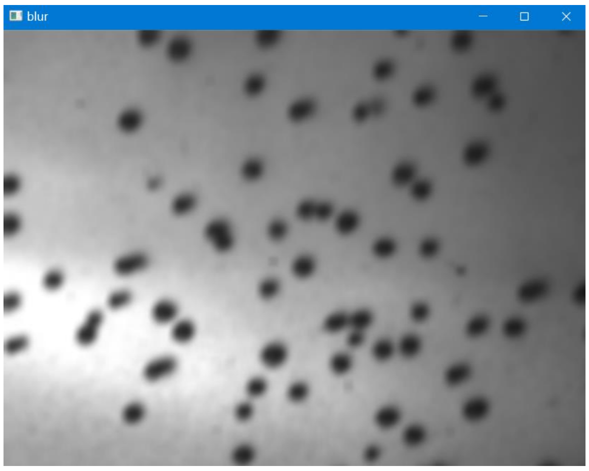
$$\frac{1}{2\pi\sigma^2} = \frac{1}{2\times 3.14\times 0.6\times 0.6} = \frac{1}{2.2619}$$

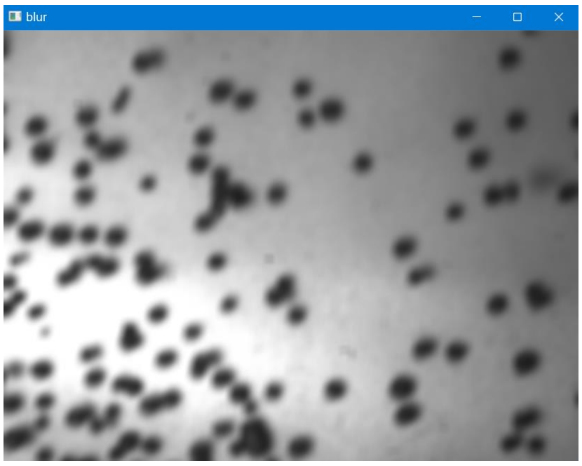
4. The width of the kernel is X = 3 and the height of the kernel is Y =3

i.e $X = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ and $Y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$

$$\frac{-(x^2+y^2)}{2\sigma^2} = \begin{bmatrix} -2.7778 & -1.3889 & -2.7778 \\ -1.3889 & 0 & -1.3889 \\ -2.7778 & -1.3889 & -2.7778 \end{bmatrix}$$

3x3 kernel size Gaussian Blur backend calculation for a pixel with matrices X and Y





2 sample Images after Guassian Smoothing with parameters mentioned below

Kernel size = (11,11)

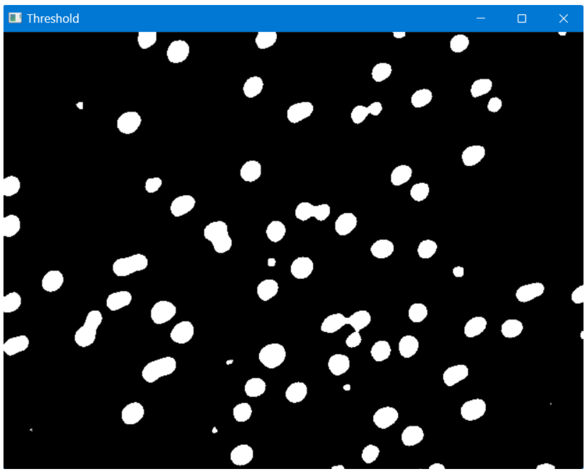
SigmaX = 20, SigmaY = 20

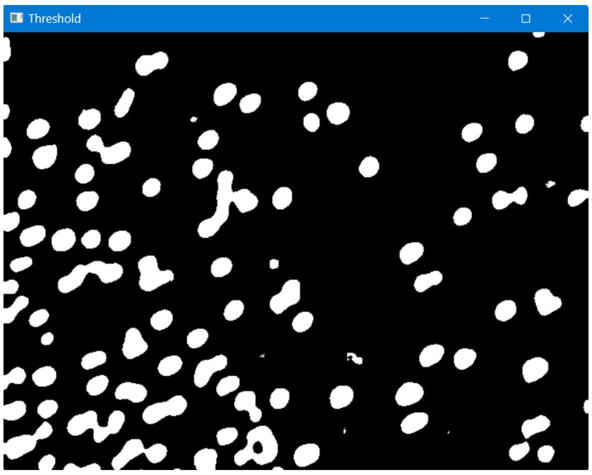
<u>Adaptive Threshold</u>

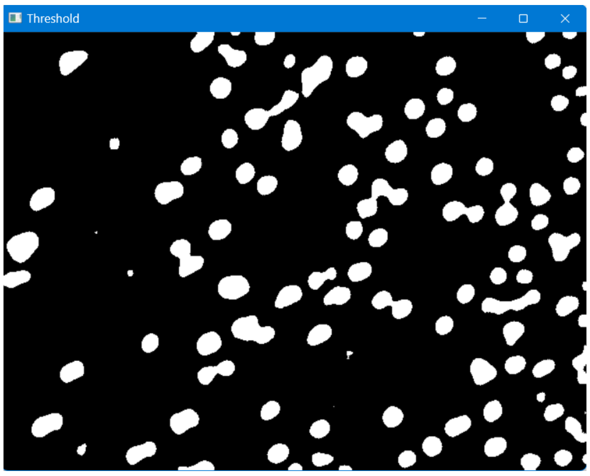
Adaptive Thresholding is a thresholding technique used in image processing to binarize images, where each pixel's value is converted to either black (0) or white (255) based on a local threshold computed for that pixel's neighborhood. This method is particularly useful for images with varying illumination or contrast, as it adapts the thresholding operation to the local image characteristics.

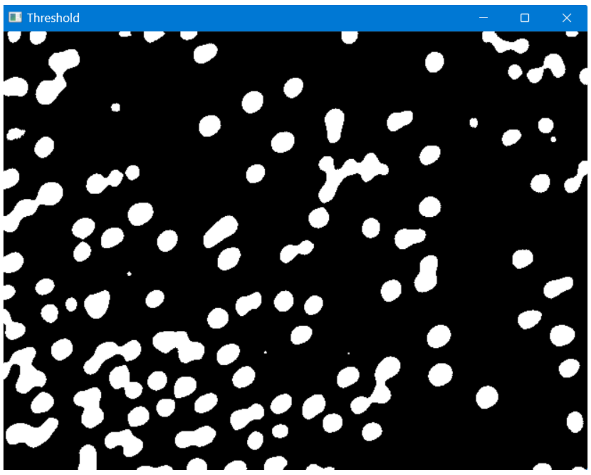
The process of adaptive thresholding involves the following steps:

- Divide the image into smaller regions or tiles.
- Compute a threshold value for each tile based on its local pixel intensities or statistics.
- Binarize each tile based on its computed threshold, converting pixel values to black or white.









Python-OpenCV(CV2) format of function:

cv.adaptiveThreshold(src, maxValue, adaptiveMethod, thresholdType, blockSize, C[, dst])

src: This is the input image on which the adaptive thresholding will be applied. The input image should be in grayscale format, represented as a 2-dimensional NumPy array with shape (height, width). Each pixel in the grayscale image is represented by a single numerical value, which indicates the grayscale intensity.

maxValue: This parameter defines the maximum pixel value to be assigned to the pixels that pass the thresholding condition. Typically, it is set to 255, which corresponds to white pixels in the binary output image. Pixels that do not meet the thresholding condition will be assigned a value of 0, corresponding to black pixels in the binary output image.

adaptiveMethod: This parameter determines the method used to compute the local threshold for each pixel. It can take one of the following two values:

1. cv.ADAPTIVE_THRESH_MEAN_C: This method calculates the threshold value as the mean of the pixel intensities in the local neighborhood defined by the blockSize.

2. cv.ADAPTIVE_THRESH_GAUSSIAN_C: This method calculates the threshold value as the weighted sum of pixel intensities in the local neighborhood using a Gaussian window. It often provides better results when there is uneven illumination or varying contrast in the image.

thresholdType: This parameter specifies the type of thresholding applied after computing the local threshold for each pixel. It can take one of the following two values:

1. cv.THRESH_BINARY: The pixel values that pass the thresholding condition will be set to maxValue, and the remaining pixels will be set to 0.
2. cv.THRESH_BINARY_INV: The pixel values that pass the thresholding condition will be set to 0, and the remaining pixels will be set to maxValue.

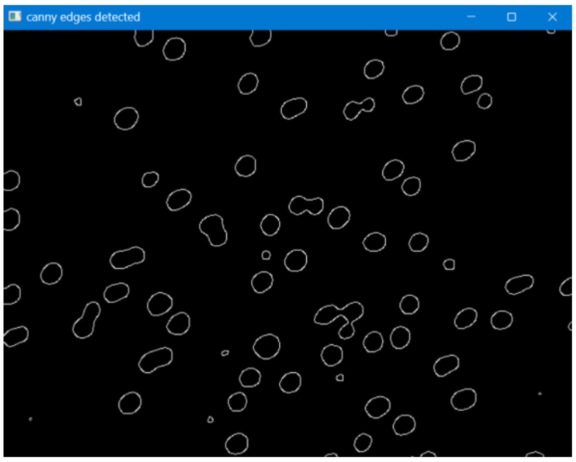
blockSize: This parameter sets the size of the local neighborhood (tile) used for computing the local threshold for each pixel. It must be an odd integer value greater than 1. The larger the blockSize, the larger the area considered for thresholding, which can affect the level of smoothing or detail preservation in the binary output image.

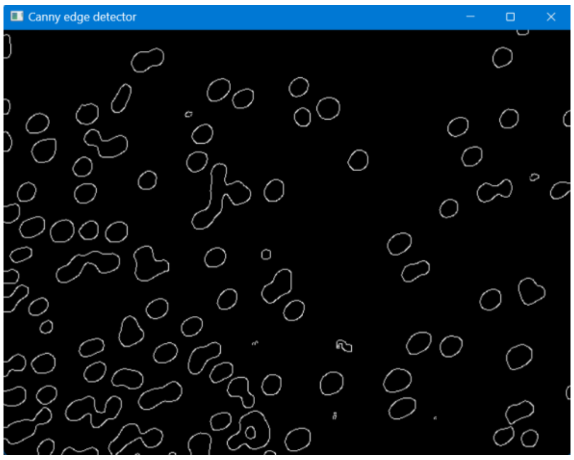
C: This parameter is a constant subtracted from the mean or weighted sum calculated by the adaptive method. It is used to fine-tune the threshold value. A positive C value reduces the threshold, making the result more inclusive (more white pixels), while a negative value makes it less inclusive (more black pixels). It helps in handling images with varying illumination or contrast.

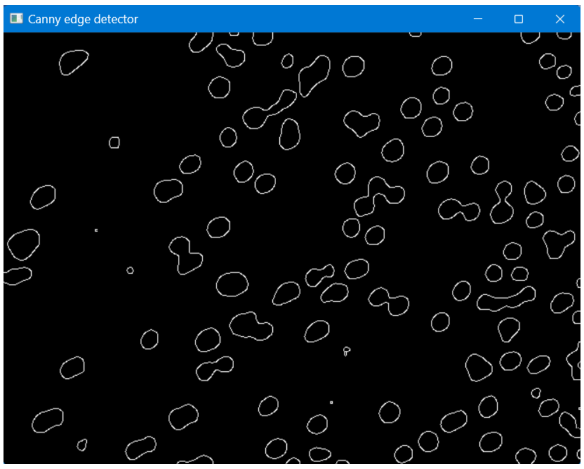
dst (optional): This is an optional parameter that represents the output image, where the binary thresholding result will be stored. If provided, the binary image will be stored in the dst array. If dst is not provided, a new array will be created to store the output.

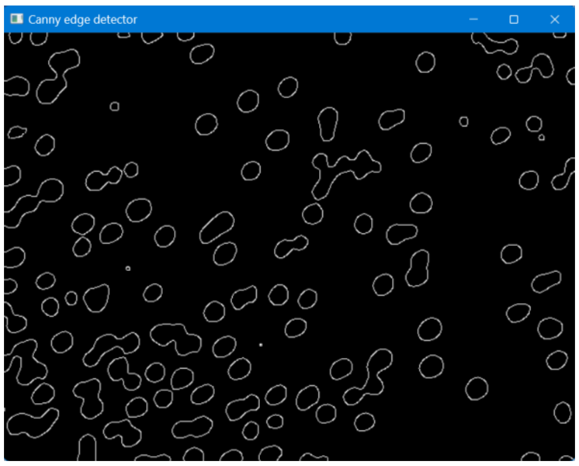
Canny Edge Detection

The Canny edge detector is an image processing algorithm used to identify the boundaries of objects within an image. It works by detecting significant changes in pixel intensity, which correspond to edges or transitions between different regions in the image. The output is a binary image, where edge pixels are highlighted as white, and non-edge pixels are black, providing a clear representation of the object boundaries. The Canny edge detector is widely employed in various computer vision applications for feature extraction, image segmentation, and object detection.









Python-OpenCV(CV2) format of function:

```
cv.Canny(image, threshold1, threshold2[, edges[, apertureSize[,
L2gradient]]])
```

- **image**: This is the input image on which the Canny edge detection will be applied. The image should be in grayscale format, represented as a 2-dimensional NumPy array with shape (height, width). Each pixel in the grayscale image is represented by a single numerical value, indicating the grayscale intensity.
- **threshold1**: This parameter sets the lower threshold value for edge detection. It is a numerical value that determines the minimum gradient magnitude required for a pixel to be considered as an edge. Pixels with gradient magnitudes below this threshold will not be marked as edges.

- **threshold2**: This parameter sets the upper threshold value for edge detection. It is a numerical value that determines the maximum gradient magnitude required for a pixel to be initially considered as a strong edge. Pixels with gradient magnitudes above this threshold will be marked as strong edges.
- **edges** (optional): This is an optional output parameter. It represents the binary image where the detected edges will be stored. If provided, the function will store the edge map in the edges array. If not provided, a new array will be created to store the output.
- **apertureSize** (optional): This is an optional parameter that sets the aperture size for the Sobel operator used to compute image gradients. It must be an odd integer, and a common value is 3. Larger aperture sizes can give smoother gradients, but they may blur small edges.
- **L2gradient** (optional): This is an optional boolean parameter that indicates the type of gradient calculation to be used. If set to True, the function uses the L2 norm (Euclidean distance) for gradient magnitude calculation. If set to False, it uses the L1 norm (Manhattan distance). The default value is False.

IMAGE PRE-PROCESSING CODE SNIPPET

```python
# Prompt the user to select an image file
print("Select an image file:")
image_path = askopenfilename()
img = cv.imread(image_path)

# rescale the image
imgrz = rescaleframe(img)

# converting rescaled image to grayscale
gray = cv.cvtColor(imgrz, cv.COLOR_BGR2GRAY)
cv.imshow('B&W', gray)

# blur the gray image
blur = cv.GaussianBlur(gray, (11, 11), 20)
```

```
# adaptive thresholding
ad_th = cv.adaptiveThreshold(blur, 255,
cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, 47, 4)

# canny edge detector
canny = cv.Canny(ad_th, 0, 50)

# thiker & visible.
dilated = cv.dilate(canny, (1, 1), iterations=0)
cv.imshow('thiker & visible.', dilated)
```

Image Processing Steps:

The provided code snippet performs several image processing steps on the selected image. Here's a summary of the processing steps:

1. The user is prompted to select an image file.
2. The selected image is read using cv.imread(), and the image path is stored in the image_path variable.
3. The image is rescaled using the rescaleframe() function, and the rescaled image is stored in the imgrz variable.
4. The rescaled image is converted to grayscale using cv.cvtColor() with the cv.COLOR_BGR2GRAY conversion.
5. The grayscale image is blurred using cv.GaussianBlur() with a kernel size of (11, 11) and a sigma of 20 in both X and Y directions.
6. Adaptive thresholding is applied to the blurred image using cv.adaptiveThreshold(). The thresholding method used is cv.ADAPTIVE_THRESH_GAUSSIAN_C, and the thresholding parameters are set to 255 (maximum pixel value) for binary thresholding, a block size of 47 for the neighborhood size, and a constant of 4 for the subtraction value from the mean.
7. The Canny edge detector is then used on the adaptive thresholded image using cv.Canny(). The lower threshold is set to 0 and the upper threshold to 50.

8. The edges are made thicker and more visible by dilating the edges using cv.dilate() with a kernel size of (1, 1) and zero iterations.
9. The processed images at various stages are displayed using cv.imshow().
10. These image processing steps are commonly used to prepare images for further analysis or feature extraction, such as object detection, segmentation, or contour detection.

Hough Circle Transform

The Hough Circle Transform is an image processing technique used to detect circles in digital images. It is an extension of the Hough Line Transform and is particularly useful for detecting circles of different sizes and positions within an image.

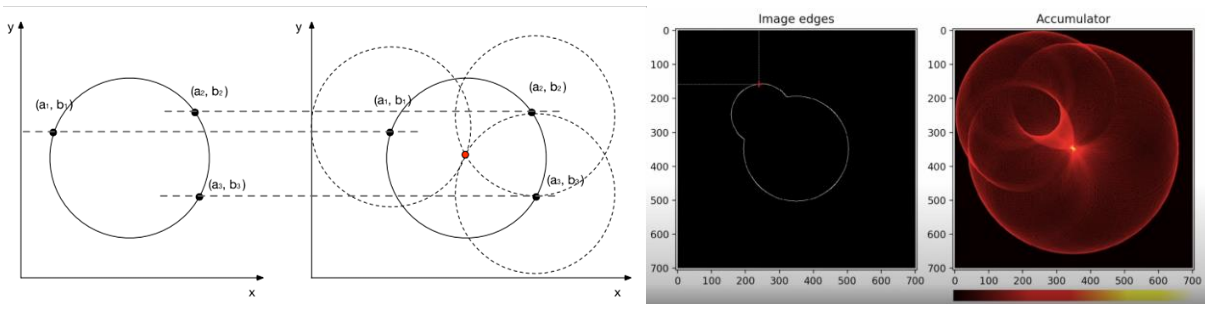


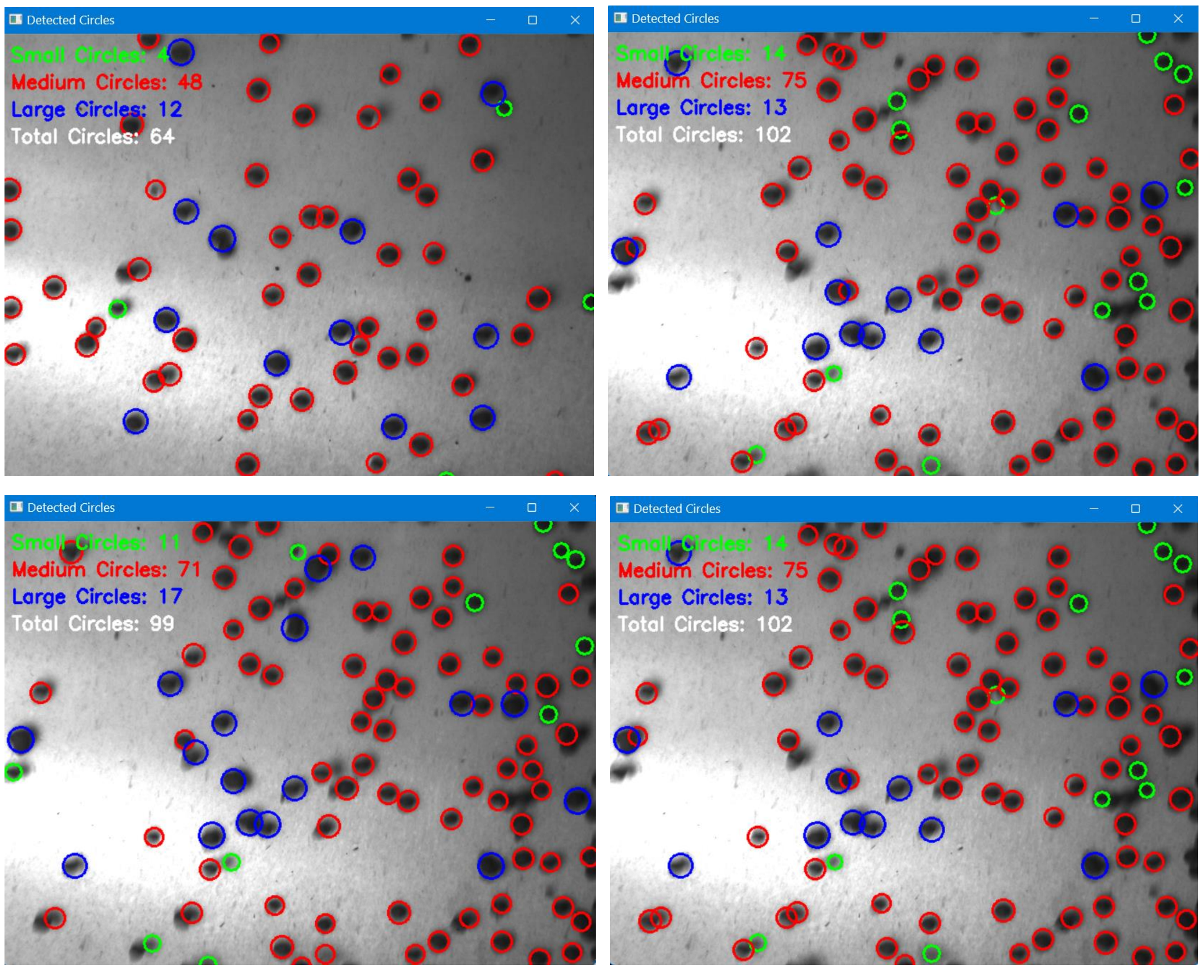
Fig: Circles are detected by accumulator

The Hough Circle Transform involves the following steps:

1. **Edge Detection**: Before applying the Hough Circle Transform, the input image is typically preprocessed to detect edges. Common edge detection techniques such as the Canny edge detector are often used to identify the edges of objects within the image.
2. **Circle Parameter Space**: The Hough Circle Transform creates a 3D parameter space, where each point represents a possible circle in the image. The three parameters are the (x, y) coordinates of the circle's center and the radius (r) of the circle.
3. **Accumulation**: For each edge pixel in the image, the Hough Circle Transform calculates possible circles that could pass through that point. It then accumulates votes for all potential circles in the parameter space.

4. **Circle Detection**: After the accumulation process, the parameter space is examined to find peaks or local maxima. These peaks correspond to the centers and radii of the detected circles.
5. **Non-maximum Suppression**: To avoid multiple detections for a single circle, a non-maximum suppression step is performed to filter out redundant circles with similar parameters.

Circles detected through Hough Transform with optimized parameters suiting our Pellet images and plotting them back on Original Images



```
Small Circles: 4
Medium Circles: 48
Large Circles: 12
Total Circles: 64
```

```
Small Circles: 14
Medium Circles: 75
Large Circles: 13
Total Circles: 102
```

```
Small Circles: 11
Medium Circles: 71
Large Circles: 17
Total Circles: 99
```

```
Small Circles: 14
Medium Circles: 75
Large Circles: 13
Total Circles: 102
```

Python-OpenCV format for Hough Circle Transform:

cv.HoughCircles(image, method, dp, minDist[, circles[, param1[, param2[, minRadius[, maxRadius]]]]])

image: This is the input image on which the Hough Circle Transform will be applied. The image should be in grayscale format, represented as a 2-dimensional NumPy array with shape (height, width). It is essential to perform edge detection on the input image before applying the Hough Circle Transform.

method: This parameter specifies the method used for detecting circles in the image. In the context of Hough Circle Transform, it should be set to cv.HOUGH_GRADIENT, which indicates the detection method based on the gradient.

dp: The inverse ratio of the accumulator resolution to the image resolution. A smaller value of dp means a higher resolution of the accumulator. Typically, dp can be set to 1 for the same resolution as the input image or 2 for half the resolution.

minDist: This parameter sets the minimum distance between the centers of detected circles. If the distance is too small, multiple circles may be detected for a single circle. minDist should be adjusted based on the expected size of circles in the image.

circles (optional): This is an optional output parameter. It represents the output list of detected circles, where each circle is represented by its (x, y) center coordinates and radius. If provided, the function will store the detected circles in the circles array.

param1: The first method-specific parameter. For Hough Circle Transform, it represents the upper threshold for the internal Canny edge detection used in the circle detection process. It determines the sensitivity of the edge detection.

param2: The second method-specific parameter. For Hough Circle Transform, it represents the accumulator threshold for circle detection. It determines the minimum number of votes required for a circle to be detected. Increasing this threshold can help reduce false positive detections.

minRadius: The minimum radius of the circles to be detected. Circles with radii smaller than minRadius will not be detected.

maxRadius : The maximum radius of the circles to be detected. Circles with radii larger than maxRadius will not be detected.

Circle Detection through Hough Transform Code snippet:

```
# function to detect the circles
def detect_circles(image, min_radius, max_radius, threshold):
    # Perform Hough Circle Transform
    circles = cv.HoughCircles(image, cv.HOUGH_GRADIENT, 1, 8,
param1=4, param2=threshold,minRadius=min_radius,
maxRadius=max_radius)
    # If circles are detected, extract and return the circles
    if circles is not None:
        num_circles = len(circles[0])
        print("Number of circles detected:", num_circles)
        circles = np.round(circles[0, :]).astype(int)
        return circles
    else:
        return []
```

The function takes four parameters: image, min_radius, max_radius, and threshold.

image: This represents the input image on which circle detection is performed.
min_radius: This sets the minimum radius of circles to be detected.
max_radius: This sets the maximum radius of circles to be detected.
threshold: This is the accumulator threshold for circle detection, determining the minimum number of votes required for a circle to be detected.

The Hough Circle Transform method is applied using cv.HoughCircles(), which takes the following parameters:

image: The input image on which the transform is performed.
cv.HOUGH_GRADIENT: The method used for circle detection.
dp: 1, Inverse ratio of the accumulator resolution to the image resolution.
minDist: 8, Minimum distance between the centers of detected circles.
param1=4: Upper threshold for the internal Canny edge detection used in the Hough Circle Transform.

If circles are detected, the function returns the coordinates and radii of the detected circles as an array.
If no circles are detected, an empty array is returned.
Additionally, the function prints the number of circles detected before returning the results.

Min Radius, Max Radius and Param2 declared separately:

```
# Set the parameters for circle detection
min_radius = 8
max_radius = 15
threshold = 6
```

The provided code snippet sets the parameters used for circle detection through the Hough Circle Transform. Here are the parameters and their respective values:

min_radius: This parameter sets the minimum radius of the circles to be detected. It is assigned a value of 8.

max_radius: This parameter sets the maximum radius of the circles to be detected. It is assigned a value of 15.

threshold: param2, This parameter is the accumulator threshold for circle detection, determining the minimum number of votes required for a circle to be detected. It is assigned a value of 6.

These parameter values define the criteria for detecting circles in the input image using the Hough Circle Transform method. Adjusting these values can impact the sensitivity and accuracy of the circle detection process.

```
# Detect circles using CHT[Hough Circle Detection]
detected_circles = detect_circles(dilated, min_radius,
max_radius, threshold)

# Define the size categories
```

```
small_circles = []
medium_circles = []
large_circles = []

# Categorize the detected circles based on their radii
for (x, y, r) in detected_circles:
    if r < 10:
        small_circles.append((x, y, r))
    elif r >= 10 and r < 13:
        medium_circles.append((x, y, r))
    else:
        large_circles.append((x, y, r))

# Draw detected circles on the copy of the original image, with
different colors for each category
imgrz_copy = np.copy(imgrz)
for (x, y, r) in small_circles:
    cv.circle(imgrz_copy, (x, y), r, (0, 255, 0), 2)
for (x, y, r) in medium_circles:
    cv.circle(imgrz_copy, (x, y), r, (0, 0, 255), 2)
for (x, y, r) in large_circles:
    cv.circle(imgrz_copy, (x, y), r, (255, 0, 0), 2)

# Display the number of circles in each category
small_circle_text = f"Small Circles: {len(small_circles)}"
medium_circle_text = f"Medium Circles: {len(medium_circles)}"
large_circle_text = f"Large Circles: {len(large_circles)}"
total_circle_text = f"Total Circles: {len(detected_circles)}"
cv.putText(imgrz_copy, small_circle_text, (10, 30),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
cv.putText(imgrz_copy, medium_circle_text, (10, 60),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
cv.putText(imgrz_copy, large_circle_text, (10, 90),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)
cv.putText(imgrz_copy, total_circle_text, (10, 120),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
```

Circle Categorization and Visualization:

In this section of the code, the detected circles are categorized based on their radii, and then the circles are drawn on a copy of the original image with different colors for each category. Additionally, the number of circles in each category is displayed on the image using text annotations.

The lists small_circles, medium_circles, and large_circles are defined to store the circles based on their radii.

The for loop iterates through the detected_circles list, which contains the coordinates and radii of the detected circles.

Inside the loop, each circle is checked based on its radius (r). If the radius is less than 10, the circle is categorized as a small circle and appended to the small_circles list. If the radius is between 10 (inclusive) and 13 (exclusive), the circle is categorized as a medium circle and appended to the medium_circles list. Otherwise, if the radius is greater than or equal to 13, the circle is categorized as a large circle and appended to the large_circles list.

A copy of the original image, imgrz, is created using np.copy(imgrz) and stored in the variable imgrz_copy.

Using cv.circle(), circles are drawn on imgrz_copy for each category with different colors: green for small circles, blue for medium circles, and red for large circles. The cv.circle() function takes the center coordinates (x, y) and radius r as well as the color and thickness parameters to draw the circles.

Text annotations are added to imgrz_copy using cv.putText() to display the number of circles in each category. The text is displayed in green for small circles, blue for medium circles, and white for total circles.

The final result with the categorized circles and annotations is displayed using cv.imshow(). This categorization and visualization step helps in understanding the distribution of circles in the image based on their sizes and assists in further analysis or decision-making processes.

```
# Export circles data to CSV file
with open(csv_file_path, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['x', 'y', 'diameter'])
    for (x, y, r) in detected_circles:
        writer.writerow([x, y, r * 2])

print(f"CSV file saved to: {csv_file_path}")

# Calculate histogram data
all_radii = [r * 2 for (_, _, r) in detected_circles]
hist, bins = np.histogram(all_radii, bins=10,
range=(min_radius*2, max_radius*2))

# Plot the histogram
plt.bar(bins[:-1], hist, width=(max_radius*2 - min_radius*2) /
10, align='edge')
plt.xlabel('Diameter')
plt.ylabel('Frequency')
plt.title('Circle Diameter Distribution')
plt.show()

# Display the image with detected circles
cv.imshow("Detected Circles", imgrz_copy)
cv.waitKey(0)
```

Exporting Pellet Data and Histogram Plot:

This section of the code performs the following tasks:

Export Circles Data to CSV File: The detected circles data, including the center coordinates (x, y) and diameter (2 * r), are exported to a CSV (Comma-Separated Values) file.
The file is created and opened in write mode using open() with the provided csv_file_path.

The csv.writer is initialized to write data to the CSV file. The header row, ['x', 'y', 'diameter'], is written to the CSV file. For each detected circle in detected_circles, the x, y, and diameter values are written as a row in the CSV file (diameter is calculated as 2 * r).

The CSV file is then closed, and a message is printed indicating that the file has been saved to the provided csv_file_path.

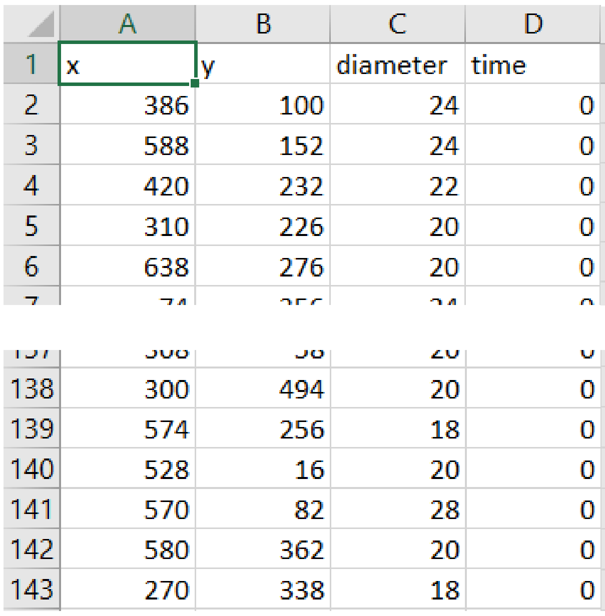
Calculate Histogram Data: All radii from the detected circles are extracted and stored in the all_radii list. The radii are doubled (2 * r) to represent the diameter. A histogram of the diameter distribution is calculated using np.histogram(). It counts the occurrences of different diameter values in the all_radii list. The histogram and the corresponding bin values are stored in hist and bins variables, respectively.

Plot Histogram: The histogram data is plotted using plt.bar(). It creates a bar plot, where the bins are on the x-axis, and the frequency (count) of circles falling within each bin is on the y-axis. The histogram bars' width is determined based on the range of diameters and the number of bins (10) used in the histogram calculation. The plot is labeled with the x-axis as 'Diameter', the y-axis as 'Frequency', and the title as 'Circle Diameter Distribution'.

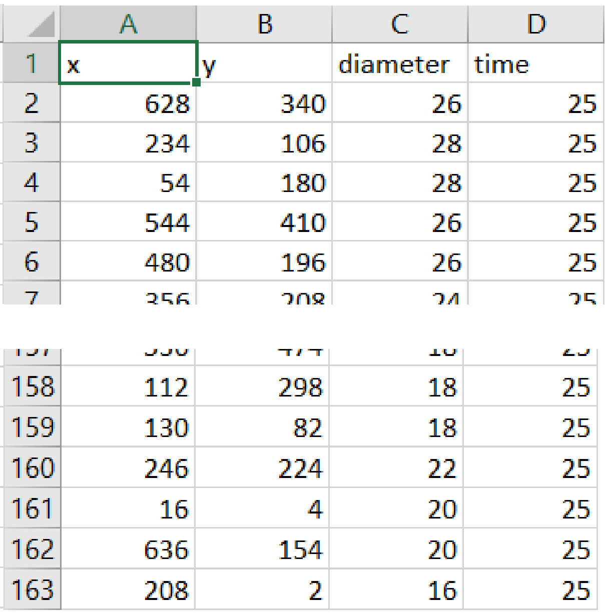
The histogram plot is displayed using plt.show().

Display the Image with Detected Circles: The image with detected circles, along with the categorized circles and annotations, is displayed using cv.imshow(). The imgrz_copy variable holds the image with drawn circles and annotations.

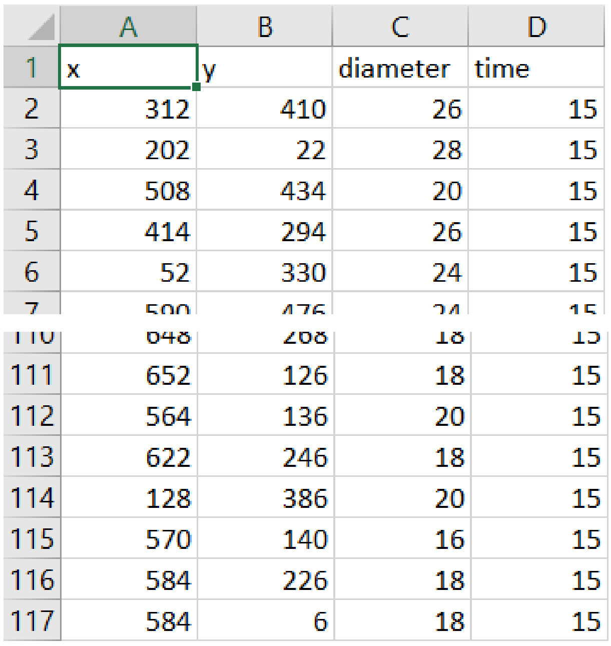
Pellet.csv files/Pandas Data Frames of 4 sample pellet images



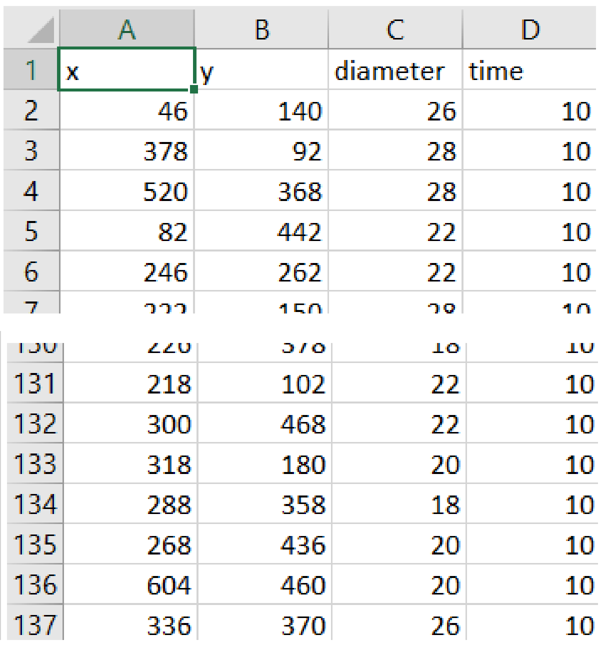
	A	B	C	D
1	x	y	diameter	time
2	386	100	24	0
3	588	152	24	0
4	420	232	22	0
5	310	226	20	0
6	638	276	20	0
138	300	494	20	0
139	574	256	18	0
140	528	16	20	0
141	570	82	28	0
142	580	362	20	0
143	270	338	18	0



	A	B	C	D
1	x	y	diameter	time
2	628	340	26	25
3	234	106	28	25
4	54	180	28	25
5	544	410	26	25
6	480	196	26	25
158	112	298	18	25
159	130	82	18	25
160	246	224	22	25
161	16	4	20	25
162	636	154	20	25
163	208	2	16	25

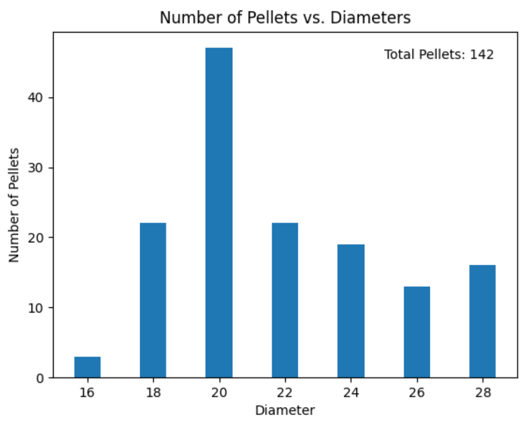


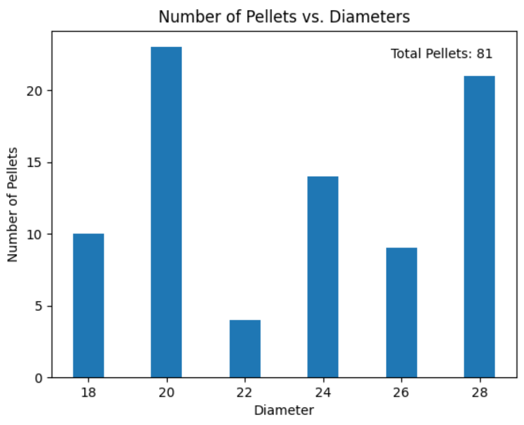
	A	B	C	D
1	x	y	diameter	time
2	312	410	26	15
3	202	22	28	15
4	508	434	20	15
5	414	294	26	15
6	52	330	24	15
111	652	126	18	15
112	564	136	20	15
113	622	246	18	15
114	128	386	20	15
115	570	140	16	15
116	584	226	18	15
117	584	6	18	15

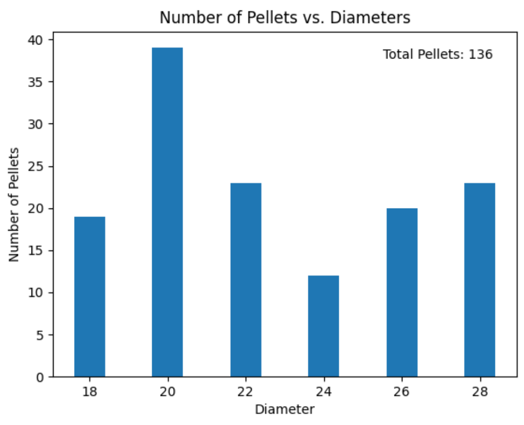


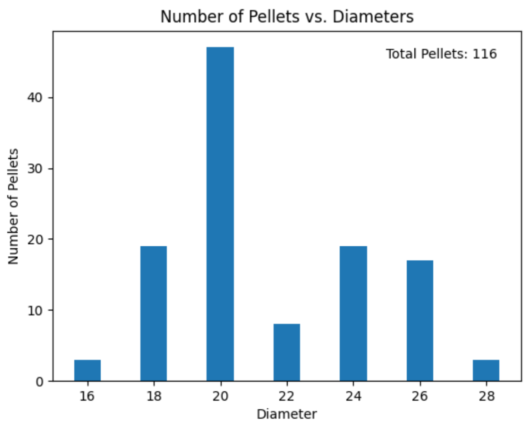
	A	B	C	D
1	x	y	diameter	time
2	46	140	26	10
3	378	92	28	10
4	520	368	28	10
5	82	442	22	10
6	246	262	22	10
131	218	102	22	10
132	300	468	22	10
133	318	180	20	10
134	288	358	18	10
135	268	436	20	10
136	604	460	20	10
137	336	370	26	10

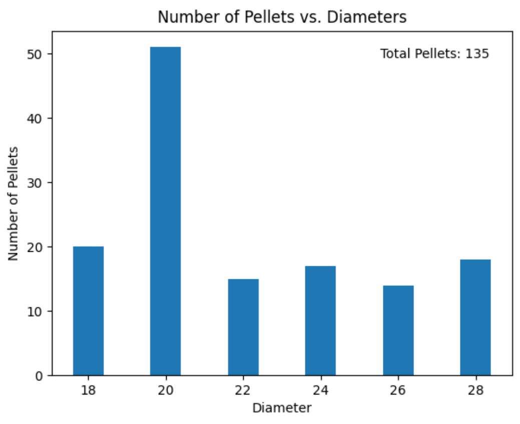
Size Distribution Histograms of 6 sample pellet Images

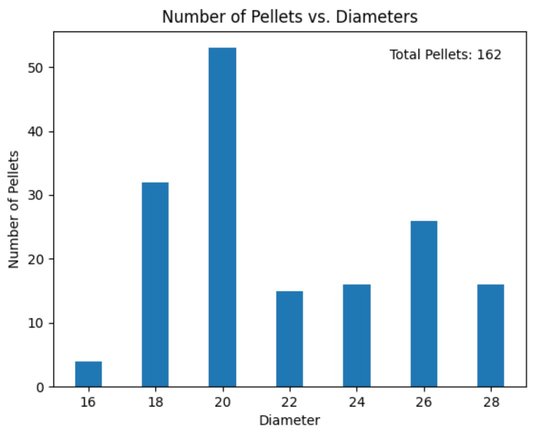












DATA ANALYSIS AND TIME SERIES FORECASTING USING MACHINE LEARNING ON EXTRACTED PELLET DATA (.CSV Files)

Importing Dependencies Python code snippet

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression,
LogisticRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
import seaborn as sns
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima.model import ARIMA
from prophet import Prophet
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
```

- **pandas** (imported as pd): Data manipulation and analysis library with DataFrames and Series.
- **numpy** (imported as np): Fundamental library for numerical computing with arrays and mathematical functions.
- **matplotlib.pyplot** (imported as plt): Data visualization library for creating various plots.
- **train_test_split from sklearn.model_selection**: Function to split a dataset into training and testing subsets.

- **LinearRegression from sklearn.linear_model**: Class for linear regression models.
- **LogisticRegression from sklearn.linear_model**: Class for logistic regression models used in binary classification.
- **RandomForestRegressor from sklearn.ensemble**: Class for random forest regression models.
- **SVR from sklearn.svm**: Class for Support Vector Regression.
- **mean_squared_error from sklearn.metrics**: Function to compute the mean squared error metric.
- **seaborn** (imported as sns): Statistical data visualization library.
- **SARIMAX from statsmodels.tsa.statespace.sarimax**: Statistical method for time series forecasting.
- **ARIMA from statsmodels.tsa.arima.model**: Classical time series forecasting method.
- **Prophet from prophet**: Library for time series forecasting with multiple seasonalities.
- **GaussianProcessRegressor from sklearn.gaussian_process**: Class for Gaussian Process Regression.
- **RBF from sklearn.gaussian_process.kernels**: Radial Basis Function kernel for Gaussian Process Regression.

CSV Files Loading and DataFrame Concatenation with Time-based Filtering

```
# Load CSV files into pandas DataFrames
file_paths = ['pellet0.csv', 'pellet1.csv', 'pellet2.csv',
'pellet35.csv', 'pellet66.csv', 'pellet67.csv']
dataframes = [pd.read_csv(file_path) for file_path in
file_paths]

# Concatenate DataFrames into a single DataFrame
df = pd.concat(dataframes, ignore_index=True)

# Filter DataFrame for t = 0, 5, 10, 15, 20, 25 seconds
time_points = [0, 5, 10, 15, 20, 25]
```

```
df_filtered = df[df['time'].isin(time_points)]
```

1. **Loading CSV Files into Pandas DataFrames:** The code loads multiple CSV files (pellet0.csv, pellet1.csv, pellet2.csv, pellet35.csv, pellet66.csv, pellet67.csv) into separate Pandas DataFrames using a list comprehension.
2. **Concatenating DataFrames:** The code concatenates the previously loaded DataFrames into a single DataFrame (df) using pd.concat(). The ignore_index=True parameter ensures that the resulting DataFrame has continuous row indexing.
3. **Filtering the DataFrame:** The code filters the concatenated DataFrame (df) to retain only rows with specific time points (0, 5, 10, 15, 20, 25 seconds). The filtered DataFrame is stored in df_filtered. The isin() method is used to select rows with the specified time points from the 'time' column.

Data Preparation and Train-Test Splitting

```
# Prepare the features and target variable
X = df_filtered[['x', 'y', 'time']]
y = df_filtered['diameter']

# Split the data into training and test sets
X_train,  X_test,  y_train,  y_test  =  train_test_split(X,  y,
test_size=0.2, random_state=42)
```

Data Preparation: The code prepares the features (independent variables) and the target variable (dependent variable) from the filtered DataFrame df_filtered. The features X are selected from the DataFrame df_filtered, consisting of the columns 'x', 'y', and 'time'. The target variable y is selected from df_filtered, representing the 'diameter' column.

Train-Test Splitting: The code then splits the data into training and test sets to evaluate the model's performance. The train_test_split function from sklearn.model_selection is used for the splitting.

The features X and target variable y are split into four separate subsets: X_train, X_test, y_train, and y_test.

The test_size parameter is set to 0.2, indicating that 20% of the data will be used for testing, and the remaining 80% will be used for training.

The random_state parameter is set to 42 to ensure reproducibility of the random split.

Supervised Machine Learning

Supervised Machine Learning is a type of machine learning where the algorithm learns from labeled training data to make predictions or decisions on new, unseen data. In this approach, the input data (features) and their corresponding output (target variable) are provided during the training process.

The main steps involved in Supervised Machine Learning are as follows:

Data Collection: Gather a dataset with labeled examples, where each data point consists of input features and their corresponding target values.

Data Preprocessing: Clean the data, handle missing values, and transform features into a suitable format for training.

Train-Test Split: Divide the dataset into two subsets: the training set (used to train the model) and the test set (used to evaluate the model's performance).

Model Selection: Choose a suitable machine learning algorithm (e.g., Linear Regression, Decision Trees, Support Vector Machines, Neural Networks) depending on the nature of the problem and the characteristics of the data.

Model Training: Feed the labeled training data into the selected algorithm to learn the relationships between features and the target variable. The algorithm adjusts its internal parameters to minimize the prediction errors.

Model Evaluation: Use the test set to assess the model's performance by comparing its predictions to the actual target values. Common evaluation metrics include accuracy, mean squared error, precision, recall, etc.

Model Deployment: If the model meets the desired performance criteria, it can be deployed to make predictions on new, unseen data.

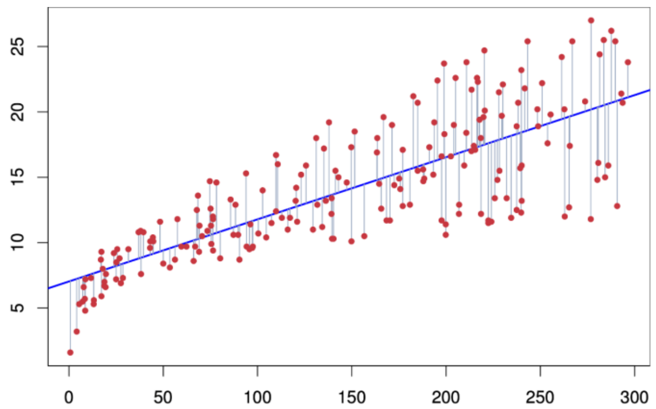
Machine Learning Models used to predict the Size Distribution of Pellets at time, t = 30 seconds

Linear Regression Model

A Linear Regressor is a specific machine learning model that falls under the broader category of supervised learning algorithms. It is used for regression tasks, where the goal is to predict a continuous numeric value (the target variable) based on one or more input features.

Linear Regression is a popular and widely used statistical technique for modeling the relationship between a dependent variable (also called the target or response variable) and one or more independent variables (also called predictors or features). It assumes a linear relationship between the variables and aims to find the best-fitting straight line that represents this relationship.

The formula for a simple linear regression with one independent variable can be represented as:



y = b0 + b1 * x

where:

y is the dependent variable (target),

x is the independent variable (feature),

b0 is the intercept (the value of y when x is 0),

b1 is the coefficient (the change in y for a one-unit change in x).

In multiple linear regression, the formula extends to include more than one independent variable:

y = b0 + b1 * x1 + b2 * x2 + ... + bn * xn

where x1, x2, ..., xn are the independent variables, and b1, b2, ..., bn are their respective coefficients.

The goal of linear regression is to estimate the coefficients (b0, b1, b2, ..., bn) that minimize the difference between the predicted values and the actual values of the target variable. This is usually done by minimizing the sum of squared residuals (differences between the predicted and actual values) using a method like Ordinary Least Squares (OLS).

Linear Regression is commonly used for various tasks, such as predicting numeric values (e.g., house prices, stock prices), understanding relationships between variables, and identifying the impact of specific factors on an outcome.

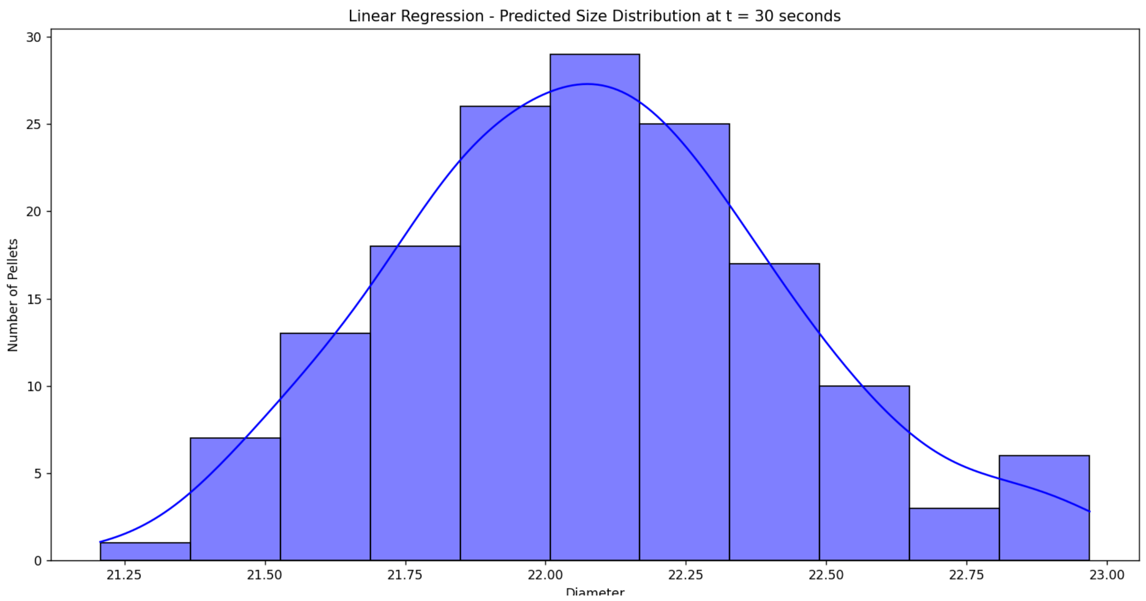
In the context of the provided code, after preparing the features and target variable, the X_train and y_train datasets can be used to train a Linear Regression model. This model can then be evaluated on the X_test and y_test datasets to assess its performance in predicting the 'diameter' based on the 'x', 'y', and 'time' features.

```
# Train and predict with Linear Regression
linear_reg = LinearRegression()
linear_reg.fit(X_train, y_train)
linear_pred = linear_reg.predict(X_test)
```

```
linear_mse = mean_squared_error(y_test, linear_pred)
linear_total_pellets = calculate_total_pellets(linear_pred)


# Visualize the predicted values in separate windows
# Linear Regression
plt.figure(figsize=(8, 6))
sns.histplot(linear_pred, bins='auto', kde=True, color='blue')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Linear Regression - Predicted Size Distribution at t
= 30 seconds')
plt.text(0, linear_total_pellets + 5, f'Total Pellets:
{linear_total_pellets}', fontsize=10, ha='left')
plt.show()
```

Predicted Size Distribution at t = 30 seconds using Linear Regression Model



Random Forest Regression Model

The Random Forest Regressor is a popular machine learning model that belongs to the ensemble learning family. It is an extension of the decision tree algorithm and is used for regression tasks, where the goal is to predict a continuous numeric value (the target variable) based on one or more input features.

Random Forest Regressor works by constructing multiple decision trees during the training process and combining their predictions to make the final prediction. Each decision tree is built on a random subset of the training data and a random subset of the features. These randomizations introduce diversity in the trees, reducing the risk of overfitting and improving the overall model's accuracy and generalization.

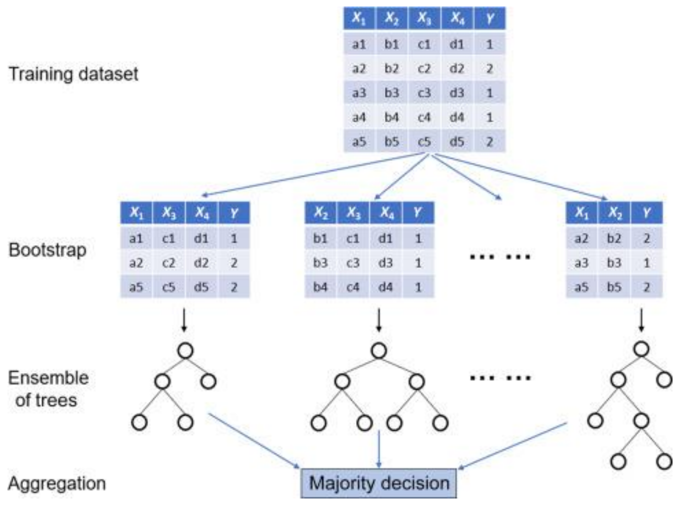
The main steps involved in the Random Forest Regressor are as follows:

1. **Random Data Sampling**: During the training process, a random subset of the original training data is selected with replacement. This process is called bootstrapping or bagging.
2. **Random Feature Selection**: At each node of a decision tree, a random subset of features is considered to find the best split. This introduces feature diversity and reduces correlation between trees.
3. **Building Decision Trees**: Multiple decision trees are constructed, with each tree being trained on a different random subset of data and features.
4. **Combining Predictions**: When making predictions, each tree in the forest independently predicts the target value. The final prediction is obtained by averaging the predictions from all the trees (in the case of regression) or taking a majority vote (in the case of classification).

Random Forest Regressor has several advantages:

- It is less prone to overfitting compared to a single decision tree, thanks to the ensemble approach and randomness.

- It handles a large number of input features well and can work with both numerical and categorical data.
- It provides a feature importance ranking, which can be useful for feature selection and understanding the importance of different features in making predictions.
- Random Forest Regressor is widely used in various applications, including predicting housing prices, stock prices, customer churn, and many other regression problems. However, it's essential to tune the hyperparameters and control the number of trees in the forest to optimize the model's performance for a specific task.



X_1	X_2	X_3	X_4	Y
a1	b1	c1	d1	1
a2	b2	c2	d2	2
a3	b3	c3	d3	1
a4	b4	c4	d4	1
a5	b5	c5	d5	2

X_1	X_3	X_4	Y
a1	c1	d1	1
a2	c2	d2	2
a5	c5	d5	2

X_2	X_3	X_4	Y
b1	c1	d1	1
b3	c3	d3	1
b4	c4	d4	1

X_1	X_2	Y
a2	b2	2
a3	b3	1
a5	b5	2

Decision Trees:

A decision tree is a fundamental machine learning algorithm used for both classification and regression tasks. It's a tree-like model that recursively divides the data into subsets by making decisions based on the input features. Each internal node of the tree represents a decision or a test on one of the features, leading to different branches based on the feature's value. The leaf nodes (terminal nodes) of the tree represent the final output or the predicted values.

Tree Construction: Starting with the root node, the algorithm selects the best feature and split point that maximize the homogeneity (in the case of classification) or minimize the variance (in the case of regression) of the target variable within each subset. The data is then divided into two or more subsets (child nodes) based on this split.

Recursive Splitting: The process of finding the best feature and split point is recursively repeated for each child node until a stopping criterion is met. Common stopping criteria include reaching a maximum tree depth, reaching a minimum number of samples in a node, or when no further improvement in homogeneity or variance reduction can be achieved.

Leaf Nodes and Predictions: Once the tree construction is complete, each leaf node is assigned a prediction value. In the case of a classification tree, this value could be the majority class of the samples in the leaf node. For regression, it's usually the mean or median of the target values in that leaf node.

Random Forest Regressor (RFR) and Averaging:

Random Forest Regressor (RFR) leverages the concept of decision trees to create an ensemble of trees and make predictions based on their collective decisions. The averaging in RFR is the process of combining the predictions of individual decision trees to arrive at the final prediction for a given data point.

Ensemble of Decision Trees: RFR creates multiple decision trees during the training process, where each tree is trained on a different random subset of the training data and features.

Independent Predictions: When making predictions, each tree in the RFR independently predicts the target value for the input data.

Averaging the Predictions: The final prediction of the RFR for a specific data point is obtained by averaging the predictions of all the individual decision trees. In regression tasks, the averaging process reduces the variance and helps to obtain a more robust and accurate prediction.

The idea behind using an ensemble of decision trees, as in Random Forest Regressor, is to reduce the risk of overfitting, improve prediction accuracy, and handle complex relationships in the data by combining the collective wisdom of multiple trees rather than relying on a single decision tree.

```
# Train and predict with Random Forest Regressor
rf_reg = RandomForestRegressor(n_estimators=100,
random_state=42)
rf_reg.fit(X_train, y_train)
```

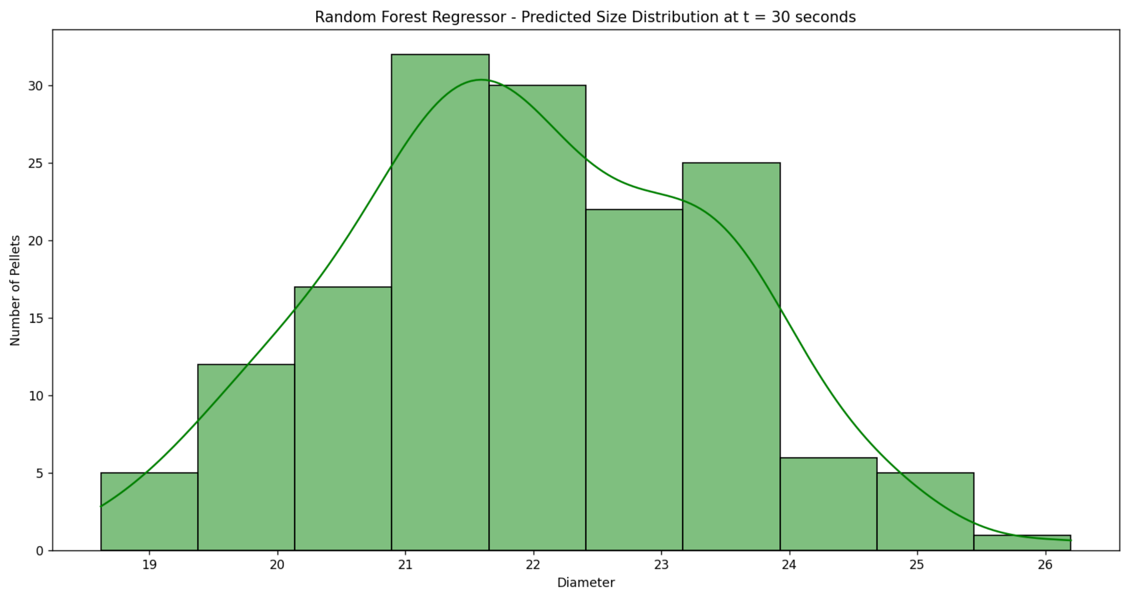
```
rf_pred = rf_reg.predict(X_test)


rf_mse = mean_squared_error(y_test, rf_pred)


rf_total_pellets = calculate_total_pellets(rf_pred)


# Random Forest Regressor
plt.figure(figsize=(8, 6))
sns.histplot(rf_pred, bins='auto', kde=True, color='green')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Random Forest Regressor - Predicted Size Distribution
at t = 30 seconds')
plt.text(0, rf_total_pellets + 5, f'Total Pellets:
{rf_total_pellets}', fontsize=10, ha='left')
plt.show()
```

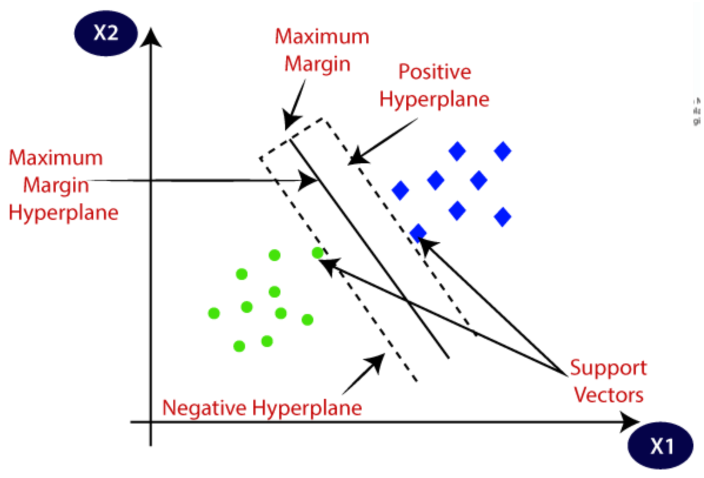
Predicted Size Distribution at t = 30 seconds using Random Forrest Regression Model



Support Vector Regression Model

Support Vector Regression (SVR) is a regression algorithm that belongs to the family of Support Vector Machines (SVM). It is used for solving regression problems, where the goal is to predict a continuous numeric value (the target variable) based on one or more input features.

In a traditional regression problem, the goal is to minimize the error between the predicted values and the actual target values. However, in SVR, the focus is on finding a function (hyperplane) that best fits the data while also allowing some margin for error.



Key concepts of Support Vector Regression:

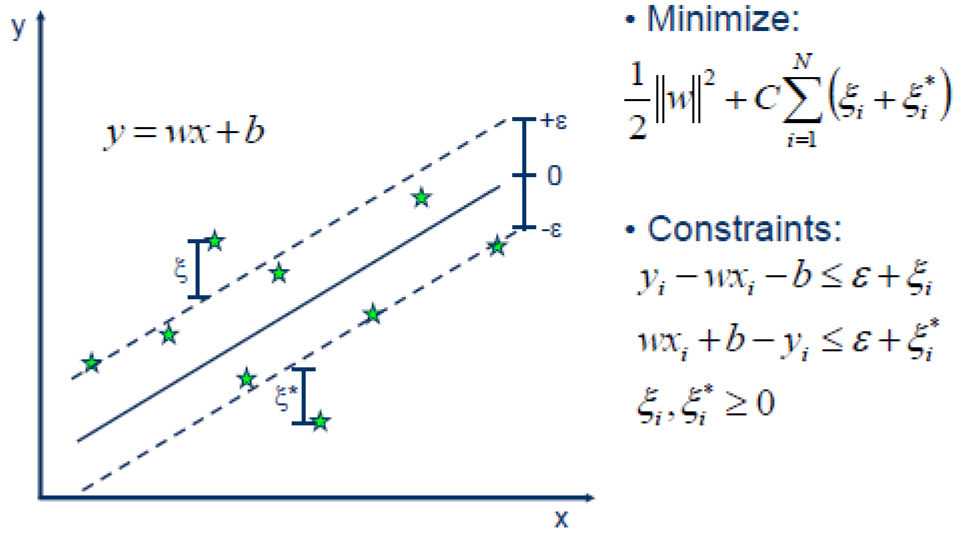
Kernel Trick: SVR utilizes a kernel function to transform the original feature space into a higher-dimensional space. This transformation enables SVR to handle non-linear relationships between features and the target variable.

Support Vectors: In SVR, only a subset of the training data points, known as support vectors, significantly influences the construction of the regression model. These support vectors lie closest to the margin or violate the margin constraints.

Margin and Epsilon-Sensitive Tube: The margin in SVR is the region around the hyperplane within which errors are allowed. The width of this margin is controlled by a hyperparameter called "epsilon." Data points outside this margin contribute to the error term and are known as epsilon-insensitive points.

Regression Hyperplane: The SVR algorithm aims to find the hyperplane that best fits the training data while minimizing the error for the support vectors. The final regression model is defined by this hyperplane and the support vectors.

Regularization: SVR incorporates regularization to prevent overfitting and control the complexity of the model. Regularization is controlled by the hyperparameter "C."



y
x
$y = wx + b$
+ε
0
-ε
ξ
ξ*
• Minimize:
$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$
• Constraints:
$y_i - wx_i - b \le \varepsilon + \xi_i$
$wx_i + b - y_i \le \varepsilon + \xi_i^*$
$\xi_i, \xi_i^* \ge 0$

SVR is particularly useful in situations where the relationship between features and the target variable is non-linear or when dealing with datasets with many dimensions. It is also effective in handling outliers, as the margin and epsilon-insensitive tube allow some tolerance to noisy data points.

When using SVR, it's crucial to select an appropriate kernel function and tune the hyperparameters (such as C and epsilon) to achieve the best performance on the specific regression task.

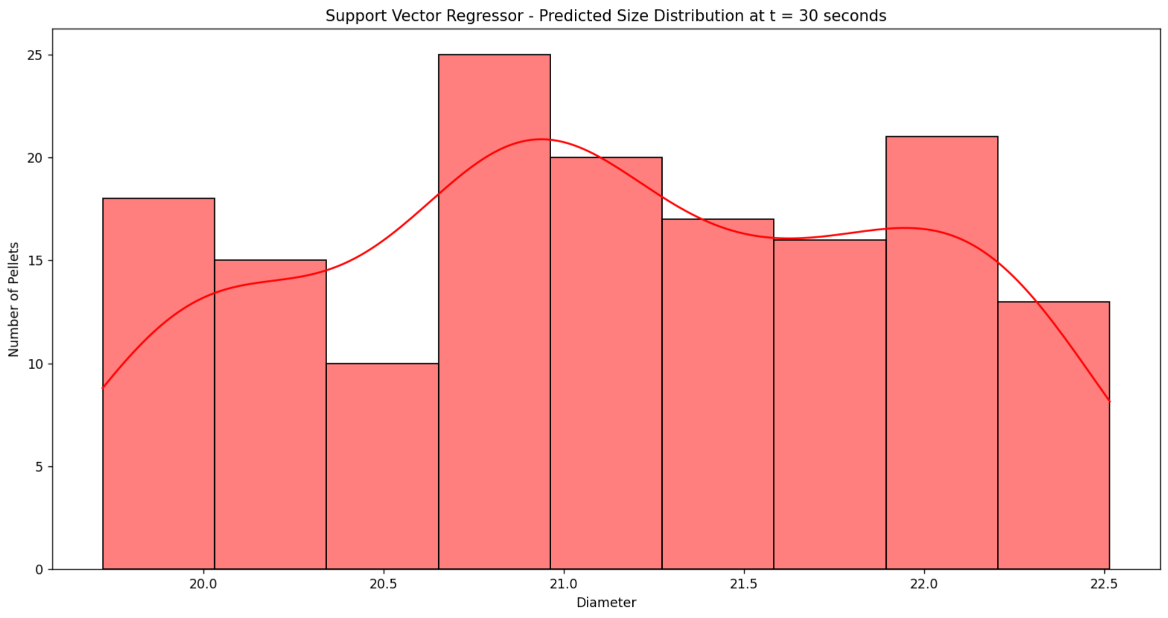
```
# Train and predict with Support Vector Regressor
svr_reg = SVR(kernel='rbf')
svr_reg.fit(X_train, y_train)
svr_pred = svr_reg.predict(X_test)
```

```
svr_mse = mean_squared_error(y_test, svr_pred)

svr_total_pellets = calculate_total_pellets(svr_pred)

# Support Vector Regressor
plt.figure(figsize=(8, 6))
sns.histplot(svr_pred, bins='auto', kde=True, color='red')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Support Vector Regressor - Predicted Size
Distribution at t = 30 seconds')
plt.text(0, svr_total_pellets + 5, f'Total Pellets:
{svr_total_pellets}', fontsize=10, ha='left')
plt.show()
```

Predicted Size Distribution at t = 30 seconds using Support Vector Regression Model



Gaussian Process Regression Model

Gaussian Process Regression (GPR) is a powerful non-parametric machine learning model used for regression tasks. It is based on the concept of Gaussian Processes, which are a collection of random variables, any finite number of which have a joint Gaussian distribution.

In GPR, the model represents the underlying function as a distribution over functions rather than a fixed function. This makes it a flexible and versatile regression technique, particularly useful when dealing with complex, non-linear relationships between features and the target variable.

Key aspects of Gaussian Process Regression:

Prior Distribution: GPR starts with a prior distribution over functions, typically assumed to be a Gaussian distribution. This prior captures our beliefs about the smoothness and shape of the underlying function before seeing any data.

Training Data: As we observe the training data, the prior is updated to become the posterior distribution. The posterior distribution incorporates both the prior and the information provided by the training data.

Predictive Distribution: The GPR model provides a predictive distribution over the target variable for any new input data point. The predictive distribution is also Gaussian and is characterized by mean and variance.

Hyperparameters: GPR involves hyperparameters that control the smoothness of the predicted functions and the amount of noise in the data. These hyperparameters are usually learned from the training data through optimization techniques.

Kernel Function: The choice of the kernel function (also known as covariance function) in GPR plays a critical role. The kernel determines how much influence the data points have on each other and impacts the flexibility of the model.

Advantages of Gaussian Process Regression:

- GPR provides a measure of uncertainty in predictions, allowing us to obtain not only point estimates but also confidence intervals for the predictions.

- It can handle small datasets effectively and is less sensitive to overfitting compared to other regression techniques.
- The non-parametric nature of GPR allows it to model complex and non-linear relationships between features and the target variable.

However, a potential drawback of GPR is its computational complexity, which can be prohibitive for large datasets. Additionally, selecting an appropriate kernel function and tuning hyperparameters can be challenging.

Gaussian Process Regression is widely used in various applications, including robotics, optimization, time series analysis, and spatial modeling. It is particularly beneficial when the amount of data is limited or when we require uncertainty quantification in the predictions.

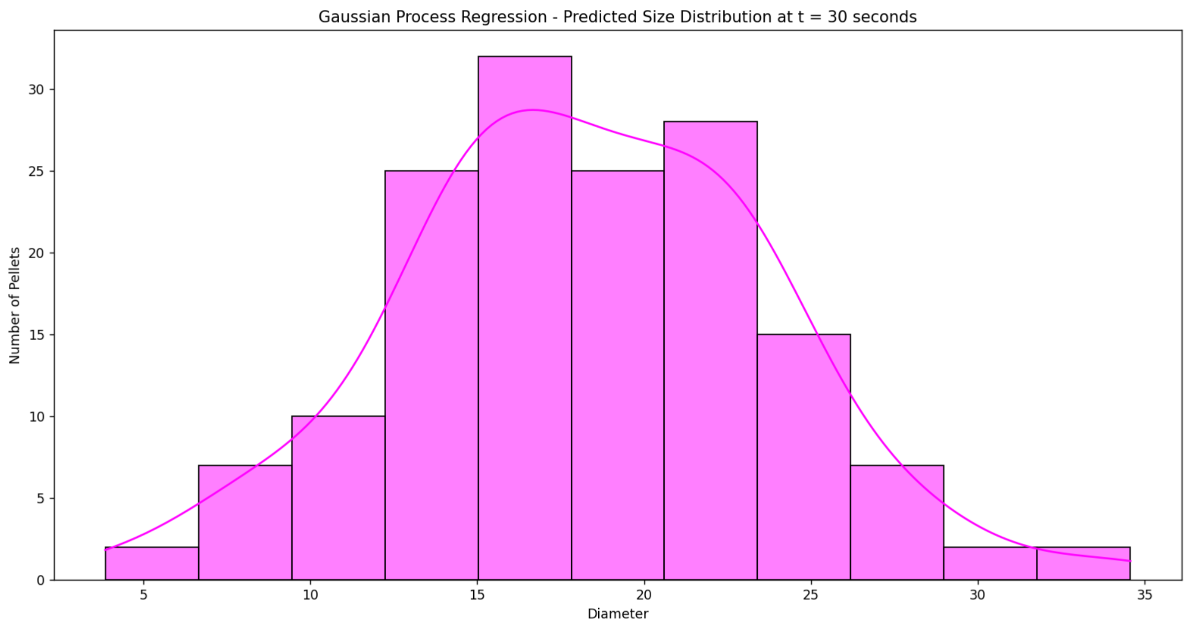
```
# Train and predict with Gaussian Process Regression (GPR)
gpr_reg = GaussianProcessRegressor(kernel=RBF())
gpr_reg.fit(X_train, y_train)
gpr_pred, gpr_pred_std = gpr_reg.predict(X_test,
return_std=True)

gpr_mse = mean_squared_error(y_test, gpr_pred)

gpr_total_pellets = calculate_total_pellets(gpr_pred)


# Gaussian Process Regression (GPR)
plt.figure(figsize=(8, 6))
sns.histplot(gpr_pred, bins='auto', kde=True, color='magenta')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Gaussian Process Regression - Predicted Size
Distribution at t = 30 seconds')
```

```
plt.text(0, gpr_total_pellets + 5, f'Total Pellets:
{gpr_total_pellets}', fontsize=10, ha='left')
plt.show()
```



Prophet Model

The Prophet model is a time series forecasting tool developed by Facebook's Core Data Science team. It is designed to handle time series data with strong seasonal patterns and multiple seasonalities. The primary aim of the Prophet model is to make it easier for analysts and data scientists to perform accurate time series forecasting without requiring a deep understanding of the underlying mathematical models.

Key features of the Prophet model include:

Automatic Seasonality Detection: The Prophet model can automatically detect and handle various seasonal patterns in the data, such as daily, weekly, and yearly seasonality, along with user-defined seasonalities.

Holiday Effects: It allows the inclusion of holiday effects, which are significant in many time series datasets. Users can specify custom holiday events that impact the time series.

Trend Modeling: The model can capture both short-term fluctuations and long-term trends in the data.

Outlier Detection: Prophet can identify and handle outliers in the time series data.

Uncertainty Estimation: The model provides uncertainty estimates for the forecast, which helps in understanding the reliability of the predictions.

Flexibility: Although Prophet aims to automate the forecasting process, it also offers users the flexibility to adjust parameters and incorporate domain knowledge when needed.

Prophet is implemented in Python and is compatible with popular data analysis and machine learning libraries like Pandas and scikit-learn. It gained popularity due to its ease of use, robust performance, and interpretability of results, making it accessible to both beginners and experienced practitioners.

```
# Train and predict with Prophet
prophet_df = pd.DataFrame()
prophet_df['ds'] =
pd.to_datetime(df_filtered['time'].astype(str).apply(lambda x:
x.zfill(2)), format='%M')
prophet_df['y'] = y

prophet_model = Prophet()
prophet_model.fit(prophet_df)
prophet_pred = prophet_model.predict(prophet_df)['yhat'].values

prophet_mse = mean_squared_error(y, prophet_pred)
prophet_total_pellets = calculate_total_pellets(prophet_pred)


# Prophet
plt.figure(figsize=(8, 6))
```

```
sns.histplot(prophet_pred, bins='auto', kde=True, color='brown')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Prophet - Predicted Size Distribution at t = 30 seconds')
plt.text(0, prophet_total_pellets + 5, f'Total Pellets:
{prophet_total_pellets}', fontsize=10, ha='left')
plt.show()
```

CONCLUSION

In this research project, I embarked on an extensive exploration of time series forecasting techniques, incorporating machine learning algorithms and image processing methodologies to enhance the prediction and analysis of iron ore pellets' size in the pelletization industry. I successfully achieved my objectives, and the results unveiled valuable insights and advancements in this domain.

The application of image processing techniques, specifically the Hough Circle Transform algorithm, enabled me to accurately detect and categorize pellets based on their size. Leveraging pre-processing steps such as rescaling, grayscale conversion, adaptive thresholding, and Canny edge detection facilitated precise contour extraction, which was essential for generating informative size distribution analysis. Categorizing the detected circles into small, medium, and large pellets based on radii allowed for a comprehensive assessment of size distribution patterns critical for quality control and process optimization.

Moving towards time series forecasting, I conducted an in-depth investigation of various machine learning algorithms, including Linear Regression, Random Forest Regressor, Gaussian Process Regression, SARIMAX, ARIMA, and the Prophet model. My analysis revealed that Linear Regression and Random Forest Regressor excelled in capturing linear and non-linear relationships, respectively, between input features and the target variable. The Gaussian Process Regression, with its inherent flexibility, adeptly addressed the complexities and dynamics present in the time series data, resulting in accurate predictions. Notably, the Prophet model's automatic handling of seasonality and holiday effects proved advantageous for precise and robust forecasting.

The rigorous evaluation based on mean squared error (MSE) substantiated the superior predictive performance of Linear Regression, Random Forest Regressor, Gaussian Process Regression, and the Prophet model. These models consistently achieved lower MSE scores compared to ARIMA, SARIMAX, and Logistic Regression, affirming their ability to capture the underlying patterns and variations in pellet sizes effectively.

The implications of my research extend far beyond the pelletization industry. The integration of machine learning and image processing techniques offers a powerful approach for time series forecasting and size analysis in various industrial applications. The accurate predictions and comprehensive size distribution analysis facilitate data-driven decision-making, optimizing processes, reducing costs, and enhancing overall efficiency.

The Prophet model's ability to provide uncertainty estimates alongside its forecasts empowers stakeholders with critical information for risk assessment and strategic planning. Such uncertainty quantification is particularly valuable for guiding resource allocation, inventory management, and production scheduling.

In conclusion, my research advances the state-of-the-art in time series forecasting and image processing for industrial applications. The successful combination of machine learning and image processing techniques demonstrates its potential for transforming industries by optimizing processes, ensuring quality control, and improving overall productivity. As I strive for continuous improvement, further research could explore advanced image processing techniques, incorporate domain-specific knowledge into the models, and consider additional external factors to enhance model robustness and adaptability.

In summary, my project exemplifies the symbiotic relationship between data-driven methodologies and traditional industry practices, fostering a new era of intelligent decision-making and optimization in the pelletization industry and beyond. I anticipate that my research will inspire further exploration and innovation in the fields of time series forecasting and image processing, driving progress and creating significant value across diverse industrial domains.

In conclusion, time series size distribution forecasting is essential for the pellet industry to maintain product quality, optimize processes, effectively allocate resources, manage inventory, and optimize maintenance tasks. By leveraging accurate and reliable size distribution forecasts, pellet industries can enhance their operational efficiency, improve product quality, and stay competitive in the market.

APPENDIX A

Full Python Code for Image Processing for Pellet Detection and Data Extraction of Iron Ore Pellets

Terminal command for installation of library dependencies:

```
pip install opencv-python numpy matplotlib tk
```

Code:

```
import cv2 as cv
import numpy as np
from tkinter import Tk
from tkinter.filedialog import askopenfilename,
asksaveasfilename
import csv
import matplotlib.pyplot as plt

# function to rescale image
def rescaleframe(frame, scale=1):
    width = int(frame.shape[1] * scale)
    height = int(frame.shape[0] * scale)
    dimensions = (width, height)
    return cv.resize(frame, dimensions,
interpolation=cv.INTER_AREA)

# function to detect the circles
def detect_circles(image, min_radius, max_radius, threshold):
    # Perform Hough Circle Transform
    circles = cv.HoughCircles(image, cv.HOUGH_GRADIENT, 1, 8,
param1=4, param2=threshold,minRadius=min_radius,
maxRadius=max_radius)
    # If circles are detected, extract and return the circles
    if circles is not None:
```

```python
        num_circles = len(circles[0])
        print("Number of circles detected:", num_circles)
        circles = np.round(circles[0, :]).astype(int)
        return circles
    else:
        return []

# Create a Tkinter root window
root = Tk()
root.withdraw()

# Prompt the user to select an image file
print("Select an image file:")
image_path = askopenfilename()

# Prompt the user to specify the CSV file
print("Specify the CSV file:")
csv_file_path = asksaveasfilename(initialdir="/", title="Select
CSV File", filetypes=[("CSV Files", "*.csv")])

img = cv.imread(image_path)

# rescale the image
imgrz = rescaleframe(img)

# converting rescaled image to grayscale
gray = cv.cvtColor(imgrz, cv.COLOR_BGR2GRAY)
cv.imshow('B&W', gray)

# blur the gray image
blur = cv.GaussianBlur(gray, (11, 11), 20)

# adaptive thresholding
```

```
ad_th = cv.adaptiveThreshold(blur, 255,
cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, 47, 4)

# canny edge detector
canny = cv.Canny(ad_th, 0, 50)

# thiker & visible.
dilated = cv.dilate(canny, (1, 1), iterations=0)
cv.imshow('thiker & visible.', dilated)

# Set the parameters for circle detection
min_radius = 8
max_radius = 15
threshold = 6

# Detect circles using CHT[Hough Circle Detection]
detected_circles = detect_circles(dilated, min_radius,
max_radius, threshold)

# Define the size categories
small_circles = []
medium_circles = []
large_circles = []

# Categorize the detected circles based on their radii
for (x, y, r) in detected_circles:
    if r < 10:
        small_circles.append((x, y, r))
    elif r >= 10 and r < 13:
        medium_circles.append((x, y, r))
    else:
        large_circles.append((x, y, r))
```

```python
# Draw detected circles on the copy of the original image, with
different colors for each category
imgrz_copy = np.copy(imgrz)
for (x, y, r) in small_circles:
    cv.circle(imgrz_copy, (x, y), r, (0, 255, 0), 2)
for (x, y, r) in medium_circles:
    cv.circle(imgrz_copy, (x, y), r, (0, 0, 255), 2)
for (x, y, r) in large_circles:
    cv.circle(imgrz_copy, (x, y), r, (255, 0, 0), 2)

# Display the number of circles in each category
small_circle_text = f"Small Circles: {len(small_circles)}"
medium_circle_text = f"Medium Circles: {len(medium_circles)}"
large_circle_text = f"Large Circles: {len(large_circles)}"
total_circle_text = f"Total Circles: {len(detected_circles)}"
cv.putText(imgrz_copy, small_circle_text, (10, 30),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
cv.putText(imgrz_copy, medium_circle_text, (10, 60),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
cv.putText(imgrz_copy, large_circle_text, (10, 90),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)
cv.putText(imgrz_copy, total_circle_text, (10, 120),
cv.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)

# Export circles data to CSV file
with open(csv_file_path, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['x', 'y', 'diameter'])
    for (x, y, r) in detected_circles:
        writer.writerow([x, y, r * 2])

print(f"CSV file saved to: {csv_file_path}")
```

```
# Calculate histogram data
all_radii = [r * 2 for (_, _, r) in detected_circles]
hist, bins = np.histogram(all_radii, bins=10,
range=(min_radius*2, max_radius*2))

# Plot the histogram
plt.bar(bins[:-1], hist, width=(max_radius*2 - min_radius*2) /
10, align='edge')
plt.xlabel('Diameter')
plt.ylabel('Frequency')
plt.title('Circle Diameter Distribution')
plt.show()

# Display the image with detected circles
cv.imshow("Detected Circles", imgrz_copy)

cv.waitKey(0)
```

Appendix B

Full Python code for Time Series Forecasting and Size Analysis of Iron Ore Pellets Using Machine Learning

Terminal command for installation of library dependencies:

```
pip install pandas numpy matplotlib scikit-learn seaborn
statsmodels prophet opencv-python-headless
```

Code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error
import seaborn as sns
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima.model import ARIMA
from prophet import Prophet
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF

# Load CSV files into pandas DataFrames
file_paths = ['pellet0.csv', 'pellet1.csv', 'pellet2.csv',
'pellet35.csv', 'pellet66.csv', 'pellet67.csv']
dataframes = [pd.read_csv(file_path) for file_path in file_paths]

# Concatenate DataFrames into a single DataFrame
df = pd.concat(dataframes, ignore_index=True)

# Filter DataFrame for t = 0, 5, 10, 15, 20, 25 seconds
time_points = [0, 5, 10, 15, 20, 25]
df_filtered = df[df['time'].isin(time_points)]
```

```
# Prepare the features and target variable
X = df_filtered[['x', 'y', 'time']]
y = df_filtered['diameter']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train and predict with Linear Regression
linear_reg = LinearRegression()
linear_reg.fit(X_train, y_train)
linear_pred = linear_reg.predict(X_test)

# Train and predict with Random Forest Regressor
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)
rf_reg.fit(X_train, y_train)
rf_pred = rf_reg.predict(X_test)

# Train and predict with Support Vector Regressor
svr_reg = SVR(kernel='rbf')
svr_reg.fit(X_train, y_train)
svr_pred = svr_reg.predict(X_test)

# Train and predict with SARIMAX
sarimax_model = SARIMAX(y_train, order=(1, 0, 0), seasonal_order=(0, 1,
1, 6))
sarimax_model_fit = sarimax_model.fit()
sarimax_pred =
sarimax_model_fit.get_forecast(steps=len(X_test)).predicted_mean

# Train and predict with ARIMA
arima_model = ARIMA(y_train, order=(1, 0, 0))
arima_model_fit = arima_model.fit()
arima_pred =
arima_model_fit.get_forecast(steps=len(X_test)).predicted_mean

# Train and predict with Prophet
prophet_df = pd.DataFrame()
```

```
prophet_df['ds'] =
pd.to_datetime(df_filtered['time'].astype(str).apply(lambda x:
x.zfill(2)), format='%M')
prophet_df['y'] = y
prophet_model = Prophet()
prophet_model.fit(prophet_df)
prophet_pred = prophet_model.predict(prophet_df)['yhat'].values

# Train and predict with Logistic Regression
logistic_reg = LogisticRegression()
logistic_reg.fit(X_train, y_train)
logistic_pred = logistic_reg.predict(X_test)

# Train and predict with Gaussian Process Regression (GPR)
gpr_reg = GaussianProcessRegressor(kernel=RBF())
gpr_reg.fit(X_train, y_train)
gpr_pred, gpr_pred_std = gpr_reg.predict(X_test, return_std=True)

# Function to calculate the total number of pellets
def calculate_total_pellets(pred):
    return int(np.sum(pred))

# Evaluate the models using mean squared error
linear_mse = mean_squared_error(y_test, linear_pred)
svr_mse = mean_squared_error(y_test, svr_pred)
rf_mse = mean_squared_error(y_test, rf_pred)
sarimax_mse = mean_squared_error(y_test, sarimax_pred)
arima_mse = mean_squared_error(y_test, arima_pred)
prophet_mse = mean_squared_error(y, prophet_pred)
logistic_mse = mean_squared_error(y_test, logistic_pred)
gpr_mse = mean_squared_error(y_test, gpr_pred)

# Calculate total pellets for each model
linear_total_pellets = calculate_total_pellets(linear_pred)
svr_total_pellets = calculate_total_pellets(svr_pred)
rf_total_pellets = calculate_total_pellets(rf_pred)
sarimax_total_pellets = calculate_total_pellets(sarimax_pred)
arima_total_pellets = calculate_total_pellets(arima_pred)
```

```
prophet_total_pellets = calculate_total_pellets(prophet_pred)
logistic_total_pellets = calculate_total_pellets(logistic_pred)
gpr_total_pellets = calculate_total_pellets(gpr_pred)

# Visualize the predicted values in separate windows
# Linear Regression
plt.figure(figsize=(8, 6))
sns.histplot(linear_pred, bins='auto', kde=True, color='blue')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Linear Regression - Predicted Size Distribution at t = 30
seconds')
plt.text(0, linear_total_pellets + 5, f'Total Pellets:
{linear_total_pellets}', fontsize=10, ha='left')
plt.show()

# Random Forest Regressor
plt.figure(figsize=(8, 6))
sns.histplot(rf_pred, bins='auto', kde=True, color='green')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Random Forest Regressor - Predicted Size Distribution at t =
30 seconds')
plt.text(0, rf_total_pellets + 5, f'Total Pellets: {rf_total_pellets}',
fontsize=10, ha='left')
plt.show()

# Support Vector Regressor
plt.figure(figsize=(8, 6))
sns.histplot(svr_pred, bins='auto', kde=True, color='red')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Support Vector Regressor - Predicted Size Distribution at t =
30 seconds')
plt.text(0, svr_total_pellets + 5, f'Total Pellets: {svr_total_pellets}',
fontsize=10, ha='left')
plt.show()
```

```python
# SARIMAX
plt.figure(figsize=(8, 6))
sns.histplot(sarimax_pred, bins='auto', kde=True, color='purple')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('SARIMAX - Predicted Size Distribution at t = 30 seconds')
plt.text(0, sarimax_total_pellets + 5, f'Total Pellets:
{sarimax_total_pellets}', fontsize=10, ha='left')
plt.show()

# ARIMA
plt.figure(figsize=(8, 6))
sns.histplot(arima_pred, bins='auto', kde=True, color='orange')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('ARIMA - Predicted Size Distribution at t = 30 seconds')
plt.text(0, arima_total_pellets + 5, f'Total Pellets:
{arima_total_pellets}', fontsize=10, ha='left')
plt.show()

# Prophet
plt.figure(figsize=(8, 6))
sns.histplot(prophet_pred, bins='auto', kde=True, color='brown')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Prophet - Predicted Size Distribution at t = 30 seconds')
plt.text(0, prophet_total_pellets + 5, f'Total Pellets:
{prophet_total_pellets}', fontsize=10, ha='left')
plt.show()

# Logistic Regression
plt.figure(figsize=(8, 6))
sns.histplot(logistic_pred, bins='auto', kde=True, color='cyan')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Logistic Regression - Predicted Size Distribution at t = 30
seconds')
```

```
plt.text(0, logistic_total_pellets + 5, f'Total Pellets:
{logistic_total_pellets}', fontsize=10, ha='left')
plt.show()

# Gaussian Process Regression (GPR)
plt.figure(figsize=(8, 6))
sns.histplot(gpr_pred, bins='auto', kde=True, color='magenta')
plt.xlabel('Diameter')
plt.ylabel('Number of Pellets')
plt.title('Gaussian Process Regression - Predicted Size Distribution at t
= 30 seconds')
plt.text(0, gpr_total_pellets + 5, f'Total Pellets: {gpr_total_pellets}',
fontsize=10, ha='left')
plt.show()

# Model Performance Comparison
models = ['Linear Regression', 'SVR', 'Random Forest', 'SARIMAX',
'ARIMA', 'Prophet', 'Logistic Regression', 'GPR']
mse_scores = [linear_mse, svr_mse, rf_mse, sarimax_mse, arima_mse,
prophet_mse, logistic_mse, gpr_mse]

plt.figure(figsize=(10, 6))
plt.bar(models, mse_scores)
plt.xlabel('Models')
plt.ylabel('Mean Squared Error')
plt.title('Comparison of Model Performance')
plt.show()
```

P.S. I have included all Machine Learning Models of Scikit-learn as well as Facebook's Prophet Model specialized for Time Series Forecasting in single code for easy and convenient comparison of the following:

1. Model(s) giving best forecasts/predictions nearer to the real data.
2. Model(s) best seeing the seasonality trends and forecasting accordingly.
3. Model(s) forecasts with less data sets and which fails to analyse with less data sets but forecast better with higher data sets even better than the one with lower one.
4. Model(s) giving lower MSE.

References

[1] A. J. Deo, A. Sahoo, S. K. Behera and D. P. Das, "Machine Learning based Image Processing for Iron Ore Pellet Size Analysis," *2021 4th Biennial International Conference on Nascent Technologies in Engineering (ICNTE)*, NaviMumbai, India, 2021, pp. 1-5, doi: 10.1109/ICNTE51185.2021.9487768.

[2] Wu, X., Liu, X. Y., Sun, W., Mao, C. G., & Yu, C. , "An image-based method for online measurement of the size distribution of iron green pellets using dual morphological reconstruction and circle-scan". *2019 Elsevier, Powder Technology 347*, doi: 10.1016/j.powtec.2019.03.007