Potato Chip Classifier

## Important Facts to be added in the report and the important applications and the motive of the project and an insight of the working of the model:

# OBJECTIVE:

The aim and the objective of the project is to assess the quality of the potato chip images, for this we have used the Pepsico R&D Potato lab Dataset using the Support Vector Regression with the Histogram of Oriented Gradients (HOG)

# HOG:

The histogram of oriented gradients (HOG) is a feature descriptor used in computer vision and image processing for the purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.

Robert K. McConnell of Wayland Research Inc. first described the concepts behind HOG without using the term HOG in a patent application in 1986.In 1994 the concepts were used by Mitsubishi Electric Research Laboratories. However, usage only became widespread in 2005 when Navneet Dalal and Bill Triggs, researchers for the French National Institute for Research in Computer Science and Automation (INRIA), presented their supplementary work on HOG descriptors at the Conference on Computer Vision and Pattern Recognition (CVPR). In this work they focused on pedestrian detection in static images, although since then they expanded their tests to include human detection in videos, as well as to a variety of common animals and vehicles in static imagery.

Histogram of oriented gradients is now one of the most powerful decriptor. The idea behind, is that object shape can be easily and well characterized by the distribution of local intensity gradients.

HOG features descriptor is that local object appearance and shape within an image can be described by the distribution of density distribution of gradients. The implementation of this descriptor can be achieved by dividing the image into small regions called a cell. Each cell compiles a histogram of gradient direction for the pixel within the cell. HOG method has four steps to extract the object.

## THE THEORY BEHIND HOG:

The essential thought behind the histogram of oriented gradients descriptor is that local object appearance and shape within an image can be described by the distribution of intensity gradients or edge directions. The image is divided into small connected regions called cells, and for the pixels within each cell, a histogram of gradient directions is compiled. The descriptor is the concatenation of these histograms. For improved accuracy, the local histograms can be contrast-normalized by calculating a measure of the intensity across a larger region of the image, called a block, and then using this value to normalize all cells within the block. This normalization results in better invariance to changes in illumination and shadowing.

The HOG descriptor has a few key advantages over other descriptors. Since it operates on local cells, it is invariant to geometric and photometric transformations, except for object orientation. Such changes would only appear in larger spatial regions. Moreover, as Dalal and Triggs discovered, coarse spatial sampling, fine orientation sampling, and strong local photometric normalization permits the individual body movement of pedestrians to be ignored so long as they maintain a roughly upright position. The HOG descriptor is thus particularly suited for human detection in images

## CALCULTION OF HOG:

* GRADIENT COMPUTATION:

The first step of calculation in many feature detectors in image pre-processing is to ensure normalized color and gamma values. However, this step can be omitted in HOG descriptor computation, as the ensuing descriptor normalization essentially achieves the same result. Image pre-processing thus provides little impact on performance. Instead, the first step of calculation is the computation of the gradient values. The most common method is to apply the 1-D centred, point discrete derivative mask in one or both of the horizontal and vertical directions.

First step is calculating the gradient values by applying 1-D centered to obtain the point of discrete derivative mask in horizontal and vertical direction as follow:

=

Here the and the values signify the derivation masks at each point(pixel) of the image. Derivation masks are nothing but the masks that are used for edge detection of any given figure or an image. If the object image is I, we can obtain x and y derivative using convolution operation:

For calculating the magnitude of the gradient formula is given by

The Gradient Orientation can be represented as:

In the third step, there is the HOG descriptor to normalize cell and histogram from entire block region to be a vector form. The last step, the block normalization is performed by using the L2 norm as follow:

After process HOG normalization, the windows descriptor is needed to collect descriptor from all the block and change into vector form.

* ORIENTATION BINNING:

The second step of calculation is creating the cell histograms. Each pixel within the cell casts a weighted vote for an orientation-based histogram bin based on the values found in the gradient computation. The cells themselves can either be rectangular or radial in shape, and the histogram channels are evenly spread over 0 to 180 degrees or 0 to 360 degrees, depending on whether the gradient is “unsigned” or “signed”. Dalal and Triggs found that unsigned gradients used in conjunction with 9 histogram channels performed best in their human detection experiments, while noting that signed gradients lead to significant improvements in the recognition of some other object classes, like cars or motorbikes. As for the vote weight, pixel contribution can either be the gradient magnitude itself, or some function of the magnitude. In tests, the gradient magnitude itself generally produces the best results. Other options for the vote weight could include the square root or square of the gradient magnitude, or some clipped version of the magnitude.

* BLOCK NORMALISATION:

In the third step, there is the HOG descriptor to normalize cell and histogram from entire block region to be a vector form. The last step, the block normalization is performed by using the L2 norm as follow:

After process HOG normalization, the windows descriptor is needed to collect descriptor from all the block and change into vector form.

* SUMMARY:

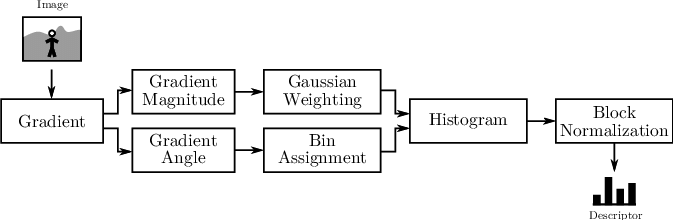


Fig 1: Working of the HOG

# MEAN SQUARED ERROR:

In statistics, the mean squared error (MSE) or mean squared deviation (MSD) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value. MSE is a risk function, corresponding to the expected value of the squared error loss. The fact that MSE is almost always strictly positive (and not zero) is because of randomness or because the estimator does not account for information that could produce a more accurate estimate. In machine learning, specifically empirical risk minimization, MSE may refer to the empirical risk (the average loss on an observed data set), as an estimate of the true MSE (the true risk: the average loss on the actual population distribution).

The MSE is a measure of the quality of an estimator. As it is derived from the square of Euclidean distance, it is always a positive value that decreases as the error approaches zero.

The MSE is the second moment (about the origin) of the error, and thus incorporates both the variance of the estimator (how widely spread the estimates are from one data sample to another) and its bias (how far off the average estimated value is from the true value). For an unbiased estimator, the MSE is the variance of the estimator. Like the variance, MSE has the same units of measurement as the square of the quantity being estimated. In an analogy to standard deviation, taking the square root of MSE yields the root-mean-square error or root-mean-square deviation (RMSE or RMSD), which has the same units as the quantity being estimated; for an unbiased estimator, the RMSE is the square root of the variance, known as the standard error.

# IMPLEMENTATION:

* OVERVIEW:

The moto of our program is to be able to distinguish between defective and non-defective potato chips. For this we have used python programming language to implement the machine learning model which we trained. We used an available pepsico dataset available on Kaggle for our implementation. The concepts involved in our program include linear Support Vector Regression and Histogram of Oriented Gradients for the implementation of the program and we had explained about them earlier. How we have implemented them in our program is by importing in built library scikit learn which is a popular machine learning library.

* CODE:

import cv2

import numpy as np

from sklearn.svm import SVR

from sklearn.metrics import mean\_squared\_error

from skimage.feature import hog

import os

def load\_images\_from\_folder(folder):

    images = []

    for filename in os.listdir(folder):

        img = cv2.imread(os.path.join(folder, filename), cv2.IMREAD\_GRAYSCALE)

        if img is not None:

            images.append(img)

    return images

def resize\_images(images, new\_size=(64, 64)):

    resized\_images = [cv2.resize(img, new\_size) for img in images]

    return resized\_images

# Update the paths for the training dataset

train\_defective\_path = r'C:\Users\Dell\Desktop\Python\Pepsico RnD Potato Lab Dataset\Train\Defective'

train\_non\_defective\_path = r'C:\Users\Dell\Desktop\Python\Pepsico RnD Potato Lab Dataset\Train\Non-Defective'

# Load training images and labels

train\_defective\_images = load\_images\_from\_folder(train\_defective\_path)

train\_non\_defective\_images = load\_images\_from\_folder(train\_non\_defective\_path)

# Resize training images

train\_defective\_images\_resized = resize\_images(train\_defective\_images)

train\_non\_defective\_images\_resized = resize\_images(train\_non\_defective\_images)

# Assign labels (1 for defective, -1 for non-defective)

train\_defective\_labels = -1 \* np.ones(len(train\_defective\_images\_resized))

train\_non\_defective\_labels = np.ones(len(train\_non\_defective\_images\_resized))

# Combine training images and labels

X\_train = train\_defective\_images\_resized + train\_non\_defective\_images\_resized

y\_train = np.concatenate([train\_defective\_labels, train\_non\_defective\_labels])

# Extract HOG features for each training image

hog\_features\_train = [hog(img, orientations=8, pixels\_per\_cell=(16, 16), cells\_per\_block=(1, 1)) for img in X\_train]

X\_train\_hog = np.array(hog\_features\_train)

# Create and train the Support Vector Regressor (SVR)

svr = SVR(kernel='linear', C=1.0)  # You can choose different kernels and adjust parameters, c is he hyper parameter that we have used

svr.fit(X\_train\_hog, y\_train)

# Update the single path for the testing dataset

test\_dataset\_path = r'C:\Users\Dell\Desktop\Python\Pepsico RnD Potato Lab Dataset\Test\Tester'

# Load testing images from the combined dataset

test\_images = load\_images\_from\_folder(test\_dataset\_path)

# Resize testing images

test\_images\_resized = resize\_images(test\_images)

# Extract HOG features for each testing image

hog\_features\_test = [hog(img, orientations=8, pixels\_per\_cell=(16, 16), cells\_per\_block=(1, 1)) for img in test\_images\_resized]

X\_test\_hog = np.array(hog\_features\_test)

# Make predictions on the test set

y\_pred = svr.predict(X\_test\_hog)

# Create dummy labels for y\_test (you should replace this with your actual labels)

y\_test = np.ones(len(y\_pred))

# Display the number of defective and non-defective pieces in the combined test set

num\_defective = np.sum(y\_pred < 0)

num\_non\_defective = np.sum(y\_pred >= 0)

print(f"Number of Defective Pieces: {num\_defective}")

print(f"Number of Non-Defective Pieces: {num\_non\_defective}")

# Evaluate the model using mean squared error

mse = mean\_squared\_error(y\_test, y\_pred)

print(f"Mean Squared Error: {mse:.4f}")

* CODE EXPLAINATION:

Here in the first segment of the code we have imported the necessary libraries for performing the desired functions.

We are using the **load\_images\_from\_folder** function for loading gray scale images from a specified folder using OpenCV. It returns the list of the loaded images.

In the **resize\_images** function to resize the list of the given images from its default size 2974 x 2974 to a newer compressed size of 64 x 64. It then returns a list of resized images.

After defining those two functions, next we specify the file paths for both defective and the non-defective datasets. In the successive lines we next resize the images from its original form to its newer compressed of the size 64 x 64.

After compressing the datasets, we assign labels to the data sets in a way such that -1 as the label for the defective pieces and 1 as the label for the non-defective datasets.

Label X\_train is a list that combines two sets of resized training images, consisting of both resized and compressed versions of the defective and non-defective images, by concatenating both the defective and the non-defective datasets we finally form the X\_train label.

Y train is same like the X\_train variable but then it serves as the corresponding target values for the regression task.

Next, we use the HOG function which we had imported from the scikit-image library. hog\_features\_train is a list comprehension that iterates through each image (img) in the X\_train dataset. For each image, the hog function is applied with specific parameters:

1. orientations=8: Number of orientation bins for the histograms.
2. pixels\_per\_cell=(16, 16): Size (in pixels) of each cell for computing the histograms.
3. cells\_per\_block=(1, 1): Number of cells in each block for normalization.
4. The result is a set of HOG features for each image.

X\_train\_hog Array:

X\_train\_hog is created by converting the list of HOG features (hog\_features\_train) into a NumPy array using np.array().

The resulting array is a two-dimensional array where each row corresponds to the HOG features of a specific training image.

An SVR model is instantiated using the SVR class from scikit-learn.

kernel='linear': Specifies the type of kernel used in the SVR. In this case, a linear kernel is chosen, but other options include 'rbf' (Radial basis function) and 'poly' (Polynomial kernel).

C=1.0: The C parameter is a regularization parameter that controls the trade-off between fitting the training data well and having a smooth decision function. It is a hyperparameter, and in this case, it is set to 1.0.

svr.fit(X\_train\_hog, y\_train):

The fit method is called on the SVR model to train it on the provided data. X\_train\_hog: The input features (HOG features) of the training dataset. y\_train: The target values corresponding to the training dataset. In the context of SVR, these are the labels indicating whether each sample is defective or non-defective.

Now after implementing the HOG and SVR, we are setting up a variable for setting up the testing data set path. Followed by that next we load the images and after that we resized them in the same dimension in which we did to our training dataset. Followed by that we next extract the HOG values from the given testing data set, after that using the SVR model we can make predictions using the HOG values which were provided as the inputs.

Now we, assign dummy labels for testing we replace our original labels with these. Finally in the end we display the number of defective and the non-defective pieces in the combined test set based on the predictions, these predictions are made on the test data set which we have given already as the input for checking its features with respect to the training data set which we had given earlier. Alas, in the end by using the **mean\_squared\_error** value we can determine the accuracy of the model. We print the MSE value in the end as well.

# OUTPUT:

Testing data set:

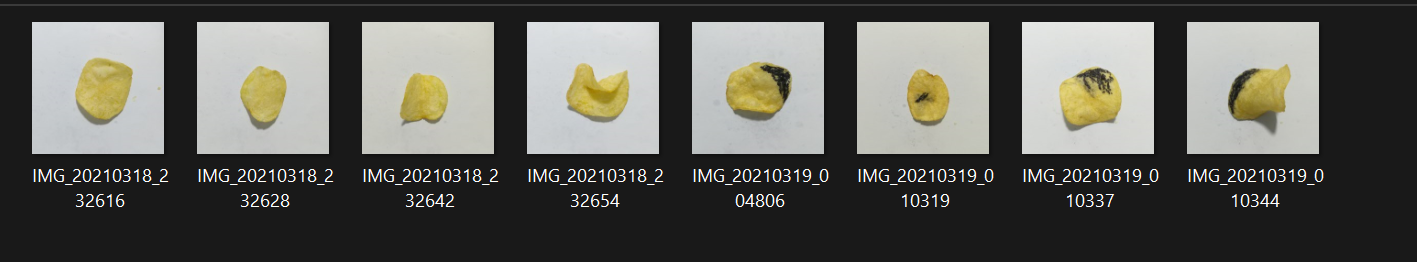


Fig2: Testing Data set

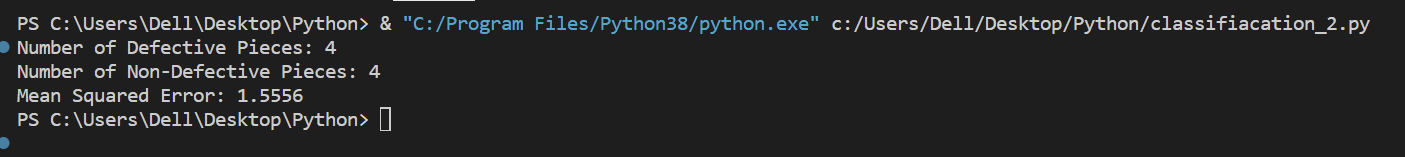


Fig3: Output obtained

Here our program was able to correctly identify both of the Defective and the Non-Defective pieces of the chips and count them correctly and at the end we have the mean squared value of the program calculated below.