**Robotic Activity detection using vector learning models**

**Synopsis:**

Our project aims to demonstrate, the integration of robotics into various domains has led to advancements in automation and efficiency, but effective human-robot interaction remains a challenge. This project presents a novel approach to robotic activity detection utilizing vector learning models with diverse machine learning (ML) algorithms. By accurately recognizing and understanding human activities, our system can enhance the autonomy and adaptability of robots in diverse settings, ranging from manufacturing and logistics to healthcare.

**SYSTEM ENVIRONMENT**

2.1 Hardware Requirements:

Processor : Intel Core i4 (10th Gen)

Ram : 4.0 GB

2.2 Software Requirements

Operating System : Windows 10

Framework : Google colab

Language : python

**2.3 About the technology:**

Python:

Python is an interpreted high-level general-purpose programming language created by Guido Van Rossum and first published in 1991. Python's design philosophy emphasizes code readability with significant whitespace. Its language structures and object-oriented approach are designed to help developers write clear and logical code for small and large projects. Python is dynamically typed and garbage

Google Colab :

Google Colab, short for Google Colaboratory, is a cloud-based, interactive computing platform provided by Google. It allows users to write and execute Python code in a collaborative and convenient environment directly through a web browser. Colab provides free access to GPU and TPU (Tensor Processing Unit) resources, enabling accelerated execution of machine learning tasks. Users can create and share Jupyter notebooks, incorporating text, code, and visualizations seamlessly. Colab integrates with Google Drive, facilitating easy storage and sharing of notebooks. Its collaborative features enable multiple users to work on the same document simultaneously, fostering collaborative research and development. Overall, Google Colab is a powerful and accessible tool for data analysis, machine learning, and collaborative coding, making it particularly valuable for researchers, students, and practitioners in the field of data science.

Scikit Learn:

Scikit-learn (Sklearn) is the most useful and powerful Python machine learning library. It provides a number of powerful tools for machine learning and statistical modeling, including classification, regression, clustering and dimensionality reduction through a Python consistent interface. Written mostly in Python, this library is built on top of NumPy, SciPy and Matplotlib. Originally called scikits.learn, it was originally developed by David Cournapeau as a Google Summer Code Project in 2007. Later, in 2010, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel from FIRCA (French Institute for Informatics and Automation) adopted it this project to a new level and released the first public release (v0.1 beta) on February 1, 2010

**EXISTING SYSTEM**

One common approach was to use feature extraction techniques to transform raw sensor data into feature vectors, which were then fed into machine learning algorithms for classification. Raw sensor data is collected from various sensors embedded in the robot or its environment. This could include data from accelerometers, gyroscopes, cameras, or other sensors depending on the specific application.

Raw sensor data often needs to be preprocessed to remove noise, filter out irrelevant information, and normalize the data. This could involve techniques such as signal filtering, normalization, and segmentation. Once the data is preprocessed, features are extracted from the sensor data to represent different aspects of the robot's activity. Feature extraction techniques could include statistical measures (mean, variance, etc.), time-domain features, frequency-domain features (using Fourier transforms), or even more complex features depending on the nature of the data and the activity being detected.

The extracted features are combined into a feature vector for each data sample. This feature vector represents the input to the machine learning model.In this phase, a machine learning algorithm is trained on a labeled dataset containing feature vectors and corresponding activity labels. Common machine learning algorithms used for this purpose include Decision Tree, Random Forests, k-Nearest Neighbors (k-NN), or even simpler classifiers like Naive Bayes.

Once the model is trained, it is tested on a separate dataset to evaluate its performance. Performance metrics such as accuracy, precision, recall, and F1-score are commonly used to assess the model's effectiveness in detecting robotic activities. Once the model has been trained and evaluated satisfactorily, it can be deployed to detect activities in real-time or near real-time on the robotic platform.

While this approach can be effective for certain types of robotic activity detection, it often requires significant domain expertise in feature engineering and may not capture complex patterns in the data as effectively as deep learning approaches. However, it can still be useful in situations where labeled training data is limited, or computational resources are constrained.

**PROPOSED SYSTEM**

Before the advent of more sophisticated deep learning techniques, robotic activity detection often relied on vector learning models. Raw sensor data is collected from various sensors embedded in the robotic system, capturing relevant information about the environment and the robot's movements. Raw sensor data undergoes preprocessing steps to remove noise, filter out irrelevant information, and normalize the data. Additionally, specific preprocessing techniques are applied to prepare the data for feature extraction.

Gray-Level Co-occurrence Matrix (GLCM) method is applied to extract texture features from the preprocessed sensor data. GLCM calculates the spatial relationship of pixel values to derive texture information, providing valuable insights into the underlying patterns present in the data. The texture features extracted using the GLCM method are combined into feature vectors, which serve as input to the machine learning model. Each feature vector represents a snapshot of the robot's activity at a given point in time.

A Support Vector Machine (SVM) classifier is chosen as the learning model for activity detection. SVM is particularly well-suited for binary classification tasks and is effective in handling high-dimensional feature spaces. The SVM model is trained on a labeled dataset containing feature vectors and corresponding activity labels. During training, the SVM algorithm learns to differentiate between different activities based on the extracted features.

GridSearchCV is employed to fine-tune the hyperparameters of the SVM model. This involves systematically searching through a grid of hyperparameter values and selecting the combination that yields the highest accuracy on a validation set. By optimizing the hyperparameters, the model's performance can be significantly improved. The performance of the SVM model is evaluated using cross-validation techniques to ensure robustness and generalizability. Various performance metrics such as accuracy, precision, recall, and F1-score are computed to assess the model's effectiveness in detecting robotic activities.

Once the SVM model has been trained and evaluated satisfactorily, it is deployed for real-time activity detection on the robotic platform. The model continuously processes incoming sensor data, classifying the robot's activities with high accuracy and enabling intelligent decision-making in dynamic environments. By leveraging the GLCM method for feature extraction, coupled with the powerful classification capabilities of the SVM algorithm and hyperparameter tuning using GridSearchCV, the proposed system aims to achieve accurate and reliable robotic activity detection in real-world scenarios.

**Advantages of the proposed system:**

The proposed system for robotic activity detection using vector learning models with Support Vector Machine (SVM) classification offers several advantages:

Effective Feature Representation:

GLCM method captures texture features from raw sensor data, providing a rich representation of the underlying patterns and characteristics of robotic activities. This allows the SVM model to make more informed decisions based on the extracted features.

Robust Classification:

SVM is known for its ability to handle high-dimensional feature spaces and effectively separate classes in non-linearly separable data. By leveraging SVM, the proposed system can achieve robust classification of various robotic activities with high accuracy.

Optimized Performance:

Hyperparameter tuning using GridSearchCV optimizes the SVM model's performance by systematically searching through different parameter combinations and selecting the ones that yield the best results. This ensures that the model achieves its maximum potential in terms of accuracy and generalization.

Generalizability:

The use of cross-validation techniques during model evaluation ensures that the SVM model generalizes well to unseen data and is not overfitting to the training set. This enhances the model's ability to accurately detect robotic activities in diverse real-world environments.

Real-Time Deployment:

Once trained and evaluated, the SVM model can be deployed for real-time activity detection on robotic platforms. Its efficient computational characteristics make it suitable for deployment in resource-constrained environments, enabling timely decision-making and control.

Interpretability:

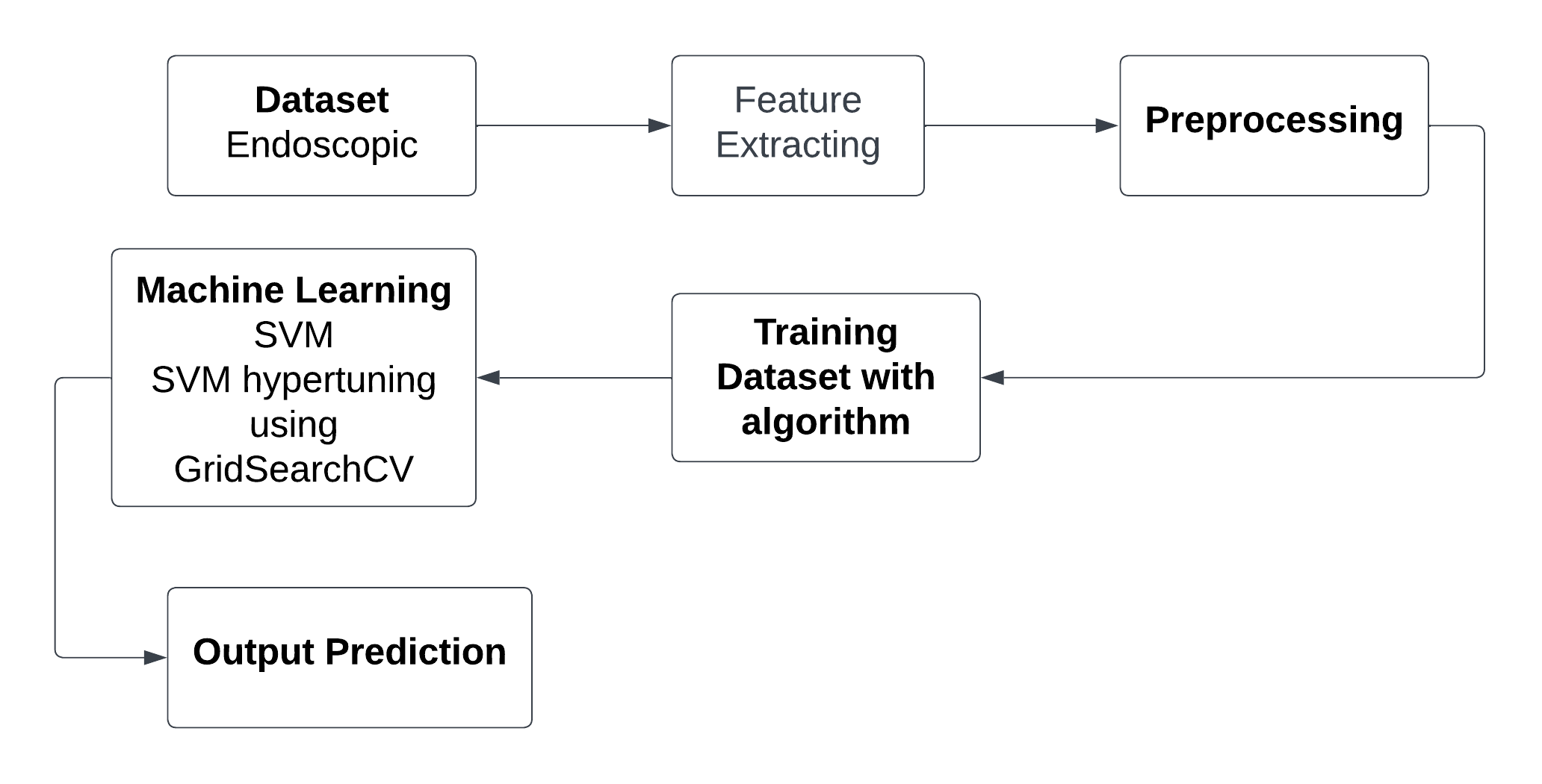
SVM models offer good interpretability, allowing users to understand the decision-making process behind activity detection. This can be crucial for debugging and refining the system, as well as for gaining insights into the underlying dynamics of robotic activities.

Scalability:

The proposed system can scale to accommodate large datasets and complex activity detection tasks. With appropriate computational resources, it can handle increasing volumes of data and adapt to evolving requirements in robotic applications.

**SYSTEM DESIGN:**

Robotic Activity Detection is designed by the below systematic diagram:



**Dataset Description:**

The dataset comprises a collection of endoscopic images captured during robotic procedures involving interventions within the intestine. These images are sourced from various medical institutions or research facilities specializing in gastrointestinal surgeries and endoscopy. The dataset is divided into two main subsets:

Training Set: This subset contains a large number of endoscopic images used for training machine learning models. These images cover a wide range of robotic activities performed within the intestine, such as tissue manipulation, suturing, cutting, and exploration.

Validation Set: A smaller subset of endoscopic images reserved for model validation and performance evaluation. These images are distinct from those in the training set and are used to assess the generalization capability of the trained models.

The images are typically high-resolution, capturing fine details of the intestinal tissues and surrounding structures. Endoscopic images may be captured in grayscale or color, depending on the imaging equipment and settings used during the procedure. The images provide a view of the interior of the intestine, showing various anatomical features, lesions, and surgical instruments. Each image in the dataset is annotated with one or more labels indicating the robotic activity or intervention taking place within the intestine. Annotations are performed by medical experts or trained annotators familiar with gastrointestinal procedures. To enhance the diversity and robustness of the training data, various data augmentation techniques may be applied.

**Pre-Processing:**

In preprocessing, compute texture features from each endoscopic image using the Gray-Level Co-occurrence Matrix (GLCM) method. GLCM calculates the spatial relationships of pixel intensities to capture texture information. Extract features such as dissimilarity, contrast, energy, homogeneity, and correlation from the GLCM matrices. Save the extracted texture features for each image as rows in a CSV file. Each column in the CSV represents a specific texture feature, and each row corresponds to a different image. Ensure proper labeling or identification of the images to maintain data integrity. Apply Min-Max scaling to normalize the feature values within a predefined range. Min-Max scaling ensures that all features contribute equally to the analysis and prevents features with larger magnitudes from dominating the learning process. Scale each feature independently, preserving the distribution of the data while standardizing the range. The preprocessed dataset consists of a CSV file containing rows representing individual endoscopic images and columns representing the normalized texture features extracted using the GLCM method. By following these preprocessing steps, the raw endoscopic images are transformed into a structured and normalized dataset suitable for training and evaluating machine learning models for robotic activity detection.

**Machine learning algorithm**

**Support Vector**

Support Vector Machine (SVM) is a powerful and versatile machine learning algorithm renowned for its efficacy in both classification and regression tasks. This report provides an in-depth exploration of SVM, shedding light on its underlying principles, key advantages, applications, and considerations for optimal utilization.

Principles:

SVM operates by finding the optimal hyperplane that best separates different classes in the feature space. This hyperplane is determined by support vectors, which are data points closest to the decision boundary. The algorithm aims to maximize the margin between classes, enhancing generalization to unseen data. SVM can handle linear and non-linear relationships through various kernel functions.

Advantages:

Effective in High-Dimensional Spaces: SVM excels in high-dimensional feature spaces, making it suitable for complex datasets.

Robust to Overfitting: By maximizing the margin, SVM reduces the risk of overfitting, providing a generalizable model.

Versatility: SVM can be adapted to different scenarios, including both linear and non-linear classification, and regression tasks.

Applications:

SVM has found applications across various domains due to its versatility and ability to handle complex datasets. Some notable applications include:

Image Classification: Recognizing objects in images.

Text Classification: Spam detection, sentiment analysis.

Bioinformatics: Protein structure prediction, gene classification.

Finance: Credit scoring, stock price prediction.

Healthcare: Disease diagnosis, outcome prediction.

Considerations:

Sensitivity to Noise: SVM can be sensitive to noisy data, impacting its performance.

Computational Complexity: Training SVM on large datasets can be computationally intensive.

Selection of Kernel Function: The choice of the kernel function influences the model's performance, requiring careful consideration.

Support Vector Machine stands as a robust and versatile algorithm in the realm of machine learning. Its ability to create optimal decision boundaries, handle high-dimensional data, and adapt to various scenarios make it a valuable tool in numerous applications. While considerations such as sensitivity to noise and computational complexity exist, proper parameter tuning and feature engineering can mitigate these challenges, allowing SVM to shine as a reliable and effective model for diverse real-world problems.

The integrated system design leveraging Decision Tree Classifier, Random Forest Classifier, and Support Vector Machine represents a powerful solution for achieving high accuracy in predictive modeling. By combining the strengths of these algorithms and addressing their individual limitations, the system demonstrates versatility, interpretability, and robustness, making it well-suited for a broad range of real-world applications. Ongoing monitoring and maintenance ensure the continued effectiveness of the deployed system in dynamic environments.

**Libraries used in the implementation:**

NumPy: NumPy is a fundamental library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions. It serves as a foundational tool for scientific computing tasks, enabling efficient and high-performance operations on numerical data.

Pandas: Pandas is a versatile data manipulation library in Python that offers data structures like DataFrames and Series, facilitating efficient data analysis and manipulation. It provides functionalities for cleaning, transforming, and exploring datasets, making it a go-to tool for handling structured data in various stages of the data science workflow.

Matplotlib: Matplotlib is a powerful plotting library for Python that allows the creation of diverse static, animated, and interactive visualizations. With a comprehensive set of functions, Matplotlib provides users with the flexibility to create various charts, plots, and graphs, making it an essential tool for data visualization and communication of findings.

Seaborn: Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for creating aesthetically pleasing and informative statistical graphics. Seaborn simplifies the process of generating complex visualizations, including heatmaps, pair plots, and violin plots, while maintaining customization options for advanced users.

Metrics (Accuracy, Classification, Confusion Matrix, ROC AUC): In the context of machine learning evaluation, metrics play a crucial role. Accuracy represents the proportion of correctly classified instances, serving as a fundamental measure of model performance. Classification metrics, such as precision, recall, and F1-score, provide insights into the model's ability to correctly identify instances of a particular class. The confusion matrix presents a comprehensive summary of true positive, true negative, false positive, and false negative predictions. Lastly, the ROC AUC (Receiver Operating Characteristic - Area Under the Curve) is a performance metric for binary classification models, illustrating the trade-off between sensitivity and specificity across different thresholds, providing a holistic view of the model's discriminatory power. These metrics collectively aid in assessing and optimizing the performance of machine learning models.

**CODING:**

%matplotlib inline

import math

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

import cv2

import numpy as np

import os

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import accuracy\_score

from sklearn.svm import SVC

from tqdm import tqdm

from skimage import measure

from skimage import io

from skimage.color import rgb2gray

from skimage.feature import graycomatrix, graycomatrix, graycoprops

from skimage import img\_as\_ubyte

path='/content/drive/MyDrive/train'

imgList = os.listdir('/content/drive/MyDrive/train')

imgPath = []

for path in imgList:

pathImg = f'/content/drive/MyDrive/train/{path}'

imgPath.append(pathImg)

imgPath.sort()

ax = []

ay = []

az = []

aa = []

ab = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0], levels=256)

ax.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ay.append(graycoprops(glcm, 'correlation')[0, 0])

az.append(graycoprops(glcm, 'homogeneity')[0, 0])

aa.append(graycoprops(glcm, 'contrast')[0, 0])

ab.append(graycoprops(glcm, 'energy')[0, 0])

# 4.18879, 2.35619, 3.92699, 1.74533, 3.49066, 5.23599, 5.93412

bx = []

by = []

bz = []

ba = []

bb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0.785398], levels=256)

bx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

by.append(graycoprops(glcm, 'correlation')[0, 0])

bz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ba.append(graycoprops(glcm, 'contrast')[0, 0])

bb.append(graycoprops(glcm, 'energy')[0, 0])

cx = []

cy = []

cz = []

ca = []

cb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[1.0472], levels=256)

cx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

cy.append(graycoprops(glcm, 'correlation')[0, 0])

cz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ca.append(graycoprops(glcm, 'contrast')[0, 0])

cb.append(graycoprops(glcm, 'energy')[0, 0])

dx = []

dy = []

dz = []

da = []

db = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[2.0944], levels=256)

dx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

dy.append(graycoprops(glcm, 'correlation')[0, 0])

dz.append(graycoprops(glcm, 'homogeneity')[0, 0])

da.append(graycoprops(glcm, 'contrast')[0, 0])

db.append(graycoprops(glcm, 'energy')[0, 0])

ex = []

ey = []

ez = []

ea = []

eb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[3.14159], levels=256)

ex.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ey.append(graycoprops(glcm, 'correlation')[0, 0])

ez.append(graycoprops(glcm, 'homogeneity')[0, 0])

ea.append(graycoprops(glcm, 'contrast')[0, 0])

eb.append(graycoprops(glcm, 'energy')[0, 0])

fx = []

fy = []

fz = []

fa = []

fb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[4.18879], levels=256)

fx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

fy.append(graycoprops(glcm, 'correlation')[0, 0])

fz.append(graycoprops(glcm, 'homogeneity')[0, 0])

fa.append(graycoprops(glcm, 'contrast')[0, 0])

fb.append(graycoprops(glcm, 'energy')[0, 0])

imgName = os.listdir('/content/drive/MyDrive/train')

imgName.sort()

endo = pd.DataFrame({

'label': 0,

'image\_name': imgName,

'ax': ax, 'ay': ay, 'az': az, 'aa': aa, 'ab': ab,

'bx': bx, 'by': by, 'bz': bz, 'ba': ba, 'bb': bb,

'cx': cx, 'cy': cy, 'cz': cz, 'ca': ca, 'cb': cb,

'dx': dx, 'dy': dy, 'dz': dz, 'da': da, 'db': db,

'ex': ex, 'ey': ey, 'ez': ez, 'ea': ea, 'eb': eb,

'fx': fx, 'fy': fy, 'fz': fz, 'fa': fa, 'fb': fb

})

endo.rename(columns = {0: "image\_name", 1: "label"}, inplace = True)

endo = endo.sort\_values('image\_name')

endo.to\_csv('endo.csv', index=False)

endo.head()

path1='/content/drive/MyDrive/valid'

imgList1= os.listdir('/content/drive/MyDrive/valid')

imgPath1 = []

for path1 in imgList1:

pathImg1 = f'/content/drive/MyDrive/valid/{path1}'

imgPath1.append(pathImg1)

imgPath1.sort()

ax = []

ay = []

az = []

aa = []

ab = []

for patch in imgPath1:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0], levels=256)

ax.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ay.append(graycoprops(glcm, 'correlation')[0, 0])

az.append(graycoprops(glcm, 'homogeneity')[0, 0])

aa.append(graycoprops(glcm, 'contrast')[0, 0])

ab.append(graycoprops(glcm, 'energy')[0, 0])

# 4.18879, 2.35619, 3.92699, 1.74533, 3.49066, 5.23599, 5.93412

bx = []

by = []

bz = []

ba = []

bb = []

for patch in imgPath1:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0.785398], levels=256)

bx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

by.append(graycoprops(glcm, 'correlation')[0, 0])

bz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ba.append(graycoprops(glcm, 'contrast')[0, 0])

bb.append(graycoprops(glcm, 'energy')[0, 0])

cx = []

cy = []

cz = []

ca = []

cb = []

for patch in imgPath1:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[1.0472], levels=256)

cx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

cy.append(graycoprops(glcm, 'correlation')[0, 0])

cz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ca.append(graycoprops(glcm, 'contrast')[0, 0])

cb.append(graycoprops(glcm, 'energy')[0, 0])

dx = []

dy = []

dz = []

da = []

db = []

for patch in imgPath1:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[2.0944], levels=256)

dx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

dy.append(graycoprops(glcm, 'correlation')[0, 0])

dz.append(graycoprops(glcm, 'homogeneity')[0, 0])

da.append(graycoprops(glcm, 'contrast')[0, 0])

db.append(graycoprops(glcm, 'energy')[0, 0])

ex = []

ey = []

ez = []

ea = []

eb = []

for patch in imgPath1:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[3.14159], levels=256)

ex.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ey.append(graycoprops(glcm, 'correlation')[0, 0])

ez.append(graycoprops(glcm, 'homogeneity')[0, 0])

ea.append(graycoprops(glcm, 'contrast')[0, 0])

eb.append(graycoprops(glcm, 'energy')[0, 0])

fx = []

fy = []

fz = []

fa = []

fb = []

for patch in imgPath1:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[4.18879], levels=256)

fx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

fy.append(graycoprops(glcm, 'correlation')[0, 0])

fz.append(graycoprops(glcm, 'homogeneity')[0, 0])

fa.append(graycoprops(glcm, 'contrast')[0, 0])

fb.append(graycoprops(glcm, 'energy')[0, 0])

imgName1 = os.listdir('/content/drive/MyDrive/valid')

imgName1.sort()

endo1 = pd.DataFrame({

'label': 1,

'image\_name': imgName1,

'ax': ax, 'ay': ay, 'az': az, 'aa': aa, 'ab': ab,

'bx': bx, 'by': by, 'bz': bz, 'ba': ba, 'bb': bb,

'cx': cx, 'cy': cy, 'cz': cz, 'ca': ca, 'cb': cb,

'dx': dx, 'dy': dy, 'dz': dz, 'da': da, 'db': db,

'ex': ex, 'ey': ey, 'ez': ez, 'ea': ea, 'eb': eb,

'fx': fx, 'fy': fy, 'fz': fz, 'fa': fa, 'fb': fb

})

endo1.rename(columns = {0: "image\_name", 1: "label"}, inplace = True)

endo1.head()

endo1 = endo1.sort\_values('image\_name')

endo1.to\_csv('endo1.csv', index=False)

endo1.head()

df = pd.DataFrame(endo)

df = pd.concat([df, endo1], ignore\_index=True)

df.rename(columns = {0: "image\_names", 1: "label"}, inplace = True)

df.to\_csv('features.csv', index=False)

df.head(1300)

df.shape

array=df.values

x\_feature=array[:,2:]

y\_label=array[:,2].astype('int')

print(x\_feature.shape)

print(y\_label.shape)

X\_train,X\_test,Y\_train,Y\_test=train\_test\_split(x\_feature,y\_label,test\_size=0.10,random\_state=7)

# Normalise the data after splitting to avoid information leak between train and test set.

scaler\_norm = MinMaxScaler()

X\_train = scaler\_norm.fit\_transform(X\_train)

X\_test = scaler\_norm.fit\_transform(X\_test)

svm\_classifier = SVC(random\_state=0)

svm\_classifier.fit(X\_train, Y\_train)

y\_pred = svm\_classifier.predict(X\_test)

accuracy\_svm = accuracy\_score(Y\_test, y\_pred)

print('Model accuracy score of svm:', accuracy\_svm)

cm\_svm = confusion\_matrix(Y\_test, y\_pred)

print('Confusion matrix of svm\n\n', cm\_svm)

# Plot Confusion Matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm\_svm, annot=True, fmt='d', cmap='cividis', cbar=False)

plt.title('Confusion Matrix of svm')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

clr\_svm = print(classification\_report(Y\_test, y\_pred, zero\_division=0))

import matplotlib.pyplot as plt

import seaborn as sns

class\_report = classification\_report(Y\_test, y\_pred, output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='YlOrRd')

plt.title('Classification Report Heatmap')

plt.show()

# SVM hypertuning using GridSeachCV

model\_SVC=SVC()

kfold=KFold(n\_splits=10)

param\_grid = {'C': [1, 10, 100, 500, 1000],

'gamma': [1, 0.1, 0.01, 0.001, 0.0001],

'kernel': ['rbf']}

grid=GridSearchCV(estimator=model\_SVC,param\_grid=param\_grid,scoring='accuracy',cv=kfold,verbose=3)

grid\_result=grid.fit(X\_train,Y\_train)

print("Best: %f using %s" % (grid\_result.best\_score\_,grid\_result.best\_params\_))

from sklearn.metrics import confusion\_matrix

# Predict on the test set using the best model found

y\_pred = grid\_result.best\_estimator\_.predict(X\_test)

# Calculate confusion matrix

cm = confusion\_matrix(Y\_test, y\_pred)

print('Confusion matrix\n\n', cm)

import seaborn as sns

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='cividis', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

# Generate a classification report

clr = print(classification\_report(Y\_test, y\_pred, zero\_division=0))

import matplotlib.pyplot as plt

import seaborn as sns

class\_report = classification\_report(Y\_test, y\_pred, output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='RdPu')

plt.title('Classification Report Heatmap')

plt.show()

**Framework Code:**

import tkinter as tk

from tkinter import ttk

%matplotlib inline

import math

import matplotlib.pyplot as plt

import os

from skimage import measure

from skimage.color import rgb2gray

from skimage.util import img\_as\_ubyte

from skimage import io

from skimage.feature import graycomatrix, graycoprops

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

from tqdm import tqdm

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.metrics import roc\_auc\_score, roc\_curve, auc, precision\_recall\_fscore\_support

import seaborn as sns

import matplotlib.pyplot as plt

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from PIL import Image, ImageTk

from sklearn.model\_selection import train\_test\_split

import numpy as np

import pandas as pd

# Load your dataset here

imgList = os.listdir('Endo/train')

imgPath = []

for path in imgList:

pathImg = f'Endo/train/{path}'

imgPath.append(pathImg)

imgPath.sort()

ax = []

ay = []

az = []

aa = []

ab = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0], levels=256)

ax.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ay.append(graycoprops(glcm, 'correlation')[0, 0])

az.append(graycoprops(glcm, 'homogeneity')[0, 0])

aa.append(graycoprops(glcm, 'contrast')[0, 0])

ab.append(graycoprops(glcm, 'energy')[0, 0])

bx = []

by = []

bz = []

ba = []

bb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0.785398], levels=256)

bx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

by.append(graycoprops(glcm, 'correlation')[0, 0])

bz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ba.append(graycoprops(glcm, 'contrast')[0, 0])

bb.append(graycoprops(glcm, 'energy')[0, 0])

cx = []

cy = []

cz = []

ca = []

cb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[1.0472], levels=256)

cx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

cy.append(graycoprops(glcm, 'correlation')[0, 0])

cz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ca.append(graycoprops(glcm, 'contrast')[0, 0])

cb.append(graycoprops(glcm, 'energy')[0, 0])

dx = []

dy = []

dz = []

da = []

db = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[2.0944], levels=256)

dx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

dy.append(graycoprops(glcm, 'correlation')[0, 0])

dz.append(graycoprops(glcm, 'homogeneity')[0, 0])

da.append(graycoprops(glcm, 'contrast')[0, 0])

db.append(graycoprops(glcm, 'energy')[0, 0])

ex = []

ey = []

ez = []

ea = []

eb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[3.14159], levels=256)

ex.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ey.append(graycoprops(glcm, 'correlation')[0, 0])

ez.append(graycoprops(glcm, 'homogeneity')[0, 0])

ea.append(graycoprops(glcm, 'contrast')[0, 0])

eb.append(graycoprops(glcm, 'energy')[0, 0])

fx = []

fy = []

fz = []

fa = []

fb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[4.18879], levels=256)

fx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

fy.append(graycoprops(glcm, 'correlation')[0, 0])

fz.append(graycoprops(glcm, 'homogeneity')[0, 0])

fa.append(graycoprops(glcm, 'contrast')[0, 0])

fb.append(graycoprops(glcm, 'energy')[0, 0])

imgName = os.listdir('Endo/train')

imgName.sort()

endo = pd.DataFrame({

'label': 0,

'image\_name': imgName,

'ax': ax, 'ay': ay, 'az': az, 'aa': aa, 'ab': ab,

'bx': bx, 'by': by, 'bz': bz, 'ba': ba, 'bb': bb,

'cx': cx, 'cy': cy, 'cz': cz, 'ca': ca, 'cb': cb,

'dx': dx, 'dy': dy, 'dz': dz, 'da': da, 'db': db,

'ex': ex, 'ey': ey, 'ez': ez, 'ea': ea, 'eb': eb,

'fx': fx, 'fy': fy, 'fz': fz, 'fa': fa, 'fb': fb

})

endo.rename(columns = {0: "image\_name", 1: "label"}, inplace = True)

endo = endo.sort\_values('image\_name')

endo.to\_csv('endo.csv', index=False)

endo.head()

imgList = os.listdir('Endo/valid')

imgPath = []

for path in imgList:

pathImg = f'Endo/valid/{path}'

imgPath.append(pathImg)

imgPath.sort()

ax = []

ay = []

az = []

aa = []

ab = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0], levels=256)

ax.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ay.append(graycoprops(glcm, 'correlation')[0, 0])

az.append(graycoprops(glcm, 'homogeneity')[0, 0])

aa.append(graycoprops(glcm, 'contrast')[0, 0])

ab.append(graycoprops(glcm, 'energy')[0, 0])

bx = []

by = []

bz = []

ba = []

bb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[0.785398], levels=256)

bx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

by.append(graycoprops(glcm, 'correlation')[0, 0])

bz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ba.append(graycoprops(glcm, 'contrast')[0, 0])

bb.append(graycoprops(glcm, 'energy')[0, 0])

cx = []

cy = []

cz = []

ca = []

cb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[1.0472], levels=256)

cx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

cy.append(graycoprops(glcm, 'correlation')[0, 0])

cz.append(graycoprops(glcm, 'homogeneity')[0, 0])

ca.append(graycoprops(glcm, 'contrast')[0, 0])

cb.append(graycoprops(glcm, 'energy')[0, 0])

dx = []

dy = []

dz = []

da = []

db = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[2.0944], levels=256)

dx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

dy.append(graycoprops(glcm, 'correlation')[0, 0])

dz.append(graycoprops(glcm, 'homogeneity')[0, 0])

da.append(graycoprops(glcm, 'contrast')[0, 0])

db.append(graycoprops(glcm, 'energy')[0, 0])

ex = []

ey = []

ez = []

ea = []

eb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[3.14159], levels=256)

ex.append(graycoprops(glcm, 'dissimilarity')[0, 0])

ey.append(graycoprops(glcm, 'correlation')[0, 0])

ez.append(graycoprops(glcm, 'homogeneity')[0, 0])

ea.append(graycoprops(glcm, 'contrast')[0, 0])

eb.append(graycoprops(glcm, 'energy')[0, 0])

fx = []

fy = []

fz = []

fa = []

fb = []

for patch in imgPath:

image\_rgb = io.imread(patch)

# Check if the image is grayscale

if len(image\_rgb.shape) == 2: # Grayscale image

image = img\_as\_ubyte(image\_rgb) # No need to convert to grayscale

else: # RGB image

image = img\_as\_ubyte(rgb2gray(image\_rgb)) # Convert to grayscale

glcm = graycomatrix(image, distances=[4], angles=[4.18879], levels=256)

fx.append(graycoprops(glcm, 'dissimilarity')[0, 0])

fy.append(graycoprops(glcm, 'correlation')[0, 0])

fz.append(graycoprops(glcm, 'homogeneity')[0, 0])

fa.append(graycoprops(glcm, 'contrast')[0, 0])

fb.append(graycoprops(glcm, 'energy')[0, 0])

imgName1 = os.listdir('Endo/valid')

imgName1.sort()

endo1 = pd.DataFrame({

'label': 1,

'image\_name': imgName1,

'ax': ax, 'ay': ay, 'az': az, 'aa': aa, 'ab': ab,

'bx': bx, 'by': by, 'bz': bz, 'ba': ba, 'bb': bb,

'cx': cx, 'cy': cy, 'cz': cz, 'ca': ca, 'cb': cb,

'dx': dx, 'dy': dy, 'dz': dz, 'da': da, 'db': db,

'ex': ex, 'ey': ey, 'ez': ez, 'ea': ea, 'eb': eb,

'fx': fx, 'fy': fy, 'fz': fz, 'fa': fa, 'fb': fb

})

endo1.rename(columns = {0: "image\_name", 1: "label"}, inplace = True)

endo1 = endo1.sort\_values('image\_name')

endo1.to\_csv('endo1.csv', index=False)

endo1.head()

df = pd.DataFrame(endo)

df = pd.concat([df, endo1], ignore\_index=True)

df.rename(columns = {0: "image\_names", 1: "label"}, inplace = True)

df.to\_csv('features.csv', index=False)

array=df.values

x\_feature=array[:,2:]

y\_label=array[:,2].astype('int')

print(x\_feature.shape)

print(y\_label.shape)

X\_train,X\_test,Y\_train,Y\_test=train\_test\_split(x\_feature,y\_label,test\_size=0.10,random\_state=7)

scaler\_norm = MinMaxScaler()

X\_train = scaler\_norm.fit\_transform(X\_train)

X\_test = scaler\_norm.fit\_transform(X\_test)

# Initialize classifiers

svm\_classifier = SVC(random\_state=0)

model\_SVC=SVC()

kfold=KFold(n\_splits=10)

param\_grid = {'C': [1, 10, 100, 500, 1000],

'gamma': [1, 0.1, 0.01, 0.001, 0.0001],

'kernel': ['rbf']}

grid=GridSearchCV(estimator=model\_SVC,param\_grid=param\_grid,scoring='accuracy',cv=kfold,verbose=3)

# Tkinter GUI

root = tk.Tk()

root.title("Classifier Metrics")

root.geometry("400x400")

# Load background image

background\_image = Image.open("b1.jpg") # Replace with your image file

background\_photo = ImageTk.PhotoImage(background\_image)

background\_label = tk.Label(root, image=background\_photo)

background\_label.place(relwidth=1, relheight=1)

# Project label

project\_label = tk.Label(root, text="Robotic Activity detection using Vector learning models", font=("Helvetica", 12), bg="white")

project\_label.pack(pady=10)

# Labels for dataset information

r\_dataset\_label = tk.Label(root, text="Dataset: Endoscopic", font=("Helvetica", 11),foreground="blue",width=20)

r\_dataset\_label.pack(pady=10, padx=10)

# Training Data Label

r\_train\_data\_label = tk.Label(root, text="Training Data: 90%", font=("Helvetica", 11),foreground="blue",width=20)

r\_train\_data\_label.pack(pady=10, padx=10)

# Testing Data Label

r\_test\_data\_label = tk.Label(root, text="Testing Data: 10%", font=("Helvetica", 11), foreground="blue",width=20)

r\_test\_data\_label.pack(pady=10, padx=10)

# Function to train classifiers

def train\_svm\_classifier():

global svm\_classifier, X\_train, y\_train

svm\_classifier.fit(X\_train, Y\_train)

print("SVM Classifier trained successfully.")

def train\_svc\_classifier():

global grid\_result, grid\_result\_best\_estimator\_, grid\_result\_best\_score\_, grid\_result\_best\_params\_, X\_train, Y\_train

grid\_result=grid.fit(X\_train,Y\_train)

grid\_result\_best\_estimator\_ = grid\_result.best\_estimator\_

grid\_result\_best\_score\_ = grid\_result.best\_score\_

grid\_result\_best\_params\_ = grid\_result.best\_params\_

print("SVC Classifier trained successfully.")

# Function to calculate metrics and show charts for SVM

def show\_svm\_metrics():

global svm\_classifier, X\_test, Y\_test

# Predict the Test set results

y\_pred = svm\_classifier.predict(X\_test)

# Confusion Matrix

cm\_svm = confusion\_matrix(Y\_test, y\_pred)

print('Confusion matrix of svm\n\n', cm\_svm)

# Plot Confusion Matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm\_svm, annot=True, fmt='d', cmap='cividis', cbar=False)

plt.title('Confusion Matrix of svm')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

def show\_report\_svm():

# Predict the Test set results

y\_pred = svm\_classifier.predict(X\_test)

# Classification Report

clr\_svm = print(classification\_report(Y\_test, y\_pred, zero\_division=0))

# Plot Classification Report

class\_report = classification\_report(Y\_test, y\_pred, output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='RdPu')

plt.title('Classification Report Heatmap of svm')

plt.show()

def calculate\_accuracy\_svm():

global svm\_classifier, X\_test, Y\_test

# Predict the Test set results

y\_pred = svm\_classifier.predict(X\_test)

# Accuracy

accuracy\_svm = accuracy\_score(Y\_test, y\_pred)

print('Model accuracy score of svm:', accuracy\_svm)

# Plot Accuracy

plt.figure(figsize=(6, 4))

plt.bar(["Accuracy"], [accuracy\_svm], color='blue')

plt.title('Model Accuracy of svm')

plt.ylabel('Accuracy')

plt.show()

# Function to calculate metrics and show charts for DTC

def show\_svc\_metrics():

global grid\_result\_best\_estimator\_, X\_test, Y\_test

# Predict the Test set results

y\_pred = grid\_result\_best\_estimator\_.predict(X\_test)

# Confusion Matrix

cm\_svc = confusion\_matrix(Y\_test, y\_pred)

print('Confusion matrix of svc\n\n', cm\_svc)

# Plot Confusion Matrix

plt.figure(figsize=(8, 6))

sns.heatmap(cm\_svc, annot=True, fmt='d', cmap='cividis', cbar=False)

plt.title('Confusion Matrix of dtc')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

def show\_report\_svc():

# Predict the Test set results

y\_pred = grid\_result\_best\_estimator\_.predict(X\_test)

# Classification Report

clr\_svc = print(classification\_report(Y\_test, y\_pred, zero\_division=0))

# Plot Classification Report

class\_report = classification\_report(Y\_test, y\_pred, output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='RdPu')

plt.title('Classification Report Heatmap of svc')

plt.show()

def calculate\_accuracy\_svc():

global grid\_result\_best\_estimator\_, grid\_result\_best\_score\_,grid\_result\_best\_params\_, X\_test, Y\_test

# Predict the Test set results

# Accuracy

print("Best: %f using %s" % (grid\_result\_best\_score\_,grid\_result\_best\_params\_))

# Plot Accuracy

plt.figure(figsize=(6, 4))

plt.bar(["Accuracy"], [accuracy\_svc], color='blue')

plt.title('Model Accuracy of svc')

plt.ylabel('Accuracy')

plt.show()

# SVM Frame

svm\_frame = tk.Frame(root)

svm\_frame.pack(side=tk.TOP, pady=10)

# SVM Train Button

svm\_train\_button = tk.Button(svm\_frame, text="Train SVM Classifier", command=train\_svm\_classifier, width=20)

svm\_train\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVM Metrics Button

svm\_metrics\_button = tk.Button(svm\_frame, text="SVM Accuracy", command=calculate\_accuracy\_svm, width=20)

svm\_metrics\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVM matrix Button

svm\_metrics\_button = tk.Button(svm\_frame, text="SVM Confusion Matrix", command=show\_svm\_metrics, width=20)

svm\_metrics\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVM report Button

svm\_report\_button = tk.Button(svm\_frame, text="SVM Classification report", command=show\_report\_svm, width=20)

svm\_report\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVC Frame

svc\_frame = tk.Frame(root)

svc\_frame.pack(side=tk.TOP, pady=10)

# SVC Train Button

svc\_train\_button = tk.Button(svc\_frame, text="Train SVC Classifier", command=train\_svc\_classifier, width=20)

svc\_train\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVC Metrics Button

svc\_metrics\_button = tk.Button(svc\_frame, text="SVC Accuracy", command=calculate\_accuracy\_svc, width=20)

svc\_metrics\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVC Matrix Button

svc\_matrix\_button = tk.Button(svc\_frame, text="SVC Confusion Matrix", command=show\_svc\_metrics, width=20)

svc\_matrix\_button.pack(side=tk.LEFT, padx=5, pady=5)

# SVC Matrix Button

svc\_report\_button = tk.Button(svc\_frame, text="SVC Classification report", command=show\_report\_svc, width=20)

svc\_report\_button.pack(side=tk.LEFT, padx=5, pady=5)

# Run the Tkinter event loop

root.mainloop()

**RESULTS AND DISCUSSION:**

**Dataset:**

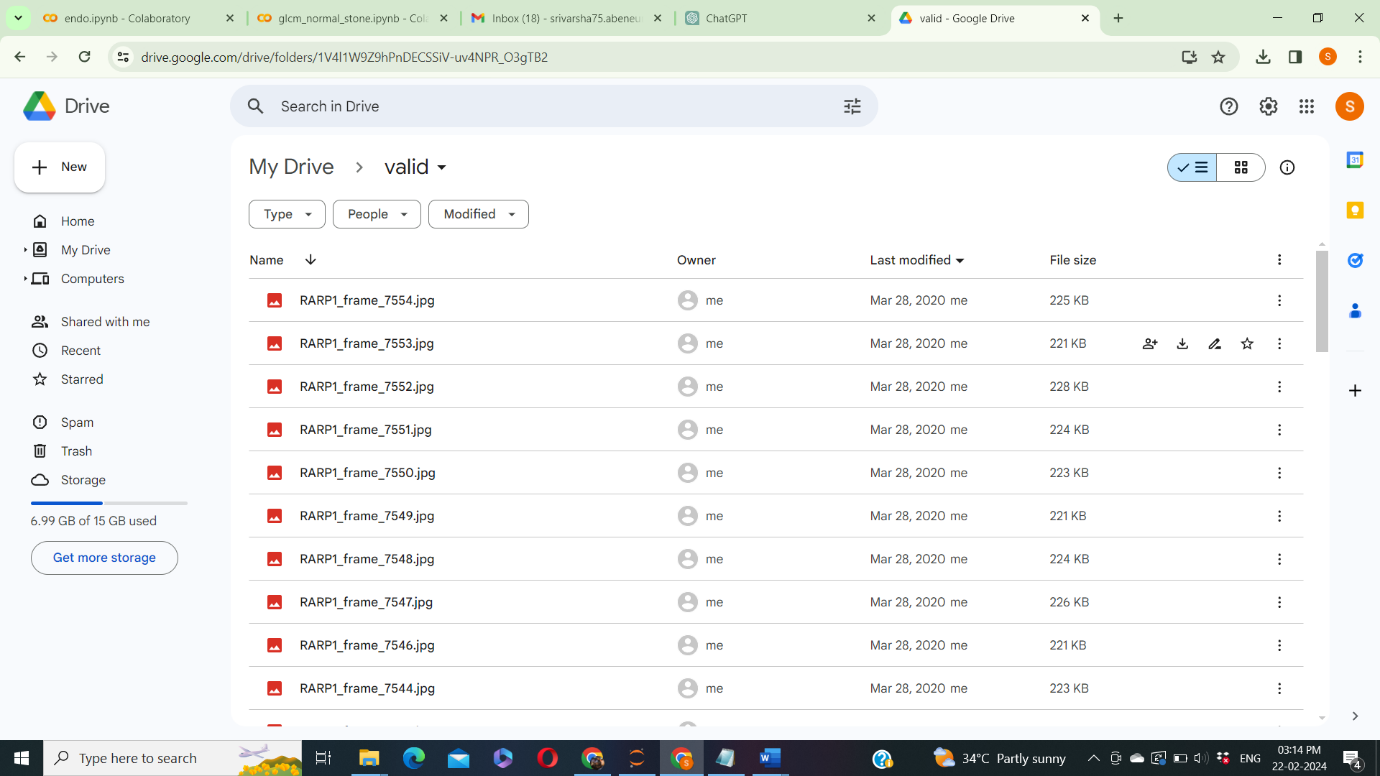


Figure 1: Image Dataset

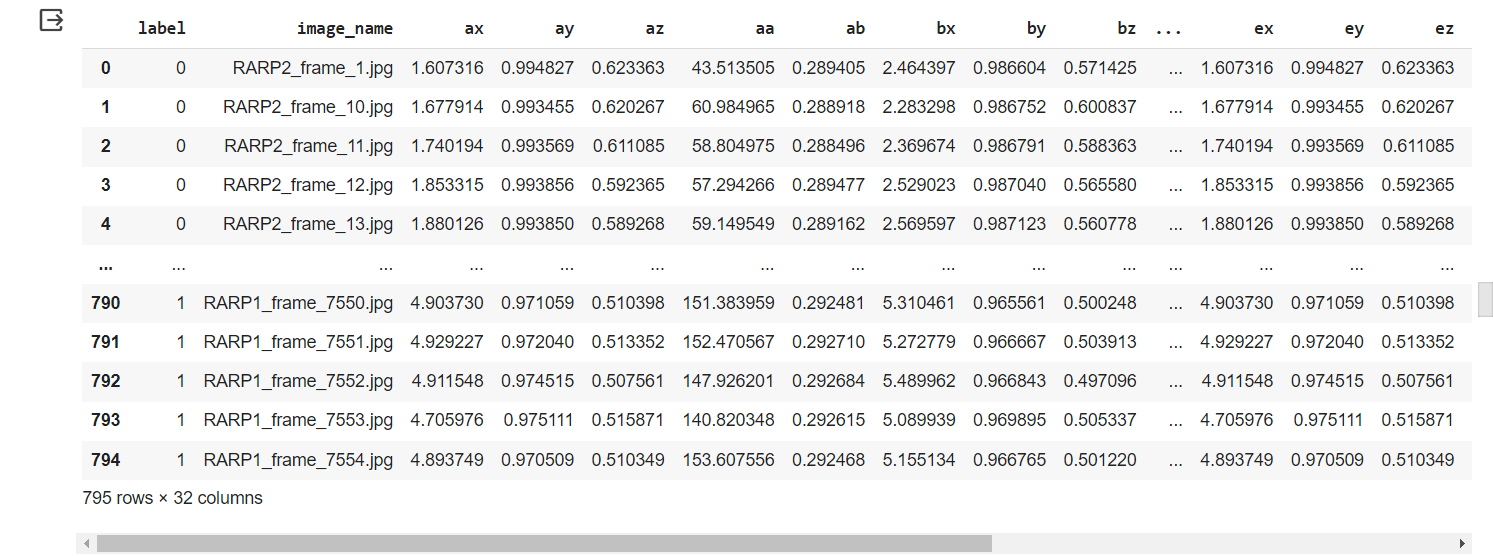


Figure 2: CSV Dataset

**Results:**

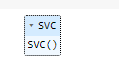


Figure 3: Support Vector Classifier Algorithm

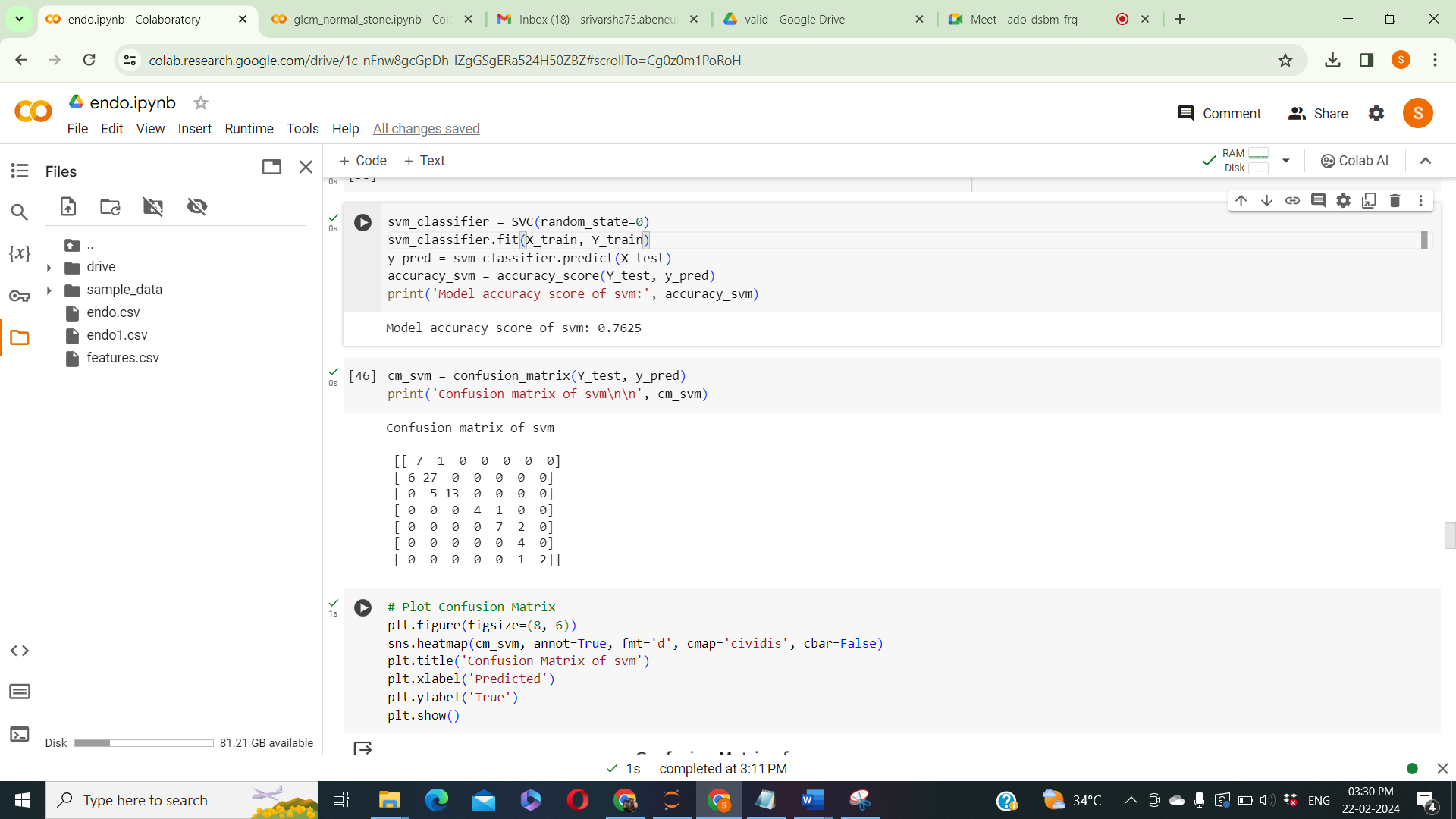


Figure 4: Accuracy calculation of support vector classifier algorithm

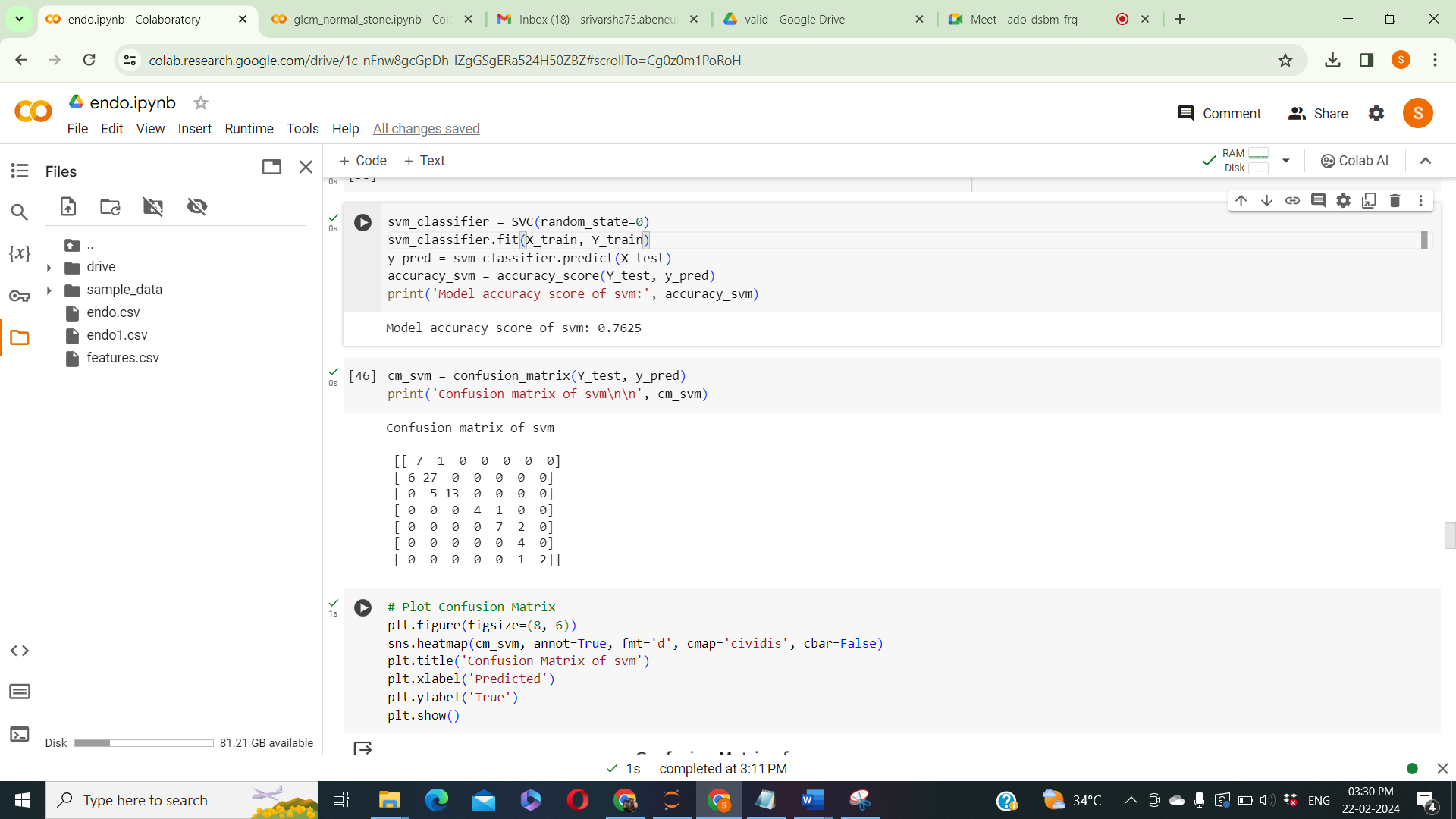


Figure 5: Confusion matrix of support vector classifier algorithm

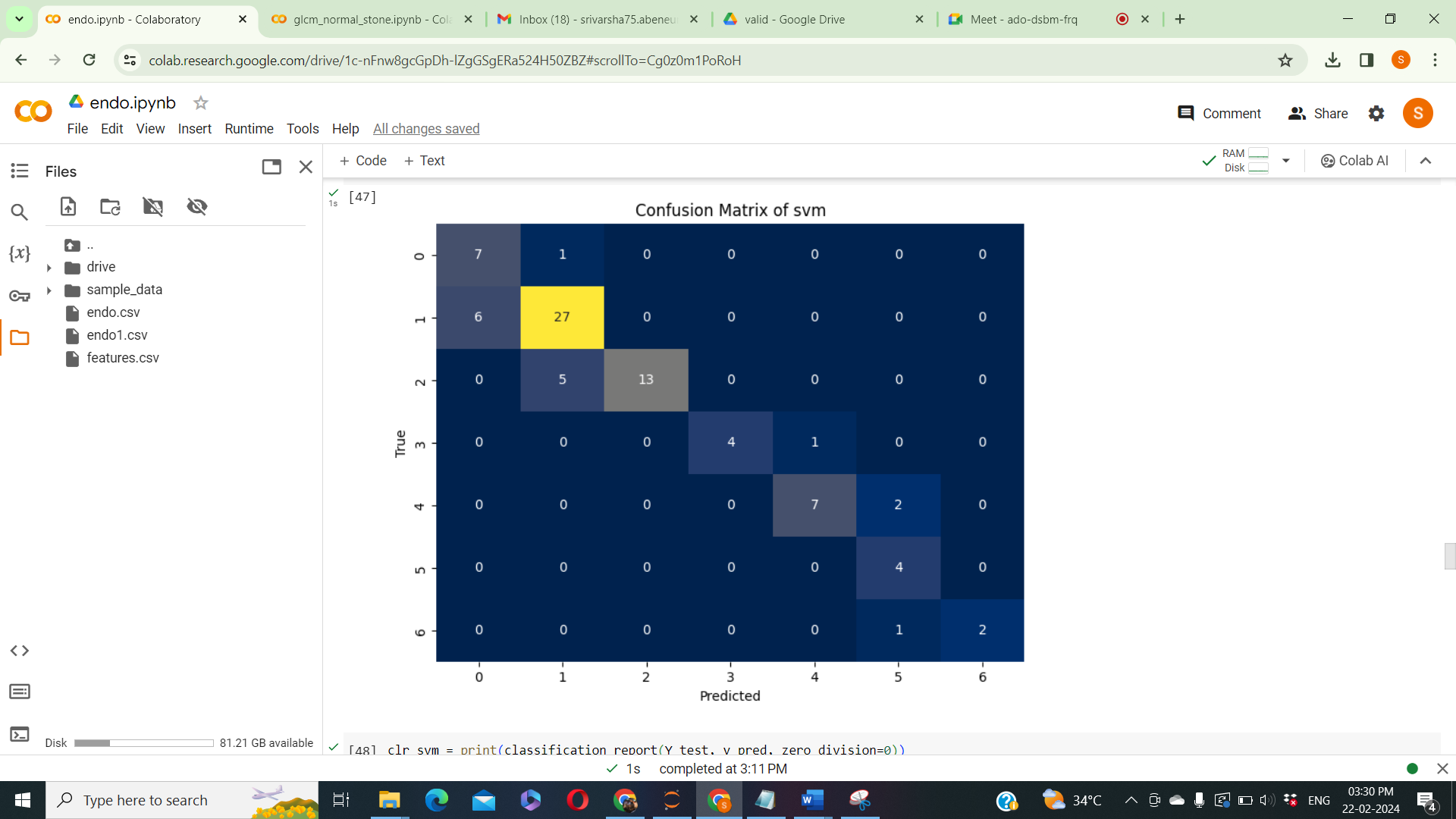


Figure 6: Confusion matrix graph of Support vector classifier algorithm

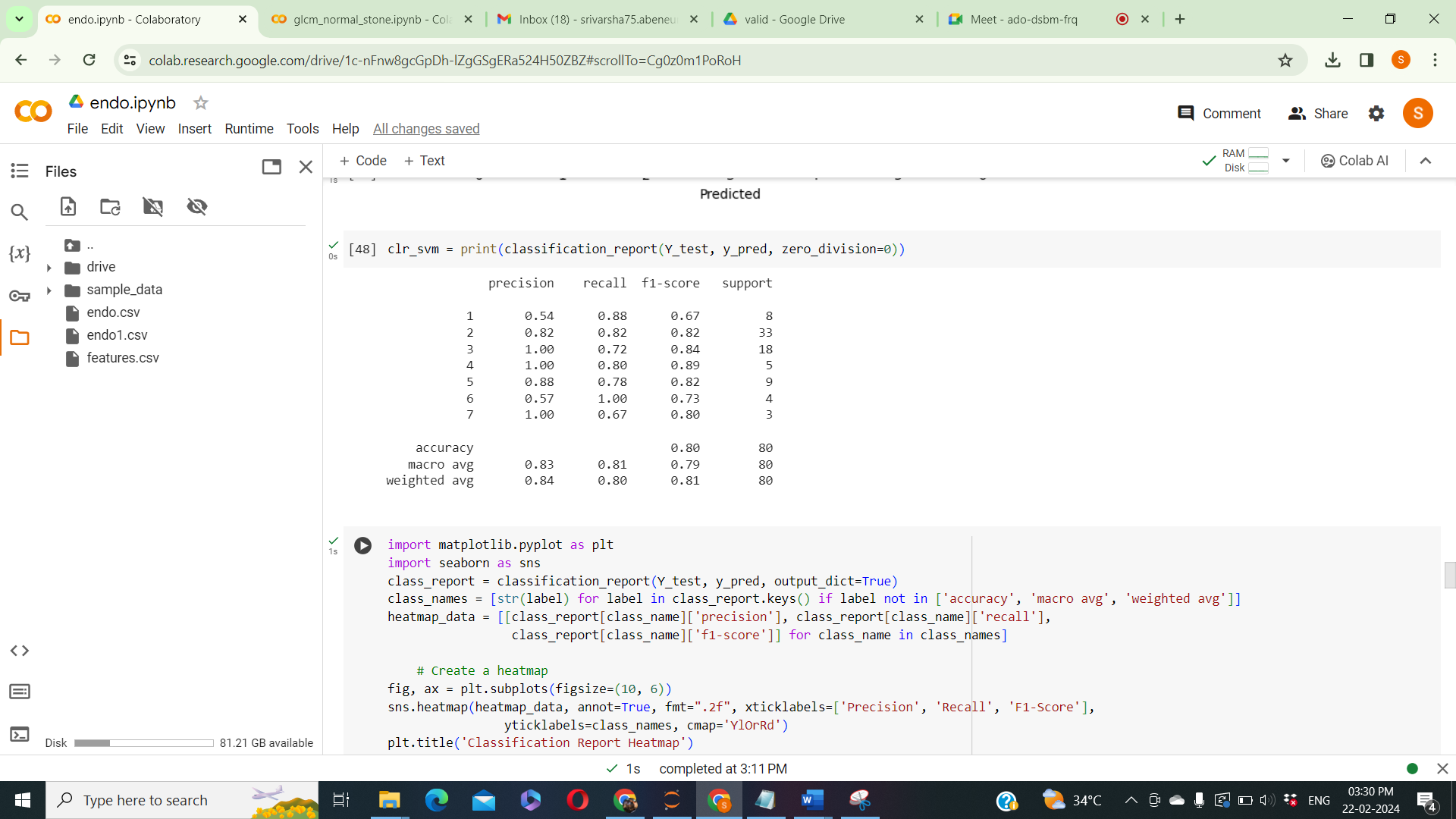


Figure 7: Classification report of support vector classifier algorithm



Figure 8: Classification report graph of Support vector classifier algorithm

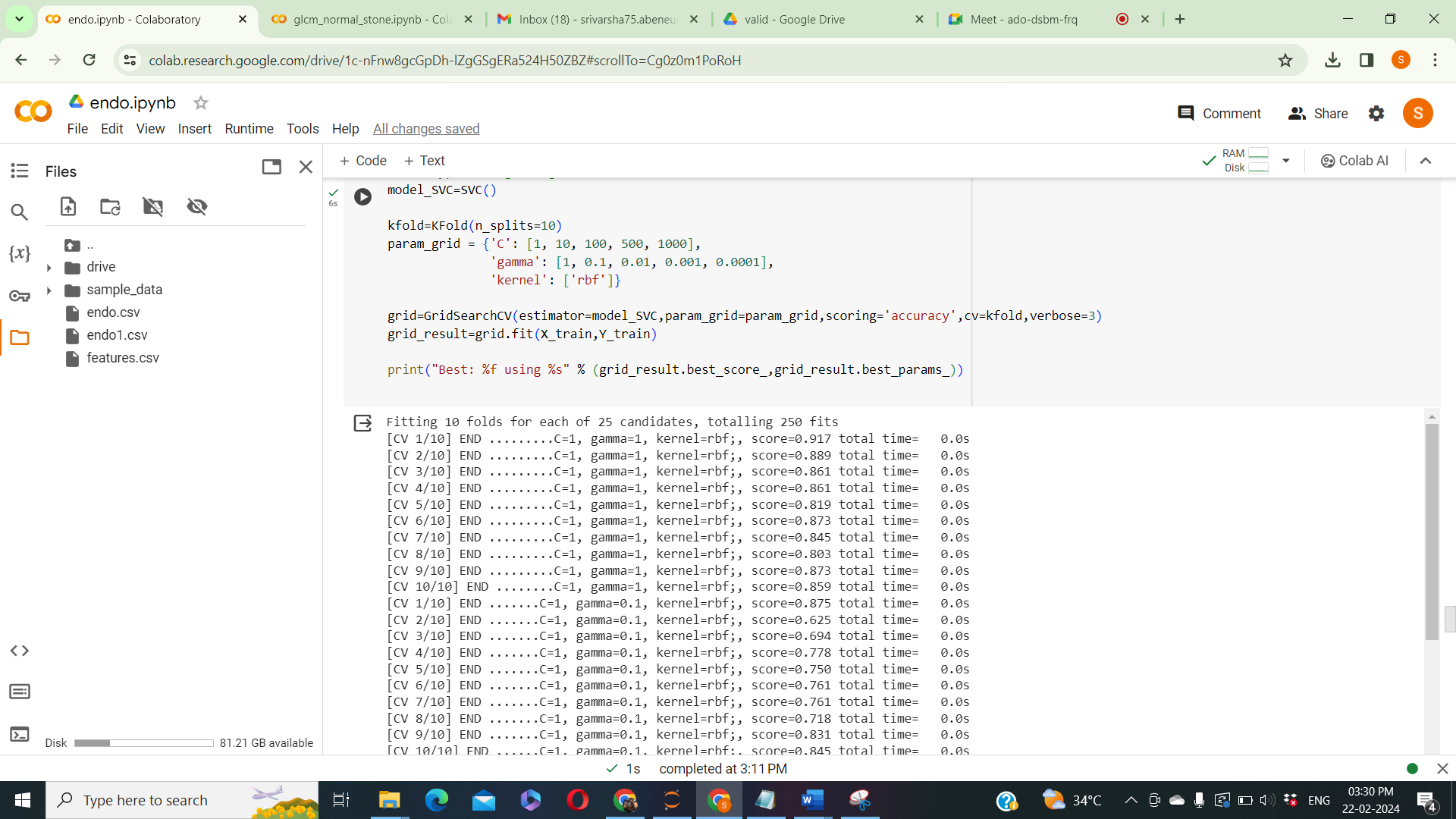


Figure 8: SVM hypertuning using GridSearchCV

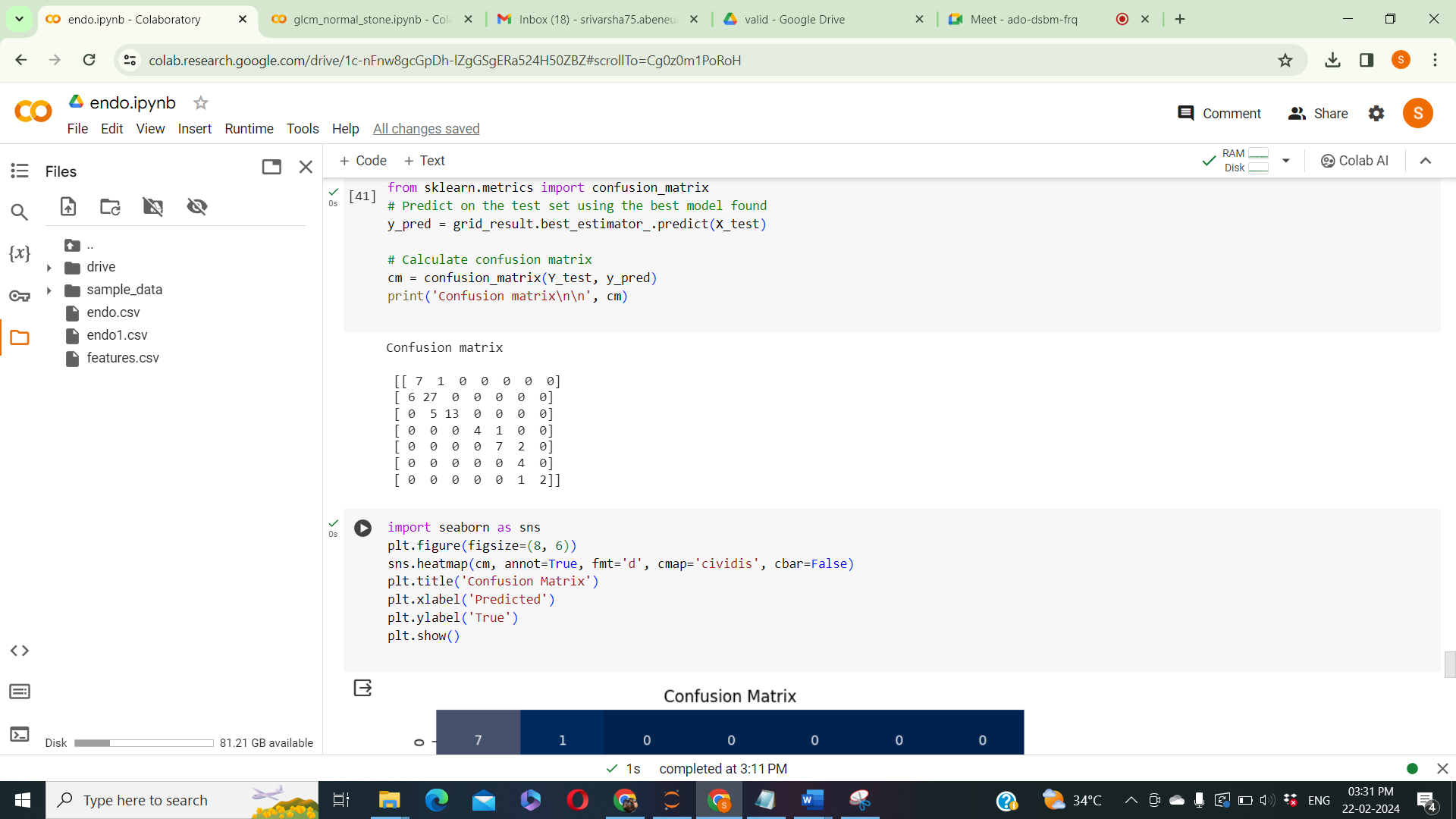


Figure 9: Confusion matrix of SVM hypertuning using GridSeachCV

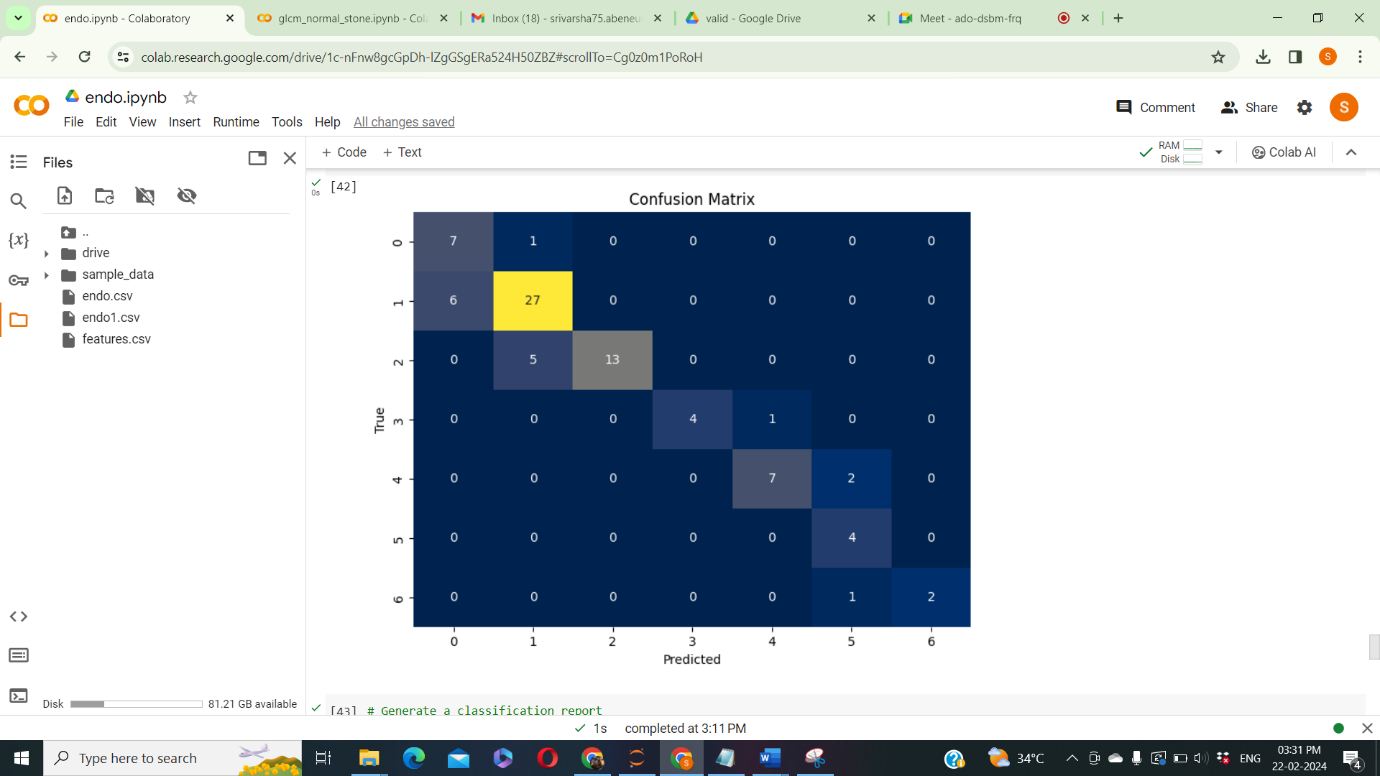


Figure 10: Confusion matrix graph of SVM hypertuning using GridSeachCV

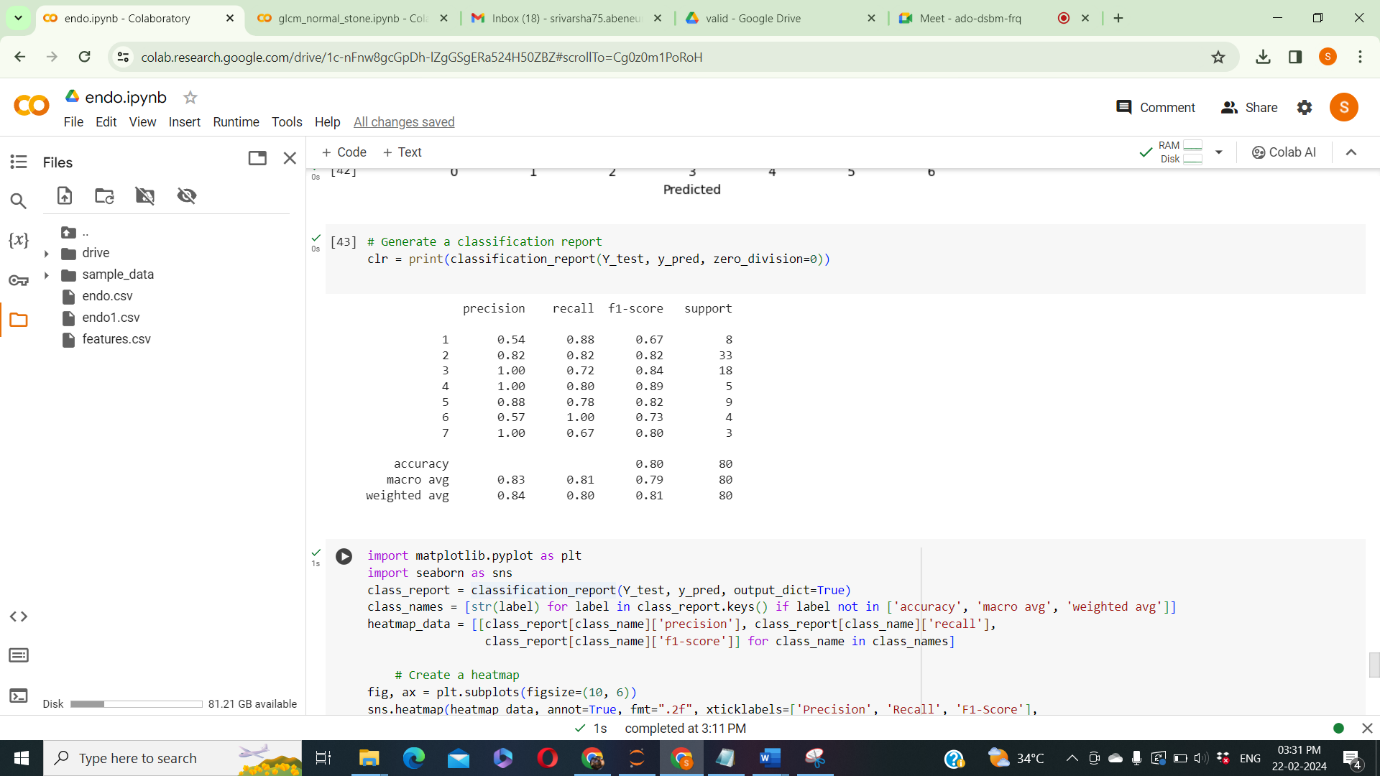


Figure 11: Classification report of SVM hypertuning using GridSeachCV



Figure 12: Classification report graph of SVM hypertuning using GridSeachCV

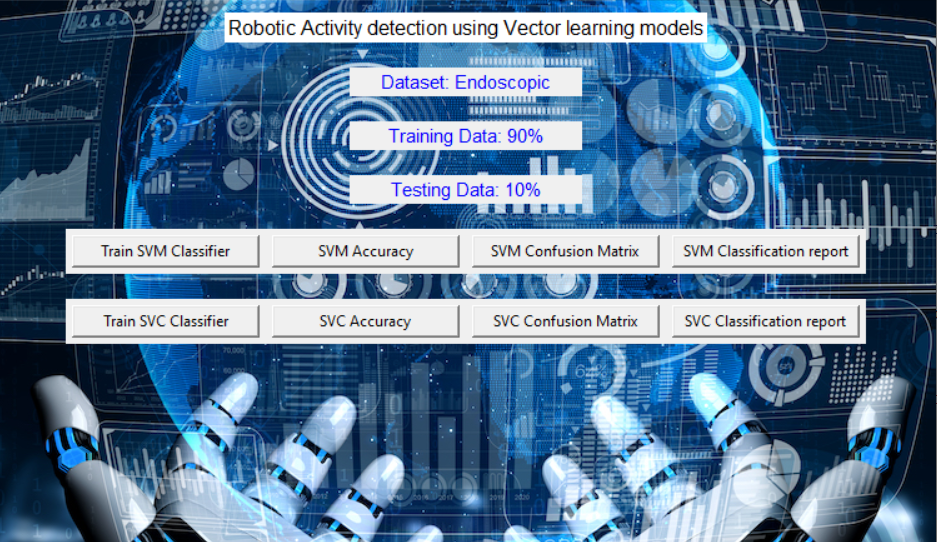


Figure 13: Frame Work Design





Figure 14: Classifier Training

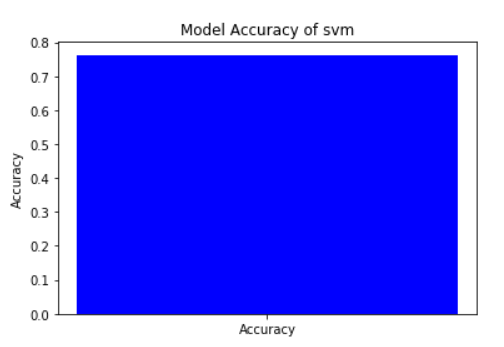


Figure 15: Accuracy Calculation Graph of SVM

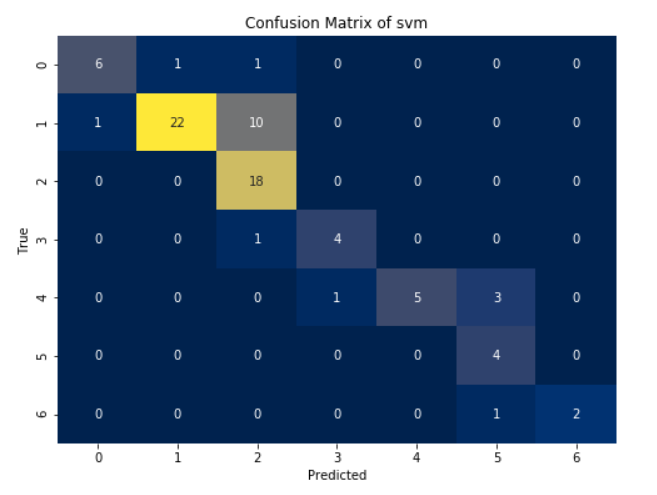


Figure 16: Confusion matrix graph of SVM

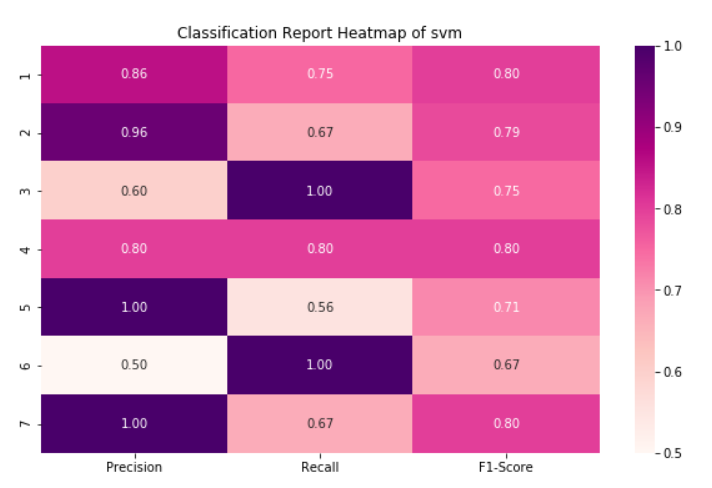


Figure 17: Classification report graph of SVM

Figure 18: Accuracy Calculation using SVC

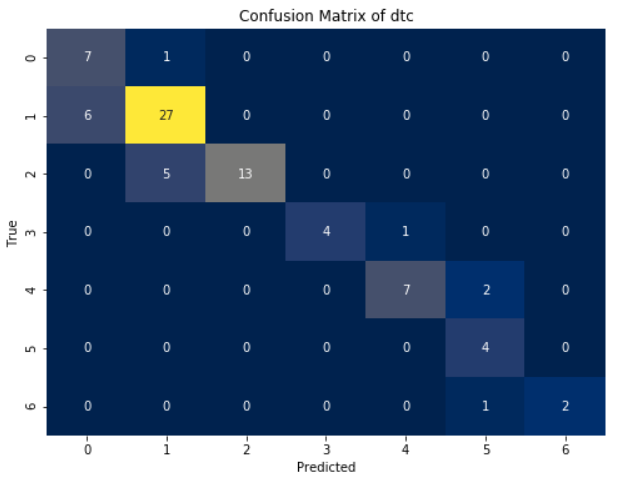


Figure 19: Confusion matrix graph of SVC

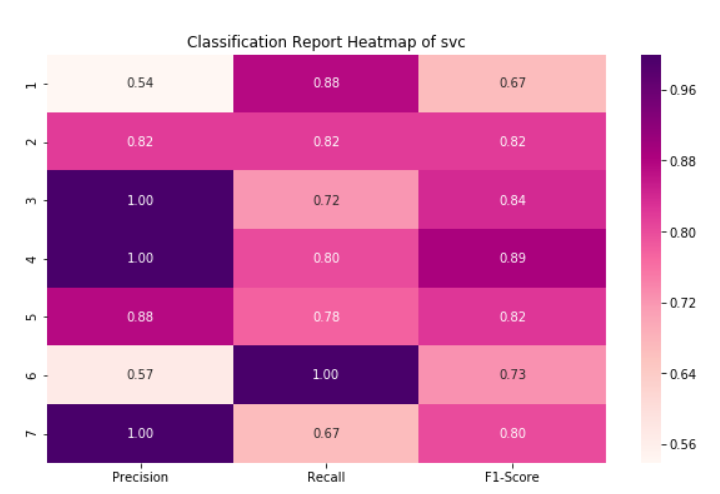


Figure 20: Classification report of SVC

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1 – score | Support |
| 1 | 0.54 | 0.88 | 0.67 | 8 |
| 2 | 0.82 | 0.82 | 0.82 | 33 |
| 3 | 1.00 | 0.72 | 0.84 | 18 |
| 4 | 1.00 | 0.80 | 0.89 | 5 |
| 5 | 0.88 | 0.78 | 0.82 | 9 |
| 6 | 0.57 | 1.00 | 0.73 | 4 |
| 7 | 1.00 | 0.80 | 0.80 | 3 |
| accuracy |  |  | 0.80 | 80 |
| Macro avg | 0.83 | 0.81 | 0.79 | 80 |
| Weighted avg | 0.84 | 0.80 | 0.81 | 80 |

Table 1: classification report of SVM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1 – score | Support |
| 1 | 0.54 | 0.88 | 0.67 | 8 |
| 2 | 0.82 | 0.82 | 0.82 | 33 |
| 3 | 1.00 | 0.72 | 0.84 | 18 |
| 4 | 1.00 | 0.80 | 0.89 | 5 |
| 5 | 0.88 | 0.78 | 0.82 | 9 |
| 6 | 0.57 | 1.00 | 0.73 | 4 |
| 7 | 1.00 | 0.80 | 0.80 | 3 |
| accuracy |  |  | 0.80 | 80 |
| Macro avg | 0.83 | 0.81 | 0.79 | 80 |
| Weighted avg | 0.84 | 0.80 | 0.81 | 80 |

Table 2: Classification report of SVM hypertuning using GridSeachCV

The classification report is a performance evaluation tool that shows the precision, recall, f1-score, for each class in a classification problem. In training images using the deep learning model, the classification report would provide information about how well the model performed in classifying images into different categories. The precision represents the percentage of correctly classified images among all the images classified as belonging to a specific class. The recall represents the percentage of correctly classified images among all the images that actually belong to a specific class. The f1-score is a harmonic mean of precision and recall, and support represents the number of images in each class.

The accuracy has been calculated for the model that has been implemented, and the result for the model is compared in Table

|  |  |
| --- | --- |
| Algorithms | Accuracy |
| SVM | 76 |
| SVM hypertuning using GridSeachCV | 95 |

Table 3: Accuracy comparison of algorithm.

|  |  |  |
| --- | --- | --- |
| Dataset Count | Training Value | Testing Value |
| 795 | 90 | 10 |

Table 4: Consist of dataset count, Training and Testing percentage.

**CONCLUSION**

In conclusion, the proposed approach for robotic activity detection using vector learning models, particularly leveraging the Gray-Level Co-occurrence Matrix (GLCM) method followed by Support Vector Machine (SVM) classification and hyperparameter tuning using GridSearchCV, offers a robust and effective framework for analysing endoscopic images and detecting various activities within the intestine. By extracting texture features using GLCM, the system captures intricate patterns and characteristics present in the images, providing rich representations of the underlying robotic activities. The SVM classifier, known for its ability to handle high-dimensional feature spaces and nonlinear data, effectively distinguishes between different activities, enabling accurate detection even in complex surgical environments.

Through hyperparameter tuning using GridSearchCV, the system optimizes the SVM model's performance, ensuring high accuracy and generalization capability. By systematically exploring different parameter combinations, the model achieves its maximum potential in terms of activity detection accuracy. This approach offers several advantages, including effectiveness in feature representation, robust classification capability, optimized performance through hyperparameter tuning, and scalability for real-world deployment. By accurately detecting robotic activities in endoscopic images, this system contributes to improving surgical outcomes and enhancing patient care in gastrointestinal surgeries. Further advancements and refinements in this approach hold the potential to revolutionize robotic-assisted interventions and contribute to the advancement of minimally invasive surgical techniques.

**REFERNCES**

Guevara, Jorge, et al. "Robotic activity recognition for flexible manufacturing systems using support vector machines."

Rizzi, Alessandro, et al. "Activity recognition in industrial robotic cells using hidden Markov models." Robotics and Computer-Integrated Manufacturing

Andreadis, Ioannis, et al. "Robotic manipulation activity recognition using PCA and HMMs."

Khan, Muhammad Usman, and M. Rizwan Jameel Qureshi. "Dynamic recognition of robotic arm activity using fuzzy logic." 2016 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST).

Muaaz, Muhammad, et al. "Activity recognition in intelligent robotic environments using machine learning techniques." International Journal of Advanced Computer Science and Applications