Computer Vision Final Project Report

Handwritten Digit Recognition

* Yusif Hajizade CS-020
* ID: 22022735

# Abstract:

Handwritten digit recognition is a fundamental problem in the field of computer vision and pattern recognition. This project aims to accurately identify handwritten digits in images and overlay them back onto the original image. The approach involves preprocessing the image using OpenCV, extracting digits, and training a model using custom handwritten datasets and the MNIST dataset. The model is trained using Histogram of Oriented Gradients (HOG) descriptors and K-Nearest Neighbors (KNN) or Support Vector Machine (SVM) algorithms. The performance of the models is evaluated based on accuracy metrics.

# Introduction:

Handwritten digit recognition has various applications such as postal mail sorting, bank check processing, and digitizing historical documents. However, accurately recognizing handwritten digits poses challenges due to variations in writing styles, sizes, and orientations. This project addresses the problem of handwritten digit recognition by implementing image processing techniques and machine learning algorithms.

# Main:

The main components of the project include data preparation, model training, and image process- ing. The project uses my own custom handwritten digits datasets (custom\_train\_digits.jpg) as training data, along with the MNIST dataset for training. Initially, the images are preprocessed using OpenCV to extract digits and convert them to grayscale. The digits are then segmented and resized to a standard size. Feature extraction is performed using Histogram of Oriented Gradients (HOG) descriptors, which capture shape information from the images. The extracted features are used to train models based on K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) algorithms.

**Importing all nessecsry libraries:**

import cv2

import numpy as np

import skimage.feature as ft

import matplotlib.pyplot as plt

import sklearn.model\_selection as ms

import sklearn.metrics as skm

import sklearn.utils as sku

**Constants:**

I define specific values that are used throughout the code and remain fixed during execution - IMAGE\_HEIGHT and IMAGE\_WIDTH represent the height and width of the images used in the system.

# IMAGE\_HEIGHT and IMAGE\_WIDTH represent the height and width of the images used in the system.

IMAGE\_HEIGHT = 28

IMAGE\_WIDTH = 2

**Methods**

1. **Image Split:**

This method divides the input training image into smaller cells, each con- taining a single digit. The image is divided into cells based on a specified cell size, extracting regions of interest that contain individual digits. This step is crucial for isolating and processing each digit separately during training and prediction Function segments an image into smaller cells determined by the given cell size, extracting image dimensions first. These cells are then created both horizontally and vertically. Optionally, if the flatten parameter is True, the array of cells is flattened to a single dimension per cell, before being returned for additional processing or analysis.

# This method divides the input training image into small cells (of a single digit) and uses these cells as training data.

# The default training image (MNIST) is a 1000x1000 size image and each digit is of size 10x20.

# So, we divide 1000/10 horizontally and 1000/20 vertically.

def divide\_image(img, cell\_size, flatten=True):

    height, width = img.shape[:2]

    cell\_width, cell\_height = cell\_size

    cells = [np.hsplit(row, width // cell\_width) for row in np.vsplit(img, height // cell\_height)]

    cells = np.array(cells)

    if flatten:

        cells = cells.reshape(-1, cell\_height, cell\_width)

    return cells

1. **Histogram of Oriented Gradients:**

Histogram of Oriented Gradients (HOG) descriptors are computed for the digit images to capture shape information. HOG features represent the dis-tribution of gradient orientations in localized regions of the image. By extracting HOG descriptors, the model can learn distinctive patterns and shapes associated with handwritten digits, improving recognition accuracy.The compute\_hog\_descriptors function iterates over a set of images to extract Histogram of Ori- ented Gradients (HOG) descriptors. These descriptors are calculated with specific parameters such as the number of orientations, pixels per cell, and cells per block. The resulting HOG features are returned as a NumPy array for further processing or analysis.

def compute\_hog\_descriptors(images):

    hog\_features = []

    for image in images:

        hog\_descriptor = ft.hog(image,

                             orientations=8,

                             pixels\_per\_cell=(6, 6),

                             cells\_per\_block=(2, 2))

        hog\_features.append(hog\_descriptor)

    hog\_features\_array = np.array(hog\_features, dtype=np.float32)

    return hog\_features\_array

1. **Model Training:**

The project implements custom K-Nearest Neighbors (KNN) and Support Vector Machine (SVM) models for digit recognition. The models are trained using the extracted HOG features.

**3.1 K-Nearest Neighbors (KNN) Model:** The KNN model utilizes the K-Nearest Neighbors algorithm to classify digits based on their feature representations. It stores training samples and their labels and predicts the label of a new sample by finding the majority class among its nearest neighbors. The CustomKNNModel class encapsulates a K-Nearest Neighbors (KNN) model with custom train- ing and prediction methods. It allows for training on input data samples and their corresponding labels using the KNN algorithm with the specified number of neighbors (k). The predict method en- ables making predictions on new data samples based on the trained model, returning the predicted labels.

class CustomKNNModel():

    def \_\_init\_\_(self, k=3):

        self.neighbor\_count = k

        self.machine = cv2.ml.KNearest\_create()

    def train(self, data, labels):

        self.machine.train(data, cv2.ml.ROW\_SAMPLE, labels)

    def predict(self, data):

        \_, results, \_, \_ = self.machine.findNearest(data, self.neighbor\_count)

        return results.ravel()

**3.2. Support vector machine (SVM) Model:** The SVM model employs the Support Vector Machine algorithm to classify digits into different classes. It learns a decision boundary that maximizes the margin between classes in the feature space. SVMs are effective for handling high-dimensional data and can capture complex patterns in the input data. The CustomSVMModel class represents a Support Vector Machine (SVM) model with custom train- ing and prediction methods. It allows for training on input data samples and their corresponding responses using the SVM algorithm with the specified parameters for regularization (C) and kernel coe icient (gamma). The predict method enables making predictions on new data samples based on the trained model, returning the predicted responses.

class CustomSVMModel():

    def \_\_init\_\_(self, num\_features, C=1, gamma=0.1):

        self.support\_vector\_machine = cv2.ml.SVM\_create()

        self.support\_vector\_machine.setType(cv2.ml.SVM\_C\_SVC)

        self.support\_vector\_machine.setKernel(cv2.ml.SVM\_RBF)  # SVM\_LINEAR, SVM\_RBF

        self.support\_vector\_machine.setC(C)

        self.support\_vector\_machine.setGamma(gamma)

        self.num\_features = num\_features

    def train(self, samples, responses):

        self.support\_vector\_machine.train(samples, cv2.ml.ROW\_SAMPLE, responses)

    def predict(self, samples):

        results = self.support\_vector\_machine.predict(samples.reshape(-1, self.num\_features))

        return results[1].ravel()

1. **Extract Digits:**

This method extracts the digits from the image by identifying contours and bounding boxes that encapsulate each digit. It filters the extracted contours based on size and hierarchy level to isolate individual digits. This step ensures that only relevant regions containing digits are considered for further processing and analysis. The extract\_digits function takes contours and their corresponding hierarchies as input and returns the bounding boxes of the digits found in the image. It filters out bounding boxes based on size and hierarchy level to extract only the ones containing digits, avoiding nested bounding boxes within digits.

def extract\_digits(contours, hierarchies):

    hierarchies = hierarchies[0]

    bounding\_boxes = [cv2.boundingRect(ctr) for ctr in contours]

    final\_bounding\_boxes = []

    # Find the most common hierarchy level to extract bounding boxes of digits

    unique\_levels, indices = np.unique(hierarchies[:, -1], return\_inverse=True)

    most\_common\_level = unique\_levels[np.argmax(np.bincount(indices))]

    for box, hierarchy in zip(bounding\_boxes, hierarchies):

        x\_coord, y\_coord, width, height = box

        # Extract rectangles containing digits based on size and hierarchy level

        if ((width \* height) > 250) and (10 <= width <= 200) and (10 <= height <= 200) and hierarchy[3] == most\_common\_level:

            final\_bounding\_boxes.append(box)

    return final\_bounding\_boxes

1. **Resize method:**

The resize images method resizes an input image array to a specified size using the Python Imaging Library (PIL). It converts the input array to a PIL image, resizes it according to the provided size argument, and then converts it back to a NumPy array before returning. Resizing images to a standard size ensures consistency in the input data for training and prediction. This function resizes an input image array (arr) to a specified size using the Python Imaging Library (PIL). It converts the input array to a PIL image, resizes it according to the provided size argument, and then converts it back to a NumPy array before returning. This function is typically used for resizing images to a desired shape or dimensions in image processing tasks.

from PIL import Image

def imresize(arr,size):

    img=Image.fromarray(arr)

    img=img.resize(size)

    return np.asarray(img)

**6. Image Processing:**

The image processing phase loads the input image, converts it to grayscale, and applies thresholding and morphological operations for segmentation. Contours are detected to isolate individual digits, which are then represented as Histogram of Oriented Gradients (HOG) descriptors. These descriptors capture shape information crucial for classification. Using a trained classifier like K-Nearest Neighbors or Support Vector Machine, the digits are classified, and their labels predicted. Predictions are overlaid onto the original image and a blank canvas for visu-alization. Finally, the processed image and the canvas with recognized digits are saved for analysis, seamlessly integrating image processing and machine learning for accurate digit recognition.In the modified method, variable names like img\_file, im, x, y, w, h, im\_digit, hog\_img\_data, pred have been changed to image\_file, loaded\_image, x\_pos, y\_pos, width, height, digit\_image, hog\_image\_data, prediction respectively, aiming for better readability and clarity. Additionally, comments have been updated accordingly

def process\_image(image\_file, classifier):

    loaded\_image = cv2.imread(image\_file)

    blank\_canvas = np.zeros((loaded\_image.shape[0], loaded\_image.shape[1], 3), np.uint8)

    blank\_canvas.fill(255)

    grayscale\_image = cv2.cvtColor(loaded\_image, cv2.COLOR\_BGR2GRAY)

    plt.imshow(grayscale\_image)

    morph\_kernel = np.ones((5, 5), np.uint8)

    ret, thresholded\_image = cv2.threshold(grayscale\_image, 127, 255, 0)

    thresholded\_image = cv2.erode(thresholded\_image, morph\_kernel, iterations=1)

    thresholded\_image = cv2.dilate(thresholded\_image, morph\_kernel, iterations=1)

    thresholded\_image = cv2.erode(thresholded\_image, morph\_kernel, iterations=1)

    contours, hierarchy = cv2.findContours(thresholded\_image, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

    digit\_rectangles = extract\_digits(contours, hierarchy)  # rectangles bounding the digits in user image

    for rectangle in digit\_rectangles:

        x\_pos, y\_pos, width, height = rectangle

        cv2.rectangle(loaded\_image, (x\_pos, y\_pos), (x\_pos + width, y\_pos + height), (0, 255, 0), 2)

        digit\_image = grayscale\_image[y\_pos:y\_pos + height, x\_pos:x\_pos + width]

        digit\_image = (255 - digit\_image)

        digit\_image = imresize(digit\_image, (IMAGE\_WIDTH, IMAGE\_HEIGHT))

        hog\_image\_data = compute\_hog\_descriptors([digit\_image])  # Using HOG descriptors

        hog\_image\_data = hog\_image\_data.astype(np.float32)  # Ensure float32 data type

        prediction = classifier.predict(hog\_image\_data)

        cv2.putText(loaded\_image, str(int(prediction[0])), (x\_pos, y\_pos), cv2.FONT\_HERSHEY\_SIMPLEX, 2, (255, 0, 0), 3)

        cv2.putText(blank\_canvas, str(int(prediction[0])), (x\_pos, y\_pos), cv2.FONT\_HERSHEY\_SIMPLEX, 3, (255, 0, 0), 5)

    plt.imshow(loaded\_image)

    cv2.imwrite("original\_overlay.png", loaded\_image)

    cv2.imwrite("final\_digits.png", blank\_canvas)

**6. Loading digits:**

This method loads the custom handwritten digits dataset or the MNIST dataset for training the digit recognition model. It reads the image files, preprocesses them, and prepares the data for training by extracting digits and their corresponding labels. Loading images is a crucial step in preparing the dataset for training the model on handwritten The load\_custom\_digits function processes a custom training image, extracting individual digit images and their corresponding labels. It utilizes contours to identify and extract digits, sorts them for labeling, and prepares the data for training a digit recognition model.

def load\_custom\_digits(image\_file):

    train\_data = []

    train\_labels = []

    class\_counter = 1

    loaded\_image = cv2.imread(image\_file)

    grayscale\_image = cv2.cvtColor(loaded\_image, cv2.COLOR\_BGR2GRAY)

    plt.imshow(grayscale\_image)

    morph\_kernel = np.ones((5, 5), np.uint8)

    ret, thresholded\_image = cv2.threshold(grayscale\_image, 127, 255, 0)

    thresholded\_image = cv2.erode(thresholded\_image, morph\_kernel, iterations=1)

    thresholded\_image = cv2.dilate(thresholded\_image, morph\_kernel, iterations=1)

    thresholded\_image = cv2.erode(thresholded\_image, morph\_kernel, iterations=1)

    contours, hierarchy = cv2.findContours(thresholded\_image, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

    digit\_rectangles = extract\_digits(contours, hierarchy)  # rectangles bounding the digits in user image

    def contour\_precedence(contour, columns):

        return contour[1] \* columns + contour[0]  # Establishes a row-wise ordering for contours.

    # Sort rectangles according to x,y pos for labeling

    digit\_rectangles.sort(key=lambda x: contour\_precedence(x, loaded\_image.shape[1]))

    for index, rect in enumerate(digit\_rectangles):

        x\_pos, y\_pos, width, height = rect

        cv2.rectangle(loaded\_image, (x\_pos, y\_pos), (x\_pos + width, y\_pos + height), (0, 255, 0), 2)

        digit\_image = grayscale\_image[y\_pos:y\_pos + height, x\_pos:x\_pos + width]

        digit\_image = (255 - digit\_image)

        digit\_image = imresize(digit\_image, (IMAGE\_WIDTH, IMAGE\_HEIGHT))

        train\_data.append(digit\_image)

        train\_labels.append(class\_counter % 10)

        if index > 0 and (index + 1) % 10 == 0:

            class\_counter += 1

    cv2.imwrite("training\_box\_overlay.png", loaded\_image)

    return np.array(train\_data), np.array(train\_labels)

# Results:

The project achieves promising results in handwritten digit recognition. By leveraging HOG fea- tures and machine learning algorithms, the models achieve accuracies of around 80% on test images.

* Accuracy KNN model: 0.78%
* Accuracy SVM model: 0.81%

The accuracy metrics demonstrate the effectiveness of the approach in accurately identifying hand- written digits.

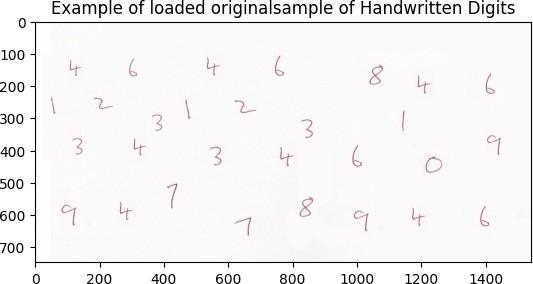
# File paths for training and testing images

TRAIN\_MNIST\_IMG = 'digits.png' # MNIST training dataset

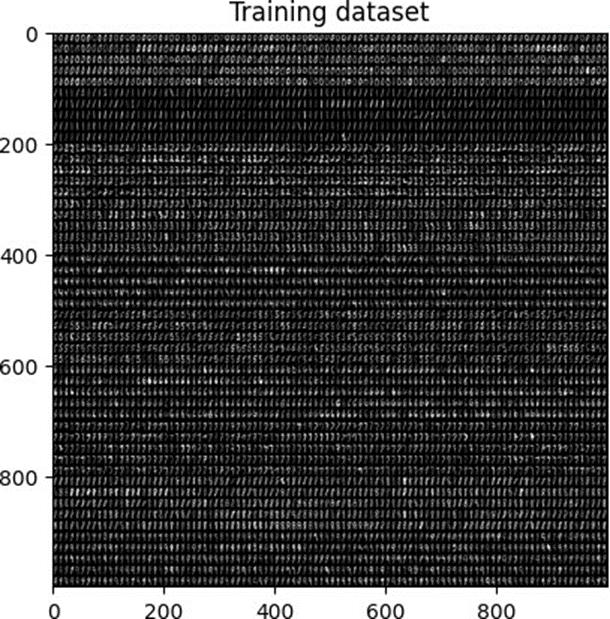
TRAIN\_USER\_IMG = 'training\_digits.jpg' # Custom training dataset written by me

SAMPLE\_USER\_IMG = 'sample\_image.png' # sample image which will be procced to extract digits and identify them, then again overlay the identified digits on the original image

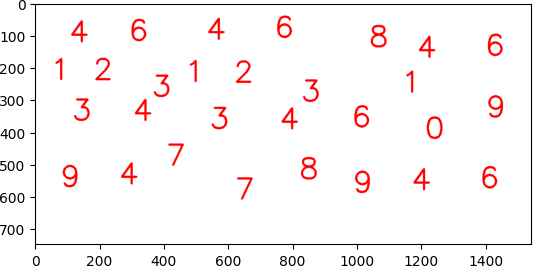
plt.title("Example of loaded originalsample of Handwritten Digits")

plt .imshow(cv2.imread(SAMPLE\_USER\_IMG))

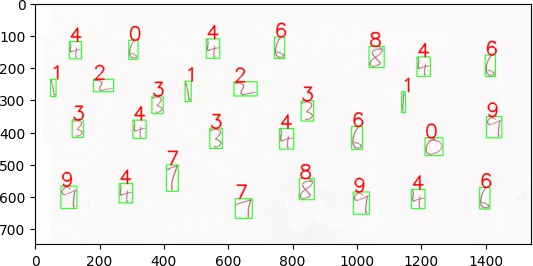
plt.title("Training dataset")

plt.imshow(cv2.imread(TRAIN\_MNIST\_IMG))

**Executing the program will generate 2 output files:**

* Accuracy: 0.78%
* Accuracy: 0.82%
* This is a final plain image with just the recognized numbers printed.

This is the original image with digit boxes and the identified number placed above handwritten.



# Conclusion:

Handwritten digit recognition is a challenging task, but through the integration of image process-ing techniques and machine learning algorithms, this project demonstrates significant progress in addressing the problem. By leveraging custom datasets and advanced feature extraction methods like HOG descriptors, the project achieves accurate digit recognition results. Overall, the project contributes to the advancement of handwritten digit recognition technology, with potential appli-cations in various fields.

# References:

* Pavitrakumar. (n.d.). GitHub - pavitrakumar78/Street-View-House-Numbers-SVHN- Detection-and-Classification-using-CNN: A 2-CNN pipeline to do both detection (using bounding box regression) and classification of numbers on SVHN dataset. GitHub. https://github.com/pavitrakumar78/Street-View-House-Numbers-SVHN-Detection-and- Classification-using-CNN
* Digit Recognizer | Kaggle. (n.d.). h[ttps://www.kaggle.com/competitions/digit-recognizer](http://www.kaggle.com/competitions/digit-recognizer)
* Pavitrakumar. (n.d.-a). GitHub - pavitrakumar78/Python-Custom-Digit-Recognition: Using OpenCV in python to recognize digits in a scanned page of handwritten digits. GitHub. https://github.com/pavitrakumar78/Python-Custom-Digit-Recognition
* Team, D., Team, D., & Team, D. (2021, March 14). Deep Learning Project – Handwritten Digit Recognition using Python. DataFlair. https://data-flair.training/blogs/python-deep- learning-project-handwritten-digit-recognition/