Solving Sudoku Puzzles with Computer Vision

Vaibhav Thakkar
Roll no: 170778
vaithak@iitk.ac.in

June 12, 2020

Abstract

A web app was created which allows user to input an image of the 9×9 Sudoku puzzle or choose from the some example images, the parsed Sudoku grid is shown as output to user, also if the parsed puzzle is valid by the rules of Sudoku then the output of the solved puzzle is also shown to the user, otherwise an error message is displayed, in this case the user can edit any of the digits parsed by the algorithm in sidebar of the app and then the puzzle will be solved by the updated grid provided by the user.

1 Introduction

Sudoku puzzles are very famous and can be seen at various places like newspapers, articles, mobile apps etc. Due to such a large popularity and interesting algorithmic techniques for solving the puzzle, there exists a lot of solvers which take as input the puzzle and outputs the solved puzzle. This project aims to takes this a step ahead and tries to take the input puzzle as an image, thus removing the manual typing of the puzzle for user. Although there are various sizes of Sudoku, we will consider the standard 9×9 Sudoku puzzle only.

2 Methodology

The whole process is implemented as a pipeline rather than a single algorithm, where each part of the pipeline can be fine tuned and improved. The image below represent the pipeline.

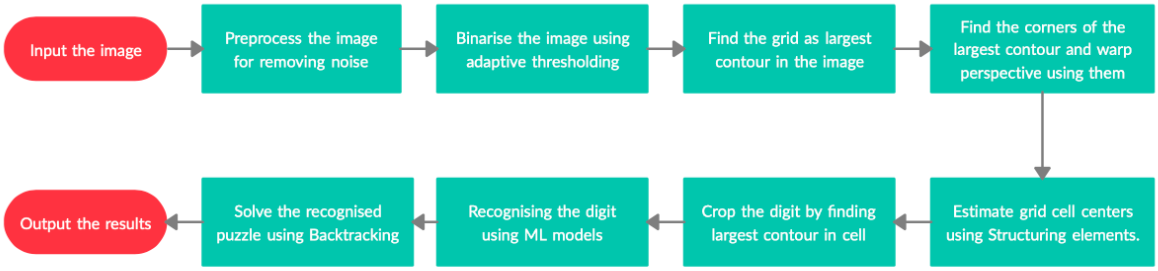

Figure 1: Complete Pipeline

2.1 Preprocessing

The input image can have various sources of noise which needs to be eliminated to properly parse the grid and then recognising the digits accurately. For, removing these noises, the following steps are done:

2.1.1 Contrast enhancement

Our app should be able to deal with various lighting conditions and to handle that we first improve the contrast of the image using Contrast Limited Adaptive Histogram Equalization.

2.1.2 Smoothing

The image is smoothed using a Gaussian blurring filter, which is a low pass filter and thus helps in removing the noise (which is of high frequency). Along with this, a bilateral filter is also used which is used for smoothing the image but still preserving the edges in the image.

2.2 Binarize the image

After preprocessing the image, our next task is to convert the image into binary color format, rather than the standard RGB colorspace. For this we first convert the image into grayscale, then use adaptive thresholding to convert the image into binary colorspace. Adaptive thresholding performs better than thresholding with a constant parameter, as it calculates threshold constant for each pixel by taking into account the intensities of its surrounding pixels.

2.3 Finding the boundaries of the grid

Now that we have binarized the image, we can move ahead for finding the main grid of the puzzle and extracting it from the image. For this task, we find contours (curve joining all the continuous points on the boundary) in the image and we assume that the biggest contour in the image will be puzzle grid, which is a reasonable assumption.

2.4 Finding the corners of the grid and Warping the perspective

After we have located the boundary of the grid, we can find four corner points using OpenCV's approxPolyDP function , which returns the minimum number of points required to approximate the polygon such that the difference of perimeter of the approximate polygon and the actual shape's perimeter is within epsilon (hyperparameter passed to the function). After we have obtained the points, we will order them by considering the following:

Note that the x and y coordinates considered below are image coordinates in which y coordinate increases on going down.

1. The top left corner of the grid has lowest value of x+y.
2. The bottom right corner of the grid has highest value of x+y.
3. The bottom left corner of the grid has lowest value of of x-y.
4. The top right corner of the grid has highest value of x-y.

The above considerations can handle a large variety of orientations in which the grid points towards the camera.

Using these four ordered points we do a perspective transformation such that the grid is extracted from the image with its axis aligned to the image axes.

2.5 Finding cells in the grid

After the grid is extracted, we will find each individual cell in the grid. An easy way of doing this is dividing both the axes into 9 equal parts, this approach is easy to implement but can result in various errors, for ex: there can be a very minor error in perspective transform which results in a slightly slanted grid lines, it can cause errors in recognizing digits in the later step. Also, this step assumes that the image is taken on a flat surface, which may not be the case.
To deal with this, we will use the approach described in (Yang, 2015), which uses two separate structuring elements for refining the position of center of a cell.
Below are the images taken from her report (one orange rectangle shows a rectangle of all ones) :

Figure 2: Left: the two structuring elements used. Right: refining a cell center's vertical position using SE A

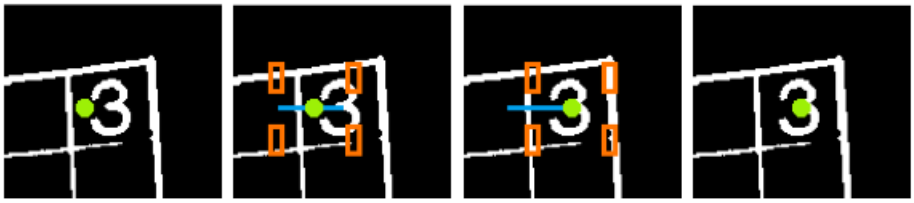

Figure 3: Refining a cell center's horizontal position, starting where vertical refinement left off. The final cell center estimate is shown in the far right image.

Using the center of the cell found, the corners of the cell are found and then a perspective transform is taken to obtain a (28×28) sized image.

2.6 Extracting digits from the cell

Now our aim is to extract the digit from the cell obtained, to do that we will again assume that the digit will have the largest contour in the cell, using this very reasonable assumption we find the boundary of the digit using which we find the bounding rectangle of the digit, which can easily found by considering the extreme top, bottom, right, left point of the contour found.
Also, to deal with the empty cells, we keep a fixed constant which defines the minimum required number of pixels in the cell to consider be as a filled.

2.7 Recognising extracted digits

Now that the digits from all the cells are extracted, our aim is to recognize the digits.
For this task, we will employ Machine learning techniques.
This task can be considered as a multi-class classification problem with number of classes=10, even though the number of classes should be 9, but I trained the model on MNIST handwritten digits data set considering all the classes, although all the images with label 0 can be removed for training but this didn't improve the accuracy much, so all classes were considered.
The dataset for handwritten digits were considered because the input image can also be partially filled by the user, so we also need to recognise handwritten digits.
Mainly three models were considered for classification:

2.7.1 Softmax Regression

Softmax regression is the extension of logistic regression to multiple classes, it is also a linear classifier. In this model probability of a data point belonging to a particular class is assumed to be defined by softmax function, where $\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$.

The complete model for this linear classifier can be represented as:

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(\theta^{(1)\top} x) \\ \exp(\theta^{(2)\top} x) \\ \vdots \\ \exp(\theta^{(K)\top} x) \end{bmatrix} \tag{1}$$

Instead of sending the raw pixels as features to the classifier, we send the HOG descriptors of the image to the classifier. In the HOG feature descriptor, the histograms of directions of gradients also known as oriented gradients are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of gradients is large around edges and corners,they are useful as the corners and edges can help significantly in analysing an image.

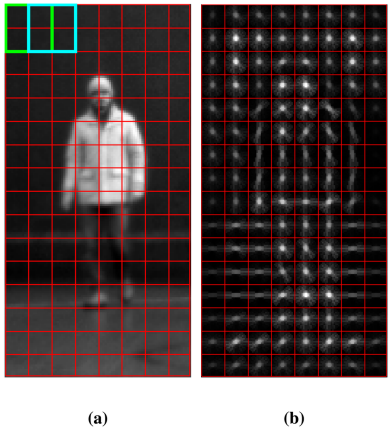

Figure 4: An example of HOG descriptors of an image

2.7.2 Gradient Boosted trees

Next model used is Gradient Boosted trees, this model is one type of ensemble models in which multiple weak decision tree classifiers (highly biased) are used sequentially to make a strong classifier, it does this by assigning higher weights to the data points which were misclassified by the previous classifier and then feeding them to the next weak classifier. Boosted trees helps in reducing bias from the classifier and along with it can also decrease variance.
I used the **xgboost** library for implementing the Gradient boosted tree model. This model was used with loss function as cross entropy which is same as that of softmax regression. A better accuracy than softmax regression was achieved from this model.
For this model also, we passed HOG descriptors as features in place of sending the raw pixels as features.

2.7.3 Convolutional Neural Networks

Artificial Neural networks can be used for learning very complex non linear functions and are widely used for classification.
Convolutional neural networks are specially used for classifying images as they have an extra part before the layers of ANN, this part is used for extracting the features from the image, thus it can be considered as the replacement of the HOG feature descriptors in the previous models, difference being that the features are also learnt from the during the training.
The output of this feature extraction part is then fed to completely connected ANN with ReLU activation function.
The image below explains the architecture of CNN used, the actual architecture used have some minor differences in the hyperparameters used.

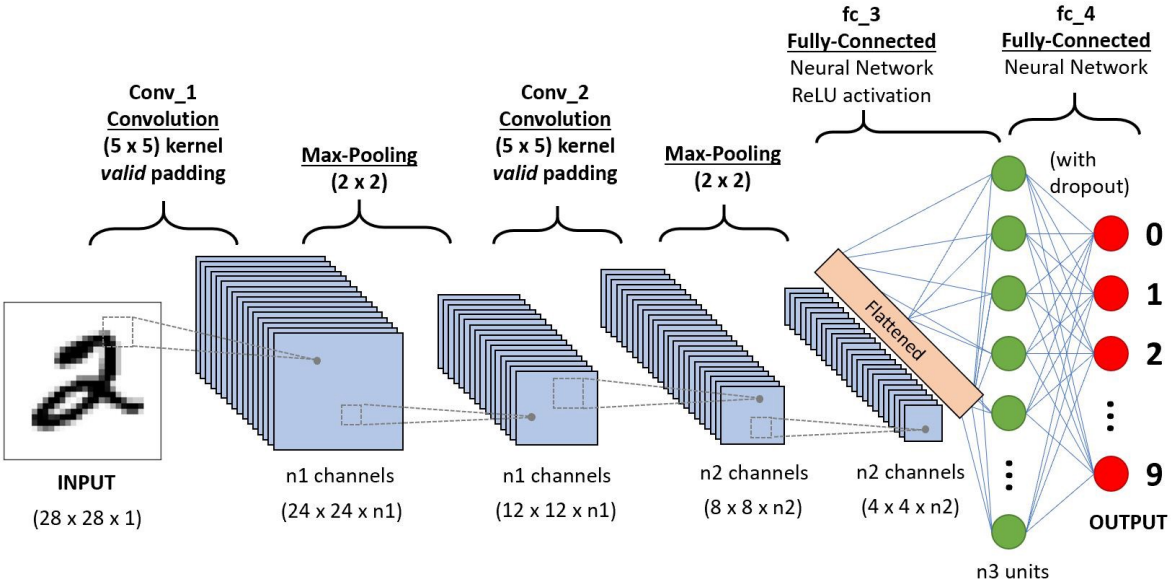

Figure 5: An example architecture of CNN

2.8 Solving the parsed Sudoku

Now that we have parsed (extracted and recognised) the puzzle, we need to verify the puzzle extracted is valid and then solve it.
For solving the puzzle we use the standard backtracking approach.
In backtracking, we first start with a sub-solution and if this sub-solution doesn't give us a correct final answer, then we just come back and change our sub-solution. We are going to solve our Sudoku in a similar way. The steps which we will follow are:

1. If there are no unallocated cells, then the Sudoku is already solved. We will just return true.
2. Or else, we will fill an unallocated cell with a digit between 1 to 9 so that there are no conflicts in any of the rows, columns, or the 3x3 sub-matrices.
3. Now, we will try to fill the remaining unallocated cells recursively and if this happens successfully, then we will return true.
4. If at any step we are unable to fill the cell with any digit, we will come back and change the digit we used to fill the cell. If there is no digit which fulfills the need, then we will just return false as there is no solution of this Sudoku.

3 Tools used

All the steps in the pipeline including the web interface was made using Python programming language, except the solver which uses backtracking to solve the parsed Sudoku puzzle was written in C++.

1. **OpenCV library**: Used for various image processing steps implemented in the complete pipeline.
2. **Keras**, **sklearn** libraries: Used for implementations of Machine Learning models for recognising digits.
3. **Swig**: for proving Python wrapping for my C++ implementation of Sudoku solver using backtracking.
4. **Streamlit**: Used for creating the web interface of the app.

4 Links for code and demo

Source code on Github: **https://github.com/vaithak/SudokuImageSolver**
The app is running on my personal server: **http://104.215.248.88/sudoku**
Video link for demo of the app: **https://youtu.be/zLT7nHLe0bs**

References

Yang, W. (2015). *Sudoku Solver.* `https://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Wang.pdf`.

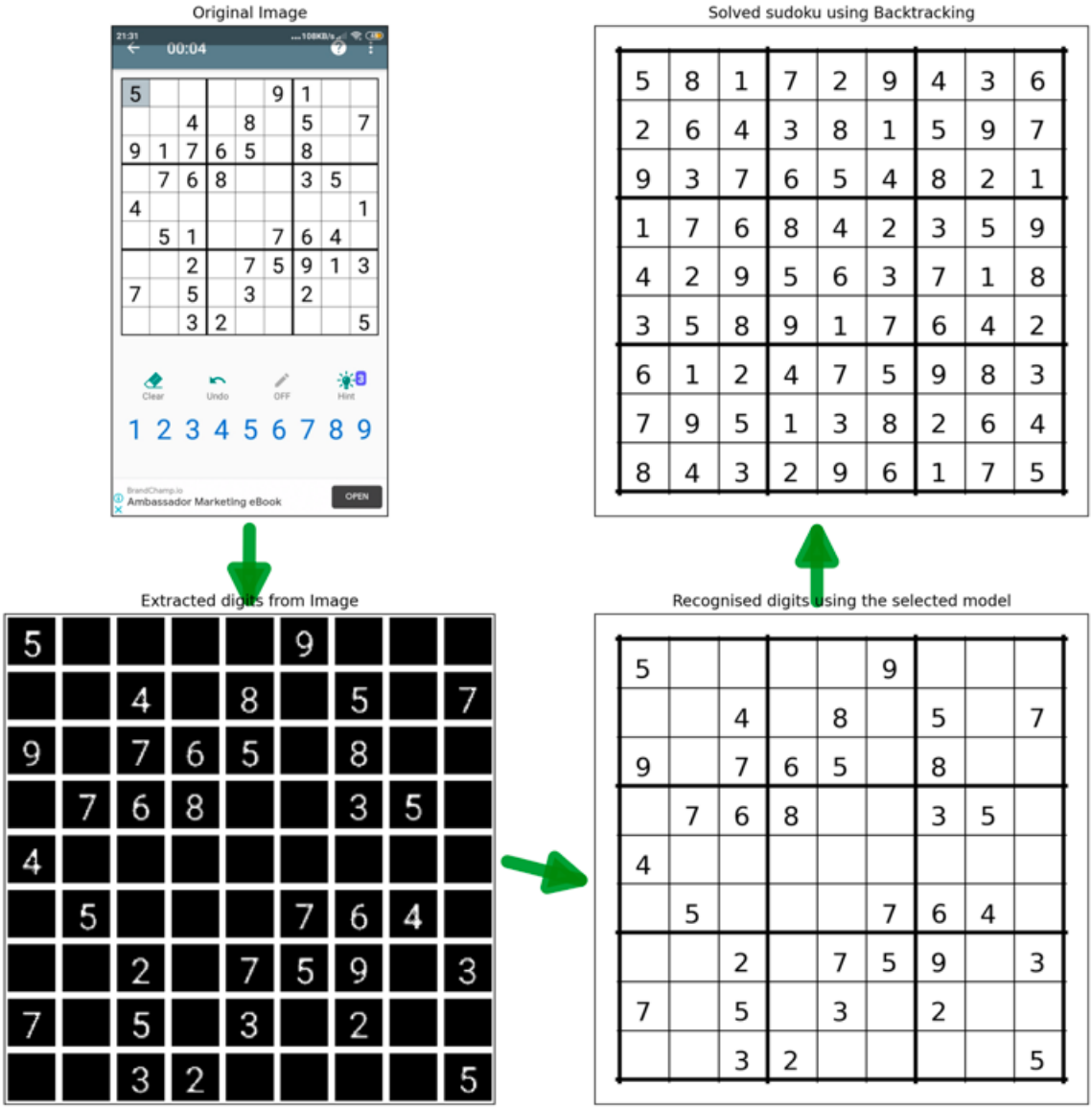

Figure 6: An example showing working of the app

5 Appendix

I am listing the code for most important files, other files and data can be found at the github link provided above.

processing.py

```
import cv2
import numpy as np

def basic_preprocessing(img: np.ndarray) -> np.ndarray:
    # create a CLAHE object for Histogram equalisation and improvng the contrast.
    img_plt = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    clahe = cv2.createCLAHE(clipLimit=0.8, tileGridSize=(8,8))
    enhanced = clahe.apply(img_plt)
```

```
    # Edge preserving smoother:
    # https://dsp.stackexchange.com/questions/60916/what-is-the-bilateral-filter-category-lpf-hp
    x, y = max(img.shape[0]//200, 5), max(img.shape[1]//200, 5)
    blurred = cv2.GaussianBlur(enhanced, (x+(x+1)%2, y+(y+1)%2), 0)
    blurred = cv2.bilateralFilter(blurred,7,75,75)
    return blurred

# requires a grayscale image as input
def to_binary(img: np.ndarray) -> np.ndarray:
    # opening for clearing some noise
    se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
    opened = cv2.morphologyEx(img, cv2.MORPH_OPEN, se)

    thresholded_img = cv2.adaptiveThreshold(opened, 255, cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_B
    inverted = cv2.bitwise_not(thresholded_img)

    if(img.shape[0] > 1000 and img.shape[1] > 1000):
        se = np.ones((2,2))
        eroded = cv2.erode(inverted, se, iterations=1)
    else:
        se = np.ones((2,2))
        eroded = cv2.erode(inverted, se, iterations=1)

    return eroded


def processImage(img: np.ndarray) -> np.ndarray:
    preprocessed = basic_preprocessing(img)
    binary = to_binary(preprocessed)

    if(img.shape[0] > 1000 and img.shape[1] > 1000):
      kernel = np.ones((3,3))
      dilated = cv2.dilate(binary, kernel, iterations=3)
      eroded = cv2.erode(dilated, kernel, iterations=3)
    else:
      eroded = binary

    return eroded
```

grid_extraction.py

```
import cv2
import numpy as np
import operator

# Note: Pass processed images only
def find_largest_contour(img: np.ndarray) -> (bool, np.ndarray):
    # find contours in the edged image, keep only the largest
    contours, hierarchy = cv2.findContours(img.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPL
    grid_cnt = np.array(sorted(contours, key=lambda x: cv2.contourArea(x), reverse=True))
    status, main_contour = False, np.array([])
```

```
    if len(grid_cnt) != 0:
        status, main_contour = True, grid_cnt[0]

    return (status, main_contour)


def perspective_transform(image: np.ndarray, corners: np.ndarray) -> np.ndarray:
    # Reference: https://stackoverflow.com/questions/57636399/how-to-detect-sudoku-grid-board-in

    def order_corner_points(corners):
        # Bottom-right point has the largest (x + y) value
        # Top-left has point smallest (x + y) value
        # Bottom-left point has smallest (x - y) value
        # Top-right point has largest (x - y) value
        bottom_r, _ = max(enumerate([pt[0][0] + pt[0][1] for pt in corners]), key=operator.item
        top_l = (bottom_r + 2)%4
        left_corners = [corners[i] for i in range(len(corners)) if((i!=bottom_r) and (i!=top_l))
        bottom_l, _ = min(enumerate([pt[0][0] - pt[0][1] for pt in left_corners]), key=operator.
        top_r = (bottom_l + 1)%2

        return (corners[top_l][0], left_corners[top_r][0], corners[bottom_r][0], left_corners[bo

    # Order points in clockwise order
    ordered_corners = order_corner_points(corners)
    top_l, top_r, bottom_r, bottom_l = ordered_corners

    # Determine width of new image which is the max distance between
    # (bottom right and bottom left) or (top right and top left) x-coordinates
    width_A = np.sqrt(((bottom_r[0] - bottom_l[0]) ** 2) + ((bottom_r[1] - bottom_l[1]) ** 2))
    width_B = np.sqrt(((top_r[0] - top_l[0]) ** 2) + ((top_r[1] - top_l[1]) ** 2))
    width = max(int(width_A), int(width_B))

    # Determine height of new image which is the max distance between
    # (top right and bottom right) or (top left and bottom left) y-coordinates
    height_A = np.sqrt(((top_r[0] - bottom_r[0]) ** 2) + ((top_r[1] - bottom_r[1]) ** 2))
    height_B = np.sqrt(((top_l[0] - bottom_l[0]) ** 2) + ((top_l[1] - bottom_l[1]) ** 2))
    height = max(int(height_A), int(height_B))

    # Construct new points to obtain top-down view of image in
    # top_r, top_l, bottom_l, bottom_r order
    dimensions = np.array([[0, 0], [width - 1, 0], [width - 1, height - 1],
                    [0, height - 1]], dtype = "float32")

    # Convert to Numpy format
    ordered_corners = np.array(ordered_corners, dtype="float32")

    # Find perspective transform matrix
    matrix = cv2.getPerspectiveTransform(ordered_corners, dimensions)

    # Return the transformed image
    return cv2.warpPerspective(image, matrix, (width, height))
```

```
# Main function for extracting the grid from the image
def extractGrid(processed_img: np.ndarray) -> (bool, np.ndarray):
    status, main_contour = find_largest_contour(processed_img)
    if status == False:
        return (status, main_contour)

    peri = cv2.arcLength(main_contour, True)
    approx = cv2.approxPolyDP(main_contour, 0.01 * peri, True)
    transformed_processed = perspective_transform(processed_img, approx[0:4])
    # For debugging
    # print(approx)
    # transformed_original = perspective_transform(orig_img, approx)
    # print(approx.shape, approx[0], approx[1], approx[2], approx[3])

    return (True, transformed_processed)
```

digit_extraction.py

```
import cv2
import numpy as np


def get_square_centers(transformed_img):
  lines_X = np.linspace(0, transformed_img.shape[1], num=10, dtype=int)
  lines_Y = np.linspace(0, transformed_img.shape[0], num=10, dtype=int)
  centers_X = [(lines_X[i] + lines_X[i-1])//2 for i in range(1, len(lines_X))]
  centers_Y = [(lines_Y[i] + lines_Y[i-1])//2 for i in range(1, len(lines_Y))]

  return centers_X, centers_Y

def extract_digit_from_cell(digit):
  if(np.sum(digit) < 255*5):
    return digit

  contours, _ = cv2.findContours(digit.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
  grid_cnt = np.array(sorted(contours, key=lambda x: cv2.contourArea(x), reverse=True))
  mask = np.zeros_like(digit)
  cv2.drawContours(mask, grid_cnt, 0, 255, -1) # Draw filled contour in mask
  out = np.zeros_like(digit) # Extract out the object and place into output image
  out[mask == 255] = digit[mask == 255]

  # Now crop
  (y, x) = np.where(mask == 255)
  (topy, topx) = (np.min(y), np.min(x))
  (bottomy, bottomx) = (np.max(y), np.max(x))
  out = out[topy:bottomy+1, topx:bottomx+1]
  #out = cv2.resize(out, (16,16), interpolation=cv2.INTER_AREA)

  # Now place on top of black image of size same as passed image in center
  res = np.zeros_like(digit)
```

```
  hh, ww = res.shape[0], res.shape[1]
  h, w = out.shape[0], out.shape[1]
  yoff = round((hh-h)/2)
  xoff = round((ww-w)/2)

  # use numpy indexing to place the resized image in the center of background image
  res[yoff:yoff+h, xoff:xoff+w] = out
  return res

def centering_se(shape: (int, int), shape_ones: (int, int)):
  x = np.zeros(shape)
  assert (shape_ones[0] < shape[0]) and (shape_ones[1] < shape[1])
  width = shape_ones[0]
  height = shape_ones[1]

  rows, cols = shape
  for i in range(width):
    for j in range(height):
      x[i][j], x[rows-1-i][j], x[i][cols-1-j], x[rows-1-i][cols-1-j] = 1, 1, 1, 1

  return x


def recentre(img: np.ndarray, prev_center: (int, int), h_se: np.ndarray, v_se: np.ndarray, h_mo
  # reference: https://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Wang.pdf
  max_res, max_center = 0, prev_center

  for i in range(v_mov_range[0], v_mov_range[1]):
    curr_center = (prev_center[0] + 0, prev_center[1] + i)
    start_row = max(curr_center[1] - v_se.shape[0]//2, 0)
    start_col = max(curr_center[0] - v_se.shape[1]//2, 0)
    partial = img[start_row:start_row+v_se.shape[0], start_col:start_col+v_se.shape[1]]

    curr_dot = np.sum(partial*(v_se[0:partial.shape[0], 0:partial.shape[1]]))
    # curr_dot = np.sum(img[x1:x1+v_se.shape[0], y1:y1+v_se.shape[1]]*(v_se))
    # print(curr_center, curr_dot)
    if max_res <= curr_dot:
      max_res = curr_dot
      max_center = curr_center

  # # print("max_center after v_se: ", max_center)
  prev_center = max_center
  max_res = 0
  for i in range(h_mov_range[0], h_mov_range[1]):
    curr_center = (prev_center[0] + i, prev_center[1] + 0)
    start_row = max(curr_center[1] - h_se.shape[0]//2, 0)
    start_col = max(curr_center[0] - h_se.shape[1]//2, 0)
    partial = img[start_row:start_row+h_se.shape[0], start_col:start_col+h_se.shape[1]]

    curr_dot = np.sum(partial*(h_se[0:partial.shape[0], 0:partial.shape[1]]))
    # print(curr_center, curr_dot)
```

```
    if max_res <= curr_dot:
      max_res = curr_dot
      max_center = curr_center

  # print("max_center after h_se: ", max_center)
  return max_center


def preprocess_digit(digit_img):
  # remove possible edges from border
  digit_img[0:3,:] = 0
  digit_img[:,0:3] = 0
  digit_img[-3:,:] = 0
  digit_img[:,-3:] = 0

  # dilating and eroding the digit
  if(np.sum(digit_img) < 255*30):
    return np.zeros_like(digit_img)

  return digit_img


def extractDigits(transformed_img):
  centers_X, centers_Y = get_square_centers(transformed_img)
  centers = [(centers_X[i], centers_Y[j]) for i in range(len(centers_X)) for j in range(len(cen
  kernel_shape = (centers_X[1] - centers_X[0], centers_Y[1] - centers_Y[0])

  ones_length = (kernel_shape[0]+kernel_shape[1])//20
  v_se = centering_se(kernel_shape, (2,ones_length))
  h_se = centering_se(kernel_shape, (ones_length,2))
  new_centers = []
  for i in range(len(centers)):
    v_mov_range, h_mov_range = (-kernel_shape[0]//8, kernel_shape[0]//8), (-kernel_shape[1]//8,
    if (i<9)             : h_mov_range = (-kernel_shape[1]//32, kernel_shape[1]//8)
    elif (i>71)          : h_mov_range = (-kernel_shape[1]//8, kernel_shape[1]//32)
    if (i%9 == 0)        : v_mov_range = (-kernel_shape[0]//32, kernel_shape[0]//8)
    elif ((i+1)%9 == 0) : v_mov_range = (-kernel_shape[0]//8, kernel_shape[0]//32)
    new_centers.append(recentre(transformed_img, centers[i], h_se, v_se, h_mov_range, v_mov_rang

  digits = []
  for center in new_centers:
    top_l = [center[0]-kernel_shape[1]//2, center[1]-kernel_shape[0]//2]
    top_r = [center[0]+kernel_shape[1]//2, center[1]-kernel_shape[0]//2]
    bottom_l = [center[0]-kernel_shape[1]//2, center[1]+kernel_shape[0]//2]
    bottom_r = [center[0]+kernel_shape[1]//2, center[1]+kernel_shape[0]//2]

    M = cv2.getPerspectiveTransform(np.float32([top_l, top_r, bottom_l, bottom_r]), np.float32([
    dst = cv2.warpPerspective(transformed_img,M,(28,28))
    dst = dst.astype('uint8')
    dst_mod = preprocess_digit(dst)
    dst_mod = extract_digit_from_cell(dst_mod)
```

```
    digits.append(dst_mod)

  return digits
```

recognise_digits.py

```
import numpy as np
import cv2
from sklearn.metrics import accuracy_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import joblib
import os
import xgboost as xgb
from tensorflow.keras.models import Sequential, load_model

models = ["CNN", "XGBOOST", "Softmax", "RandomForest", "GNB"]
files = ["CNN.h5", "XGBOOST.bin", "Softmax.pkl", "RandomForest.pkl", "GNB.pkl"]

# Digits array contains digits of grid in column major order
def predictDigits(digits: np.ndarray, model: int):
    assert (len(digits) == 81) and (model>=0) and (model<=4)

    res = ""
    for i in range(9):
        for j in range(9):
            digit = predictDigit(digits[j*9 + i], model)
            res += str(digit)

    return res
    #return "009000780830019000610000403001900027000040000590008300905000072000590048082000900"


def preprocess(x_vec):
    x_vec = x_vec.astype(np.uint8)
    x_vec = 255 - x_vec
    hog = cv2.HOGDescriptor((28, 28), (14, 14), (7, 7), (14, 14), 12)
    return hog.compute(x_vec).reshape(1, -1)

# Take decision based on probability vector of each class
# cost_r: Cost of rejection (In our case we will mark it as empty or no digit = 0 in Sudoku)
# cost_w: Cost of wrong classification (In our case we will mark it as empty or no digit = 0 in
def take_decision(probabilities, cost_r=10, cost_w=30):
  assert cost_w != 0

  # Reference: https://www.cs.ubc.ca/~murphyk/Teaching/CS340-Fall07/dtheory.pdf
  pred_class = np.argmax(probabilities)
  if(probabilities[pred_class] > (1 - (cost_r/cost_w))):
    return pred_class

  # reject => No digit => 0 for our case
  return 0
```

```
def preprocess_for_CNN(digit: np.ndarray):
  digit = 255 - digit
  digit = digit/255
  digit = digit.reshape((1, 28, 28, 1))
  return digit

def predictDigit(digit: np.ndarray, model):
    # Less than 10 pixels coloured
    if np.sum(digit) < 10*255:
        return 0

    if models[model]=="XGBOOST":
      clf = xgb.XGBClassifier(objective="multi:softmax", booster="gbtree", num_classes=10, )
      clf.load_model("DigitsRecogniser/models/" + files[model])
      prob = clf.predict_proba(preprocess(digit))
      prob = np.array([x[1] for x in prob])
    elif models[model]=="CNN":
      clf = load_model("DigitsRecogniser/models/" + files[model])
      processed_digit = preprocess_for_CNN(digit)
      prob = clf.predict(processed_digit)
      prob = prob[0]
    else:
      clf = joblib.load("DigitsRecogniser/models/" + files[model])
      prob = clf.predict_proba(preprocess(digit))
      prob = prob[0]

    return take_decision(prob, 8, 10)
```

sudoku_gen.cpp

```
#include <iostream>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

#define UNASSIGNED 0

using namespace std;

class Sudoku {
private:
  int grid[9][9];
  int solnGrid[9][9];
  int guessNum[9];
  int gridPos[81];
  int difficultyLevel;
```

```
  bool grid_status;

public:
  Sudoku ();
  Sudoku (string, bool row_major=true);
  void createSeed();
  void printGrid();
  bool solveGrid();
  string getGrid();
  void countSoln(int &number);
  void genPuzzle();
  bool verifyGridStatus();
  void printSVG(string);
  void calculateDifficulty();
  int  branchDifficultyScore();
};

// START: Get grid as string in row major order
string Sudoku::getGrid()
{
  string s = "";
  for(int row_num=0; row_num<9; ++row_num)
  {
    for(int col_num=0; col_num<9; ++col_num)
    {
      s = s + to_string(grid[row_num][col_num]);
    }
  }

  return s;
}
// END: Get grid as string in row major order


// START: Generate random number
int genRandNum(int maxLimit)
{
  return rand()%maxLimit;
}
// END: Generate random number


// START: Create seed grid
void Sudoku::createSeed()
{
  this->solveGrid();

  // Saving the solution grid
  for(int i=0;i<9;i++)
  {
    for(int j=0;j<9;j++)
```

```
    {
      this->solnGrid[i][j] = this->grid[i][j];
    }
  }
}
// END: Create seed grid


// START: Intialising
Sudoku::Sudoku()
{

  // initialize difficulty level
  this->difficultyLevel = 0;

  // Randomly shuffling the array of removing grid positions
  for(int i=0;i<81;i++)
  {
    this->gridPos[i] = i;
  }

  random_shuffle(this->gridPos, (this->gridPos) + 81, genRandNum);

  // Randomly shuffling the guessing number array
  for(int i=0;i<9;i++)
  {
    this->guessNum[i]=i+1;
  }

  random_shuffle(this->guessNum, (this->guessNum) + 9, genRandNum);

  // Initialising the grid
  for(int i=0;i<9;i++)
  {
    for(int j=0;j<9;j++)
    {
      this->grid[i][j]=0;
    }
  }

  grid_status = true;
}
// END: Initialising


// START: Custom Initialising with grid passed as argument
Sudoku::Sudoku(string grid_str, bool row_major)
{
  if(grid_str.length() != 81)
  {
    grid_status=false;
```

```
  return;
}

// First pass: Check if all cells are valid
for(int i=0; i<81; ++i)
{
  int curr_num = grid_str[i]-'0';
  if(!((curr_num == UNASSIGNED) || (curr_num > 0 && curr_num < 10)))
  {
    grid_status=false;
    return;
  }

  if(row_major) grid[i/9][i%9] = curr_num;
  else          grid[i%9][i/9] = curr_num;
}

// Second pass: Check if all columns are valid
for (int col_num=0; col_num<9; ++col_num)
{
  bool nums[10]={false};
  for (int row_num=0; row_num<9; ++row_num)
  {
    int curr_num = grid[row_num][col_num];
    if(curr_num!=UNASSIGNED && nums[curr_num]==true)
    {
      grid_status=false;
      return;
    }
    nums[curr_num] = true;
  }
}

// Third pass: Check if all rows are valid
for (int row_num=0; row_num<9; ++row_num)
{
  bool nums[10]={false};
  for (int col_num=0; col_num<9; ++col_num)
  {
    int curr_num = grid[row_num][col_num];
    if(curr_num!=UNASSIGNED && nums[curr_num]==true)
    {
      grid_status=false;
      return;
    }
    nums[curr_num] = true;
  }
}

// Fourth pass: Check if all blocks are valid
for (int block_num=0; block_num<9; ++block_num)
```

```
  {
    bool nums[10]={false};
    for (int cell_num=0; cell_num<9; ++cell_num)
    {
      int curr_num = grid[((int)(block_num/3))*3 + (cell_num/3)][((int)(block_num%3))*3 + (cell_n
      if(curr_num!=UNASSIGNED && nums[curr_num]==true)
      {
        grid_status=false;
        return;
      }
      nums[curr_num] = true;
    }
  }

  // Randomly shuffling the guessing number array
  for(int i=0;i<9;i++)
  {
    this->guessNum[i]=i+1;
  }

  random_shuffle(this->guessNum, (this->guessNum) + 9, genRandNum);

  grid_status = true;
}
// END: Custom Initialising


// START: Verification status of the custom grid passed
bool Sudoku::verifyGridStatus()
{
  return grid_status;
}
// END: Verification of the custom grid passed


// START: Printing the grid
void Sudoku::printGrid()
{
  for(int i=0;i<9;i++)
  {
    for(int j=0;j<9;j++)
    {
      if(grid[i][j] == 0)
        cout<<".";
      else
        cout<<grid[i][j];
      cout<<"|";
    }
    cout<<endl;
  }
```

```
  cout<<"\nDifficulty of current sudoku(0 being easiest): "<<this->difficultyLevel;
  cout<<endl;
}
// END: Printing the grid


// START: Helper functions for solving grid
bool FindUnassignedLocation(int grid[9][9], int &row, int &col)
{
    for (row = 0; row < 9; row++)
    {
        for (col = 0; col < 9; col++)
        {
            if (grid[row][col] == UNASSIGNED)
                return true;
        }
    }

    return false;
}

bool UsedInRow(int grid[9][9], int row, int num)
{
    for (int col = 0; col < 9; col++)
    {
        if (grid[row][col] == num)
            return true;
    }

    return false;
}

bool UsedInCol(int grid[9][9], int col, int num)
{
    for (int row = 0; row < 9; row++)
    {
        if (grid[row][col] == num)
            return true;
    }

    return false;
}

bool UsedInBox(int grid[9][9], int boxStartRow, int boxStartCol, int num)
{
    for (int row = 0; row < 3; row++)
    {
        for (int col = 0; col < 3; col++)
        {
            if (grid[row+boxStartRow][col+boxStartCol] == num)
                return true;
```

```
        }
    }

    return false;
}

bool isSafe(int grid[9][9], int row, int col, int num)
{
    return !UsedInRow(grid, row, num) && !UsedInCol(grid, col, num) && !UsedInBox(grid, row - ro
}

// END: Helper functions for solving grid


// START: Modified Sudoku solver
bool Sudoku::solveGrid()
{
    int row, col;

    // If there is no unassigned location, we are done
    if (!FindUnassignedLocation(this->grid, row, col))
       return true; // success!

    // Consider digits 1 to 9
    for (int num = 0; num < 9; num++)
    {
        // if looks promising
        if (isSafe(this->grid, row, col, this->guessNum[num]))
        {
            // make tentative assignment
            this->grid[row][col] = this->guessNum[num];

            // return, if success, yay!
            if (solveGrid())
                return true;

            // failure, unmake & try again
            this->grid[row][col] = UNASSIGNED;
        }
    }

    return false; // this triggers backtracking

}
// END: Modified Sudoku Solver


// START: Check if the grid is uniquely solvable
void Sudoku::countSoln(int &number)
{
  int row, col;
```

```
  if(!FindUnassignedLocation(this->grid, row, col))
  {
    number++;
    return ;
  }


  for(int i=0;i<9 && number<2;i++)
  {
      if( isSafe(this->grid, row, col, this->guessNum[i]) )
      {
        this->grid[row][col] = this->guessNum[i];
        countSoln(number);
      }

      this->grid[row][col] = UNASSIGNED;
  }

}
// END: Check if the grid is uniquely solvable


// START: Gneerate puzzle
void Sudoku::genPuzzle()
{
  for(int i=0;i<81;i++)
  {
    int x = (this->gridPos[i])/9;
    int y = (this->gridPos[i])%9;
    int temp = this->grid[x][y];
    this->grid[x][y] = UNASSIGNED;

    // If now more than 1 solution , replace the removed cell back.
    int check=0;
    countSoln(check);
    if(check!=1)
    {
      this->grid[x][y] = temp;
    }
  }
}
// END: Generate puzzle


// START: Printing into SVG file
void Sudoku::printSVG(string path="")
{
  string fileName = path + "svgHead.txt";
  ifstream file1(fileName.c_str());
  stringstream svgHead;
```

```
  svgHead << file1.rdbuf();

  ofstream outFile("puzzle.svg");
  outFile << svgHead.rdbuf();

  for(int i=0;i<9;i++)
  {
    for(int j=0;j<9;j++)
    {
      if(this->grid[i][j]!=0)
      {
        int x = 50*j + 16;
        int y = 50*i + 35;

        stringstream text;
        text<<"<text x=\""<<x<<"\" y=\""<<y<<"\" style=\"font-weight:bold\" font-size=\"30px\">"

        outFile << text.rdbuf();
      }
    }
  }

    outFile << "<text x=\"50\" y=\"500\" style=\"font-weight:bold\" font-size=\"15px\">Difficult
<<this->difficultyLevel<<"</text>\n";
    outFile << "</svg>";

}
// END: Printing into SVG file


// START: Calculate branch difficulty score
int Sudoku::branchDifficultyScore()
{
   int emptyPositions = -1;
   int tempGrid[9][9];
   int sum=0;

   for(int i=0;i<9;i++)
  {
    for(int j=0;j<9;j++)
    {
      tempGrid[i][j] = this->grid[i][j];
    }
  }

   while(emptyPositions!=0)
   {
     vector<vector<int> > empty;

     for(int i=0;i<81;i++)
     {
```

```
            if(tempGrid[(int)(i/9)][(int)(i%9)] == 0)
            {
              vector<int> temp;
              temp.push_back(i);

              for(int num=1;num<=9;num++)
              {
                if(isSafe(tempGrid,i/9,i%9,num))
                {
                  temp.push_back(num);
                }
              }

              empty.push_back(temp);
            }

        }

        if(empty.size() == 0)
        {
          cout<<"Hello: "<<sum<<endl;
          return sum;
        }

        int minIndex = 0;

        int check = empty.size();
        for(int i=0;i<check;i++)
        {
          if(empty[i].size() < empty[minIndex].size())
              minIndex = i;
        }

        int branchFactor=empty[minIndex].size();
        int rowIndex = empty[minIndex][0]/9;
        int colIndex = empty[minIndex][0]%9;

        tempGrid[rowIndex][colIndex] = this->solnGrid[rowIndex][colIndex];
        sum = sum + ((branchFactor-2) * (branchFactor-2)) ;

        emptyPositions = empty.size() - 1;
      }

      return sum;

}
// END: Finish branch difficulty score


// START: Calculate difficulty level of current grid
void Sudoku::calculateDifficulty()
```

```
{
  int B = branchDifficultyScore();
  int emptyCells = 0;

  for(int i=0;i<9;i++)
  {
    for(int j=0;j<9;j++)
    {
        if(this->grid[i][j] == 0)
           emptyCells++;
    }
  }

  this->difficultyLevel = B*100 + emptyCells;
}
// END: calculating difficulty level


// START: The main function
int main(int argc, char const *argv[])
{
  // Initialising seed for random number generation
  srand(time(NULL));

  // Creating an instance of Sudoku
  Sudoku *puzzle = new Sudoku();

  // Creating a seed for puzzle generation
  puzzle->createSeed();

  // Generating the puzzle
  puzzle->genPuzzle();

  // Calculating difficulty of puzzle
  puzzle->calculateDifficulty();

  // testing by printing the grid
  puzzle->printGrid();

  // Printing the grid into SVG file
  string rem = "sudokuGen";
  string path = argv[0];
  path = path.substr(0,path.size() - rem.size());
  puzzle->printSVG(path);
  cout<<"The above sudoku puzzle has been stored in puzzles.svg in current folder\n";
  // freeing the memory
  delete puzzle;

  return 0;
}
// END: The main function
```