Recognition of electrical components in circuits

Alpen-Adria-Universität Klagenfurt

UNIVERSITÄT
KLAGENFURT

Alexandra Wissiak, Daniel Holzfeind, Muamer Hrncic

February 2022

Contents

1 Introduction

Electrical engineers often use to draw sketches of circuits by hand on paper. In most project or prototyping phases it is common to share ideas among colleagues. Therefore little sketches become big really quickly. If sketches lead to some useful circuit or parts of some bigger circuit, someone has to redraw the circuit in some suitable software, which is redundant work. Another example is the following: An old robotic cell has to be modernized and it is necessary to re-use and to extend the hardware. Every good control cabinet has some kind of electrical hardware plan. Now imagine there are no digital versions of the plan or there are only read-only data files. In both examples a sketch recognition program would be very useful to automate tasks.

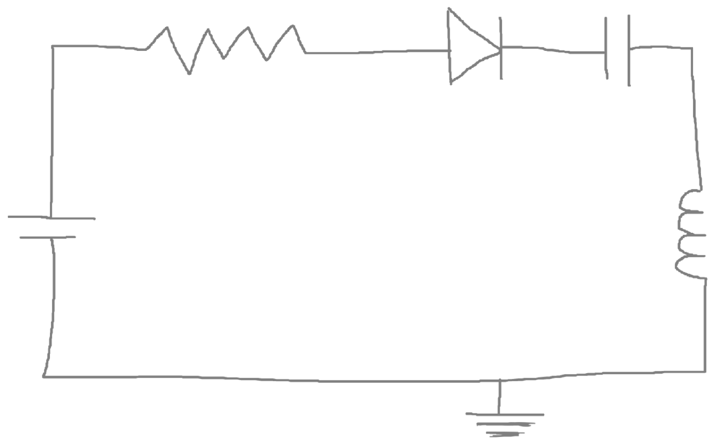


Figure 1: Simple electrical circuit

Any valuable software for this problem must be able to perform segmentation and recognition of single components and it has to recognize the connections between the components. It must be able to recognize quickly or at least in a tolerable time, have high accuracy and it should be easy to extend the software to new components. There are several approaches to the problem using different techniques. This project uses ideas and approaches from [1] combined with feature detection described [2] for component recognition and [3] for line segment detection. Most of the methods used for the implementation are described in [4] and other sources. The code is written in **Python** and uses the library **open-cv** and the Python binding **ocrd-fork-pylsd**. The code works on several platform like **Linux and Windows**,

2 The problem

The problem can be described as follows. Given a hand drawn sketch as input, the aim is to recognize all components and connecting wires correctly and to create appropriate output, like a connection table, for further tasks. The recognition process involves steps combining techniques of image processing and data mining. The main steps are shown in 2. Image preprocessing transforms the image into appropriate data for further steps. Segmentation of component symbols and connection wires between them splits the image into separate parts to enhance recognition. These are crucial steps since if the objects in the image are not segmented properly the recognition process will not be able to identify the components. A poor preprocessing or segmentation process can cause multiple classifications for a single components or it classifies connection lines as a component symbols. The components have to be classified correctly by the recognition process and as fast as possible to give a quick response to the user.

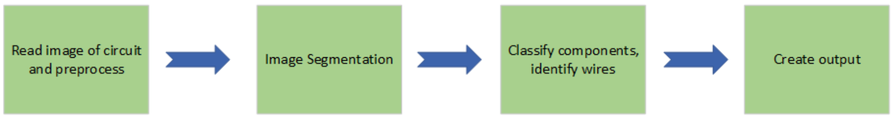
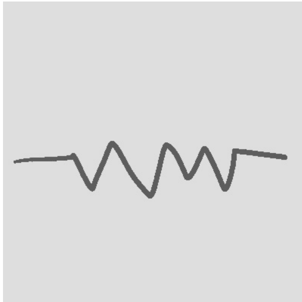


Figure 2: Component recognition process

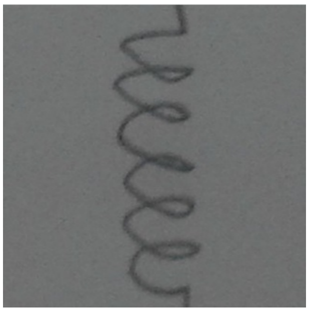
3 Datasets

The datasets used for the model include images of diodes, resistors and inductors. Let us denote these three components as the **type-1-components**. For every component there are 100 images of the single component as can be seen in the following figures.

(a) Diode



(b) Resistor



(c) Inductor

Figure 3: Electrical components

These datasets are used to train and to test a multiclass support vector machine (MC-SVM) for object recognition and classification. Recognition of line components like source, capacitor and ground, denoted by **type-0-components** is done in another way, which will be described in further sections, so there is no train or test data for these kind of components.

4 System Workflow

The overall workflow for the recognition process includes the following steps:

- Data preparation: Read and resize the image.
- Image preprocessing: Transform the image into separate parts for **type-0-components** and **type-1-components** segmentation and recognition.
- Segmentation: Recognize and classify **type-0-components** and store them.
- Feature extraction: Remove line objects and detect lines to get an image with the remaining components, namely the **type-1-components**.
- Classification: Contour search of possible remaining components to get the bounding rectangles. Classify the remaining components with the help of the found rectangles and the SVM.

- Output: Show the image of the circuit classified components and with highlighted lines of connected components.

The procedure involving all steps of the recognition process is in the file **main.py**:

```
import cv2 as cv
import imutils
import matplotlib.pyplot as plt

from electrical_comp_SVM import predict
from image_processing import image_preproc
from image_processing import get_line_comp
from image_processing import draw_classified_comp
from image_processing import remove_lines
from image_processing import get_bounding_rect
from image_processing import hl_lines_and_connect

def get_comp_templates():
    ...

if __name__ == "__main__":
    ###################################################
  ...
    ###################################################
    #Read image
    img_circuit = cv.imread("examples/circuit.png")

    # Resize image.
    img_circuit = imutils.resize(img_circuit, width=640)
    ...
    ###################################################
    # First things first
    # Preprocess image and get image variants for the different steps
    thinned, thres_line, thres_comp, endpoints = image_preproc(img_circuit, blurkernelsize=7, blocksize=7, c=2, morphIterations=1, kernelsize=3)
    ...
    ###################################################
    # First step:
    # 1) Try to detect line components source, ground, capacitor if possible.
    # 2) Store them to the global placeholder for all components
    comp_boxes = get_line_comp(endpoints)
    ...
    ###################################################
    # Second step:
    # 1) Remove the line objects.
    # 2) Try to detect lines.
    # 3) Remove them from threshold image.
    # 4) Apply morphological closing to the image.
    # 5) Get an image which contains diodes, resistors and inductances,
    #   if present in the orig. image of the circuit.

    img_reduced = remove_lines(thres_line, comp_boxes)
    ...
    ###################################################
    # Third Step:
    # 1) Try to find the contours of possible remaining components
    # 2) Get the bounding rectangles of the found contours w.r.t. to a minimum area

```

```python
    bndg_rectangles = get_bounding_rect(img_reduced)
    ###################################################

    # Fourth Step:
    # Try to classify the remaining components in the reduced threshold image
    # with the help of the found bounding rectangles, which play the role as region of interest
    comp_boxes = predict(thres_comp, bndg_rectangles, comp_boxes)
  ...
    ###################################################

    # Last step:
    fig6 = plt.figure(num=6, figsize=(14,8))
    # Highlight connected lines ang get connecting components
    img_out, connections = hl_lines_and_connect(
        img_circuit, thres_comp, comp_boxes)
    # Plot the image of the classified components
    plt.subplot(111),plt.imshow(img_out),plt.title("Classified components and highlighted lines")
    fig6.savefig("output/4_output.png")
```

Listing 1: main.py

4.1 Data preparation and image preprocessing

The data preparation and image preprocessing step is done in the following part. The image of the circuit is read and resized in lines 9 and 12. The preprocessing step is done in line 17 and returns the images and data for further steps:

```python
1 import cv2 as cv
2 import imutils
3 ...
4 if __name__ == "__main__":
5     ###################################################
6     ...
7     ###################################################
8     #Read image
9     img_circuit = cv.imread("examples/circuit.png")
10
11     # Resize image.
12     img_circuit = imutils.resize(img_circuit, width=640)
13     ...
14     ###################################################
15     # First things first
16     # Preprocess image and get image variants for the different steps
17     thinned, thres_line, thres_comp, endpoints = image_preproc(img_circuit, blurkernelsize=7, blocksize=7, c=2, morphIterations=1, kernelsize=3)
18     ...
```

Image preprocessing is done in the following method in the file **image_processing.py**. The threshold image for line component segmentation is created in line 7 denoted by **thres_line**. For **thres_line** morphological operations of dilating and thinning are applied to enhance the classification process for **type-0-components**. The method of thinning is described in [5]. The image for **type-1-components** classification is also the original threshold image, denoted by

thres_comp, but with no further operations, since thinned lines are not good for feature extraction. The variable **endpoints** contains the coordinates of the line endpoints.

```
def image_preproc(src, blurkernelsize=9, blocksize=7, c=2, morphIterations=2, kernelsize=3):
    #Convert source image to grayscale
    gray = cv.cvtColor(src, cv.COLOR_BGR2GRAY)
    #Apply Gaussian filter to denoise image with a 9x9 kernel
    img = cv.GaussianBlur(gray, (blurkernelsize, blurkernelsize), 0)
    #Apply adaptive thresholding
    thres_line = cv.adaptiveThreshold(
        img, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, blocksize, c)
    #Make copy of threshold image
    thres_comp = thres_line.copy()
    #Get kernel for morphological  operations
    kernel = cv.getStructuringElement(cv.MORPH_RECT, (kernelsize, kernelsize))
    #Dilate image
    dilated = cv.dilate(thres_line, kernel, iterations=morphIterations)
    #Thin image with Zhang-Suen thinning algorithm
    thinned = ZhangSuen_thin(dilated)
    #Save skeleton of image
    cv.imwrite("stages/endpoints.png", thinned)
    #Save threshold image
    cv.imwrite("stages/threshold.png", thres_comp)
    #Get skeleton points
    endpoints = get_endpoints(thinned)
    return thinned, thres_line, thres_comp, endpoints
```

The output of the first steps is shown in 4.

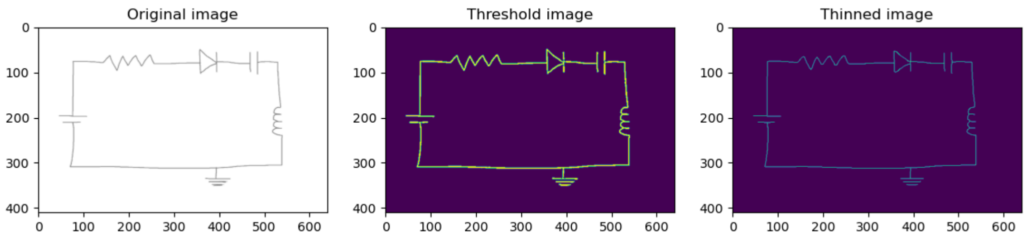


Figure 4: Read image and preprocess

4.2 Segmentation, Classification of type-0-components

The segmentation process is done with the method **get_line_comp(endpoints)**

```
import cv2 as cv
import imutils
...
if __name__ == "__main__":
    ####################################################
    ...
    ####################################################
    # First step:
    # 1) Try to detect line components source, ground, capacitor if
     possible.
    # 2) Store them to the global placeholder for all components
    comp_boxes = get_line_comp(endpoints)
    ...
    ####################################################
    ...
```

which first determines the coordinates of all vertical and horizontal lines of possible satisfying proper conditions for belonging to the class of **type-0-components**. After this the bounding rectangles of the components are determined and the classified objects are stored to the list **comp_boxes** as can be seen in 5a and 5b. The result of step 1 is shown in 6.

```

def get_line_comp(endpoints):
    #Get the endpoints of the vertical lines and the horizontal
    lines
    vertical_lines, horizontal_lines = get_lines(endpoints)
    #Try to detect vertical and horizontal components and store the
     bounding boxes
    #to the global placeholder for all components
    comp_boxes = get_line_comp_boxes(
        vertical_lines) + get_line_comp_boxes(horizontal_lines)
    return comp_boxes
```

(a) Threshold picture (b) Removed line components

Figure 5: Step 2

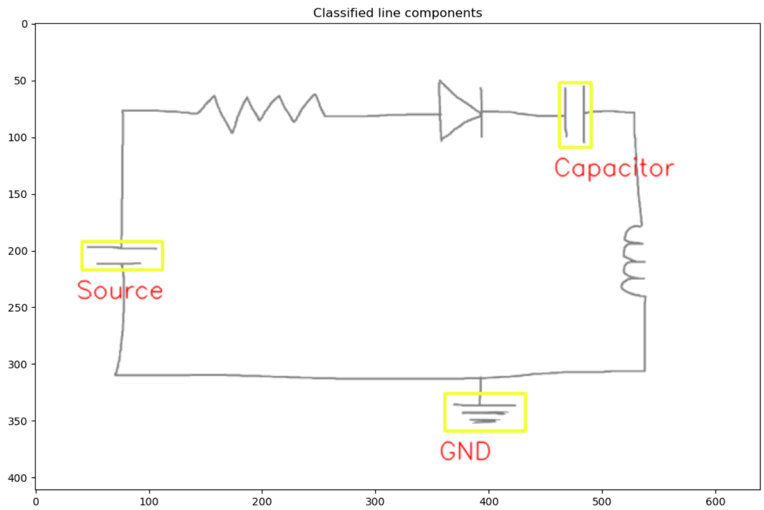


Figure 6: Classification of line components

4.3 Feature extraction

The second step is preparing the image without line components for feature extraction.The connecting lines of the components disturb the process feature extraction so they have to be removed.

```
import cv2 as cv
import imutils
...
if __name__ == "__main__":
    ...
    ####################################################
    # Second step:
    # 1) Remove the line objects.
    # 2) Try to detect lines.
    # 3) Remove them from threshold image.
    # 4) Apply morphological closing to the image.
    # 5) Get an image which contains diodes, resistors and
    inductances,
    #   if present in the orig. image of the circuit.
    img_reduced = remove_lines(thres_line, comp_boxes)
    ...
    ####################################################
    ...
```

Lines are removed with the method **remove_lines** which is implemented in the following way:

```
def remove_lines(thres_line, comp_boxes, vertlow=80, verthigh=100,
    horlow=10, horhigh=170, kernelsize=11):
    #Remove found line components from threshold image
    for ((x, y, w, h), _) in comp_boxes:
        thres_line[y:y+h, x:x+w] = 0

```

```
    cv.imwrite("stages/remove_lines.png", thres_line)

    #Detect all lines in the reduced threshold image using the line segment detection module from pylsd
    lines = lsd(thres_line)
    #Remove all lines which satisfy a certain angle condition.
    for line in lines:
        #Save endpoint coordinates of the line
        xs, ys, xe, ye, _ = line
        #Calculate delta x
        dx = xs - xe
        #Calculate delta x
        dy = ys - ye
        #Get the angle of the line in the correct quadrant.
        #We look for angles in degrees.
        ang = np.abs(np.rad2deg(np.arctan2(dy, dx)))

        #If the angle of a line is not in the strip
        #(horlow,horhigh) it is maybe a horizontal line
        #or if the angle of a line is in the strip
        #(vertlow, verthigh) it is maybe a vertical line.
        if (ang < horlow or horhigh < ang or
           (vertlow < ang < verthigh)):
            #If it is a horizontal line or a vertical,
            #remove the line by setting
            #all pixel on the line to black.
            cv.line(thres_line, (int(xs), int(ys)),
                    (int(xe), int(ye)), (0, 0, 0), 6)

    #Get kernel for the closing operation
    kernel = cv.getStructuringElement(cv.MORPH_RECT, (kernelsize, kernelsize))

    #Return the closed image
    return cv.morphologyEx(thres_line, cv.MORPH_CLOSE, kernel)
```

The result of this step is shown in 7. The thresholds of the remaining components are the non-black pixels.

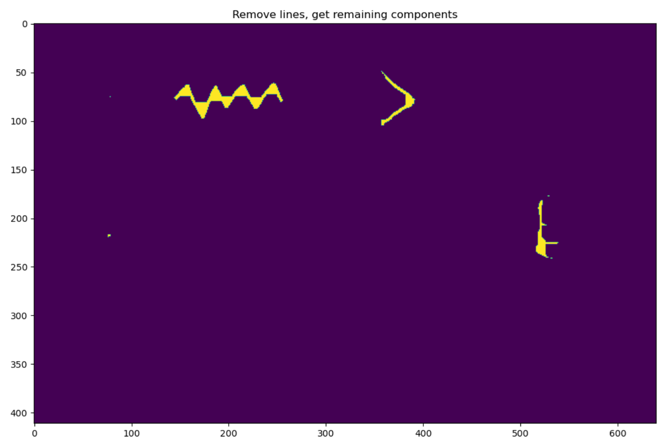


Figure 7: Removing lines for feature extraction

4.4 Classification of type-1-components

4.4.1 Overall Process

Classification of possible remaining components is done by contour search in order to get the bounding rectangles.

```python
import cv2 as cv
import imutils
...
if __name__ == "__main__":
    ...
    ####################################################
    # Third Step:
    # 1) Try to find the contours of possible remaining components
    # 2) Get the bounding rectangles of the found contours w.r.t. to a minimum area
    bndg_rectangles = get_bounding_rect(img_reduced)

    ####################################################

    # Fourth Step:
    # Try to classify the remaining components in the reduced threshold image
    # with the help of the found bounding rectangles, which play the role as region of interest
    comp_boxes = classify(thres_comp, bndg_rectangles, comp_boxes)
    ####################################################
    ...
```

The method **get_bounding_rect** first searches for contours not smaller than a certain area. After that the bounding rectangles of all contours which are large enough are calculated and returned for the classification process.

```python
def get_bounding_rect(img_reduced, border=10, minarea=75):
    #List to hold all bounding rectangles of eventuall componentes
    bndg_rectangles = []
    # Find remaining parts of components through contours in reduced image
    contours = cv.findContours(
        img_reduced, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)[0]
    #Iterate over all contours
    for contour in contours:
        #Calculate area of contour
        area = cv.contourArea(contour)
        #If it is not large enough discard the contour
        if area < minarea:
            continue
        else:
            #Otherwise get the bounding rectangle
            x, y, w, h = cv.boundingRect(contour)
            #Get the longer side of the rectangle
            longer_side = max(w, h)
            #Recalculate the x and y coordinates of
            #the bounding rectangle w.r.t the longer side
            x = int((2 * x + w - longer_side)/2)
            y = int((2 * y + h - longer_side)/2)
            #Append the bounding rectangle, by setting a new origin
            #and new margin, defined by the parameter border
            bndg_rectangles.append(
                [x - border, y - border, x + border + longer_side, y + border + longer_side])

```

```
    #Return all created rectangles
    return bndg_rectangles
```

Afterwards the possible components are classified with the help of the found rectangles and the SVM. This is done by the method **classify** in the file **electrical_comp_SVM.py**. As mentioned before we extract **Histogramm of Orientated Gradients features** for the classification. The HOG descriptor considers the structure or the shape of an object. It provides the edge direction, which is done by extracting the gradient and orientation of the edges locally which means, the image is broken down into small regions. For each region, the gradients and orientation are calculated. As final step the descriptor generates a histogram for every region using the gradients and orientations of the pixel values.

```
def classify(img, rects, boxes):
    #Load data of the SVM
    svm = cv.ml.SVM_load("data/02_svm/trained.dat")
    #Create HOG descriptor for feature detection
    hog = get_HOG_descriptor();
    #Save number of classes
    nrClasses = svm.getTermCriteria()[0]
    #Save number of variable
    nrVars = svm.getVarCount()

    #Iterate over all reactangles
    for xs, ys, xe, ye in rects:
        #Get region of interest and resize the image to 100x100,
        #using cubic interpolation
        region = cv.resize(img[ys : ye, xs : xe], (100,100),
        interpolation = cv.INTER_CUBIC)
        #Compute HOG features
        descriptors = hog.compute(region)
        #Predict the class of the component
        _, result = svm.predict(
        np.array(descriptors, np.float32).reshape(-1,nrVars))
        #Store predicted class
        cl = int(result[0][0]) + nrClasses
        boxes.append([[int(xs), int(ys), int(xe - xs),
        int(ye - ys)],cl])
    return boxes
```

4.4.2 Support Vector Machine Training and Testing

As mentioned before we use a Multi Class Support Vector Machine to classify the components diodes, resistors and inductors. Before we were able to use this model, we had to train the model with the train data from above and the performance was tested. Training can be performed within the file **train.py** which is implemented as follows. First we load the paths to the train data and create the labels. Then the SVM is trained with the data calling the method **train_component_SVM()** from the file **electrical_comp_SVM.py** and afterwards a simple test is performed.

```python
import cv2 as cv
import numpy as np
from electrical_comp_SVM import load_data, train_component_SVM, get_HOG_descriptor

if __name__ == '__main__':

    #Path to the train data
    train_path = "data/01_train"
    #Load the train data and the labels
    train, labels  = load_data(train_path)
    #Train the SVM. Caution! This takes a while and demands a lot of ressources.
    comp_svm = train_component_SVM(train, labels)
    #Test the trained SVM with a test image
    test = cv.imread("data/circuit.png",0)
    #Create a HOG descriptor
    hog = get_HOG_descriptor();
    #Compute the HOG descriptors of the test image
    test_hog = hog.compute(test)
    #Classify the components
    re = comp_svm.predict(np.array(test_hog ,np.float32).reshape(-1, comp_svm.getVarCount()))
    #Print the result of the classification
    print(re)
```

The method to train the SVM is implemented in the following way. We create a global HOG descriptor and initialize the SVM. We use the radial basis function kernel because it behaved well in both training and testing.

```python
def train_component_SVM(train, labels, resize = 100, kernelsize = 9, threshval = 11, const = 1):
    #Create a Hog descriptor
    hog = get_HOG_descriptor();
    #List to hold the calculated HOG descriptors
    descriptors = []
    #Create an instance of a support vector machine
    svm = cv.ml.SVM_create()
    #Set kernel type to radial basis function
    svm.setKernel(cv.ml.SVM_RBF)
    #Set SVM type as C-Support Vector Classification type.
    #n-class classification (n     2), allows imperfect separation
    #of classes with penalty multiplier C for outliers.
    svm.setType(cv.ml.SVM_C_SVC)
    ...
```

Every training image is resized and blurred. After this a threshold image is created and HOG features are extracted and appended to the global training data set.

```python
def train_component_SVM(train, labels, resize = 100, kernelsize = 9, threshval = 11, const = 1):
    ...
    #Actual image
    act_img = 0
    #Collect train data
    for path in train:
        #Read image as Grayscale image
        img = cv.imread(path, 0)
        #Resize the image to get a usefull amount of features
        resized = cv.resize(img, (resize,resize), interpolation = cv.INTER_CUBIC)

```

```
        #Blur the image to reduce noise
        blurred = cv.GaussianBlur(resized,
        (kernelsize,kernelsize) , 0)
        #Apply adaptive thresholding to the blurred image
        threshold = cv.adaptiveThreshold(blurred, 255,
        cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV,
        threshval, const)
        #Save the threshold image
        cv.imwrite('data/00_threshold_img/' + str(act_img)
        + '.jpg', threshold)
        #Compute the hog descriptors and append them to the
        #overall list
        descriptors.append(hog.compute(threshold))
    ...
```

Since it is desired to get rotation invariance by rotations of $90°$ the latter process is performed for every of the 4 rotations of every train image.

```
def train_component_SVM(train, labels, resize = 100, kernelsize =
    9, threshval = 11, const = 1):
    ...
    for path in train:
    ...
        #Get image shape
        rows,cols = resized.shape
        #Rotate the image three time by 90 degrees and repeat
        #above process.
        #This ensures kind of a rotation invariance in the case of
        #normal circuits.
        #This means that components are aligned either vertical
        #or horizontal.
        for i in [1,2,3]:
            #Get the rotation matrix which rotates the picture by
            #i*90 degrees, around the image center
            #and keep the original scale
            rotmatrix  = cv.getRotationMatrix2D((cols/2,rows/2),
            i*90, 1)
            #Save the rotated picture
            rotated = cv.warpAffine(threshold, rotmatrix,
            (cols,rows))
            #Compute the hog descriptors of the rotated image
            #and append them to the overall list
            descriptors.append(hog.compute(rotated))
            #Store the image
            cv.imwrite('data/00_threshold_img/'+
            str(act_img)+'_'+str(i)+'.jpg', rotated)
        #Increase image count
        act_img += 1
```

After collecting the data the actual SVM is trained and its data is stored for later use.

```
def train_component_SVM(train, labels, resize = 100,
kernelsize = 9, threshval = 11, const = 1):
    ...
    print("Threshold pictures are created.")
    print("SVM-Training starts now.")
    svm.trainAuto(np.array(descriptors, np.float32), cv.ml.
    ROW_SAMPLE, np.array(labels))
    print("SVM is trained.")
    print("SVM data will be stored to: data/02_svm/trained.dat")
    svm.save('data/02_svm/trained.dat')
```

Every training process is followed by a test process to measure the accuracy of the model. In our case the method for testing is implemented in **test.py**. First we load the paths to the test data and create the labels. Then the SVM is test with the data calling the method **test_component_SVM()** from the file **electrical_comp_SVM.py**. From this step we get the number of correctly classified components and the total number of classified components for every class. With this we calculate the precision and the recall of the trained SVM.

```python
from electrical_comp_SVM import load_data, test_component_SVM
if __name__ == '__main__':
    #Path to the test data
    image_path = "data/02_test"
    #Load the train data and the labels
    test, labels = load_data(image_path)
    nrLabels = len(labels)
    #Classify the test data images and get the correctly
    #classified components
    #and the total classified components
    class_corr, class_total = test_component_SVM(test, labels)
    #Calculate the precision and the recall of the classification
    for i in range(len(class_corr)):
        #The precision for class i is the ratio of
        #true positives for class i and total classified objects
        #in class i
        precision = class_corr[i] / float(class_total[i])
        #The recall for class i is the ratio of true positives for
        #class i and the total number of classes which is 3 in
        #this case, since the three line components are classified
        #with other methods
        recall = class_corr[i] / float((nrLabels/3))
        #print the result
        print("{} \nprecision :{:.3f}\trecall :{:.3f}".format(
            i, precision, recall,))
        print("{}\t{}".format(class_corr[i], nrLabels/3))
```

These performance metrics are calculated using a test set which is equal to 20% of the total dataset. Each test sample was rotated three times by $90°$ to obtain different rotations of the components so there were 80 test samples of each component label. The accuracy and the response of the model can be seen in 8.

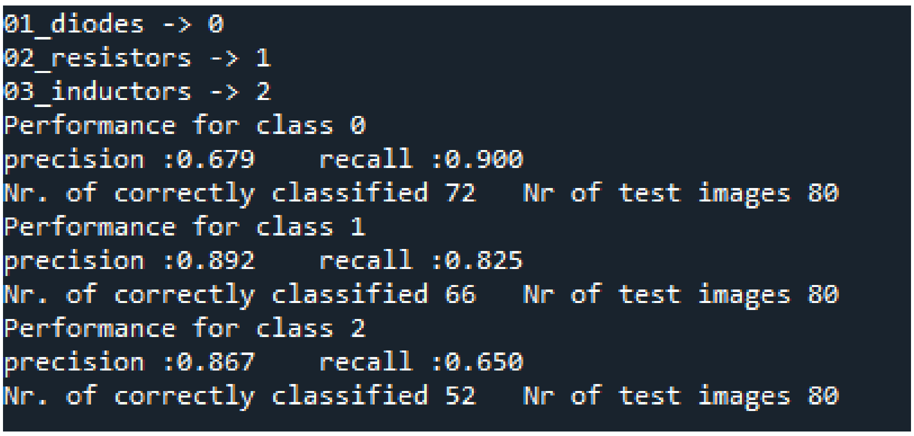


Figure 8: Test result

One can observe that the accuracy for diodes is a lot smaller than for resistors and inductors, allthough the shape of resistors and inductors is similar. Nevertheless the accuracy is not that bad.

4.5 The last steps

The last two steps consist of drawing all classified components and highlighting the connection lines of the components, which is shown in 9 and 10. Beside that we store coordinates of the endpoints of the connection lines to the components into the variable **connections**, which can be used for further processing.

```
import cv2 as cv
import imutils
...
if __name__ == "__main__":
    ...
    ####################################################
    # Fourth Step:
    ...
    # Draw the rectangles of all classified components
    draw_classified_comp(img_circuit, comp_boxes, 1)
    ...
    ####################################################

    # Last step:
    ...
    # Highlight connected lines ang get connecting components
    img_out, connections = hl_lines_and_connect(
        img_circuit, thres_comp, comp_boxes)
    # Plot the image of the classified components
    plt.subplot(111),plt.imshow(img_out),plt.title("Classified
    components and highlighted lines")
    fig6.savefig("output/4_output.png")
    ####################################################
```

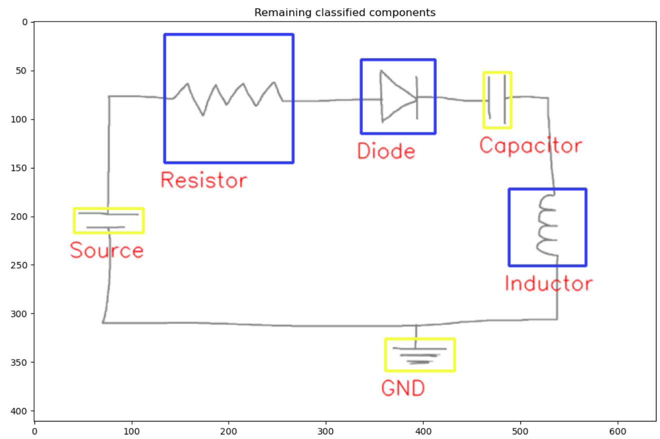


Figure 9: All classified components

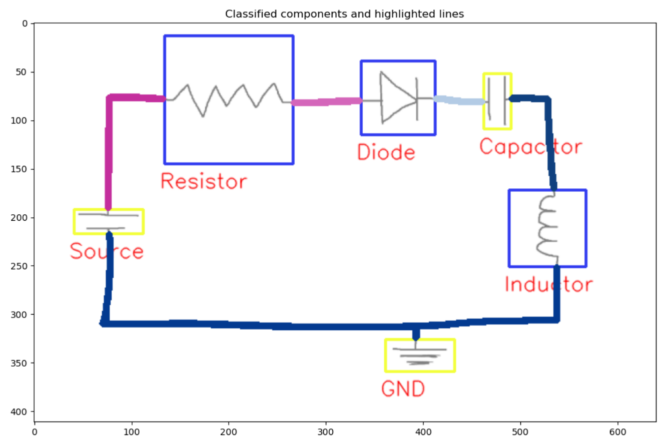


Figure 10: Highlighted connection lines

5 Results and Discussion

The model is limited to six components due to the data requirements for each component. Nevertheless a dataset that includes more components can expand supported components shapes, so it would be possible to handle more complex circuit schematics. The methods are implemented in a such a way to have flexibility in parameter selection. Depending on the circuit sketch the results vary from really good to sufficiently good. Choosing different parameters can improve the classification process a lot. The main challenges are segmentation and classification and most errors and false classifications come from poor preprocessing of the images and the segmentation processes. Further improvements can be achieved through bigger data sets of train data and more homogenous training images, which means that the images used for training should have lookalike contrast and shape. The following pictures show classification results for different circuits and single component drawings.

Pictures 11 and 12 show that the choice of parameters can be important. Since the circuit used is just the rotated from latter sections usually there should be no problems, but since the image is resized in a different way, that is we stretch the original width instead of shrinking it, we get different results. The diode is classified as source component and in the right upper corner the algorithm found a diode but there is no diode.

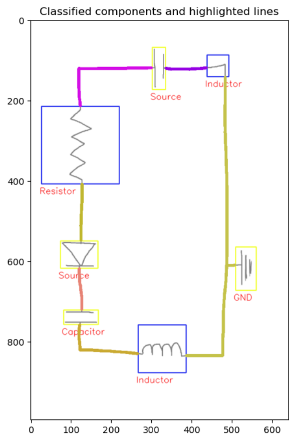


Figure 11: Default reprocessing parameters

The preprocessing parameters used for this circuit are

```python
import cv2 as cv
import imutils
...
if __name__ == "__main__":
    ...
    ###################################################
    # First things first
    # Preprocess image and get image variants for the different steps
    #params for circuit, circuit_2, circuit_3
    #thinned, thres_line, thres_comp, endpoints = image_preproc(img_circuit, blurkernelsize=7, blocksize=7, c=2, morphIterations=1, kernelsize=3)
    #params for circuit_1
    thinned, thres_line, thres_comp, endpoints = image_preproc(img_circuit, blurkernelsize=11, blocksize=5, c=2, morphIterations=2, kernelsize=3)

    ###################################################
```

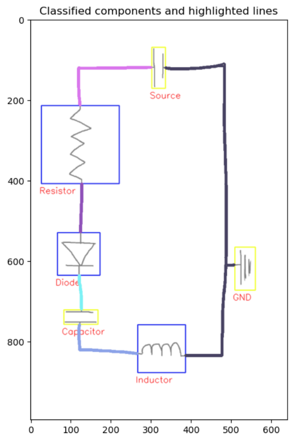


Figure 12: Better results

For the next two circuits shown in 13 and 14 the model performed good with default parameters. In the first the circuit the model classified an output port as diodes, which is alright because the model does not know about the shape of port, which should be a circle. This deficit can be removed by applying Hough Circle detection before the line component detection step, in order to remove circle object and to classify them. Hough transform is described in [4], [6], [7] and other sources.

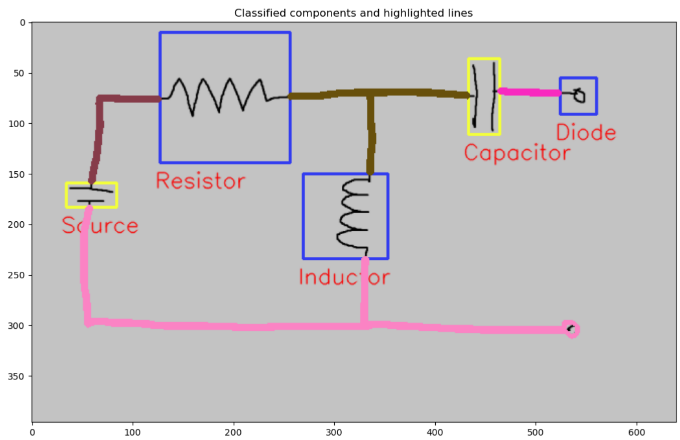


Figure 13: Results for circuit_2.png

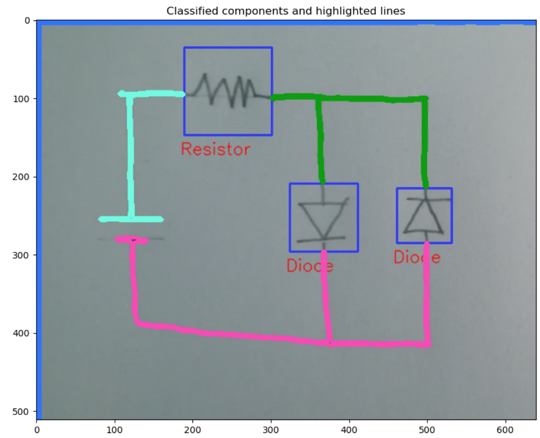


Figure 14: Results for circuit_3.png

For the next example in 15 the resize factor has to be changed to 500 instead of 640. The remaining parameters are default. One can observe that the classification boxes of the parallel connection of the resistor and the inductor are overlapping, which is just a visual drawback but everything else is correct.

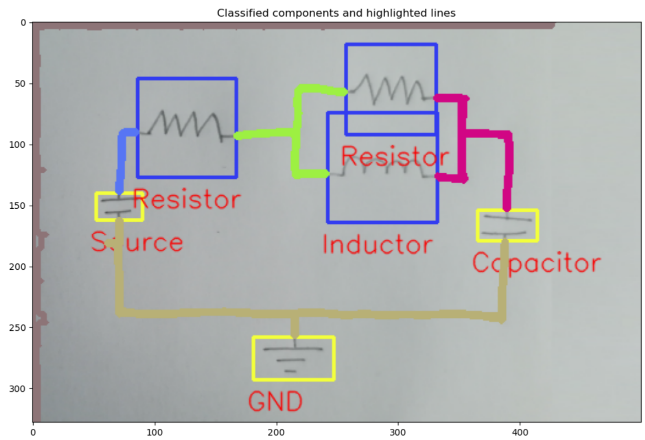


Figure 15: Results for circuit_4.png

Now to the last example shown in 16 which contains random drawings of single components not connected with each other. One can observe that only one diode is correctly classified, which speaks for the precision score of the SVM. There are also other falsely classified components like the $45°$ resistor in the upper right, which is clear, since the SVM is not trained for arbitrary rotation. To get this kind of invariance on should use Fourier descriptors or SIFT descriptors (scale invariant feature transform) described in [8].The inductor and the other two resistors in the lower half are also not classified correctly. As mentioned to get good results the training of the SVM and the segmentation processes are crucial. On the other hand it is impossible to find the perfect set of parameters and to catch any error because there is no uniform handwriting and images differ in sizes. Maybe it is better to provide a method for manual classification of such components.

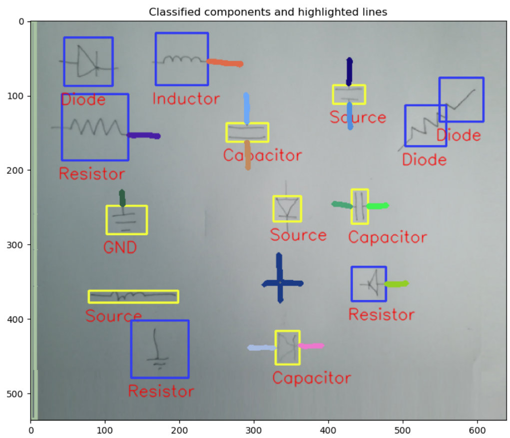


Figure 16: Results for single_comp.png

6 Conclusion and final remarks

The problem of circuit recognition can be solved and it is a nice combination of image processing techniques and data mining. This project implements different methods to recognize hand drawn circuits with up to 6 different components. Depending on the handwriting of the circuit artist the implementation delivers good or not so good classification results. The code is flexible in parameter selection and the model can be extended for a more accurate recognition process of different components and circuits. Further tasks would be to get better precision and to extend the model for more components.

References

[1] Yu-Chen Liu and Yao Xiao. Circuit sketch recognition. 2014.

[2] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, 2, 06 2005.

[3] Rafael Gioi, Jeremie Jakubowicz, Jean-Michel Morel, and Gregory Randall. Lsd: A fast line segment detector with a false detection control. *IEEE transactions on pattern analysis and machine intelligence*, 32:722–32, 04 2010.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

[5] Wei Chen, Lichun Sui, Zhengchao Xu, and Yu Lang. Improved zhang-suen thinning algorithm in binary line drawing applications. *2012 International Conference on Systems and Informatics, ICSAI 2012*, 05 2012.

[6] Virendra Yadav, Saumya Batham, Anuja Acharya, and Rahul Paul. Approach to accurate circle detection: Circular hough transform and local maxima concept. pages 1–5, 02 2014.

[7] Allam Shehata, Sherien Mohammad, Mohamed Abdallah, and Mohammad Ragab. A survey on hough transform, theory, techniques and applications. 02 2015.

[8] Dawid G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision, 20–25 September, 1999, Kerkyra, Corfu, Greece, Proceedings*, volume 2, pages 1150–1157, 1999.