Local Binary Pattern Features for Texture Classification

A how-to on enhancing textures using LBP.

Muhammad Ardi · Follow

Published in Becoming Human: Artificial Intelligence Magazine
15 min read · Jan 11

Listen Share More

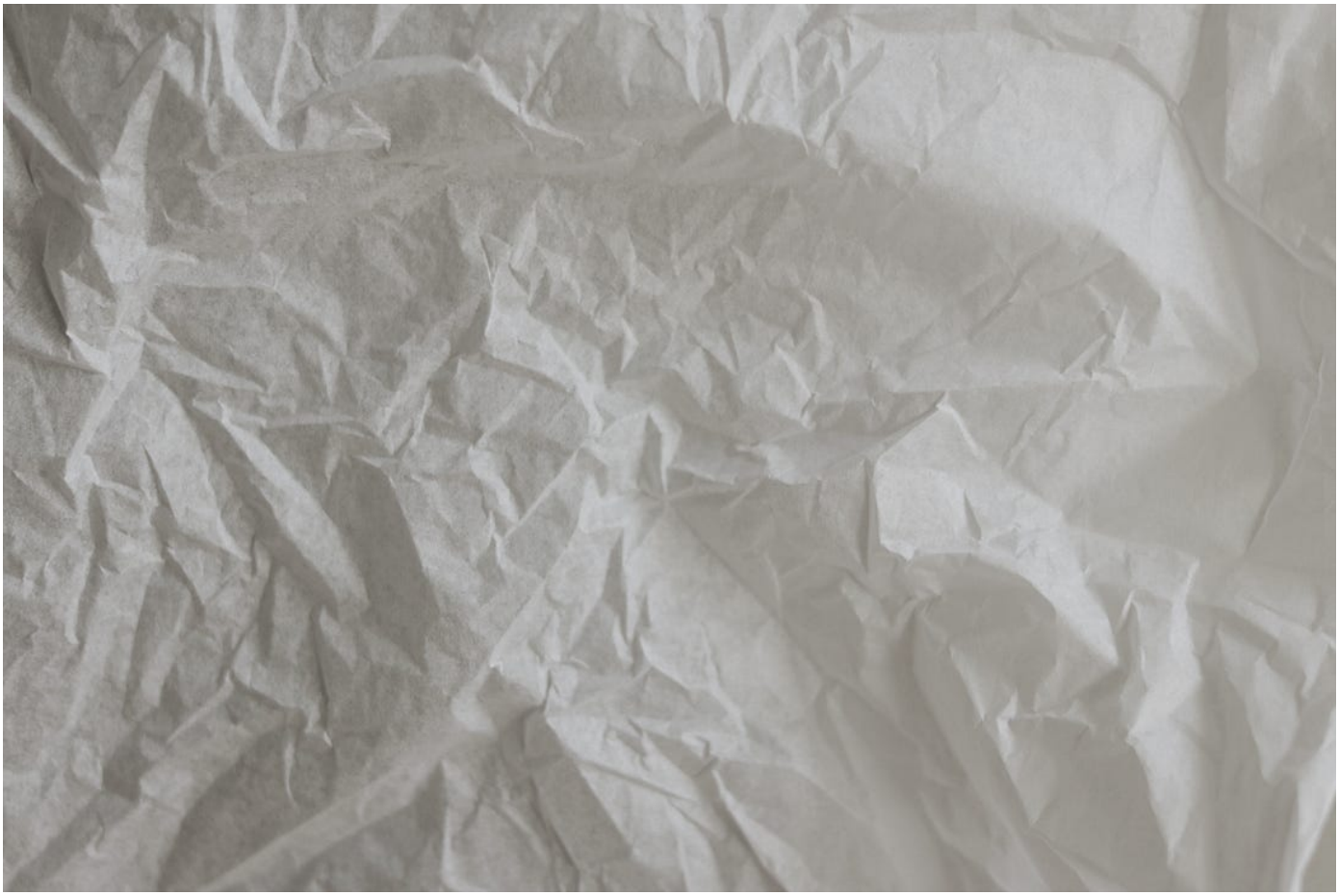


Photo by Jessica Delp on Unsplash

It's been nearly a year since my last post on Medium. Starting from today, in this early 2023, I plan to write more codes as well as blog posts. *Hopefully* I will be consistent with my words, lol :)

Introduction

In this article I want to prove that even though deep learning is the current state-of-the-art approach on classifying images, yet this does not necessarily mean that handcrafted features combined with traditional machine learning is not that powerful. In several cases (including this one) the traditional approach might still be preferred thanks to its simplicity and its ability to work on relatively small dataset. Furthermore, the limited computational budget might also be another reason not to use deep learning [1].

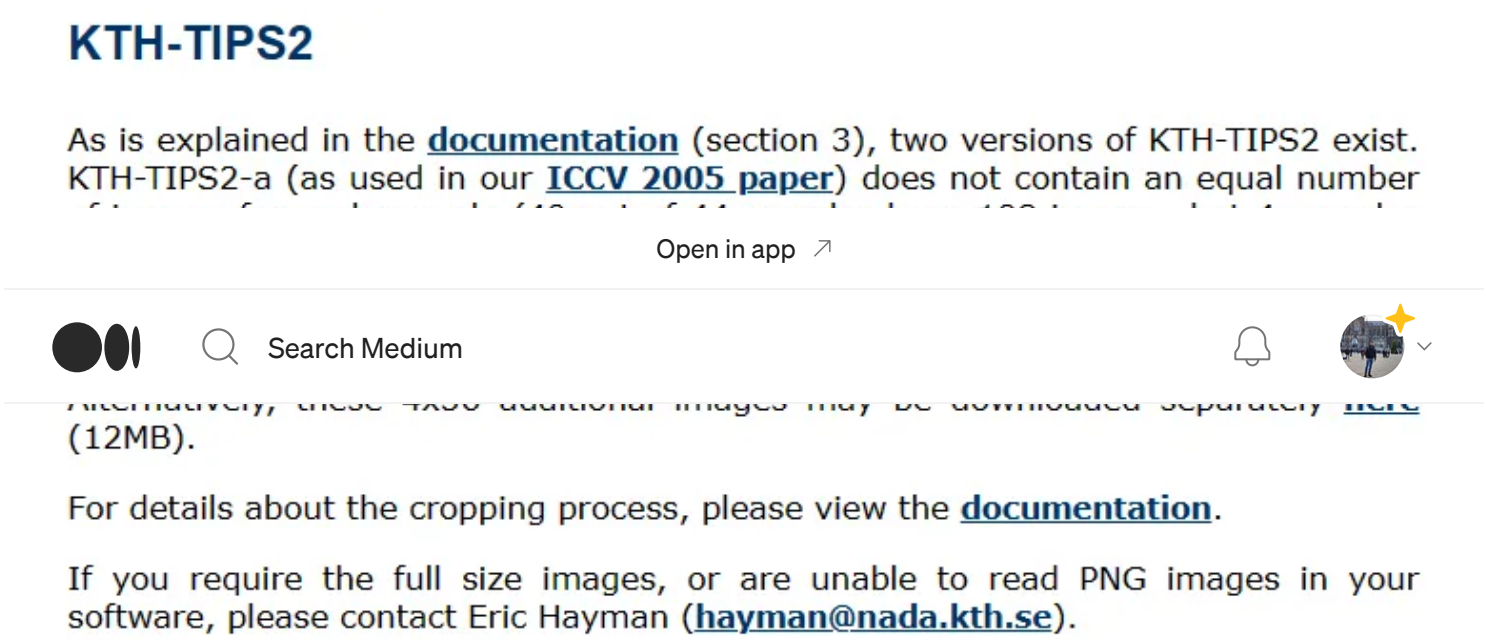
Here I attempt to classify texture images in KTH-TIPS2-b dataset in which the texture of all the images in the dataset will be enhanced using LBP (Local Binary Patterns). The histogram of those LBP features is going to be constructed, forming a single dimensional array that acts as the feature vector. Several machine learning models, i.e., SVM (Support Vector Machine), Logistic Regression and KNN (K-Nearest Neighbors) are then trained using those feature vectors.

Below is the arrangement of this article:

1. The KTH-TIPS2-b Dataset
2. LBP in a Nutshell
3. Data Loading and Visualization
4. Preprocessing and Creating Labels
5. Train/Test Split
6. Extracting LBP features
7. Generating Histograms
8. Model Training
9. Evaluation

The KTH-TIPS2-b Dataset

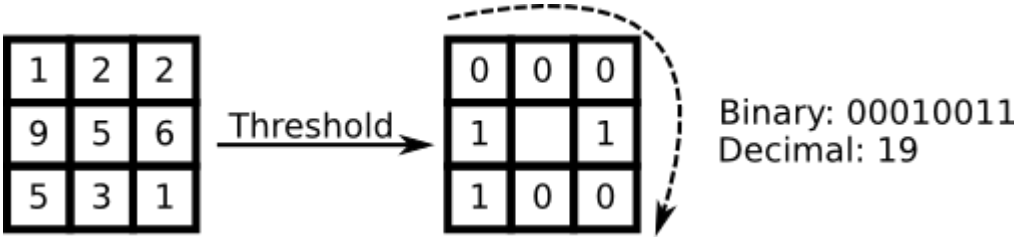
If you are interested to follow along with me, you need to download the dataset that I am using. You can do that by opening this webpage and click the KTH-TIPS2-b link [2]. This dataset consists of 11 texture classes in which each of those contains exactly 432 images. Most of those RGB images have the size of 200×200 pixels. — Yes, it is "most of" since some of those are slightly larger or smaller for some reasons. We will get into this later in the code.



What the official website of KTH-TIPS2-b dataset looks like [2].

LBP in a Nutshell

LBP was first invented by Ojala et al. back in 1996 [3], and since then LBP is considered as one of the best feature extraction techniques when it comes to enhancing textures. There are actually plenty of nuts and bolts inside of the process for extracting LBP features. However, the main idea is simply comparing a pixel with its surrounding neighbors and generating a new pixel value according to those neighbors.



Creating a binary sequence by thresholding neighbor pixels with its center pixel value [4].

In the illustration above, assume that we have a large image, and we will take only the patch of size 3×3. The center pixel of this patch is going to act as the threshold value such that any neighbor that has lower value than the center will be encoded as 0, otherwise 1. These binary values form a sequence which will be converted to decimal. This decimal number is going to be used to update the center pixel value. And that's it, we will do this process such that all pixels within the entire image acts as the center pixel — i.e., we will take 3×3 patch from top-left corner of the entire image striding all the way to the bottom-right. Below is what an LBP features of an image look like.

A picture of me standing on a bridge. Original (left), LBP features (right).

Nowadays there are several packages that implements LBP in it, such as Scikit Image, Mahotas and OpenCV [5]. In this project we will use the first one since that is the only one that I am familiar with.

Data Loading and Visualization

Alright, so I guess we are now ready to write the code. Let's start by importing all the required modules.

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt

from tqdm import tqdm

from skimage.feature import local_binary_pattern
from sklearn.model_selection import train_test_split

from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

As I have mentioned earlier, instead of implementing the LBP feature extraction from scratch, we are going to use the *local_binary_pattern()* function from Scikit-Image (*skimage*) module which will help us by much. Furthermore, the three machine learning models is also imported from Scikit-Learn (*sklearn*) since implementing them from scratch are not quite trivial either. One module that is not quite necessary — yet I take it anyway — is the *tqdm*, which is useful to display progress bar on a *for* loop.

After importing the required modules, I create a *load_images()* function which I think the name is self-explanatory. This function accepts a directory address where a bunch of images are stored. Inside of this function, every single file name in the directory is read by *os.listdir()* which afterwards the actual images are loaded using *cv2.imread()* one by one. Keep in mind that the image loaded using this function is read as BGR — thanks to the nature of the OpenCV module. This is the reason that I directly change the color channel order using *cv2.cvtColor()*.

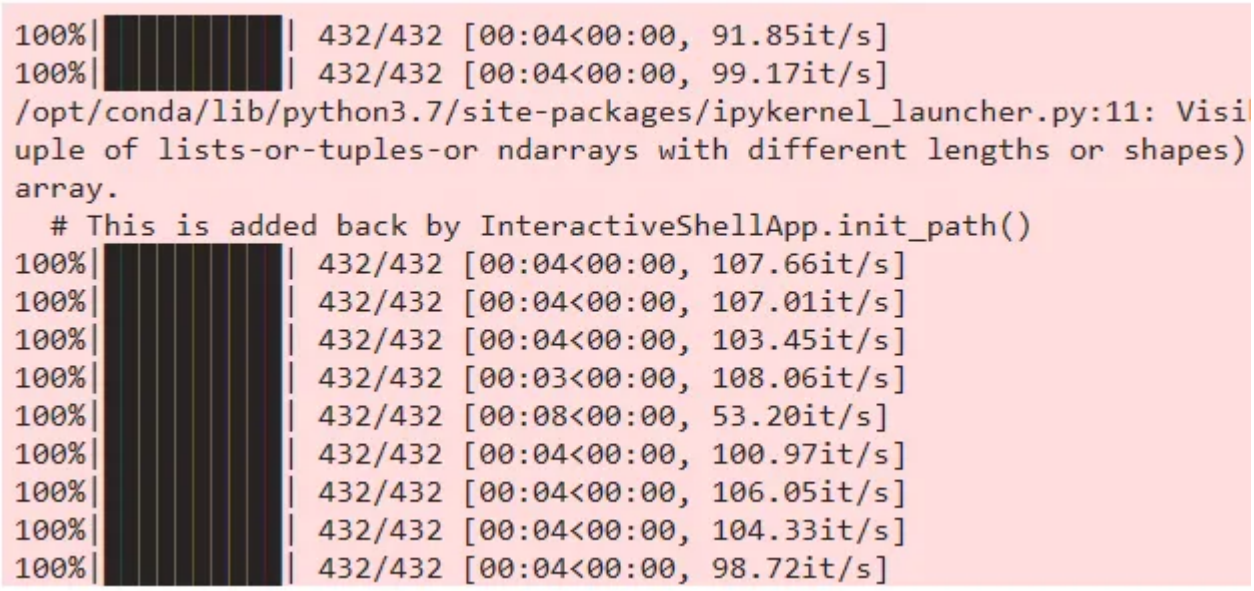
```
def load_images(path):
```

```
    images = []
    filenames = os.listdir(path)

    for filename in tqdm(filenames):
        image = cv2.imread(os.path.join(path, filename))
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        images.append(image)

    return np.array(images)
```

We then call this *load_images()* function 11 times where the loaded images are stored in 11 different arrays.

```
main_dir = '/kaggle/input/kthtips2b/KTH-TIPS2-b/'

classnames = ['aluminium_foil', 'brown_bread', 'corduroy',
              'cork', 'cotton', 'cracker', 'lettuce_leaf',
              'linen', 'white_bread', 'wood', 'wool']

class_0 = load_images(main_dir+classnames[0])
class_1 = load_images(main_dir+classnames[1])
class_2 = load_images(main_dir+classnames[2])
class_3 = load_images(main_dir+classnames[3])
class_4 = load_images(main_dir+classnames[4])
class_5 = load_images(main_dir+classnames[5])
class_6 = load_images(main_dir+classnames[6])
class_7 = load_images(main_dir+classnames[7])
class_8 = load_images(main_dir+classnames[8])
class_9 = load_images(main_dir+classnames[9])
class_10 = load_images(main_dir+classnames[10])
```

Once the code above is run, your notebook will probably show a warning that looks something like as follows. This is essentially because the images that we are loading does not have the exact same dimension. But don't worry about that since all the images are still loaded properly.



The warning that appears when the previous code is run.

Just to ensure, here is a proof that it is indeed the case. After running the code below, you will see that only the *class_0* and *class_10* which the image dimensions are truly uniform. It is worth noting that each element in the tuple of (432, 200, 200, 3) denotes the number of images, height, width, and the number of channels, respectively. In a Numpy array, if the next axis of the array has different size — i.e., in this case different image dimension— then the *shape* attribute does not return the number of elements in those non-uniform axes.

```
classes = [class_0, class_1, class_2, class_3,
           class_4, class_5, class_6, class_7,
           class_8, class_9, class_10]

for class_images in classes:
    print(class_images.shape)
```

```
(432, 200, 200, 3)
(432,)
(432,)
(432,)
(432,)
(432,)
(432,)
(432,)
(432,)
(432, 200, 200, 3)
```

class_0 and **class_10** are the two classes which the image sizes are uniform.

Now what are actually the exact size of those non uniform images? We can check that simply by running the following code. Here I attempt to print out all unique image dimensions in *class_6* (*lettuce_leaf*).

```
shapes = []
for image in class_6:
    if image.shape not in shapes:
        shapes.append(image.shape)

shapes
```

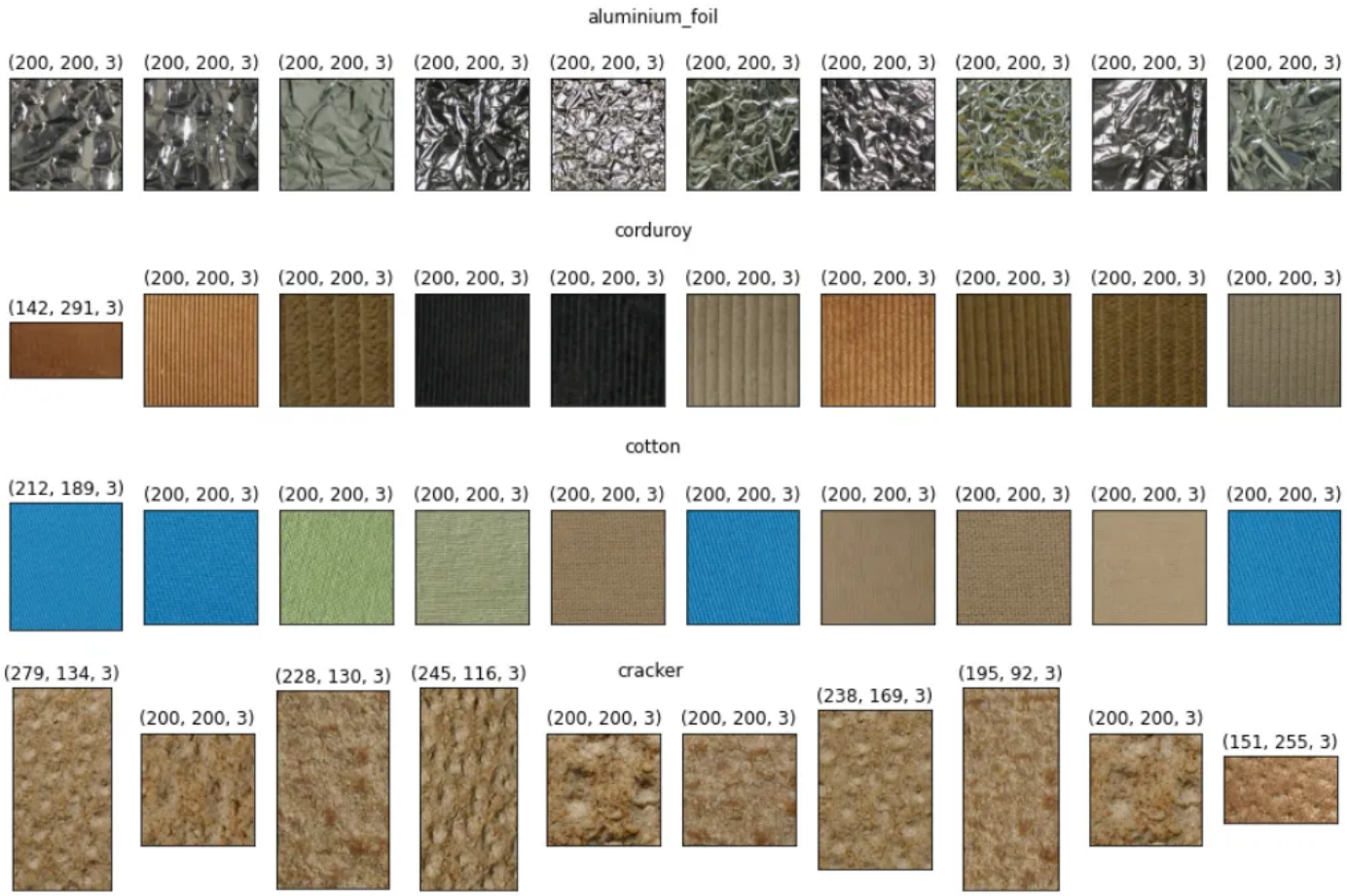
```
[(200, 200, 3),
 (211, 190, 3),
 (119, 207, 3),
 (243, 166, 3),
 (198, 201, 3),
 (220, 163, 3),
 (217, 166, 3),
 (207, 193, 3),
 (204, 196, 3)]
```

The lettuce leaf class contains images of different sizes.

Now let's create a new function that allows us to display several images in the KTH-TIPS2-b dataset. Here I decided to display several images from four different classes. I recommend you playing around with the below code to see what the texture images of

other classes look like. You can also change the value of *start_index* so that you can see more images in the class.

```
def show_raw_images(images, classname, start_index=0):
    fig, axes = plt.subplots(ncols=10, nrows=1, figsize=(16, 2.5))
    plt.suptitle(classname)

    index = start_index
    for i in range(10):
        axes[i].imshow(images[index])
        axes[i].set_title(images[index].shape)
        axes[i].get_xaxis().set_visible(False)
        axes[i].get_yaxis().set_visible(False)
        index += 1
    plt.show()

show_raw_images(class_0, classnames[0], start_index=20)
show_raw_images(class_2, classnames[2])
show_raw_images(class_4, classnames[4])
show_raw_images(class_5, classnames[5])
```



The first 10 images from class aluminium foil, corduroy, cotton, and cracker.

Preprocessing and Creating Labels

After all images have been successfully loaded, now that I want to preprocess those images. The preprocessing steps are simple, we will only convert them to grayscale and resize to a specific size. Grayscale conversion is needed in this case since essentially LBP feature extraction does not require color information. To the resizing, I decided to resize the images to 150×150. The two preprocessing steps are wrapped in the *preprocess_images()* function with the help of *cv2.cvtColor()* and *cv2.resize()*.

```
def preprocess_images(images):
    preprocessed_images = []
    for image in tqdm(images):
        image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
        image = cv2.resize(image, dsize=(150,150))
        preprocessed_images.append(image)
```

```
    return np.array(preprocessed_images)
```

After the above code block has been run, the next thing to do is to call the function one by one for each image array of different classes. The following code might look quite ugly as everything is literally hard-coded.

> *To be honest I am not sure how to do this in a better way since neither np.append() nor np.vstack() work in this case due to the non-uniform array shape which I explained earlier. But anyhow, I guess it's completely okay as long as it works as expected.*

```
class_0_preprocessed = preprocess_images(class_0)
class_1_preprocessed = preprocess_images(class_1)
class_2_preprocessed = preprocess_images(class_2)
class_3_preprocessed = preprocess_images(class_3)
class_4_preprocessed = preprocess_images(class_4)
class_5_preprocessed = preprocess_images(class_5)
class_6_preprocessed = preprocess_images(class_6)
class_7_preprocessed = preprocess_images(class_7)
class_8_preprocessed = preprocess_images(class_8)
class_9_preprocessed = preprocess_images(class_9)
class_10_preprocessed = preprocess_images(class_10)
```

As all images have been preprocessed, now that we can put them in a single array which I name *all_images*. This can simply be achieved using *np.vstack()*. The shape of *all_images* array shows that our resizing process that we did previously works properly as now our images have the dimension of 150×150 pixels. The value of 4752 also indicates that we have successfully put 432 images from 11 classes into a single array. Look at the subsequent code for the details.

```
all_images = np.vstack((class_0_preprocessed, class_1_preprocessed,
                        class_2_preprocessed, class_3_preprocessed,
                        class_4_preprocessed, class_5_preprocessed,
                        class_6_preprocessed, class_7_preprocessed,
                        class_8_preprocessed, class_9_preprocessed,
```

```
                            class_10_preprocessed))
all_images.shape
```

```
(4752, 150, 150)
```

The shape of **all_images** array.

Fortunate for us, the number of samples of all class are 432 which simplifies our work a little bit. The following code shows how I create the label of each sample. Our *labels* will be a long single-dimensional array with the length of 4752 — matches exactly with the length of *all_images*.

```
no_of_samples = 432
labels = np.array([0]*no_of_samples + [1]*no_of_samples + \
                  [2]*no_of_samples + [3]*no_of_samples + \
                  [4]*no_of_samples + [5]*no_of_samples + \
                  [6]*no_of_samples + [7]*no_of_samples + \
                  [8]*no_of_samples + [9]*no_of_samples + \
                  [10]*no_of_samples)
labels[:500]
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

The first 500 labels. The number of zeros — as well as the other labels — is 432.

Train/Test Split

Train/test split might be considered as a standard way to find out whether our model suffers from overfitting. I am going to do this process using the *train_test_split()* function which is taken from Sklearn. The parameter *test_size* is set to 0.3, which basically means that we are going to use 30% of our dataset for testing purpose.

```
X_train, X_test, y_train, y_test = train_test_split(all_images,
                                                    labels,
                                                    test_size=0.3)

print('X_train.shape\t', X_train.shape)
print('X_test.shape\t', X_test.shape)
print('y_train.shape\t', y_train.shape)
print('y_test.shape\t', y_test.shape)
```

```
X_train.shape    (3326, 150, 150)
X_test.shape     (1426, 150, 150)
y_train.shape    (3326,)
y_test.shape     (1426,)
```

The number of train and test data.

After splitting the data using the code above, we can see that the number of training samples is 3326, while the number of samples for testing purpose is 1426.

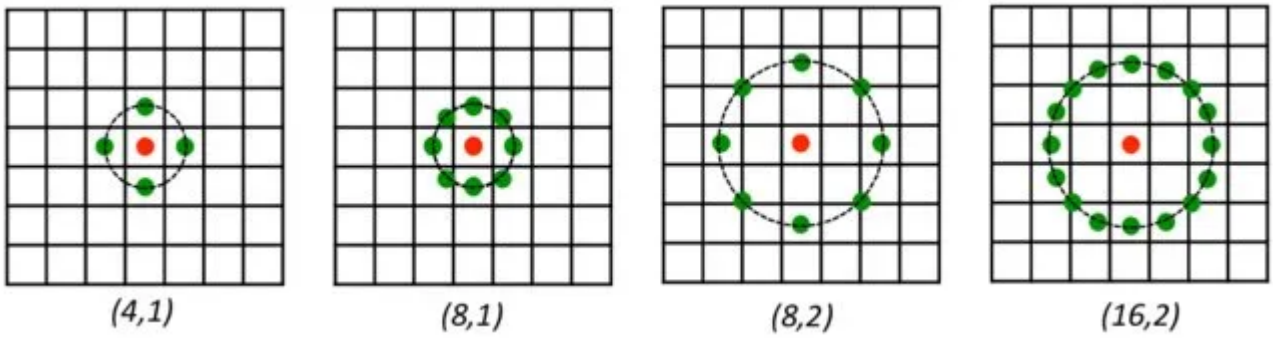
Extracting LBP Features

Alright, now let's move on to the main topic of this writing: LBP. Here is the function to extract LBP features.

```
def extract_lbp(images):
    lbps = []
    for image in tqdm(images):
        lbp = local_binary_pattern(image, P=8, R=1)
        lbps.append(lbp)

    return np.array(lbps)
```

What we are actually doing inside the *extract_lbp()* function is to iterate through an array of images in which every single of those will be processed using *local_binary_pattern()* function that we have already imported from Scikit-Image. If we take a look at the function, we can see that it takes three parameters: the image itself, P, and R.

The P parameter denotes the number of sampling points that will be thresholded by the center pixel, while the R parameter determines the sampling point radius. The figure below displays how different P and R value affects the sampling point locations.



What LBP with different P and R looks like [6].

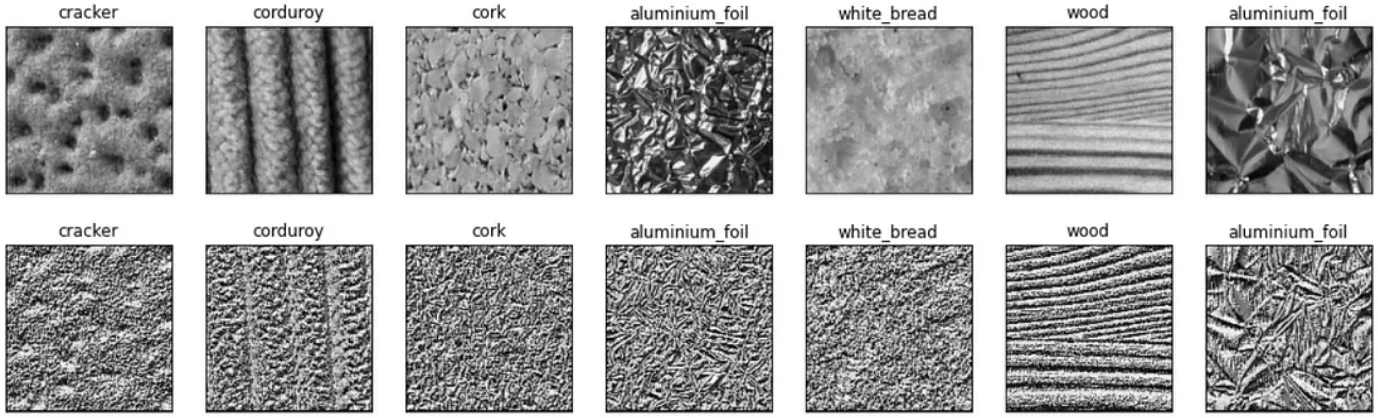
In our case, I decided to set the value of the two variables to 8 and 1, respectively, in which the illustration in the above figure is shown in the second image from the left.

Now let's get back to the code. — As the *extract_lbp()* function has been defined, we can now pass *X_train* and *X_test* to it. The resulting LBP features are then stored in *X_train_lbp* and *X_test_lbp*.

```
X_train_lbp = extract_lbp(X_train)
X_test_lbp = extract_lbp(X_test)
```

If you are wondering what the resulting images will be, you can run the following code and see the results.

```
def show_images_with_labels(images, labels, start_index=0):
    fig, axes = plt.subplots(ncols=7, nrows=1, figsize=(18, 2.5))

    index = start_index
    for i in range(7):
        axes[i].imshow(images[index], cmap='gray')
        axes[i].set_title(classnames[labels[index]])
        axes[i].get_xaxis().set_visible(False)
        axes[i].get_yaxis().set_visible(False)
        index += 1
    plt.show()

show_images_with_labels(X_train, y_train)
show_images_with_labels(X_train_lbp, y_train)
```

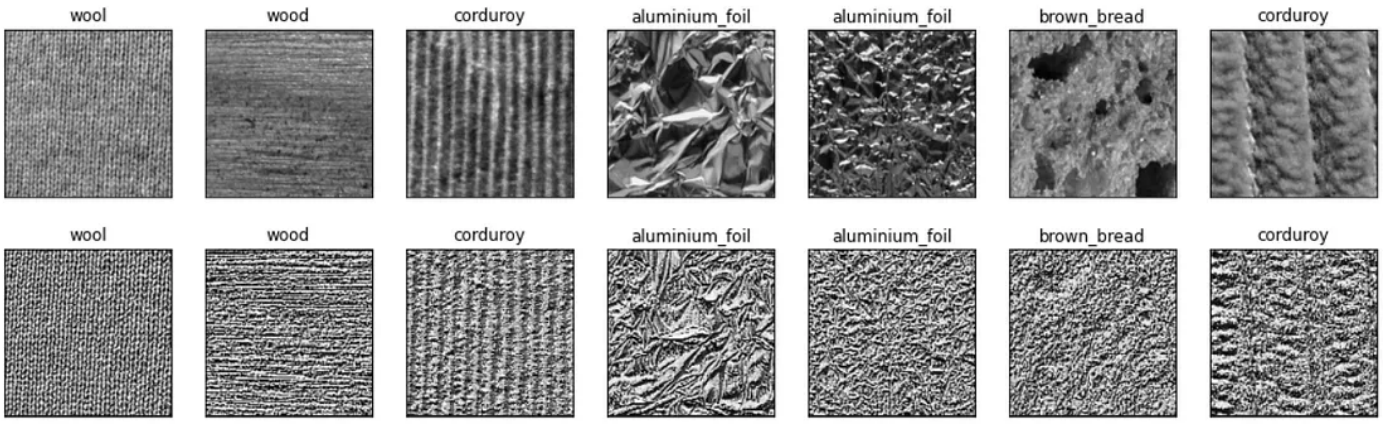


The original texture image (top) and their corresponding LBP features (bottom).

The function that I am showing here is quite a bit similar to *show_raw_images()* that we defined earlier. The point of the code is that we will visualize 7 images along with its corresponding LBP features. According to these outputs, we can see that the textural details of LBP features look clearer as compared to the original images.

We can also show some other images in the *X_train* simply by changing the value for *start_index* parameter.

```
show_images_with_labels(X_train, y_train, start_index=7)
show_images_with_labels(X_train_lbp, y_train, start_index=7)
```



Several other images from training set (top) and its LBP features (bottom).

Generating Histograms

Machine learning models basically can only be trained whenever the samples are in form of a single-dimensional array. Thus, the LBP features that we stored in *X_train_lbp* and *X_test_lbp* still need to be processed. There are actually several ways to do this, and I decided to convert the images into histogram of pixel intensity levels.

However, the histogram that I am going to create here is quite different since I also want it to be able to preserve spatial information a little bit. In order to do so, first I am going to divide the original image into several sub-images. The histogram of each sub-image is then extracted and concatenated together with the histograms that come from the same image. This histogram generation process is very similar to the illustration below.

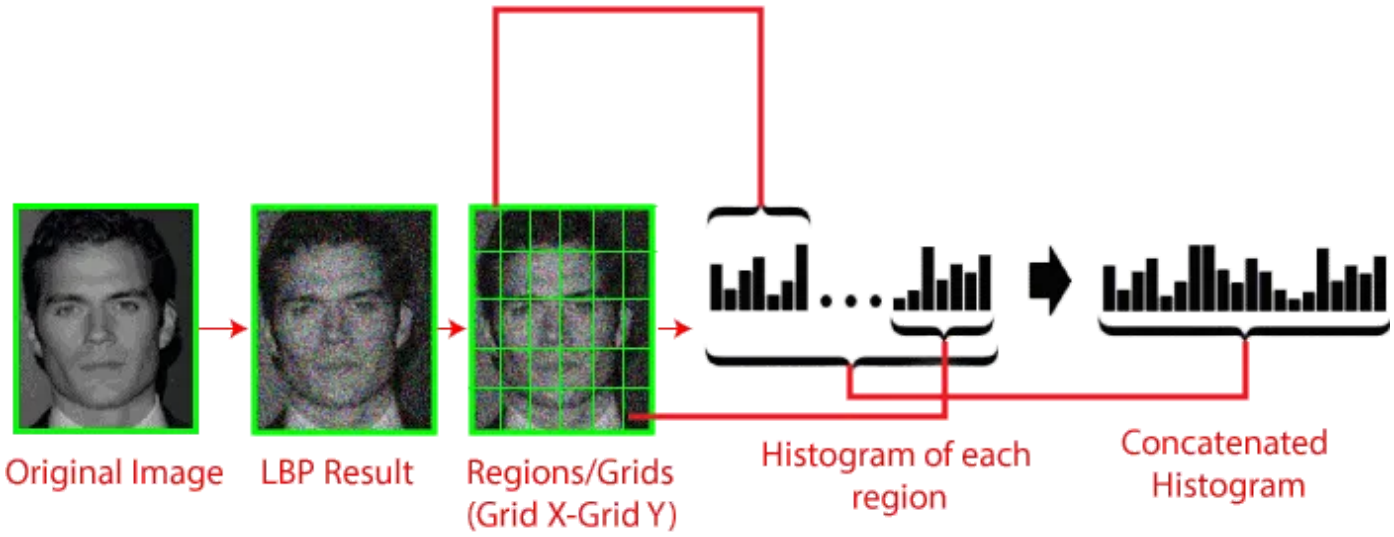


Illustration of how the histogram for an image is generated [7].

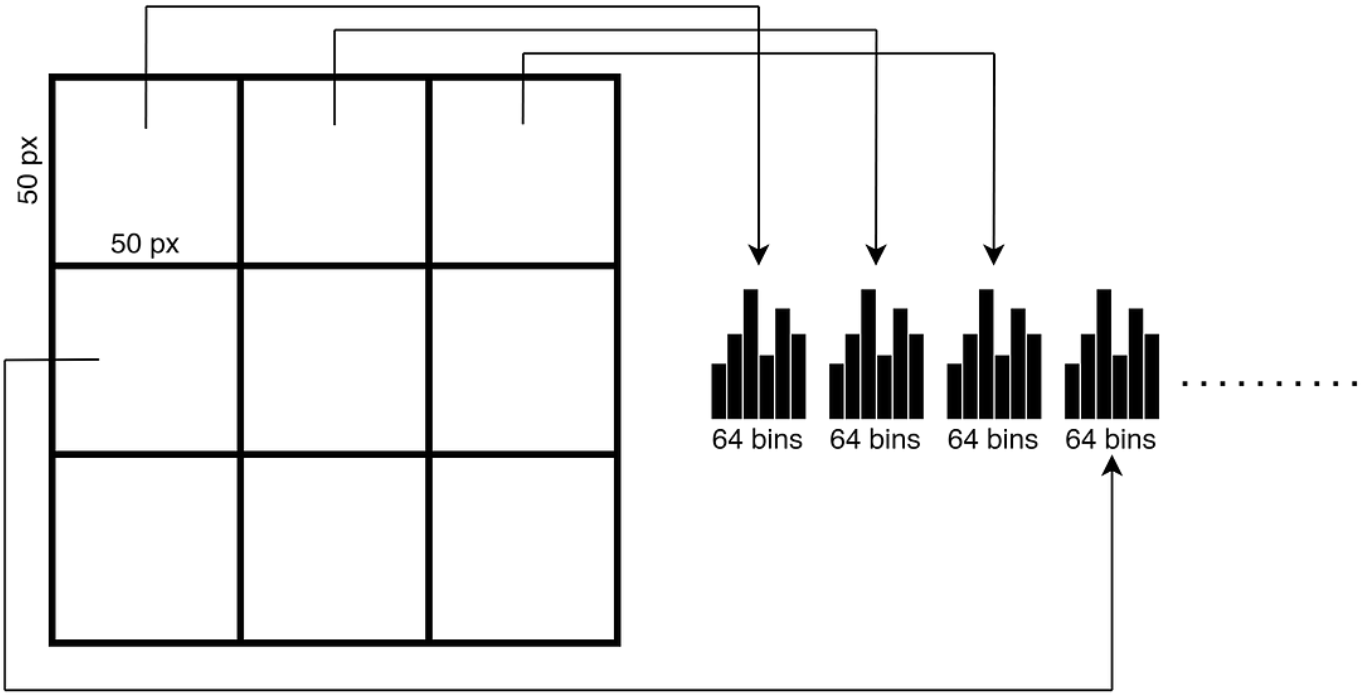
One question: but why do we do this? The reason is that if we don't divide an image this way, we are going to completely lose spatial information since histogram essentially works simply by counting the number of occurrences of each pixel intensity value. And this is done without taking into account the coordinate where a pixel is located.

My code implementation for this part is quite tricky for me. The *create_histogram()* function works by accepting three inputs: *images*, *sub_images_num*, and *bins_per_sub_images*. The first parameter is an array of images, the second one is the

number of sub-images that we are going to obtain for each row and column, while the last one is the number of bins of the sub-image histogram.

```
def create_histograms(images, sub_images_num, bins_per_sub_images):
    all_histograms = []
    for image in tqdm(images):
        grid = np.arange(0, image.shape[1]+1, image.shape[1]//sub_images_num)

        sub_image_histograms = []

        for i in range(1, len(grid)):
            for j in range(1, len(grid)):
                sub_image = image[grid[i-1]:grid[i], grid[j-1]:grid[j]]

                sub_image_histogram = np.histogram(sub_image, bins=bins_per_sub_i
                sub_image_histograms.append(sub_image_histogram)

        histogram = np.array(sub_image_histograms).flatten()
        all_histograms.append(histogram)

    return np.array(all_histograms)
```

In this case, I decided to set the *sub_images_num* to 3 and the *bins_per_sub_images* to 64. This means that our LBP feature, which has the dimension of 150×150, will be divided into 3 rows and 3 columns. Hence causing an image to contain 9 sub-images with the size of 50×50 pixels each.



How the histograms of the 9 sub-images are concatenated.

We can also know that the final histogram length is going to be 576 which is obtained by multiplying 64 by 9. Later on, the long 1-dimensional array will act as the representation of a single image as well as the feature vector to be employed to train several machine learning models.

```
X_train_hist = create_histograms(X_train_lbp, sub_images_num=3, bins_per_sub_image
X_test_hist = create_histograms(X_test_lbp, sub_images_num=3, bins_per_sub_images=

print('X_train_hist\t', X_train_hist.shape)
print('X_test_hist\t', X_test_hist.shape)
```

```
100%|██████████| 3326/3326 [00:05<00:00, 599.01it/s]
100%|██████████| 1426/1426 [00:02<00:00, 599.21it/s]
X_train_hist    (3326, 576)
X_test_hist     (1426, 576)
```

The final histogram length of each image is 576.

Model Training

As I have mentioned earlier, there are three machine learning models to be trained in this project, namely SVM, Logistic Regression, and KNN. I set the parameters of SVM and Logistic Regression to its default while the K value for KNN is set to 1.

```
model_svm = SVC()
model_svm.fit(X_train_hist, y_train)

print('SVM train acc\t:', model_svm.score(X_train_hist, y_train))
print('SVM test acc\t:', model_svm.score(X_test_hist, y_test))
```

```
SVM train acc    : 0.8406494287432351
SVM test acc     : 0.8274894810659187
```

Accuracy score obtained by SVM.

```
model_logreg = LogisticRegression()
model_logreg.fit(X_train_hist, y_train)

print('Logreg train acc\t:', model_logreg.score(X_train_hist, y_train))
print('Logreg test acc\t\t:', model_logreg.score(X_test_hist, y_test))
```

```
Logreg train acc        : 0.8767288033674083
Logreg test acc         : 0.8513323983169705
/opt/conda/lib/python3.7/site-packages/sklearn/linear_
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale
    https://scikit-learn.org/stable/modules/preprocess
Please also refer to the documentation for alternative
    https://scikit-learn.org/stable/modules/linear_mod
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG,
```

Accuracy score obtained by Logistic Regression.

```
model_knn = KNeighborsClassifier(n_neighbors=1)
model_knn.fit(X_train_hist, y_train)

print('KNN train acc\t:', model_knn.score(X_train_hist, y_train))
print('KNN test acc\t:', model_knn.score(X_test_hist, y_test))
```

```
KNN train acc    : 1.0
KNN test acc     : 0.9242636746143057
```

Accuracy score obtained by KNN with K=1.

Based on the experiment results above, we can see that KNN is the model that performs best in this case, in which it obtains the accuracy score of 92.4% on testing data. Due to this reason, I am going to bring the KNN model to the evaluation stage to see some prediction results.

> *It might be worth noting that if you try to re-run this code again, you will probably obtain different results thanks to the nature of random splitting made by the train_test_split() function as well as the random weight initialization in the ML models (especially the SVM and Logistic Regression).*

Evaluation

In Scikit-Learn, we can simply predict new data by calling the *predict()* method and passing that new data as the argument. Below is the code for that.

```
predictions = model_knn.predict(X_test_hist)
```

Since I want to focus on displaying the misclassified images hence, I create a new function named *find_misclassifications()* to do so. The function is quite simple, it works by iterating through all actual labels and the predicted class. A sample is said to be misclassified whenever the label and the predicted class is different. If this happens,

then the function will automatically store the indices where the misclassification occurs.

```
def find_misclassifications(labels, preds):
    indices = []
    for i, (label, pred) in enumerate(zip(preds, labels)):
        if pred != label:
            indices.append(i)

    return np.array(indices)

misclassifications = find_misclassifications(y_test, predictions)
misclassifications
```

```
array([   3,   34,   40,   51,   58,   66,   67,  109,  115,  119,  121,
        133,  139,  142,  144,  146,  158,  162,  216,  262,  268,  275,
        277,  296,  302,  308,  314,  319,  329,  350,  362,  376,  377,
        415,  418,  423,  457,  461,  483,  491,  521,  553,  563,  564,
        585,  614,  618,  623,  635,  659,  683,  688,  703,  734,  749,
        761,  764,  765,  779,  804,  815,  830,  831,  835,  844,  855,
        863,  873,  901,  902,  906,  926,  951,  963,  970,  989, 1008,
       1014, 1056, 1058, 1080, 1099, 1105, 1129, 1137, 1138, 1143, 1144,
       1157, 1169, 1181, 1227, 1236, 1251, 1267, 1275, 1286, 1300, 1302,
       1304, 1307, 1317, 1318, 1324, 1328, 1329, 1348, 1395])
```

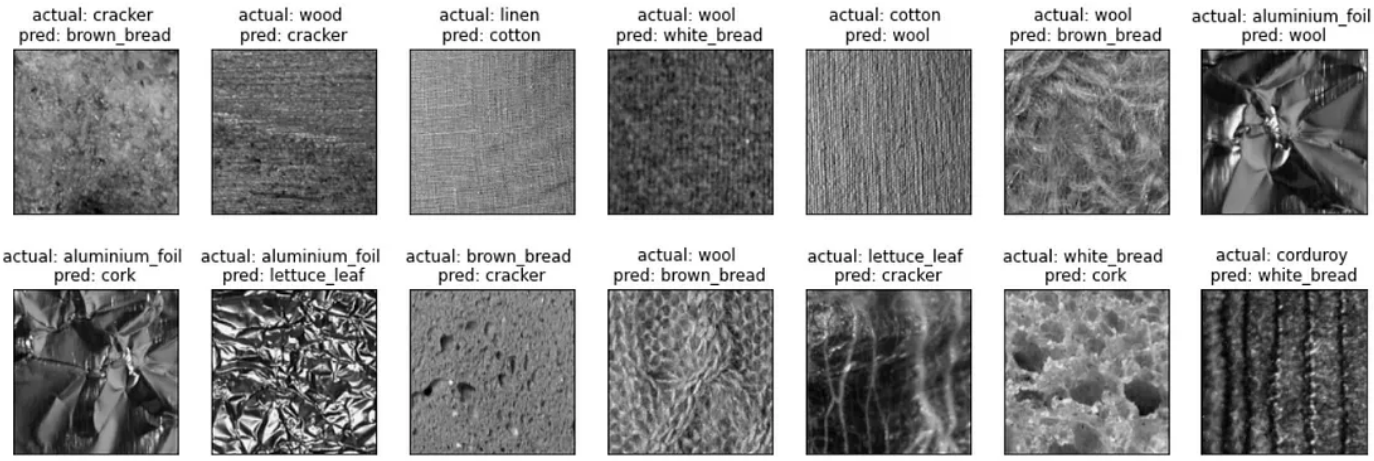
The indices of the misclassified samples on testing data.

And here we go, the output of the code above is basically an array which stores the indices of all misclassified images. There are 108 misclassifications in total out of 1426 samples in testing set. This number exactly matches with the accuracy score that we get from the *score()* method. We can simply calculate it by dividing 1318 (numer of correct predictions) by 1426 and obtain the percentage of 92.4 as well.

We can now show those misclassifications by using the *show_misclassifications()* function.

```
def show_misclassifications(images, misclassified, labels, preds, start_index=0):
    fig, axes = plt.subplots(ncols=7, nrows=2, figsize=(18, 6))
```

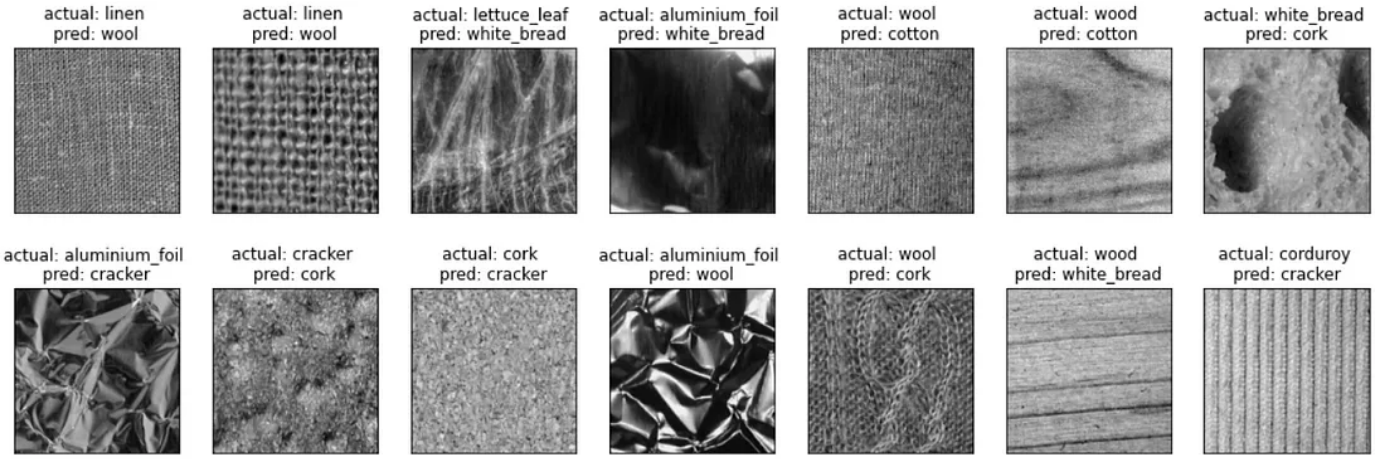
```
        index = start_index
        for i in range(2):
            for j in range(7):
                axes[i,j].imshow(images[misclassified[index]], cmap='gray')
                axes[i,j].set_title(f'actual: {classnames[labels[misclassified[index]]
                                    f'pred: {classnames[preds[misclassified[index]]]}
                axes[i,j].get_xaxis().set_visible(False)
                axes[i,j].get_yaxis().set_visible(False)
                if index == (index+14):
                    break
                index += 1
        plt.show()

show_misclassifications(X_test, misclassifications, y_test, predictions, start_in
```



The first 14 misclassified images in the testing set.

Again, if you want to see more images, you can always set the *start_index* to other numbers.

```
show_misclassifications(X_test, misclassifications,
                        y_test, predictions, start_index=14)
```



Several other misclassifications made by KNN.

Final Words

And that's it! We have successfully proven that the combination of traditional feature extraction method and a machine learning model — especially the KNN is able to achieve pretty good accuracy score — well, at least in this texture classification task. As the future work, I do recommend you play around with the parameters, such as the number of histogram bins, the number of sub-images, and the other ML hyperparameters. Furthermore, you may also try to use other texture features like HOG (Histogram of Oriented Gradients) or GLCM (Gray Level Co-occurrence Matrix) to find out whether they can produce even better classification accuracy score.

I hope you find this article helpful, thanks for reading!

Note: the notebook used in this article can be downloaded from this link.

References

[1] Matt Przybyla. *When to Avoid Deep Learning*. Towards Data Science. https://towardsdatascience.com/when-to-avoid-deep-learning-a7cfe3635022 [Accessed January 8, 2023].

[2] Eric Hayman and Barbara Caputo. *The KTH-TIPS and KTH-TIPS2 Image Databases*. https://www.csc.kth.se/cvap/databases/kth-tips/download.html [Accessed January 8, 2023].

[3] Timo Ojala et al., *A comparative study of texture measures with classification based on featured distributions*. Pattern Recognition. https://www.sciencedirect.com/science/article/abs/pii/0031320395000674 [Accessed January 8, 2023].

[4] *Local binary patterns*. Julia Images. http://juliaimages.org/ImageFeatures.jl/stable/tutorials/lbp/ [Accessed January 8, 2023].

[5] Adrian Rosebrock. *Local Binary Patterns with Python & OpenCV*. Pyimagesearch. https://pyimagesearch.com/2015/12/07/local-binary-patterns-with-python-opencv/ [Accessed January 8, 2023].

[6] Yanal Wazaefi. *Automatic diagnosis of melanoma from dermoscopic images of melanocytic tumors: Analytical and comparative approaches*. Research Gate. https://www.researchgate.net/publication/295647033_Automatic_diagnosis_of_melanoma_from_dermoscopic_images_of_melanocytic_tumors_Analytical_and_comparative_approaches/figures?lo=1 [Accessed January 9, 2023].

[7] *Face recognition and Face detection using the OpenCV*. Java T Point. https://www.javatpoint.com/face-recognition-and-face-detection-using-opencv [Accessed January 9, 2023].

Image Processing | Image Classification | Machine Learning | Feature Engineering | Artificial Intelligence

Follow

Written by Muhammad Ardi

109 Followers · Writer for Becoming Human: Artificial Intelligence Magazine

A machine learning, deep learning, computer vision, and NLP enthusiast. Doctoral student of Computer Science, Universitas Gadjah Mada, Indonesia.

Recommended from Medium

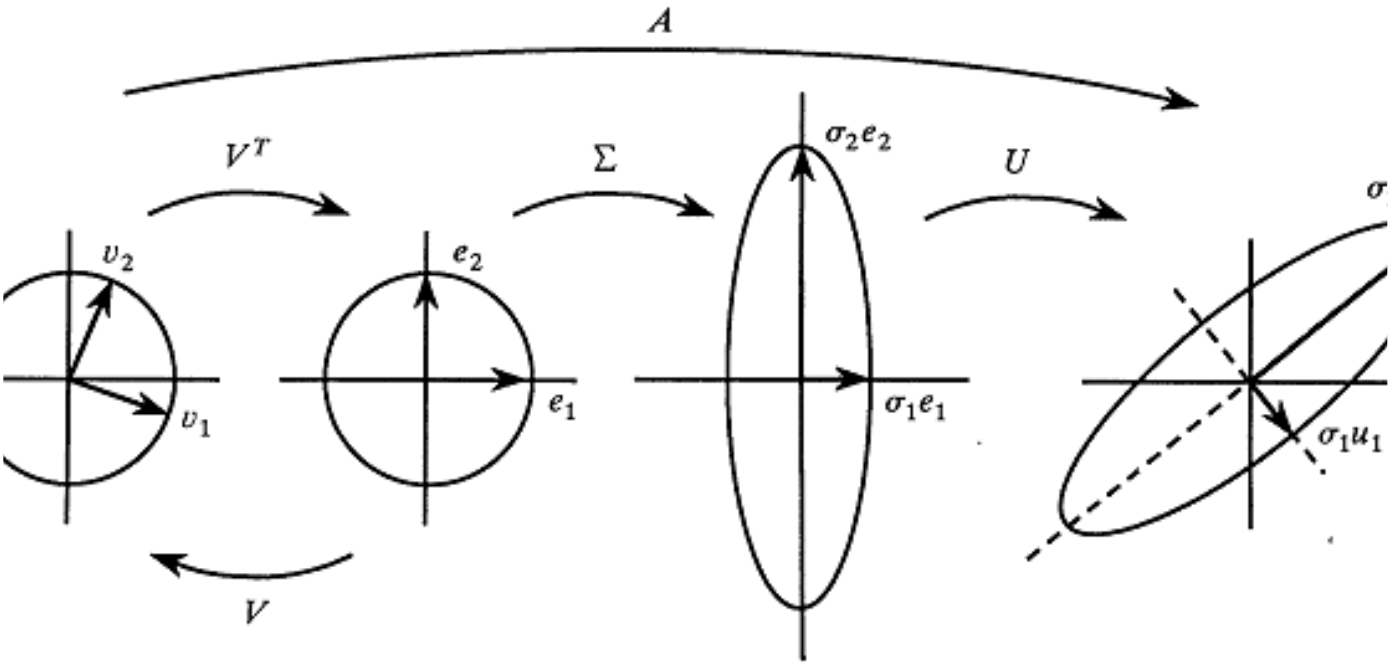
hengtao tantai

Use weighted loss function to solve imbalanced data classification problems

Imbalanced datasets are a common problem in classification tasks, where number of instances in one class is significantly smaller than...

8 min read · Feb 27

9



R. Gupta in Geek Culture

Singular Value Decomposition: Calculation using EigenValues and EigenVectors in Python

Well, I have gone through lots of articles explaining what PCA is and how to find the principal component for the feature matrix. Nearly...

6 min read · Dec 29, 2022

15

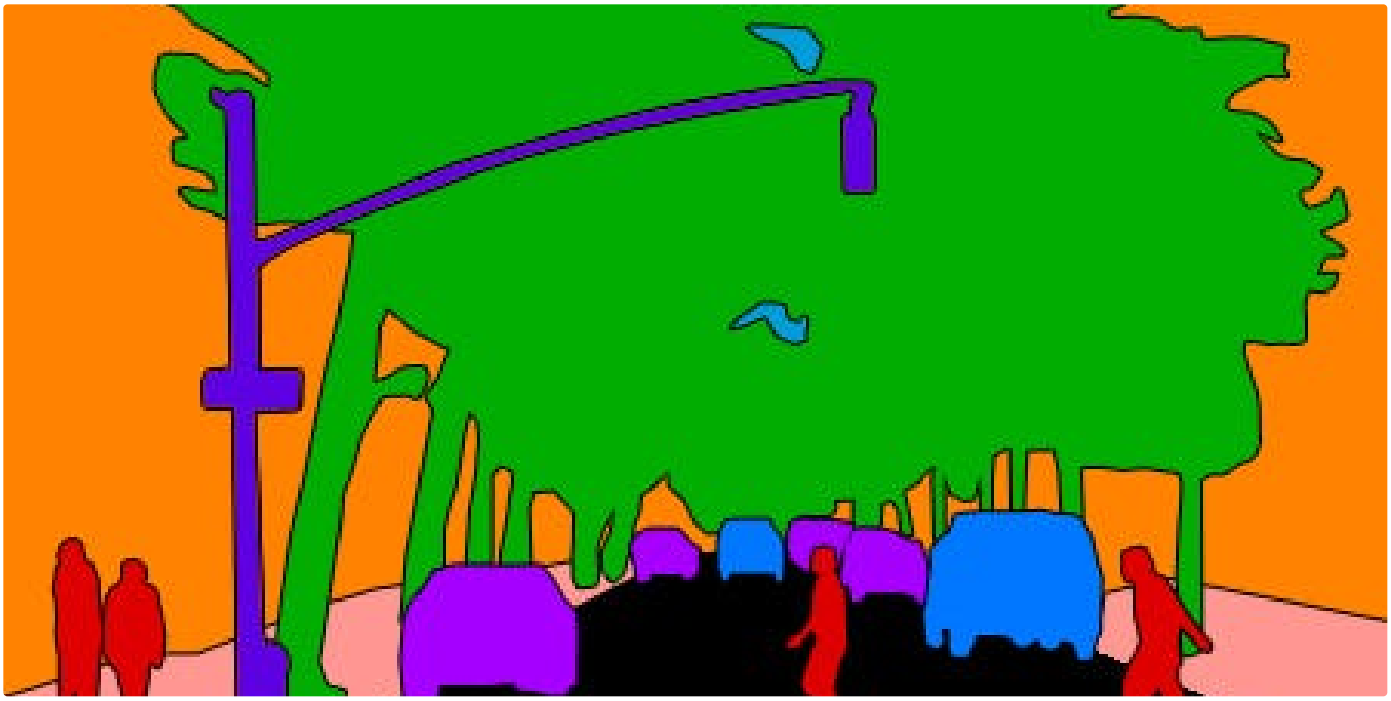
Lists

What is ChatGPT?

9 stories · 78 saves

Staff Picks

339 stories · 94 saves



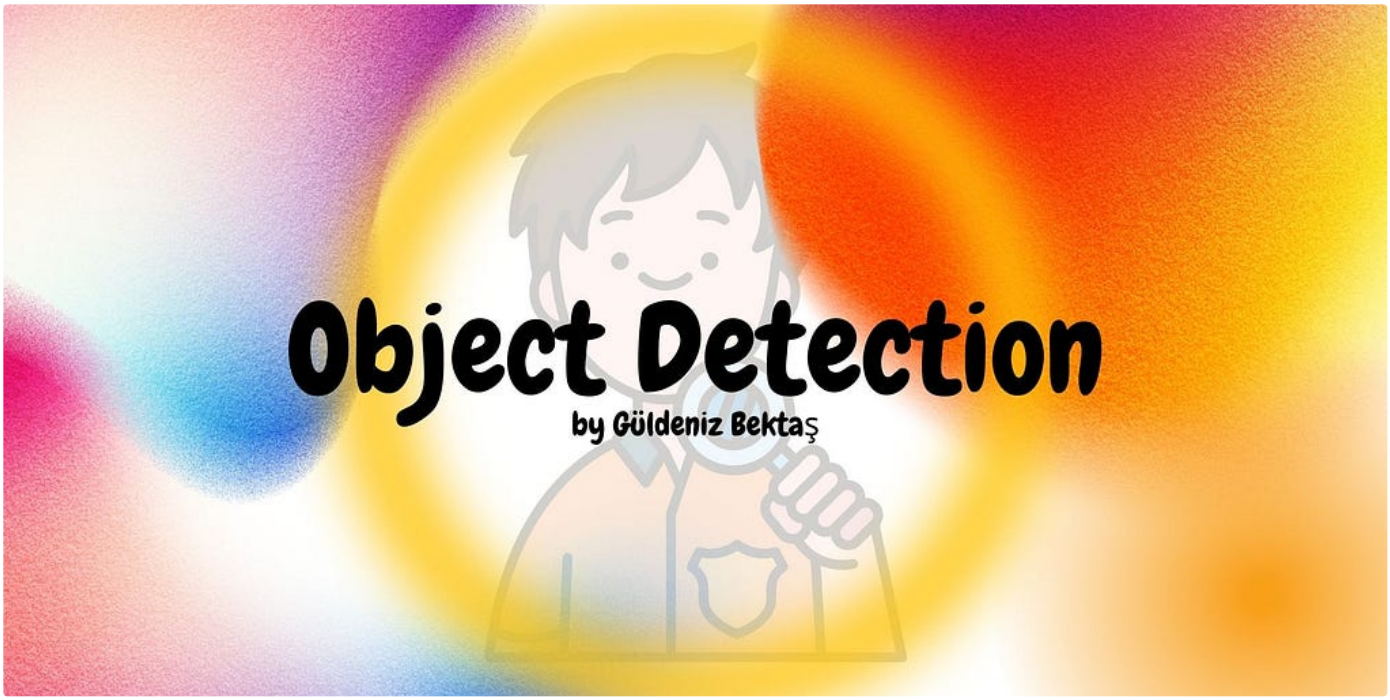
Satya

Real Time Semantic Segmentation with Deep learning

Introduction:

5 min read · Dec 12, 2022

45



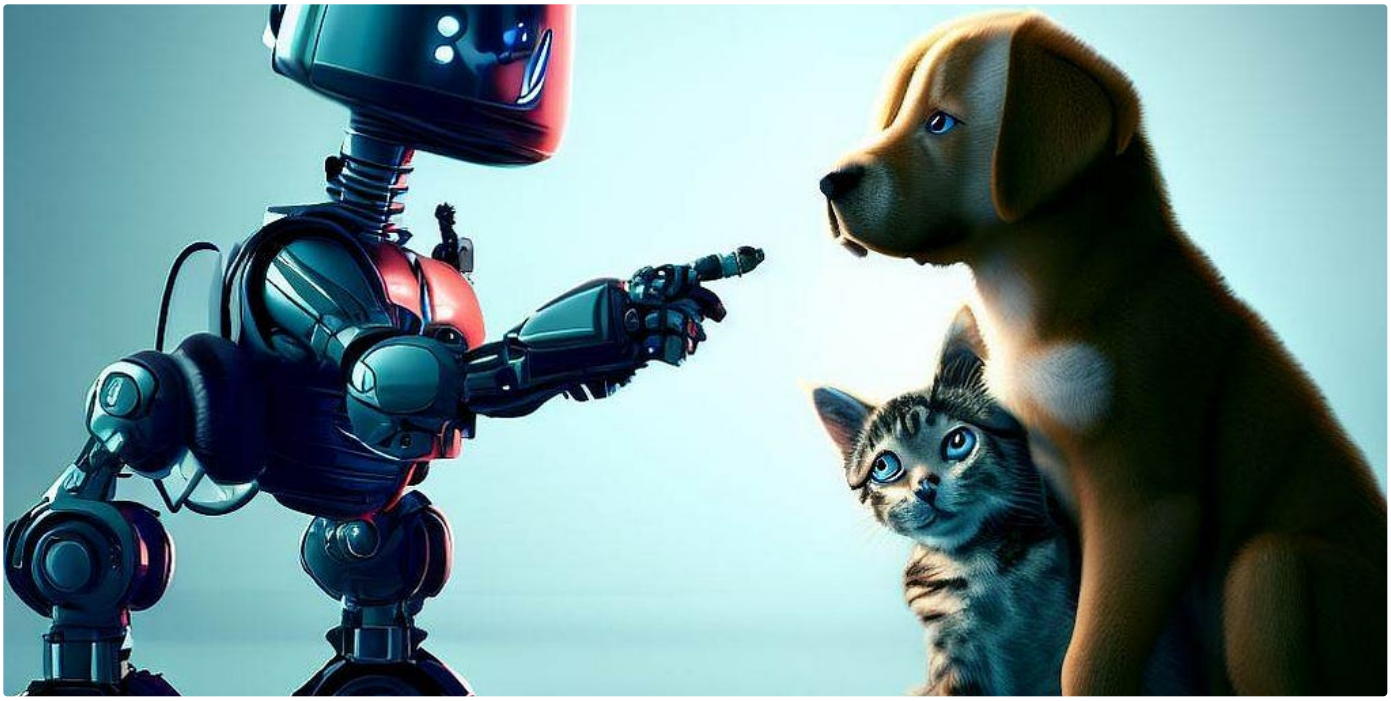
Güldeniz Bektaş

Object Detection 101

Everything you need to know to start object detection projects.

11 min read · Jan 11

101

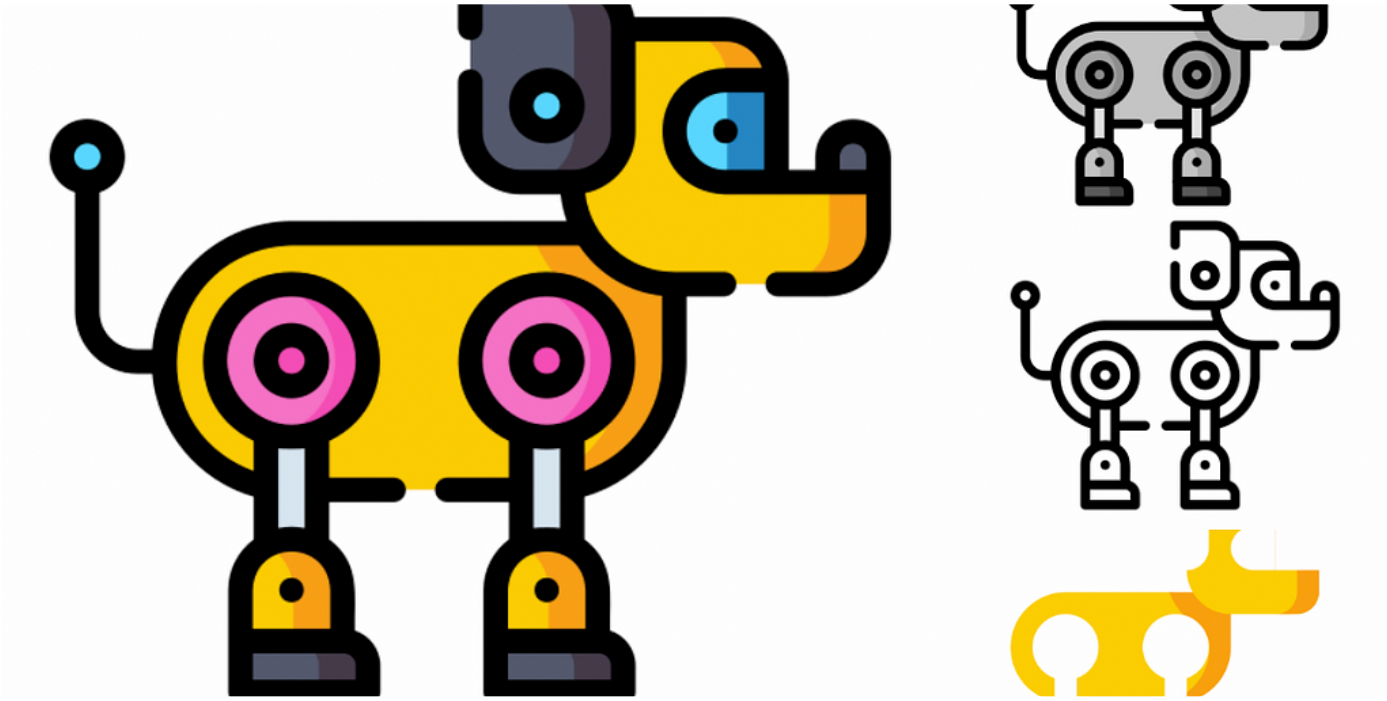


TheAIPlusBlog

Image Classification

Classification of Dogs and Cats using TensorFlow CNN. This Project contains a Simple CNN implementation, Data Augmenting and Reducing...

10 min read · 2 days ago



Conor O'Sullivan in Towards Data Science

Feature Engineering with Image Data

Cropping, grayscale, RGB channels, intensity thresholds, edge detection and colour filters

· 11 min read · Dec 5, 2022

83