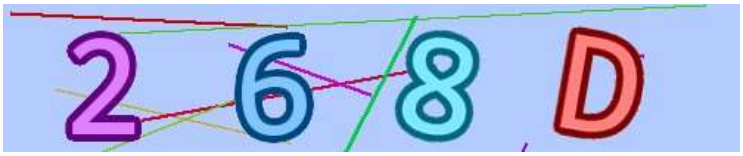
# KILLcaptcha

# CS771A: Introduction to Machine Learning
# Summer 2022-23

Karamjeet Singh (200488)

## 1.1 Image Processing:



The provided code snippet represents a pre-processing function for images. The function takes an image as input and performs several operations to modify and extract the last character of HexaCAPTCHA.

```
top_left = (0, 0) top_right = (image.shape[0]-1, 0) bottom_left =
(0, image.shape[1]-1) bottom_right = (image.shape[0]-1,
image.shape[1]-1)
```

Initially, the function identifies the four corners of the image using the image.shape attribute and assigns them to respective variables: top_left, top_right, bottom_left, and bottom_right.

```
color_frequencies = {} for corner in [top_left, top_right, bottom_left,
bottom_right]:
    color = image[corner] color_tup = tuple(color) if
    color_tup not in color_frequencies:
    color_frequencies[color_tup] = 0
    color_frequencies[color_tup] += 1
```

Next, the function creates a dictionary called color_frequencies to keep track of the frequency of each color present in the four corners of the image. It iterates over the corners and retrieves the

color value at each corner. The color is converted to a tuple, and if it is not present in the color_frequencies dictionary, a new key-value pair is added with the color tuple as the key and the frequency initialized to 0. The frequency of the color is then incremented by 1. This process calculates the frequency of colors present at the four corners of the image.

Preprint. Under review.

---

```
max_color = max(color_frequencies, key=color_frequencies.get)
R = max_color[0]
G = max_color[1]
B = max_color[2]
```

---

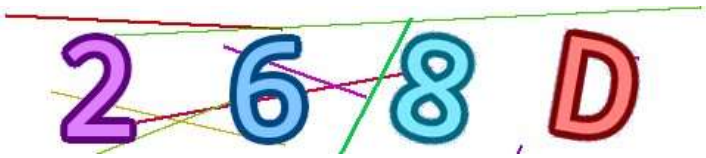
Subsequently, the most frequent color is determined by finding the key with the maximum value in the color_frequencies dictionary. The red, green, and blue (RGB) values of the most frequent color are extracted from the key tuple.

---

```
h,w,c = image.shape for i in
range(h):
    for j in range(w):
        if R == image[i,j,0] and G == image[i,j,1] and B == image[i,j,2]:
          image[i,j,0] = 255 image[i,j,1] = 255
          image[i,j,2] = 255
```

---

The function then proceeds to iterate over each pixel of the image using nested loops. It checks if the RGB values of the current pixel match the most frequent color. If a match is found, the RGB values of that pixel are set to 255, effectively converting it to white.




---

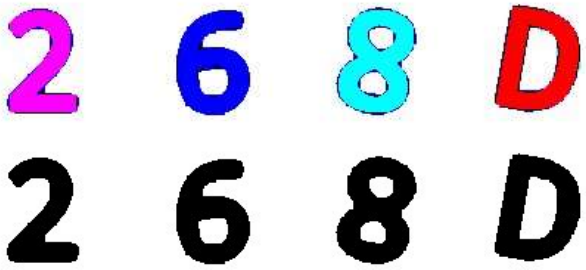
```
def remove_lines(img):
   _, binary = cv2.threshold(img, 210, 255, cv2.THRESH_BINARY) kernels =
   np.ones((5,5), np.uint8) dilated = cv2.dilate(binary, kernels, iterations=1)
   return dilated

h,w,c = img2.shape for i in
    range(h): for j in range(w):
                if 255 != img2[i,j,0]: img2[i,j,0] = 0
                    img2[i,j,1] = 0 img2[i,j,2] = 0
                if 255 != img2[i,j,1]: img2[i,j,0] = 0
                    img2[i,j,1] = 0 img2[i,j,2] = 0
```

```
if 255 != img2[i,j,2]: img2[i,j,0] = 0
    img2[i,j,1] = 0 img2[i,j,2] = 0
```

---

Following the remove_lines step, the function iterates over each pixel of the modified image again. It checks if any of the RGB values are not equal to 255 (indicating a non-white pixel). If any of the RGB values are not 255, they are set to 0, resulting in black pixels.



---

```
return img2[:,350:450,:]
```

---

Finally, the function selects a specific region of interest within the modified image by slicing it using the syntax [:, 350:450, :]. This slice extracts all rows of the image and selects columns ranging from index 350 to 450 (excluding the endpoint). The resulting image region is returned as the output of the function.

In summary, the provided code snippet applies various modifications to an input image, including converting the background color to white, removing lines, converting non-white pixels to black, and extracting the last character in the image.

NOTE - Only last character will decide whether the hexaCaptcha is even or odd. So we will use only that part to train the model

## 1.2 Model Training & Extraction:

---

```
# Pre-Processing and storing last character images:
num_train = 2000 image_train = [] for i in range(num_train): image =
cv2.imread("/Users/himeshagrawal/Downloads/assn2/train/%d.png" % i)
image_train.append(image)
```

---

The code loops through a range of 2000 and uses cv2.imread to read each image from the specified directory (*"/Users/himeshagrawal/Downloads/assn2/train/"*) using the corresponding index ($i$) in the filename. The images are then appended to the image_train list.

---

```
# Reading and storing training labels: file =
open("/Users/himeshagrawal/Downloads/assn2/train/labels.txt", "r") label_train =
file.read().splitlines() file.close()
label_train = np.array(label_train)
```

---

The code opens the "labels.txt" file located in the specified directory and reads its contents. The read labels are split into lines using splitlines(). The resulting label_train list is then converted into a numpy array.

---

```
# Converting the image\_train list into a numpy array:
image_train = np.array(image_train)
```

---

The image_train list, which contains the loaded training images, is converted into a numpy array.

---

```
# Preprocessing and saving processed images:
p4 = np.zeros((2000,100,100,3)) for i in
range(2000):
    p4[i] = pre_processing(image_train[i])
                        cv2.imwrite(f'/Users/himeshagrawal/Downloads/assn2/output/final_{i}.png', p4[i])
```

---

An empty numpy array p4 with dimensions (2000,100,100,3) is created to store the preprocessed images. The code then loops through the range of 2000 and preprocesses each image_train[i] using a pre_processing function (not shown in the provided code snippet). The preprocessed image is stored in p4[i]. Finally, cv2.imwrite is used to save the processed image as "final_i.png" in the specified output directory ("/Users/himeshagrawal/Downloads/assn2/output/").

This code snippet is similar to the one provided earlier, with a minor difference in the data directory path. It loads images from a directory, converts them to grayscale, resizes them, and flattens them. Then, it splits the dataset into training and testing sets using the train_test_split function from the sklearn.model_selection module. Here's a breakdown of the code:

---

```
# Importing the required libraries: import os
import numpy as np from PIL import Image
from sklearn.model_selection import train_test_split from sklearn.svm
import SVC
from sklearn.metrics import classification_report from
sklearn.metrics import accuracy_score from sklearn.neighbors
import KNeighborsClassifier

# Setting up the data directory and image size:
data_dir = '/Users/himeshagrawal/Downloads/assn2/seg_image' image_size = (100, 100)

# Loading and preprocessing the images: images =
[] for i in range(2000):
    image_path = os.path.join(data_dir, str(i) + ".png") image =
    Image.open(image_path).convert("L") image =
    np.array(image.resize(image_size)).flatten() images.append(image)
```

---

This code segment loops through a range of 2000, assuming that there are 2000 images in the specified directory. It constructs the image path using os.path.join and opens the image using PIL's Image.open. The image is then converted to grayscale using the convert("L") method. After that, the image is resized to the specified image_size using the resize method and converted into a 1D

array (flattened) using numpy's flatten method. The flattened image is then appended to the images list.

---

```
# Splitting the dataset into training and testing sets:
X_train, X_test, y_train, y_test = train_test_split(images, label_train, test_size=0.2, random_state=42)
```

---

The train_test_split function is used to split the dataset into training and testing sets. The images variable contains the preprocessed images, and label_train is assumed to be defined elsewhere (not shown in the provided code snippet). The test_size parameter specifies the proportion of the dataset to be used for testing (in this case, 20%). The random_state parameter ensures reproducibility by setting the random seed to 42.

After running this code, you will have X_train, X_test, y_train, and y_test variables containing the training and testing data, ready to be used for further processing and model training.

Here's a breakdown of the modified code:

---

```
# Importing the required libraries:
from sklearn.linear_model import LogisticRegression from
sklearn.model_selection import GridSearchCV from
sklearn.metrics import classification_report

# Initializing the logistic regression classifier:
lr = LogisticRegression()

# Defining the parameter grid for fine-tuning: param_grid = {
    'C': [0.1, 1, 10],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga']
}
```

---

In this example, the parameter grid includes different values for 'C' (inverse of regularization strength), 'penalty' (type of regularization), and 'solver' (solver algorithm).

---

```
# Performing grid search with cross-validation: grid_search =
GridSearchCV(lr, param_grid, cv=5) grid_search.fit(X_train, y_train)
```

---

The GridSearchCV class is used to perform grid search with cross-validation. The logistic regression classifier (lr), parameter grid (param_grid), and the number of cross-validation folds (cv=5) are specified. The fit method is then called on the grid_search object to perform the hyperparameter search and cross-validation training.

---

```
# Getting the best model and its parameters: best_model =
grid_search.best_estimator_ best_params =
grid_search.best_params_
```

---

Once the grid search is complete, the best_estimator_ attribute of the grid_search object returns the best model obtained during the search. The best_params_ attribute provides the set of hyperparameters that produced the best model performance.

---

```
# Making predictions on the test set: y_pred =
best_model.predict(X_test)

# Calculating the classification report: classification_rep =
classification_report(y_test, y_pred) print(classification_rep)
```

---

The classification_report function is used to generate a report with various classification metrics such as precision, recall, F1-score, and support. It compares the predicted labels (y_pred) with the true labels (y_test) from the test set. The resulting report is stored in the classification_rep variable and printed.

By running this modified code, you will perform grid search with cross-validation to find the best hyperparameters for logistic regression. The resulting best model will be used to make predictions on the test set, and the classification report will provide detailed performance metrics.

The additional code snippet you provided demonstrates how to save the trained decision tree classifier model using the pickle module. Here's an explanation of the code:

---

```
# Importing the pickle module: import

pickle model = best_model

# Defining the file path to store the model:
model_file = '/Users/himeshagrawal/Downloads/assn2/model_lr.pkl'
# Saving the model to a file:
with open(model_file, 'wb') as file: pickle.dump(model, file)
```

---

The "with" statement is used to open the file in write mode ("wb" - write bytes). The "pickle.dump" function is then used to serialize the "model" object and write it to the opened file. This process saves the model as a binary file with the ".pkl" extension.

After running this code snippet, the trained decision tree classifier model will be saved to the specified file path as a pickled object. You can later load this saved model using the pickle module for further use or deployment.

## 2.1 Model Selection:

| Model Name | Parity Score | Model Size | Time per image |
|---|---|---|---|
| Decision Tree | 1.0 | 4 KB | 0.478 Sec |
| Logistic Regressions | 1.0 | 81 KB | 0.383 Sec |
| SVM | 1.0 | 12,083 KB | 0.357 Sec |
| K-Nearest Neighbour | 1.0 | 16,384 KB | 0.366 Sec |

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| EVEN | 0.97 | 0.98 | 0.98 | 112 |
| ODD | 0.98 | 0.97 | 0.97 | 88 |
| accuracy | | | 0.97 | 200 |
| macro avg | 0.98 | 0.97 | 0.97 | 200 |
| weighted avg | 0.98 | 0.97 | 0.97 | 200 |

Figure 1: Classification report for Decision Tree

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| EVEN | 1.00 | 1.00 | 1.00 | 112 |
| ODD | 1.00 | 1.00 | 1.00 | 88 |
| accuracy | | | 1.00 | 200 |
| macro avg | 1.00 | 1.00 | 1.00 | 200 |
| weighted avg | 1.00 | 1.00 | 1.00 | 200 |

Figure 2: Classification report for Logistic Regression

## 2.2 Logistic regression (Implemented):

Logistic regression is a popular statistical model used for binary classification tasks, where the goal is to predict the probability of an instance belonging to a certain class. Despite its name, logistic regression is a classification algorithm, not a regression algorithm.

Here's a detailed explanation of how logistic regression works:

Assumptions: Logistic regression assumes that the relationship between the features (input variables) and the logodds of the target variable (dependent variable) is linear. It also assumes that the errors or residuals of the model follow a logistic distribution.

Model Representation: In logistic regression, the target variable is typically binary, taking values like 0 or 1. The logistic regression model represents the probability of the target variable being 1 given the input features, denoted as

$$P(y = 1|x)$$

The probability of the target variable being 0 is given by

$$P(y = 0|x) = 1 - P(y = 1|x)$$

Logistic Function (Sigmoid Function): To model the relationship between the input features and the target variable, logistic regression uses the logistic function, also known as the sigmoid function. The sigmoid function maps any real-valued number to a value between 0 and 1, which can be interpreted as a probability. The sigmoid function is defined as follows:

$\sigma(z) = 1/(1+ e^{(-z)})$ where z is the linear combination of input features and their corresponding weights, plus a bias term.

Linear Combination: The linear combination, denoted as z, is calculated as the dot product of the input features (x) and their corresponding weights ($\theta$), plus the bias term ($\theta_0$):

$z = \theta_0 + \theta_1 * x_1 + \theta_2 * x_2 + \ldots + \theta_n * x_n$

Here, $\theta_0$ is the intercept term, and $\theta_1$, $\theta_2$, ..., $\theta_n$ are the weights associated with the input features $x_1, x_2, \ldots, x_n$.

Log-Odds: The log-odds (logit) is the natural logarithm of the odds ratio, and it represents the linear relationship between the input features and the target variable. The log-odds (logit) is given by:

$logit(p) = log(p/(1 - p)) = z$ where p is the probability of the target variable being 1 is $(P(y = 1|x))$.

Maximum Likelihood Estimation: The logistic regression model is trained using the maximum likelihood estimation (MLE) technique. The goal is to find the set of weights ($\theta$) that maximizes the likelihood of the observed data. This involves optimizing a cost function called the log-loss or cross-

entropy loss, which measures the difference between the predicted probabilities and the actual class labels.

Decision Boundary: Once the logistic regression model is trained and the weights are learned, a decision boundary is established to classify new instances. The decision boundary is a threshold probability (usually 0.5) above which an instance is classified as belonging to class 1, and below which it is classified as belonging to class 0.

Logistic regression has several advantages, including simplicity, interpretability, and efficiency.

## 3. Miscellaneous:

Training Algorithm used: Logistic Regression as discussed in Section 1.2 and Section 2.2
Hyperparameter Search & Validation Procedures: GridSearch with cross-validation as discussed in Section 1.2 and Section 2.1

## 4. References:


[1] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[2] "Pattern Recognition and Machine Learning" by Christopher M. Bishop. Chapter 4 specifically focuses on classification and logistic regression

[3] OpenCV Documentation - https://opencv.org/

[4] OpenCV Tutorials - https://www.geeksforgeeks.org/opencv-python-tutorial/

[5] ChatGPT - https://chat.openai.com/

[6] Python Tutorials - https://www.w3schools.com/python/