جامعة نيويورك أبوظبي

NYU | ABU DHABI

Error Correcting Algorithms for Image Reconstruction and Probabilistic Analysis: Repetition and Hamming Codes

Under the kind guidance of
Professor Saif Eddin Jabari

Aman Sunesh
Sheena Chiang
Demarce Williams
Seoyoon Jung
Damiane Kapanadze
Akram Akhmetov
Bekbash Yernar

Course Name:
Probability and Statistics for Engineers
Section 002
Course Code: ENGR-UH_2010Q
Fall 2024

New York University Abu Dhabi
October 11, 2024

Contents

1 Introduction

1.1 Objective and Scope

This project investigates error-correcting codes in digital image processing, focusing on improving the reliability of image transmission in noisy environments. The implementation, testing, and analysis of two popular algorithms—the repetition and hamming codes—are discussed in detail. In addition, a quantitative examination is done with probabilistic models of random variables. Chebyshev's and Chernoff's inequalities are used to determine the efficiency of the algorithms in managing noise and corrupted data.

1.2 Problem Motivation & Applications

Satellites launched into outer space have to send binary data of the images they take in outer space across hundreds of millions of kilometers, subjecting this information to background electromagnetic waves that potentially tamper the bits, randomly flipping 0s to 1s and 1s to 0s. Such interference can result in noise that blurs the final transmitted image if left uncorrected. However, images from outer space are rare and precious information that are combed and analyzed thoroughly. Thus, it is ideal for satellite images to be as clear as possible.

In order to combat data interference, Error Correcting Codes (ECCs) are utilized to retrieve the original data without errors. They provide receivers on Earth with the ability to extract bits as close to the transmitted bits as possible, allowing acquired satellite images to be clearer, more accurate, and true to form. There are several error-correcting algorithms, including Repetition Codes, Hamming Code, and Reed-Solomon. These algorithms are commonly used to correct bit-flip errors in everyday applications. The most widely used is the Reed-Solomon algorithm, which is implemented in QR codes, wireless communication systems such as NASA's Deep Space Network [1], and data storage technologies like flash memory [2].

2 Implementation

In this project, we experiment with both repetition codes and Hamming codes to correct errors in transmitted images.

2.0.1 Software and Tools

- **Programming Language:** Python, due to its extensive libraries and ease of handling image data and binary operations. And unlike Matlab, it is open source, so anyone with a laptop and access to the Internet can run our program.
- **Libraries:**
  - **random** for generating random numbers to add noise to image.
  - **NumPy** for numerical operations.
  - **PIL (Python Imaging Library)** for handling image loading and processing.
  - **Matplotlib** for displaying images.

2.1 Repetition

2.1.1 Overview of Repetition Codes

Repetition codes are a basic yet effective method for correcting errors in data transmission, especially when data must travel through noisy channels that might corrupt the information. In this project, we specifically focus on a "Repetition (3,1) code" strategy.

Repetition (3,1) Code: This method involves taking each individual bit from the original data and replicating it three times. For instance, if a bit in the original data is '1', it would be encoded as '111'. Similarly, a '0' would be encoded as '000'. This replication increases the robustness of the data against potential disruptions during transmission.

By replicating each bit, we create redundancy. This redundancy is beneficial because it allows the receiving end of a communication system to detect and correct errors. If noise affects the transmission and changes one of the bits, the original bit can still be recovered through a process known as majority voting. For example, if the original trio '111' is corrupted to '101' during transmission, the system can determine the intended bit was likely a '1' because '1' appears more frequently than '0'.

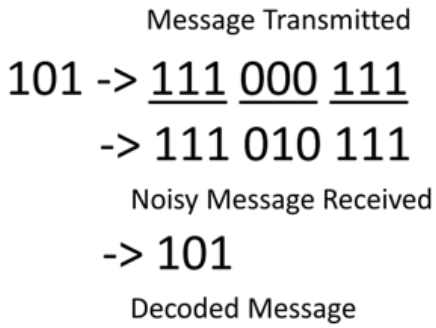


Figure 1: Example of Repetition Logic

2.1.2 Application in Image Processing

A digital image is composed of pixels, the smallest unit of an image displayed on digital devices. Each pixel represents a color, encoded through combinations of Red, Green, and Blue (RGB) values, typically ranging from 0 to 255, represented in the form of a tuple as (R, G, B). These values define the intensity of the colors in the pixel, where (0,0,0) represents black and (255,255,255) represents white. For the purpose of implementing repetition codes, these RGB tuples provide a structured way to access and manipulate the color information of each pixel. By treating each component of the tuple as a separate entity, we can apply repetition codes to each color channel independently.

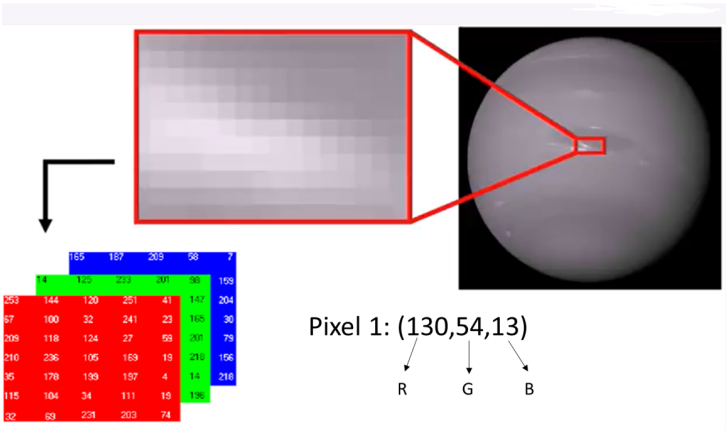


Figure 2: Image Composition: Pixel Structure and RGB Components

2.1.3 Encoding Process

The initial step in the encoding process involves decomposing the original image into three separate images, corresponding to the Red, Green, and Blue components. This was achieved using Python's NumPy library to manipulate the image array and PIL library to handle image operations. The encoding process involved the following steps:

Reading the Original Image: The original image was loaded into a NumPy array, preserving its RGB components.

```
img_name = input("Enter Image Name with extension: ")
original_image_array = np.array(Image.open(img_name))
height, width, channels = original_image_array.shape
```

Creating Redundant Color Images: The image array `original_image_array` is split into three separate arrays, each corresponding to one of the RGB components.

- **Red Channel:** `Red_Image = np.stack([original_image_array[:,:,0]]*3, axis=-1)`
- **Green Channel:** `Green_Image = np.stack([original_image_array[:,:,1]]*3, axis=-1)`
- **Blue Channel:** `Blue_Image = np.stack([original_image_array[:,:,2]]*3, axis=-1)`

Here, `original_image_array[:,:,0]` extracts all the red values across the image, `[:,:,1]` does the same for green, and `[:,:,2]` for blue. The function `np.stack([...]*3, axis=-1)` replicates each color channel three times along the last axis. This operation effectively creates three separate images—each containing only one color component (Red, Green, or Blue) of the original image, with each pixel's color value repeated three times for redundancy.

The aim here is straightforward: by repeating each color value three times, we create an opportunity to correct errors for each color component in every pixel through majority

voting at the decoding stage. If a single color value is altered due to noise or error, the other two can vote it out, thus preserving the original color information.

Saving the Encoded Images: Each encoded image, representing the red, green, and blue components of the original image, was saved separately.

```
# Convert arrays to images, and save or display
red_img = Image.fromarray(Red_Image)
green_img = Image.fromarray(Green_Image)
blue_img = Image.fromarray(Blue_Image)

# Save the images
red_img.save('Red_Image.png')
green_img.save('Green_Image.png')
blue_img.save('Blue_Image.png')
```

2.1.4 Noise Introduction

Once the images representing the Red, Green, and Blue components were encoded and saved, the next step involved the intentional introduction of noise. This process simulates the potential disturbances that might occur during the transmission of these images.

The encoded Red, Green, and Blue images were passed through a classification tool available at Hugging Face's FGSM Project (huggingface.co/spaces/niniack/fgsm-project), which simulates noise by performing a Fast Gradient Sign Method (FGSM) attack. This process not only adds perturbations to the images, mimicking real-world interference, but also classifies each image as an object. After adding noise, the altered images are usually misclassified, demonstrating the impact of the perturbations and highlighting the need for error correction. The noisy images were then saved as `Red_Image_With_Noise.png`, `Green_Image_With_Noise.png`, and `Blue_Image_With_Noise.png`, setting the stage for the decoding and error-correction phase.

2.1.5 Decoding Process

After saving the noisy images (Red, Green, and Blue), the next step is to decode them and correct any errors introduced by the noise. The goal of decoding is to restore the original pixel values by fixing the errors in each color component of the noisy images.

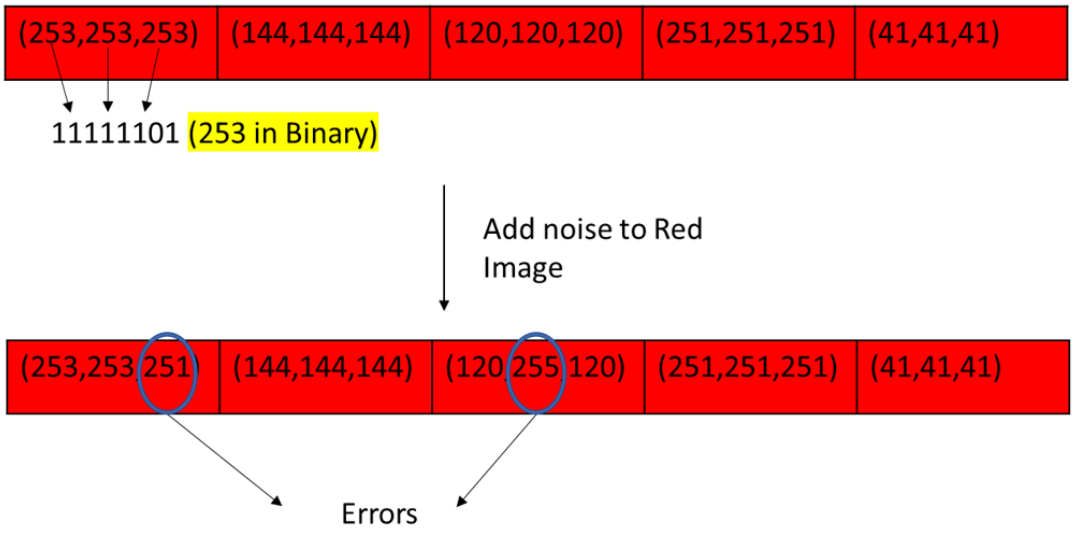


Figure 3: Example of the Impact of Noise on R, G, and B Values

Converting Noisy Images to Binary: The R, G, and B color components in the noisy Red, Green, and Blue images was converted into an 8-bit binary string for each color channel (R, G, B). This binary representation simplifies the process of applying majority voting to detect and correct errors in the noisy images. For example, a pixel value of 253 would be represented as the binary string `11111101`, allowing us to perform bit-level operations. The function `convert_to_binary` is designed to transform each RGB value into a binary format.

```
def convert_to_binary(image_array):
    # Get dimensions
    height, width, _ = image_array.shape

    # Initialize binary image array
    binary_image = np.zeros((height, width, 3), dtype='U8')

    # Convert each RGB value to a binary string
    for i in range(height):
        for j in range(width):
            # Red channel
            binary_image[i, j, 0] = format(image_array[i, j, 0], '08b')
            # Green channel
            binary_image[i, j, 1] = format(image_array[i, j, 1], '08b')
            # Blue channel
            binary_image[i, j, 2] = format(image_array[i, j, 2], '08b')

    return binary_image
```

After defining the conversion function, it is applied to the noisy images to prepare them for the error correction process. This step is essential as it ensures that each channel of every pixel is represented in binary form, facilitating the identification and correction of transmission errors through majority voting.

```
# Convert noisy images to binary
binary_red_with_noise = convert_to_binary(red_with_noise)
binary_green_with_noise = convert_to_binary(green_with_noise)
binary_blue_with_noise = convert_to_binary(blue_with_noise)
```

Majority Voting for Error Correction: Now that we have our three noisy images, with each pixel's pseudo Red, Green, and Blue components converted into binary format, we perform majority voting to determine the corrected value for each component in all pixels of each image (Red, Green, and Blue). This process allows us to recover the original signal by evaluating the most common bit among the replicated data at each bit position. If noise has altered any of the bits, majority voting helps in identifying and correcting these errors based on the assumption that the majority of the data (two out of three values) remains unchanged despite the noise.

- For each bit position in the 8-bit binary representation of the three repeated values, we check which bit appears most frequently (0 or 1).
- For example, if the three values are `11111101`, `11111101`, and `11111100`, then for each bit position, we take the majority bit. In this case, the sixth bit might be 1 in two cases and 0 in one case, so the majority bit is 1. This process is repeated for all 8 bits to form the corrected binary string.

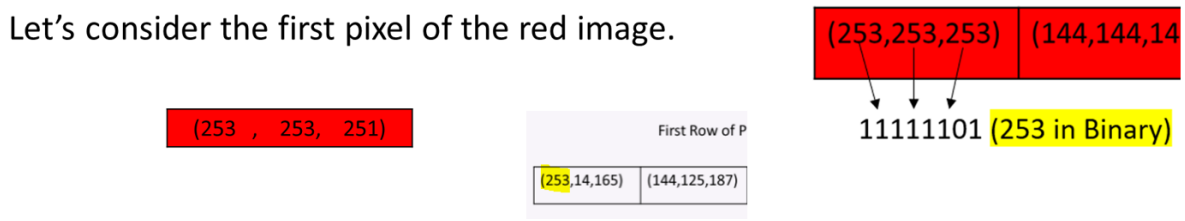
Below is the Python function used for performing majority voting. It iterates over each bit position of the binary strings and uses a counter to tally the most frequent bit at each position, thus determining the corrected value for that bit.

```
def majority_vote(bit_lists):
    voted_binary_string = ''
    for i in range(8):  # For each bit position
        bit_column = [bits[i] for bits in bit_lists]
        bit_counter = Counter(bit_column)
        most_common_bit, _ = bit_counter.most_common(1)[0]
        voted_binary_string += most_common_bit
    return voted_binary_string
```

Converting Corrected Binary Values Back to Integer: The corrected binary string is then converted back into an integer value, which restores the original pixel value for each color channel.

```
for i in range(height):
    for j in range(width):
        red_voted_binary_string = majority_vote(red_bit_lists)
        green_voted_binary_string = majority_vote(green_bit_lists)
        blue_voted_binary_string = majority_vote(blue_bit_lists)

        final_image_array[i, j, 0] = int(red_voted_binary_string, 2)
        final_image_array[i, j, 1] = int(green_voted_binary_string, 2)
        final_image_array[i, j, 2] = int(blue_voted_binary_string, 2)
```



Channels (Integer)	Corresponding Bitstring	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
253	11111101	1	1	1	1	1	1	0	1
253	11111101	1	1	1	1	1	1	0	1
251	11111011	1	1	1	1	1	0	1	1
Majority Voting		1	1	1	1	1	1	0	1

Figure 4: Example of Majority Voting on a Pixel

Repeating for All Pixels: This error-correction process is performed for every pixel in the noisy images for all three color channels (Red, Green, and Blue).

Reconstructing the Final Corrected Image: Once all the pixels are corrected, the corrected Red, Green, and Blue values are combined to form the final decoded image. This restored image should closely match the original image before any noise was added.

```
# Initialize the final image array
final_image_array = np.zeros((height, width, 3), dtype=np.uint8)
```

```
# Convert the numpy array to an image and save or display it
final_image = Image.fromarray(final_image_array, 'RGB')
final_image.save('Corrected_Image.png')
```

2.2 Hamming Code

2.2.1 Overview of Hamming Codes

Hamming codes are error-correcting codes that allow for the detection and correction of single-bit errors. Hamming codes add parity bits to data bits to identify the location of errors, enabling correction without needing the original data for comparison.

The general Hamming code is denoted as Hamming(n, k), where:

- n is the total number of bits (data + parity).
- k is the number of data bits. For example, Hamming(7,4) has 7 total bits: 4 data bits and 3 parity bits.

2.2.2 Encoding Hamming(7,4)

The Hamming(7, 4) method involves calculating three (3) parity bits based on four (4) data bits from the original data set, producing a seven (7) bit output. The parity bits are placed at specific positions in the encoded output to provide a basis for locating and correcting errors.

Suppose we want to encode the 4-bit data `1011` using the Hamming(7,4) method.

Bit Positions:

- The total bit positions are numbered: 1, 2, 3, 4, 5, 6, 7.
- Data bits go in positions 3, 5, 6, and 7.
- Parity bits are placed at positions 1, 2, and 4.

Encoding Structure:

p_1	p_2	d_1	p_3	d_2	d_3	d_4
_	_	1	_	0	1	1

The parity bits are then calculated using the exclusive OR (XOR) operator as follows:

- p_1: Covers bits 1, 3, 5, 7
  $p_1 = d_1 \oplus d_2 \oplus d_4 = 1 \oplus 0 \oplus 1 = 0$
- p_2: Covers bits 2, 3, 6, 7
  $p_2 = d_1 \oplus d_3 \oplus d_4 = 1 \oplus 1 \oplus 1 = 1$
- p_3: Covers bits 4, 5, 6, 7
  $p_3 = d_2 \oplus d_3 \oplus d_4 = 0 \oplus 1 \oplus 1 = 0$

The coverage of each parity bit in Hamming(7,4) encoding is determined by the positions where the binary representation of the bit index has a 1 in specific places (least significant, second, or third least significant bit), as illustrated in the figure below.

Parity Bit	Covers Bit Positions	Binary Representation of Positions	Reason for Coverage	Data Bits Checked
p1	1, 3, 5, 7	001, 011, 101, 111	Positions where the least significant bit is 1	d1, d2, d4
p2	2, 3, 6, 7	010, 011, 110, 111	Positions where the second least significant bit is 1	d1, d3, d4
p3	4, 5, 6, 7	100, 101, 110, 111	Positions where the third least significant bit is 1	d2, d3, d4

Figure 5: Data Bit Assignments for Parity Calculations in Hamming (7,4) Encoding

Note that at least one position is located in each group and the intersections among groups.

Original Message: 1011 $\rightarrow$ **Hamming(7, 4)** $\rightarrow$ 0110011
Encoded Message to be Transmitted: 0110011

2.2.3 Decoding: Hamming(7,4)

During decoding, the receiver checks if the received data has any errors by recalculating the parity (syndrome) bits and comparing them with the received parity bits. If there is a mismatch, the error position is determined by combining the results of the parity checks (similar to a binary number), and the erroneous bit is corrected.

For example, let's say the following bits were received after transmission: 0110001.

The syndrome bits are recalculated as follows:

- p_1: Covers bits 1, 3, 5, 7
  $p_1 = d_1 \oplus d_3 \oplus d_5 \oplus d_7 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$
- p_2: Covers bits 2, 3, 6, 7
  $p_2 = d_2 \oplus d_3 \oplus d_6 \oplus d_7 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$
- p_3: Covers bits 4, 5, 6, 7
  $p_3 = d_4 \oplus d_5 \oplus d_6 \oplus d_7 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$

The error is located at the intersection of the parity bits, based on their combination as shown in the Venn diagram in the figure below.

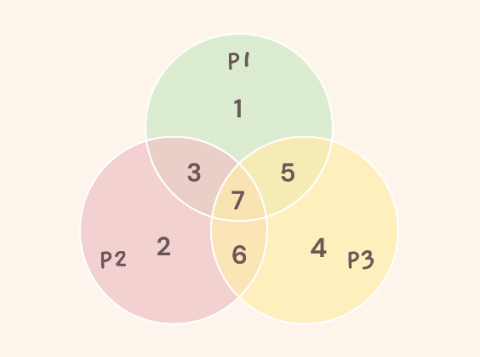


Figure 6: Venn Diagram of Error Positions for Hamming (7,4) Decoding

Since parity bits p_2 and p_3 both produced a 1, the error is located at their intersection.

Error located at position: 6

0110001 (received) $\rightarrow$ single-bit error corrected to 0110011

2.2.4 Application in Image Correction

Digital images are composed of pixels in an array. Pixels in an RGB image are composed of three 8-bit values, one for each color channel (Red, Green, Blue). The Hamming (7,4) method can be used to detect and correct single-bit errors in each pixel of an image. The Hamming code is applied to each color channel (red, green, blue) separately, allowing for easier manipulation of pixels while maintaining the image dimensions.

2.2.5 Encoding Image Pixels

Each pixel value (8 bits) is split into two halves: upper (first 4 bits) and lower (last 4 bits). Each half is encoded using Hamming (7,4) encoding, which generates 7 bits of encoded data from the original 4 data bits.

```
upper_half = binary_string[:4]  # First four bits
lower_half = binary_string[4:]  # Last four bits

encoded_upper = hamming_encode(upper_half_array)
encoded_lower = hamming_encode(lower_half_array)
```

After encoding, the resulting 7-bit values for the upper and lower halves are concatenated and padded to fit the 24-bit RGB format for image storage. This is done for each pixel in the red, green, and blue channels.

```
pseudo_red_value = int("".join(map(str, encoded_combined_with_zeros[:8])), 2)
pseudo_green_value = int("".join(map(str, encoded_combined_with_zeros[8:16])), 2)
pseudo_blue_value = int("".join(map(str, encoded_combined_with_zeros[16:24])), 2)
```

2.2.6 Noise Introduction

After encoding, noise is introduced to simulate corruption in the image. The noise introduction code simulates errors in an image by flipping random bits in the binary representation of pixel values, mimicking real-world transmission or storage errors.

1. Pixel-Level Noise Introduction:

- The code iterates over each pixel in the image (`for i in range(height)` and `for j in range(width)`), accessing the pixel's value.
- Each pixel value, which is an 8-bit number (a color channel value), is converted into its binary form using `format(pixel_value, '08b')`.

2. Random Bit Flip:

- A random bit in this 8-bit binary string is chosen using `random.randint(0, 7)` and is flipped (changed from 0 to 1 or from 1 to 0).
- This simulates random errors occurring during data transmission.

3. Reconversion and Saving:

- The modified binary string is then converted back to an integer and stored in the image array (`noisy_image[i, j] = noisy_pixel`).
- The final noisy image is saved in the same format.

2.2.7 Decoding the Noisy Image

Once noise has been introduced, the noisy image needs to be decoded and corrected using the same Hamming logic. Each pixel is split again into upper and lower halves, and the encoded bits are extracted and decoded to recover the original data bits.

```
upper_bits = bitstring[0:7]  # First 7 bits for the upper half
lower_bits = bitstring[7:14]  # Next 7 bits for the lower half

decoded_upper = hamming_decode(upper_bits_array)
decoded_lower = hamming_decode(lower_bits_array)
```

The decoded data bits are then concatenated to form the corrected pixel value, which is stored in the final decoded image array.

```
corrected_value = int("".join(map(str, decoded_combined)), 2)
decoded_image[i, j, color_index] = corrected_value
```

2.3 Saving the Decoded Image

Finally, the decoded and corrected image is saved to a file, restoring the original pixel values after correcting any single-bit errors.

```
decoded_img = Image.fromarray(decoded_image)
decoded_img.save("decoded_image.png")
```

3 Mathematical Analysis

3.1 Repetition

3.1.1 Problem Setup

Consider a binary communication channel where each bit may be flipped with a probability p. To improve reliability, a repetition code is applied where each bit is repeated k times. After receiving the repeated bits, majority voting is used to determine the original bit. Our goal is to find the minimum number of repetitions k such that the probability of failing to correct an error (i.e., more than half of the bits are flipped) is below a threshold Q.

3.1.2 Random Variable Setup

Let X_i be independent Bernoulli random variables representing whether a bit is flipped. Specifically,

$$X_i = \begin{cases} 1 & \text{if the bit is flipped,} \\ 0 & \text{if the bit is not flipped.} \end{cases}$$

The probability of a bit being flipped is p, so $\mathbb{P}(X_i = 1) = p$ and $\mathbb{P}(X_i = 0) = 1 - p$.

Let $S = X_1 + X_2 + \cdots + X_k$, the total number of flipped bits after k repetitions. Thus, S follows a binomial distribution:

$$S \sim \text{Binomial}(k, p).$$

3.1.3 Probability Setup

We are interested in the probability of **not being able to detect/correct the error**. In other words, we are interested in finding the probability that the flipped bits occur over the majority so that the error could not be identified.

Mathematically, this translates to the probability that during more than $n/2$ of the repetition result in a flipped bit. Thus, we have:

$$\mathbb{P}(S > \frac{k}{2})$$

Using Chebyshev's inequality, we can find an appropriate value for k such that this probability is below a given threshold Q.

3.1.4 Chebyshev's Inequality Application

Chebyshev's inequality provides a bound for the probability that a random variable deviates from its mean. For a random variable X with mean μ and variance σ^2, the inequality states:

$$\mathbb{P}(|X - \mu| \geq a) \leq \frac{\sigma^2}{a^2}.$$

We will apply this inequality to the binomial random variable S.

1. **Mean and Variance**
   For $S \sim \text{Binomial}(k, p)$, the mean and variance are given by:

$$\mu_S = \mathbb{E}[S] = kp, \quad \sigma_S^2 = \text{Var}(S) = kp(1 - p).$$

2. **Threshold for Error Detection**
   The error occurs when more than half of the bits are flipped, i.e., when $S > \frac{k}{2}$. We define the deviation from the mean as $a = \frac{k}{2} - kp$, which represents the difference between the majority threshold $\frac{k}{2}$ and the mean kp. Applying Chebyshev's inequality:

$$\mathbb{P}\left(S > \frac{k}{2}\right) = \mathbb{P}\left(|S - \mu_S| > \frac{k}{2} - kp\right) \leq \frac{\sigma_S^2}{\left(\frac{k}{2} - kp\right)^2}.$$

   Substituting the mean kp and variance $\sigma_S^2 = kp(1-p)$:

$$\mathbb{P}\left(|S - kp| > \frac{k}{2} - kp\right) \leq \frac{kp(1-p)}{\left(\frac{k}{2} - kp\right)^2}.$$

3. **Finding the Optimal Number of Trials k**
   To ensure the probability of failing to correct the error is below a threshold Q, we require:

$$\frac{kp(1-p)}{\left(\frac{k}{2} - kp\right)^2} \leq Q.$$

   Simplifying this inequality:

$$k \geq \frac{p(1-p)}{Q\left(\frac{1}{2} - p\right)^2}.$$

   This gives the minimum number of repetitions k necessary to ensure the probability of not detecting the error is less than Q.

4. **Example Calculation**
   Let us consider a specific case where the bit-flip probability is $p = 0.3$ and we want the probability of not detecting an error to be below $Q = 0.01$ (1%). Using the formula derived above:

$$k \geq \frac{0.3(1-0.3)}{0.01\left(\frac{1}{2} - 0.3\right)^2}$$

$$k \geq \frac{0.3 \times 0.7}{0.01 \times 0.2^2} = \frac{0.21}{0.0004} = 525.$$

   Thus, the minimum number of repetitions required is $k = 525$.

3.1.5 Chernoff's Bound Application

1. **Mean and Variance**
   For $S \sim \text{Binomial}(k, p)$, the mean and the relative deviation from mean (δ) are given by:

$$\mu_S = \mathbb{E}[S] = kp, \quad \delta = \frac{value - mean}{mean} = \frac{\frac{n}{2} - np}{np}$$

2. **Threshold for Error detection**
   Similarly, we calculate the probability of $S > \frac{k}{2}$. Applying Chernoff's bound for binomial distribution:

$$\mathbb{P}(S > \frac{n}{2}) = \mathbb{P}(S > (1+\delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2+\delta}}$$

3. **Finding the Optimal Number of Trials k**
   To ensure the probability of failing to correct the error is below a threshold Q, we require:

$$exp(-\frac{(\frac{\frac{n}{2}-np}{np})^2 np}{2+\frac{\frac{n}{2}-np}{np}}) \leq Q$$

   Simplifying this inequality:

$$k \geq \frac{-\ln(Q)\left(\frac{1}{2p}+1\right)}{p\left(\frac{1}{2p}-1\right)^2}$$

   Again, this gives the minimum number of repetitions k necessary to ensure the probability of not detecting the error is less than Q.

4. **Example Calculation**
   Let us consider the same example situation from above, using $p = 0.3$ and $Q = 0.01$. Using the formula derived above:

$$k \geq \frac{-\ln(0.01)\left(\frac{1}{2\cdot 0.3}+1\right)}{0.3\left(\frac{1}{2\cdot 0.3}-1\right)^2}$$

$$k \geq \frac{-\ln(0.01)\left(\frac{1}{0.6}+1\right)}{0.3\left(\frac{1}{0.6}-1\right)^2} = 92.1$$

   Thus, Chernoff's bound suggests that the minimum number of repetitions required is $k = 92.1$.

5. **Observations**
   Chernoff's bounds suggest fewer repetitions (92.1) compared to Chebyshev's inequality (525) for the same p and Q values, which is a consistent trend through the increase in the p values. It is worth noting that Cheybysehv's inequality is general and only requires knowledge of the variance, but doesn't leverage the specific independent structure or tail behavior of binomial sums. Meanwhile, Chernoff's bound is much tighter since it targets the tail probability with an exponential decrease of the upper bound and accounts for the independence of the Bernoulli variables. This allows us to avoid the generality of Chebyshev's overestimating bound.
   Plots below show the minimum number of repetitions needed (n) as bit-flip probability (p) changes. Both inequalities display a sharp shoot in n near $p = 0.5$ where the bit-flip is random, which makes it difficult to reach the threshold accuracy. However, Chernoff's y-axis is in increments of $0.5e5$, compared to $0.2e6$ in Chebyshev's.

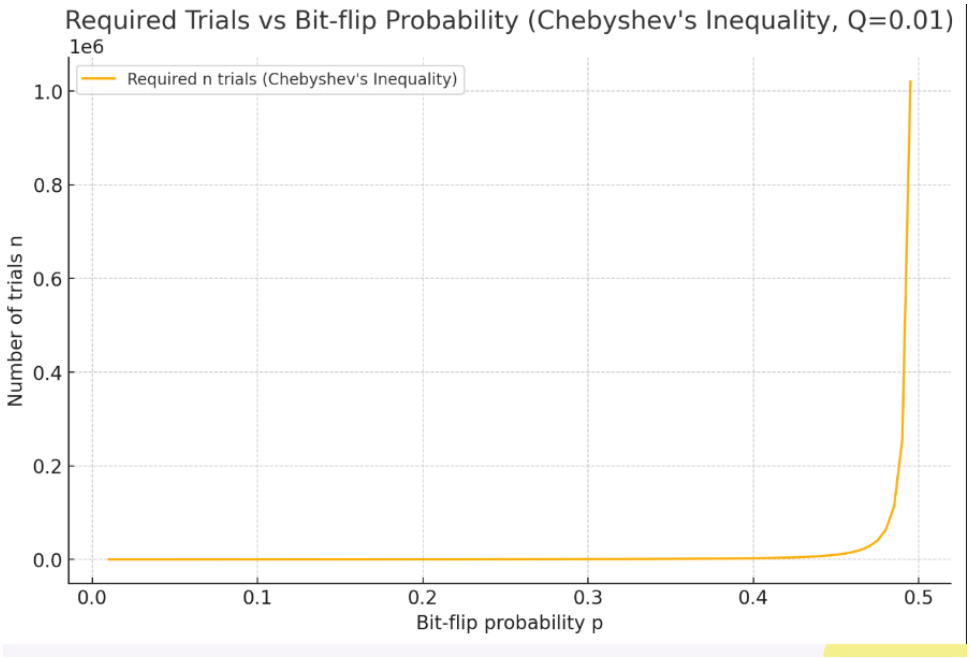


Figure 7: Trend of p versus n in Chebyshev's inequality

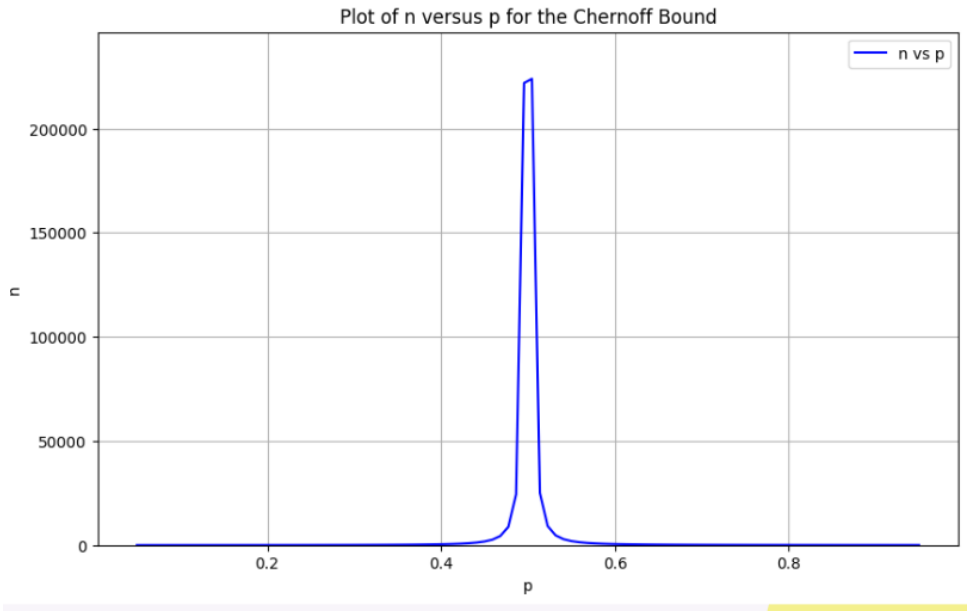


Figure 8: Trend of p versus n in Chernoff's inequality

3.2 Hamming Code

3.2.1 Problem Setup

To simplify the calculations while still illustrating the concept, let's assume we are transmitting 4 data bits, and for error detection and correction, we are useing the Hamming (7,4) code. This means that we add 3 parity bits to the 4 data bits, resulting in a total of 7 bits being sent. Additionally, let's assume the probability of a bit flipping during transmission is p.

3.2.2 Probability Analysis

Hamming code can correct up to 1 bit flip, so the overall probability of the code working will be the sum of two scenarios:

- **No bit flips:** All bits are transmitted correctly.
- **One bit flip:** Exactly one bit is flipped, which Hamming code can detect and correct.

Thus, the probability of Hamming code working is the sum of the probability of no bit flips and the probability of exactly one bit flip.

The probability of no bit flips is:

$$P(\text{no bit flips}) = (1-p)^7$$

This is because each bit has a $1-p$ chance of not flipping, and there are 7 bits in total.

The probability of exactly one bit flipping is:

$$P(1 \text{ bit flip}) = \binom{7}{1} \cdot p \cdot (1-p)^6 = 7p(1-p)^6$$

Therefore, the total probability of Hamming code working (either no flips or one correctable flip) is:

$$P(\text{Hamming code working}) = (1-p)^7 + 7p(1-p)^6$$

Now, let's consider a scenario where no error correction is applied. In this case, we are sending the 4 data bits directly without any additional protection.

The probability that each bit is transmitted correctly (i.e., without flipping) is $1-p$. Since there are 4 data bits, the probability that all 4 bits are transmitted correctly is $(1-p)^4$.

Therefore, the probability of the entire message being communicated correctly is:

$$P(\text{correct communication}) = (1-p)^4$$

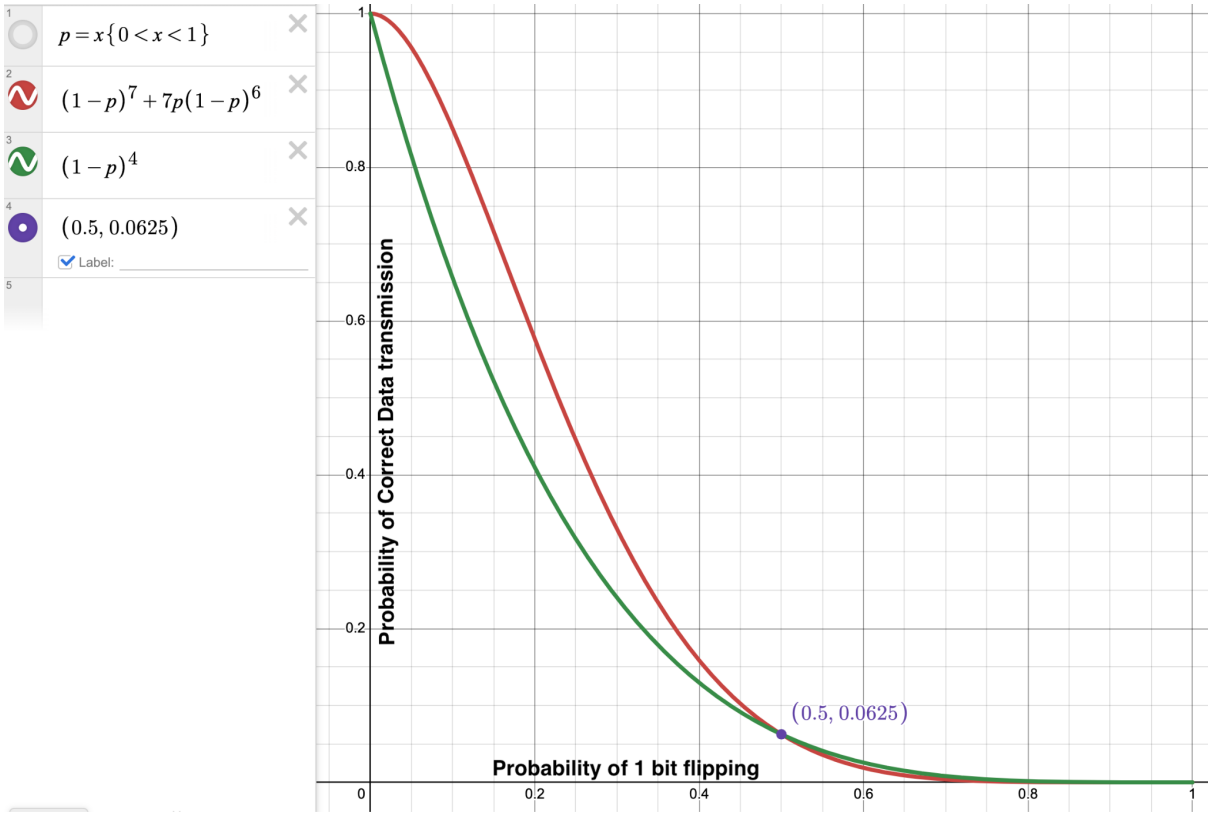


Figure 9: Comparison of Probability of Correct Communication

In the graph above:

- The **X-axis** represents p, which is the probability of a bit flipping during transmission.
- The **Y-axis** represents the probability of correct communication.
- The **red curve** shows the probability of correct communication when using Hamming code, which can correct up to 1 bit error.
- The **green curve** shows the probability of correct communication when the 4 data bits are transmitted directly, without any error correction.
- The graphs intersect at the **purple** point $(0.5, 0.0625)$.

The graph illustrates that using Hamming code **(red curve)** has better probability of correct communication compared to directly sending the data bits **(green curve)**, when p is less than 50%. And this happens because once the probability of a bit flip gets past 50%, the Hamming Code will start flipping $p(1-p)^6$ becomes so small that multiplying it 7 times doesn't compensate for the higher power / more bits. And if more than 2 bits flip, the Hamming code ends up flipping another one by "correcting" it.

But, in real life, the probability of bits flipping due to cosmic rays is very low, approximately 3.1×10^{-3} per transistor per year [3]. Therefore, let's zoom in on the X-axis of the graph to better visualize this range of low probabilities.

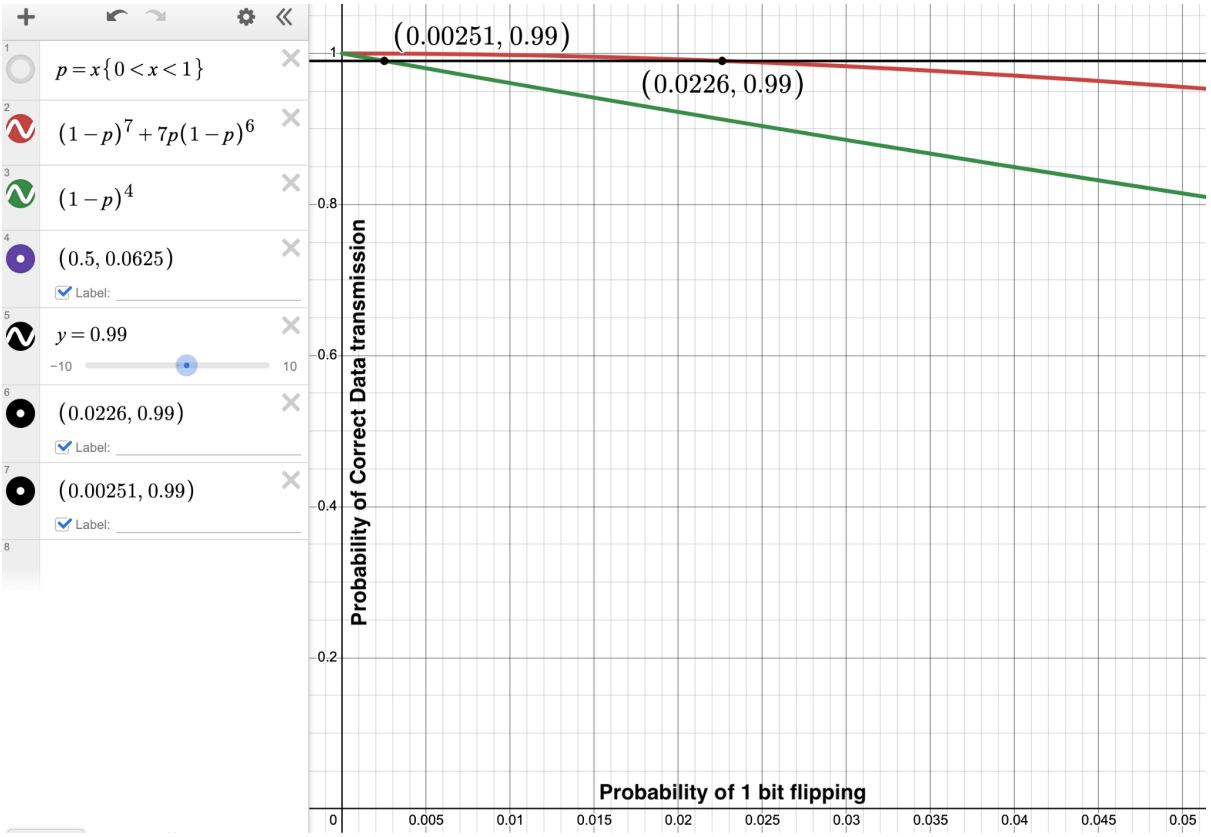


Figure 10: Zoomed-in view of the probability of correct data transmission vs. probability of bit flipping. The horizontal line at $y = 0.99$ indicates a 99% target accuracy. The points (0.00251, 0.99) and (0.0226, 0.99) highlight the conditions under which each method achieves 99% accuracy. And it is clear that the Hamming Code requires 10 times less probability of each bit flipping to transfer the data with 99% accuracy.

4 Results and Testing

4.1 Repetition

To evaluate the effectiveness of the repetition code, we conducted testing using the following image of a strawberry. This image was chosen because of its simple shape, vibrant colors and fine details, which make it suitable for assessing the accuracy of the decoding process.

Figure 11: Test Image

Next, we pass the original image to the classifier to determine if it is accurately recognized as a strawberry.

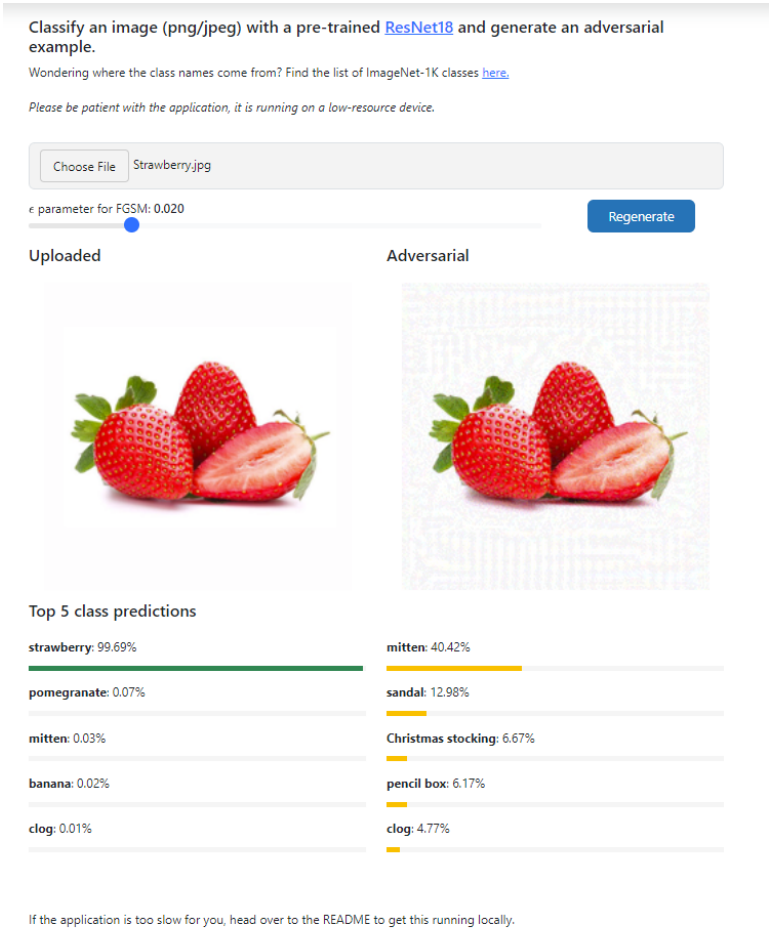


Figure 12: Original Classification of Test Image

The 99.69% strawberry prediction implies that the classifier is highly confident that the image belongs to the "strawberry" category.

Now, we split the original image into three images, each containing only its red, green, or blue channel, repeated three times.

Figure 13: Red, Green, and Blue Channel Images Before Noise

The encoded Red, Green, and Blue images were then passed through the noise simulation platform, where perturbations were added using the FGSM attack. This step simulated real-world interference, such as transmission errors, resulting in noisy versions of the strawberry image. When passed through the noise simulator, the platform displays the original classification of each image on the left and the classification of the image after adding noise on the right. The classifications of the Red, Green, and Blue channel images before and after adding noise are shown below.

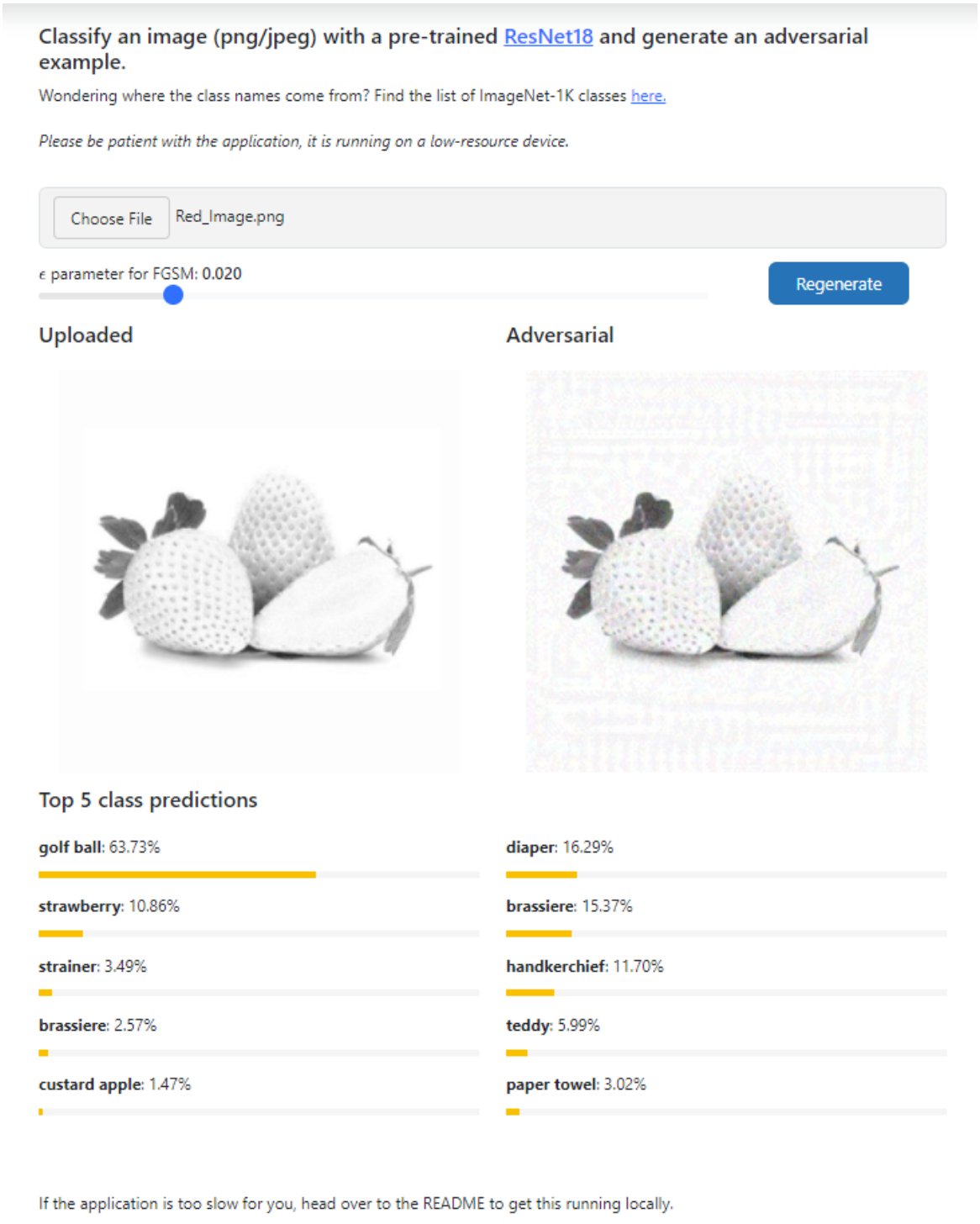


Figure 14: Red Channel Classification: Before and After Noise

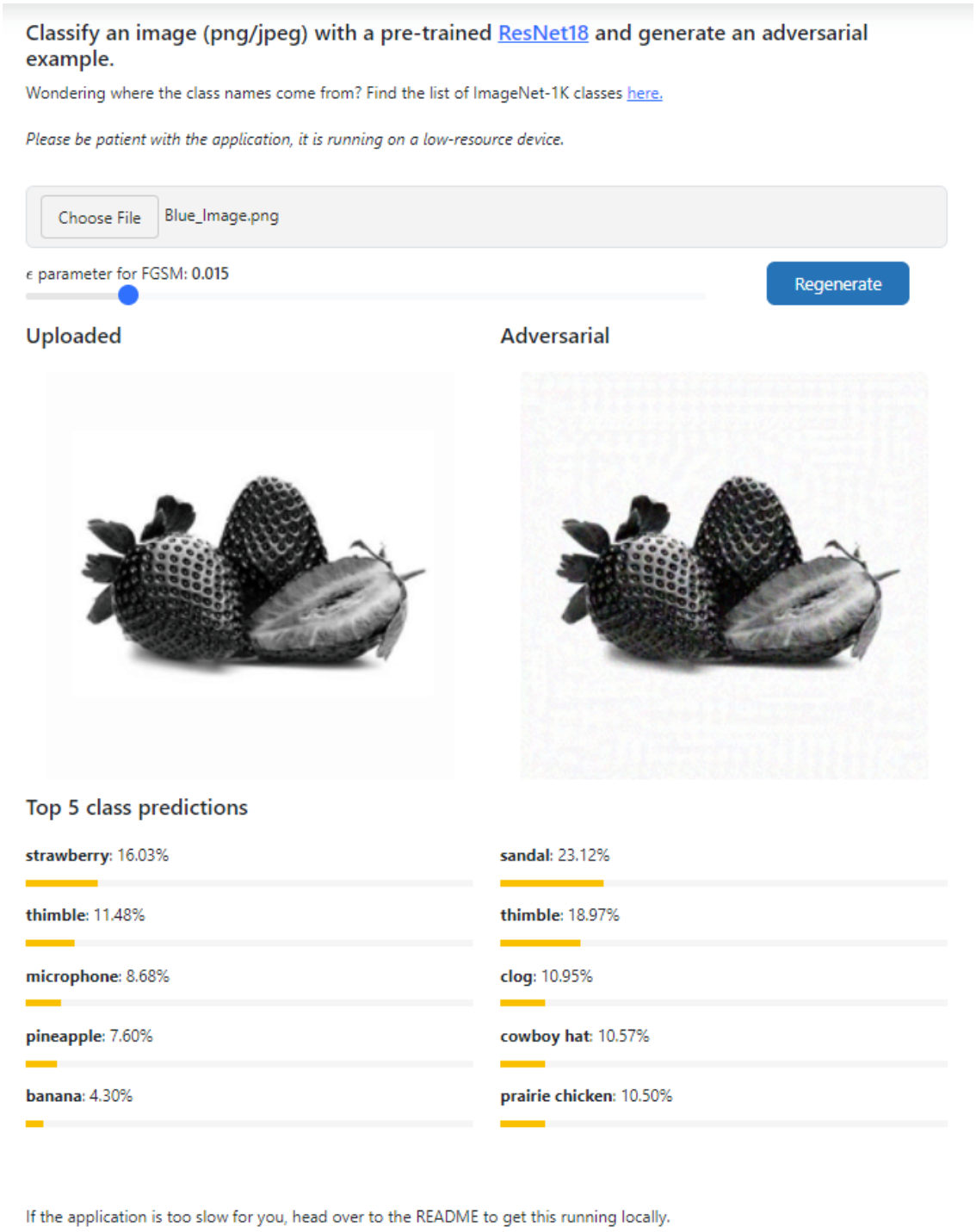


Figure 15: Blue Channel Classification: Before and After Noise

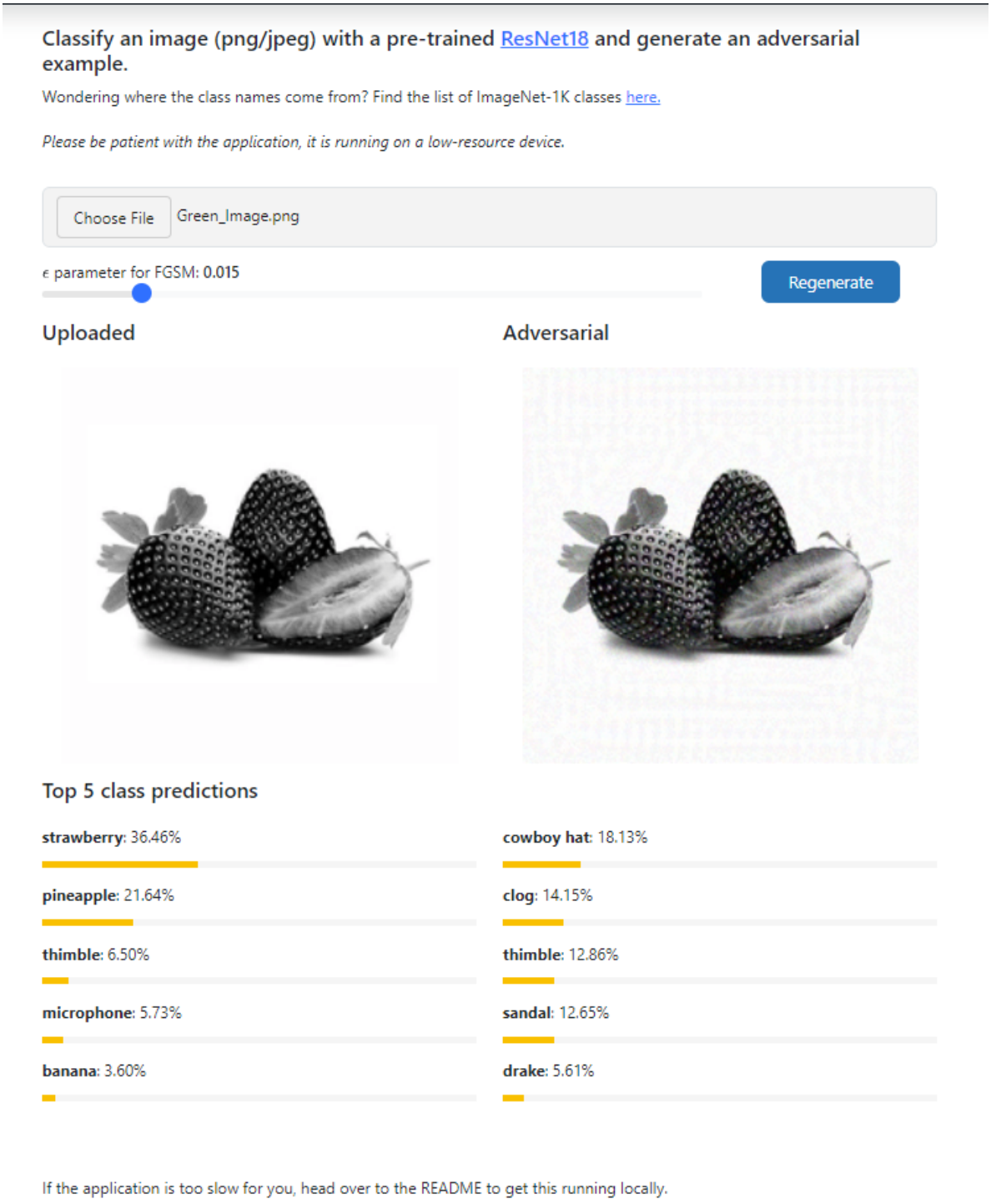


Figure 16: Green Channel Classification: Before and After Noise

Here, for example, for the green channel, it can be seen that even without the red and blue components, the image is still accurately identified as a strawberry with a confidence of 36.46%. However, as soon as noise is added to the image, the prediction changes, and it is now classified as a cowboy hat, which we need to correct.

The following are the red, green and blue channel images after adding noise

Figure 17: Red, Green, and Blue Channel Images After Noise

The noisy images were decoded using majority voting to correct pixel errors in each channel. After correcting all three channels and combining them, the final corrected image was obtained, as shown below:

Figure 18: Corrected Image

Clearly, the image still has some perturbations; however, it is significantly improved.

Finally, we passed the corrected image through the classifier to evaluate the effectiveness of the repetition code and to determine whether it would classify it as a strawberry.

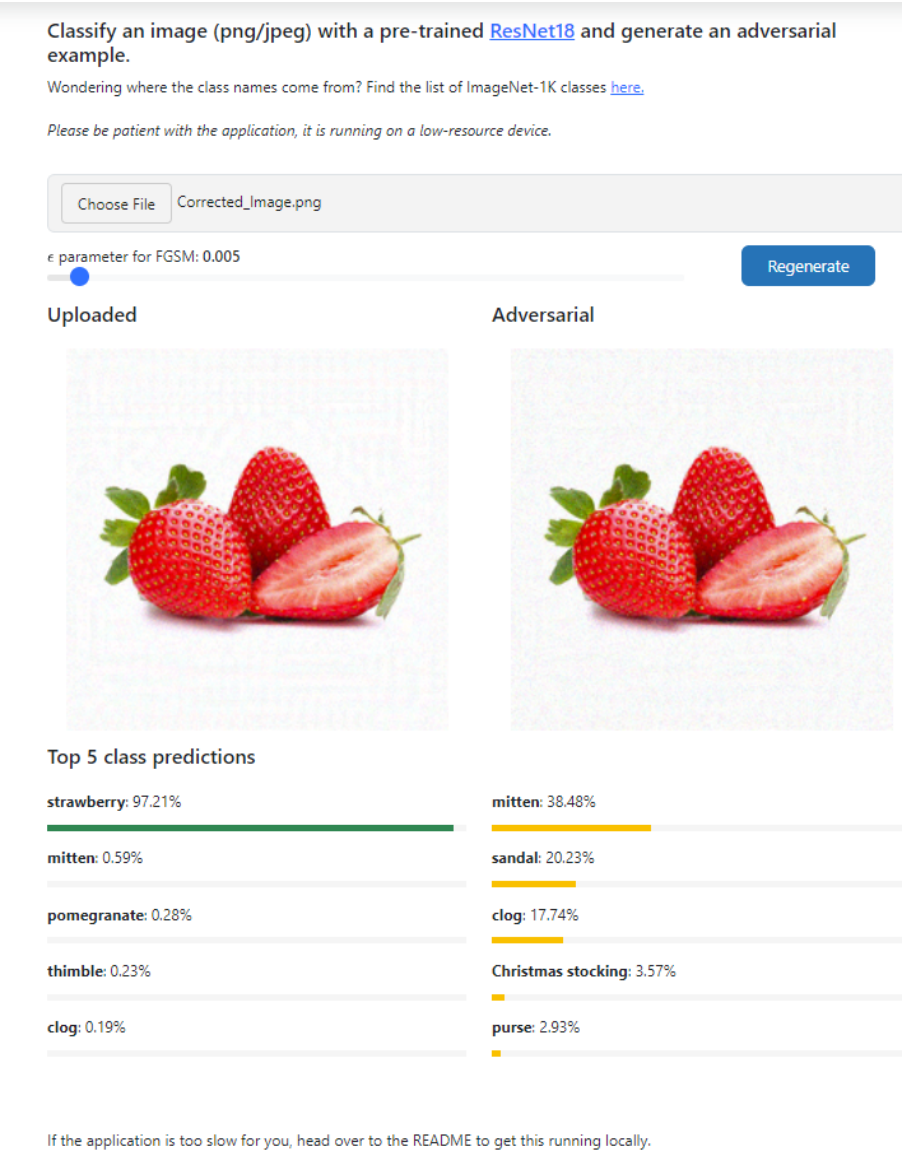


Figure 19: Classification of the Corrected Image

The classifier accurately predicts the corrected image as a strawberry with a confidence of 97.21%, which is very close to the original prediction of the test image as a strawberry, whose confidence was 99.69%. This indicates that the error correction process using the repetition (3,1) code was effective in restoring the image's classification accuracy.

4.2 Hamming Code

To assess the effectiveness of the Hamming code, we tested it using an image of a strawberry. The image was chosen for its clear shape, vibrant colors, and fine details, making it well-suited for evaluating the accuracy of the error correction and decoding process.

The same test image (Figure 11)used earlier of 99.69% confidence (Figure 12), was employed for evaluating the effectiveness of the Hamming code.

The images were separated and encoded using the Hamming(7,4) algorithm as shown in Figure 20

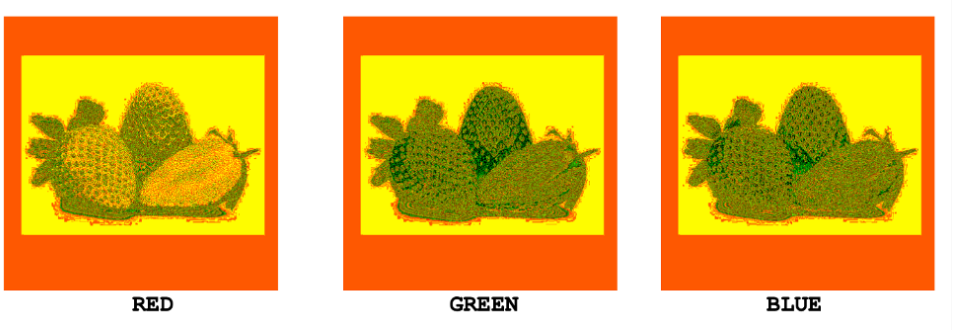


Figure 20: Red, Green, and Blue Channel Images Before Noise

The encoded Red, Green, and Blue images were then passed through the noise simulation, where perturbations were added using the Noisy algorithm we developed. This step simulated real-world interference, such as transmission errors, resulting in noisy versions of the strawberry image. The classifications of the Red, Green, and Blue channel images after adding noise are shown below.

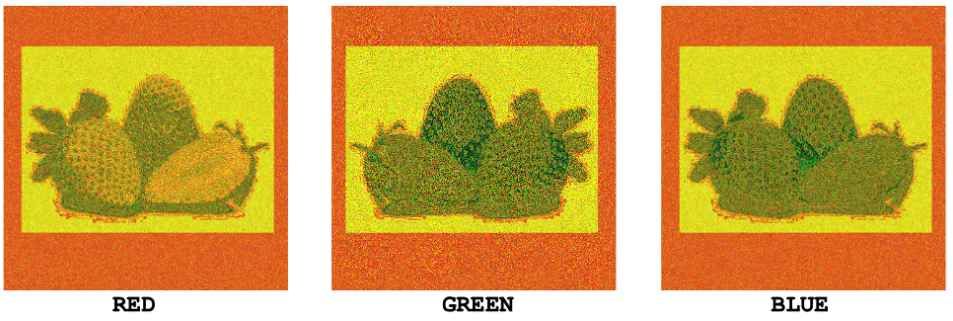


Figure 21: Red, Green, and Blue Channel Images After Noise

The noisy images were decoded using Hamming(7,4) decoder algorithm to correct pixel errors in each channel. After correcting all three channels and combining them, the final corrected image was obtained, as shown below:

Figure 22: Corrected Image Classification

To check the effectiveness of our algorithm in correcting the image, the decoded image was passed to the FGSM classifier. The classifier predicted the corrected image as a strawberry with a confidence of 99.73%, which is slightly higher than the original prediction of the test image, whose confidence was 99.69%. This indicates that the error correction process using the Hamming(7,4) code was effective in restoring the image's classification accuracy. The classification is denoted below.

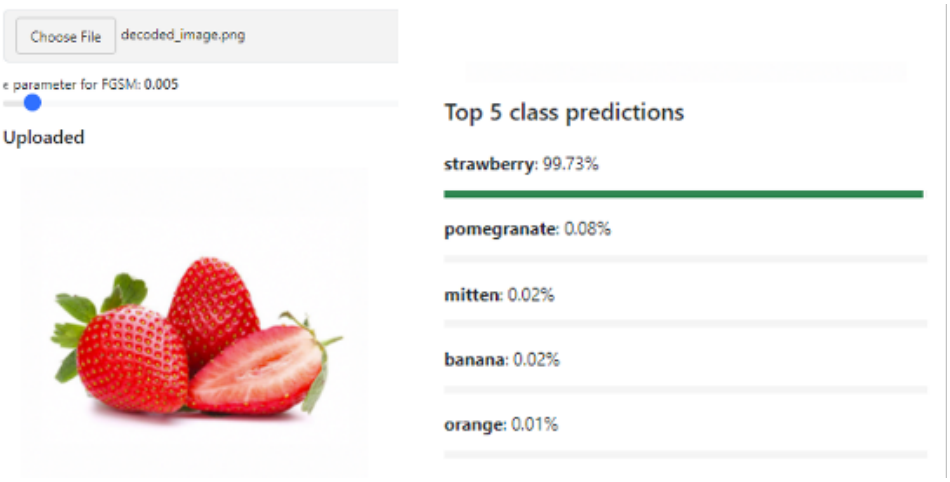


Figure 23: Corrected Image Classification

5 Conclusion

In this project, we utilized **Chebyshev's** and **Chernoff's inequalities** to determine the optimal number of repetitions k required to ensure that the probability of failing to correct a bit-flip error in a repetition code is below a desired threshold. This approach provides a conservative bound and can be adjusted for different values of the bit-flip probability p and the desired failure probability Q.

This project explored the implementation and effectiveness of **Repetition** and **Hamming Codes** as error-correcting algorithms in digital image transmission. Specifically, these codes were applied to mitigate bit-flip errors commonly occurring during satellite image transmission due to electromagnetic interference. The findings reveal the strengths and limitations of each code, as well as potential real-world applications.

Key Findings

1. Repetition Code The Repetition (3,1) Code involves replicating each bit three times, allowing for error detection and correction via majority voting. This simple redundancy strategy is advantageous due to its ease of implementation and suitability for channels with high error rates.

- **Advantages:** The Repetition Code effectively detects and corrects errors when noise is uniformly random and moderate. Increasing the number of repetitions improves correction probability, though with diminishing returns.
- **Limitations:** The main limitation is inefficiency; transmitting each bit multiple times requires substantial bandwidth, making it impractical for high-resolution images. Additionally, it fails to correct errors if more than half of the bits in a repeated set are altered by noise.

2. Hamming Code The Hamming (7,4) Code is a more sophisticated error-correcting method, adding three parity bits to each set of four data bits to detect and correct single-bit errors.

- **Advantages:** Hamming Codes are more bandwidth-efficient than Repetition Codes. Their single-bit correction capability makes them ideal for limited-bandwidth channels, such as those used by NASA's Deep Space Network.
- **Limitations:** This code can only correct single-bit errors, making it less effective in high-noise environments. Additionally, the decoding process requires parity calculations and syndrome analysis, increasing complexity compared to Repetition Codes.

Potential Real-World Applications

The project findings are applicable to several other real-world situations where the accuracy of error-correcting codes can play a critical role. It is especially relevant where reliable data transmission is crucial:

- **Space and Satellite Communications:** Repetition Codes can serve as a fail-safe where high redundancy is tolerable, while Hamming Codes can reduce error in single-bit transmissions over constrained channels.
- **Data Storage and Retrieval Systems:** Hamming Codes are used in RAM to detect and correct bit errors during data storage and retrieval, ensuring data integrity.
- **Wireless Communication Networks:** In applications like mobile networks or Wi-Fi, Hamming Codes improve data transmission by correcting minor interference-related errors, leading to smoother communication.
- **Medical Imaging and Data Transmission:** Both codes can be applied to prevent image corruption during transmission, enhancing the quality and reliability of critical medical data.

Summary

In conclusion, this analysis highlights the distinct advantages of Repetition and Hamming Codes in error correction. While Repetition Codes offer robustness through redundancy, they come at the cost of high bandwidth usage. Conversely, Hamming Codes provide a more efficient solution, particularly effective when the bit-flip probability is below 0.5—a common scenario in satellite communications. Our mathematical analysis demonstrates Hamming Codes' superior ability to maintain transmission integrity using fewer resources compared to Repetition Codes. This project illustrates the efficacy of these codes and their broad applicability in fields that depend on accurate data transmission. Future research could explore hybrid approaches or advanced error-correcting codes that combine the advantages of both methods, optimizing correction in high-noise, bandwidth-constrained environments.

References

[1] H. Fredricksen, "Error Correction for Deep Space Network Teletype Circuits," *NASA*, Jun. 1968. Available: https://ntrs.nasa.gov/api/citations/19680016272/downloads/19680016272.pdf

[2] Q. Huang, S. Lin, and K. Abdel-Ghaffar, "Error-Correcting Codes for Flash Coding," *IEEE Trans. Inf. Theory*, vol. 57, no. 9, pp. 6097–6108, 2011. Available: https://doi.org/10.1109/tit.2011.2162262

[3] D. Binder, E. C. Smith, and A. B. Holman, "Satellite Anomalies from Galactic Cosmic Rays," *IEEE Trans. Nucl. Sci.*, vol. 22, no. 6, pp. 2675–2680, 1975. Available: https://doi.org/10.1109/tns.1975.4328188